# 10 Kontrollstrukturen

Die sequenzielle Programmausführung kann durch sogenannte „**Kontrollstrukturen**“ beeinflusst werden. Diese erlauben es, an bestimmten Stellen im Code an andere Stellen zu springen. Dadurch können in Abhängigkeit von der Bewertung von Ausdrücken Anweisungen übergangen oder ausgeführt werden.

## 10.1 Blöcke – Kontrollstruktur für die Sequenz

Der folgende Code zeigt einen **Block**:

```
{
  Anweisung 1
  Anweisung 2
  Anweisung 3
  ...
  Anweisung n
}
```

Die geschweiften Klammern `{` und `}` stellen die Blockbegrenzer dar. Die Anweisungen zwischen den Blockbegrenzern werden sequenziell abgearbeitet. Ein Block wird deshalb auch als Kontrollstruktur für die Sequenz bezeichnet.

Ein Block ist eine Sequenz von Anweisungen.

Ein Block zählt syntaktisch als eine einzige Anweisung.

Erfordert die Syntax eines Programms genau eine einzige Anweisung, wie beispielsweise bei der `if`-Struktur, so können dennoch mehrere Anweisungen geschrieben werden, wenn man sie in Form eines Blockes zusammenfasst.

Blöcke werden noch ausführlich in Kapitel → 11.1 behandelt.

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_10.

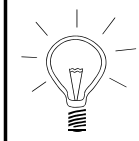

J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_10

## 10.2 `if` und `else` – Einfache Selektion

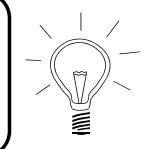
Für die **einfache Selektion** werden die Schlüsselwörter **`if`** und **`else`** benötigt. Die Syntax ist die Folgende:

```
if (Bedingung)
  Anweisung 1
else
  Anweisung 2
```

Bei der einfachen Selektion wird die Bedingung in runden Klammern notiert. Ist die Bedingung wahr, so wird `Anweisung 1` ausgeführt. Ist die Bedingung nicht wahr, so wird `Anweisung 2` ausgeführt.

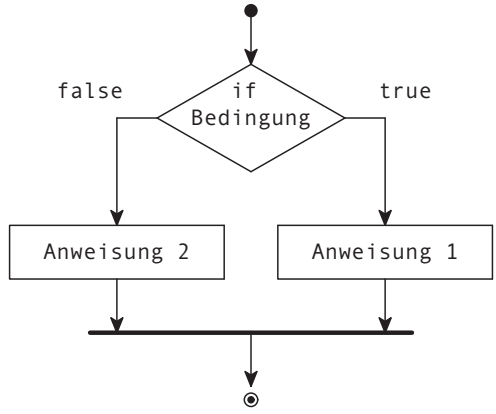


Eine **Bedingung** gilt als wahr, wenn der Ausdruck zu einer von `0` verschiedenen Zahl ausgewertet wird und als falsch, wenn der Ausdruck zu `0` ausgewertet wird.



Die beiden Anweisungen nach der Bedingung und nach dem `else` werden als die Zweige der Kontrollstruktur bezeichnet. Die `if`-Struktur „verzweigt“ somit zu einem der beiden Anweisungen. Im Englischen spricht man auch von „branching“.

Das folgende Bild zeigt das zugehörige Diagram:



Soll mehr als eine einzige Anweisung ausgeführt werden, so ist ein Block mit geschweiften Klammern zu verwenden, der syntaktisch als eine einzige Anweisung zählt:

```
if (Bedingung) {
  Anweisung 1.1
  Anweisung 1.2
} else {
  Anweisung 2.1
  Anweisung 2.2
}
```

Heutzutage gilt es als unschön, eine `if`-Struktur ohne Block, sprich ohne geschweifte Klammer zu schreiben. Dennoch wird die Schreibweise ohne Klammern in seltenen Fällen auch heute noch verwendet, um kompakteren Code zu schreiben.

Der `else`-Zweig ist optional. Entfällt der `else`-Zweig, so spricht man von einer bedingten Anweisung:

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

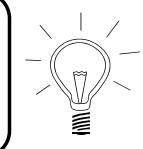
Bei einer bedingten Anweisung wird die Anweisung nur dann ausgeführt, wenn die Bedingung zutrifft. Trifft die Bedingung nicht zu, so wird sofort zu der Anweisung nach der bedingten Anweisung gesprungen.

Für die Bedingung in den runden Klammern werden am häufigsten Vergleichs-Operatoren → 9.6 wie beispielsweise `a < 100` verwendet. Die Bedingung kann jedoch ein beliebiger Ausdruck sein, der zu einem numerischen Wert ausgewertet wird. Es zählt einzig und alleine, ob dieser Wert `0` oder eine von `0` verschiedene Zahl ist. Somit können auch andere numerische Werte als Bedingung herhalten. Folgende beiden Zeilen sind somit äquivalent:

```
if (a != 0) ...
if (a) ...
```

### 10.2.1 Geschachtelte if-Strukturen

Im if-Zweig darf eine beliebige Anweisung stehen. Da die if-Struktur selbst als Anweisung gilt, können somit auch geschachtelte if-Strukturen entstehen.

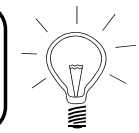


Ein einfaches Beispiel ist eine Struktur wie die folgende:

```
if (a > 1000)
  if (a > 1000000)
    printf("Sehr grosse Zahl");
  else
    printf("Grosse Zahl");
```

Hierbei ist zu beachten, dass in dieser komplexen Struktur nur ein else für die beiden if-Zweige existiert.

Der else-Zweig wird immer mit dem letzten if verbunden, für das noch kein else-Zweig existiert.

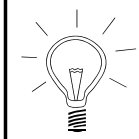


So gehört im obigen Beispiel der else-Zweig somit zur Bedingung a > 1000000. Im Code wird die Zuordnung der Fälle durch Einrücken der Codezeilen visuell kenntlich gemacht. Für den Compiler haben solche Einrückungen jedoch keinerlei Bedeutung. Bei komplexerem Code empfiehlt sich somit, mittels geschweifter Klammern Blöcke zu definieren:

```
if (a > 1000) {
  if (a > 1000000) {
    printf("Sehr grosse Zahl");
  } else {
    printf("Grosse Zahl");
  }
}
```

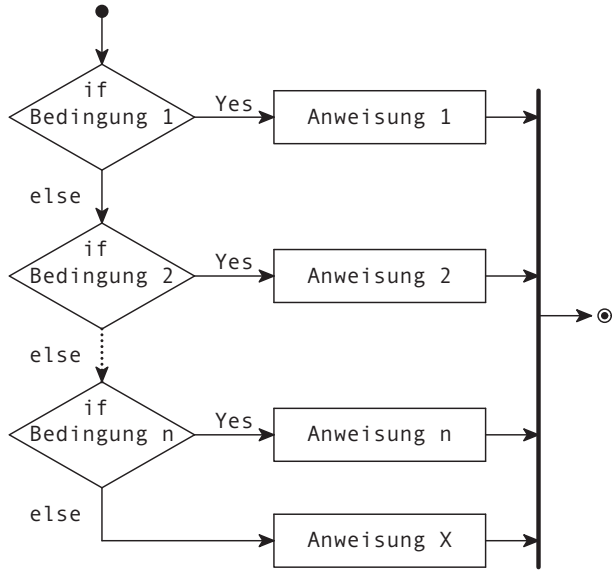
### 10.2.2 else if – Mehrfache Selektion

Im else-Zweig darf ebenfalls eine beliebige Anweisung stehen, also auch erneut eine if-Struktur. Dieses somit entstandene **else if** ist eine einfache Möglichkeit, eine Auswahl aus verschiedenen Fällen zu treffen.



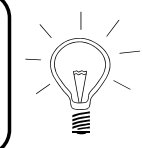
Durch wiederholtes Einsetzen einer bedingten Anweisung in den jeweils letzten else-Zweig entsteht eine Mehrfach-Selektion:

```
if (Bedingung 1)
  Anweisung 1
else if (Bedingung 2)
  Anweisung 2
...
else if (Bedingung n)
  Anweisung n
else
  Anweisung X
```




In der angegebenen Reihenfolge wird ein Vergleich nach dem anderen durchgeführt. Bei der ersten Bedingung, die wahr ist, wird die zugehörige Anweisung abgearbeitet und danach die if-Struktur abgebrochen. Dabei kann wiederum statt einer einzelnen Anweisung stets auch ein Block von Anweisungen stehen.

Der letzte `else`-Zweig ist optional. Hier werden alle anderen Fälle behandelt, die in den vorherigen Bedingungen nicht explizit aufgeführt sind.

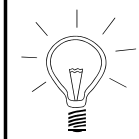


Dieser `else`-Zweig wird normalerweise entweder für die Behandlung von Daten verwendet, welche keine Spezialbehandlung benötigen, oder zum Abfangen von Fehlern oder noch nicht ausprogrammierten Programmteilen.

Mit `else if` können somit ganz einfach verschiedene Bedingungen geprüft werden. Die grundlegende Entscheidung, welcher Zweig denn nun ausgeführt werden soll, wird jedoch grundsätzlich immer nur zwischen zwei Möglichkeiten (`0` oder nicht `0`) gefällt. Die `switch`-Struktur bietet eine andere Art der mehrfachen Selektion:

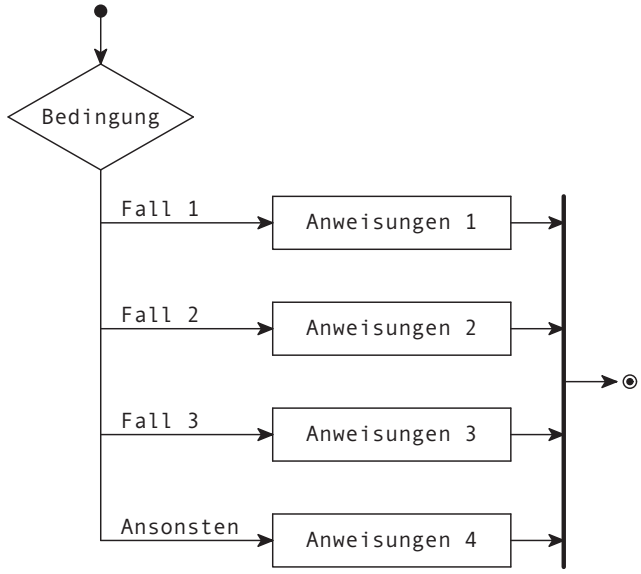
## 10.3 `switch` – Fallunterscheidung

Für eine **Mehrfach-Selektion** basierend auf einer Integer-Zahl kann die **`switch`**-Struktur verwendet werden. Die `switch`-Struktur verzweigt direkt zu dem gewünschten „**Fall**“, welcher dem gegebenen Integer-Wert entspricht.



```
switch (Bedingung) {
  case 1:
    Anweisungen 1
    break;
  case 2:
    Anweisungen 2
    break;
  ...
  case n:
    Anweisungen n
    break;
  default:
    Anweisungen X
    break;
}
```

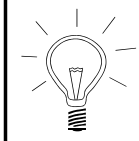
Die vorangegangene switch-Struktur wird durch das folgende Diagramm visualisiert:




Jeder Fall wird mit dem Schlüsselwort **case**, der gewünschten Integer-Konstanten 1, ..., n und einem Doppelpunkt eingeleitet. Dies wird als case-Marke (auf Englisch „case label“) bezeichnet.

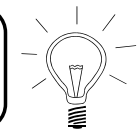
Die Integer-Konstanten können auch konstante Ausdrücke wie beispielsweise `(4 * 3)` sein.

> Hat der Ausdruck der Bedingung den gleichen Wert wie einer der konstanten Ausdrücke der case-Marken, so wird die Ausführung des Programms mit der Anweisung hinter dieser case-Marke weitergeführt. Stimmt keiner der konstanten Ausdrücke mit dem switch-Ausdruck überein, so wird zur Anweisung nach der **`default`**-Marke (auf Englisch „default label“) gesprungen.



Die default-Marke ist optional. Wenn keine default-Marke in der switch-Struktur hingeschrieben wird, wird das Programm bei Nichtzutreffen aller aufgeführten konstanten Ausdrücke mit der Anweisung nach der switch-Struktur fortgeführt.

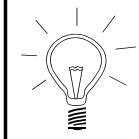
Innerhalb einer switch-Struktur müssen alle case-Marken voneinander verschieden sein.



Wird bei der Auswertung der Bedingung eine passende case-Marke gefunden, werden die dort stehenden Anweisungen bis zum break ausgeführt. **break** springt dann zu der auf die switch-Struktur folgenden Anweisung. Die break-Anweisung wird ausführlich in Kapitel → 10.8 behandelt.

Die Reihenfolge der case-Marken ist beliebig. Auch die default-Marke muss nicht als letzte stehen. Dennoch hat es sich eingebürgert, dass die case-Marken normalerweise nach aufsteigenden Werten geordnet sind und die default-Marke am Schluss steht.

Fehlt die break-Anweisung, so werden die nach der nächsten case-Marke folgenden Anweisungen abgearbeitet. Dies geht so lange weiter, bis ein break gefunden wird oder bis das Ende der switch-Struktur erreicht ist.



Grundsätzlich sollte jeder Fall einer switch-Struktur mit einer break-Anweisung abgeschlossen werden. Wird eine break-Anweisung vergessen, führt das in aller Regel zu schwer zu entdeckenden Fehlern. Nur in ganz seltenen Fällen ist das Weglassen der break-Anweisung wirklich gewünscht.


### 10.3.1 enum-Aufzählungs-Konstanten als Bedingung

Die switch-Struktur eignet sich besonders gut für Fallunterscheidungen. Entsprechend wird diese Struktur häufig verwendet, um zwischen **Aufzählungs-Konstanten** zu unterscheiden, welche mittels des **enum**-Typs → 7.2.5 definiert wurden.

Der Code wird dadurch lesbarer, da anstelle von Zahlen symbolische Konstanten im Code stehen.

ampel.c

```
#include <stdio.h>

enum Color {RED, ORANGE, GREEN};

int main(void) {
  enum Color light = RED;

  switch (light) {
    case RED:
      printf("Stopp!");
      break;
    case ORANGE:
      printf("Warten...");
      break;
    case GREEN:
      printf("Los!");
      break;
    default:
      printf("Ampel defekt.");
  }
  return 0;
}
```

Die Ausgabe lautet:

```
Stopp!
```

Das Programm führt in der vorliegenden Form selbstverständlich stets zum selben Resultat, da als Wert von `light` im Programm `RED` fest vorgegeben wird. Nützlich wäre eine solche `switch`-Struktur beispielsweise in einer Funktion, welche die Variable `light` als Parameter definiert.

Die `break`-Anweisung am Ende des `default`-Falles kann ohne Probleme weggelassen werden, da die `switch`-Struktur an dieser Stelle schon fertig abgearbeitet ist. Würde man jedoch in diesem Beispiel die anderen `break`-Anweisungen vergessen, so wäre das Resultat `Stopp!Warten...Los!Ampel defekt.`

Es ist zu beachten, dass symbolische Konstanten nicht nur mittels `enum`, sondern auch mittels `#define` erstellt werden können → 21.2.1. Die Verwendung von `enum` ist jedoch klar bevorzugt.

### 10.3.2 Reihen von case-Marken

Die case-Marken einer switch-Struktur führen selbst keinen Code aus, sondern markieren nur den Ort, zu welchem gesprungen werden soll. Da jede case-Marke auf die jeweils darauffolgende Anweisung zeigt, ist es auch möglich, mehrere case-Marken hintereinander zu schreiben, welche schlussendlich allesamt auf dieselbe Anweisung zeigen.

Im folgenden Beispiel wird ausgewertet, ob eine gewürfelte Zahl gerade oder ungerade ist:

wuerfel.c

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

int main(void) {
  srand((int)time(NULL));
  int zahl = rand() % 6 + 1;

  switch (zahl) {
    case 1:
    case 3:
    case 5:
      printf("Ungerade Zahl gewuerfelt: %d\n", zahl);
      break;
    case 2:
    case 4:
    case 6:
      printf("Gerade Zahl gewuerfelt: %d\n", zahl);
      break;
  }
  return 0;
}
```

### 10.3.3 Unterschiede zwischen `switch` und `else if`

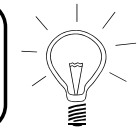
Sowohl mit `switch` als auch mit `else if` kann **Mehrfach-Selektion** abgearbeitet werden. Dennoch werden die beiden Strukturen in unterschiedlichen Situationen genutzt.

Grundsätzlich gilt: Die `if`-Struktur eignet sich gut für logische Entscheidungen, wohingegen die `switch`-Struktur für Fallunterscheidungen gebraucht wird.

Die folgenden Unterschiede bestehen:

- `switch` prüft in der Bedingung auf die Gleichheit von Integer-Werten mit den vorhandenen `case`-Marken, wohingegen bei `else if` getestet wird, ob die Bedingung `0` oder nicht `0` ist.
- Die Bedingung der `if`-Struktur erlaubt beliebige Werte, solange sie zu einem Null-Wert ausgewertet werden. Dies beinhaltet Gleitpunkt-Zahlen sowie Pointer.
- Die Bedingung der `switch`-Struktur erlaubt explizit nur Integer oder Ausdrücke, welche zu solchen ausgewertet werden. Zeichen-Literale → 6.5.3 wie `'A'` können beispielsweise als Integer-Zahlen interpretiert werden, ein Pointer oder eine Gleitpunkt-Zahl jedoch nicht.
- Die Effizienz der `switch`-Struktur ist gegenüber der `else if`-Anweisung in der Regel besser, da bedingt durch die konstanten `case`-Marken der Compiler bessere Optimierungsmöglichkeiten hat. Im Falle der `else if`-Bedingungen kann die Auswertung normalerweise erst zur Laufzeit erfolgen.
- Die Übersichtlichkeit beziehungsweise Erweiterbarkeit ist bei `switch` besser als bei `else if`.
- Moderne Compiler können bei `switch` auf die vollständige Auflistung aller Aufzählungs-Konstanten eines Aufzählungs-Typs `enum` → 7.2.5 prüfen. Dadurch werden keine Fälle vergessen.

> Falls möglich, sollte statt umfangreicher `else if`-Konstruktionen bevorzugt `switch` benutzt werden.



### 10.3.4 Definition von Variablen innnerhalb der switch-Struktur 

Innerhalb der geschweiften Klammern einer switch-Struktur können Variablen definiert werden. Hierbei ist jedoch Vorsicht geboten:

> Werden innerhalb einer switch-Struktur Variablen definiert, so sind sie grundsätzlich im gesamten switch-Block sichtbar. Sie werden jedoch nur in demjenigen case-Fall initialisiert, in welchem ihre Definition steht. Steht die Initialisierung vor sämtlichen case-Fällen, wird sie niemals initialisiert, was zu einem undefinierten Verhalten führt.


Variablen werden dementsprechend innerhalb einer switch-Struktur nur selten definiert. Um die damit entstehenden Probleme zu lösen, gibt es einige Möglichkeiten, welche jedoch meist den Code unleserlicher machen:

- Die Variable wird einfach außerhalb der Struktur definiert. Gewisse Fälle könnten diese Variable jedoch nie brauchen, weswegen die Initialisierung dann möglicherweise überflüssig war. Außerdem befindet sich die Variable dann in dem Block außerhalb, wo sie nicht hingehört.

- Umklammerung eines Falls mit geschweiften Klammern. Dadurch entsteht ein eigener Code-Block, was die Sichtbarkeit und Lebensdauer der Variablen klar definiert. Dies sieht aber nicht schön aus.

- Aufruf einer Funktion anstelle der Ausprogrammierung des Falls innerhalb der switch-Struktur. Wird eine Funktion aufgerufen, kann dort die Variable ohne Probleme definiert und initialisiert werden. Leider ist dafür jedoch ein kostspieliger Funktionsaufruf nötig.

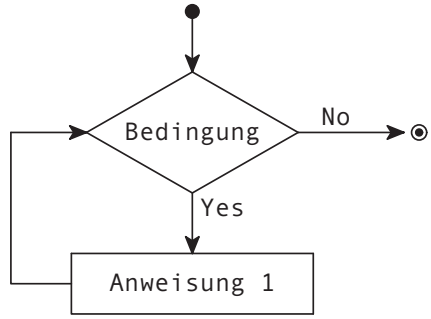
Es ist erwähnenswert, dass die Definition von Variablen innerhalb einer switch-Struktur vor dem Standard C99 nicht möglich war.

Auf Beispiele und weitere Erläuterungen wird hier verzichtet.

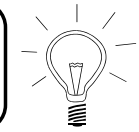
## 10.4 while – Bedingte Schleife

Die Syntax der **while**-Schleife lautet:

```
while (Bedingung)
  Anweisung
```




In einer `while`-Schleife wird eine Anweisung in Abhängigkeit von einer Bedingung wiederholt ausgeführt.



Die Bedingung wird vor jedem Durchgang ausgewertet und die Anweisung nur dann ausgeführt, wenn das Resultat ungleich `0` ist. Dieser Vorgang wiederholt sich danach, indem der Ausdruck erneut ausgewertet und die Anweisung ausgeführt wird, solange, bis der Ausdruck der Bedingung `0` ergibt.

Sollen mehrere Anweisungen ausgeführt werden, so ist ein Block mit geschweiften Klammern zu verwenden.


Sobald die Bedingung nicht mehr erfüllt ist, fährt das Programm mit der ersten Anweisung nach der Schleife fort.

Da die Bedingung vor der Ausführung der Anweisung bewertet wird, kann es sein, dass die Anweisung niemals ausgeführt wird, da die Bedingung bereits zu Beginn nicht erfüllt ist. Es wird deswegen auch von einer „abweisenden“ Schleife gesprochen.

Folgendes einfaches Beispiel gibt von einem gegebenen String Zeichen für Zeichen aus, bis das erste Mal der Buchstabe `'e'` auftaucht.

while.c

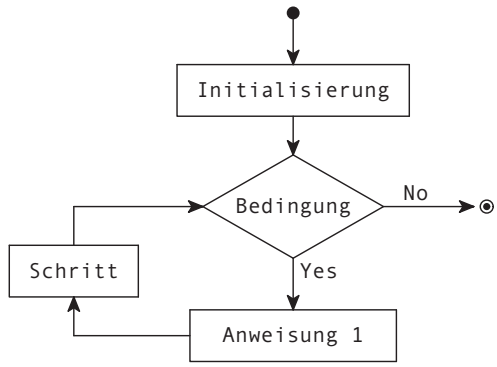
```
#include <stdio.h>

int main(void) {
  const char* text = "Hallo Welt";

  int pos = 0;
  while (text[pos] && text[pos] != 'e') {
    printf("%c", text[pos]);
    ++pos;
  }
  return 0;
}
```

Für den Eingabe-String `Hallo Welt` wird somit `Hallo W` ausgegeben. Es ist zu beachten, dass die Bedingung zusätzlich zur Erkennung des Buchstabens `'e'` auch noch überprüft, ob der Buchstabe an Position pos nicht der Null-Character `'\0'` ist. Damit wird das Ende des Strings erkannt. Ohne diese zusätzliche Prüfung würde das Programm bei einem Eingabe-String ohne den Buchstaben `'e'` über das Stringende hinauslaufen und auf unzulässigen Speicher zugreifen.

## 10.5 for – Zählschleife

Die Syntax der **for**-Schleife lautet:

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



Die runden Klammern der for-Schleife enthalten drei durch Semikolons getrennte Ausdrücke:

- **Initialisierung**: Hier wird üblicherweise eine Variable definiert und initialisiert, welche im Verlauf der Schleife als **Laufvariable** dient. Andere gebräuchliche Namen sind Zählvariable oder Schleifenindex.
- **Bedingung**: Ein Ausdruck, der ähnlich wie bei einer while-Schleife vor jedem Durchlauf ausgewertet wird. Nur wenn dieser Ausdruck erfüllt ist, wird die Anweisung ausgeführt.
- **Schritt**: Wird am Ende jedes Durchlaufs automatisch ausgeführt.

Sollen mehrere Anweisungen ausgeführt werden, so ist ein Block mit geschweiften Klammern zu verwenden.


Die for-Schleife wird häufig als **Zählschleife** verwendet. Folgendes ist ein typisches Beispiel für eine for-Schleife:

increment.c

```
#include <stdio.h>
#define COUNT 10

int main(void) {
  for (int i = 0; i < COUNT; ++i) {
    printf("%d ", i);
  }
  return 0;
}
```

Die Ausgabe dieses Programmes lautet:

```
0 1 2 3 4 5 6 7 8 9
```

Die Laufvariable i wird in diesem Beispiel definiert und mit 0 initialisiert. Ein Durchlauf findet nur dann statt, wenn i < 10 gilt. Nach jedem Durchlauf wird die Laufvariable um 1 erhöht.

Die for-Schleife wird wie die while-Schleife eine „abweisende Schleife“ genannt, da vor Ausführung eines Durchlaufs die Bedingung geprüft wird und somit eine Schleife auch niemals durchlaufen werden kann.

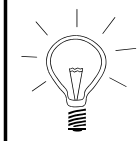
### 10.5.1 Inhalt der Laufvariablen außerhalb der for-Schleife

Die Laufvariable wird üblicherweise direkt in den runden Klammern der for-Schleife definiert und initialisiert. Dadurch ist die Sichtbarkeit und Lebensdauer der Laufvariable klar auf die for-Schleife begrenzt.

Eine Variable zu definieren, ist jedoch nicht zwingend. Auch bereits außerhalb definierte Variablen können als Laufvariablen dienen. Diese Variablen haben selbstverständlich einen größeren Sichtbarkeitsbereich und eine längere Lebensdauer als die Schleife. Vor dem Standard C99 war dies gar die einzige Möglichkeit, eine Laufvariable zu definieren.

Nachdem die for-Schleife beendet wurde, bleibt in so einem Falle der aktuelle Wert der Laufvariablen erhalten, selbst wenn die Schleife mit der break-Anweisung → 10.8 verlassen wird.

> Die Laufvariable einer for-Schleife, die bereits vor der Schleife definiert wird, enthält nach dem Verlassen der Schleife immer den zuletzt in der Schleife verwendeten Wert beziehungsweise den Wert, welcher zum Abbruch der Bedingung führte.



Die seit C99 gegebene Möglichkeit, die Variable innerhalb der runden Klammern zu definieren, wird in den meisten Fällen bevorzugt. Die Einschränkung der Sichtbarkeit der Laufvariablen ist sinnvoll, denn in den meisten Fällen ist der Zustand der Laufvariablen nach Abarbeitung der Schleife nicht mehr von Bedeutung. Wenn Variablen nicht sichtbar sind, werden unbeabsichtigte Fehler mit ihnen vermieden.

### 10.5.2 Dekrementierende for-Schleifen

Dieses Beispiel zeigt eine Schleife, bei welcher der Wert der Laufvariablen in jedem Durchgang verringert wird:

decrement.c

```
#include <stdio.h>
#define COUNT 10

int main(void) {
  for (int i = COUNT - 1; i >= 0; --i) {
    printf("%d ", i);
  }
  return 0;
}
```

Die Ausgabe dieses Programmes lautet:

```
9 8 7 6 5 4 3 2 1 0
```

Interessant hierbei ist, dass bei einer dekrementierenden Schleife üblicherweise mit dem Wert `Startwert - 1` initialisiert wird und die Laufvariable danach auf größer gleich `0` geprüft wird. Dies deswegen, da Laufvariablen häufig für Indizes von Arrays verwendet werden, welche bekanntlich in C von `0` anfangen zu zählen → 8.3.

### 10.5.3 Beispiel für Array-Indizes in einer for-Schleife

Häufig werden for-Schleifen für die Abarbeitung von ganzen Arrays verwendet. Das folgende Programm zeigt, wie ganz einfach die Fibonacci-Zahlen berechnet und in einem Array gespeichert werden können:

fibonacci.c

```
#include <stdio.h>

#define COUNT 10

int main(void) {
  int array[COUNT];
  array[0] = 1;
  array[1] = 1;
  printf("%d ", array[0]);
  printf("%d ", array[1]);

  for (int i = 2; i < COUNT; ++i) {
    array[i] = array[i - 1] + array[i - 2];
    printf("%d ", array[i]);
  }

  return 0;
}
```

Die Ausgabe dieses Programmes lautet:

```
1 1 2 3 5 8 13 21 34 55
```

Das Programm weist dem aktuellen Array-Element array[i] immer die Summe der beiden vorangegangenen Array-Einträge zu. Damit kein fehlerhafter Zugriff stattfindet, müssen die beiden ersten Array-Einträge manuell gefüllt und die Laufvariable muss mit dem Index 2 initialisiert werden.

### 10.5.4 Komplizierteres Beispiel

Alle drei Ausdrücke der for-Schleife können beliebig komplex werden. Folgendes (wenig sinnvolles) Beispiel hat folgende Anweisungen:

- **Initialisierung**: Variable i mit Startwert 5 und Variable k mit Startwert 1.
- **Bedingung**: Variable i muss kleiner als COUNT sein und außerdem muss Variable k kleiner als 10 sein.
- **Schritt**: Variable i wird um k erhöht und k um 1.

steps.c

```
#include <stdio.h>
#define COUNT 100

int main(void) {
  for (int i = 5, k = 1; i < COUNT && k < 10; i += k, ++k) {
    printf("%d ", i);
  }
  return 0;
}
```

Die Ausgabe dieses Programmes lautet:

```
5 6 8 11 15 20 26 33 41
```

Es ist zu beachten, dass bei der Initialisierung die beiden Variablen mithilfe des Kommas definiert wurden. So können zwei oder mehrere Variablen mit demselben Typ definiert werden (siehe Kapitel → 7.4.1). Für die Schritt-Anweisung wurde das Komma des Sequenz-Operators → 9.9.3 verwendet. Dies ist ein kleiner Trick, welcher manchmal genutzt wird, um mehr als nur eine Varaible im Schritt-Ausdruck zu verändern.

Auch wenn dieses Beispiel wenig sinnvoll war, zeigt es doch, wie kompliziert die Ausdrücke werden können und wie schwierig es werden kann, hierbei den Überblick zu behalten. Die Initialisierungs-, Bedingungs- und Schritt-Ausdrücke der for-Schleife sollten wann immer möglich sehr einfach gehalten werden.

### 10.5.5 Verschachtelte for-Schleifen

Zum Abschluss hier ein Beispiel mit zwei verschachtelten Schleifen. Es sollen die Primzahlen aufgezählt werden:

prim.c

```
#include <stdio.h>
#define MAX 30

int main(void) {
  for (int i = 2; i <= MAX; i = i + 1) {
    int istTeilbar = 0;

    for (int k = 2; k < i; ++k) {
      if (i % k == 0) {
        istTeilbar = 1;
      }
    }

    if (!istTeilbar) {
      printf("%d ", i);
    }
  }
  return 0;
}
```

Die Ausgabe dieses Programmes lautet:

```
2 3 5 7 11 13 17 19 23 29
```

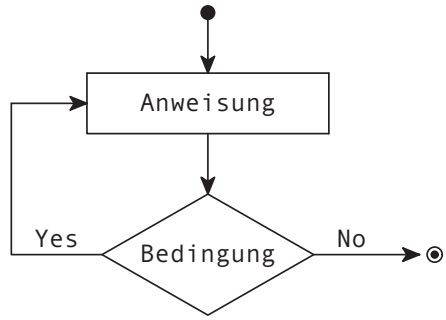
Das Programm berechnet die Primzahlen bis MAX. Dazu wird in einer äußeren for-Schleife die Laufvariable i von 2 bis MAX hochgezählt. Dann prüft die innere Schleife, die nach Teilern von i sucht, ob i modulo k den Wert 0 ergibt, also ob i durch k ohne Rest teilbar ist. Ist das der Fall, dann ist i keine Primzahl, da sie ja in zwei Faktoren ohne Rest zerlegt werden kann.

Diese Schleife läuft so lange, wie k kleiner als i ist. Falls i beim gesamten Durchlaufen der inneren Schleife nicht durch k ohne Rest teilbar war, liegt somit eine Primzahl vor und sie wird auf dem Bildschirm ausgegeben.

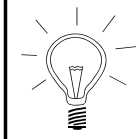
## 10.6 do while – Annehmende bedingte Schleife

Die Syntax der **do while**-Schleife ist:

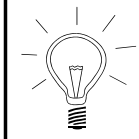
```
do {
  Anweisung
} while (Bedingung);
```




Bei der do while-Schleife wird zuerst die Anweisung ausgeführt und erst danach wird die Bedingung zum ersten Mal ausgewertet. Genauso wie bei der while-Schleife wird danach die Anweisung solange ausgeführt, wie die Bedingung erfüllt ist. Sobald die Bedingung nicht mehr erfüllt ist, wird mit der ersten Anweisung nach der do while-Schleife fortgefahren.



Die Anweisung der do while-Schleife wird auf jeden Fall mindestens einmal durchlaufen, da die Bewertung des Ausdrucks erst am Ende der Schleife erfolgt. Die do while-Schleife wird somit als eine „annehmende Schleife“ bezeichnet. Im Gegensatz dazu können die for- und while-Schleife durchaus ihre Anweisung überhaupt nicht ausführen, nämlich dann, wenn die Schleifenbedingung von Anfang an nicht erfüllt ist.



Im Gegensatz zu allen anderen Schleifen wird bei der do while- Schleife ein Semikolon nach der Schleifenbedingung verlangt, weil hier das Ende der Anweisung erreicht ist.


In folgendem Beispiel wird eine Zahlenreihe von 0 bis 9 ausgegeben:

dowhile.c

```
#include <stdio.h>

int main(void) {
  int i = 0;
  do {
    printf("%d ",i);
    ++i;
  } while (i < 10);

  return 0;
}
```

Die Ausgabe dieses Programmes lautet:

```
0 1 2 3 4 5 6 7 8 9
```

Es ist zu beachten, dass die do while-Schleife äußerst selten benutzt wird. Zum einen ist der Fall, dass eine Schleife mindestens einmal durchlaufen werden muss, tendenziell rar. Zum anderen ist die Schleife weniger gut zu lesen, da sich die Bedingung am Ende befindet.

## 10.7 Endlosschleifen und leere Ausdrücke bei Schleifen

Eine **Endlosschleife** ist eine Schleife, bei welcher die Bedingung niemals fehlschlägt. Der Ausdruck der Schleife wird somit unendlich oft ausgeführt, sofern die Schleife nicht anderweitig unterbrochen wird.

Ein einfaches Beispiel einer Endlosschleife ist das Folgende:

```
while (1)
  Anweisung
```

Auch mit der `for`-Schleife kann entsprechend ganz einfach eine Endlosschleife erzeugt werde. Bei der `for`-Schleife gibt es jedoch noch eine andere Möglichkeit:

Jeder der drei Ausdrücke einer `for`-Schleife kann leer sein. Die Strichpunkte müssen aber trotz fehlendem Ausdruck stehen bleiben. Fehlt der Ausdruck der Initialisierung, wird nichts initialisiert. Fehlt der Ausdruck des Schritts, wird nach einem Durchgang nichts automatisch verändert. Fehlt der Ausdruck der Bedingung, so gilt die Bedingung immer als `1` und die Schleife wird somit eine Endlosschleife. Die geläufigste Form ist dabei, alle drei Ausdrücke wegzulassen, wie im folgenden Beispiel:

```
for ( ; ; )
  Anweisung
```

Endlosschleifen können entweder gewollt oder ungewollt sein. Ungewollte Endlosschleifen führen zum „Aufhängen“ des Programmes, ein Zustand, in welchem das Programm nichts mehr tut und schlussendlich vom System gestoppt werden muss.

Solche ungewollte Endlosschleifen können passieren, wenn die Bedingung der Schleife nicht korrekt programmiert wurde oder wenn die Bedingung sich während der Abarbeitung der Anweisung niemals ändert.

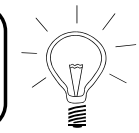
Schleifen, bei denen sich die Bedingung nie ändert, können jedoch auch gewollt programmiert werden. Man nutzt dann häufig die **`break`**-Sprunganweisung → 10.8, um die Schleife zu einem beliebigen Zeitpunkt zu verlassen. Formal ist die Schleife immer noch eine Endlosschleife, aber das Abbruchkriterium wird nicht in der Bedingung, sondern innerhalb der Schleife geprüft. Ein einfaches Beispiel ist folgende Uhr:

busywait.c

```
// MSVC meldet fuer veraltete, unsichere C-Funktionen wie
// localtime einen Fehler. So wird dieser Fehler ausgeschaltet:
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <time.h>
#ifdef _WIN32
  #include <windows.h>
  #define SleepFunction Sleep
#else
  #include <unistd.h>
  #define SleepFunction sleep
#endif
```

```
int main(void) {
  while (1) {
    time_t t = time(0);
    struct tm* now = localtime(&t);

    printf("%d:%d:%d\n", now->tm_hour, now->tm_min, now->tm_sec);
    fflush(stdout);

    if (now->tm_hour == 12) {
      printf("Lunch!");
      break;
    }

    SleepFunction(1);
  }
  return 0;
}
```

Endlosschleifen ohne `break`-Anweisung sind in der Regel nicht sinnvoll.

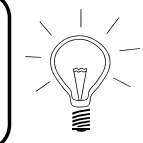


Sinnvolle Anwendungen von Endlosschleifen ohne `break`-Anweisung gibt es jedoch: Beispielsweise arbeiten moderne GUI-Programme mittels sogenannter „Application-Loops“, welche einfach nur Eingaben abwarten und nach Bearbeitung einer Eingabe sofort wieder in Wartestellung gehen und dies unendlich lange wiederholen. Auch sogenannte „demons“, welche das Betriebssystem unterstützen, arbeiten häufig mit einer unendlichen Warteschleife. Auch beim Thema Multithreading → 23 werden häufig solche Loops verwendet.

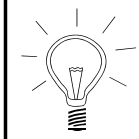
## 10.8 break – Abbruch-Anweisung

Schleifen und switch-Strukturen können mittels der Anweisung break frühzeitig abgebrochen werden.

Mit der **break**-Anweisung kann eine do while-, while- und for-Schleife und eine switch-Struktur abgebrochen werden.



Abgebrochen wird immer nur die aktuelle Schleife beziehungsweise switch-Struktur. Sind mehrere Schleifen oder switch-Strukturen geschachtelt, wird lediglich die innerste verlassen.



Beispiele zur switch-Struktur können in Kapitel → 10.3 nachgelesen werden. Das folgende Beispiel zeigt das Verlassen einer for-Schleife mit break:

break.c

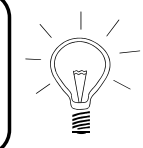
```
#include <stdio.h>
#include <string.h>

int main(void) {
  const char* text = "Hallo Welt";
  int pos;

  for (pos = 0; pos < strlen(text); ++pos) {
    if (text[pos] == 'e')
      break;
  }

  printf("Das erste e ist an Stelle %d.\n", pos);
  return 0;
}
```

Die Ausgabe lautet:

```
Das erste e ist an Stelle 7.
```

## 10.9 continue – Fortsetzungs-Anweisung

Die **continue**-Anweisung ist wie break eine Sprunganweisung. Im Gegensatz zu break wird aber eine Schleife nicht verlassen, sondern ein neuer Durchgang gestartet.



Die continue-Anweisung kann auf die do while-, die while- und die for-Schleife angewandt werden. Bei do while und while wird nach continue direkt zum Bedingungstest der Schleife gesprungen. Bei der for-Schleife wird zuerst noch der Schritt ausgeführt.

Angewandt wird die continue-Anweisung zum Beispiel, wenn an einer gewissen Stelle des Schleifenrumpfes mit einem Test festgestellt werden kann, ob der restliche Teil noch ausgeführt werden muss beziehungsweise darf.

Das folgende Beispiel zeigt die Verwendung von continue in einer while-Schleife. Die continue-Anweisung führt hier dazu, dass nur gerade Zahlen aufsummiert werden.

continue.c

```
#include <stdio.h>

int main(void) {
  int sum = 0;
  for (int i = 0; i < 20; ++i) {
    if (i % 2) {
      continue;
    }
    sum += i;
  }
  printf("Summe: %d\n", sum);

  return 0;
}
```

Die Ausgabe lautet:

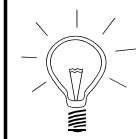
```
Summe: 90
```

## 10.10 return – Rücksprung-Anweisung

Die return-Anweisung wird ausführlich in Kapitel → 11.4 behandelt.

## 10.11 goto – Sprunganweisung und Marken

Mit **goto** wird ohne Bedingung an eine gegebene Marke gesprungen. Eine **Marke** (auf Englisch „**label**“) wird durch Angabe eines Namens, gefolgt von einem Doppelpunkt : in den Code geschrieben.



Die Verwendung von Sprüngen und Marken ist in der Assembler-Programmierung eine Selbstverständlichkeit. In C wird diese Funktionalität mit der goto-Anweisung nachgebildet.

Eine Marke wird definiert durch einen Namen, der mit einem Doppelpunkt abgeschlossen ist. Stehen darf eine Marke vor jeder beliebigen Anweisung. Die Gültigkeit einer Marke erstreckt sich über die Funktion, in welcher sie definiert wurde, muss also nicht vorher deklariert werden. Gleichzeitig kann aber auch nicht aus einer Funktion herausgesprungen werden.

Um an eine bestimmte Marke zu springen, wird einfach die goto-Anweisung mit entsprechender Marke geschrieben:

```
Anweisungen
goto HierGehtEsWeiter;
Anweisungen
HierGehtEsWeiter:
Anweisungen
```

In der strukturierten Programmierung ist die goto-Anweisung nicht erlaubt.

Der Streit um die goto-Anweisung ist legendär: Viele Aufsätze und Gegenaufsätze um dieses Thema wurden verfasst. Am berühmtesten ist wohl der Artikel „Go To Statement Considered Harmful“ [Dij68], was im Deutschen etwa „Goto ist gefährlich“ bedeutet.

Die in diesem Kapitel beschriebenen Kontrollstrukturen sind allesamt mächtig genug, um einen Einsatz von goto komplett zu vermeiden. Nur noch äußerst selten wird diese Sprunganweisung verwendet, beispielsweise um das Herausspringen aus mehrfach geschachtelten Schleifen im Fehlerfall einfacher zu gestalten. Aber diese Fälle sind nahezu ausgestorben.

## 10.12 Übungsaufgaben

### Aufgabe 1: Maximum berechnen

a) Schreiben Sie eine Funktion, welche 3 Integer-Zahlen erwartet und das Maximum der 3 Zahlen zurückgibt. Nutzen Sie hierfür eine `if`-Struktur.

```
int getMax3(int value1, int value2, int value3);
```

b) Schreiben Sie eine Funktion, welche beliebig viele Zahlen als Array erwartet. Nutzen Sie hierfür eine `for`-Schleife.

```
int getMaxn(const int* values, int arrayCount);
```

c) Schreiben Sie eine weitere Funktion, welche ein Array von Integer-Zahlen erwartet und prüft, ob sämtliche Zahlen kleiner sind als ein gegebenes Maximum. Falls dem so ist, wird `1` zurückgegeben, ansonsten `0`. Nutzen Sie dabei die `break`- oder `return`-Anweisung, um frühzeitig abzubrechen, wenn möglich.

```
int areAllLower(const int* values, int arrayCount, int max);
```

### Aufgabe 2: Zwei Schleifen mit gleicher Funktionalität

Die Schleifen `while` und `for` verhalten sich unterschiedlich, doch können sie grundsätzlich immer durch die jeweils andere Schleife ersetzt werden.

Programmieren Sie folgende Aufgabe einmal mit einer `for`- und einmal mit einer `while`-Schleife. Schreiben Sie hierfür zwei Funktionen, einmal mit einer Index-Berechnung und einmal mit einer `const char*` Variable. Verwenden Sie zur Berechnung der Länge des gegebenen Strings die `strlen()`-Funktion der `<string.h>` Bibliothek.

Berechnen Sie die Quersumme einer Zahl. Die Zahl ist gegeben als ein String. Beispielsweise `"623518894436"`. Die einfache Quersumme dieser Zahl ist die Summe aller Ziffern, in diesem Beispiel somit `59`. Stellen Sie sicher, dass nur die Ziffern `0` - `9` gezählt werden!

Was sind die Vor- und Nachteile der jeweiligen Funktionen? Gibt es möglicherweise eine goldene Mitte?

**Aufgabe 3: Römische Zahlen**

a) Schreiben Sie eine Funktion, welche eine einzelne römische Ziffer in die entsprechende Dezimalzahl umwandelt. Eine einzelne römische Ziffer soll dabei in einem `char` Typ verpackt sein. Verwenden Sie die folgende Umwandlungstabelle.

```
I = 1  V = 5  X = 10  L = 50  C = 100  D = 500  M = 1000
```

b) Erweitern Sie dieses Programm so, dass auch Kleinbuchstaben erkannt werden.

c) Erweitern Sie das Programm so, dass ein ganzer String römischer Ziffern gelesen werden kann. Addieren Sie dabei die eingelesenen Ziffern und geben Sie das Resultat aus. Testen Sie Ihre Implementation mit der römischen Zahl `MDCCCCLXXXIIII`, welche `1984` ergeben sollte.

d) Betrachten Sie die folgenden römischen Buchstabenkombinationen:

```
IV = 4  IX = 9  XL = 40  XC = 90  CD = 400  CM = 900
```

Erweitern Sie Ihr Programm, so dass diese Kombinationen erkannt werden können. Testen Sie Ihre Implementation mit der römischen Zahl `MCMLXXXIV`, welche ebenfalls 1984 ergeben sollte.

Achtung, diese letzte Aufgabe kann auf den ersten Blick schwierig sein. Tipp: Lesen Sie wie bisher die Ziffern einzeln ein. Speichern Sie dabei die im vorherigen Schleifendurchgang benutzte Ziffer in einer zusätzlichen Variablen und korrigieren Sie in jedem Schleifendurchlauf die Summe, falls eine der oben angegebenen Buchstabenkombinationen auftaucht.