Programmierung 1

Funktionen I

Dr. Monika Schak

Woche 5

19. November 2025

Wiederholung

- 7 primitive Datentypen in C:
  - `char` für ASCII-Zeichen
  - `short`, `int`, `long` für Ganzzahlen
  - `float`, `double` für Gleitkommazahlen, Darstellung über Vorzeichen, Mantisse und Exponent (Achtung Rundungsfehler!)
  - `bool` für Wahrheitswerte
- Konstanten mit `#define` oder `const`
- Definition neuer Typnamen über `typedef`
- Auswertungsreihenfolge emtsprechend **Assoziativität** und **Präzedenz**

Funktionen

- Kleine Unterprogramme, die ein Teilproblem einer Aufgabe lösen können.
- Grundsatz: **Don’t repeat yourself**
- Vorteile: Bessere Lesbarkeit, Wiederverwendbarkeit, Zeitersparnis
- Ermöglichen Zusammenarbeit!

Funktionen

- Bisher schon einige Funktionen kennen gelernt und verwendet, z.B. Funktionen für Ein- und Ausgabe und die `main()`-Funktion
- `main()` kommt genau einmal im C-Programm vor und kann als sogenanntes Hauptprogramm betrachtet werden

```
int main (int argc, char* argv[]) {
  return 0; // kann nach C99 entfallen
}
```

- `argc` → Zähler für Kommandozeilen-Argumente
- `argv[]` → Parameter als String interpretiert (Feld mit Zeigern auf C-Strings)
- `argc` und `argv[]` können weggelassen werden, wenn sie nicht benötigt werden

Funktionsaufbau in C

Rückgabetyp `Funktionsname` (Parameterliste) {
  /* Anweisungsblock mit Anweisungen = Rumpf*/
}

- Bei void gibt es keinen Rückgabewert
- Parameterliste: Es gibt 0, 1 oder mehr Parameter, die durch ein Komma getrennt sind. Jeder Parameter wird mit Typ und Name angegeben
- Den Funktionskopf mit Rückgabetyp, Funktionsname und Parameterliste nennt man Signatur
- Eine return-Anweisung beendet die Ausführung einer Funktion mit einem Rückgabewert, der zum Rückgabetyp passen muss. Ist der Rückgabetyp void, ist keine return-Anweisung nötig.

Beispiel: BMIs berechnen

```c
#include <stdio.h>

int main () {
  float gewichtA, groesseA, bmiA;

  printf("----- Person A ------\n");
  printf("Gewicht eingeben (in kg): ");
  scanf("%f", &gewichtA);

  printf("Größe eingeben (in m): ");
  scanf("%f", &groesseA);

  bmiA = gewichtA / (groesseA * groesseA);
  printf("BMI: %.1f\n\n", bmiA);

  float gewichtB, groesseB, bmiB;

  printf("----- Person B ------\n");
  printf("Gewicht eingeben (in kg): ");
  scanf("%f", &gewichtB);

  printf("Größe eingeben (in m): ");
  scanf("%f", &groesseB);

  bmiB = gewichtB / (groesseB * groesseB);
  printf("BMI: %.1f\n\n", bmiB);

  float gewichtC, groesseC, bmiC;

  printf("----- Person C ------\n");
  printf("Gewicht eingeben (in kg): ");
  scanf("%f", &gewichtC);

  printf("Größe eingeben (in m): ");
  scanf("%f", &groesseC);

  bmiC = gewichtC / (groesseC * groesseC);
  printf("BMI: %.1f\n\n", bmiC);

  return 0;
}
```

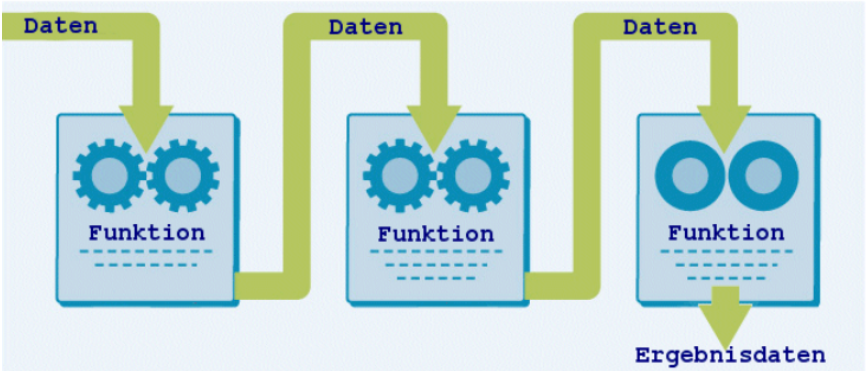
Strukturierte Programmierung

- Schrittweises Verfeinern: Top-Down-Ansatz
- Ziel: Programmiersprache soll uns beim Aufschreiben komplexerer Lösungen entgegen kommen
- Lösungsansatz zum vorigen Beispiel:

Strukturierte Programmierung

- Modularisierung und Strukturierung
  - Verbergen von Implementierungsdetails
  - Lokale Änderungen haben **lokale Auswirkungen**
- Schnittstellen-Spezifikation
  - Wie soll sich der Algorithmus *nach außen* verhalten?
  - Formulierung von *Vor- und Nachbedingungen* an Startwerte und Ergebnisse

Typischer Programmablauf



Definition von Funktionen

```
3  float berechneBMI (char person) {
4    float gewicht, groesse, bmi;
5
6    printf("----- Person %c ------\n", person);
7    printf("Gewicht eingeben (in kg): ");
8    scanf("%f", &gewicht);
9
10   printf("Größe eingeben (in m): ");
11   scanf("%f", &groesse);
12
13   bmi = gewicht / (groesse * groesse);
14   printf("BMI: %.1f\n\n", bmi);
15
16   return bmi;
17 }
```

```
19 int main () {
20   float bmiA, bmiB, bmiC;
21
22   bmiA = berechneBMI('A');
23   bmiB = berechneBMI('B');
24   bmiC = berechneBMI('C');
25
26   return 0;
27 }
```

- Beim Aufschreiben der Funktion sind die tatsächlichen Variablen (bzw. Werte) noch nicht bekannt, mit denen sie später aufgerufen wird
- Aufruf (und Ausführung) der Funktion ist zu unterscheiden von der Definition, was sie grundsätzlich tun soll

Ablauf bei Verwendung

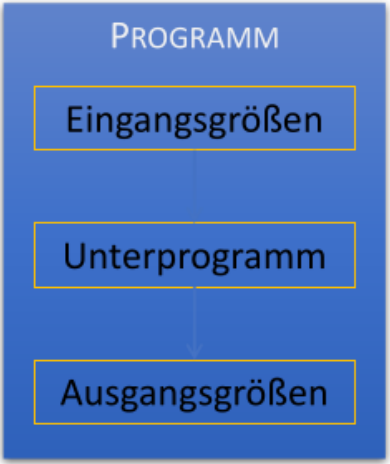
```
float berechneBMI (char person) {
  float gewicht, groesse, bmi;

  // ...

  return bmi;
}
```

- `float` ist der Typ des Rückgabewerts
- `char person` ist der formale Parameter

```
float bmiA;
bmiA = berechneBMI('A');
```

- `'A'` ist der aktuelle Parameter
- Der Rückgabewert der Funktion wird dann in `bmiA` gespeichert

Schnittstellen



- Zwischen Funktionen und Umgebung muss definierte Schnittstelle existieren:
  - Übergabe der Eingansdaten
  - Rückgabe des Ergebnisses
- Vorbedingung? Keine.
- Nachbedingung? Es gilt: $BMI = \frac{\text{Gewicht}}{\text{Größe}^2}$ mit den eingegebenen Werten.

Sichtbarkeit von Variablen

- Variablen, die innerhalb einer Funktion oder einem Block deklariert werden, heißen **lokale Variablen**
- Andere Funktionen können nicht darauf zugreifen!
- Die Lebensdauer lokaler Variablen ist auf die Dauer der Ausführung des Anweisungsblocks beschränkt
- Formale Parameter verhalten sich wie lokale Variablen.

Sichtbarkeit von Variablen

- Variablen (auch formale Parameter) sind nur in dem Anweisungsblock sichtbar (d.h. verwendbar), in dem sie deklariert wurden
  $\rightarrow$ Formale Parameter sind also nur im Rumpf der sie deklarierenden Funktion sichtbar
- Variablen haben einen Gültigkeitsbereich, einen sog. Scope
  - **Lokal** (zur Umgebung, die die Variable umgibt), d.h. nach dem Verlassen von Rumpf/Anweisungsblock endet die Lebensdauer.
  - Deklariert man in einer Funktion eine Variable mit dem Spezifizierer `static`, bleibt der Wert nach dem Funktionsende erhalten (schlechter Stil!)
  - **Global** (außerhalb aller Funktionen deklariert): Durchkreuzt leider das Konzept modularer Programmierung

Lokale Variablen

```
int main () {
  int i = 333;
  if (i == 333) {
    int i = 111;
    printf("%d\n", i);
  }
  printf("%d\n", i);
}
```

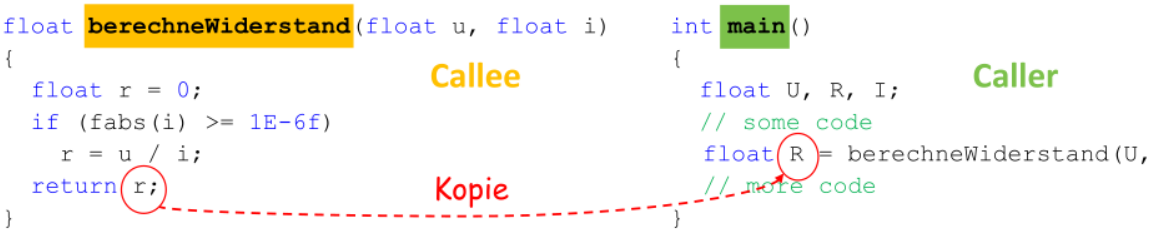
```
int main () {
  int i = 333;
  if (i == 333) {
    i = 111;
    printf("%d\n", i);
  }
  printf("%d\n", i);
}
```

Funktionsaufruf

- Funktionen werden aufgerufen mit Namen und Liste von Parametern (Argumenten)
- Argumente müssen in Datentyp, Reihenfolge und Anzahl der Signatur entsprechen
- Alle Argumente (aktuelle Parameter) werden von links nach rechts ausgewertet und an die formalen Parameter gebunden
- Die Variablenwerte werden dabei kopiert!

Beenden durch return

- Wird in aufgerufener Funktion (Callee) eine return-Anweisung ausgeführt, wird der entsprechende Ausdruck zurückgegeben
- In der aufrufenden Funktion (Caller) wird der Aufruf durch den Rückgabewert ersetzt
- Die Kontrolle geht dabei an den Caller zurück
- In jeder nicht-void-Funktion muss es mindestens eine (ggfs. auch mehrere) return Anweisung(en) geben



Definition vs Deklaration

- Quellcode wird beim Kompilieren „von oben nach unten“ durchgegangen
- Alles, was verwendet werden soll, muss vorher durch Definition oder zumindest Deklaration dem Compiler bekannt gemacht werden

Zuerst deklariert:

```
void hilfe ();

int main () {
  hilfe();
  return 0;
}

void hilfe () {
  printf("Test\n");
}
```

Zuerst definiert:

```
void hilfe () {
  printf("Test\n");
}

int main () {
  hilfe();
  return 0;
}
```

Definition vs. Deklaration

- Beispiel Definition:

```
int func (int a, int b) {
  return (a + b) * (a + b);
}
```

  Eigentliche Funktion, Implementierung meist nach main() bzw. in eigener Quelldatei
- Beispiel Deklaration:

```
int func (int a, int b);
```

  Funktionsprototyp, Angabe vor main() bzw. in Header-File
- Reihenfolge von Deklaration, Definition und Funktionsaufruf wichtig
- Zusammengehörige Deklarationen meist in Header-File zusammengefasst
- Funktion wird vor erstem Aufruf definiert oder als Funktionsprototyp vorher deklariert (*foward declaration*) und später, also nach main() oder in anderer Quelldatei, definiert

Vordefinierte Funktionen

- Mathematische Funktionen, z.B. sind deklariert in `<math.h>`

```
double sin (double x);                  double exp (double x);
double cos (double x);                  double log (double x);
double tan (double x);                  double log10 (double x);
double asin (double x);                 double log2 (double x);
double acos (double x);                 double sqrt (double x);
double atan (double x);                 double ceil (double x);
double atan2 (double y, double x);      double floor (double x);
double pow (double x, double y);        double fabs (double x);
```

  Hinweis: Winkel werden alle im Bogenmaß (Radiant) übergeben

- Weitere wichtige Funktionen finden sich u.a. in `<stdlib.h>` ($\rightarrow$ später), z.B. Absolutbetrag für Ganzzahlen (`int abs(int n);`)

Unterprogramme

- Funktionen können beim Aufruf mit Werten versorgt werden: Aktuelle Parameter (Variablen oder Werte, die beim Funktionsaufruf übergeben werden) müssen in Anzahl, Typ und Reihenfolge mit formalen Parametern übereinstimmen
- Die formalen Parameter verhalten sich innerhalb des Funktionsrumpfes wie Variablen. Außerhalb des Rumpfes sind Parameter nicht sichtbar (können nicht verwendet werden)
- Funktionen werden manchmal auch Unterprogramme genannt. Zudem unterscheidet man manchmal zwischen den Begriffen Funktion sowie Prozedur
- Funktionen liefern Rückgabewerte bestimmten Typs, können daher in Ausdrücken stehen
- Prozeduren haben keine Rückgabewerte (Rückgabetyp void), werden bei Aufruf behandelt wie Anweisungen