9 Functions

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

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

http://sys.cs.fau.de/lehre/ss24
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
- 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()`

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

Implementation in `libspicicboard`

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

Syntax:  \[\text{type identifier ( formalParam\text{opt} ) \{ block \}}\]

- **type**  
  type of the returned value,  
  \texttt{void} if nothing is returned  

- **identifier**  
  name of the function, which is used  
  for calling it  

- **formalParam\text{opt}**  
  list of formal parameters:  
  \texttt{type\text{1 id\text{1 opt}, \ldots, type\text{n id\text{n opt}}}  
  (parameter identifiers are optional)  
  \texttt{void}, if no parameter is expected  

- \{ \text{block} \}  
  implementation; formal parameters  
  can be used as local variables  

Examples:

```java
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++) {
  }
}
```
Function Calls

Syntax: \( \text{identifier} \left( \text{actParam} \right) \)

- **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)

Examples:

```java
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 (\( \mapsto 9-4 \)).

```java
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 \( b \) (\( \mapsto 9-4 \)).

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

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

Note:

- arrays are always passed *by reference*

- the order in which parameters are evaluated is *undefined!*
Functions can call themselves (recursion)

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

Recursive definition of the factorial function.

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
- Parameters are copied, and a function call is performed

Rule: When possible, avoid any recursion when writing system-level code!
Functions can call themselves (recursion)

```java
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)

```java
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 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!
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: \( \text{type identifier (formalParam)}; \)

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);
}

// 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;
}
```
Functions shall **should** be **declared** in the code prior to being used.
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!
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);
}
```
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);
}
```

Function `foo()` is not **declared**; the compiler **warns** but accepts any actual parameters.
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!
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?
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!
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 \( \sim \) **no type safety**
- The compiler does **not** warn in this case. The problems remain!

**Example:**

```c
#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);
}
```

Function `foo()` has been declared with **empty** list of formal parameters \( \sim \) this is a formally **valid call!**
Functions 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()` would define a **parameterless** method
  - In C, one has to explicitly write `void foo(void)`
- 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!**