Einführung in die Programmierung

Vorlesungsteil 3
Imperative Kontrollstrukturen

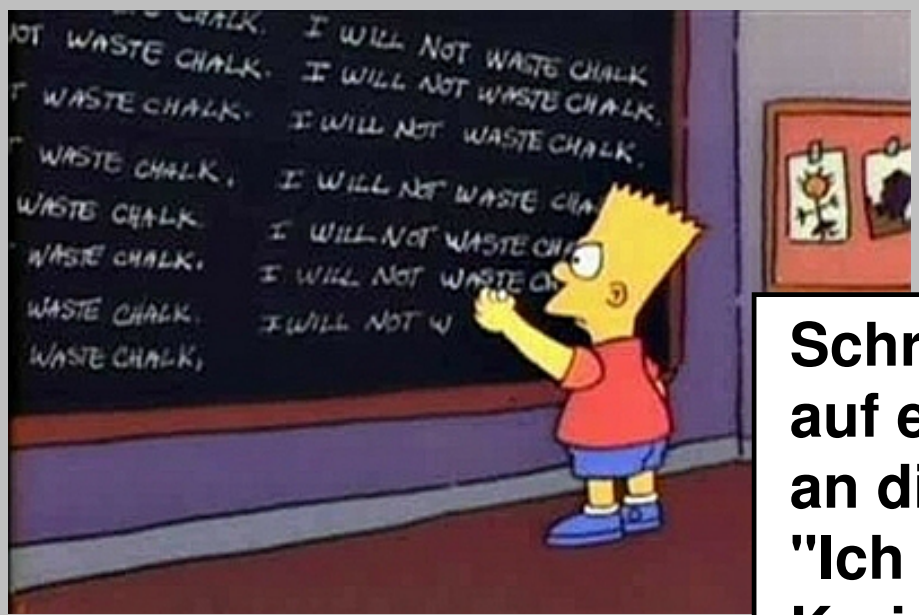
PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Imperativ heißt befehlsorientiert



Bildquelle: www.wikipedia.org

Computer

Programm

Schreibe 20 mal auf englisch an die Tafel: "Ich darf keine Kreide verschwenden!"

Kurze Befehle können viel Ausführungsarbeit nach sich ziehen.

Kontrollstrukturen erschließen die Rechenkraft des Computers

"Der Computer ist unendlich dumm, aber auch unendlich fleissig."

Kontrollstrukturen erschließen die Rechenkraft des Computers

**"Der Computer ist unendlich dumm,
aber auch unendlich fleissig."**

Der Computer kann *gleiche Anweisungsfolgen* (Berechnungsschritte) mit immer wieder anderen oder sich verändernden Werten sehr oft *wiederholen*.

Kontrollstrukturen erschließen die Rechenkraft des Computers

"Der Computer ist unendlich dumm, aber auch unendlich fleissig."

Der Computer kann *gleiche Anweisungsfolgen* (Berechnungsschritte) mit immer wieder anderen oder sich verändernden Werten sehr oft *wiederholen*.

Dadurch wird in der imperativen Programmierung eine *kurze Beschreibung langwieriger Berechnungen* möglich.

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Imperative Kontrollstrukturen

Imperative Kontrollstrukturen sind die Beschreibungsmittel, mit denen der Programmierer im Programmquelltext festlegt, *welche Befehle unter welchen Bedingungen in welcher Reihenfolge* durch den Computer abgearbeitet werden.
Dazu gehören:

Imperative Kontrollstrukturen

Imperative Kontrollstrukturen sind die Beschreibungsmittel, mit denen der Programmierer im Programmquelltext festlegt, *welche Befehle unter welchen Bedingungen in welcher Reihenfolge* durch den Computer abgearbeitet werden.
Dazu gehören:

- **Befehlsfolgen**,
  auch *Anweisungsfolgen*, *Sequenzen* oder *Blöcke* genannt

Imperative Kontrollstrukturen

Imperative Kontrollstrukturen sind die Beschreibungsmittel, mit denen der Programmierer im Programmquelltext festlegt, *welche Befehle unter welchen Bedingungen in welcher Reihenfolge* durch den Computer abgearbeitet werden.
Dazu gehören:

- **Befehlsfolgen**,
  auch *Anweisungsfolgen*, *Sequenzen* oder *Blöcke* genannt
- **Fallunterscheidungen**,
  auch *Verzweigungen* oder *bedingte Anweisungen* genannt

Imperative Kontrollstrukturen

Imperative Kontrollstrukturen sind die Beschreibungsmittel, mit denen der Programmierer im Programmquelltext festlegt, *welche Befehle unter welchen Bedingungen in welcher Reihenfolge* durch den Computer abgearbeitet werden.
Dazu gehören:

- **Befehlsfolgen**,
  auch *Anweisungsfolgen*, *Sequenzen* oder *Blöcke* genannt
- **Fallunterscheidungen**,
  auch *Verzweigungen* oder *bedingte Anweisungen* genannt
- **Schleifen**,
  auch *iterative Konstrukte* (engl. iterate = wiederholen)

Imperative Kontrollstrukturen

Imperative Kontrollstrukturen sind die Beschreibungsmittel, mit denen der Programmierer im Programmquelltext festlegt, *welche Befehle unter welchen Bedingungen in welcher Reihenfolge* durch den Computer abgearbeitet werden.
Dazu gehören:

- **Befehlsfolgen**,
  auch *Anweisungsfolgen*, *Sequenzen* oder *Blöcke* genannt
- **Fallunterscheidungen**,
  auch *Verzweigungen* oder *bedingte Anweisungen* genannt
- **Schleifen**,
  auch *iterative Konstrukte* (engl. iterate = wiederholen)

$\Longrightarrow$ Imperative Kontrollstrukturen *steuern* (engl. control) den Programmablauf.

Alle 32 Schlüsselwörter von C

C als besonders kompakte Programmiersprache

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

im ANSI-Standard C90

ergänzt durch vorbelegte Bezeichner (z.B. `main`), Bibliotheksfunktionen (z.B. `printf`) und Operatoren (z.B. $+$ und $<$)

Alle 32 Schlüsselwörter von C

C als besonders kompakte Programmiersprache

`auto`	`double`	`int`	`struct`
`break`	**`else`**	`long`	**`switch`**
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	**`for`**	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	**`if`**	`static`	**`while`**

im ANSI-Standard C90

ergänzt durch vorbelegte Bezeichner (z.B. `main`), Bibliotheksfunktionen (z.B. `printf`) und Operatoren (z.B. $+$ und $<$)

Block

Ein **Block** beschreibt die *Hintereinanderausführung* (sequentielle Abarbeitung) von Anweisungen. Jeder Block wird durch `{` und `}` begrenzt.

```
{
   Deklarationen; //lokale Variablen und Konstanten
   Anweisung 1;
   Anweisung 2;
   :
   Anweisung k;
}
```

- Hinter schließender Klammer `}` am Blockende kein Semikolon
- Jeder Block gilt von außen betrachtet wie einzelne Anweisung
- Rumpf jeder Funktion ist stets ein Block (vgl. `main`-Funktion)
- Variablen und Konstanten gelten stets innerhalb des Blocks, in welchem sie deklariert wurden

Blockschachtelungen und Lebensdauer von Variablen

```c
#include <stdio.h>

int main(void)
{
    int a = 3;

    {
        int a = 5;
        printf("a: %d\n", a); //a ist 5
    }

    {
        int a = 7;
        printf("a: %d\n", a); //a ist 7
        {
            int a = 9;
            printf("a: %d\n", a); //a ist 9
        }
    }
    printf("a: %d\n", a); //a ist 3
    return 0;
}
```

- Lokale Deklarationen überlagern bei Namensgleichheit stets globalere Deklarationen.
- Sobald die Programmabarbeitung ein Blockende erreicht, endet die Lebensdauer der dort deklarierten Variablen und Konstanten, und ihr Speicherplatz wird freigegeben.

Fallunterscheidungen in der Programmierung

www.webdesign.org

Fallunterscheidung mit `if else`

In Abhängigkeit davon, ob eine vorgegebene *Bedingung* erfüllt ist oder nicht, werden unterschiedliche Blöcke abgearbeitet.

```
if (Bedingung)      //Falls Bedingung erfuellt
  Block 1
else                //sonst
  Block 2
```

Fallunterscheidung mit `if else`

In Abhängigkeit davon, ob eine vorgegebene *Bedingung* erfüllt ist oder nicht, werden unterschiedliche Blöcke abgearbeitet.

```
if (Bedingung)      //Falls Bedingung erfuellt
  Block 1
else                //sonst
  Block 2
```

- **`Bedingung`** ist ein Ausdruck, der einen *Ganzzahlwert* als Ergebnis liefert
- Ist die Bedingung *wahr* (ungleich 0), wird **`Block 1`** abgearbeitet
- Ist die Bedingung *falsch* (gleich 0), wird **`Block 2`** abgearbeitet
- Bedingung stets in runde Klammern **`()`** einbetten
- **`else`**-Zweig darf weggelassen werden, wenn **`Block 2`** leer ist
- Gesamtes **`if`**-**`else`**-Konstrukt nach außen wie einzelner Block aufgefasst

Beispiel: Maximum aus zwei Zahlen

max2.c

```c
#include <stdio.h>

int main(void)
{
  double a, b, max;

  printf("Maximum aus zwei Zahlen\n");
  printf("Erste Zahl: ");
  scanf("%lf", &a);
  printf("Zweite Zahl: ");
  scanf("%lf", &b);

  if (a > b)
  {
    max = a;
  }
  else
  {
    max = b;
  }

  printf("Maximum: %lf\n", max);
  return 0;
}
```

- Besteht der Block im if- oder else-Zweig nur aus einer einzigen Anweisung, dann dürfen die geschweiften Blockklammern weggelassen werden.
- Dies verschlechtert aber die Lesbarkeit des Quelltextes, deshalb sei davon abgeraten.

`if-else` zur Gültigkeits- und Plausibilitätsprüfung

Beispiel: Quadratisches Polynom $x^2 + px + q$ besitzt reelle Nullstellen

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

Berechnung nur dann ausführen, wenn sie zulässig ist

```
if (p*p/4.0 < q)
{
  printf("keine reellen Nullstellen");
}
else
{
  double z = -p/2.0;
  double x1 = z + sqrt(z*z - q);
  double x2 = z - sqrt(z*z - q);
  printf("x1:  %lf, x2:  %lf", x1, x2);
}
```

Bedingungen dürfen Boolesche Operatoren enthalten

Negation

x	1–x
0	1
1	0

```
!x
```

UND

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

```
x && y
```

ODER

x	y	x+y–xy
0	0	0
0	1	1
1	0	1
1	1	1

```
x || y
```

Prioritaet

Bedingungen dürfen Boolesche Operatoren enthalten

```
int esRegnet = 1;
int habeSchirm = 0;
int t = 20; //Temperatur in Grad Celsius

if ((esRegnet) && (!habeSchirm) && (t > 0))
{
  printf("Ich werde nass.");
}
```

- Merke: Vergleiche `==`, `!=`, `<`, `<=`, `>`, `>=` liefern bei wahr den Wert `1`, sonst `0`
- Dies kann unmittelbar zur kompakten Notation arithmetischer Ausdrücke genutzt werden, z.B. `int x = (a > b) * c;`
- Bedingungen bitte stets sauber klammern und den Überblick über die Klammerungen behalten

Mehrfachverzweigungen mit `else if`

zum Auswählen aus mehr als zwei Alternativen:

```
if (Bedingung 1)
  Block 1
else if (Bedingung 2)
  Block 2
else if (Bedingung 3)
  Block 3
:
else if (Bedingung k)
  Block k
else
  Block k+1
```

Mehrfachverzweigungen mit `else if`

zum Auswählen aus mehr als zwei Alternativen:

```
if (Bedingung 1)
   Block 1
else if (Bedingung 2)
   Block 2
else if (Bedingung 3)
   Block 3
:
else if (Bedingung k)
   Block k
else
   Block k+1
```

- Das letzte `else` ist optional und bezieht sich auf `Bedingung k`
- Abarbeitung verfolgt das Ausschließungsprinzip:
  Ist `Bedingung 1` nicht erfüllt, teste `Bedingung 2` usw.

Beispiel: Body-Mass-Index

bmi.c

```c
#include <stdio.h>

int main(void)
{
  double h = 1.77; //Koerpergroesse in m
  double m = 83.1; //Koerpermasse in kg
  double bmi = m / (h * h);

  if (bmi < 18.5) {
    printf("Untergewichtig\n");
  }
  else if (bmi < 25) {
    printf("Normalgewichtig\n");
  }
  else if (bmi < 30) {
    printf("Leicht uebergewichtig\n");
  }
  else {
    printf("Fettleibig\n");
  }
  return 0;
}
```

Fallunterscheidungen mit `switch case`

zum Auswählen aus einem Pool ganzzahliger Werte als Alternativen:

```
switch (Ganzzahlausdruck)
{
  case Konst_1:  Block 1
                 break;        // optional
  case Konst_2:  Block 2
                 break;        // optional
  :
  case Konst_k:  Block k
                 break;        // optional
  default:        Block k+1    // optional
}
```

Fallunterscheidungen mit `switch case`

zum Auswählen aus einem Pool ganzzahliger Werte als Alternativen:

```
switch (Ganzzahlausdruck)
{
  case Konst_1:  Block 1
                 break;          // optional
  case Konst_2:  Block 2
                 break;          // optional
  :
  case Konst_k:  Block k
                 break;          // optional
  default:        Block k+1      // optional
}
```

- Die einzelnen **`break`**-Befehle sowie die **`default`**-Zeile können weggelassen werden
- **`Konst_1`** bis **`Konst_k`** müssen jeweils Konstantenwerte sein, (Formel)ausdrücke sind dort nicht zulässig

Funktionsweise von `switch case`

- Hinter **`switch`** angegebener **`Ganzzahlausdruck`** mit den Konstantenwerten beginnend ab **`Konst_1`** auf *Gleichheit* verglichen

Funktionsweise von `switch case`

- Hinter **`switch`** angegebener **`Ganzzahlausdruck`** mit den Konstantenwerten beginnend ab **`Konst_1`** auf *Gleichheit* verglichen
- Sobald erstmalig Übereinstimmung gefunden (*„Match“*), von dort an ALLE weiteren Blöcke bis zum nächsten **`break`** oder Ende des **`switch`**-Konstrukts ausgeführt (*„Durchfalleffekt“*)

Funktionsweise von `switch case`

- Hinter **`switch`** angegebener **`Ganzzahlausdruck`** mit den Konstantenwerten beginnend ab **`Konst_1`** auf *Gleichheit* verglichen
- Sobald erstmalig Übereinstimmung gefunden (*„Match“*), von dort an ALLE weiteren Blöcke bis zum nächsten **`break`** oder Ende des **`switch`**-Konstrukts ausgeführt (*„Durchfalleffekt“*)
- Wenn keinerlei Übereinstimmung, Block des optionalen **`default`**-Falls ausgeführt

Beispiel: Würfel mit Pünktchenausgabe

wuerfel.c

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

int main(void)
{
  int augenzahl;

  srand(time(NULL));
  augenzahl = rand() % 6 + 1;
  switch (augenzahl)
  {
    case 1: printf("\n     \n     \n  *  \n     \n     \n\n"); break;
    case 2: printf("\n    *\n     \n     \n*    \n\n"); break;
    case 3: printf("\n    *\n     \n  *  \n*    \n\n"); break;
    case 4: printf("\n*   *\n     \n     \n*   *\n\n"); break;
    case 5: printf("\n*   *\n     \n  *  \n*   *\n\n"); break;
    case 6: printf("\n*   *\n     \n*   *\n*   *\n\n"); break;
    default : printf("\n\n");
  }
  return 0;
}
```

Den „Durchfall“ ohne `break` geschickt nutzen

tagepromonat2015.c – Monatslängen in Tagen 2015

```c
#include <stdio.h>

int main(void)
{
  int monat = 11; //1: Januar, 2: Februar, ..., 12: Dezember
  int anzTage;

  switch(monat)
  {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: anzTage = 31; break;

    case 4:
    case 6:
    case 9:
    case 11: anzTage = 30; break;

    case 2: anzTage = 28; break;
    default: anzTage = -1; //Fehlerfall
  }
  printf("Anzahl Tage: %d\n", anzTage);
  return 0;
}
```

Gegenüberstellung `if-else` und `switch-case`

Für welche Einsatzszenarien empfiehlt sich welches Konstrukt?

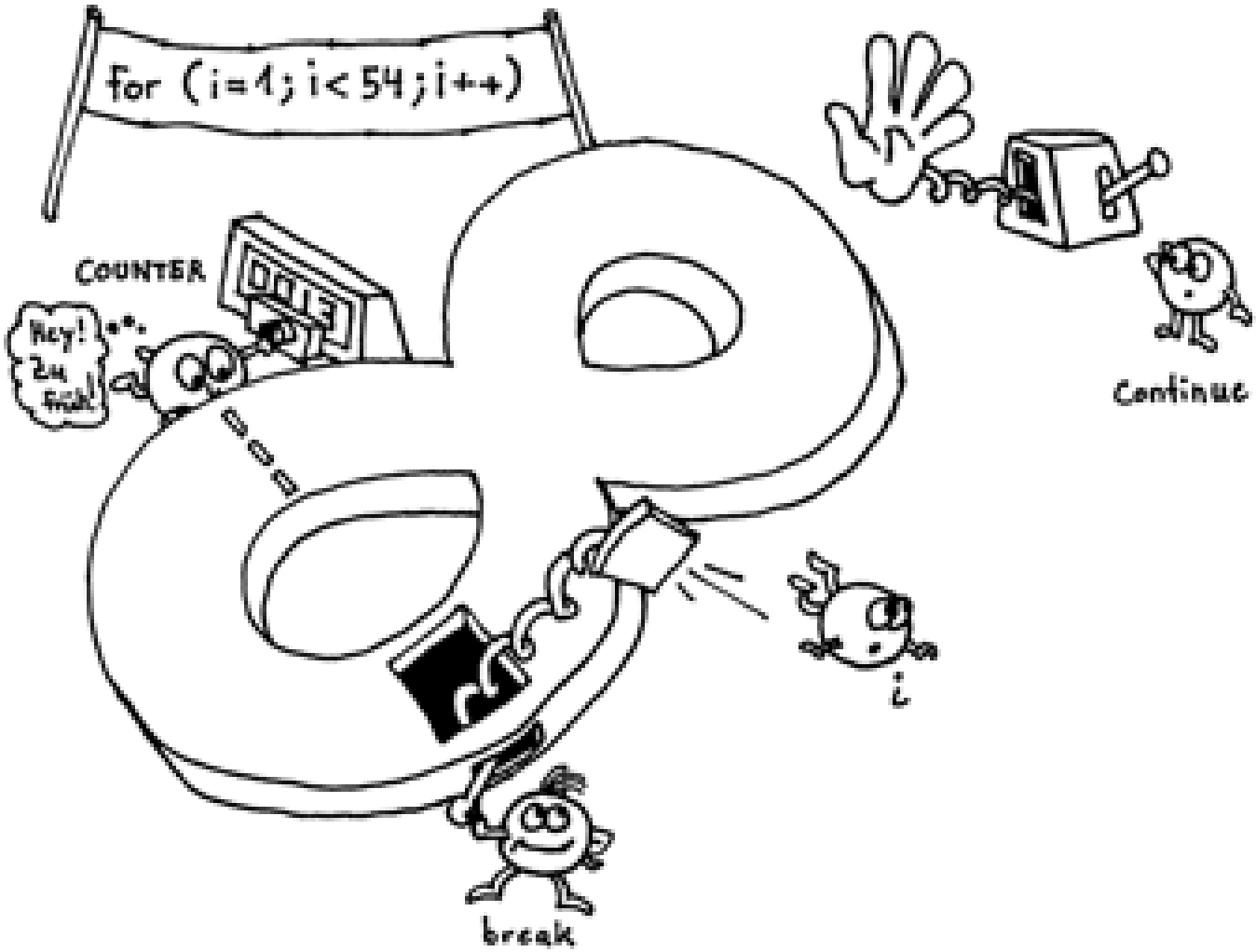
`if-else`

- flexibler
- < und > in Bedingungen zulässig, somit Intervalle beschreibbar
- Vergleiche nicht auf Ganzzahlen eingeschränkt
- Quelltext schwerfälliger und schreibaufwendiger
- compilertechnisch weniger optimierbar

`switch-case`

- im Maschinenprogramm nach Compilierung oft schneller ausführbar
- übersichtlicher im Quelltext und damit geringere Wahrscheinlichkeit für Programmierfehler (wie vergessene Fallalternativen)
- durch recht starre Strukturvorgabe weniger beschreibungsmächtig

Schleifen in der Programmierung



openbook.galileocomputing.de

Schleifen in der Programmierung

- Eine *Schleife* dient dazu, einen Block von Anweisungen *wiederholt* abzuarbeiten.

Schleifen in der Programmierung

- Eine *Schleife* dient dazu, einen Block von Anweisungen *wiederholt* abzuarbeiten.
- Mit jedem *Schleifendurchlauf* (jeder *Iteration*) können sich innerhalb des Blocks Variablenwerte verändern

Schleifen in der Programmierung

- Eine *Schleife* dient dazu, einen Block von Anweisungen *wiederholt* abzuarbeiten.
- Mit jedem *Schleifendurchlauf* (jeder *Iteration*) können sich innerhalb des Blocks Variablenwerte verändern
- Vor oder nach jedem Durchlauf (je nach Schleifenart) wird eine frei definierbare *Bedingung* geprüft.

Schleifen in der Programmierung

- Eine *Schleife* dient dazu, einen Block von Anweisungen *wiederholt* abzuarbeiten.
- Mit jedem *Schleifendurchlauf* (jeder *Iteration*) können sich innerhalb des Blocks Variablenwerte verändern
- Vor oder nach jedem Durchlauf (je nach Schleifenart) wird eine frei definierbare *Bedingung* geprüft.
- Ist die Bedingung erfüllt (ungleich 0), wird die Schleife (erneut) durchlaufen.

Schleifen in der Programmierung

- Eine *Schleife* dient dazu, einen Block von Anweisungen *wiederholt* abzuarbeiten.
- Mit jedem *Schleifendurchlauf* (jeder *Iteration*) können sich innerhalb des Blocks Variablenwerte verändern
- Vor oder nach jedem Durchlauf (je nach Schleifenart) wird eine frei definierbare *Bedingung* geprüft.
- Ist die Bedingung erfüllt (ungleich 0), wird die Schleife (erneut) durchlaufen.
- Ist sie nicht erfüllt (gleich 0), endet die Abarbeitung der Schleife und das Programm wird mit dem nächsten Befehl unmittelbar hinter dem Block der Schleife fortgesetzt.

Schleifen in der Programmierung

- Eine *Schleife* dient dazu, einen Block von Anweisungen *wiederholt* abzuarbeiten.
- Mit jedem *Schleifendurchlauf* (jeder *Iteration*) können sich innerhalb des Blocks Variablenwerte verändern
- Vor oder nach jedem Durchlauf (je nach Schleifenart) wird eine frei definierbare *Bedingung* geprüft.
- Ist die Bedingung erfüllt (ungleich 0), wird die Schleife (erneut) durchlaufen.
- Ist sie nicht erfüllt (gleich 0), endet die Abarbeitung der Schleife und das Programm wird mit dem nächsten Befehl unmittelbar hinter dem Block der Schleife fortgesetzt.
- Schleifen können auch unendlich oft durchlaufen werden (*Endlosschleife*), dann ist das Programm abgestürzt und kann per Tastatur durch *Strg-C* abgebrochen werden. Endlosschleifen sollten vermieden werden.

Schleifen in der Programmierung

- Eine *Schleife* dient dazu, einen Block von Anweisungen *wiederholt* abzuarbeiten.
- Mit jedem *Schleifendurchlauf* (jeder *Iteration*) können sich innerhalb des Blocks Variablenwerte verändern
- Vor oder nach jedem Durchlauf (je nach Schleifenart) wird eine frei definierbare *Bedingung* geprüft.
- Ist die Bedingung erfüllt (ungleich 0), wird die Schleife (erneut) durchlaufen.
- Ist sie nicht erfüllt (gleich 0), endet die Abarbeitung der Schleife und das Programm wird mit dem nächsten Befehl unmittelbar hinter dem Block der Schleife fortgesetzt.
- Schleifen können auch unendlich oft durchlaufen werden (*Endlosschleife*), dann ist das Programm abgestürzt und kann per Tastatur durch *Strg-C* abgebrochen werden. Endlosschleifen sollten vermieden werden.
- Schleifenarten: **`while`**, **`do while`** und **`for`**

`while`-Schleife

Prüfung der Bedingung **vor** jedem Schleifendurchlauf

```
while (Bedingung)
  Block
```

- Solange (engl. while) `(Bedingung)` erfüllt, führe `Block` aus
- Ist `(Bedingung)` beim ersten Test nicht erfüllt, wird `Block` gar nicht ausgeführt
- kopfgesteuerte Schleife

Beispiel für eine `while`-Schleife (pi-reihe.c)

Annäherung von π durch endlichen Teil der Reihe $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$

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

int main(void)
{
  int k = 1;
  int n = 20; //Anzahl Summanden
  double x = 0.0;

  while (k <= n)
  {
    x = x + 1.0 / (k*k);
    printf("Bei k=%d:  %lf\n", k, sqrt(6*x));
    k = k + 1;
  }
  return 0;
}
```

Schleifenabbruch mit **break** (pi-reihe2.c)

```c
#include <stdio.h>
#include <math.h>
#define EPSILON 1e-5

int main(void)
{
  int k = 1;
  double y = 0.0, x = 0.0;

  while (1)
  {
    y = x;
    x = x + 1.0 / (k*k);
    if ((x-y)/x < EPSILON*EPSILON)
    {
      break;
    }
    printf("Bei k = %d ist pi %lf\n", k, sqrt(6*x));
    k = k + 1;
  }
  return 0;
}
```

- **while (1) {...}** definiert eine *Endlosschleife*
- Bei Erreichen des Schlüsselwortes **break** sofortiger Schleifenabbruch

Vorzeitige Rückkehr zur Bedingungsprüfung (continue.c)

```c
#include <stdio.h>

int main(void)
{
  int i = 1;

  while (i < 40)
  {
    if (i % 8 == 0)
    {
      i = i + 7;
      continue;
    }
    i = i + 1;
    printf("i ist %d\n", i);
  }
  return 0;
}
```

```
i ist 2
i ist 3
i ist 4
i ist 5
i ist 6
i ist 7
i ist 8
i ist 16
i ist 24
i ist 32
i ist 40
```

- Bei Erreichen des Schlüsselwortes **`continue`** sofortiges Verlassen des Schleifenblocks und Rückkehr zur Bedingungsprüfung
- **`continue`** ermöglicht mitunter kompakte Notation, führt aber oft zu schwer verständlichem Quelltext

`do while`-Schleife

Prüfung der Bedingung **nach** jedem Schleifendurchlauf

```
do
   Block
while (Bedingung);
```

- Führe `Block` aus, solange `(Bedingung)` erfüllt.
- `Block` wird mindestens einmal durchlaufen
- fußgesteuerte Schleife

`do while`-Schleife

Prüfung der Bedingung **nach** jedem Schleifendurchlauf

```
do
   Block
while (Bedingung);
```

- Führe `Block` aus, solange `(Bedingung)` erfüllt.
- `Block` wird mindestens einmal durchlaufen
- fußgesteuerte Schleife
- kann als Endlosschleife fungieren: `do {...} while (1);`

`do while`-Schleife

Prüfung der Bedingung **nach** jedem Schleifendurchlauf

```
do
   Block
while (Bedingung);
```

- Führe `Block` aus, solange `(Bedingung)` erfüllt.
- `Block` wird mindestens einmal durchlaufen
- fußgesteuerte Schleife
- kann als Endlosschleife fungieren: `do {...} while (1);`
- `break` und `continue` erlaubt mit gleicher Wirkung wie in `while`-Schleife

Beispiel: Wiederhole Eingabe, solange ungültig

age.c

```c
#include <stdio.h>

int main(void)
{
  int age;

  do
  {
    printf("Bitte geben Sie Ihr Alter in Jahren ein: ");
    scanf("%d", &age);
  } while ((age < 0) || (age > 130));
  return 0;
}
```

`for`-Schleife: Intention einer Zählschleife

Prüfung der Bedingung **vor** jedem Schleifendurchlauf

```
for (Initialisierung; Bedingung; Zaehlschritt)
  Block
```

1. Zuerst Anweisung `Initialisierung` ausgeführt
2. Danach `Bedingung` geprüft. Falls nicht erfüllt (gleich 0): Schleifenabarbeitung beendet
3. Ansonsten Abarbeitung der Anweisungen im `Block`
4. Anschließend Abarbeitung der Anweisung im `Zaehlschritt`
5. Weiter mit 2.

`for`-Schleife als Zählschleife (for10.c)

```c
#include <stdio.h>

int main(void)
{
  int i;

  for (i = 1; i <= 10; i++)
  {
    printf("i ist %d\n", i);
  }
  return 0;
}
```

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

`for`-Schleife als Zählschleife (for10.c)

```c
#include <stdio.h>

int main(void)
{
  int i;

  for (i = 1; i <= 10; i++)
  {
    printf("i ist %d\n", i);
  }
  return 0;
}
```

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

- kopfgesteuerte Schleife
- Komponenten **`Initialisierung`**, **`Bedingung`** und/oder **`Zaehlschritt`** dürfen auch leergelassen werden
- Endlosschleife: **`for(;;) {...}`**
- **`break`** und **`continue`** erlaubt mit gleicher Wirkung wie in **`while`**-Schleife

Einfacher Pseudozufallszahlengenerator

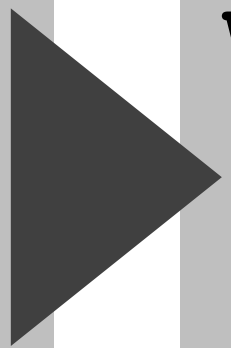
pseudozfg.c

```c
#include <stdio.h>

int main(void)
{
  int i;
  int x = 5; //Startwert

  for (i = 1; i <= 6; i++)
  {
    x = (4*(x + 7)) % 15;
    printf("Pseudozufallszahl: %d\n", x);
  }
  return 0;
}
```

$\Longrightarrow$ Periodenlänge: 6, dann wiederholt sich Zahlenfolge

Simulation einer `for`-Schleife als `while`-Schleife

```
for ( A; B; C)
{
  D;
}
```



```
A;
while (B)
{
  D;
  C;
}
```

$\Longrightarrow$ Alle drei Schleifenarten **`while`**, **`do while`** und **`for`** besitzen *gleiche Ausdrucksmächtigkeit*, können sich also beliebig gegenseitig simulieren. Eigentlich würde zum Programmieren eine beliebige der drei Schleifenarten ausreichen.

Gegenüberstellung `while`, `do while` und `for`

Wofür empfiehlt sich welche Schleifenart?

`while`	`do while`	`for`
• Schleifen, die u.U. gar nicht durchlaufen werden • Bedingung muss vor Schleifendurch-lauf auswertbar sein • numerische Berechnungen, bei denen mit jeder Iteration Werte aktualisiert werden	• Schleifen, die mindestens einmal durchlaufen werden müssen • Bedingung muss erst nach Schleifendurchlauf auswertbar sein • Eingaben, die solange wiederholt werden, bis gültige Werte vorliegen	• Zählschleife mit bekannter Anzahl Durchläufe • schrittweises / elementweises Durchlaufen von Zahlenfolgen a_i über einen Index i • Aufzählungen

Ineinanderschachteln mehrerer Schleifen

Beispiel: Alle dreibuchstabigen Wörter aus den Zeichen A, B und C (abc.c)

```c
#include <stdio.h>

int main(void)
{
  char x, y, z;

  for (x = 1; x <= 3; x++)
  {
    for (y = 1; y <= 3; y++)
    {
      for (z = 1; z <= 3; z++)
      {
        printf("%c%c%c\n", x+64, y+64, z+64);
      }
    }
  }
  return 0;
}
```

```
AAA
AAB
AAC
ABA
ABB
ABC
ACA
ACB
ACC
BAA
BAB
BAC
BBA
BBB
BBC
BCA
BCB
BCC
CAA
CAB
CAC
CBA
CBB
CBC
CCA
CCB
CCC
```

$\Longrightarrow$ insgesamt $3 \cdot 3 \cdot 3 = 27$ Kombinationen erzeugt