04 - Funktionen

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Rückblick: Offenes Problem

```
int main() {
    int sum = 0, start = 0, end = 10;
    for (int i = start; i <= end; ++i) {
        sum += i;
    }
    int sum2 = 0; start = 10; end = 35; // Für Intervall [10,35]
    for (int i = start;
        sum2 += i;
    }
}
```

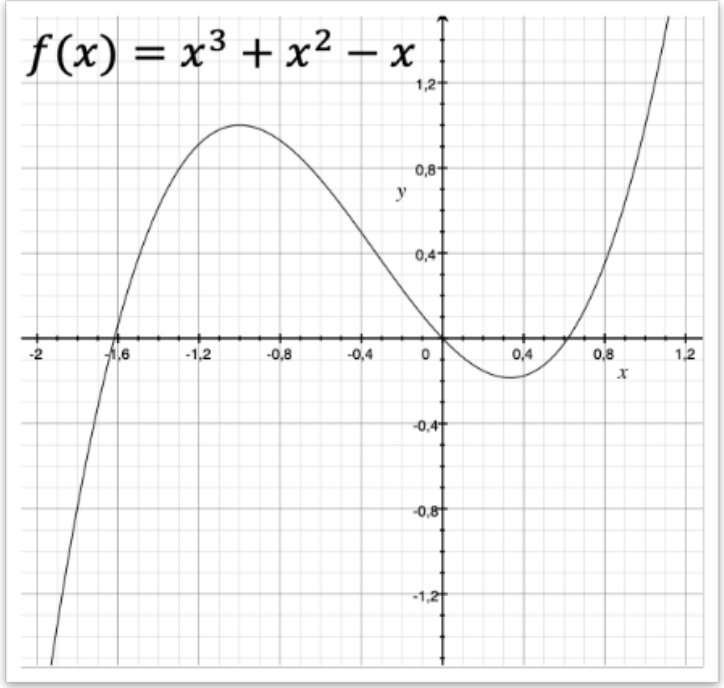
Dieses Problem wird heute gelöst! 😀

- Bisher: Duplizierung von Codeabschnitten

(Mathematische) Funktionen

- Mathematische Funktionen bisher bekannt aus dem Schulunterricht
- $f : \mathbb{R} \to \mathbb{R}$
- Beispiel

  $f : f(x) = x^3 + x^2 - x$ mit Eingabe $x = 0.5$
- Schritte für $y = x^3 + x^2 - x$
  1. Berechnung von $0,5^3 + 0,5^2 - 0,5$
  2. Ausgabe des Ergebnisses
  3. Zuweisung des Ergebnisses an y



Funktionen in Programmiersprachen sind sehr ähnlich aufgebaut!

Funktionen in C

- Eine Funktion in C besteht ebenfalls aus einem
  - Namen,
  - Funktionsparametern und
  - einer Rückgabe
- Formaler Aufbau:

  <Rückgabewert> <Name> (<Parameter 1>, <Parameter 2>{, ...) { <Code-Block> }
- Bereits bekanntes Beispiel: int main()
  - main() hat leere Liste von Eingabeparametern
  - Gibt einen Wert vom Typ int zurück (mithilfe des Schlüsselworts `return`)
  - Ebenfalls schon bekannt: `return 0;`

Funktionen in C

Angewandt auf das Sinus-Beispiel ergibt sich folgendes:

C Run ▶

```c
#include <math.h> // Mathematik-Funktionen
#include <stdio.h>

int main() {
    double x = 0.5, y, z;
    y = sin(x); // sin(): Funktion mit 1 Parameter
    z = sin(1.5);
    printf("y: %f, z: %f", y, z);
    return 0;
}
```

(Zur Referenz: `sin()` auf CPPReference)

Funktionsnamen

- Es gelten die gleichen Regeln wie für Variablen
  - Erlaubt sind: `a..z,A..Z, 0..9, _`
  - Am Anfang muss ein Buchstabe stehen
- Namenskonventionen wie *Snake-Case* oder *Camel-Case* sind auch hier üblich

```
void print_number(int number);
void printNumber(int number);
void PrinT____num123(int number); // Unschön
```

Funktionsparameter

- Werden als Eingabewerte mit definierten Typen der Funktion übergeben
- Beliebige Anzahl an Parametern möglich
- Funktionsweise analog zu Variablen:

  Deklaration **jedes** Parameters mit Namen und Typ
- Beispiel

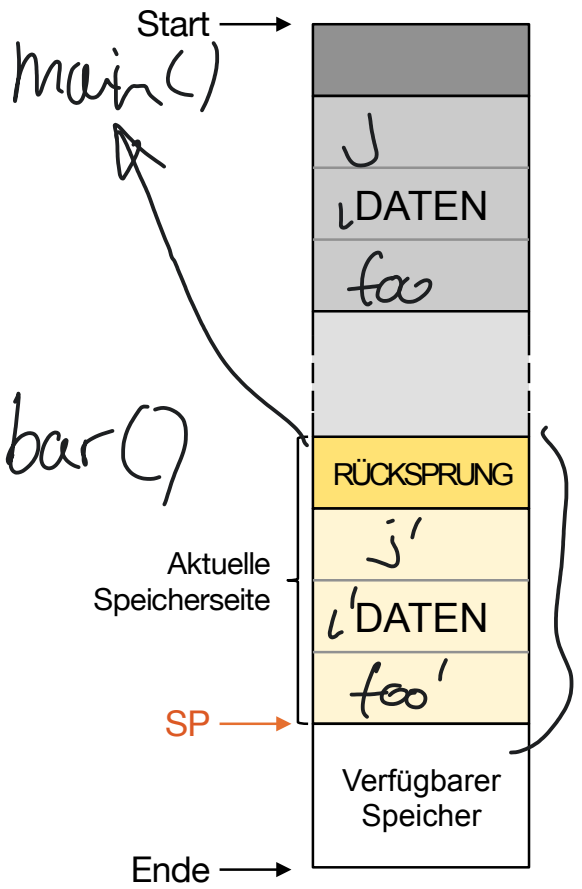
```
int sum(int summand1, int summand2);
```

Arbeitsweise der Funktionsparameter

- Ausgangspunkt: Kommaseparierte Liste von Parametern
- Jeder Parameter ist eine lokale Variable
  - Existiert nur innerhalb der Funktion
  - Verwendbar wie jede Variable
- Wichtig: Beim Aufurf ist Reihenfolge der Parameter relevant
- Bei Aufruf: Werte für jeden Parameter werden in separaten Speicher **kopiert**

Einschub: Ablage auf dem Stack

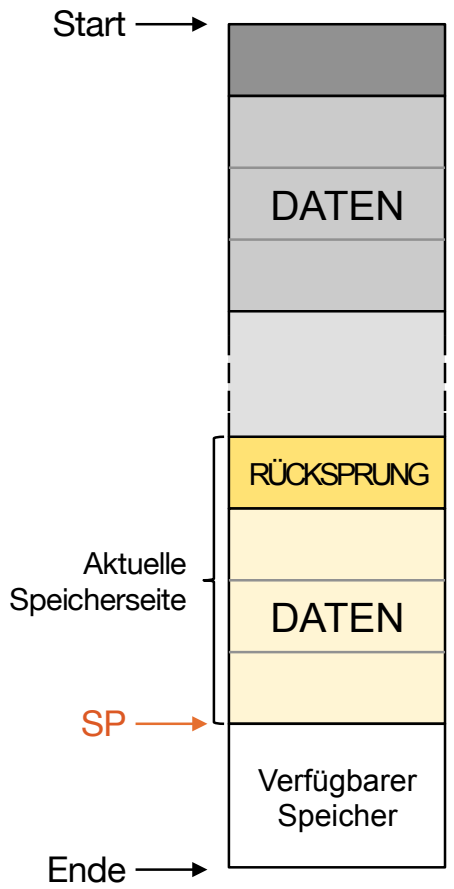
- Realisierung mittels Arbeitsstapels (*Stack*)
- Parameter/Variablen innerhalb eines Code-Blocks landen bei Deklaration auf dem Stack
- Ablage erfolgt gemäß Reihenfolge der Deklaration
- Beim Verlassen des Blocks: Aufräumen - Herunternehmen in umgekehrter Reihenfolge



Einschub: Ablage auf dem Stack

- Beispiel: Aufruf von `sum(2, 3)`
- Ablauf
  1. Bei Aufruf werden Parameter `2` und `3` auf den Stack kopiert
  2. Rechner springt an den Beginn der Funktion (ähnlich wie `goto`)
  3. Variablen `summand1` / s
  4. Funktion wird ausführ
  5. Aufräumen der Kopien
  6. Zurückspringen **hinter** den Funktionsaufruf

Zurück zu den Funktionen!



- Der Stack wird an späterer Stelle noch einmal etwas genauer betrachtet

Beispiel - Funktion mit zwei Parametern

C Run ▶

```c
#include <stdio.h>
float sum_of_two_params(float summand1, int summand2) {
    float sum = summand1 + summand2;
    return sum;
}
int main() {
    float first_sum = sum_of_two_params(2.9f, 5);
    float second_sum = sum_of_two_params(5, 2.9f);
    printf("first = %.2f, second = %.2f\n", first_sum, second_sum);
}
```

Was wird für `second_sum` ausgegeben?

A: `7.0`

B: `7.9`

C: Kompiliert nicht

D: Undefiniertes Verhalten

Beispiel - Funktion mit vertauschten Parametern

C Run ▶

```c
#include <stdio.h>

void print_float_and_int(float f, int i) {
    printf("Float %.2f\n", f);
    printf("Int %d\n", i);
}

int main() {
    print_float_and_int(1, 2.73);
}
```

Rückgabewerte von Funktionen

- Beliebiger Typ für Rückgabe des Ergebnisses
- Rückgabetyp muss spezifiziert werden
- Rückgabe muss mit dem Schlüsselwort `return` erfolgen
- Zurückgegebener Wert muss zum o. g. Typ passen
- **Kein** Rückgabewert möglich → Schlüsselwort `void`

```
int number_x2(int number) { return number * 2; } // Ok
void ret_int() { return 1.5; } // So nicht erlaubt (falscher Typ)
void do_nothing() { printf("Just print"); }
```

Exkurs: Schlüsselwort `void`

- `void` ist ein spezieller Variablen-Typ: Steht für einen leeren Wert
- Gilt *formal* als unvollständiger Typ, kann nicht vervollständigt werden
- Benötigt nicht zwingend ein `return` innerhalb der Funktion…

  … kann aber für einen vorzeitigen Abbruch genutzt werden

Exkurs: Schlüsselwort `void`

C Run ▶

```c
#include <stdio.h>



void print() {
    printf("Message\n");
}

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

C Run ▶

```c
#include <stdio.h>

void print2(int num) {
    if (num < 2) {
        // vorzeitiger Rücksprung
        return;
    }
    else {
        printf("Message\n");
    }
}

int main() {
    print2(1);
    return 0;
}
```

Arrays als Parameter

- Problem: Arrays enthalten nicht ihre Größe
- Lösung: Länge als **zusätzlichen** Parameter angeben

```
void print_array(int array[], int n);
```

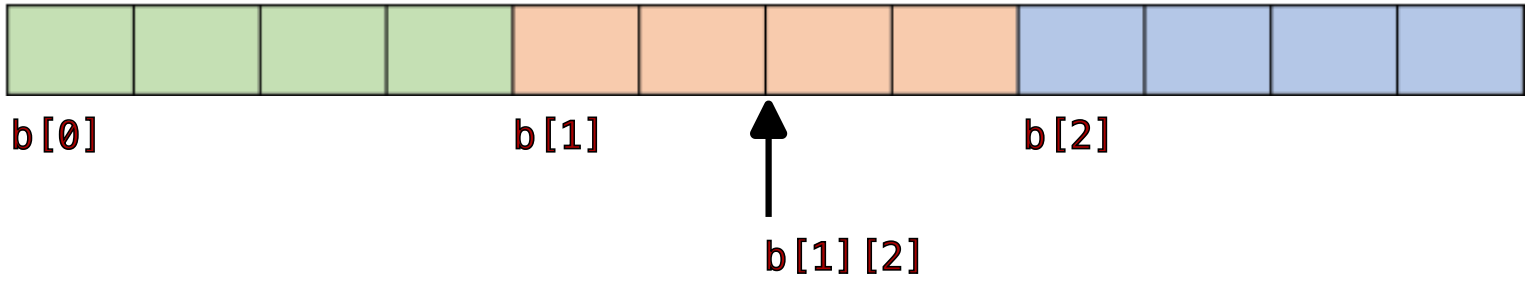
- Wichtig: So bitte nicht, Compiler ignoriert die Größenangabe

```
void print_array(int array[5]);
```

- 🚨 Achtung: Haltet die Array-Grenzen ein
  - Zugriffe nur auf Indizes aus $[0, n-1]$
  - Alles andere greift auf nicht-existenten Speicher zu

Funktionen: Mehrdimensionale Arrays als Parameter

- Es wird noch schlimmer: Bei Arrays mit mehr als einer Dimension **muss** die genaue Größe aller Dimensionen nach der ersten angegeben werden
- Grund: Array ist aufeinanderfolgende und im Speicher zusammenhängende Menge von Werten
- Compiler muss wissen, wie groß jede Zeile ist
- Beispiel: Array `int b[3][4]`
  - Zugriff auf `b[1][2]` : Drittes Element der zweiten Spalte



Die Standardbibliothek

- Mit *C-Compiler* wird die sog. Standardbilbiothek, die *libc*, geliefert
- Bekanntgabe der Funktionen erfolgt in Header-Dateien (→ nächstes Kapitel)
  - Siehe Zeilen mit `#include <XX>`
  - Bereits bekannt: `stdio.h`, `math.h`, `stdbool.h`
- Weitere wichtige Dateien
  - `stdlib.h` → u.a. Speicherverwaltung
  - `time.h` → Zeit, Datum
  - `string.h` → Speichermanipulation + String-Operationen
- Vollständige Liste auf cppreference.com → Schaut unbedingt mal rein!

Rekursion

- Rekursion (lateinisch: *recurrere – wiederkehren*):
  - Ein sich wiederholender Vorgang, prinzipiell unendlich
  - Häufig auftretende Strategie zur Problemlösung in der Algorithmik (Mathematik und Informatik)
- Rekursion entsteht durch das Aufrufen einer Funktion durch sich selbst
  - Schrittweise Vereinfachung von komplexen Problemen
  - Kurze und elegante Lösungen
- Welches rekursives Vorgehen kennt Ihr?

Beispiel - Rekursion

C Run ▶

```c
#include <stdio.h>

int sum(int num) {
    if(num > 0) {
        return num + sum(num -1);
    }
    return num; // 0
}

int main() {
    printf("%d\n", sum(10));
}
```

- `sum()` bildet Summe der Zahlen im Intervall $[1, num]$
- **Wichtig:** Abbruchbedingung zwingend erforderlich → `(num > 0)`
- Ansonsten: Unbeschränktes Selbstaufrufen; Endlosrekursion

Seiteneffekte von Rekursion

- Rekursion ist für die Beschreibung von Problemen gut geeignet
- Für praktische Programmierung (in C) nur eingeschränkt verwendbar
- Beispiel `sum()`
  - Pro Aufruf Kopie von `num` bzw. `num - 1` im Speicher
  - Erst **nach** Erreichen der Abbruchbedingung wieder freigegeben
- Sehr problematisch bei vielen Aufrufen (z.B. `sum(10000)`)

Alternative zur Rekursion

- Besser: **Iterative** Herangehensweise, meist über Schleifen
- Für komplexere Probleme meist mehr Aufwand
- Zeigt aber bessere Performance

```
int sum(int num) {
    int sum = 0;
    for(int i = 1; i <= num; ++i) {
        sum += 1;
    }
}
```

Deklaration und Definition

- Analog zu Variablen: Vor der ersten Benutzung müssen Funktionen bekannt sein
- Unterscheidung zwischen Deklaration und Definition
- Deklaration macht Signatur bekannt
  - Rückgabetyp Bezeichner(Parameter);
  - Beispiel: `int sum(int a, int b);`
- Definition spezifiziert die Anweisungen

  → Funktionsrumpf, als Code-Block `{}`
- Deklaration und Definition können getrennt oder zusammen erfolgen

Trennung von Deklaration und Definition

- Platzverbrauch: **Eine** Zeile pro Deklaration ⇔ **beliebig viele** pro Definition
- Vorteile einer Trennung
  1. Bessere Ordnung, Übersicht und Lesbarkeit für schnelles Nachschauen (Programmierer:innen)
  2. Beschleunigung de… …ilation Units*) → Übersetzer muss nur Signature…
  3. Schnellere Analyse…
- Für *kleine* Projekte… …eit nachrangig
- Aber: Sehr wichtig bei *großen* Projekten mit mehreren Millionen Zeilen

Die praktische Umsetzung dieser Trennung wird erst im nächsten Kapitel betrachtet

Ausblick: Wertetausch der Funktionsparameter

- Funktionen sind mächtig
- Fehlt uns noch etwas?
  Ja!
- Szenario: Tauschen zweiter Variablen

C Run ▶

```c
#include <stdio.h>
void swap_values(int one, int two) {
    int temp;
    temp = one;
    one = two;
    two = temp;
}

int main() {
    int x = 1, y = 2;
    swap_values(x, y);
    printf("x = %d, y = %d", x, y);
}
```

Ausblick: Wertetausch der Funktionsparameter

- Problem 1: C erlaubt nur *einen* Wert als Rückgabe
  - `return` kann also nur einen der beiden Werte zurückgeben
  - Lässt sich umgehen → eigenen Datentyp erstellen (→ Kapitel 7)
- Problem 2: Parameter werden **immer** kopiert (*Copy by Value*)
  - Zuweisung in Funktion verändert die Kopie, *nicht* das Original
  - Statt Kopie wird Speicheradressen übergeben und mit dieser gearbeitet → Kapitel 9

Zusammenfassung

- Einen Streifzug durch die Welt der Funktionen
- Das Wissen wird noch vertieft! → spätere Kapitel
- Deklaration und Definition von Funktionen
- Rekursion und ihre Probleme
- Übergabe von Funktionsparametern

04 - Funktionen