

Einführung in die Programmierung

MIT DER PROGRAMMIERSPRACHE C

PROF. DR. THOMAS GABEL

Überblick über die Vorlesung

1. Algorithmen, Programme und Software
2. Einstieg in die Programmierung mit C
3. **Strukturiertes Programmieren in C**
4. Effizientes Programmieren in C
5. Fortgeschrittene Aspekte der Programmierung in C



3. Strukturiertes Programmieren in C

1. **Kontrollstrukturen**
2. Einfache Anweisungen
3. Anweisungsblöcke
4. Verzweigungsanweisungen
5. Funktionsaufrufe
6. Schleifenanweisungen
7. Sprunganweisungen
8. Parameterübergabemechanismen

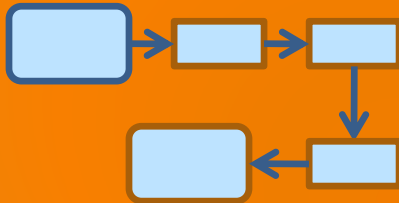


Kontrollstrukturen

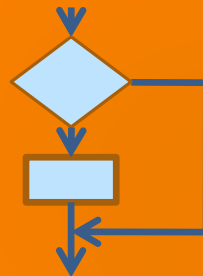
Die Reihenfolge der Ausführung von **Aktionen** wird unter Verwendung von **Kontrollstrukturen** festgelegt.

Es lässt sich mathematisch beweisen, dass alle Algorithmen unter Verwendung von lediglich drei Kontrollstrukturen formulierbar sind:

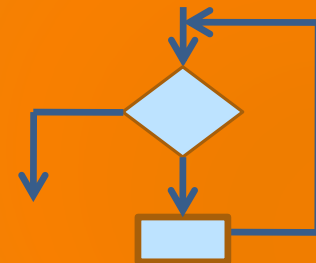
Sequenz



Alternative

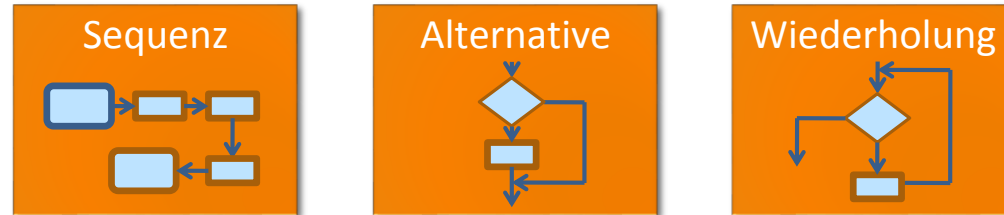


Wiederholung



Anweisungen

Anweisungen sind dazu da, um Kontrollstrukturen umzusetzen.



Anweisungen sind programmiersprachliche Beschreibungsmittel.

- Einfache Anweisungen beschreiben Aktionen.
- Zusammengesetzte Anweisungen beschreiben, wie mehrere Anweisungen in Folge auszuführen sind.
- Zusammengesetzte Anweisungen bilden Teile von Algorithmen.

Bereits kennengelernte Anweisungen:

- Zuweisung
- Sequenzen von Anweisungen / Blöcke

Kontrollstrukturen: Überblick

Überblick über die Arten von Anweisungen

- zur Realisierung von Kontrollstrukturen in C

Einfache Anweisungen

- Wertzuweisung
- Funktionsaufruf

Anweisungsblöcke

Schleifenanweisungen

- while-Schleife
- do-Schleife
- for-Anweisung

Verzweigungsanweisungen

- bedingte Anweisung
- Fallunterscheidung
- Auswahlentscheidung

Sprunganweisungen

- Abbruchanweisung
- Rückgabeeinweisungen

3. Strukturiertes Programmieren in C

1. Kontrollstrukturen
2. **Einfache Anweisungen**
3. Anweisungsblöcke
4. Verzweigungsanweisungen
5. Funktionsaufrufe
6. Schleifenanweisungen
7. Sprunganweisungen
8. Parameterübergabemechanismen



Kontrollstrukturen: Überblick

Überblick über die Arten von Anweisungen

- zur Realisierung von Kontrollstrukturen in C

Einfache Anweisungen

- Wertzuweisung
- Funktionsaufruf

Anweisungsblöcke

Schleifenanweisungen

- while-Schleife
- do-Schleife
- for-Anweisung

Verzweigungsanweisungen

- bedingte Anweisung
- Fallunterscheidung
- Auswahlentscheidung

Sprunganweisungen

- Abbruchanweisung
- Rückgabeeinweisungen

Einfache Anweisung: Wertzuweisung (Wdh.)

Die **Wertzuweisung** ist eine elementare Aktion, bei der einer Variable ein konstanter oder zu berechnender Wert zugewiesen wird.

Pseudocode

```
berechne sum als Summe x+y
```

Flussdiagramm

```
setze sum auf x+y
```

C

```
sum = x+y;
```

- Ziel der Wertzuweisung wird durch den Namen auf der linken Seite des Zuweisungsoperators „=“ gegeben.
- Zuzuweisender Wert ergibt sich aus der Auswertung des Ausdrucks auf der rechten Seite. Abschluss mit Semikolon.
- Bei Zuweisung geht der alte Wert des Ziels verloren.

3. Strukturiertes Programmieren in C

1. Kontrollstrukturen
2. Einfache Anweisungen
3. **Anweisungsblöcke**
4. Verzweigungsanweisungen
5. Funktionsaufrufe
6. Schleifenanweisungen
7. Sprunganweisungen
8. Parameterübergabemechanismen



Anweisungen: Überblick

Überblick über die Arten von Anweisungen

- zur Realisierung von Kontrollstrukturen in C



Anweisungsblock (Aktionssequenz) (Wdh.)

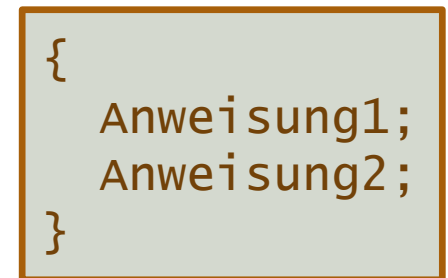
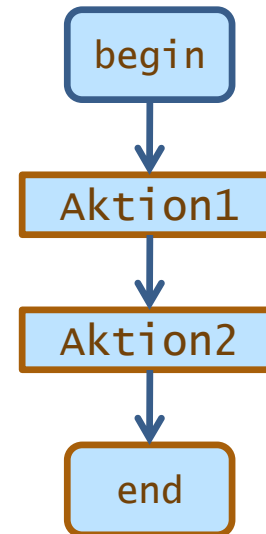
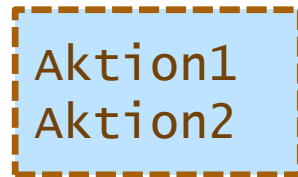
Mehrere Anweisungen können zu einem **Anweisungsblock** zusammengesetzt werden.

Damit wird eine sequentielle Reihenfolge festgelegt, die besagt, dass eine Anweisung erst dann ausgeführt werden kann, wenn alle vorherigen Anweisungen beendet sind.

Pseudocode

Flussdiagramm

C



- Anweisungsblöcke werden mit geschweiften Klammern geklammert.
- Einzelne Anweisungen werden mit Semikolon abgeschlossen.

Anweisungsblöcke in C (Wdh.)

Statt von einem **Anweisungsblock** spricht man in Programmiersprachen auch von einer **zusammengesetzter Anweisung**, einer **Aktionssequenz** oder **Block**.

Die Reihenfolge der Anweisungen im Programmtext legt die Reihenfolge ihrer Ausführung fest.

Die Zeichen „{“ und „}“ dienen als Klammerung.

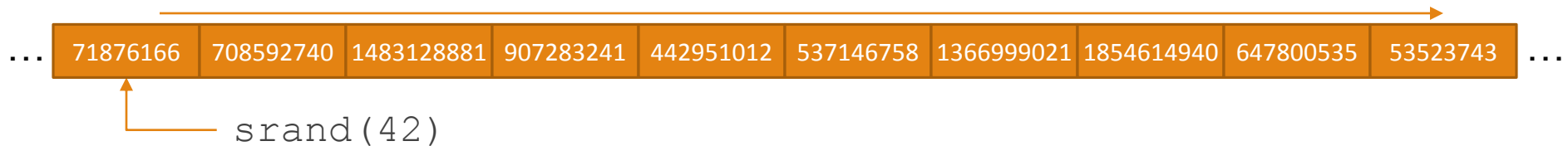
Beispiel:

```
...  
i = 4 + 2*j; // Anweisung mit Semikolon abgeschlossen  
...  
{ // Beginn des Blockes  
    int x;  
    int y;  
    int sum;  
    scanf( "%d", &x );  
    scanf( "%d", &y );  
    sum = x + y;  
    printf( "%d", sum );  
} // Blockende, kein Semikolon!
```

Zwischenschub: Zufallsgenerator in C

Mit der in `stdlib.h` enthaltenen Funktion **`rand()`** lassen sich **Zufallszahlen** erzeugen.

- Aufruf von `rand()` liefert eine zufällige natürliche Zahl zwischen 0 und `RAND_MAX`
- `RAND_MAX` ist eine „sehr große“ Zahl; rechnerarchitekturabhängig; mind. 32767
- Die erzeugte Zahl ist pseudozufällig.
- Pseudozufällig heißt: Aufeinanderfolgende Aufrufe von `rand()` liefern die gleiche Folge von Zufallszahlen aus einer (sehr langen) Zahlensequenz.



Startpunkt innerhalb jener Zahlensequenz kann mit **`srand()`** gesetzt werden.

- sogenanntes „seeding“
- Vorteil/Nutzen: Wiederholtes Laufenlassen eines Programms mit gleichem Seeding erzeugt innerhalb des Programms die gleiche Folge von Zufallszahlen.
- Frage: Wie bekomme ich „echte“ Zufallszahlen?
- Antwort: Durch `srand(time(NULL));`, d.h. Seeding mit Systemzeit.
 - erfordert `#include <time.h>`

Beispiel:

Anweisungsblöcke / Zufallszahlen

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

```
int main(void)
{
```

```
    srand(time(NULL)); // use current time as seed for random generator
    int random_value = rand();
    printf("Zufallszahl aus [0,%d]: %d\n", RAND_MAX, random_value);
```

```
    int lower, upper;
    printf("Bitte untere Grenze angeben: ");
    scanf("%d", &lower);
    printf("Bitte obere Grenze angeben: ");
    scanf("%d", &upper);
    random_value = (rand() % (upper-lower+1)) + lower;
    printf("Zufallszahl zwischen %d und %d: %d\n",
           lower, upper, random_value );
```

```
}
```

Aufgabe: Schreiben Sie ein Programm, das

- a) eine Zufallszahl sowie RAND_MAX ausgibt
- b) den Nutzer einen Wertebereich [von,bis] eingeben lässt und dann eine Zufallszahl in diesem Intervall ermittelt und ausgibt!

3. Strukturiertes Programmieren in C

1. Kontrollstrukturen
2. Einfache Anweisungen
3. Anweisungsblöcke
4. **Verzweigungsanweisungen**
5. Funktionsaufrufe
6. Schleifenanweisungen
7. Sprunganweisungen
8. Parameterübergabemechanismen



Anweisungen: Überblick

Überblick über die Arten von Anweisungen

- zur Realisierung von Kontrollstrukturen in C

Einfache Anweisungen

- Wertzuweisung
- Funktionsaufruf

Anweisungsblöcke

Schleifenanweisungen

- while-Schleife
- do-Schleife
- for-Anweisung

Verzweigungsanweisungen

- bedingte Anweisung
- Fallunterscheidung
- Auswahl

Alternative

Sprunganweisungen

- Abbruchanweisung
- Rückgabanweisungen

Verzweigungsanweisungen (1)

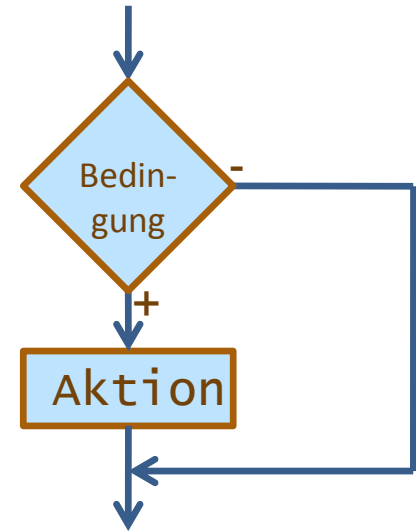
- **bedingte Anweisung**
 - Fallunterscheidung
 - Auswahlanweisung

Idee: **Bedingte Anweisungen** werden in Abhängigkeit vom Erfülltsein bzw. Nichterfülltsein einer Bedingung ausgeführt.

Pseudocode

falls Bedingung erfüllt,
so führe Aktion aus

Flussdiagramm



C

- Schlüsselwort **if**
- Bedingung dient der Entscheidung
- Klammerung des Bedingungsteils ist verpflichtend
- Werte den Ausdruck aus; ergibt dessen Auswertung „wahr“, so führe die Anweisung aus. Ansonsten ist die Ausführung sofort beendet.

Anweisung bezeichnet hier einfache Anweisungen wie auch Anweisungsblöcke.

`if (<boolescherAusdruck>) <Anweisung>`

Verzweigungsanweisungen (2)

- **bedingte Anweisung**
- Fallunterscheidung
- Auswahlanweisung

Bemerkungen zu logischen Bedingungen in C

- **Es existiert kein logischer Datentyp in C!**
 - In anderen Sprachen (Java, C++) existiert ein solcher (boolean, bool).
- Daher gilt als „falsch“ (false): jeder Ausdruck, dessen Wert null (**== 0**) ist
- Daher gilt als „wahr“ (true): jeder Ausdruck, dessen Wert ungleich null (**!= 0**) ist
- Um komplexere logische Bedingungen zu konstruieren, kann und sollte man die kennengelernten **logischen Operatoren** benutzen.

Beispiele: `if (v) ... ;`
`if (!w) ... ;`
`if (a > 2) ... ;`
`if (b <= 3 && c > 42) ... ;`
`if (d != 0 && (e < -3 || f == 'X')) ... ;`

Besonderheiten:

- unbedingt auf Prioritäten achten → am besten Klammern setzen
- Ausdrücke mit **&&** oder **||** werden nur solange von links nach rechts ausgewertet, bis das Ergebnis feststeht

Verzweigungsanweisungen (3)

- **bedingte Anweisung**
 - Fallunterscheidung
 - Auswahlanweisung

Bemerkungen zum Anweisungsteil der bedingten Anweisung

- im Anweisungsteil kann entweder eine einfache Anweisung oder ein Anweisungsblock enthalten sein

```
if (<boolescherAusdruck>) <Anweisung>
```

<Anweisung> bezeichnet hier einfache Anweisungen wie auch Anweisungsblöcke.

Beispiele:

- einzelne Anweisung, einzelne Anweisung im Block, Block mit mehreren Anweisungen

```
if ( a <= 2 )  
    printf("Gut");
```

```
if ( a <= 2 )  
{  
    printf("Gut");  
}
```

```
char n;  
if ( a <= 2 )  
{  
    printf("Gut");  
    n = 'G';  
}
```

Verzweigungsanweisungen (4)

- bedingte Anweisung
- **Fallunterscheidung**
- Auswahlanweisung

Bei der bedingten Anweisung wird lediglich eine Anweisung angegeben, die bei Erfülltsein ausgeführt wird.

Bei der **Fallunterscheidung** wird eine weitere Anweisung formuliert, die bei Nichterfülltsein ausgeführt wird.

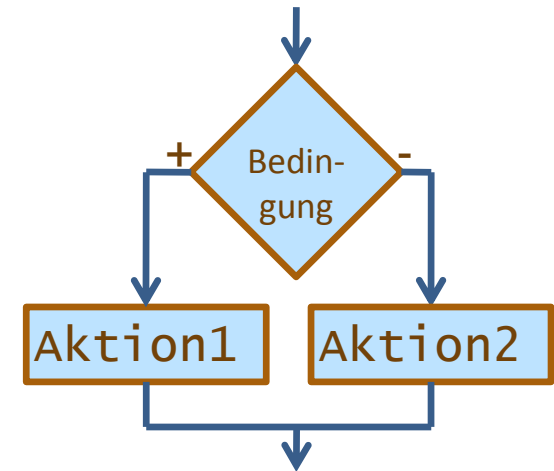
C

- Schlüsselwörter: **if** und **else**
- Klammerung des Bedingunsteils ist verpflichtend

falls Bedingung erfüllt,
so führe Aktion1 aus,
anderenfalls führe
Aktion2 aus

```
if (<boolescherAusdruck>)  
    <Anweisung1>  
else  
    <Anweisung2>
```

Anweisung bezeichnet hier (wie auch im Folgenden)
sowohl einfache Anweisungen als auch Anweisungsblöcke.



Verzweigungsanweisungen (5)

- bedingte Anweisung
- **Fallunterscheidung**
- Auswahlanweisung

Beispiele

- ohne Blockklammern

```
int x;  
int y;  
int quo;  
scanf("%d", &x);  
scanf("%d", &y);  
if (y>0)  
    quo = x/y;  
else  
    printf("Divisor soll größer als null sein.\n");
```

- mit Blockklammerung

```
if (y>0)  
{  
    quo = x/y;  
}  
else  
    printf("Divisor soll größer als null sein.\n");
```

Verzweigungsanweisungen (6)

- bedingte Anweisung
- **Fallunterscheidung**
- Auswahlanweisung

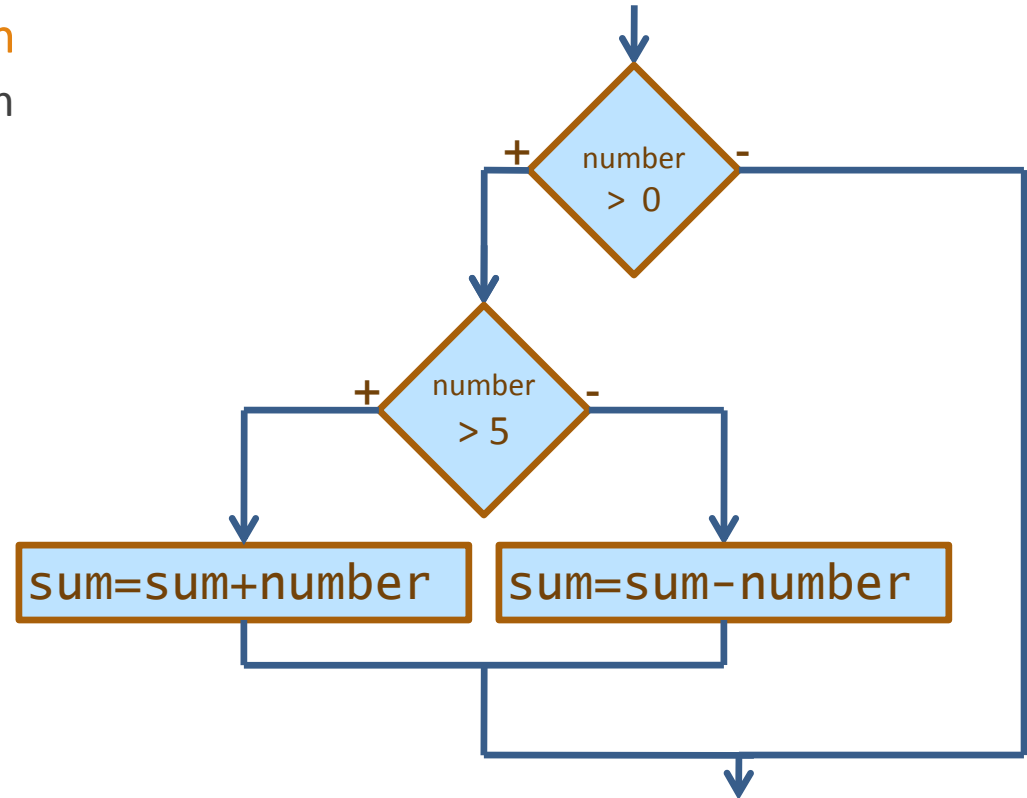
Geschachtelte bedingte Anweisungen

- beliebige Schachtelungen sind möglich

Beispiel:

- im Quellcode

```
...  
if (number > 0)  
{  
    if (number > 5)  
        sum = sum + number;  
    else  
        sum = sum - number;  
}
```



Verzweigungsanweisungen (7)

- bedingte Anweisung
- **Fallunterscheidung**
- Auswahlanweisung

Geschachtelte bedingte Anweisungen

- im Beispiel ist dank Klammerung klar, welche Zeilen wie zusammengehören

```
if (number > 0)
{
    if (number > 5)
        sum = sum + number;
    else
        sum = sum - number;
}
```

Frage: Wie ist das folgende Beispiel zu verstehen?

```
if (number > 0)
if (number > 5)
sum = sum + number;
else
sum = sum - number;
```

?

Regel: Ein `else` wird immer mit dem letzten freien `if` verbunden, für das es noch keinen `else`-Teil gibt!

- Empfehlung: Zur Sicherheit lieber explizite Klammern `{...}` setzen.

```
if (number > 0)
    if (number > 5)
        sum = sum + number;
    else
        sum = sum - number;
```

```
if (number > 0)
    if (number > 5)
        sum = sum + number;
else
    sum = sum - number;
```


Verzweigungsanweisungen (8)

- bedingte Anweisung
- **Fallunterscheidung**
- Auswahlanweisung

Mehrfachauswahl mittels Fallunterscheidung

- Eine kaskadierte **if-else**-Anweisung kann benutzt werden, um eine Art der Mehrfachauswahl zu implementieren.

Beispiel:

- Pseudo-Code-artig

```
if (ausdruck1)
    anweisung1;
else if (ausdruck2)
    anweisung2;
else if (ausdruck3)
    anweisung3;
...
else if (ausdruck_m)
    anweisung_m;
else
    anweisung_n;
```

mit Klammerung (nur bis 4)

```
if (ausdruck1)
{
    anweisung1;
}
else
{
    if (ausdruck2)
    {
        anweisung2;
    }
    else
    {
        if (ausdruck3)
        {
            anweisung3;
        }
        else
        {
            anweisung4;
        }
    } //end of else2
} //end of else1
```

Frage: Geht das auch eleganter?

Sonderfall: Der Fragezeichen-Operator

C besitzt einen trinären Operator (der also drei Argumente erwartet)

- der **Fragezeichen-Operator** (engl. Question Mark): `? :`

Dieser arbeitet wie eine `if`-Anweisung, die zusätzlich einen Wert zurückgibt!

```
int i = 10;
int j;
j = (i == 10) ? 20 : 5; /* note 3 args */
/* "(i == 10) ? 20 : 5" means:
   * "If i equals 10 then 20 else 5." */
```

Benutzung:

- gelegentlich
- sehr platzsparend

Verzweigungsanweisungen (9)

- bedingte Anweisung
- Fallunterscheidung
- **Auswahlanweisung**

Auswahlanweisungen erlauben es, in Abhängigkeit von dem Wert eines Ausdrucks direkt in einen von endlich vielen Fällen zu verzweigen.

- C-Umsetzung: **switch**-Anweisung

Pseudo-Code-artige Notation:

Bemerkungen:

- Wert des Ausdrucks `<ausdruck>` must ganzzahlig sein
- Ganzzahlkonstanten müssen paarweise verschieden sein
- der **default**-Fall ist optional
 - wird ausgeführt, wenn der Ausdruck mit keiner der Ganzzahlkonstanten übereinstimmt
- leere Anweisungsfolgen sind möglich
- **break**-Anweisung innerhalb oder am Ende einer Anweisungsfolge bewirkt das sofortige Verlassen der **switch**-Anweisung

```
switch (<ausdruck>)  
{  
    case <ganzzahlkonstante1>:  
        <anweisungsfolge1>  
    case <ganzzahlkonstante2>:  
        <anweisungsfolge2>  
    ...  
    case <ganzzahlkonstanteN>:  
        <anweisungsfolgeN>  
    default:  
        <anweisungsfolgeN+1>  
}
```

Verzweigungsanweisungen (10)

- bedingte Anweisung
- Fallunterscheidung
- **Auswahanweisung**

Auswahanweisungen erlauben es, in Abhängigkeit von dem Wert eines Ausdrucks direkt in einen von endlich vielen Fällen zu verzweigen.

- C-Umsetzung: **switch**-Anweisung

Beispiel →

Bemerkungen:

- **break** erzwingt Verzweigung zum Ende der Auswahanweisung
- Jede Anweisungsfolge kann als Block mit geschweiften Klammern implementiert sein.
- Achtung: Fehlt eine **break**-Anweisung, so wird automatisch in den folgenden Fall hineingelaufen!

```
int mark;  
scanf("%d", &mark );  
switch (mark)  
{  
    case 1:  
        printf("Sehr gut");  
        break;  
    case 2: printf("Gut"); break;  
    case 3: printf("Befriedigend"); break;  
    case 4:  
    {  
        printf("Ausreichend");  
        break;  
    }  
    case 5: printf("Genügend"); break;  
    case 6: printf("Ungenügend"); break;  
    default: printf("Keine Zensur");  
}
```

Beispiele zu `if` (1)

Notwendig: Überprüfung von Variablen und entsprechende Verzweigung

```
int a = 10;
if (a < 20)
{
    printf("less than 20\n");
}
else
{
    printf("not less than 20\n");
}
```

Test: `0` bedeutet “false”, alles andere ist “true”:

```
if (1) /* true, likewise 2, 3, ... */
{
    printf("less than 20\n");
} else
{
    printf("not less than 20\n");
}
```

Beispiele zu `if` (2)

Frage: Wo ist der Fehler?

```
int a = 0;
if (a = 10)
{
    printf("a equals 10\n");
}
else
{
    printf("a doesn't equal 10\n");
}
```

Hier sollte wohl Folgendes stehen:

```
int a = 0;
if (a == 10) /* not always true */
{
    printf("a equals 10\n");
}
else [snip]
```

Beispiele zu `if` (3)

Bekannt: Die **else**-Klausel ist optional.

else-if für mehrere Fälle:

```
int a = 0;
if (a == 10) {
    printf("a equals 10\n");
} else if (a < 10) {
    printf("a is less than 10\n");
} else {
    printf("a is greater than 10\n");
}
```

Besser: **switch**-Anweisung für dieses Muster:

<pre>void do_stuff(int i) { switch (i) { case 0: printf("zero\n"); break; case 1:</pre>	<pre> printf("one\n"); break; default: printf("...\n"); break; } }</pre>
---	---

Beispiele zu `if` (4)

Häufiger Fehler:

```
switch (i)
{
    case 0:  /* Start here if i == 0 */
        printf("zero\n");
        /* oops, forgot the break */
    case 1:  /* "fall through" from case 0 */
        printf("one\n");
}
```

Kein `break` beim `case 0`: gesetzt: So etwas ist nur sehr selten erwünscht ...

3. Strukturiertes Programmieren in C

1. Kontrollstrukturen
2. Einfache Anweisungen
3. Anweisungsblöcke
4. Verzweigungsanweisungen
5. **Funktionsaufrufe**
6. Schleifenanweisungen
7. Sprunganweisungen
8. Parameterübergabemechanismen



Anweisungen: Überblick

Überblick über die Arten von Anweisungen

- zur Realisierung von Kontrollstrukturen in C

Einfache Anweisungen

- Wertzuweisung
- Funktionsaufruf

Anweisungsblöcke

Schleifenanweisungen

- while-Schleife
- do-Schleife
- for-Anweisung

Verzweigungsanweisungen

- bedingte Anweisung
- Fallunterscheidung
- Auswahlentscheidung

Sprunganweisungen

- Abbruchanweisung
- Rückgabeeinweisungen

Einfache Anweisung: Funktionsaufruf (1)

Funktionen (Prozeduren) sind Folgen von Anweisungen, die benutzt werden, um Teilaufgaben wiederverwendbar zu formulieren. ➔ „Unterprogramme“

Funktionen (Prozeduren) erlauben es, von konkreten Anweisungen bzw. Anweisungssequenzen zu abstrahieren.

- Funktionen legen die **Parameter** einer zusammengesetzten Anweisung fest und geben ihr einen **Namen**.

Dies ermöglicht

- **Wiederverwendung** von Quellcode (Vermeidung von dupliziertem Quellcode)
 - denn eine Funktion kann von beliebigen Stellen im Programm aus aufgerufen werden
- Schnittstellenbildung und Information Hiding

Syntax einer Funktionsdefinition:

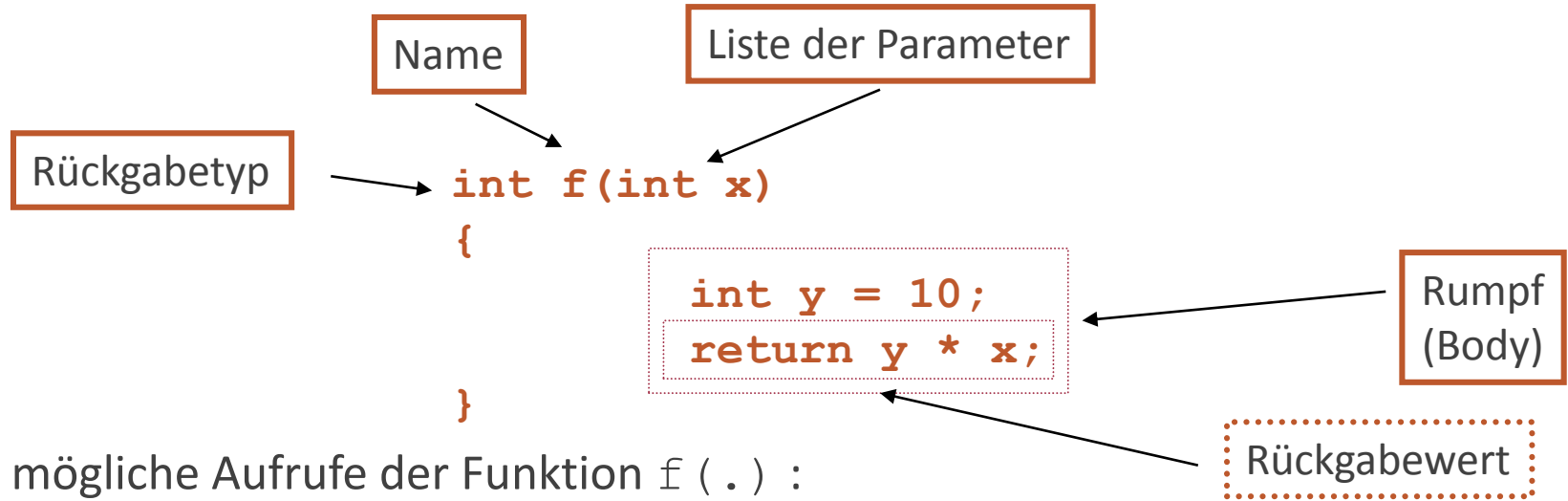
```
[Speicherkl.] Rückgabetyt NAME ( [Parameterliste u.-definition] )  
Block
```

Syntax eines Funktionsaufrufes:

```
NAME ( [Parameterliste] );
```

Einfache Anweisung: Funktionsaufruf (2)

Funktionen nehmen Parameter entgegen und geben Werte zurück



mögliche Aufrufe der Funktion `f (.)` :

```
/* in another function... */
```

```
int res;
```

```
int i = 10;
```

```
res = f(10);
```

```
res = f(5 + 5);
```

```
res = f(i);
```

```
res = f(i*5 + i/2);
```

```
int g( int a, int b)
```

```
{
```

```
...
```

```
}
```

Beispiel für eine Funktion, die zwei Parameter erwartet:

Einfache Anweisung: Funktionsaufruf (3)

Bemerkungen:

- Funktionen (können) definieren
 - Aufrufparameter: mehrere
 - Rückgabewerte: einer
- Aufrufparameterliste:
 - kommasepariert
 - je Parameter eine Typangabe

Beispiel: Absolutbetrag

```
int x;  
int a;  
printf("Eingabe von x: ");  
scanf("%d", &x);  
int result;  
if (x<0)  
    result = -x;  
else  
    result = x;  
a = result;  
printf("Absolutwert: %d", a);
```

```
int x;  
int a;  
printf("Eingabe von x: ");  
scanf("%d", &x);  
a = AbsWert( x );  
printf("Absolutwert: %d", a);
```

```
int AbsWert( int v )  
{  
    if (v<0)  
        return -v;  
    else  
        return v;  
}
```

Einfache Anweisung: Funktionsaufruf (4)

Die **Definition einer Funktion** legt die Aufgaben der Funktion durch eine Anweisungsfolge fest.

- Beispiel: vorige Folie → Bildung des Absolutwertes einer Ganzzahl

Durch die Definition einer Funktion wird diese **nicht ausgeführt**, sondern sie wird dem **Compiler bekanntgemacht**.

Beim **Funktionsaufruf** (im Beispiel: `AbsWert (x) ;`) erfolgt ein Sprung an die Definitionsstelle.

- Sodann werden sämtliche Anweisungen der Funktion abgearbeitet.
- Danach wird das Programm mit dem Befehl fortgesetzt, der dem Aufruf der Funktion folgt.
 - im Beispiel: `printf (...)`
- Funktionen ermöglichen den rekursiven Aufruf von Funktionen.

Funktionen (1)

Die **return**-Anweisung

- dient der sofortigen Beendigung einer Funktion
- dient der Rückgabe eines Wertes an den aufrufenden Programmteil
 - sofern es sich um eine Funktion mit Rückgabetyt ungleich `void` handelt
- kann damit zur Ergebnissrückgabe genutzt werden, d.h. Ergebnisse von Berechnungen innerhalb der Funktion werden dem aufrufenden Programmteil mitgeteilt

Syntax: **return** <ausdruck>;

- Der Datentyp des Rückgabewertes (Typ von <ausdruck>) muss mit dem Rückgabetyt der Funktion kompatibel sein.

Beispiel:

```
int AbsWert( int v )
{
    int abs;
    if (v<0)
        abs = -v;
    else
        abs = v;
    return abs;
}
```

Funktionen (2)

Nicht alle Funktionen geben Werte zurück

```
void print_number(int i)
{
    printf("number is: %d\n", i);
}
```

Rückgabewert ist **void** (d.h. nichts wird zurückzugeben)

```
void print_number(int i)
{
    printf("number is: %d\n", i);
    return; /* unnecessary */
}
```

return-Anweisung ist nicht nötig (außer mitten in der Funktion)

Aufruf der obigen Funktion:

```
/* In another function... */
int i = 10;
print_number(20);
print_number(i);
```


Funktionen (3)

Nicht alle Funktionen benötigen Parameter!

- Die parameterlose Funktion ist die einfachste Form eines Unterprogramms in C.
- stark begrenzter Einsatzbereich

```
int five(void)
{
    return 5;
}
```

Aufruf von Funktionen ohne Parameter:

```
int value;
value = five();
```

Nach Ausführung: **value** hat den Wert 5.

Man beachte die **Klammern** () beim Funktionsaufruf!

Funktionen (4)

In C werden dem Nutzer bzw. Entwickler viele nützliche „**Standardfunktionen**“ bereitgestellt.

- Eine Funktion `AbsWert(int x)` muss man sich nicht selbst implementieren.

Wichtige **Standardfunktionen** und wo sie deklariert sind (→ `#include` erforderlich):

- `stdio.h` → Ein- und Ausgabe (`printf`, `scanf` etc.)
- `string.h` → Arbeit mit Zeichenketten und Speicherbereichen: kopieren, vergleichen, durchsuchen etc.
- `math.h` → Sinus, Cosinus, Potenzfunktionen, Wurzel, Logarithmus, etc.
- `stdlib.h` → diverse Hilfsfunktionen (z.B. Umwandlung einer Zeichenkette in eine Zahl, Zufallszahlengenerator)
- `malloc.h` → Hilfsfunktionen zur Speicherverwaltung (`malloc`, `free`, etc.)

Speicherort dieser Include-Dateien:

- typischerweise unter `/usr/include`
- kann von System zu System variieren

Funktionen (5)

Funktionsdeklarationen (Prototypen)

- Zu jeder (später) definierten Funktion existiert die „erste Zeile;“ noch einmal in diesem Prototypenblock.
- Also: Auflistung alle Funktionen im Stil
`Rückgabetyyp Name(Parameterliste);`

Warum ist dies erforderlich?

- Bei zwei Funktionen A und B kann es sein, dass unter gewissen Bedingungen aus A die Funktion aufgerufen wird.
- Außerdem kann aus der Funktion B die Funktion A aufgerufen werden.
- Beim Analysieren des Quelltexts (von oben nach unten) wird der Compiler,
 - wenn er bei A angelangt ist, die Kenntnis von B voraussetzen
 - wenn er bei B angelangt ist, die Kenntnis von A voraussetzen.
- Mit einer Vorab-Deklaration aller implementierten Funktionen im Funktionsdeklarationsblock kann dieses Dilemma aufgelöst werden.



Erinnerung: Vom Quellcode zum Programm (1)

Kompilieren und Linken

- Das Kompilieren und Linken eines Programms erfolgt grundsätzlich mit dem Befehl

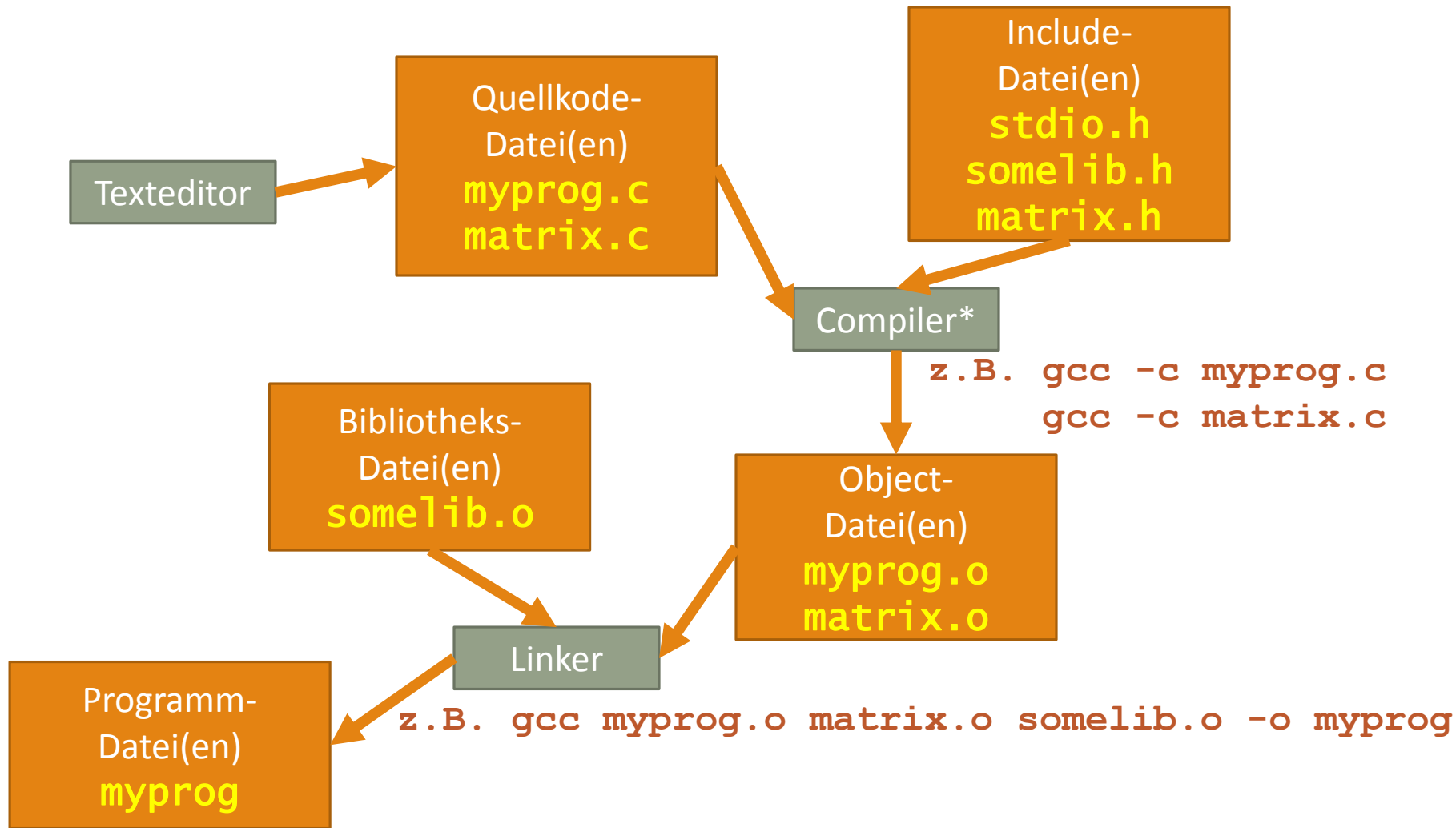
```
gcc [-parameter [...]] dateiname [...]
```

- Bsp.: Der Parameter `-o name` (output) bewirkt, dass das ausführbare Programm einen definierten Namen bekommt.
- Frage: Was ist, wenn ich meinen Quellcode über mehrere c-Dateien verteile? Wenn ich beispielsweise einige Funktionen in eine separate c-Datei „auslagere“?
 - Für jede c-Datei wird eine o-Datei (Object File) erzeugt!
 - Mehrere o-Dateien können zu einem ausführbaren Programm verschmolzen werden!

Beispiele:

- übersetze und linke `hello.c` mit Debugger-Informationen (`g`) und Mathe-Bibliothek (`lm`)
gcc -g -lm hello.c -o hello
- übersetze die Datei `matrix.c` ohne zu linkern, d.h. ohne eine ausführbare Datei zu erstellen
 - Object-Datei `matrix.o` entsteht
gcc -c matrix.c
- linke zwei Object-Dateien zu einem ausführbaren Programm
gcc matrix.o myprog.o -o meinProgramm

Vom Quellcode zum Programm (2)



* inkl. Präprozessor-Anwendung

Lokale und globale Variablen (1)

Definition: Eine **globale Variable** wird außerhalb von Funktionsdefinitionen definiert. Sie existiert während der gesamten Ausführungszeit des Programms.

Definition: Eine **lokale Variable** wird innerhalb einer Funktion definiert und existiert daher auch nur, während die Funktion ausgeführt wird.

- Sie kann „wiedergeboren“ werden, wenn die Funktion mehrere Male aufgerufen wird.

Die Aussagen für lokale und globale Variablen gelten **analog** für lokale und globale Konstanten!

Beispiel:

```
#include <stdio.h>
int someVariable; //globale Var.
void someFunction( void )
{
    int otherVar1; //lokale Var.
    printf("Hello world!\n");
}
int main( void )
{
    int otherVar2; //lokale Var.
    someVariable = 42;
    someFunction();
}
```

Lokale und globale Variablen (2)

Fazit: Variablendeklarationen können **lokal** oder **global** sein

- lokal: innerhalb einer Funktion
- global: außerhalb von Funktionen
- global: von jeder Funktion aus zugreifbar

```
int x;           /* Global variable */
int y = 10;      /* Initialized global variable */

int foo(int z)
{
    int w;       /* local variable */
    x = 42;      /* assign to a global variable */
    w = 10;      /* assign to a local variable */
    return (x + y + z + w);
}
```

Bemerkung: Globale Variablen **sollten weitestgehend vermieden werden!**

- können von jeder Funktion geändert werden → Fehlersuche schwierig
- sind niemals unbedingt notwendig → aber praktisch (und daher verführerisch)

Lebensdauer, Gültigkeitsbereich und Sichtbarkeitsbereich von Variablen

Lebensdauer (Wann existiert eine Variable?)

- globale Variable: gesamte Laufzeit des Programms
- lokale Variable: Ausführungszeit der zugehörigen Funktion
 - lokale Variable existiert stets neu / wieder, wenn die zugehörige Funktion mehrfach aufgerufen wird

Gültigkeitsbereich (In welchen Programmzeilen ist der Name einer Variable bekannt?)

- globale Variable: in allen folgenden Funktionen bekannt
- lokale Variable: nur in ihrer Funktion bekannt

Sichtbarkeitsbereich (In welchen Programmzeilen kann ich auf eine Variable zugreifen?)

- Eine Variable ist sichtbar, wenn sie gültig ist und von keiner lokalen Variablen mit gleichem Namen überdeckt wird.
- Achtung: Solche Überdeckungen sind durchaus erlaubt!

3. Strukturiertes Programmieren in C

1. Kontrollstrukturen
2. Einfache Anweisungen
3. Anweisungsblöcke
4. Verzweigungsanweisungen
5. Funktionsaufrufe
6. **Schleifenanweisungen**
7. Sprunganweisungen
8. Parameterübergabemechanismen



Anweisungen: Überblick

Überblick über die Arten von Anweisungen

- zur Realisierung von Kontrollstrukturen in C

Einfache Anweisungen

- Wertzuweisung
- Funktionsaufruf

Anweisungsblöcke

Schleifenanweisungen

- while-Schleife
- do-Schleife
- for-Anweisung

Wiederholung

Verzweigungsanweisungen

- bedingte Anweisung
- Fallunterscheidung
- Auswahlentscheidung

Sprunganweisungen

- Abbruchanweisung
- Rückgabeeinweisungen

Schleifenanweisungen (1)

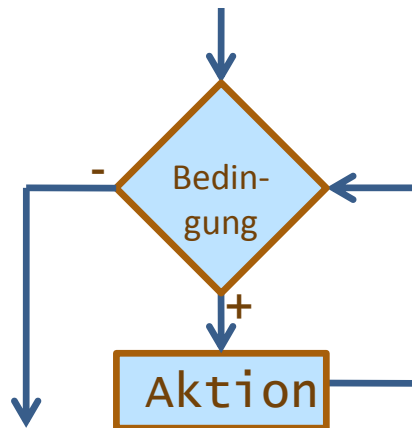
• while-Schleife

- do-Schleife
- for-Schleife

Die **abweisende Schleife** (while-Schleife) enthält eine Anweisung, die solange ausgeführt wird, wie die Schleifenbedingung erfüllt ist.

- Beachte: Die Bedingung wird **vor** jeder Ausführung der Anweisung ausgewertet, daher die Bezeichnung „abweisende“ Schleife.
- Sonderfall: Bedingung ist immer erfüllt
 - Endlosschleife
 - Ausbruch nur mit Sprunganweisung möglich
- Schlüsselwort in C: **while**

solange wie
Bedingung erfüllt,
führe Aktion aus



```
while (<boolescherAusdruck>)  
    <Anweisung>
```

Schleifenanweisungen (2)

- **while-Schleife**

- do-Schleife
- for-Schleife

Beispiel: 10x Hallo Welt!

```
int number = 10;
while ( number > 0 )
{
    printf("Hallo Welt!");
    number--;
}
```

Beispiel: Berechnung der Fakultät

```
int number;
scanf( "%d", &number );

int factorial = 1;
while ( number > 0 )
{
    factorial = factorial * number;
    number--;
}
printf("Fakultät ist %d", factorial);
```

Schleifenanweisungen (3)

- **while-Schleife**

- do-Schleife
- for-Schleife

Nützlich, wenn Anzahl der Iterationen nicht im Voraus bekannt ist.

- Aber: Gefahr. So können auch Endlosschleifen entstehen!
- Beispiel:

```
int a;
while (1)                                /* or: for (;;) */
{
    scanf("%d ", &a);
    printf("a = %d\n", a);
    if (a >= 0)
    {
        break;                          /* get out of loop */
    }
}
```

Schleifenanweisungen (4)

- while-Schleife
- do-Schleife
- for-Schleife

Neben der abweisenden Schleifenanweisung sind die nicht-abweisende Schleife (do-Schleife) sowie die Zählschleife (for-Schleife) gebräuchlich.

Frage: Warum noch weitere Schleifenanweisungen?

Antwort:

- Die do- und for-Schleife sind häufig verwendete Kontrollstrukturen, so dass die Einführung eines eigenen programmiersprachlichen Konstrukts zur Verbesserung der Lesbarkeit begründbar ist.
- Formal käme man ohne diese weiteren Schleifentypen aus.
- Jede for-/do-Schleife kann in eine while-Schleife mit gleicher Funktionalität umformuliert werden.

Schleifenanweisungen (5)

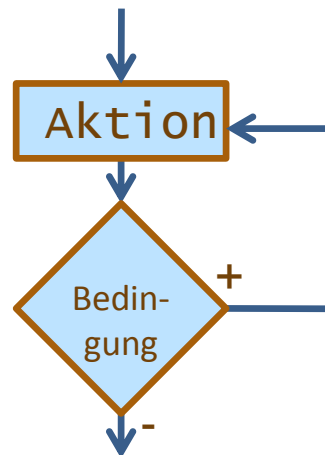
- while-Schleife
- **do-Schleife**
- for-Schleife

Die **nicht-abweisende Schleife** (do-Schleife) enthält eine Anweisung, die 1x ausgeführt und dann solange wiederholt wird, wie die Schleifenbedingung erfüllt ist.

- Beachte: Die Bedingung wird **nach** jeder Ausführung der Anweisung ausgewertet, daher die Bezeichnung „nicht-abweisende“ Schleife.
- Sonderfall: Bedingung ist immer erfüllt
 - Endlosschleife
 - Ausbruch mit Sprunganweisung möglich
- Schlüsselwörter in C: **do** und **while**

führe Aktion aus,
solange wie
Bedingung erfüllt

```
do  
    <Anweisung>  
while (<boolescherAusdruck>);
```



Zählschleife (1)

- while-Schleife
- do-Schleife
- **for-Schleife**

Die **Zählschleife** (for-Schleife) besteht aus

- einem Zähler,
- einer Zählerbedingung,
- einem Zählermodifikator und
- einer Anweisung (bzw. Anweisungsblock), die solange wiederholt wird, wie die Zählerbedingung erfüllt ist.

Nach jeder Ausführung der Anweisung verändert der Zählermodifikator den aktuellen Stand des Zählers.

- Zählerbedingung wird **vor** jeder Ausführung der Anweisung ausgewertet.
→ Zählschleife ist **Spezialfall** der abweisenden Schleife.
- Sonderfall: Zählerbedingung immer erfüllt → Endlosschleife

Zählschleife (2)

- while-Schleife
- do-Schleife
- **for-Schleife**

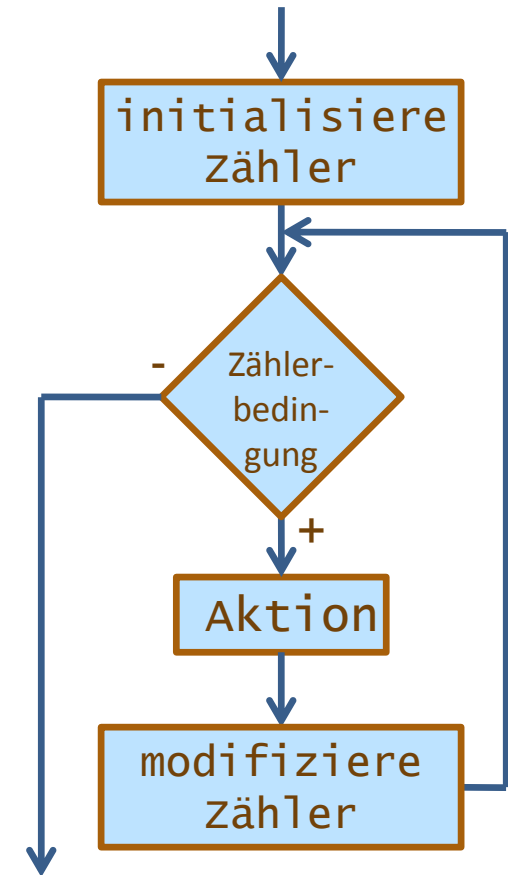
initialisiere Zählvariable
solange wie Zählerbedingung
erfüllt,
führe Aktion aus
und modifiziere Zähler

dürfen beliebige Ausdrücke sein

```
for ( <cInit>; <cCond>; <cModi> )  
    <Anweisung>
```

- cInit: Zähler initialisieren
- cCond: Zählerbedingung
- cModi: Zählermodifikation
- Beispiel in C:

```
int c;  
for ( c=0; c<10; c++ )  
    printf("Zählerstand: %d\n", c);
```



Zählschleife (3)

- while-Schleife
- do-Schleife
- **for-Schleife**

Verallgemeinerung der Zählschleife:

- **Leere Ausdrücke sind erlaubt!**
- Anweisung wird ausgeführt, **solange** `ausdr2` wahr ist → Abbruchkriterium
- `ausdr1` wird **initial 1x** ausgeführt → Initialisierungsausdruck
- `ausdr3` wird **nach** jedem Schleifendurchlauf ausgeführt
- Realisierung als Zählschleife ist die übliche (aber nicht einzige) Ausprägung einer for-Schleife

```
for ( <ausdr1>; <ausdr2>; <ausdr3> )  
    <Anweisung>
```

Beispiel: Zinseszinsberechnung

Anforderungsspezifikation:

- Ein Bankkunde legt einen festen Geldbetrag für einen vorgegebenen Zeitraum fest. Zinsen werden dem Guthaben jährlich jeweils am Ende des Jahres gutgeschrieben und dann mitverzinst (Zinseszins). Die Laufzeit der Geldanlage ist frei wählbar, soll aber eine ganzzahlige Anzahl Jahre umfassen.
- Wie groß ist das Guthaben nach Ablauf eines jeden Jahres innerhalb der Laufzeit?

Zählschleife (4)

- while-Schleife
- do-Schleife
- **for-Schleife**

Beispiel: Zinseszinsberechnung

```
double startkapital;  
printf("Bitte Startkapital eingeben: ");  
scanf("%lf", &startkapital);  
double zinssatz; //z.B. 2.5%  
printf("Bitte Zinssatz eingeben: ");  
scanf("%lf", &zinssatz);  
int laufzeit;  
printf("Bitte Laufzeit eingeben: ");  
scanf("%d", &laufzeit);  
int jahr;  
double kapital = startkapital;  
for ( jahr=1; jahr<=laufzeit; jahr++ )  
{  
    double zinsen = kapital * (zinssatz/100.0);  
    kapital = kapital + zinsen;  
    printf("Kapital am Ende vom %d-ten Jahr: %lf\n",  
           jahr, kapital);  
}
```