Programmierung 1

Funktionen II

Dr. Monika Schak

Woche 6

26. November 2025

Wiederholung

- Funktionen dienen der Modularisierung und Strukturierung
- Aufbau: <Rückgabetyp> <Funktionsname> (<Parameterliste>) { ... }
- Aufruf: <Funktionsname> (<Argumente>) $\rightarrow$ Argumente müssen in Typ, Reihenfolge und Anzahl der Signatur entsprechen
- Aktuelle Parameter werden beim Aufruf kopiert
- Funktionen müssen vor dem ersten Aufruf deklariert/definiert worden sein $\rightarrow$ Zwei Möglichkeiten: erst deklarieren, dann verwenden, dann definieren **oder** erst definieren, dann verwenden
- Variablen sind nur in dem Block sichtbar, in dem sie deklariert wurden

Eigene Header-Dateien

Warum?

- Größere Projekte werden schnell unübersichtlich $\rightarrow$ Modularisierung
- Trennung von Schnittstelle und Implementierung
- Wiederverwendbarkeit von Funktionen

Was gehört in die Header-Datei (`.h`)?

- Funktionsprototypen/-deklarationen
- Konstanten (`#define`, `const`)
- Typdefinitionen (`typedef`, `struct`)
- Schnittstellendokumentation (Kommentare!)

Was gehört in die C-Datei (`.c`)?

- Definition/Implementierung der Funktionen
- Interne Hilfsfunktionen

Eigene Header-Dateien verwenden

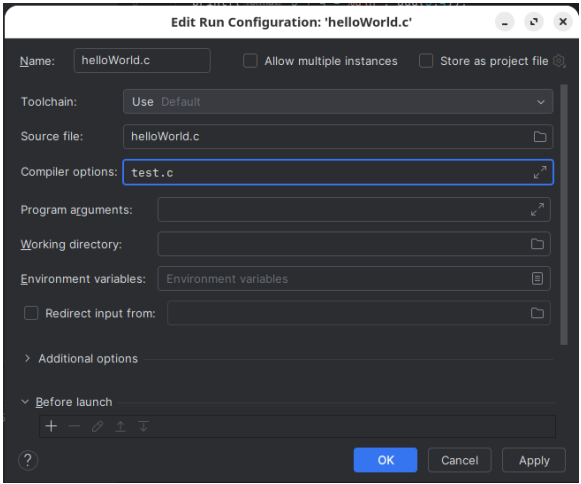
- Über `#include "test.h"` einbinden
  - `#include "..."` für eigene Header
  - `#include <...>` für Systembibliotheken
- Include-Guards: verhindern doppelte Einbindung (automatisch durch CLion erstellt)

```
#ifndef CODE_TEST_H
#define CODE_TEST_H

#endif //CODE_TEST_H
```

- Run Configuration anpassen, damit der Linker seine Arbeit machen kann!

Run Configuration



Best Practices

Lesbarkeit ist wichtiger als Kürze!

- Sinnvolle Funktionsnamen verwenden
- Nur eine Aufgabe pro Funktion
- Trennung von Logik und Ausgabe
- Kommentare, die erklären warum etwas gemacht wird
- Fehlerbehandlung, z.B. „falsche“ Eingaben

$\rightarrow$ Wenn man es nach 2 Wochen nicht mehr versteht, ist es schlecht geschrieben.

Beispiel

- Erstellen Sie ein Programm, das zwei ganze Zahlen vom Benutzer einliest, daraus die folgende Werte berechnet und deren Ergebnis ausgibt:
  - Summe
  - Produkt
  - Maximum
  - Minimum
- **Wichtig:** Das Programm soll in zwei Teile gegliedert werden. Einmal das Hauptprogramm, das die Ablaufsteuerung sowie Ein- und Ausgaben übernimmt. Und eine Bibliothek für die mathematischen Berechnungen (`math_utils`).

Wiederholung: Iteration

- Anweisungen im Schleifenrumpf werden wiederholt ausgeführt, bis die Schleife mittels Abbruchbedingung beendet wird.
- Iterative Lösung der Fakultätsfunktion:
  - Entsprechend Formel: $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n = \prod_{k=1}^{n} k$, wobei $0! = 1$
  - C-Code mit Schleife:

```
unsigned int fakultaet (unsigned int n) {
  unsigned int res = 1;
  for (unsigned int i = 2; i <= n; i++) {
    res *= i;
  }
  return res;
}
```

Rekursive Fakultätsberechnung

- Statt iterativ alternativ **rekursiv** möglich
- Rekursive Definition der Fakultät:

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

- C-Code:

```
unsigned int fac(unsigned int n) {
  if (n == 0) {
    return 1;
  }
  return n * fac(n - 1);
}
```

Rekursive Funktionen

- Eine Funktion wie `fac(n)`, die sich wieder selbst aufruf, heißt rekursiv. Ein solcher Aufruf kann auch indirekt über eine weitere Funktion erfolgen.
- Durch Rekursion wird ein Problem „auf sich selbst“ zurückgeführt. Das funktioniert nur, wenn das Problem durch diese Rückführung einfacher wird und wenn es eine Abbruchbedingung für die Rekursion gibt.
- Eine rekursive Funktion besteht immer aus:
  - einem oder mehreren Basisfällen, die als Abbruchbedingung dienen und meist einfach implementiertbar sind, wie z.B. für `fac(0)`.
  - dem rekursiven Fall.

Fibonacci-Folge

- Die Fibonacci-Zahlen sind rekursiv wie folgt definiert:

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n-1) + fib(n-2) & \text{falls } n \geq 2 \end{cases}$$

- Tabellarisch:

n	0	1	2	3	4	5	6	7	8	9	...
fib(n)	0	1	1	2	3	5	8	13	21	34	...

Implementierung

- Rekursiv

```
unsigned int fibonacciRek (unsigned int n) {
  if (n <= 1)
    return n;
  return fibonacciRek (n-1) + fibonacciRek (n-2);
}
```

- Iterativ

```
unsigned int fibonacciIter (unsigned int n) {
  unsigned int f0 = 0, f1 = 1, res = n;
  for (int i=2; i<=n; i++) {
    res = f0 + f1;
    f0 = f1;
    f1 = res;
  }
  return res;
}
```

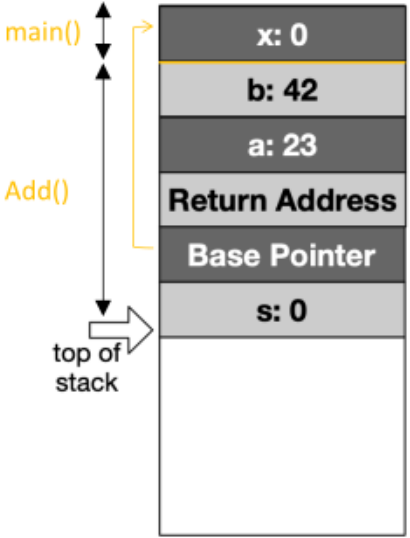
Rekursion

- Auf zwei Punkte achten:
  - **Rekursiver Fall/Rekursionsschritt**: Argumente des rekursiven Aufrufs müssen Aufgabe darstellen, die einfacher zu lösen ist als die, die Aufrufer übergeben wurde → Argumente müssen „kleiner“ werden.
  - **Basisfall/Terminierung**: Bei jedem Aufruf prüfen, ob Aufgabe ohne erneute Rekursion gelöst werden kann.
- Rekursion ist Programmiermethode
  - Dabei wird ein Problem gelöst, indem es auf einfachere Instanz zurückgeführt wird.
  - Was einfacher bzw. kleiner bedeutet, hängt von den verwendeten Datentypen ab:
    Integer → kleinere Zahl, Basisfall oft 0 oder 1
    Array → nur Teil des Arrays wird noch verwendet, Basisfall oft leeres Array
    String → kürzerer String, Basisfall oft leerer String

Einschub: Call Stack

- Funktion braucht Platz, um lokale Variablen zu speichern. Dieser Bereich heißt Call Stack, der aus Speicherplätzen mit numerischer Adresse besteht. Variablen ordnen diesen Adressen Namen zu.
- Alle Daten und Variablen eines Funktionsaufrufs befinden sich im Stack Frame.
- Bei jedem Funktionsaufruf wird ein neuer Stack Frame angelegt. Dadurch entsteht ein Aufrufstapel (sog. Call Stack). Alle Parameter (sowie danach die Rückgabewerte) werden kopiert.
- Nach Beendigung wird der Stack Frame an das Betriebssystem zurückgegeben. (Achtung: Stack Size ist endlich! Kann bei vielen Rekursionen zum Stack Overflow führen.)

Einschub: Call Stack




```
int Add(int a, int b)
{
    int s = 0;
    s = a + b;

    return s;
}

int main()
{
    int x = 0;

    x = Add(23,42);

    return 0;
}
```

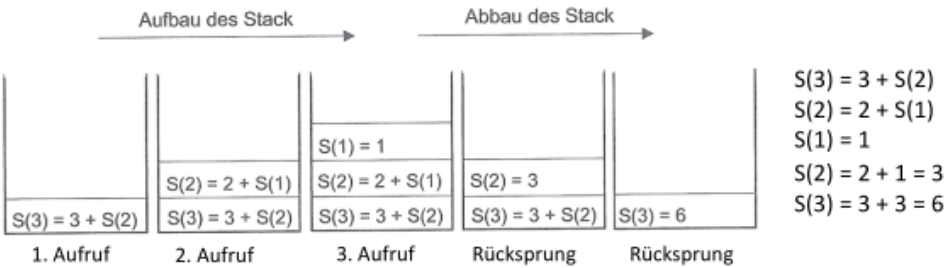
Rekursive Summenfunktion

- Summenformel: $S(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
- Rekursive Formulierung:

$$S(n) = \begin{cases} n + S(n-1) & \text{falls } n > 1 \\ 1 & \text{falls } n = 1 \end{cases}$$

```
unsigned int sum (unsigned int n) {
  if (n == 0) {
    return 0;
  }
  return n + sum(n-1);
}
```

- Berechnungsschritte für $n = 3$:



Probleme und Grenzen

- Iteration vs Rekursion
  - Prinzipiell immer beides möglich
  - Rekursion: einfach, aber meist ineffizient, gut für Strukturen wie Bäume und Listen.
  - Iteration: schwieriger, aber meist effizienter

- Vorsicht: Bei manchen rekursiven Funktionen ist die Berechnunsdauer nicht abschätzbar!
  Beispiel: Ackermann-Funktion

$$
\begin{aligned}
a(0, m) &= m + 1 && \forall m \geq 0 \\
a(n, 0) &= a(n-1, 1) && \forall n \geq 1 \\
a(n, m) &= a(n-1, a(n, m-1)) && \forall n, m \geq 1
\end{aligned}
$$

  Übersteigt selbst bei sehr kleinen Eingabewerten schnell alle Berechnungsmöglichkeiten!