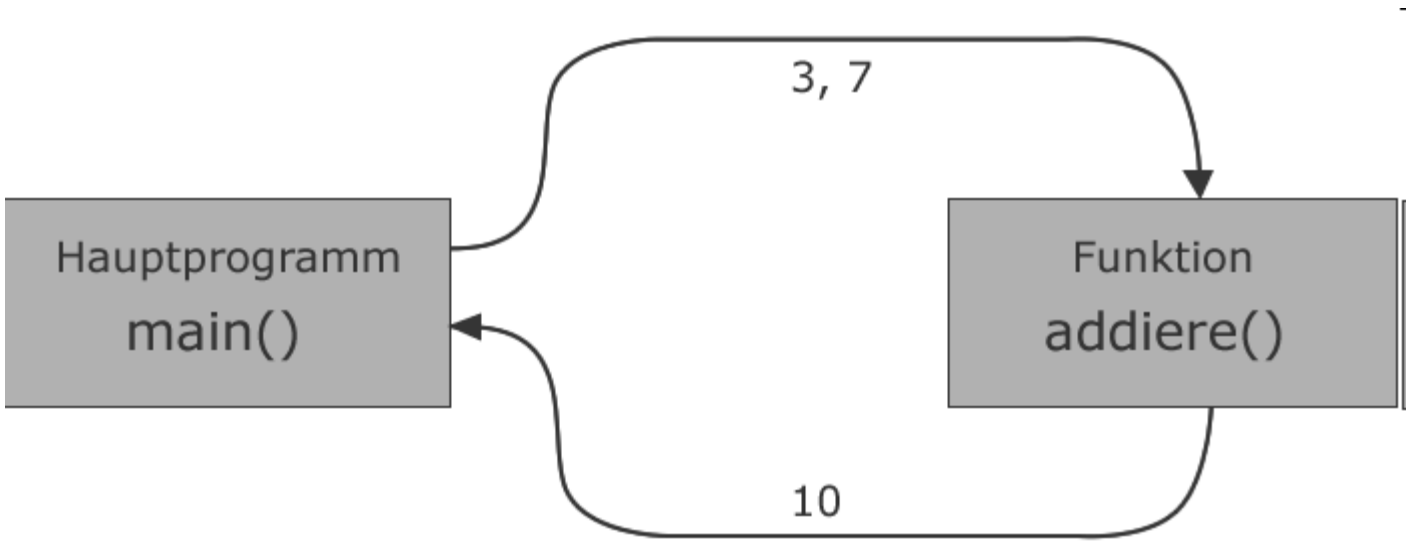
# Konzepte des prozeduralen Programmierens

Stand September 2022


- Prof. Dr. Oliver S. Lazar / Christian Frank

# 4 Funktionen

- **Divide and conquer (Teile und herrsche)**
  - große Probleme werden in mehrere Teilprobleme zerlegt
  - Teilprobleme werden in Funktionen gelöst
    - *führt zu besserer Übersichtlichkeit*



## ❑ Wozu dienen Funktionen?

Funktionen haben eine Menge Vorteile. Einige der wichtigsten sind:

- ❑ Mit Funktionen lässt sich der Quellcode besser lesen.
- ❑ Der Code kann durch Erstellen einer Funktionsbibliothek wiederverwertet werden.
- ❑ Ständig sich wiederholende Routinen können in eine Funktion gepackt werden und müssen nicht immer wieder neu geschrieben werden.
- ❑ Fehler und Veränderungen lassen sich daher auch schneller finden bzw. ausbessern, da der Code nur an einer Stelle bearbeitet werden muss.

# Eigenschaften

- **Eine Funktion hat folgende Eigenschaften:**
  - Funktionsname, ein Name unter der sie ansprechbar ist, z.B. `addiere()`
  - Parameternamen, z.B. `summand1, summand2`
  - Datentyp der Parameter, z.B. `int`
  - Datentyp des Rückgabewertes, z.B. `int`
    - *Rückgabe erfolgt mittels* `return`
  - Anweisungsblock

```
#include<stdio.h>

int addiere(int summand1, int summand2) {
    return (summand1 + summand2);
}

int main() {
    int summe = addiere(3, 7);
    printf("Summe von 3 und 7 ist %d\n", summe);
    return 0;
}
```

```
Summe von 3 und 7 ist 10
```

# Funktionen

- **In C hat jede Funktion ihren eigenen lokalen Bereich**
  - unterschiedliche Funktionen können gleiche Variablennamen verwenden
  - z.B. `int x` (siehe Codebeispiel)
  - `void` zeigt an, dass diese Funktion keinen Wert zurückliefert

```
#include<stdio.h>

void loesche_bild(){
    int x;
    for(x=1; x<=25; x++)
        printf("\n");
}

int main() {
    int x = 7;
    …
    return 0;
}
```

## Funktionen

- übergebene Parameter sind nur innerhalb der Funktion änderbar (wie lokale nicht-static Variablen - siehe später)
  - keine „bleibende“ Wertänderung nach außen
  - *call by value*

- „bleibende“ Wertänderungen für
  - globale Variablen (siehe später)
  - lokale static-Variablen (siehe später)
  - per Zeiger übergebene Variablen (siehe später)
    - *call by reference*

- Rückgabe (maximal) eines Wertes nach außen

→ return

# Funktionen

- Beispiel mit if/else

```
int max(int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```

- Beispiel mit char

```
#include<stdio.h>

char abc(int nummer) {
    return (64 + nummer);
}

int main() {
    printf(" 1. Alphabet-Buchstabe: %c\n", abc(1));
    printf("12. Alphabet-Buchstabe: %c\n", abc(12));
    printf("24. Alphabet-Buchstabe: %c\n", abc(24));
    return 0;
}
```

Laut ASCII-Tabelle beginnen die Großbuchstaben ab dem ASCII-Code 65.

```
1. Alphabet-Buchstabe: A
12. Alphabet-Buchstabe: L
24. Alphabet-Buchstabe: X
```

Klausur WS_17_18_Nachschreib Aufg. 7a)

## Einbinden von Funktionen der Laufzeitbibliothek

- z.B. `printf()` oder `scanf()` aus `<stdio.h>`
- diese Funktionen werden als Objektcode im Compiler mitgeliefert
  - müssen allerdings trotzdem dem Compiler zuvor bekannt gemacht werden
    → Headerdateien

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

# Aufgabe 04.01

## Multiplikationstaschenrechner

Implementiere einen Multiplikationstaschenrechner.

Folgende Programmaktionen treten auf:

- Benutzer auffordern Eingaben zu tätigen
- Eingabe von Zahl 1
- Eingabe von Zahl 2
- Ergebnis ausrechnen (Zahl 1 * Zahl 2)
- Ergebnis ausgeben

Implementiere dabei drei Funktionen:

- `float eingabeZahl()` → für die Eingabe einer Zahl durch den Benutzer
- `float multipliziere(float zahl1, float zahl2)` → multipliziert zwei Zahlen
- `void ausgabeErgebnis(float ergebnis)` → gibt das Ergebnis aus

Diese drei Funktionen stehen oberhalb der `main`-Funktion. Rufe diese Methoden aus der `main`-Methode heraus auf.

# Funktionen

- Das folgende Programm ist fehlerhaft!
- Warum?

```
#include<stdio.h>

int main() {
    float x=4711.0, y=11.0;
    printf("\nErgebnis = %f ", func(x,y));

    return 0;
}

float func(float x, float y){
    return (y/x);
}
```

# Prototypen

- **Prototypen**
  - Funktionen müssen dem Compiler bekannt gemacht werden, bevor sie verwendet werden können
    - *bisher: Funktionen wurden über dem Hauptprogramm platziert*
  - Funktionsumfang kann schnell zunehmen
    - *um zum Hauptprogramm zu gelangen, muss man immer ganz nach unten scrollen*

**Lösung:**

- Funktions-Prototypen werden dem Hauptprogramm vorangestellt
  - enthalten lediglich das Gerüst/ die Signatur der Funktion
  - Funktion ist dem Compiler so bereits frühzeitig bekannt
- der Funktionskörper kann dann an beliebiger Stelle im Code sein
- Werden der Rückgabetyp oder die Argumente weggelassen, so wird bei ANSI C **int** automatisch eingesetzt.

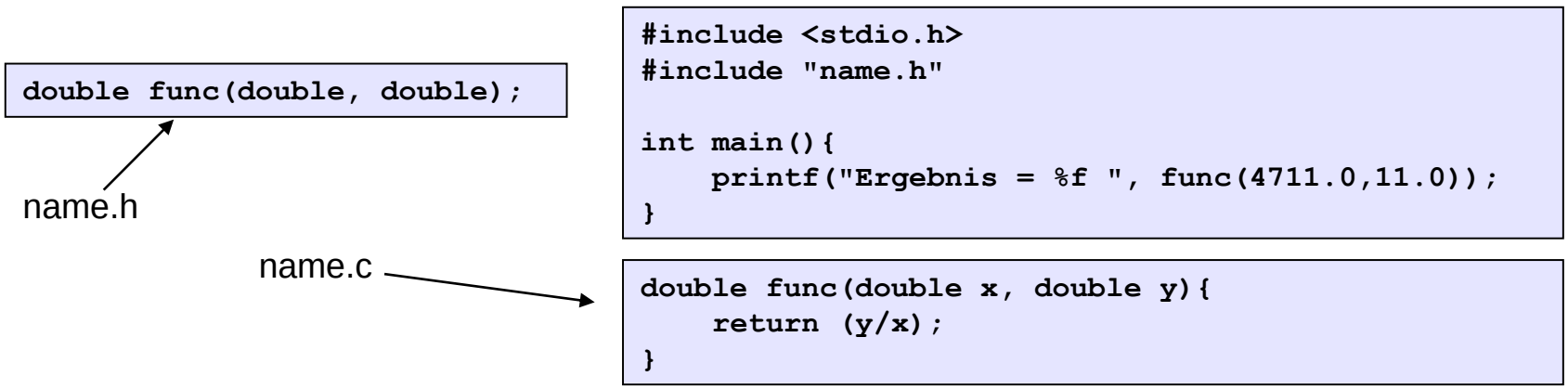
# Prototypen

## ❑ Prototypen

```
#include<stdio.h>

// Funktions-Prototypen
int eingabeZahl();
void ausgabeErgebnis(int ergebnis);

// Hauptprogramm
int main() { // Rechenvorgang
    ausgabeErgebnis(eingabeZahl()*2);
    return 0;
}

// Funktionen
int eingabeZahl() {
    int eingabe;
    printf("\nEingabe Zahl: ");
    scanf("%i", &eingabe);
    return eingabe;
}

void ausgabeErgebnis(int ergebnis) {
    printf("\nErgebnis: %i\n", ergebnis);
}
```

# Header-Dateien

- **Header-Dateien**
  - dienen aus Gründen der Übersichtlichkeit zum Auslagern von Funktionsdeklarationen (Prototypen)
  - können Makros, symbolische Konstanten und globale Variablen enthalten (später mehr)
  - unüblich ist die direkte Aufnahme von ausführbaren Anweisungen (Implementierung)
    - Implementierung ist in der c-Datei zu finden: ***name.c***
  - Headerdatei: ***name.h***
  - Einfügen von Header-Dateien in Quelltextdateien
    - `#include "name.h"`
  - Die Notation "..." anstelle von <...> wird für eigene Header-Dateien eingesetzt.



## Aufg. 04.02

Verlagere den Programmcode für das Ermitteln, welche von zwei eingegebenen Zahlen die größer oder gleich ist (Aufgabe 3.02), in eine Funktion und lagere die Funktion und ihre Deklaration in ein .h/.c-Dateienpaar aus.

1. Schritt

Schreibe den Quelltext aus Aufgabe 3.02 so um, dass der Vergleich in eine Funktion ausgelagert wird. Deklariere die Funktion als Prototyp und schreibe sie nach ans Ende des Quelltexts

2. Schritt

Lagere nun den Prototyp in eine .h Header-Datei und die Funtion in eine separate .c Datei aus. Danach muss das Makefile entsprechend angepasst werden.

Werden die neuen Datei automatisch in Git übernommen?

## Aufg. 04.03

Lies den Radius eines Kreises ein. Es soll der Umfang und der Flächeninhalt berechnet werden.

a/ Realisiere das zunächst völlig sequenziell

b/ Erstelle dann für die Berechnung von Umfang und Flächeninhalt je eine Funktion und belasse diese in Ihrer „normalen“ C-Programm-Datei

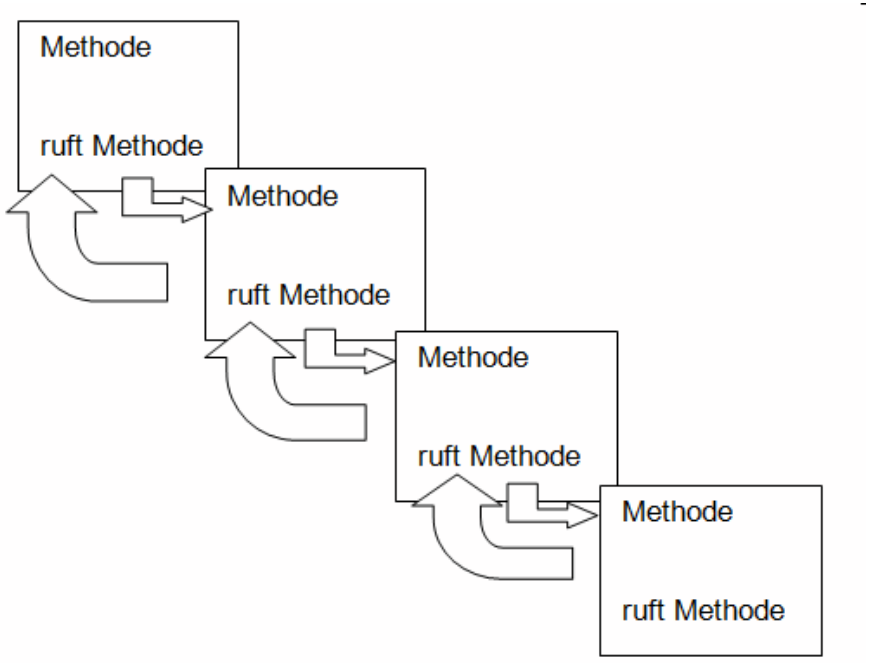
**c/ Lagere nun die Funktionen und ihre Deklarationen in ein gesondertes .h/.c-Dateienpaar aus und erstelle ein entsprechendes Makefile**

d/ Lade das Ergebnis in Dein Github-Repository

| | | |
|---|---|---|
| Hinweis: | Kreisumfang | $\rightarrow U = \pi \cdot 2 \cdot r$ |
| | Flächeninhalt | $\rightarrow A = \pi \cdot r^2$ |

## Rekursive Funktionen

- Funktionen können sich gegenseitig aufrufen
- manchmal ist es sinnvoll, dass Funktionen sich selbst aufrufen
  - Zurückführung auf sich selbst
  - insbesondere als Alternative zu Schleifenkonstrukten



## Rekursive Funktionen – Beispiel Fakultät

- Zur Wiederholung: Multiplikation der ganzen Zahlen von 1 bis n, geschrieben n!
  - z.B. $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$

- **Lösung**
  - die ersten (n-1) Faktoren des Produkts n! ergeben (n-1)!

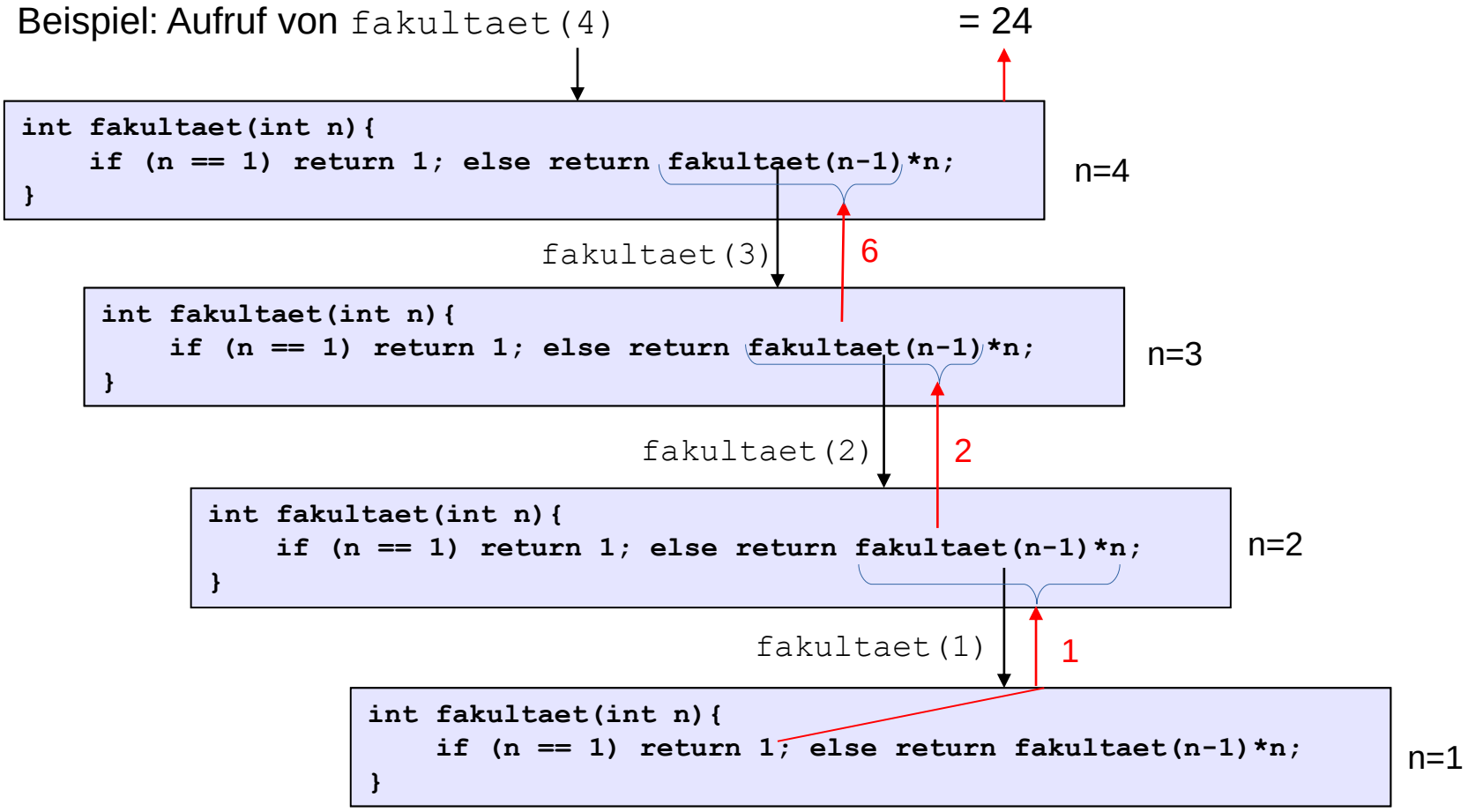
| | |
|---|---|
| $n! = (n-1)! \cdot n$ | falls $n>1$ |
| $n! = 1$ | falls $n=1$ |

  - Das Problem n! zu berechnen wurde auf das gleichartige, aber kleinere Problem (n-1)! zu berechnen zurückgeführt (reduziert)

  - Zweite Zeile ist notwendig, damit man bei der wiederholten Anwendung der ersten Zeile irgendwann zu einem Ende kommt

# Rekursive Funktionen – Beispiel Fakultät

- Auswertung der Rekursion
- Beispiel: Aufruf von `fakultaet(4)`



# Rekursive Funktionen – Beispiel Fakultät

- Rekursive C-Implementierung

```
#include<stdio.h>

int fakultaet(int n){
          if (n == 1){
                    return 1;
          } else {
                    return n * fakultaet(n-1);
    }
}

int main() {

    int n;

          printf("\t\tFakultaetsberechnung rekursiv\n");
          printf("\nBitte n eingeben:");
          scanf("%i", &n);

          printf("%i! = %i\n\n", n, fakultaet(n));

          system("PAUSE");
          return 0;
}
```

## Aufgabe

### Aufg. 04.04

Schreibe ein ***rekursives*** Programm, das den größten gemeinsamen Teiler (GGT) zweier ganzer Zahlen berechnet.

Hilfestellung:

```
ist Zahl1 == Zahl2 dann gib Zahl1 zurück
ist Zahl1  > Zahl2 dann gib ggT(Zahl1-Zahl2, Zahl2) zurück
ist Zahl1  < Zahl2 dann gib ggT(Zahl1, Zahl2-Zahl1) zurück
```

## Aufgabe

### Aufg. 04.05

Schreibe ein ***rekursives*** Programm, das die Fibonacci-Zahl $f_n$ ausgibt. Die Fibonacci-Folge ist eine unendliche Folge von Zahlen, bei der sich die jeweils folgende Zahl durch Addition ihrer beiden vorherigen Zahlen ergibt. Errechnet werden können sie mittels ... 1+2=3, 2+3=5, 3+5=8, 5+8=13. Die Formel lautet also: **F(n+2)=F(n+1) +F(n)**

Es gilt für die ersten beiden Zahlen:
$f_0 = 0$ und $f_1 = 1$

Folge: 0, 1, 1, 2, 3, 5, 8, 13…

Beispiel: $f_6 = f_5+f_4 = 5+3 = 8$

Oder:
**F(n)=F(n-1) +F(n-2)**