

# Hardwarenahe Programmierung I

U. Kaiser, R. Kaiser, M. Stöttinger, S. Reith

(HTTP: <http://www.cs.hs-rm.de/~kaiser>

E-Mail: [robert.kaiser@hs-rm.de](mailto:robert.kaiser@hs-rm.de))

Wintersemester 2021/2022

# 5. C-Sprachelemente



<https://www.artsy.net/artwork/brian-kernighan-hello-world>

# Ausgewählte Sprachelemente von C

## In diesem Kapitel ...

- Minimal notwendiges Rüstzeug zur Erstellung kleiner, Beispielprogramme
- Dazu Vorgriff auf spätere Kapitel:
  - ▶ grundlegende Programmkonstrukte
  - ▶ Funktionen zur Tastatureingabe und Bildschirmausgabe.

# Programmrahmen

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

void main()
{
    ...
    ...
    ...

    ...
    ...
    ...
}
```



Am Anfang stehen die  
**Variablendefinitionen**

- Die beiden ersten, mit # beginnenden Zeilen übernehmen Sie einfach in Ihren Programmcode.
- Das eigentliche Programm besteht aus einem **Hauptprogramm**, das in C mit main bezeichnet werden muss. Den Zusatz void und die hinter main stehenden runden Klammern erklären wir später.
- Die auf main folgenden geschweiften Klammern umschließen den Inhalt des Hauptprogramms, der aus Variablendefinitionen und Programmcode besteht.
- Geschweifte Klammern kommen in C immer vor, wenn etwas zusammengefasst werden soll. Diese Klammern treten immer paarig auf. Sie sollten die Klammern so einrücken, dass man sofort erkennen kann, welche schließende Klammer zu welcher öffnenden Klammer gehört. Das erhöht die Lesbarkeit Ihres Codes.

# Programmrahmen

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

void main()
{
    ...
    ...
    ...
    ...
    ...
}
```

Am Anfang stehen die **Variablendefinitionen**

Dann folgt der **Programmcode**

- Die beiden ersten, mit # beginnenden Zeilen übernehmen Sie einfach in Ihren Programmcode.
- Das eigentliche Programm besteht aus einem **Hauptprogramm**, das in C mit main bezeichnet werden muss. Den Zusatz void und die hinter main stehenden runden Klammern erklären wir später.
- Die auf main folgenden geschweiften Klammern umschließen den Inhalt des Hauptprogramms, der aus Variablendefinitionen und Programmcode besteht.
- Geschweifte Klammern kommen in C immer vor, wenn etwas zusammengefasst werden soll. Diese Klammern treten immer paarig auf. Sie sollten die Klammern so einrücken, dass man sofort erkennen kann, welche schließende Klammer zu welcher öffnenden Klammer gehört. Das erhöht die Lesbarkeit Ihres Codes.

# Zahlenkonstanten

- Man unterscheidet zwischen ganzen Zahlen, z.B.:

1234

-4711

- und Gleitkommazahlen, z.B.:

1.234

-47.11

- Wichtig ist, dass bei Gleitkommazahlen ein **Dezimalpunkt** verwendet wird.

# Variablendefinitionen

- Unter einer **Variablen** verstehen wir einen mit einem Namen versehenen Speicherbereich, in dem Daten eines bestimmten Typs hinterlegt werden können.
- Das im Speicherbereich der Variablen hinterlegte Datum bezeichnen wir als den **Wert** der Variablen.

- Zu einer Variablen gehören

also:

- ▶ ein Name
- ▶ ein Typ
- ▶ ein Speicherbereich
- ▶ ein Wert

Am Anfang steht der  
Typ der Variablen

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

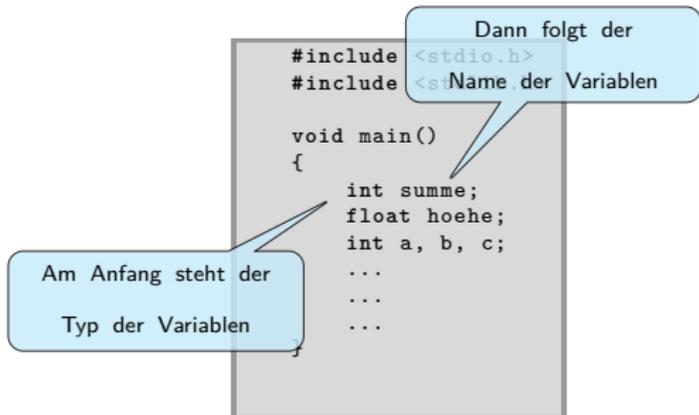
void main()
{
    int summe;
    float hoehe;
    int a, b, c;
    ...
    ...
}
```

- Als Typen betrachten wir zunächst nur `int` für ganze Zahlen und `float` für Gleitkommazahlen.
- Name, Typ und Wert einer Variablen legt der Programmierer fest. Der Speicherbereich interessiert uns hier noch nicht, da er vom Compiler vergeben wird.

# Variablendefinitionen

- Unter einer **Variablen** verstehen wir einen mit einem Namen versehenen Speicherbereich, in dem Daten eines bestimmten Typs hinterlegt werden können.
- Das im Speicherbereich der Variablen hinterlegte Datum bezeichnen wir als den **Wert** der Variablen.

- Zu einer Variablen gehören also:
  - ▶ ein Name
  - ▶ ein Typ
  - ▶ ein Speicherbereich
  - ▶ ein Wert

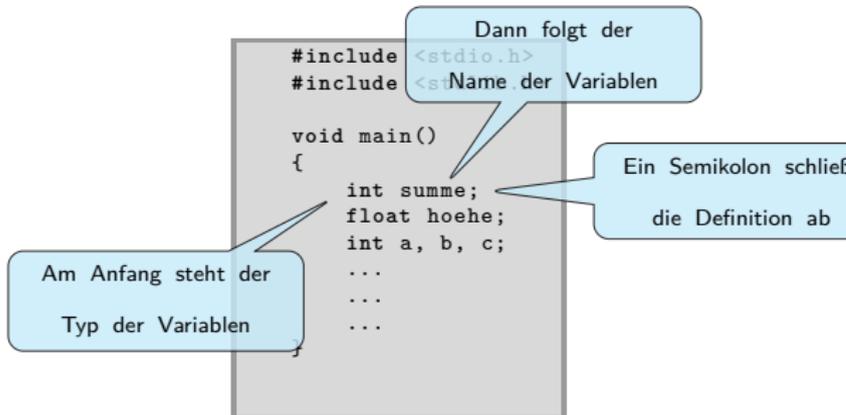


- Als Typen betrachten wir zunächst nur `int` für ganze Zahlen und `float` für Gleitkommazahlen.
- Name, Typ und Wert einer Variablen legt der Programmierer fest. Der Speicherbereich interessiert uns hier noch nicht, da er vom Compiler vergeben wird.

# Variablendefinitionen

- Unter einer **Variablen** verstehen wir einen mit einem Namen versehenen Speicherbereich, in dem Daten eines bestimmten Typs hinterlegt werden können.
- Das im Speicherbereich der Variablen hinterlegte Datum bezeichnen wir als den **Wert** der Variablen.

- Zu einer Variablen gehören also:
  - ▶ ein Name
  - ▶ ein Typ
  - ▶ ein Speicherbereich
  - ▶ ein Wert



```

#include <stdio.h>
#include <stName der Variablen

void main()
{
    int summe;
    float hoehe;
    int a, b, c;
    ...
    ...
    ...
}

```

Dann folgt der

Ein Semikolon schließt die Definition ab

Am Anfang steht der Typ der Variablen

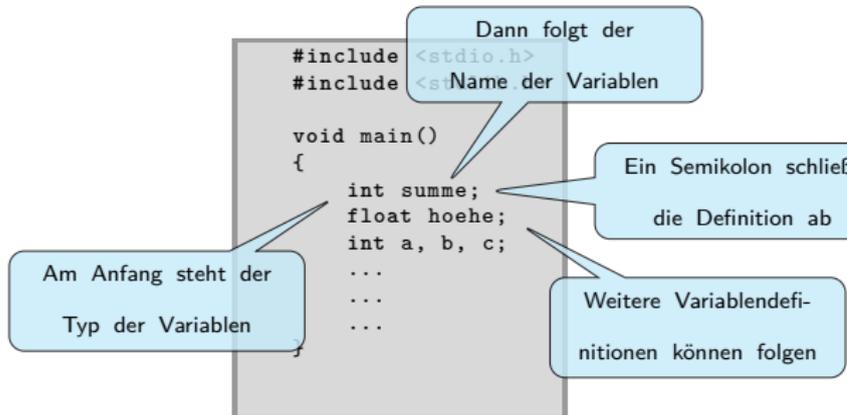
- Als Typen betrachten wir zunächst nur `int` für ganze Zahlen und `float` für Gleitkommazahlen.
- Name, Typ und Wert einer Variablen legt der Programmierer fest. Der Speicherbereich interessiert uns hier noch nicht, da er vom Compiler vergeben wird.

# Variablendefinitionen

- Unter einer **Variablen** verstehen wir einen mit einem Namen versehenen Speicherbereich, in dem Daten eines bestimmten Typs hinterlegt werden können.
- Das im Speicherbereich der Variablen hinterlegte Datum bezeichnen wir als den **Wert** der Variablen.

- Zu einer Variablen gehören also:

- ▶ ein Name
- ▶ ein Typ
- ▶ ein Speicherbereich
- ▶ ein Wert



```

#include <stdio.h>
#include <stName der Variablen

void main()
{
    int summe;
    float hoehe;
    int a, b, c;
    ...
    ...
    ...
}

```

Dann folgt der

Ein Semikolon schließt die Definition ab

Weitere Variablendefinitionen können folgen

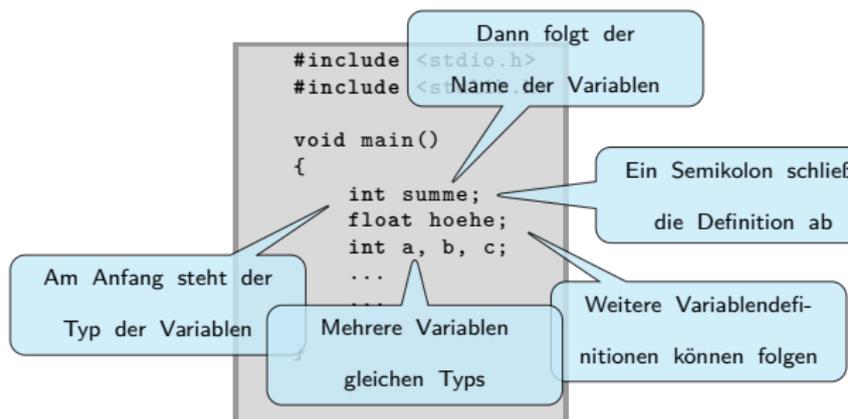
Am Anfang steht der Typ der Variablen

- Als Typen betrachten wir zunächst nur `int` für ganze Zahlen und `float` für Gleitkommazahlen.
- Name, Typ und Wert einer Variablen legt der Programmierer fest. Der Speicherbereich interessiert uns hier noch nicht, da er vom Compiler vergeben wird.

# Variablendefinitionen

- Unter einer **Variablen** verstehen wir einen mit einem Namen versehenen Speicherbereich, in dem Daten eines bestimmten Typs hinterlegt werden können.
- Das im Speicherbereich der Variablen hinterlegte Datum bezeichnen wir als den **Wert** der Variablen.

- Zu einer Variablen gehören also:
  - ▶ ein Name
  - ▶ ein Typ
  - ▶ ein Speicherbereich
  - ▶ ein Wert



- Als Typen betrachten wir zunächst nur `int` für ganze Zahlen und `float` für Gleitkommazahlen.
- Name, Typ und Wert einer Variablen legt der Programmierer fest. Der Speicherbereich interessiert uns hier noch nicht, da er vom Compiler vergeben wird.

# Wertzuweisungen

- Den Variablen können direkt bei ihrer Definition oder später im Programm Werte zugewiesen werden.

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

void main()
{
    int summe = 1;
    float hoehe = 3.7;
    int a, b = 0, c;

    a = 1;
    hoehe = a;
    a = 2;
}
```

Variablen können  
bei der Definition

initialisiert werden

- Die Werte können jederzeit durch erneute Zuweisung geändert werden.
- Der zugewiesene Wert muss zum Typ der Variablen passen. Zum Beispiel sollten Sie einer `int`-Variablen keinen `float`-Wert zuweisen. Einer `float`-Variablen kann aber durchaus ein `int`-Wert zugewiesen werden.

# Wertzuweisungen

- Den Variablen können direkt bei ihrer Definition oder später im Programm **Werte** zugewiesen werden.

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

void main()
{
    int summe = 1;
    float hoehe = 3.7;
    int a, b = 0, c;

    a = 1;
    hoehe = a;
    a = 2;
}
```

Variablen können

bei der Definition

initialisierte Variablen haben

„Zufallswerte“

- Die Werte können jederzeit durch erneute Zuweisung geändert werden.
- Der zugewiesene Wert muss zum Typ der Variablen passen. Zum Beispiel sollten Sie einer `int`-Variablen keinen `float`-Wert zuweisen. Einer `float`-Variablen kann aber durchaus ein `int`-Wert zugewiesen werden.

# Wertzuweisungen

- Den Variablen können direkt bei ihrer Definition oder später im Programm **Werte** zugewiesen werden.

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

void main()
{
    int summe = 1;
    float hoehe = 3.7;
    int a, b = 0, c;

    a = 1;
    hoehe = a;
    a = 2;
}
```

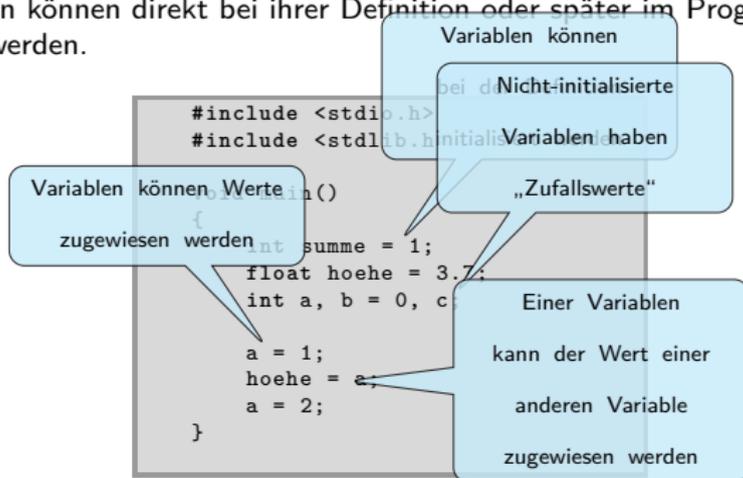
Variablen können  
bei der Definition  
Nicht-initialisierte  
Variablen haben  
„Zufallswerte“

Einer Variablen  
kann der Wert einer  
anderen Variable  
zugewiesen werden

- Die Werte können jederzeit durch erneute Zuweisung geändert werden.
- Der zugewiesene Wert muss zum Typ der Variablen passen. Zum Beispiel sollten Sie einer `int`-Variablen keinen `float`-Wert zuweisen. Einer `float`-Variablen kann aber durchaus ein `int`-Wert zugewiesen werden.

# Wertzuweisungen

- Den Variablen können direkt bei ihrer Definition oder später im Programm **Werte** zugewiesen werden.



The diagram illustrates variable assignment in C with the following code and callouts:

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int summe = 1;
    float hoehe = 3.7;
    int a, b = 0, c;

    a = 1;
    hoehe = a;
    a = 2;
}

```

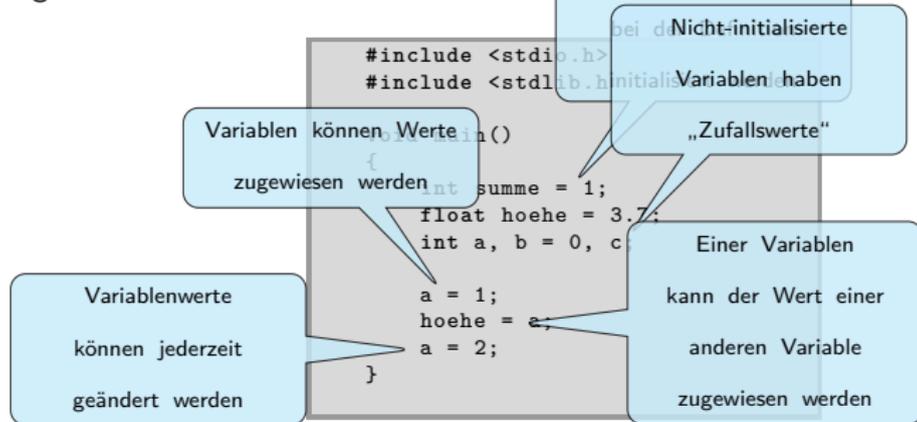
Callouts and their corresponding code elements:

- Variablen können bei der Definition zugewiesen werden** (points to `int summe = 1;`)
- Variablen können „Zufallswerte“ zugewiesen werden** (points to `int a, b = 0, c;`)
- Variablen können bei der Definition nicht-initialisierte Variablen haben** (points to `int a, b = 0, c;`)
- Einer Variablen kann der Wert einer anderen Variable zugewiesen werden** (points to `hoehe = a;`)

- Die Werte können jederzeit durch erneute Zuweisung geändert werden.
- Der zugewiesene Wert muss zum Typ der Variablen passen. Zum Beispiel sollten Sie einer `int`-Variablen keinen `float`-Wert zuweisen. Einer `float`-Variablen kann aber durchaus ein `int`-Wert zugewiesen werden.

# Wertzuweisungen

- Den Variablen können direkt bei ihrer Definition oder später im Programm **Werte** zugewiesen werden.



```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int summe = 1;
    float hoehe = 3.7;
    int a, b = 0, c;

    a = 1;
    hoehe = a;
    a = 2;
}

```

Callouts explaining the code:

- Variablen können bei der Definition initialisiert werden
- Nicht-initialisierte Variablen haben „Zufallswerte“
- Variablen können Werte zugewiesen werden
- Variablenwerte können jederzeit geändert werden
- Einer Variablen kann der Wert einer anderen Variable zugewiesen werden

- Die Werte können jederzeit durch erneute Zuweisung geändert werden.
- Der zugewiesene Wert muss zum Typ der Variablen passen. Zum Beispiel sollten Sie einer `int`-Variablen keinen `float`-Wert zuweisen. Einer `float`-Variablen kann aber durchaus ein `int`-Wert zugewiesen werden.

# Arithmetische Operationen

- Variablen und Zahlkonstanten können mit arithmetischen Operatoren verknüpft werden:

Operator	Verwendung	Bedeutung
+	$x + y$	Addition von $x$ und $y$
-	$x - y$	Subtraktion von $y$ von $x$
*	$x * y$	Multiplikation von $x$ und $y$
/	$x / y$	Division von $x$ durch $y$ ( $y \neq 0$ )
%	$x \% y$	Rest bei ganzzahliger Division von $x$ durch $y$ (Modulo-Operator, $y \neq 0$ )

- Es handelt sich um die üblichen Rechenoperationen

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

void main()
{
    int summe = 1;
    float hoehe;
    int a, b, c = 0;

    hoehe = 1.2 + 2*c;
    a = b + c;
    summe = summe + 1;
}
```

Variablenwerte können durch Formeln berechnet werden

# Arithmetische Operationen

- Variablen und Zahlkonstanten können mit arithmetischen Operatoren verknüpft werden:

Operator	Verwendung	Bedeutung
+	$x + y$	Addition von $x$ und $y$
-	$x - y$	Subtraktion von $y$ von $x$
*	$x * y$	Multiplikation von $x$ und $y$
/	$x / y$	Division von $x$ durch $y$ ( $y \neq 0$ )
%	$x \% y$	Rest bei ganzzahliger Division von $x$ durch $y$ (Modulo-Operator, $y \neq 0$ )

- Es handelt sich um die üblichen Rechenoperationen

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

void main()
{
    int summe = 1;
    float hoehe;
    int a, b, c = 0;

    hoehe = 1.2 + 2*c;
    a = b + c;
    summe = summe + 1;
}
```

Variablenwerte können  
 in Formeln können  
 verwendet werden  
 Variablen vorkommen

# Arithmetische Operationen

- Variablen und Zahlkonstanten können mit arithmetischen Operatoren verknüpft werden:

Operator	Verwendung	Bedeutung
+	$x + y$	Addition von $x$ und $y$
-	$x - y$	Subtraktion von $y$ von $x$
*	$x * y$	Multiplikation von $x$ und $y$
/	$x / y$	Division von $x$ durch $y$ ( $y \neq 0$ )
%	$x \% y$	Rest bei ganzzahliger Division von $x$ durch $y$ (Modulo-Operator, $y \neq 0$ )

- Es handelt sich um die üblichen Rechenoperationen

```

#include <stdio.h>
#include <stdlib.h>

void main()
{
    int summe = 1;
    float hoehe;
    int a, b, c = 0;

    hoehe = 1.2 + 2*c;
    a = b + c;
    summe = summe + 1;
}

```

Variablenerte können  
 in Formeln können  
 Variablen vorkommen  
 Vorsicht bei Verwendung  
 uninitialisierter Variablen:  
 Ergebnis ist undefiniert

# Arithmetische Operationen

- Variablen und Zahlkonstanten können mit arithmetischen Operatoren verknüpft werden:

Operator	Verwendung	Bedeutung
+	$x + y$	Addition von $x$ und $y$
-	$x - y$	Subtraktion von $y$ von $x$
*	$x * y$	Multiplikation von $x$ und $y$
/	$x / y$	Division von $x$ durch $y$ ( $y \neq 0$ )
%	$x \% y$	Rest bei ganzzahliger Division von $x$ durch $y$ (Modulo-Operator, $y \neq 0$ )

- Es handelt sich um die üblichen Rechenoperationen

```

#include <stdio.h>
#include <stdlib.h>

void main()
{
    int summe = 1;
    float hoehe;
    int a, b, c = 0;

    hoehe = 1.2 + 2*c;
    a = b + c;
    summe = summe + 1;
}

```

Die gleiche Variable kann auf beiden Seiten einer Zuweisung vorkommen

Variablenerte können in Formeln vorkommen

Vorsicht bei Verwendung uninitialisierter Variablen: Ergebnis ist undefiniert

# Formel­aus­drücke

- Mit Variablen, Zahlkonstanten, Operatoren und Klammern können Formel­aus­drücke gebildet werden.

```
int a;  
float b, c;  
  
a = 1;  
b = (a+1)*(a+2);  
c = (3.14*a - 2.7)/5;
```

- Es gelten die üblichen Rechenregeln (z.B. Punkt­rechnung geht vor Strich­rechnung).
- Ganze Zahlen und Gleitkommazahlen können in Formeln durchaus gemischt vorkommen. Es wird immer so lange wie möglich im Bereich der ganzen Zahlen gerechnet. Sobald aber die erste Gleitkommazahl ins Spiel kommt, wird die weitere Berechnung im Bereich der Gleitkommazahlen durchgeführt.

# Operatoren mit gleichzeitiger Wertzuweisung



- Die Variable auf der linken Seite einer Zuweisung kann auch auf der rechten Seite derselben Zuweisung vorkommen. Zunächst wird der rechts vom Zuweisungsoperator stehende Ausdruck vollständig ausgewertet, dann wird das Ergebnis der Variablen links vom Gleichheitszeichen zugewiesen. Die Anweisung  $a = a + 1;$  erhöht den Wert der Variablen  $a$  um 1<sup>1</sup>. Anweisungen wie  $a = a+5$  oder  $b = b-a$  werden in Programmen recht häufig verwendet. Man kann dann vereinfachend  $a += 5$  oder  $b -= a$  schreiben.

Insgesamt gibt es folgende Vereinfachungsmöglichkeiten:

Operator	Verwendung	Entsprechung
+=	$x += y$	$x = x + y$
-=	$x -= y$	$x = x - y$
*=	$x *= y$	$x = x * y$
/=	$x /= y$	$x = x / y$
%=	$x %= y$	$x = x \% y$

<sup>1</sup>Man sagt auch:  $a$  wird „inkrementiert“

# Inkrement und Dekrement von Variablen

- Im Fall einer Addition oder Subtraktion von 1 kann man noch einfacher formulieren:

Operator	Verwendung	Entsprechung
++	x++ bzw. ++x	x = x + 1
--	x-- bzw. --x	x = x - 1

- Diese Inkrement- und Dekrement-Operatoren gibt es in Präfix- und Postfixnotation. Das heißt, diese Operatoren können ihrem Operanden voran- oder nachgestellt werden. Im ersten Fall wird der Operator angewandt bevor der Operand in einen Ausdruck eingeht, im zweiten Fall erst danach. Dieser kleine Unterschied kann bedeutsame Auswirkungen haben:

```
int i, k;

i = 0;
k = i++;

i = 0;
k = ++i;
```

Der Wert von i wird erst nach der Zuweisung an k erhöht. Also: k = 0.

# Inkrement und Dekrement von Variablen

- Im Fall einer Addition oder Subtraktion von 1 kann man noch einfacher formulieren:

Operator	Verwendung	Entsprechung
++	x++ bzw. ++x	x = x + 1
--	x-- bzw. --x	x = x - 1

- Diese Inkrement- und Dekrement-Operatoren gibt es in Präfix- und Postfixnotation. Das heißt, diese Operatoren können ihrem Operanden voran- oder nachgestellt werden. Im ersten Fall wird der Operator angewandt bevor der Operand in einen Ausdruck eingeht, im zweiten Fall erst danach. Dieser kleine Unterschied kann bedeutsame Auswirkungen haben:

```
int i, k;

i = 0;
k = i++;

i = 0;
k = ++i;
```

Der Wert von i wird erst nach der Zuweisung an k erhöht. Also: k = 0.

Der Wert von i wird vor der Zuweisung an k erhöht. Also: k = 0.

# Operationen mit ganzzahligen Operanden



- Das Ergebnis einer arithmetischen Operation, an der nur ganzzahlige Operanden beteiligt sind, ist immer eine ganze Zahl.
- Im Falle einer Division wird eine **Division ohne Rest** (Integer-Division) durchgeführt, wenn beide Operanden ganzzahlig sind.

```
a = (100*10)/100;  
b = 100*(10/100);
```

- Rein mathematisch müsste eigentlich in beiden Fällen 10 als Ergebnis herauskommen. Im Programm wird aber wie folgt gerechnet:

```
a = (100*10)/100 = 1000/100 = 10  
b = 100*(10/100) = 100 * 0 = 0
```

- Es ergibt sich also  $a = 10$  und  $b = 0$ .
- Wenn man sich bei einer Integer-Division für den unter den Tisch fallenden Rest interessiert, kann man diesen mit dem Modulo-Operator (%) ermitteln. Der Ausdruck

```
a = 20 % 7
```

- berechnet den Rest, der bei einer Division von 20 durch 7 bleibt, und weist diesen der Variablen a zu. Die Variable a hat also anschließend den Wert 6.
- Die Integer-Division ist kein Design- oder Rechenfehler. Wir werden noch viele sinnvolle Verwendungen der Integer-Division und des Modulo-Operators kennen lernen.

# Typkonvertierung

- Manchmal möchte man, obwohl man es nur mit Integer-Werten zu tun hat, eine „richtige“ Division durchführen und das Ergebnis einer Gleitkommazahl zuweisen. Die bloße Zuweisung an eine Gleitkommazahl konvertiert das Ergebnis zwar automatisch in eine Gleitkommazahl, aber erst nachdem die Division durchgeführt wurde:

```
void main()
{
    int a = 1, b = 2;
    float x;

    x = a/b;
}
```

- Um das Problem zu lösen, ändert man für die Berechnung (und nur für die Berechnung) den Datentyp von a in float, indem man der Variablen den gewünschten Datentyp in Klammern voranstellt:

```
void main()
{
    int a = 1, b = 2;
    float x;

    x = ((float)a)/b;
}
```

- Bei den vorangestellten Klammern handelt es sich übrigens auch um einen Operator – den sogenannten Cast-Operator.

# Typkonvertierung

- Manchmal möchte man, obwohl man es nur mit Integer-Werten zu tun hat, eine „richtige“ Division durchführen und das Ergebnis einer Gleitkommazahl zuweisen. Die bloße Zuweisung an eine Gleitkommazahl konvertiert das Ergebnis zwar automatisch in eine Gleitkommazahl, aber erst nachdem die Division durchgeführt wurde:

```
void main()
{
    int a = 1, b = 2;
    float x;

    x = a/b;
}
```

Das Ergebnis der  
Division ist 0

- Um das Problem zu lösen, ändert man für die Berechnung (und nur für die Berechnung) den Datentyp von a in float, indem man der Variablen den gewünschten Datentyp in Klammern voranstellt:

```
void main()
{
    int a = 1, b = 2;
    float x;

    x = ((float)a)/b;
}
```

- Bei den vorangestellten Klammern handelt es sich übrigens auch um einen Operator – den sogenannten Cast-Operator.

# Typkonvertierung

- Manchmal möchte man, obwohl man es nur mit Integer-Werten zu tun hat, eine „richtige“ Division durchführen und das Ergebnis einer Gleitkommazahl zuweisen. Die bloße Zuweisung an eine Gleitkommazahl konvertiert das Ergebnis zwar automatisch in eine Gleitkommazahl, aber erst nachdem die Division durchgeführt wurde:

```
void main()
{
    int a = 1, b = 2;
    float x;

    x = a/b;
}
```

Das Ergebnis der Division ist 0

- Um das Problem zu lösen, ändert man für die Berechnung (und nur für die Berechnung) den Datentyp von a in float, indem man der Variablen den gewünschten Datentyp in Klammern voranstellt:

```
void main()
{
    int a = 1, b = 2;
    float x;

    x = ((float)a)/b;
}
```

a wird vor der Division in float konvertiert.  
Das Ergebnis der Division ist dann 0.5

- Bei den vorangestellten Klammern handelt es sich übrigens auch um einen Operator – den sogenannten Cast-Operator.

# Vergleichsoperatoren



- Zahlen und Variablen können untereinander verglichen werden. Die folgende Tabelle zeigt die in C verwendeten Vergleichsoperatoren:

Operator	Verwendung	Bedeutung
<	$x < y$	kleiner
<=	$x \leq y$	kleiner oder gleich
>	$x > y$	größer
>=	$x \geq y$	größer oder gleich
==	$x == y$	gleich
!=	$x != y$	ungleich

- Auf der linken bzw. rechten Seite eines Vergleichsausdrucks können beliebige Ausdrücke (üblicherweise arithmetische Ausdrücke) mit Variablen oder Zahlen stehen:

$a < 7$

$a \leq 2*(b+1)$

$a+1 == a*a$

- Das Ergebnis eines Vergleichs ist ein logischer Wert („wahr“ oder „falsch“), der in C durch 1 (wahr) oder 0 (falsch) dargestellt wird. Mit diesem Wert können wir dann, wie mit einem durch einen arithmetischen Ausdruck gewonnenen Wert, weiterarbeiten. C unterscheidet nicht zwischen arithmetischen und logischen Werten.

# Fallunterscheidungen

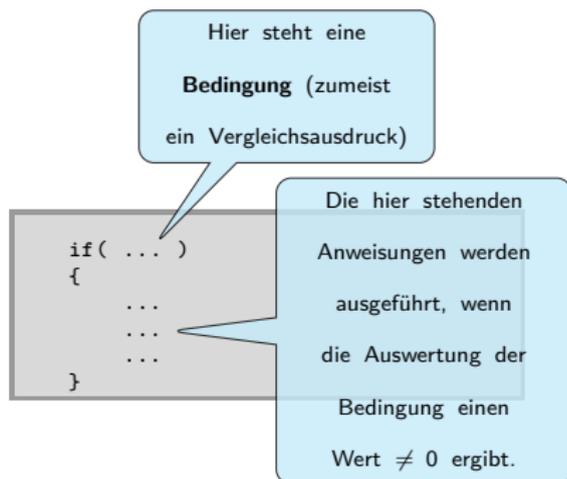
- Fallunterscheidungen kann man in C durch eine sogenannte `if`-Anweisung realisieren:

Hier steht eine  
**Bedingung** (zumeist  
ein Vergleichsausdruck)

```
if( ... )  
{  
    ...  
    ...  
    ...  
}
```

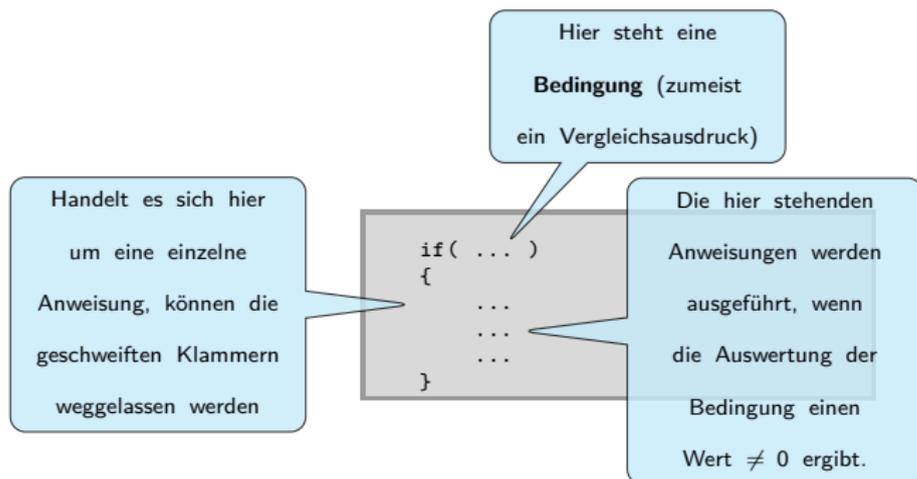
# Fallunterscheidungen

- Fallunterscheidungen kann man in C durch eine sogenannte `if`-Anweisung realisieren:



# Fallunterscheidungen

- Fallunterscheidungen kann man in C durch eine sogenannte `if`-Anweisung realisieren:



# Beispiele für Fallunterscheidungen

- Berechne den Absolutbetrag einer Variablen a:

```
if(a < 0)
    a = -a;
```

- Wenn der Wert von a kleiner als der Wert von b ist, dann tausche die Werte von a und b:

```
if(a < b)
{
    c = a;
    a = b;
    b = c;
}
```

- Weise der Variablen max den größeren der Werte von a und b zu:

```
max = a;
if( a < b )
    max = b;
```

# Vollständige Fallunterscheidung

- Um eine vollständige Alternative zu programmieren arbeitet man mit `if...else`:

Hier steht eine  
**Bedingung** (zumeist  
ein Vergleichsausdruck)

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

# Vollständige Fallunterscheidung

- Um eine vollständige Alternative zu programmieren arbeitet man mit `if...else`:

Hier steht eine  
**Bedingung** (zumeist  
ein Vergleichsausdruck)

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

Die hier stehenden  
Anweisungen werden  
ausgeführt, wenn die  
Bedingung erfüllt ist.

# Vollständige Fallunterscheidung

- Um eine vollständige Alternative zu programmieren arbeitet man mit `if...else`:

Hier steht eine  
**Bedingung** (zumeist  
ein Vergleichsausdruck)

Handelt es sich hier  
um eine einzelne  
Anweisung, können die  
geschweiften Klammern  
weggelassen werden

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

Die hier stehenden  
Anweisungen werden  
ausgeführt, wenn die  
Bedingung erfüllt ist.

# Vollständige Fallunterscheidung

- Um eine vollständige Alternative zu programmieren arbeitet man mit `if...else`:

Hier steht eine  
**Bedingung** (zumeist  
ein Vergleichsausdruck)

Handelt es sich hier  
um eine einzelne  
Anweisung, können die  
geschweiften Klammern  
weggelassen werden

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

Die hier stehenden  
Anweisungen werden  
ausgeführt, wenn die  
Bedingung erfüllt ist.

Die hier stehenden  
Anweisungen werden  
ausgeführt, wenn  
die Bedingung  
nicht erfüllt ist.

# Vollständige Fallunterscheidung

- Um eine vollständige Alternative zu programmieren arbeitet man mit `if...else`:

Hier steht eine  
**Bedingung** (zumeist  
ein Vergleichsausdruck)

Handelt es sich hier  
um eine einzelne  
Anweisung, können die  
geschweiften Klammern  
weggelassen werden

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

Die hier stehenden  
Anweisungen werden  
ausgeführt, wenn die  
Bedingung erfüllt ist.

ausgeführt, wenn  
die Bedingung  
nicht erfüllt ist.

Handelt es sich hier  
um eine einzelne  
Anweisung, können die  
geschweiften Klammern  
weggelassen werden

# Vollständige Fallunterscheidung

- Um eine vollständige Alternative zu programmieren arbeitet man mit `if...else`:

Hier steht eine  
**Bedingung** (zumeist  
ein Vergleichsausdruck)

Handelt es sich hier  
um eine einzelne  
Anweisung, können die  
geschweiften Klammern  
weggelassen werden

Dieser Teil kann  
vollständig fehlen.  
Handelt es sich hier

um eine einzelne  
Anweisung, können die  
geschweiften Klammern  
weggelassen werden

```

if( ... )
{
    ...
    ...
    ...
}
else
{
    ...
    ...
    ...
}

```

Die hier stehenden  
Anweisungen werden  
ausgeführt, wenn die  
Bedingung erfüllt ist.

Die hier stehenden  
Anweisungen werden  
ausgeführt, wenn  
die Bedingung  
nicht erfüllt ist.

# Beispiele für vollständige Fallunterscheidungen

- Berechne das Maximum der Zahlen a und b:

```
if(a < b)
    max = a;
else
    max = b;
```

- Berechne den Abstand von a und b:

```
if(a < b)
    abst = b - a;
else
    abst = a - b;
```

# Der Vergleich auf Gleichheit

- Ein Vergleich auf Gleichheit wird mit dem doppelten Gleichheitszeichen durchgeführt:

```
a = 0;
...
if(a == 1)
    b = 5;
```

- Das einfache Gleichheitszeichen bedeutet eine Zuweisung:

```
a = 0;
...
if(a = 1)
    b = 5;
```

- Im zweiten Beispiel wird zunächst der Variablen a der Wert 1 zugewiesen. Das Ergebnis dieser Zuweisung ist 1 (wahr), sodass die nachfolgende Zuweisung (b = 5) immer ausgeführt wird.
- Die Verwechslung von = und == ist einer der am häufigsten vorkommenden Fehler von Programmieranfängern.
- (Dicht gefolgt vom überflüssigen Semikolon:)

```
a = 0;
...
if(a == 1);
    b = 5;
```

# Schleifen

- Mit einer `for`-Schleife wird eine Reihe von Anweisungen wiederholt ausgeführt.

```
for( ...; ...; ... )  
{  
    ...  
    ...  
    ...  
    ...  
    ...  
    ...  
}
```

# Schleifen

- Mit einer `for`-Schleife wird eine Reihe von Anweisungen wiederholt ausgeführt.

Die **Initialisierung** wird vor dem ersten Eintritt in die Schleife einmal ausgeführt.

```
for( ...; ...; ... )  
{  
    ...  
    ...  
    ...  
    ...  
    ...  
    ...  
}
```

# Schleifen

- Mit einer `for`-Schleife wird eine Reihe von Anweisungen wiederholt ausgeführt.

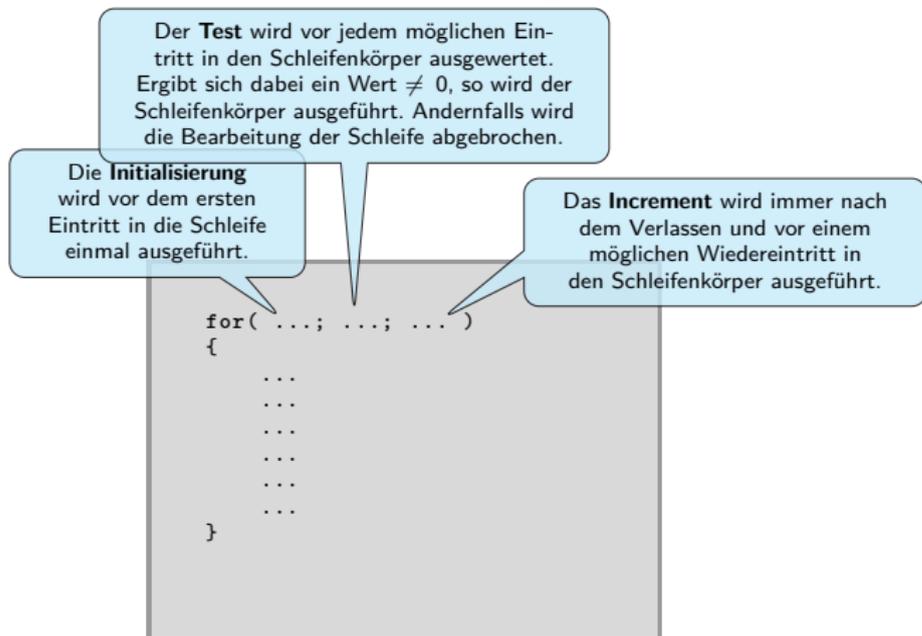
Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Die **Initialisierung** wird vor dem ersten Eintritt in die Schleife einmal ausgeführt.

```
for( ...; ...; ... )  
{  
    ...  
    ...  
    ...  
    ...  
    ...  
    ...  
}
```

# Schleifen

- Mit einer `for`-Schleife wird eine Reihe von Anweisungen wiederholt ausgeführt.



Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

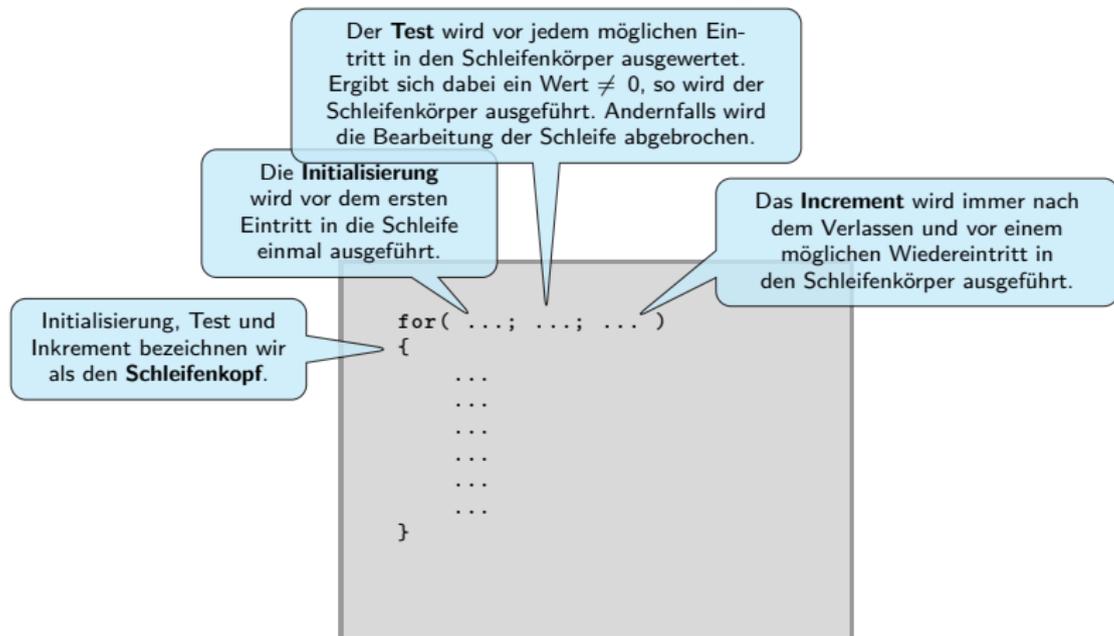
Die **Initialisierung** wird vor dem ersten Eintritt in die Schleife einmal ausgeführt.

Das **Increment** wird immer nach dem Verlassen und vor einem möglichen Wiedereintritt in den Schleifenkörper ausgeführt.

```
for( ...; ...; ... )  
{  
    ...  
    ...  
    ...  
    ...  
    ...  
    ...  
}
```

# Schleifen

- Mit einer `for`-Schleife wird eine Reihe von Anweisungen wiederholt ausgeführt.



Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Die **Initialisierung** wird vor dem ersten Eintritt in die Schleife einmal ausgeführt.

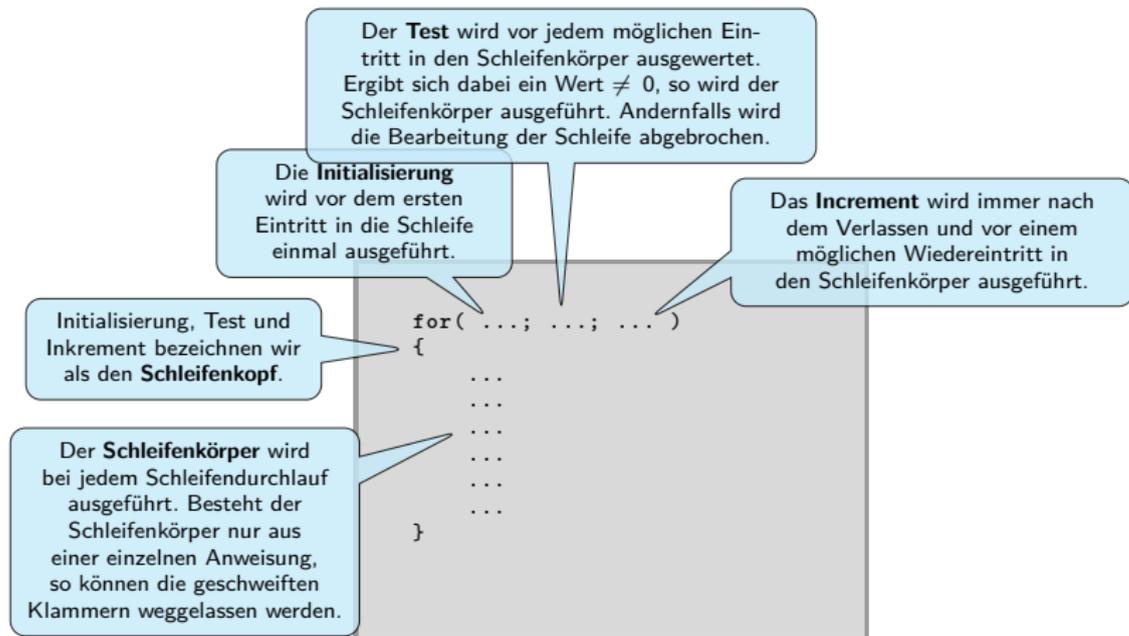
Das **Increment** wird immer nach dem Verlassen und vor einem möglichen Wiedereintritt in den Schleifenkörper ausgeführt.

Initialisierung, Test und Inkrement bezeichnen wir als den **Schleifenkopf**.

```
for( ...; ...; ... )
{
    ...
    ...
    ...
    ...
    ...
}
```

# Schleifen

- Mit einer `for`-Schleife wird eine Reihe von Anweisungen wiederholt ausgeführt.



The diagram illustrates the execution of a `for` loop. It shows a code snippet with callouts explaining its parts:

```
for( ...; ...; ... )  
{  
    ...  
    ...  
    ...  
    ...  
    ...  
    ...  
}
```

**Initialisierung, Test und Inkrement** bezeichnen wir als den **Schleifenkopf**.

Der **Schleifenkörper** wird bei jedem Schleifendurchlauf ausgeführt. Besteht der Schleifenkörper nur aus einer einzelnen Anweisung, so können die geschweiften Klammern weggelassen werden.

Die **Initialisierung** wird vor dem ersten Eintritt in die Schleife einmal ausgeführt.

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Das **Inkrement** wird immer nach dem Verlassen und vor einem möglichen Wiedereintritt in den Schleifenkörper ausgeführt.

# Beispiele für Schleifen

- Summation aller Zahlen von 1 bis 100:

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

- Das gleiche rückwärts:

```
summe = 0;
for( i = 100; i > 0; i = i - 1)
    summe = summe + i;
```

- Man kann mehrere Anweisungen durch Komma getrennt in die Initialisierung oder das Inkrement der Schleife aufnehmen.

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

# Schleifenkontrolle aus dem Schleifenkörper

- Schleifen können aus dem Schleifenkörper mit `break` und `continue` gesteuert werden:

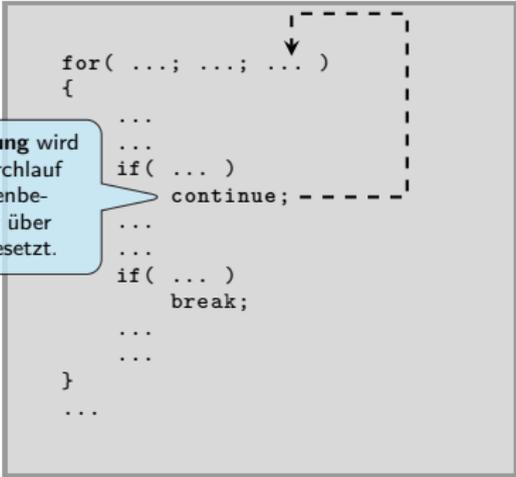
```
for( ...; ...; ... )
{
    ...
    ...
    if( ... )
        continue;
    ...
    ...
    if( ... )
        break;
    ...
    ...
}
```

# Schleifenkontrolle aus dem Schleifenkörper

- Schleifen können aus dem Schleifenkörper mit `break` und `continue` gesteuert werden:

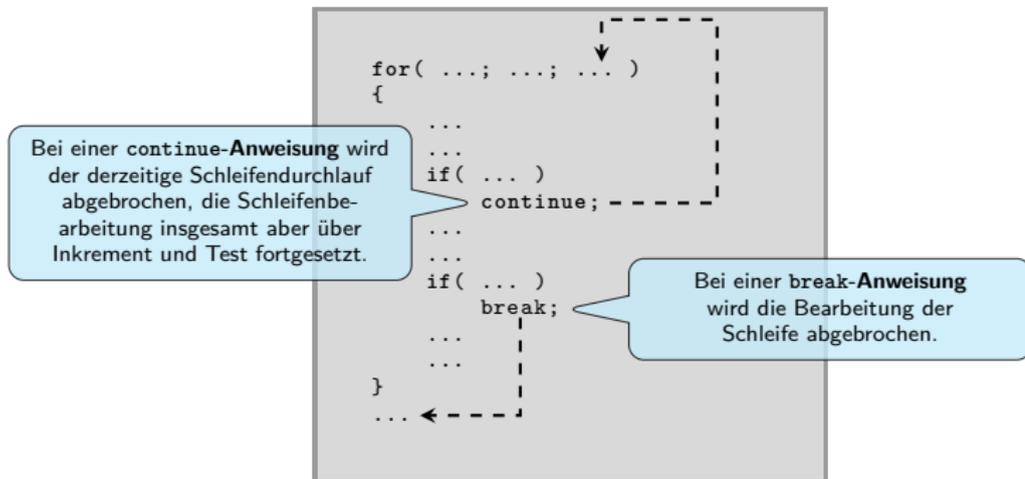
Bei einer **continue-Anweisung** wird der derzeitige Schleifendurchlauf abgebrochen, die Schleifenbearbeitung insgesamt aber über Inkrement und Test fortgesetzt.

```
for( ...; ...; ... )
{
    ...
    if( ... )
        continue;
    ...
    if( ... )
        break;
    ...
}
...
```



# Schleifenkontrolle aus dem Schleifenkörper

- Schleifen können aus dem Schleifenkörper mit `break` und `continue` gesteuert werden:



# Beispiele für Steuerung aus dem Schleifenkörper

- Alle Zahlen von 1 bis 100 werden addiert, dabei werden allerdings alle durch 7 teilbaren Zahlen übersprungen:

```
for(summe = 0, i = 1; i <= 100; i++)
{
    if( i%7 == 0)
        continue;
    summe = summe + i;
}
```

- Die Schleife wird darüber hinaus beendet, sobald sich in `summe` ein Wert größer als 1000 ergibt:

```
for(summe = 0, i = 1; i <= 100; i++)
{
    if( i%7 == 0)
        continue;
    summe = summe + i;
    if( summe > 1000)
        break;
}
```

# Vollständige Schleifensteuerung

```
for( ...; ...; ... )
{
    ...
    ...
    if( ... )
        continue;
    ...
    ...
    if( ... )
        break;
    ...
    ...
}
```

# Vollständige Schleifensteuerung

Die **Initialisierung** wird vor dem Eintritt in die Schleife einmal ausgeführt.

```
for( ...; ...; ... )  
{  
    ...  
    ...  
    if( ... )  
        continue;  
    ...  
    ...  
    if( ... )  
        break;  
    ...  
    ...  
}
```

# Vollständige Schleifensteuerung

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert `0`, so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Die **Initialisierung** wird vor dem Eintritt in die Schleife einmal ausgeführt.

```
for( ...; ...; ... )  
{  
    ...  
    ...  
    if( ... )  
        continue;  
    ...  
    ...  
    if( ... )  
        break;  
    ...  
    ...  
}
```

# Vollständige Schleifensteuerung

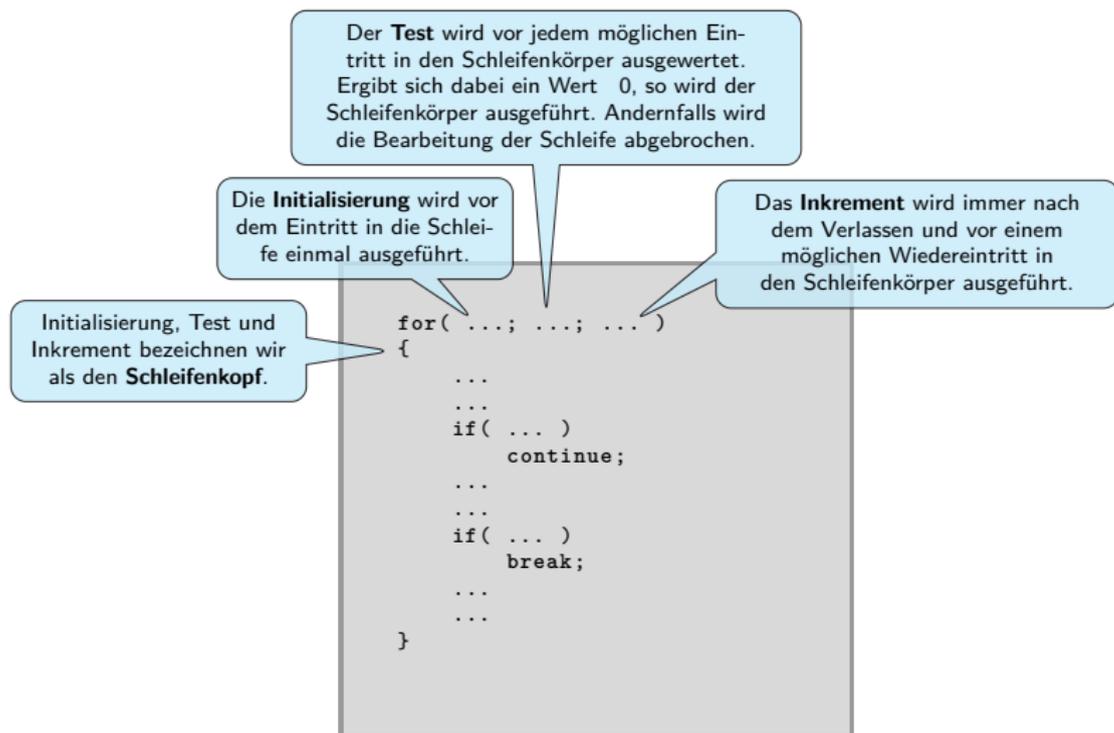
Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert 0, so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Die **Initialisierung** wird vor dem Eintritt in die Schleife einmal ausgeführt.

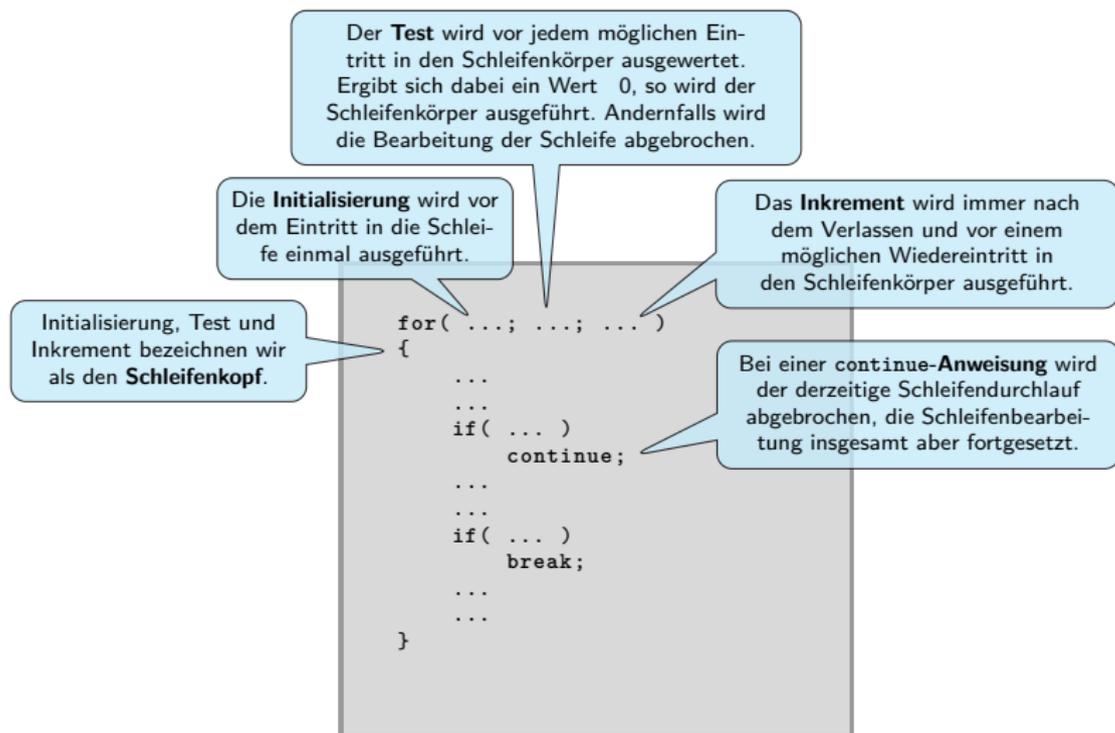
Das **Inkrement** wird immer nach dem Verlassen und vor einem möglichen Wiedereintritt in den Schleifenkörper ausgeführt.

```
for( ...; ...; ... )
{
    ...
    ...
    if( ... )
        continue;
    ...
    ...
    if( ... )
        break;
    ...
    ...
}
```

# Vollständige Schleifensteuerung



# Vollständige Schleifensteuerung



Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert 0, so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Die **Initialisierung** wird vor dem Eintritt in die Schleife einmal ausgeführt.

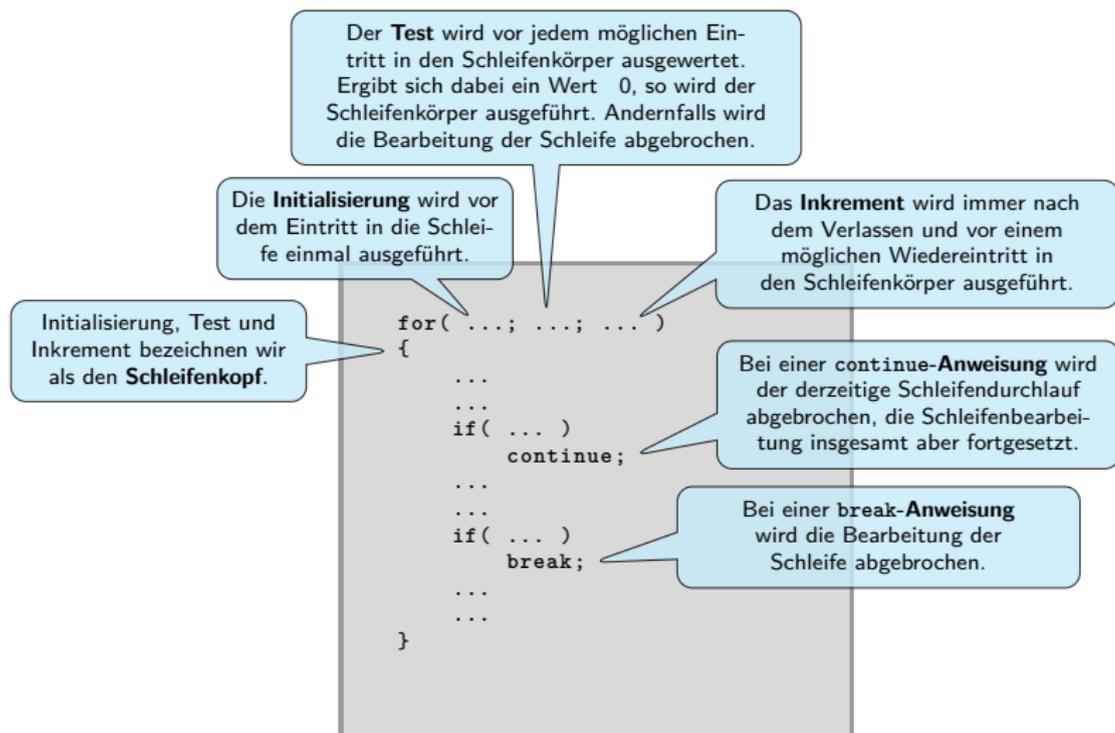
Das **Inkrement** wird immer nach dem Verlassen und vor einem möglichen Wiedereintritt in den Schleifenkörper ausgeführt.

Initialisierung, Test und Inkrement bezeichnen wir als den **Schleifenkopf**.

```
for( ...; ...; ... )
{
    ...
    if( ... )
        continue;
    ...
    if( ... )
        break;
    ...
}
```

Bei einer **continue-Anweisung** wird der derzeitige Schleifendurchlauf abgebrochen, die Schleifenbearbeitung insgesamt aber fortgesetzt.

# Vollständige Schleifensteuerung



```
for( ...; ...; ... )
{
    ...
    if( ... )
        continue;
    ...
    if( ... )
        break;
    ...
}
```

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert 0, so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Die **Initialisierung** wird vor dem Eintritt in die Schleife einmal ausgeführt.

Das **Inkrement** wird immer nach dem Verlassen und vor einem möglichen Wiedereintritt in den Schleifenkörper ausgeführt.

Initialisierung, Test und Inkrement bezeichnen wir als den **Schleifenkopf**.

Bei einer **continue-Anweisung** wird der derzeitige Schleifendurchlauf abgebrochen, die Schleifenbearbeitung insgesamt aber fortgesetzt.

Bei einer **break-Anweisung** wird die Bearbeitung der Schleife abgebrochen.

# Vollständige Schleifensteuerung

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert 0, so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Die **Initialisierung** wird vor dem Eintritt in die Schleife einmal ausgeführt.

Das **Inkrement** wird immer nach dem Verlassen und vor einem möglichen Wiedereintritt in den Schleifenkörper ausgeführt.

Initialisierung, Test und Inkrement bezeichnen wir als den **Schleifenkopf**.

```
for( ...; ...; ... )
{
    ...
    ...
    if( ... )
        continue;
    ...
    ...
    if( ... )
        break;
    ...
    ...
}
```

Bei einer **continue-Anweisung** wird der derzeitige Schleifendurchlauf abgebrochen, die Schleifenbearbeitung insgesamt aber fortgesetzt.

Der **Schleifenkörper** wird bei jedem Schleifendurchlauf ausgeführt. Besteht der Schleifenkörper nur aus einer einzelnen Anweisung, so können die geschweiften Klammern weggelassen werden.

Bei einer **break-Anweisung** wird die Bearbeitung der Schleife abgebrochen.

# Schleifen mit while

- Wenn eine Schleife keine Initialisierung und kein Inkrement benötigt, kann man statt einer for- auch eine while-Anweisung verwenden:

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

- Eigentlich ist while überflüssig, da die Funktionalität von while vollständig durch for abgedeckt ist. while(...) entspricht for( ;...;).

# Schleifen mit while

- Wenn eine Schleife keine Initialisierung und kein Inkrement benötigt, kann man statt einer for- auch eine while-Anweisung verwenden:

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

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

- Eigentlich ist while überflüssig, da die Funktionalität von while vollständig durch for abgedeckt ist. while(...) entspricht for( ;...;).

# Schleifen mit while

- Wenn eine Schleife keine Initialisierung und kein Inkrement benötigt, kann man statt einer for- auch eine while-Anweisung verwenden:

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

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

Bei einer **continue-Anweisung** wird der derzeitige Schleifendurchlauf abgebrochen, die Schleifenbearbeitung insgesamt aber fortgesetzt.

- Eigentlich ist while überflüssig, da die Funktionalität von while vollständig durch for abgedeckt ist. while(...) entspricht for( ;...;).

# Schleifen mit while

- Wenn eine Schleife keine Initialisierung und kein Inkrement benötigt, kann man statt einer for- auch eine while-Anweisung verwenden:

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

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

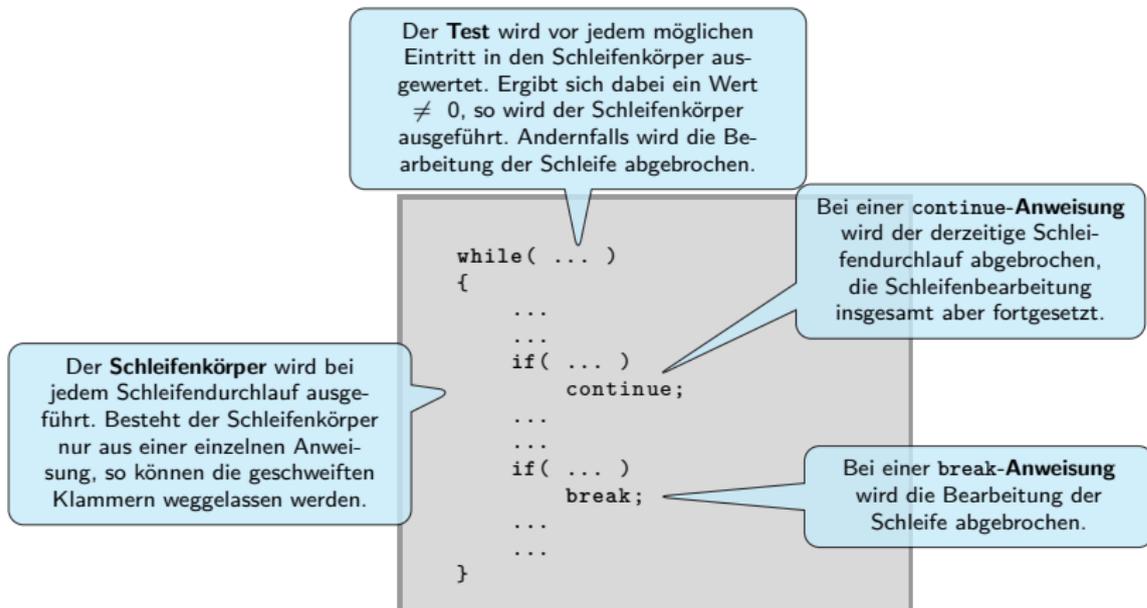
Bei einer **continue-Anweisung** wird der derzeitige Schleifendurchlauf abgebrochen, die Schleifenbearbeitung insgesamt aber fortgesetzt.

Bei einer **break-Anweisung** wird die Bearbeitung der Schleife abgebrochen.

- Eigentlich ist while überflüssig, da die Funktionalität von while vollständig durch for abgedeckt ist. while(...) entspricht for( ;...;).

# Schleifen mit while

- Wenn eine Schleife keine Initialisierung und kein Inkrement benötigt, kann man statt einer for- auch eine while-Anweisung verwenden:



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

Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert  $\neq 0$ , so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Bei einer **continue-Anweisung** wird der derzeitige Schleifendurchlauf abgebrochen, die Schleifenbearbeitung insgesamt aber fortgesetzt.

Bei einer **break-Anweisung** wird die Bearbeitung der Schleife abgebrochen.

Der **Schleifenkörper** wird bei jedem Schleifendurchlauf ausgeführt. Besteht der Schleifenkörper nur aus einer einzelnen Anweisung, so können die geschweiften Klammern weggelassen werden.

- Eigentlich ist while überflüssig, da die Funktionalität von while vollständig durch for abgedeckt ist. `while(...)` entspricht `for( ;...; )`.

# Schleifen in Schleifen

- Schleifen können im Schleifenkörper wieder Schleifen enthalten.
- Berechnung des „kleinen Einmaleins“ durch zwei ineinander geschachtelte Zählschleifen:

```
int produkt;  
for( i = 1; i <= 10; i = i + 1)  
{  
    for( k = 1; k <= 10; k = k + 1)  
        produkt = i * k;  
}
```

- Die Variable *i* durchläuft in der äußeren Schleife die Werte von 1 bis 10. Für jeden Wert von *i* durchläuft dann die Variable *k* in der inneren Schleife ebenfalls die Werte von 1 bis 10. Insgesamt wird damit die Berechnung in der inneren Schleife 100-mal für alle möglichen Kombinationen von *i* und *k* ausgeführt.

# Verschachtelte Kontrollstrukturen

- Schleifen und Fallunterscheidungen können beliebig ineinander geschachtelt werden:

```
int produkt;
for( i = 1; i <= 10; i = i + 1)
{
    if( i % 2 == 0)
    {
        for( k = 1; k <= 10; k = k + 1)
        {
            if( k % 2 == 0)
                produkt = i * k;
        }
    }
}
```

- Nur wenn  $i$  gerade ist, wird in die innere Schleife über  $k$  eingetreten und dort wird das Produkt nur dann berechnet, wenn  $k$  ebenfalls gerade ist.

# Bildschirmausgabe

- Um einen Text auf dem Bildschirm auszugeben, verwenden wir `printf` :

```
printf( "Dieser Text wird ausgegeben\n");
```

- Der auszugebende Text wird in doppelte Hochkommata eingeschlossen. Die am Ende des Textes stehende Zeichenfolge `\n` erzeugt einen Zeilenvorschub.

# Formatierte Ausgabe

- In den auszugebenden Text kann man Zahlenwerte einstreuen, indem man als Platzhalter für die fehlenden Zahlenwerte eine sogenannte Formatanweisung einfügt. Eine solche Formatanweisung besteht zum Beispiel aus einem Prozentzeichen gefolgt von dem Buchstaben d (für Dezimalwert). Die zugehörigen Werte werden dann als Konstanten oder Variablen durch Kommata getrennt hinter dem Ausgabebetext angefügt:

```
int wert = 1;
printf( "Die %d. Zeile hat %d Buchstaben!\n", wert, 26);
wert = 2;
printf( "Dies ist die %d. Zeile!\n", wert);
```

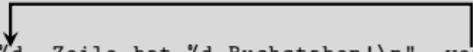
- Ausgabe:

```
Die 1. Zeile hat 26 Buchstaben!
Dies ist die 2. Zeile!
```

# Formatierte Ausgabe

- In den auszugebenden Text kann man Zahlenwerte einstreuen, indem man als Platzhalter für die fehlenden Zahlenwerte eine sogenannte Formatanweisung einfügt. Eine solche Formatanweisung besteht zum Beispiel aus einem Prozentzeichen gefolgt von dem Buchstaben d (für Dezimalwert). Die zugehörigen Werte werden dann als Konstanten oder Variablen durch Kommata getrennt hinter dem Ausgabebetext angefügt:

```
int wert = 1;
printf( "Die %d. Zeile hat %d Buchstaben!\n", wert, 26);
wert = 2;
printf( "Dies ist die %d. Zeile!\n", wert);
```



- Ausgabe:

```
Die 1. Zeile hat 26 Buchstaben!
Dies ist die 2. Zeile!
```

# Formatierte Ausgabe



- In den auszugebenden Text kann man Zahlenwerte einstreuen, indem man als Platzhalter für die fehlenden Zahlenwerte eine sogenannte Formatanweisung einfügt. Eine solche Formatanweisung besteht zum Beispiel aus einem Prozentzeichen gefolgt von dem Buchstaben d (für Dezimalwert). Die zugehörigen Werte werden dann als Konstanten oder Variablen durch Kommata getrennt hinter dem Ausgabebetext angefügt:

```
int wert = 1;
printf( "Die %d. Zeile hat %d Buchstaben!\n", wert, 26);
wert = 2;
printf( "Dies ist die %d. Zeile!\n", wert);
```

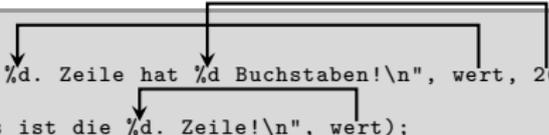
- Ausgabe:

```
Die 1. Zeile hat 26 Buchstaben!
Dies ist die 2. Zeile!
```

# Formatierte Ausgabe

- In den auszugebenden Text kann man Zahlenwerte einstreuen, indem man als Platzhalter für die fehlenden Zahlenwerte eine sogenannte Formatanweisung einfügt. Eine solche Formatanweisung besteht zum Beispiel aus einem Prozentzeichen gefolgt von dem Buchstaben d (für Dezimalwert). Die zugehörigen Werte werden dann als Konstanten oder Variablen durch Kommata getrennt hinter dem Ausgabebetext angefügt:

```
int wert = 1;
printf( "Die %d. Zeile hat %d Buchstaben!\n", wert, 26);
wert = 2;
printf( "Dies ist die %d. Zeile!\n", wert);
```



- Ausgabe:

```
Die 1. Zeile hat 26 Buchstaben!
Dies ist die 2. Zeile!
```

# Ausgabe von Gleitkommazahlen

- Zur Ausgabe von Gleitkommazahlen verwendet man die Formatanweisung %f.

```
float preis;  
preis = 10.99;  
printf( "Die Ware kostet %f EURO\n", preis);
```

- Ausgabe:

```
Die Ware kostet 10.990000 EURO
```

- Später werden wir weitere Formatanweisungen kennenlernen. Vorerst reicht die Ausgabe von Dezimalwerten und Gleitkommawerten für unsere Zwecke vollständig aus:
  - ▶ Ausgabe von Dezimalwerten mit %d
  - ▶ Ausgabe von Gleitkommawerten mit %f

# Das kleine Einmaleins mit Ausgabe



```
int produkt, i, k;
for( i = 1; i <= 10; i++)
{
    for( k = 1; k <= 10; k++)
    {
        produkt = i*k;
        printf( "%d mal %d ist %d\n", i, k, produkt);
    }
    printf( "\n");
}
```

```
1 mal 1 ist 1
1 mal 2 ist 2
1 mal 3 ist 3
1 mal 4 ist 4
1 mal 5 ist 5
1 mal 6 ist 6
1 mal 7 ist 7
1 mal 8 ist 8
1 mal 9 ist 9
1 mal 10 ist 10

2 mal 1 ist 2
2 mal 2 ist 4
2 mal 3 ist 6
.....
10 mal 7 ist 70
10 mal 8 ist 80
10 mal 9 ist 90
10 mal 10 ist 100
```

# Tastatureingabe

- Eine oder mehrere ganze Zahlen lesen wir mit scanf von der Tastatur ein:

```
int zahl1, zahl2;

printf( "Bitte geben Sie zwei Zahlen ein: ");

scanf( "%d %d", &zahl1, &zahl2);
printf( "Sie haben %d und %d eingegeben\n", zahl1, zahl2);
```

- Zugehöriger Bildschirmdialog:

```
Bitte geben Sie zwei Zahlen ein: 123 456
Sie haben 123 und 456 eingegeben
```

- Für die Eingabe von Gleitkommazahlen verwenden Sie Gleitkommavariablen und die Formatanweisung %f.
- Beim Einlesen müssen Variablen angegeben werden, denen die Werte zugewiesen werden sollen.
- **Zusätzlich muss den Variablennamen beim Einlesen ein & vorangestellt werden.**
- Die exakte Bedeutung des &-Zeichens wird später erklärt.

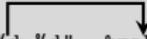
# Tastatureingabe

- Eine oder mehrere ganze Zahlen lesen wir mit scanf von der Tastatur ein:

```
int zahl1, zahl2;

printf( "Bitte geben Sie zwei Zahlen ein: ");

scanf( "%d %d", &zahl1, &zahl2);
printf( "Sie haben %d und %d eingegeben\n", zahl1, zahl2);
```



- Zugehöriger Bildschirmdialog:

```
Bitte geben Sie zwei Zahlen ein: 123 456
Sie haben 123 und 456 eingegeben
```

- Für die Eingabe von Gleitkommazahlen verwenden Sie Gleitkommavariablen und die Formatanweisung %f.
- Beim Einlesen müssen Variablen angegeben werden, denen die Werte zugewiesen werden sollen.
- **Zusätzlich muss den Variablennamen beim Einlesen ein & vorangestellt werden.**
- Die exakte Bedeutung des &-Zeichens wird später erklärt.

# Tastatureingabe

- Eine oder mehrere ganze Zahlen lesen wir mit scanf von der Tastatur ein:

```
int zahl1, zahl2;

printf( "Bitte geben Sie zwei Zahlen ein: ");

scanf( "%d %d", &zahl1, &zahl2);
printf( "Sie haben %d und %d eingegeben\n", zahl1, zahl2);
```



- Zugehöriger Bildschirmdialog:

```
Bitte geben Sie zwei Zahlen ein: 123 456
Sie haben 123 und 456 eingegeben
```

- Für die Eingabe von Gleitkommazahlen verwenden Sie Gleitkommavariablen und die Formatanweisung %f.
- Beim Einlesen müssen Variablen angegeben werden, denen die Werte zugewiesen werden sollen.
- **Zusätzlich muss den Variablennamen beim Einlesen ein & vorangestellt werden.**
- Die exakte Bedeutung des &-Zeichens wird später erklärt.

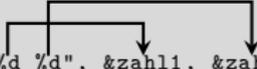
# Tastatureingabe

- Eine oder mehrere ganze Zahlen lesen wir mit scanf von der Tastatur ein:

```
int zahl1, zahl2;

printf( "Bitte geben Sie zwei Zahlen ein: ");

scanf( "%d %d", &zahl1, &zahl2);
printf( "Sie haben %d und %d eingegeben\n", zahl1, zahl2);
```



- Zugehöriger Bildschirmdialog:

```
Bitte geben Sie zwei Zahlen ein: 123 456
Sie haben 123 und 456 eingegeben
```

- Für die Eingabe von Gleitkommazahlen verwenden Sie Gleitkommavariablen und die Formatanweisung %f.
- Beim Einlesen müssen Variablen angegeben werden, denen die Werte zugewiesen werden sollen.
- **Zusätzlich muss den Variablennamen beim Einlesen ein & vorangestellt werden.**
- Die exakte Bedeutung des &-Zeichens wird später erklärt.

# Tastatureingabe

- Eine oder mehrere ganze Zahlen lesen wir mit scanf von der Tastatur ein:

```
int zahl1, zahl2;

printf( "Bitte geben Sie zwei Zahlen ein: ");

scanf( "%d %d", &zahl1, &zahl2);
printf( "Sie haben %d und %d eingegeben\n", zahl1, zahl2);
```



- Zugehöriger Bildschirmdialog:

```
Bitte geben Sie zwei Zahlen ein: 123 456
Sie haben 123 und 456 eingegeben
```

- Für die Eingabe von Gleitkommazahlen verwenden Sie Gleitkommavariablen und die Formatanweisung %f.
- Beim Einlesen müssen Variablen angegeben werden, denen die Werte zugewiesen werden sollen.
- **Zusätzlich muss den Variablennamen beim Einlesen ein & vorangestellt werden.**
- Die exakte Bedeutung des &-Zeichens wird später erklärt.

# Das kleine Einmaleins mit Ein- und Ausgabe

```
void main()
{
    int i, k;
    int maxi, maxk;
    int produkt;

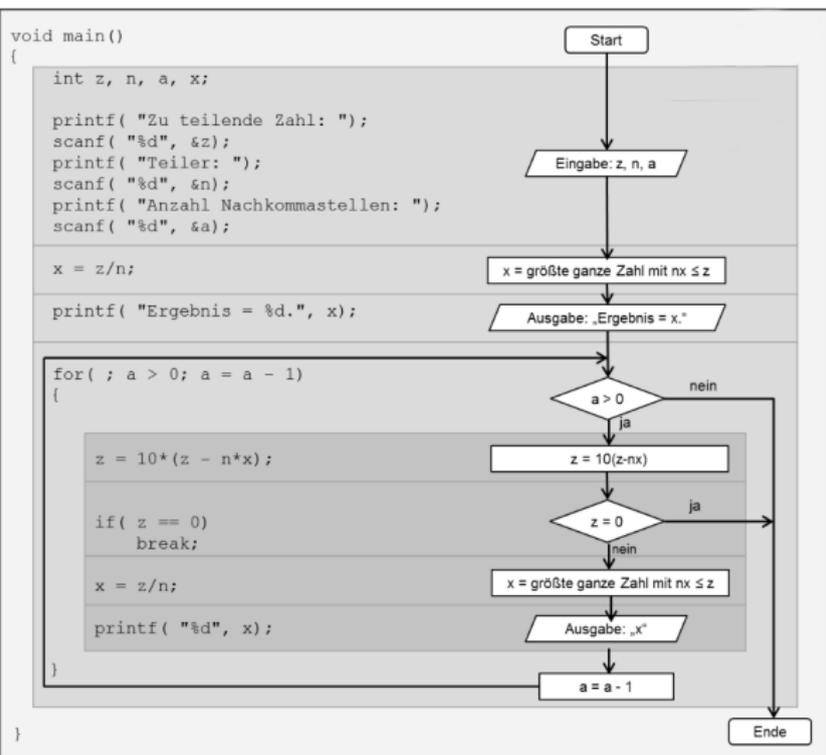
    printf( "Bitte maxi eingeben: ");
    scanf( "%d", &maxi);
    printf( "Bitte maxk eingeben: ");
    scanf( "%d", &maxk);

    for( i = 1; i <= 10; i++)
    {
        for( k = 1; k <= 10; k++)
        {
            produkt = i*k;
            printf( "%d mal %d ist %d\n", i, k, produkt);
        }
        printf( "\n");
    }
}
```

```
Bitte maxi eingeben: 3
Bitte maxk eingeben: 5
1 mal 1 ist 1
1 mal 2 ist 2
1 mal 3 ist 3
1 mal 4 ist 4
1 mal 5 ist 5

2 mal 1 ist 2
2 mal 2 ist 4
2 mal 3 ist 6
2 mal 4 ist 8
2 mal 5 ist 10
.....
3 mal 1 ist 3
3 mal 2 ist 6
3 mal 3 ist 9
3 mal 4 ist 12
3 mal 5 ist 15
```

# Das erste Programm

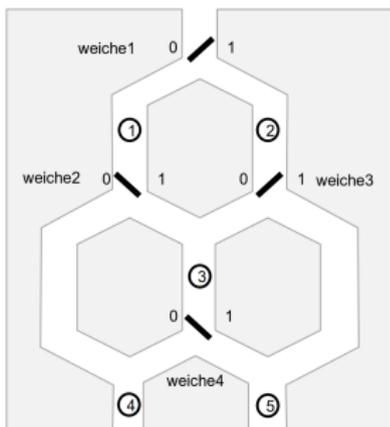


Zu teilende Zahl: 84  
 Teiler: 16  
 Anzahl Nachkommastellen: 4  
 Ergebnis = 5.25

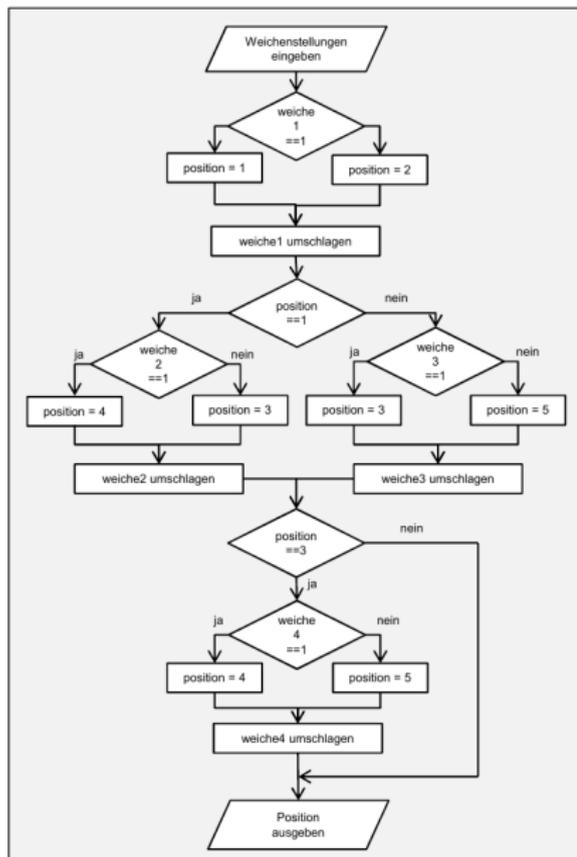
Zu teilende Zahl: 100  
 Teiler: 7  
 Anzahl Nachkommastellen: 6  
 Ergebnis = 14.285714

# Das zweite Programm

- Eine Kugel durchläuft den folgenden Parcours.



- Passiert die Kugel eine Weiche, wird sie in die entsprechende Richtung gelenkt und die Weiche schlägt um. Abhängig von den Stellungen der vier Weichen (0 oder 1) erreicht die Kugel einen von zwei möglichen Ausgängen (4 oder 5).
- Erstelle ein Programm, das nach Eingabe der Weichenstellungen durch den Benutzer den Ausgang berechnet.



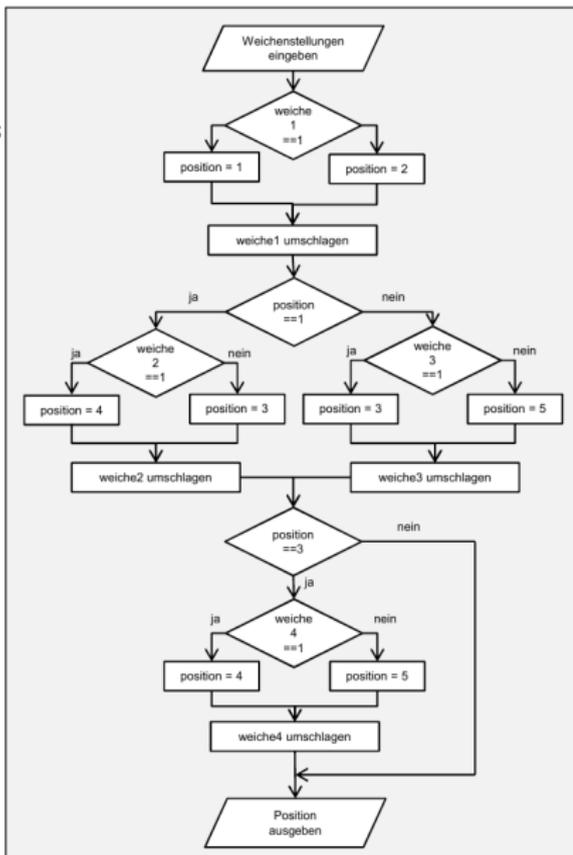
# Das zweite Programm

```

void main()
{
    int weiche1, weiche2, weiche3, weiche4;
    int position;
    printf( "Bitte geben Sie die Weichenstellungen ein: " );
    scanf( "%d %d %d %d", &weiche1, &weiche2,
        &weiche3, &weiche4;

    if( weiche1 == 1 )
        position = 1;
    else
        position = 2;
    weiche1 = 1 - weiche1;
    if( position == 1 )
    {
        if( weiche2 == 1 )
            position = 4;
        else
            position = 3;
        weiche2 = 1 - weiche2;
    }
    else
    {
        if( weiche3 == 1 )
            position = 3;
        else
            position = 5;
        weiche3 = 1 - weiche3;
    }
    if( position == 3 )
    {
        if( weiche4 == 1 )
            position = 4;
        else
            position = 5;
        weiche4 = 1 - weiche4;
    }
    printf( "Auslauf: %d, ", position);
    printf( "neue Weichenstellung %d %d %d %d\n",

```



# Das zweite Programm



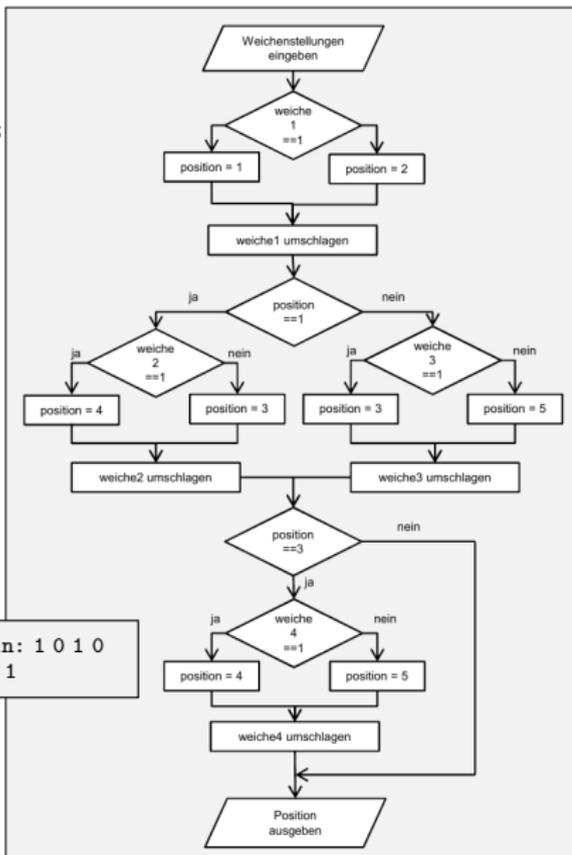
```

void main()
{
    int weiche1, weiche2, weiche3, weiche4;
    int position;
    printf( "Bitte geben Sie die Weichenstellungen ein: " );
    scanf( "%d %d %d %d", &weiche1, &weiche2,
            &weiche3, &weiche4);

    if( weiche1 == 1)
        position = 1;
    else
        position = 2;
    weiche1 = 1 - weiche1;
    if( position == 1)
    {
        if( weiche2 == 1)
            position = 4;
        else
            position = 3;
        weiche2 = 1 - weiche2;
    }
    else
    {
        if( weiche3 == 1)
            position = 3;
        else
            position = 5;
        weiche3 = 1 - weiche3;
    }
    if( position == 3)
    {
        if( weiche4 == 1)
            position = 4;
        else
            position = 5;
        weiche4 = 1 - weiche4;
    }
    printf( "Auslauf: %d, ", position);
    printf( "neue Weichenstellung %d %d %d %d\n",

```

Bitte geben Sie die Weichenstellungen ein: 1 0 1 0  
Auslauf: 5, neue Weichenstellungen 0 1 1 1



# Das zweite Programm

## ● Erweiterung des Programms auf mehrere Kugeldurchläufe

```
void main()
{
    int weiche1, weiche2, weiche3, weiche4;
    int position;
    int kugeln;

    printf( "Bitte geben Sie die Weichenstellungen ein: ");
    scanf( "%d %d %d %d", &weiche1, &weiche2, &weiche3, &weiche4);
    printf( "Bitte geben Sie die Anzahl der Kugeln ein: ");
    scanf( "%d", &kugeln);

    for( ; kugeln > 0; kugeln = kugeln - 1)
    {
        ... wie bisher ...
    }
}
```

```
Bitte geben Sie die Weichenstellungen ein: 0 1 0 1
Bitte geben Sie die Anzahl der Kugeln ein: 5
Auslauf: 5, neue Weichenstellung 1 1 1 1
Auslauf: 4, neue Weichenstellung 0 0 1 1
Auslauf: 4, neue Weichenstellung 1 0 0 0
Auslauf: 5, neue Weichenstellung 0 1 0 1
Auslauf: 5, neue Weichenstellung 1 1 1 1
```

# Das dritte Programm

- Der Benutzer soll eine von ihm festgelegte Anzahl von Zahlen eingeben. Das Programm summiert die positiven und die negativen Eingaben und gibt am Ende die Summe der negativen und der positiven Eingaben sowie die Gesamtsumme aus.

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

void main()
{
    int anzahl;
    int z;
    int summand;
    int psum;
    int nsum;

    printf( "Wie viele Zahlen sollen eingegeben werden: ");
    scanf( "%d", &anzahl);

    psum = 0;
    nsum = 0;

    for( z = 1; z <= anzahl; z = z + 1)
    {
        printf( "%d. Zahl: ", z);
        scanf( "%d", &summand);

        if( summand > 0)
            psum = psum + summand;
        else
            nsum = nsum + summand;
    }

    printf( "Summe aller positiven Eingaben: %d\n", psum);
    printf( "Summe aller negativen Eingaben: %d\n", nsum);
    printf( "Gesamtsumme: %d\n", psum + nsum);
}
```

# Das dritte Programm

- Der Benutzer soll eine von ihm festgelegte Anzahl von Zahlen eingeben. Das Programm summiert die positiven und die negativen Eingaben und gibt am Ende die Summe der negativen und der positiven Eingaben sowie die Gesamtsumme aus.

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

void main()
{
    int anzahl;
    int z;
    int summand;
    int psum;
    int nsum;

    printf( "Wie viele Zahlen sollen eingegeben werden:
scanf( "%d", &anzahl);

    psum = 0;
    nsum = 0;

    for( z = 1; z <= anzahl; z = z + 1)
    {
        printf( "%d. Zahl: ", z);
        scanf( "%d", &summand);

        if( summand > 0)
            psum = psum + summand;
        else
            nsum = nsum + summand;
    }
    printf( "Summe aller positiven Eingaben: %d\n", psum);
    printf( "Summe aller negativen Eingaben: %d\n", nsum);
    printf( "Gesamtsumme: %d\n", psum + nsum);
}
```

```
Wieviele Zahlen sollen eingegeben werden:
8
1. Zahl: 1
2. Zahl: 2
3. Zahl: -5
4. Zahl: 4
5. Zahl: 5
6. Zahl: -8
7. Zahl: 3
8. Zahl: -7
```

# Das dritte Programm

- Der Benutzer soll eine von ihm festgelegte Anzahl von Zahlen eingeben. Das Programm summiert die positiven und die negativen Eingaben und gibt am Ende die Summe der negativen und der positiven Eingaben sowie die Gesamtsumme aus.

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

void main()
{
    int anzahl;
    int z;
    int summand;
    int psum;
    int nsum;

    printf( "Wie viele Zahlen sollen eingegeben werden:
scanf( "%d", &anzahl);

    psum = 0;
    nsum = 0;

    for( z = 1; z <= anzahl; z = z + 1)
    {
        printf( "%d. Zahl: ", z);
        scanf( "%d", &summand);

        if( summand > 0)
            psum = psum + summand;
        else
            nsum = nsum + summand;
    }
    printf( "Summe aller positiven Eingaben: %d\n", psum);
    printf( "Summe aller negativen Eingaben: %d\n", nsum);
    printf( "Gesamtsumme: %d\n", psum + nsum);
}
```

Wieviele Zahlen sollen eingegeben werden:

8

1. Zahl: 1
2. Zahl: 2
3. Zahl: -5
4. Zahl: 4
5. Zahl: 5
6. Zahl: -8
7. Zahl: 3
8. Zahl: -7

Die Summe aller positiven Eingaben ist: 15  
Die Summe aller negativen Eingaben ist: -20  
Die Gesamtsumme ist: -5

# Zusammenfassung

- In diesem Kapitel wurde einige grundlegende Elemente der Sprache C eingeführt
  - ▶ Programmrahmen
  - ▶ Integer- und Gleitkomma-Zahlenkonstanten und -Variablen
  - ▶ Wertzuweisungen, Arithmetische Operationen und Formelausdrücke
  - ▶ Operatoren mit gleichzeitiger Wertzuweisung
  - ▶ Ganzzahl- und Gleitkomma-Arithmetik, Cast
  - ▶ Vergleichsoperatoren und Fallunterscheidungen
  - ▶ Schleifen mit `for` und `while`
  - ▶ Bildschirmausgabe und Tastatureingabe
- Einige erste, mit diesem Rüstzeug realisierte Programme wurden vorgestellt.