C introduction

# Control structures

Richard Mörbitz, Manuel Thieme

# Contents

# Back in control

Even though C is a sequential programming language, the program flow can branch. Use conditions to determine the behaviour of your program in certain situations.
Executing the same task multiple times can be achieved using loops.

# if...else

To make decisions during run time, you can use the truth value of an expression:

```
if (condition)
    statement1;
else
    statement2;
```

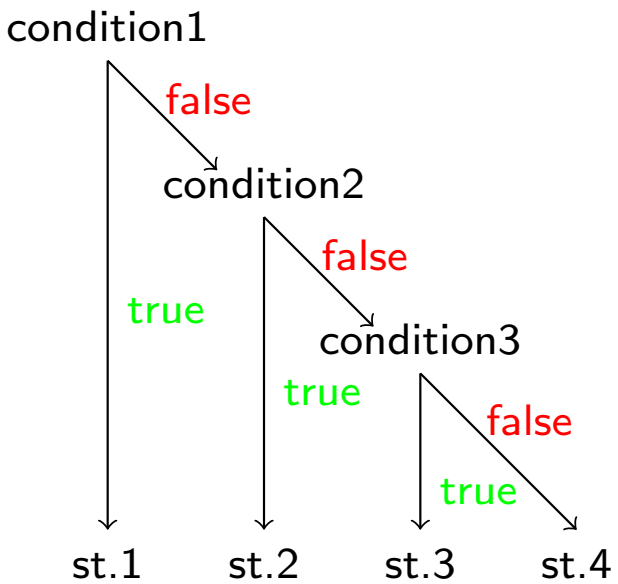
Now **statement1** is only executed if the truth value of **condition** is *true*. Otherwise **statement2** is executed. The *else* part is optional.

For multiple statements in the *if* or *else body*, use braces:

```
if (condition) {
    statement1;
    statement2;
}
```

# else if

To differentiate between more than two cases, you can use the if condition as a statement in the else body:




```
if (condition1)
    statement1;
else if (condition2)
    statement2;
else if (condition3)
    statement3;
else
    statement4;
```

# switch

If you have to check one variable for many constant values, *switch case* is your friend:

```
switch (variable) {
    case option1: statement1; break;
    case option2: statement2; break;
    case option3: statement3; break;
    default: statement4; break;
}
```

- *case option* defines a jump label
- More than one statement after it possible without braces
- All statements until the next *break;* will be executed

# A few words on style

- Typing **if (cond)** instead of **if(cond)** helps people to differentiate between control structures and function calls faster
- When starting a new block, you should type ) { rather than ){
- Do not start a new block for a single statement
- Do not put statements and conditions on the same line

```
if(cond){ statement; }   /* bad style */

if (cond) {              /* looks better, still bad style */
    statement;
}

if (cond)
    statement;           /* looks way clearer */
```

# More words on style

- if you use a block anywhere in an **if ... else** structure, put all blocks of this structure in braces

```
if (cond)          /* bad style, inconsistent */
    statement;
else {
    statement;
    statement;
}

if (cond) {        /* way better style */
    statement;
} else {
    statement;
    statement;
}
```

- notice: the *else* is on the same line as the closing if body brace

# Feedback

- You are now able to solve tasks 05 and 06.

# Loops

To repeat statements until a certain condition is met, C offers 3 different loops.

```
while (condition)
    statement;
```

```
do
    statement;
while (condition);
```

```
for (initialization; condition; statement)
    statement;
```

For multiple statements again, use braces.

# while

The execution of a loop is a continuous alternation between checking if the condition is still met and executing the statement(s).

```
1 int i = 2;
2 while (i > 0)
3     i--;
4 printf("done\n");
```

# while

The execution of a loop is a continuous alternation between checking if the condition is still met and executing the statement(s).

```
1 int i = 2;
2 while (i > 0)
3     i--;
4 printf("done\n");
```

1. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3

# while

The execution of a loop is a continuous alternation between checking if the condition is still met and executing the statement(s).

```
1 int i = 2;
2 while (i > 0)
3     i--;
4 printf("done\n");
```

1. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
2. Decrement i $\rightarrow$ i now is **1**, go back to line 2

# while

The execution of a loop is a continuous alternation between checking if the condition is still met and executing the statement(s).

```
1 int i = 2;
2 while (i > 0)
3     i--;
4 printf("done\n");
```

1. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
2. Decrement i $\rightarrow$ i now is **1**, go back to line 2
3. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3

# while

The execution of a loop is a continuous alternation between checking if the condition is still met and executing the statement(s).

```
1 int i = 2;
2 while (i > 0)
3     i--;
4 printf("done\n");
```

1. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
2. Decrement i $\rightarrow$ i now is **1**, go back to line 2
3. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
4. Decrement i $\rightarrow$ i now is **0**, go back to line 2

# while

The execution of a loop is a continuous alternation between checking if the condition is still met and executing the statement(s).

```
1 int i = 2;
2 while (i > 0)
3     i--;
4 printf("done\n");
```

1. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
2. Decrement i $\rightarrow$ i now is **1**, go back to line 2
3. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
4. Decrement i $\rightarrow$ i now is **0**, go back to line 2
5. Check (i $>$ 0) $\rightarrow$ **false** $\rightarrow$ go to line 4

# while

The execution of a loop is a continuous alternation between checking if the condition is still met and executing the statement(s).

```
1 int i = 2;
2 while (i > 0)
3     i--;
4 printf("done\n");
```

1. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
2. Decrement i $\rightarrow$ i now is **1**, go back to line 2
3. Check (i $>$ 0) $\rightarrow$ **true** $\rightarrow$ go to line 3
4. Decrement i $\rightarrow$ i now is **0**, go back to line 2
5. Check (i $>$ 0) $\rightarrow$ **false** $\rightarrow$ go to line 4
6. Print **done**

# Meanwhile...

Be careful, this

```
while (1 > 0)
    printf("Did you miss me?\n");
```

runs till the end of all days.

$\infty$ loops are common mistakes, and you will experience many of them. Check for conditions that are always true.

# do...while

The difference between *do...while* and *while* is the order of executing the statement(s) and checking the condition.
The *while* loop begins with checking, while the *do...while* loop begins witch executing the statement(s).

```
int i = 3;
do
    i--;
while (i < 1);
```

The Statement(s) in a *do ... while* loop are executed at least once.

# for

The For-Loop is comfortable for iterating. It takes three arguments.

- Initialization
- Condition
- Iteration statement

To understand how it's working, consider a program printing the numbers 1 to 10:

```
int i;
for (i = 1; i <= 10; i++)
    printf("%d\n", i);
```

- *i* is called an *index* whit iterates from the given start to a given end value
- *i, j, k* are commonly used identifiers for the index

# Saving code lines

You can define variables inside the initialization part of a for loop.

```
for (int i = 1; i <= 10; i++)
    printf("%d\n", i);
```

In that case, the variable is only available inside the for loop (as if it was declared in the body).
But you have to compile the program with *-std=c99*

```
gcc main.c -Wall -std=c99
```

# forever

The arguments for the *for loop* are optional. E.g. if you already have defined your iterating variable:

```
int i = 1;
for (; i <= 10; i++)
    printf("%d\n", i);
```

Or if you have the iteration statement in your loop body:

```
for (int i = 1; i <= 10;)
    printf("%d\n", i++);     /* why not using while? */
```

And if you're not passing anything, it runs **for**ever:

```
for (;;)
    printf("I'm still here\n");
```

Note: the semicolons are still there.

# Cancelling loops

*break*

- Ends loop execution
- Moves forward to first statement after loop

*continue*

- Ends current loop iteration
- Moves forward to next step of loop iteration
  - *while:* Jumps to condition
  - *for:* Jumps to iteration statement

# A few words on style

- Again, only use braces when there's more than one statement
- If you skip the loop body
  - Leave a comment in your code
  - Use an extra line for the empty statement

```
for (i = 1; i < 9; printf("%d\n", i++));     /* confusing */

for (i = 1; i < 9; printf("%d\n", i++))      /* clear */
    ;   /* do nothing */
```

# Exercises

- You are now able to solve tasks 07, 08 and 09