System-Level Programming

9 Functions

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

What is a Function?

- **Function** := subprogram
  - program piece (block) that has an identifier
  - parameter can be passed when calling the function
  - a return value can be passed after finishing
- Functions are elementary program pieces
  - structure extensive tasks in smaller, manageable components
  - enable a simple reuse of components
  - enable a simple exchange of components
  - hide implementation details: **black-box** principle

What is a Function?

- **Function** := subprogram
  - program piece (block) that has an identifier
  - parameter can be passed when calling the function
  - a return value can be passed after finishing
- Functions are elementary program pieces
  - structure extensive tasks in smaller, manageable components
  - enable a simple reuse of components
  - enable a simple exchange of components
  - hide implementation details: **black-box** principle

function $\mapsto$ abstraction $\hookrightarrow$ 4–1

- Identifier and parameters abstract
  - from the actual program piece
  - from the representation and usage of data
- Enables a step-by-step abstraction and refinement

Example

- Function (abstraction) `sb_led_setMask()`

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

- Implementation in `libspicboard`

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

visible: identifier & formal parameters

Example

Function (abstraction) `sb_led_setMask()`

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

Implementation in `libspicboard`

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

visible: identifier & formal parameters

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

not visible: actual implementation

Function Definitions

- Syntax: *type identifier* **(** *formalParam*$_{opt}$ **)** **{***block***}**
  - *type* — type of the returned value, `void` if nothing is returned
  - *identifier* — name of the function, which is used for calling it ↪ 5–5
  - *formalParam*$_{opt}$ — list of formal parameters: $type_1\ id_{1\,opt}, \ldots, type_n\ id_{n\,opt}$ (parameter identifiers are optional) `void`, if no parameter is expected
  - {*block*} — implementation; formal parameters can be used as local variables
- Examples:

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

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

Function Calls

- Syntax: *identifier* **(** *actParam* **)**
  - *identifier* — name of the function to be jumped into
  - *actParam* — list of actual parameters (passed values, have to be compatible in type and count to the list of formal parameters) [=Python]
- Examples:

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

Call of the `max()` function. `47` and `11` are the actual parameters, which are mapped to the formal parameters `a` and `b` of the `max()`-function (↪ 9–3).

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

Error: `text` is not promotable to type `int` actual parameter 2 does not match definition of formal parameter `b` (↪ 9–3)

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

The returned value can be ignored (even though it makes no sense here)

Function Calls – Passing of Parameters [≠Python]

- General types of parameter passing
  - *call by value* — Formal parameters are copies of the actual parameters. Changes made to the formal parameters are lost when the function is exited.
    This is the standard case in C.
  - *call by reference* — Formal parameters are references to the actual parameters. Changes made to the formal parameters directly affect the actual parameters as well.
    In C possible with the help of pointers.

↪ 13–6

- Note:
  - arrays are always passed *by reference* [≈Python]
  - the order in which parameters are evaluated is undefined!

Function Calls – Recursion [=Python]

- Functions can call themselves (recursion)

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

Recursive definition of the factorial function.

- Functions can call themselves (recursion)

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

Recursive definition of the factorial function.

A descriptive but really bad example for the use of recursion!

- Functions can call themselves (recursion)

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

Recursive definition of the factorial function.

A descriptive but really bad example for the use of recursion!

recursion ↦ $$$

Recursion leads to a significant runtime and memory cost!

For each recursion step:

- Memory has to be provided for: return address, parameters and all local variables
- Parameter are copied, and a function call is performed

Rule: When possible, **avoid any recursion** when writing system-level code!

Function Declaration

- Functions have to be declared (↦ made known) in the source code prior to being used
  - When defining a function upfront, this definition serves as declaration
  - Otherwise, (if the function is implemented "further below" in the source code or is defined in another module) it has to be declared explicitly
- Syntax: *type identifier* **(** *formalParam* **) ;**
- Example:

```c
// declaration by definition
int max(int a, int b) {
  if (a > b) return a;
  return b;
}

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

```c
// explicit declaration
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;
}
```

09-Funktionen_en

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used

Attention: C does not enforce this!

- Possibility to call functions that are **not declared** (↪ implicit declaration)
- Such calls are however **not type-safe**
  - the compiler does not know the list of formal parameters
    ⇝ it cannot verify whether the actual parameters match
  - Possibility to pass **anything**
- Modern compilers at least generate a **warning**
  ⇝ Always take compiler warnings seriously!

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used
- **Example:**

```
#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);
}
```

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used
- **Example:**

```
#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);
}
```

Function `foo()` is not **declared** ⇝ the compiler **warns** but accepts any actual parameters

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used
- **Example:**

```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()` is **defined** with formal parameters `(int, int)`. Everything that is passed as actual parameters will be interpreted as `int`!

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used
- **Example:**

```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);
}
```

What will be printed?

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used
  - Functions that are declared with an empty list of formal parameters will also accept any parameter $\rightsquigarrow$ **no type safety**
  - The compiler does **not** warn in this case. The problems remain!

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used
  - Functions that are declared with an empty list of formal parameters will also accept any parameter ⇝ **no type safety**
  - The compiler does **not** warn in this case. The problems remain!
- **Example:**

```
#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
}
```

Function `foo()` has been declared with **empty** list of formal parameters ⇝ this is a formally **valid call!**

Function Declaration (continued)

- Functions ~~shall~~ **should** be declared in the code prior to being used
  - Functions that are declared with an empty list of formal parameters will also accept any parameter ⇝ **no type safety**
  - The compiler does **not** warn in this case. The problems remain!

Attention: Risk of confusion

- In Java, `void foo()` defines a **parameterless** method
- In Python, `def foo():` defines a **parameterless** method
  - In C, one has to explicitly write `void foo(void)` ↪ 9–3
- In C, `void foo()` declares an **open** function
  - This is only useful in (rare) cases!
  - Generally it is considered bad style ⇝ point deduction in exam!

Rule: Functions always need to be **declared completely!**

References