5 Language Overview

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
A C-program (usually) consists of

- a set of global variables
- a set of (sub-)functions
  - a set of local variables
  - a set of instructions
- the function `main()`, which is the entry point for any execution
A C-program (usually) consists of

- a set of **global variables**
- a set of **(sub-)functions**
  - a set of **local variables**
  - a set of **instructions**
- the function **main()**, which is the entry point for any execution
// include files
#include <led.h>

// global variables
LED nextLED = RED0;

// subfunction 1
LED lightLED(void) {
    if (nextLED <= BLUE1) {
        sb_led_on(nextLED++);
    }
    return nextLED;
}

// subfunction 2
void wait(void) {
    volatile unsigned int i;
    for (i = 0; i < 0xffff; i++)
    {
    }
}

// main function
void main() {
    while (lightLED() < 8) {
        wait();
    }
}
// include files
#include <led.h>

// global variables
LED nextLED = RED0;

// subfunction 1
LED lightLED(void) {
    if (nextLED <= BLUE1) {
        sb_led_on(nextLED++);
    }
    return nextLED;
}

// subfunction 2
void wait(void) {
    volatile unsigned int i;
    for (i = 0; i < 0xffff; i++)
        ;
}

// main function
void main(void) {
    while (lightLED() < 8) {
        wait();
    }
}
Literals

// include files
#include <led.h>

// global variables
LED nextLED = RED0;

// subfunction 1
LED lightLED(void) {
    if (nextLED <= BLUE1) {
        sb_led_on(nextLED);
        nextLED++;
    }
    return nextLED;
}

// subfunction 2
void wait(void) {
    volatile unsigned int i;
    for (i = 0; i < 0xffff; i++)
        ;
}

// main function
void main() {
    while (lightLED() < 8) {
        wait();
    }
}

(Expression of) constants in the code

- For every primitive data type, there is at least one literal form.
  - for integers: decimal (base 10: 65535), hexadecimal (base 16, leading 0x: 0xffff), octal (base 8, leading 0: 0177777)

- The programmer can then choose the best suited form.
  - 0xffff is more handy than 65535 to represent the maximal value of a 16-bit integer
// include files
#include <led.h>

// global variables
LED nextLED = RED0;

// subfunction 1
LED lightLED(void) {
    if (nextLED <= BLUE1) {
        sb_led_on(nextLED++);
    }
    return nextLED;
}

// subfunction 2
void wait(void) {
    volatile unsigned int i;
    for (i = 0; i < 0xffff; i++)
        ;
}

// main function
void main() {
    while (lightLED() < 8) {
        wait();
    }
}
Outline the actual procedure of the program

They are hierarchically made up from three basic forms

- single instruction – expression followed by ;
  - single semicolon → empty instruction
- block – sequence of instructions, wrapped in {...}
- control structures, followed by instructions
Outline the actual **procedure** of the program

They are hierarchically made up from three basic forms

- single instruction – **expression** followed by `;`
  - single semicolon $\rightarrow$ empty instruction
- **block** – sequence of instructions, wrapped in `{...}`
- control structures, followed by instructions
Outline the actual procedure of the program

They are hierarchically made up from three basic forms

- single instruction — expression followed by ;
  - single semicolon $\mapsto$ empty instruction
- block — sequence of instructions, wrapped in {...}
- control structures, followed by instructions
Expressions

```c
// include files
#include <led.h>

// global variables
LED nextLED = RED0;

// subfunction 1
LED lightLED(void) {
    if (nextLED <= BLUE1) {
        sb_led_on(nextLED++);
    }
    return nextLED;
}

// subfunction 2
void wait(void) {
    volatile unsigned int i;
    for (i = 0; i < 0xffff; i++)
        ;
}

// main function
void main() {
    while (lightLED() < 8) {
        wait();
    }
}
```

Valid combination of operators, literals, and identifiers

- “valid” in the sense of syntax and types
- priority rules for operators determine the order, in which the expressions get handled
  - order of execution can be explicitly forced with the help of brackets ()
  - the compiler is allowed to evaluate partial expressions in the most efficient order