

There exist means of expressing the conditions under which these various processes are required to be called into play. It is not even necessary that two courses only should be possible. Any number of courses may be possible at the same time; and the choice of each may depend on any number of conditions.

Charles Babbage (1864)

Kontrollfluss-Anweisungen: Verzweigungen (branches)

Bedingte Übergänge: `if`, `switch`
Unbedingte Übergänge: `break`, `continue`

Bedingte Anweisungen: if – else

- Die einfache if-Anweisung hat die Form

```
if (condition) statement1
```

- Die allgemeine if-else-Anweisung hat die Form

```
if (condition) statement1 else statement2
```

- Dabei ist `condition` ein Boolescher Ausdruck
 - ◆ Falls `condition` zu true evaluiert wird `statement1` ausgeführt
 - ◆ Im Fall der if-else Anweisung wird sonst `statement2` ausgeführt
- Jedes `statement` ist eine Anweisung
 - ◆ Also evtl. ein Block oder wieder eine if-Anweisung, oder die leere Anweisung, etc.

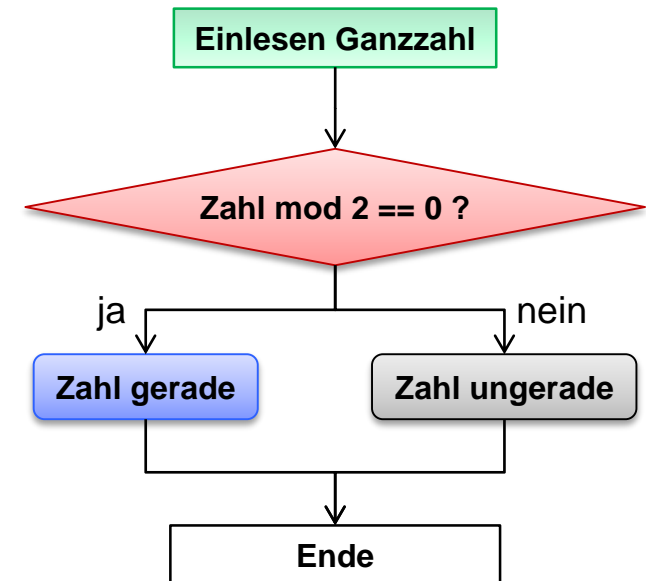
Bedingte Anweisungen: if – else

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

```
int main() {
```

```
    int zahl;  
  
    printf("Zahl eingeben: ");  
    scanf("%i",&zahl);
```

```
    if (zahl%2==0)  
        printf("%i ist gerade.\n",zahl);  
    else  
        printf("%i ist ungerade.\n",zahl);  
  
    return 0;  
}
```



Bedingte Anweisungen: Beispiele

Beispiel 6.8.6. Die Anweisung

```
if( a >= 0 )
    res = a;
else
    res = -a;
```

belegt die Variable `res` durch den Absolutwert der Variablen `a`.

Beispiel 6.8.7. Die Anweisung

```
if( a <= 0 ) {
    res = a;
}
else
    if( a+b < 0 ) {
        res = a+b;
    }
    else {
        res = b;
    }
```

belegt die Variable `res` durch:

`res = a`, falls `a` negativ oder gleich 0 ist,

`res = a+b`, falls `a` größer als Null und `a+b` negativ ist und

`res = b`, falls `a` größer als Null und `a+b` positiv oder gleich Null ist.

Bedingte Anweisungen – switch-case-default

- Eine weitere Möglichkeit einer bedingten Anweisung ist die **Fallunterscheidung** (**switch – case – default**)

Switch (c)

```
{  
    case constant_1: {statement_1; break;}  
    case constant_2: {statement_2; break;}  
        ...  
    case constant_n: {statement_n; break;}  
    default: {statementd;}  
}
```

- Nur **Konstanten** erlaubt

und zwar vom Typ
char, byte, short, int,
Character, Byte, Short, Integer
oder einem Aufzählungstyp

- Die **break**-Anweisung am Ende jeder **case**-Zeile bewirkt das **sofortige Verlassen** der **switch**-Anweisung.
 - ◆ Falls das **break** fehlt, wird der nachfolgende **case**-Fall mit ausgeführt, ohne Test ob seine Bedingung zutrifft!

Ein Beispiel:

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

```
int main() {
```

```
    int color = 1;
    printf("Please choose a color(1: red,2: green,3: blue):\n");
    scanf("%i", &color);
```

```
    switch (color) {
```

```
        case 1:
```

```
            printf("you chose red color\n");
            break;
```

```
        case 2:
```

```
            printf("you chose green color\n");
            break;
```

```
        case 3:
```

```
            printf("you chose blue color\n");
            break;
```

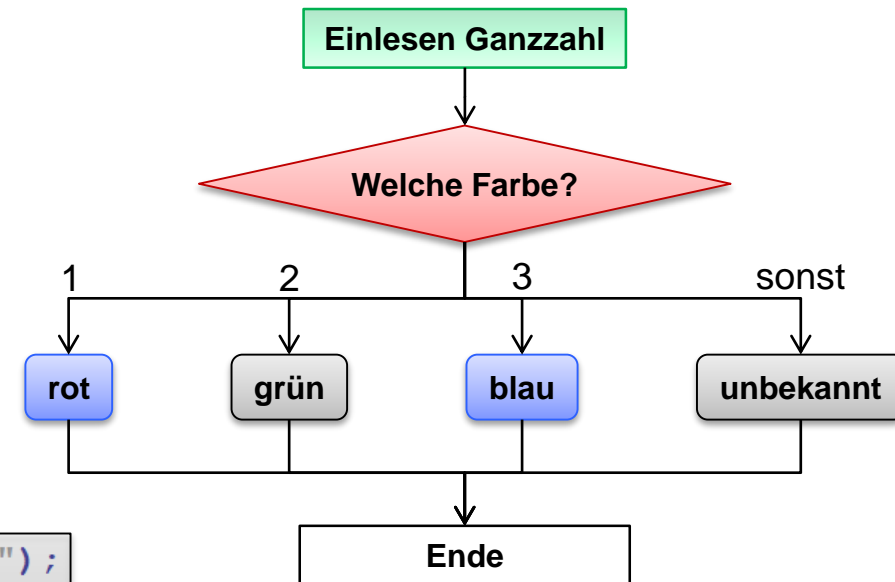
```
        default:
```

```
            printf("you did not choose any color\n");
```

```
    }
```

```
    return 0;
```

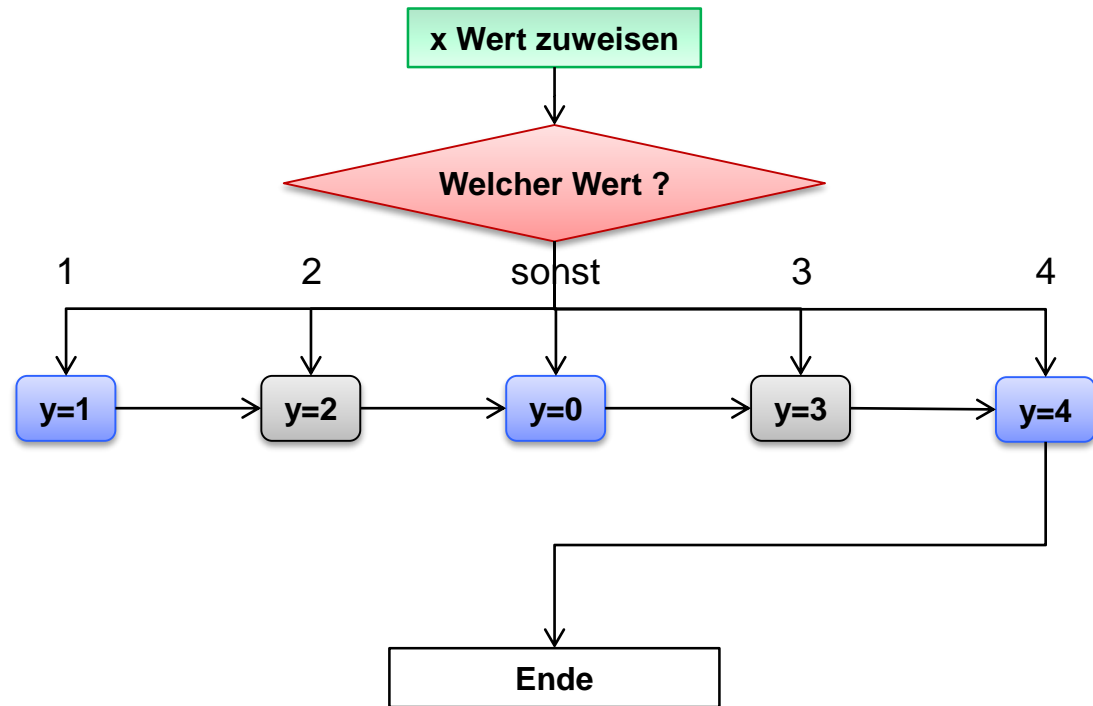
```
}
```



Bedingte Anweisungen – switch-case-default

- Beispiel: In diesem Fall setzt jeder **case** mit dem nächsten fort, da jegliche **break**-Anweisungen fehlen:

```
int x, y;  
x = 1234;  
switch(x)  
{  
    case 1:  
        y = 1;  
    case 2:  
        y = 2;  
    default:  
        y = 0;  
    case 3:  
        y = 3;  
    case 4:  
        y = 4;  
}
```



Bedingte Anweisungen – switch-case-default

Beispiel mit weggelassenem ‚break‘

Beispiel 6.8.9. Durch das Weglassen von `{anweisung; break;}` können mehrere case-Fälle zusammengefaßt werden: Das Programmstück

```
switch(c) {  
    case 0: case 1: case 2: case 3: { res = 1; break; }  
    case 4: case 5: case 8: case 9: { res = 2; break; }  
    case 6: case 7: { res = 3; break; }  
    default: { res = 0; }  
}
```

liefert:

res = 1, falls c den Wert 0, 1, 2 oder 3 hat,
res = 2, falls c den Wert 4, 5, 8 oder 9 hat,
res = 3, falls c den Wert 6 oder 7 hat und
res = 0 in allen anderen Fällen.

Schleifen-Anweisungen – while und do-while

Das Konstrukt

while (*condition*) *statement*;
führt *statement* aus, solange der
Ausdruck in *condition* zu **true** evaluiert.

Oftmals wird es sich bei *statement* um
einen Block handeln.

Die Anweisung

do *statement* **while** (*condition*);
führt *statement* aus und prüft danach
anhand von *condition*, ob der Schleifen-
durchgang wiederholt werden soll.

Bei do-Anweisungen ist
statement fast immer ein Block.

Die Anweisung

do *statement* **while** (!*condition*);
entspricht dem Konstrukt
do *statement* **until** (*condition*);
in anderen Sprachen.

repeat ... until
in Pascal

while-Schleife: Beispiel

Beispiel 6.8.11. Die folgende `while`-Schleife wird so lange ausgeführt, bis die Bedingung $(i < 10)$ nicht mehr erfüllt ist. Dieses Programmstück summiert in `res` die Zahlen von 1 bis 9 auf. Eine passende Schleifeninvariante ist $res = \sum_{j=1}^{i-1} j$. Zu Beginn gilt $0 = \sum_{j=1}^0 j$ und zum Schluß gilt $res = \sum_{j=1}^{10-1} j$.

```
{ int i = 1;
  int res = 0;
  while ( i < 10 )
  {
    res = res + i;
    i++;
  }
}
```

D.h. nach dem letzten Schleifendurchlauf

D.h. vor dem ersten Schleifendurchlauf

Die Schleife kann natürlich verallgemeinert werden, indem man den festen Wert 10 durch eine Variable x ersetzt; dann gilt zum Schluß $res = \sum_{j=1}^{x-1} j$. ❖

for-Schleife (1)

- Die Anweisung

```
for (init; condition; increment) statement
```

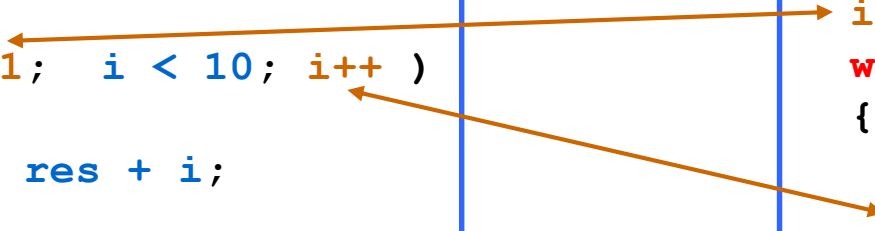
entspricht

```
{init; while (condition) {statement; increment ;} }
```

- Spezielle Syntax hierfür ist sinnvoll, da solche Schleifen häufig verwendet werden

```
{  
    int res = 0;  
    int i;  
    for (i=1; i < 10; i++ )  
    {  
        res = res + i;  
    }  
}
```

```
{  
    int res = 0;  
    int i = 1;  
    while ( i < 10 )  
    {  
        res = res + i;  
        i++;  
    }  
}
```



for-Schleife (2)

- Der **increment**-Teil kann aus einer **durch Kommata getrennten Liste von Ausdrücken** bestehen, die von links nach rechts evaluiert werden
 - ◆ Dadurch besteht oft die Alternative, Code entweder im Schleifenrumpf oder im Kopf unterzubringen
 - ◆ Der Kopf sollte immer den Code enthalten, der die Kontrolle über die Schleifendurchgänge behält; der Rumpf sollte die eigentlichen Arbeitsanweisungen enthalten
- Jeder der drei Teile in einer for-Anweisung kann auch leer sein und zum Beispiel durch ein entsprechendes Konstrukt im Schleifenkörper ersetzt werden

for-Schleife (3)

- Beispiele für „guten“ und „schlechten“ Programmierstil

1. In der folgenden Schleife wird eindeutig klargemacht, daß `i` die Laufvariable ist und nur innerhalb der Schleife Bedeutung hat.

```
{ int res=0;
  for (int i=1; i<10; i++)
    res += i;
}
```

Guter Stil 😊

2. In der folgenden Schleife wird die „Arbeit“ `res+=i` im Kopf der Schleife verrichtet. Dies ist ebenso *schlechter Stil*, wie die Schleifenkontrolle im Rumpf zu erledigen.

```
{ int res=0;
  for (int i=1; i<10; res+=i, i++)
    ;
}
```

Syntaktisch korrekt
Schlechter Stil ☹️

for-Schleifen-Beispiel: Maximum eines Array

- Die folgende Schleife berechnet das Maximum der Werte eines Array **a** der Länge **len**

```
float max = a[0];  
int i;  
for (i=1; i < len; i++)  
    if (a[i] > max)  
        max = a[i];
```

Angenommen **len** und **a** wurden passend initialisiert, z.B. als

```
int len=5;  
float[] a = {1.0, 3.7, 47.11, 0.815, 42};
```

for-Schleifen-Beispiel: Drucken eines Bitmusters

● Beispiel (Drucken eines Bitmusters)

- ◆ Wir betrachten eine Lösung für das Problem, das zu einer auf der Kommandozeile angegebenen Ganzzahl gehörende Bitmuster auszugeben.
- ◆ Wir drucken nacheinander die Bits 31 bis 0 einer Zahl **z**
- ◆ Zur leichteren Lesbarkeit sollen die Bytes jeweils durch einen Punkt separiert werden
- ◆ Wir verwenden eine Maske, in der nur das zu druckende Bit auf 1 gesetzt ist

for-Schleifen-Beispiel: Drucken eines Bitmusters

```
#include <stdio.h>

int main() {

    int z, i;
    scanf("%i", &z);

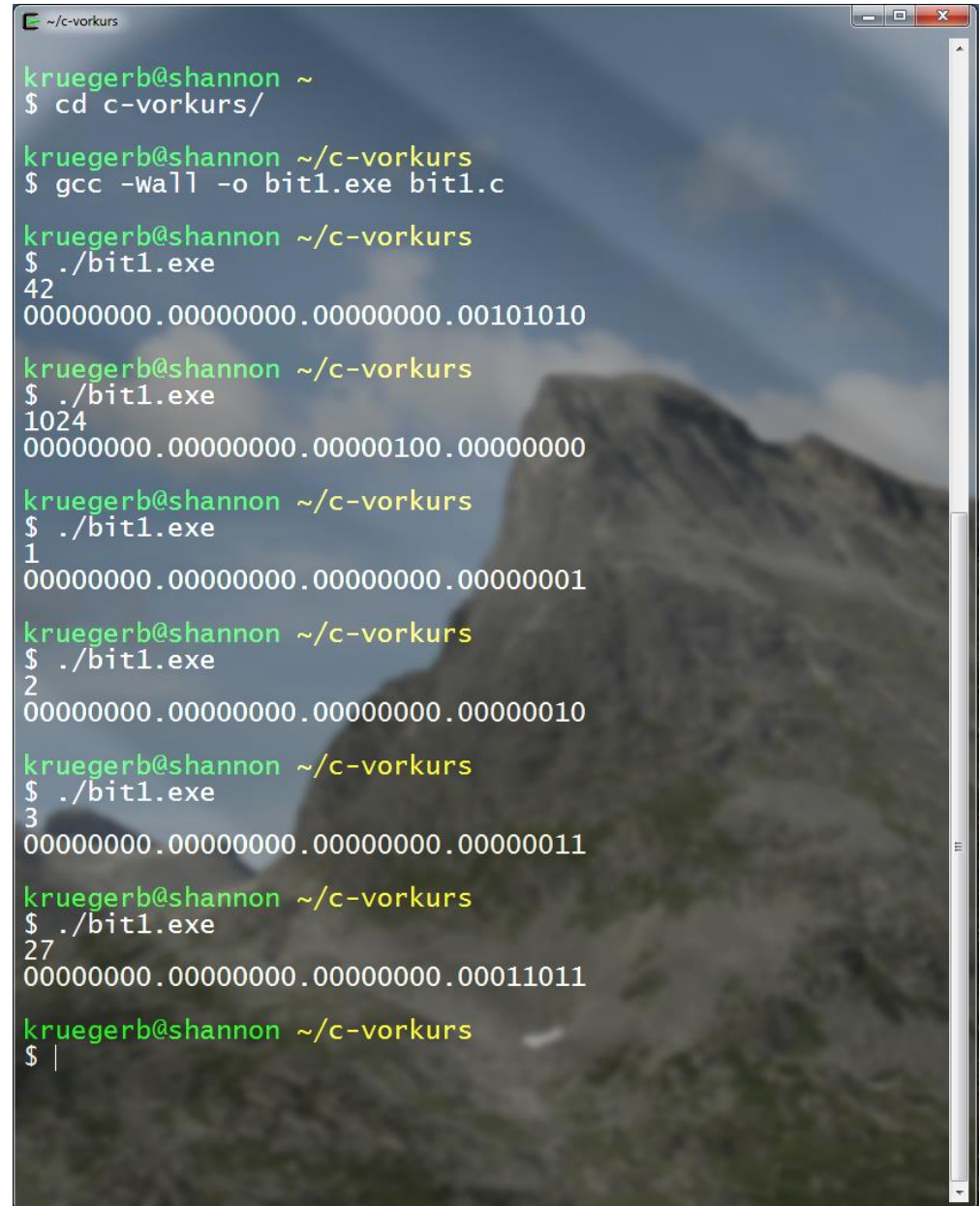
    unsigned int mask = 01 << 31;

    for (i=31; i>=0; i--) {
        if ( (z&mask) != 0)
            printf("1");
        else
            printf("0");

        if((i%8==0) && mask!=1)
            printf(".");
        mask >>= 1;
    }

    printf("\n");

    return 0;
}
```



```
kruegerb@shannon ~
$ cd c-vorkurs/

kruegerb@shannon ~/c-vorkurs
$ gcc -Wall -o bit1.exe bit1.c

kruegerb@shannon ~/c-vorkurs
$ ./bit1.exe
42
00000000.00000000.00000000.00101010

kruegerb@shannon ~/c-vorkurs
$ ./bit1.exe
1024
00000000.00000000.00000100.00000000

kruegerb@shannon ~/c-vorkurs
$ ./bit1.exe
1
00000000.00000000.00000000.00000001

kruegerb@shannon ~/c-vorkurs
$ ./bit1.exe
2
00000000.00000000.00000000.00000010

kruegerb@shannon ~/c-vorkurs
$ ./bit1.exe
3
00000000.00000000.00000000.00000011

kruegerb@shannon ~/c-vorkurs
$ ./bit1.exe
27
00000000.00000000.00000000.00011011

kruegerb@shannon ~/c-vorkurs
$ |
```


Break und continue

- **break;**
 - ◆ beendet die unmittelbar umgebende **switch-**, **while-**, **do-** oder **for-Anweisung**
- **continue;**
 - ◆ beendet den aktuellen Schleifendurchlauf, beginnt die Schleife erneut mit dem nächsten Iterationsschritt

Break und continue

- Beispiel: `#include <stdio.h>`

```
int main() {  
  
    int counter;  
    for(counter = 0; counter < 50; counter++){  
        if (counter == 20)  
            break;  
        if (counter % 3 == 0)  
            continue;  
  
        printf("%i, ", counter);  
  
    }  
    printf("schleife durchgelaufen.");  
  
    return 0;  
}
```

- Ausgabe:

1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, schleife durchgelaufen.

Labels, break und continue

- Jedes Statement kann markiert werden:
 - ◆ *label: statement*
 - ◆ Sinnvoll i.d.R. nur bei Blöcken, Schleifen
- Sprung „goto label“
- Beispiel:

```
goto skip_point;
printf("This part was skipped. \n");
skip_point:
    printf("hi there! \n");
//Only text "hi there!" is printed
```