Systemnahe Programmierung in C (SPiC)

9 Funktionen

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2022

http://sys.cs.fau.de/lehre/SS22/spic

Was ist eine Funktion?

- **Funktion** := Unterprogramm [↪ GDI, 11-01]
  - Programmstück (Block) mit einem Bezeichner
  - Beim Aufruf können Parameter übergeben werden
  - Bei Rückkehr kann ein Rückgabewert zurückgeliefert werden
- Funktionen sind elementare Programmbausteine
  - Gliedern umfangreiche Aufgaben in kleine, beherrschbare Komponenten
  - Ermöglichen die einfache Wiederverwendung von Komponenten
  - Ermöglichen den einfachen Austausch von Komponenten
  - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)

Was ist eine Funktion?

- **Funktion** := Unterprogramm [↪ GDI, 11-01]
  - Programmstück (Block) mit einem Bezeichner
  - Beim Aufruf können Parameter übergeben werden
  - Bei Rückkehr kann ein Rückgabewert zurückgeliefert werden
- Funktionen sind elementare Programmbausteine
  - Gliedern umfangreiche Aufgaben in kleine, beherrschbare Komponenten
  - Ermöglichen die einfache Wiederverwendung von Komponenten
  - Ermöglichen den einfachen Austausch von Komponenten
  - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)

Funktion ↦ Abstraktion ↪ 4–1

- Bezeichner und Parameter abstrahieren
  - Vom tatsächlichen Programmstück
  - Von der Darstellung und Verwendung von Daten
- Ermöglicht schrittweise Abstraktion und Verfeinerung

Beispiel

- Funktion (Abstraktion) `sb_led_setMask()`

```
#include <led.h>
void main(void) {
  sb_led_setMask(0xaa);
  while(1) {}
}
```

- Implementierung in der `libspicboard`

```
void sb_led_setMask(uint8_t setting)
```

Sichtbar: Bezeichner und formale Parameter

Beispiel

- Funktion (Abstraktion) `sb_led_setMask()`

```
#include <led.h>
void main(void) {
  sb_led_setMask(0xaa);
  while(1) {}
}
```

- Implementierung in der `libspicboard`

```
void sb_led_setMask(uint8_t setting)
```

Sichtbar: Bezeichner und formale Parameter

```
{
  uint8_t i = 0;
  for (i = 0; i < 8; i++) {
    if ((setting >> i) & 1) {
      sb_led_on(i);

    } else {
      sb_led_off(i);
    }
  }
}
```

Unsichtbar: Tatsächliche Implementierung

Funktionsdefinition [≈Java]

- Syntax: *Typ Bezeichner* **(** $FormaleParam_{opt}$ **)** **{***Block***}**
  - *Typ* — Typ des Rückgabewertes der Funktion, `void` falls kein Wert zurückgegeben wird [=Java]
  - *Bezeichner* — Name, unter dem die Funktion aufgerufen werden kann ↪ 5–3 [=Java]
  - $FormaleParam_{opt}$ — Liste der formalen Parameter: $Typ_1\ Bez_{1\,opt}$, ..., $Typ_n\ Bez_{n\,opt}$ (Parameter-Bezeichner sind optional) [=Java]
    `void`, falls kein Parameter erwartet wird [≠Java]
  - {*Block*} — Implementierung; formale Parameter stehen als lokale Variablen bereit [=Java]
- Beispiele:

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

```
void wait(void) {
  volatile uint16_t w;
  for (w = 0; w < 0xffff; w++) {
  }
}
```


Funktionsaufruf [=Java]

- Syntax: *Bezeichner* **(** *TatParam* **)**
  - *Bezeichner* — Name der Funktion, in die verzweigt werden soll [=Java]
  - *TatParam* — Liste der tatsächlichen Parameter (übergebene Werte, muss anzahl- und typkompatibel sein zur Liste der formalen Parameter) [=Java]
- Beispiele:

```
int x = max(47, 11);
```

Aufruf der `max()`-Funktion. `47` und `11` sind die tatsächlichen Parameter, welche nun den formalen Parametern `a` und `b` der `max()`-Funktion (↪ 9–3) zugewiesen werden.

```
char text[] = "Hello, World";
int x = max(47, text);
```

Fehler: `text` ist nicht `int`-konvertierbar (tatsächlicher Parameter 2 passt nicht zu formalem Parameter `b` ↪ 9–3)

```
max(48, 12);
```

Der Rückgabewert darf ignoriert werden (was hier nicht wirklich Sinn ergibt)

Funktionsaufruf – Parameterübergabe [≠Java]

- Generelle Arten der Parameterübergabe [↪ GDI, 14-01]
  - *Call-by-value*: Die formalen Parameter sind Kopien der tatsächlichen Parameter. Änderungen in den formalen Parametern gehen mit Verlassen der Funktion verloren. **Dies ist der Normalfall in C.**
  - *Call-by-reference*: Die formalen Parameter sind Verweise (Referenzen) auf die tatsächlichen Parameter. Änderungen in den formalen Parametern betreffen auch die tatsächlichen Parameter. **In C nur indirekt über Zeiger möglich.** ↪ 13–5
- Des weiteren gilt
  - Arrays werden in C immer *by-reference* übergeben [=Java]
  - Die Auswertungsreihenfolge der Parameter ist undefiniert! [≠Java]

Funktionsaufruf – Rekursion [=Java]

- Funktionen können sich auch selber aufrufen (Rekursion)

```
int fak(int n) {
  if (n > 1)
    return n * fak(n - 1);
  return 1;
}
```

Rekursive Definition der Fakultätsfunktion.

Funktionsaufruf – Rekursion [=Java]

- Funktionen können sich auch selber aufrufen (Rekursion)

```
int fak(int n) {
  if (n > 1)
    return n * fak(n - 1);
  return 1;
}
```

Rekursive Definition der Fakultätsfunktion.

Ein anschauliches, aber mieses Beispiel für den Einsatz von Rekursion!

Funktionsaufruf – Rekursion [=Java]

- Funktionen können sich auch selber aufrufen (Rekursion)

```
int fak(int n) {
  if (n > 1)
    return n * fak(n - 1);
  return 1;
}
```

Rekursive Definition der Fakultätsfunktion.

Ein anschauliches, aber mieses Beispiel für den Einsatz von Rekursion!

Rekursion $\mapsto$ $$$

Rekursion verursacht erhebliche Laufzeit- und Speicherkosten!
Pro Rekursionsschritt muss:

- Speicher bereit gestellt werden für Rücksprungadresse, Parameter und alle lokalen Variablen
- Parameter kopiert und ein Funktionsaufruf durchgeführt werden

Regel: Bei der systemnahen Softwareentwicklung wird möglichst auf **Rekursion verzichtet!**

Funktionsdeklaration [≠Java]

- Funktionen müssen vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
  - Eine voranstehende Definition beinhaltet bereits die Deklaration
  - Ansonsten (falls die Funktion „weiter hinten“ im Quelltext oder in einem anderen Modul definiert wird) muss sie explizit deklariert werden
- Syntax: *Typ Bezeichner* **(** *FormaleParam* **) ;**
- Beispiel:

```
// Deklaration durch Definition
int max(int a, int b) {
  if (a > b) return a;
  return b;
}

void main(void) {
  int z = max(47, 11);
}
```

```
// Explizite Deklaration
int max(int, int);

void main(void) {
  int z = max(47, 11);
}

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

Funktionsdeklaration (Forts.)

[≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert ($\mapsto$ bekannt gemacht) worden sein

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein

Achtung: C erzwingt dies nicht!

- Es ist erlaubt **nicht-deklarierte** Funktionen aufzurufen (↪ implizite Deklaration)
- Derartige Aufrufe sind jedoch **nicht typsicher**
  - Compiler kennt die formale Parameterliste nicht
    ⇝ kann nicht prüfen, ob die tatsächlichen Parameter passen
  - Man kann **irgendwas** übergeben
- Moderne Compiler generieren immerhin eine **Warnung**
  ⇝ Warnungen des Compilers immer ernst nehmen!

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein

- **Beispiel:**

```
#include <stdio.h>

int main(void) {
  double d = 47.11;
  foo(d);
  return 0;
}

void foo(int a, int b) {
  printf("foo: a:%d, b:%d\n", a, b);
}
```


Funktionsdeklaration (Forts.)

[≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein

- **Beispiel:**

```c
#include <stdio.h>

int main(void) {
  double d = 47.11;
  foo(d);
  return 0;
}

void foo(int a, int b) {
  printf("foo: a:%d, b:%d\n", a, b);
}
```

Funktion `foo()` ist nicht **deklariert** ⇝ der Compiler **warnt**, aber akzeptiert beliebige tatsächliche Parameter

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein

- **Beispiel:**

```c
#include <stdio.h>

int main(void) {
  double d = 47.11;
  foo(d);
  return 0;
}

void foo(int a, int b) {
  printf("foo: a:%d, b:%d\n", a, b);
}
```

`foo()` ist **definiert** mit den formalen Parmetern `(int, int)`. Was immer an tatsächlichen Parametern übergeben wurde, wird entsprechend interpretiert!

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein

- **Beispiel:**

Was wird hier ausgegeben?

```
#include <stdio.h>

int main(void) {
  double d = 47.11;
  foo(d);
  return 0;
}

void foo(int a, int b) {
  printf("foo: a:%d, b:%d\n", a, b);
}
```

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert ($\mapsto$ bekannt gemacht) worden sein
  - Eine Funktion, die mit leerer formaler Parameterliste deklariert wurde, akzeptiert ebenfalls beliebige Parameter $\rightsquigarrow$ **keine Typsicherheit**
  - In diesem Fall warnt der Compiler **nicht!** Die Probleme bleiben!

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
  - Eine Funktion, die mit leerer formaler Parameterliste deklariert wurde, akzeptiert ebenfalls beliebige Parameter ⇝ **keine Typsicherheit**
  - In diesem Fall warnt der Compiler **nicht!** Die Probleme bleiben!
- **Beispiel:**

```
#include <stdio.h>

void foo(); // "open" declaration

int main(void) {
  double d = 47.11;
  foo(d);
  return 0;
}

void foo(int a, int b) {
  printf("foo: a:%d, b:%d\n", a, b);
}
```

Funktion `foo` wurde mit **leerer** formaler Parameterliste deklariert ⇝ dies ist formal ein **gültiger Aufruf!**

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
  - Eine Funktion, die mit leerer formaler Parameterliste deklariert wurde, akzeptiert ebenfalls beliebige Parameter ⇝ **keine Typsicherheit**
  - In diesem Fall warnt der Compiler **nicht!** Die Probleme bleiben!

Achtung: Verwechslungsgefahr

- In Java deklariert `void foo()` eine **parameterlose** Methode
  - In C muss man dafür `void foo(void)` schreiben ↪ 9–3
- In C deklariert `void foo()` eine **offene** Funktion
  - Das macht nur in (sehr seltenen) Ausnahmefällen Sinn!
  - Schlechter Stil ⇝ Punktabzug

Regel: Funktionen werden stets **vollständig deklariert!**