

Software Construction 1 (IE1-SO1)

Organizational



Before we begin ...

- Introduction
- Personal interests

May I introduce myself?

This is me!

- My background in industry
- My background at HAW Hamburg



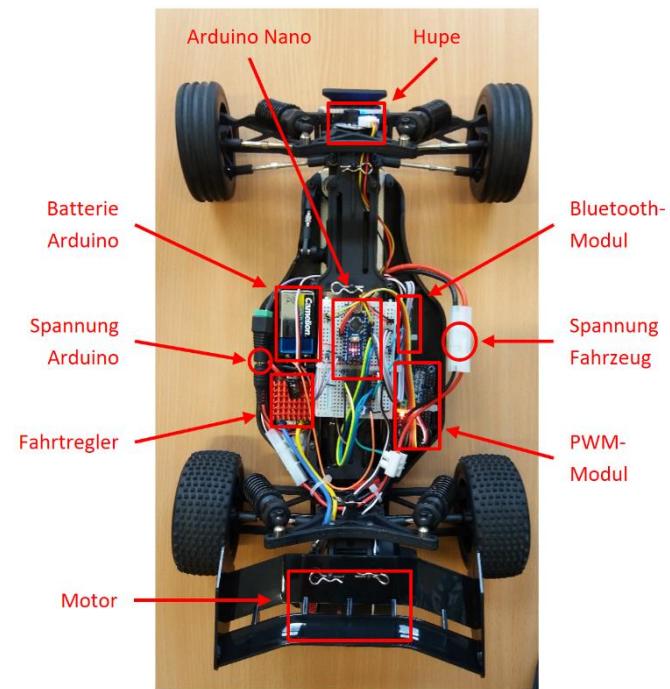
Contact:

- Before/after lecture or by e-mail (from your HAW account)
- No need to wait for the “next weekly office hours”. I respond quickly ... trust me! ☺
- Please do not send Teams chat messages.

Selected interests:

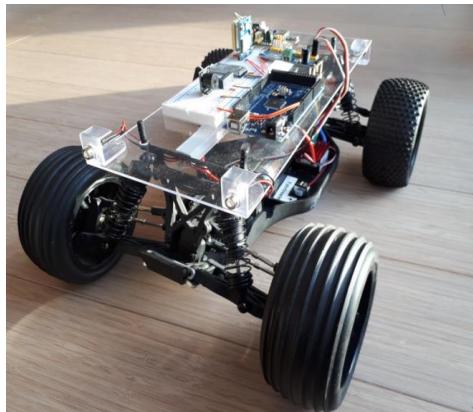
- Remote controlled models
- Steered by *Android app* (Java, *Software Construction 2*)
- *Arduino* boards for onboard control (C/C++, *Software Construction 1*)
- Onboard image processing (Python)

It all began with the first „Buggy“:



Engineering can be fun!

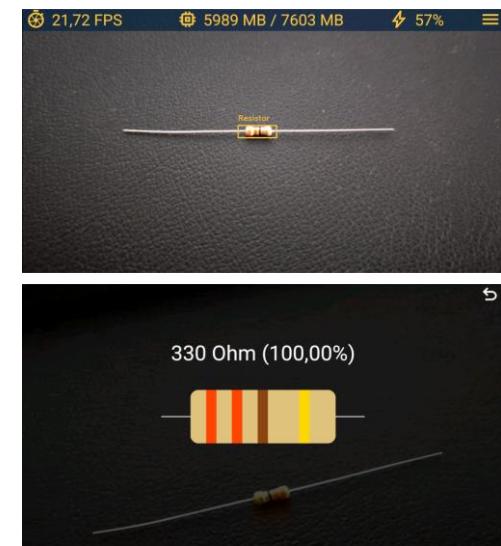
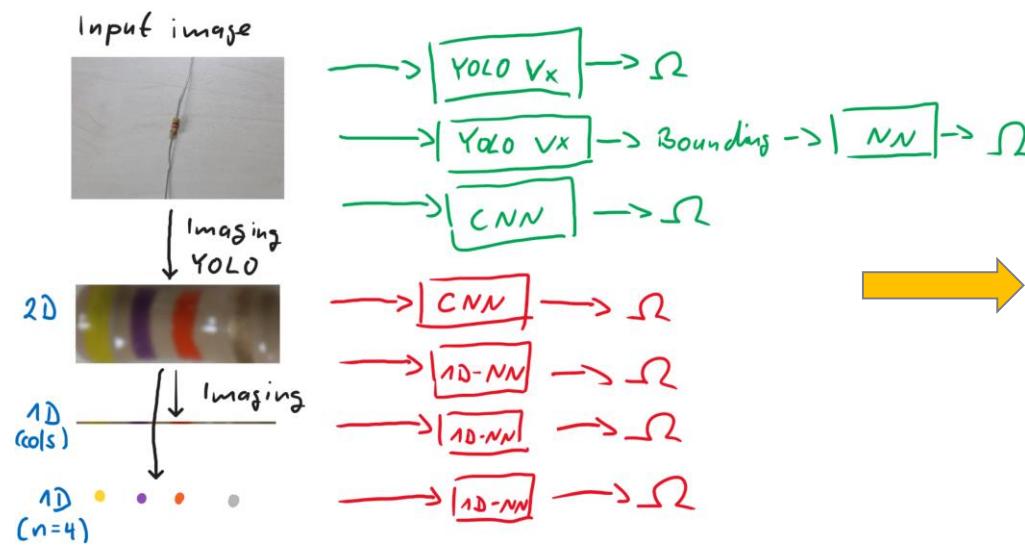
Don't know why ... many students seem to love **cars** (they are fun, though):



Let's look at other selected activities ...

Deep learning (and image processing):

- Extract and classify electrical resistors
- PC und Android apps



Deep reinforcement learning (and image processing):

- Learn to maneuver ball through labyrinths (virtual and physical)



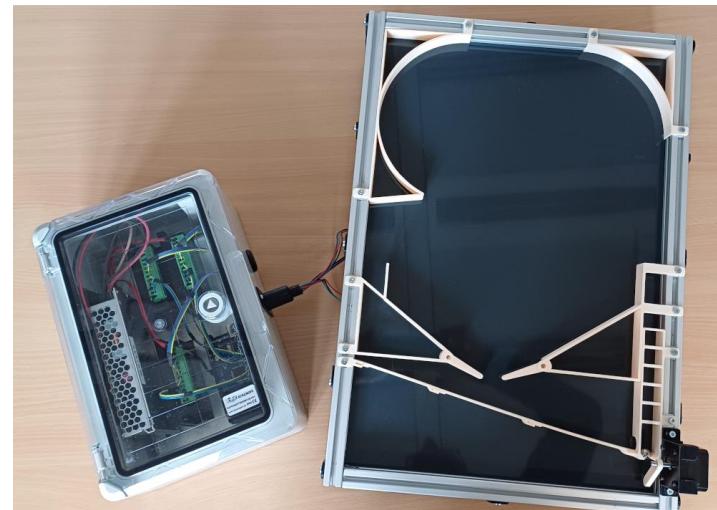
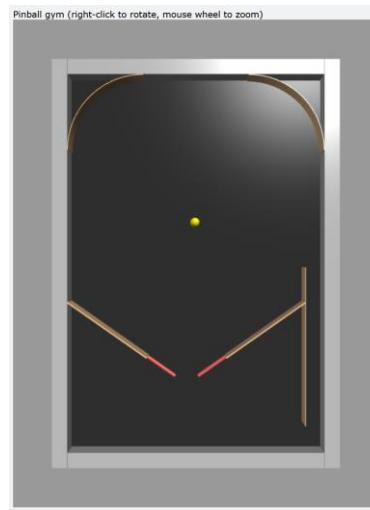
Supported by:

Allied Vision Technologies GmbH (industrial cameras), Kowa Optimed Deutschland GmbH (lenses)

 **Allied Vision** 

Deep reinforcement learning (and image processing):

- Learn to maneuver ball through labyrinths (virtual and physical)
- Learn to play pinball



Supported by:

Allied Vision Technologies GmbH (industrial cameras), Kowa Optimed Deutschland GmbH (lenses)



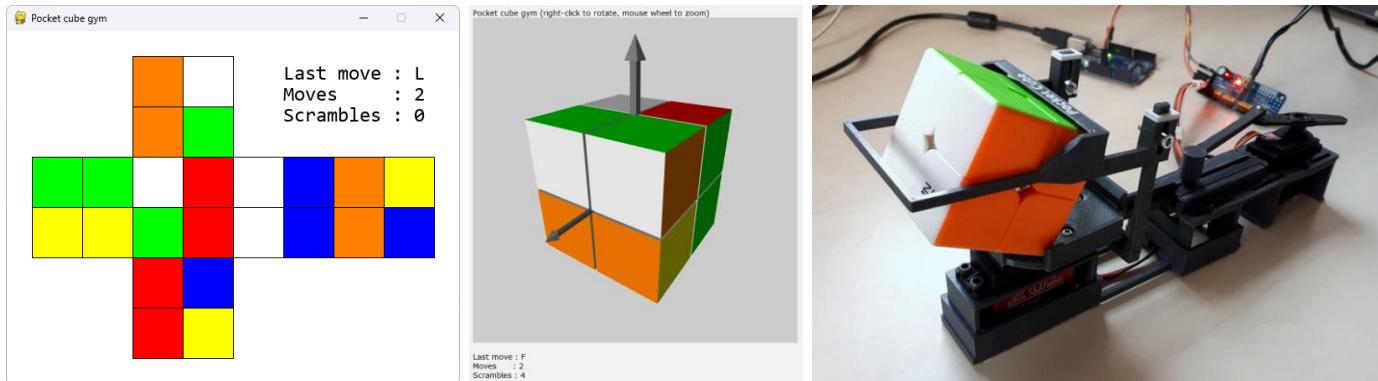
Deep reinforcement learning (and image processing):

- Learn to maneuver ball through labyrinths (virtual and physical)
- Learn to play pinball
- Learn to play the goalkeeper in table soccer



Deep reinforcement learning (and image processing):

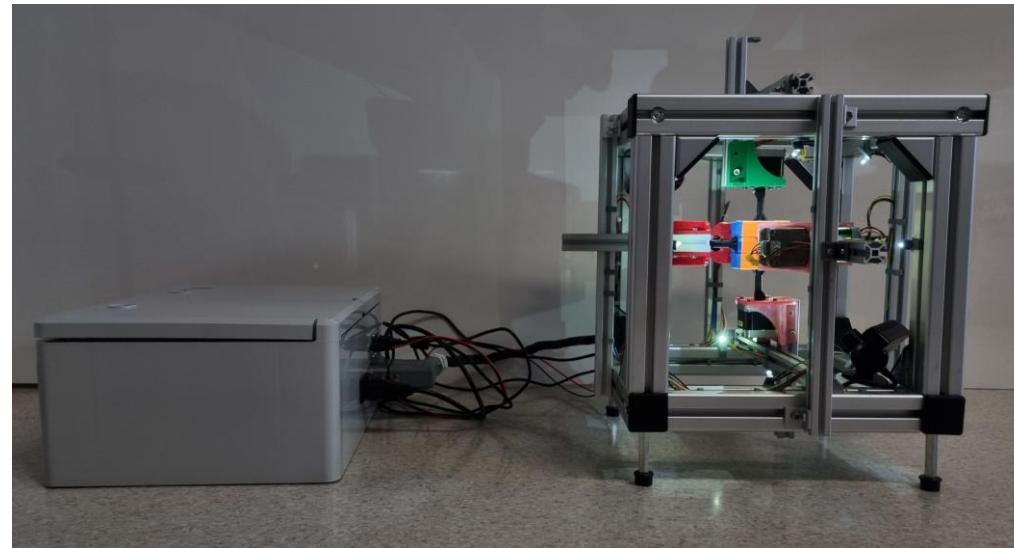
- Learn to maneuver ball through labyrinths (virtual and physical)
- Learn to play pinball
- Learn to play the goalkeeper in table soccer
- Learn to solve pocket and Rubik's cubes



Deep reinforcement learning (and image processing):

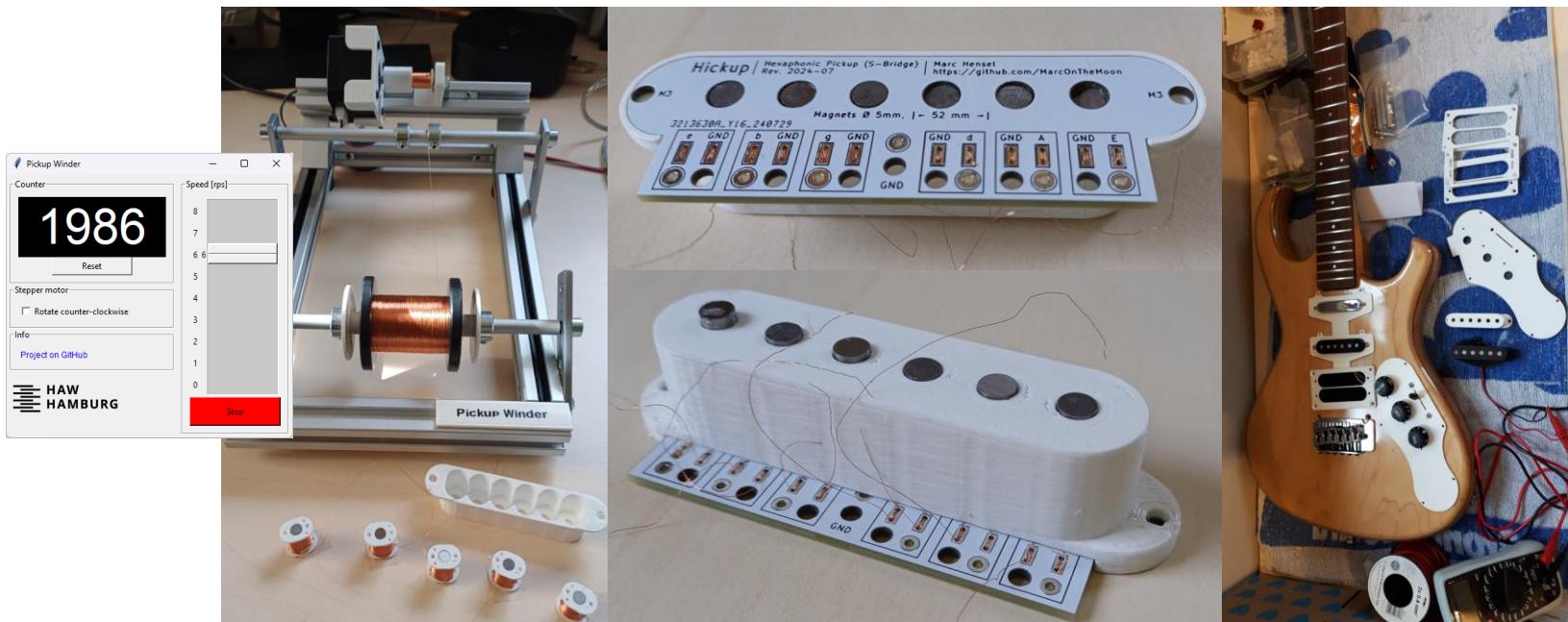
- Learn to maneuver ball through labyrinths (virtual and physical)
- Learn to play pinball
- Learn to play the goalkeeper in table soccer
- Learn to solve pocket and Rubik's cubes

Solving machine for Rubik's cubes (unscrambles in < 1 s):



Hexaphonic guitar and audio signal processing:

- Pickup development
- MIDI conversion



Supported by No.1 Guitar Center GmbH (audio interface)

Coming together from all over the world



- Who was born closest to Hamburg?
- Who was born furthest away from Hamburg (to north, east, south, west)?
- On which continent were you born?



What we want to achieve

You shall become “slightly more of an engineer” – meaning you *can do* more!

You *develop* console applications in the C programming language (e.g., to process digital images or determine distances between geographical locations) – by

- analyzing given source code,
- implementing given functional requirements, and
- implementing, debugging, and testing code in a professional IDE*.

* IDE: Integrated development environment (i.e., tool for programming)



Lectures:

- 13 **14** lecture sessions
- Participation is not mandatory

Notes:

- Avoid disturbances (e.g., be punctual, mobile phone off)
- Questions, lively discussions, and doing programming together are absolutely welcome
- Lots of review questions, programming tasks (with sample solutions), and lecture codes are provided in the workbook and as software projects.



Lecture

How we shall achieve it

Laboratory sessions:

- 5 laboratory sessions
- Dr. **Dennis Schüthe** and I will take turns in the labs SOL1.
- Successful participation in all sessions is *mandatory*
- Lab assignments are already available in the workbook.



Notes:

- You must be well-prepared for every lab.
- You must pass every lab (or will not be allowed for the lab exam).
- Teams of 2 – 3 people working together (*no one* working by himself / herself)



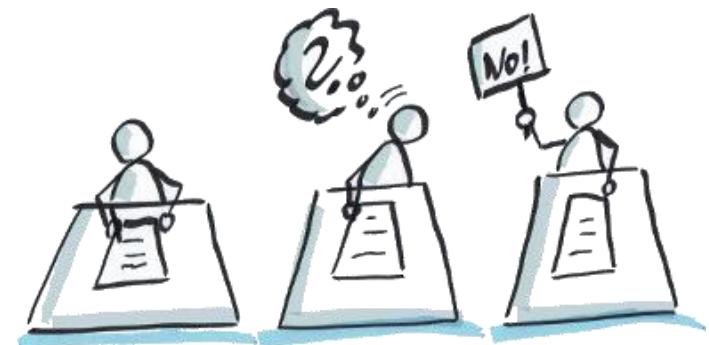
Laboratory

Laboratory exam:

- Graded in 0 (“poor”) to 15 (“excellent”), exam passed for ≥ 5
- Create source code for given functional requirements (\Rightarrow *Doing!*)
- Can also include, e.g., code analysis and questions regarding general understanding

Notes:

- The exam is mainly about *doing!*
- Available time is typically 180 minutes (in the PC pool, with German keyboard)
- Electronic submission of results
- Not permitted to use source codes, books, or such (to be confirmed)



Moodle (eLearning):

- <https://moodle.haw-hamburg.de/>
- Subscription key: Wth6waG
- Links to downloads (e. g., weekly plan, lecture slides, old exams)
- Distribution list for announcements (e. g., change of dates)

⇒ Register to this course in Moodle today!



GitHub repository:

- https://github.com/MarcOnTheMoon/coding_learners
- Workbook with exercises, questions, lab assignments, and old exams
- Sample solutions to exercises and exams
- Sample projects used in the lectures



The “bad news” section

In German:

Das Leben ist kein Ponyhof. (Life is no bowl of cherries.)



There is a key to success, but it is not secret:

- *Doing from the very beginning on (Be curious. Try things out.)*
- *Doing the lab courses by yourself*

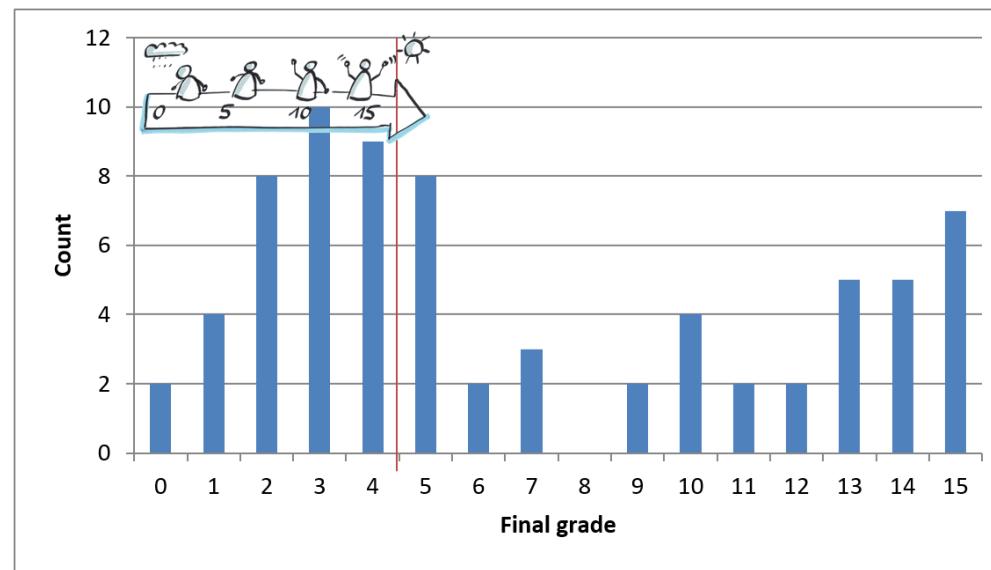


C is a language. And there is no other way of learning a language than speaking.

Exam results

To tell you the truth:

- In Germany we do use the full range of grades. \Rightarrow Yes, you can fail!
- For instance, in winter 2023/24 almost 50 % of the students failed.
- Make sure you are on the right side of the red line.



Take this very seriously:

- In Germany, students *do* get “kicked out” of universities.
- We are no exception to this.

Be aware:

- You are responsible to register for labs and/or exams.
- In case of doubt, you must prove that registration was successful.
- Make sure to register before registration deadlines.

Registration and contacts:

- Registration in *myHAW* (<https://myhaw.haw.tuhh.de>)
- Students typically get informed by the Faculty Service Bureau (FSB).
- Contact: FSB (Berliner Tor 21, stairs at left of entrance)

If you are not on the list of participants for an exam:

- Lecturers *must not* let you participate.
- If this happens, hurry to the FSB for clarification.

- Germany has strict laws on data protection and personal rights.
- Taking photos, videos, audio recordings, or such of a lecture can be a criminal offense!



- ⇒ Don't do it! (Using a mobile in a lecture is very unpolite, anyhow ...)
- ⇒ Don't even give that impression (e.g., by holding a mobile phone in your hand).

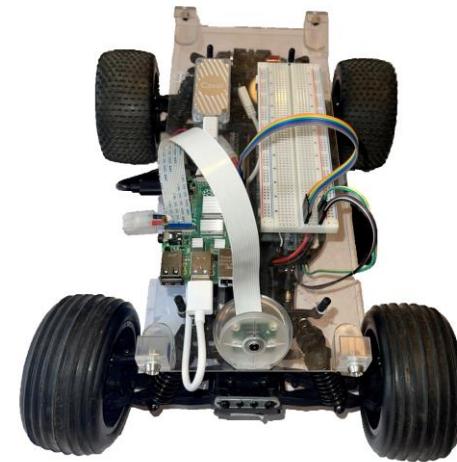
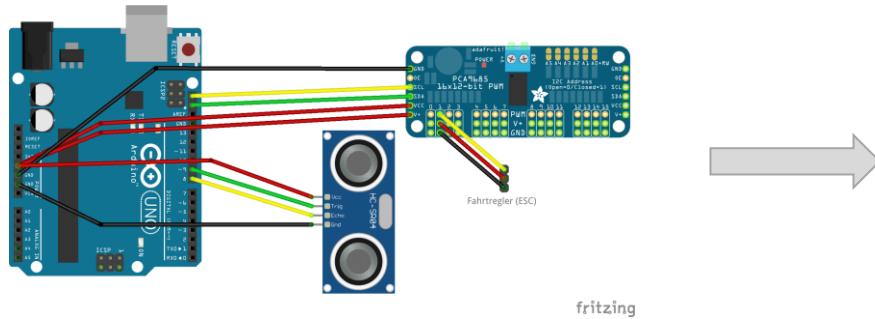
By the way, would you like me to take videos of you sleeping in a lecture? 😊

Some good news

- Life is no „Ponyhof“, but after all, it should be fun!

So, let's talk about good news:

1. Working with Arduino boards is fun!
 2. Arduino boards and components are not expensive.
 3. If time allows, I may give you a practical introduction to Arduino (e. g., control RC car).
 4. Arduino is out of scope for the exam.



Introduction:

https://github.com/MarcOnTheMoon/coding_learners



Any other business

- Student feedback
- Agenda



What do you think?

- What is frequent students' feedback on software lectures?

Some feedback:

- *"He proceeds way too fast!"*
- *"He proceeds way too slow!"*

Introduction:

- Overview of lecture topics
- Introduction to the C programming language



You can create and execute a small application (“doing”):

- Set up development environment
- Principal structure of a small C application
- Running an application

First steps in C, if time allows:

- Data types and variables
- Displaying data on the screen

Let's not forget these important points:

- Naturally, we will have a small break.
- Further agenda for the semester: Have fun learning together ☺ ...

Software Construction 1 (IE1-SO1)

1. Introduction



What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *establish* a meaningful relation between the topics discussed in the lecture.
- You *create* and *execute* a simple C application in the integrated development environment Visual Studio.



1. Lecture overview
2. C programming language
3. Programming fundamentals
4. Required software & first application
5. Source code example
6. Literature

Lecture overview

Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

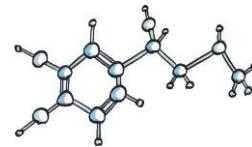
Advanced topics



5. Pointers



6. Memory management



7. Structures



8. Lists and sorting

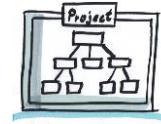
The next steps ...



9. Input and output

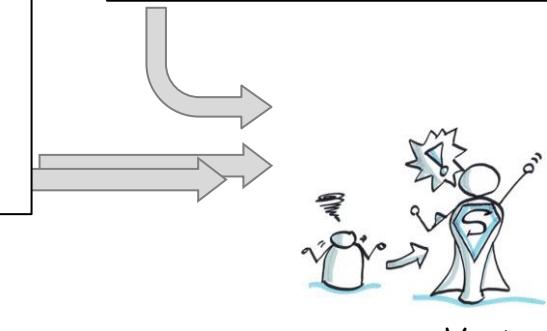


10. Bit operations ("magic")



11. Project and preprocessor

● Location



You!

C programming language

- Properties
- Major versions



Let's have a little quiz:

- In which year were you born? (Okay, no quiz – you should *know* that.)
- In which year was C “born”? (Just guess ...)

Answer:

- Dennis Ritchie developed C from 1969 – 1973
- Purpose: Language to program operating system UNIX
- Language managed by:
 - Before 1990: ANSI (American National Standards Institute)
 - Since 1990: ISO (International Organization for Standardization)

Be aware:

- C is old enough to be your parent
- But still of significant importance (as parents should be)

- You will need it!
- Do you know engineers in information technology that do not program at all?
- Creating software is fun! (Just think of own Android apps in Java.)

Why C?

- Good foundation as many other languages are similar to C (e.g., C++, Java, C#, VHDL)
- Wide-spread
- Efficient (small code, can generate fast applications)
- Flexible (e.g., system, microcontroller, and application programming*)
- Portable to other computer platforms (i.e., compile same code for different systems)
- Easy to learn (small set of keywords)
- Powerful (e.g., hardware-related programming)

* We restrict to desktop applications and will use the terms *application* and *program* interchangeably.

It helps to know C's major versions to avoid confusion:

- Several standards have been released
- Compilers might not support some features introduced after 1990

Major versions:

When	Name	Alias	Comment
1978	K&R C	Classic C	Language description in book by Brian Kernighan and Dennis Ritchie
1989	C89	ANSI C	First official standard (by ANSI)
1990	C90	ISO C	Standard by ISO, but essentially the same as C89
1999	C99		Not fully implemented by all compilers
2011	C11		Not fully implemented by all compilers, some C99 feature now optional

Practically
the same

- Our main interest: C89/C90 (= ANSI/ISO C)
- Will mention C99 and/or C11 at times, but compilers might *not* support all features

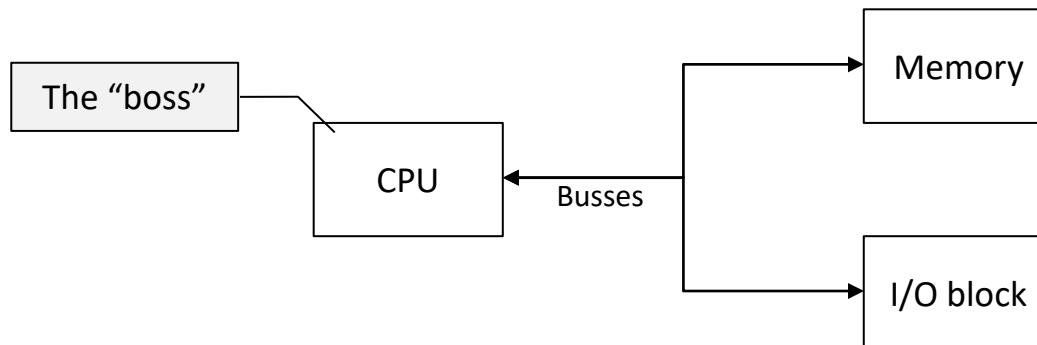
Programming fundamentals

- Why programming languages?
- Building a program

- Read the headline carefully: *Computers think in numbers*.
- All data is represented by numbers (including characters, e.g., 'H').
- All instructions are coded as numbers.

A computer contains (very high-level):

- *Central processing unit* (CPU) doing the “work” (e.g., adding numbers)
- *Random access memory* (RAM) for storing information (program code and user data)
- *Input/output* (I/O) block connecting to other devices (e.g., keyboard, display, printer)
- *Data bus, address bus, and control bus* to exchange information between the components



Example: Adding two numbers

Computer:

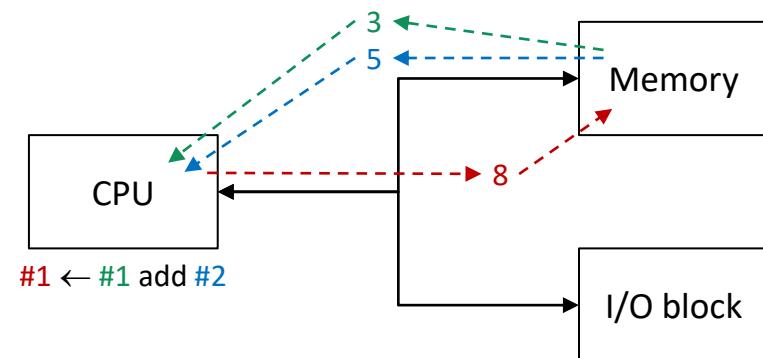
1. Copy 1st number from a memory location into CPU register #1
2. Copy 2nd number from a memory location into CPU register #2
3. Add contents of registers #1 and #2 and leave the result in register #1
4. Copy result from register #1 to a memory location

Notes:

- Memory locations are represented by a number (*address*).
- CPU registers are represented by a number.
- Instructions *copy* and *add* are represented by a number (numeric code).

Humans:

- Higher abstraction level (e.g., $a = b + c$)



The dilemma:

- Computers understand numbers (data, operations, addresses), but no human language.
- Humans understand natural language, but are typically not good at numeric codes.

Programming languages:

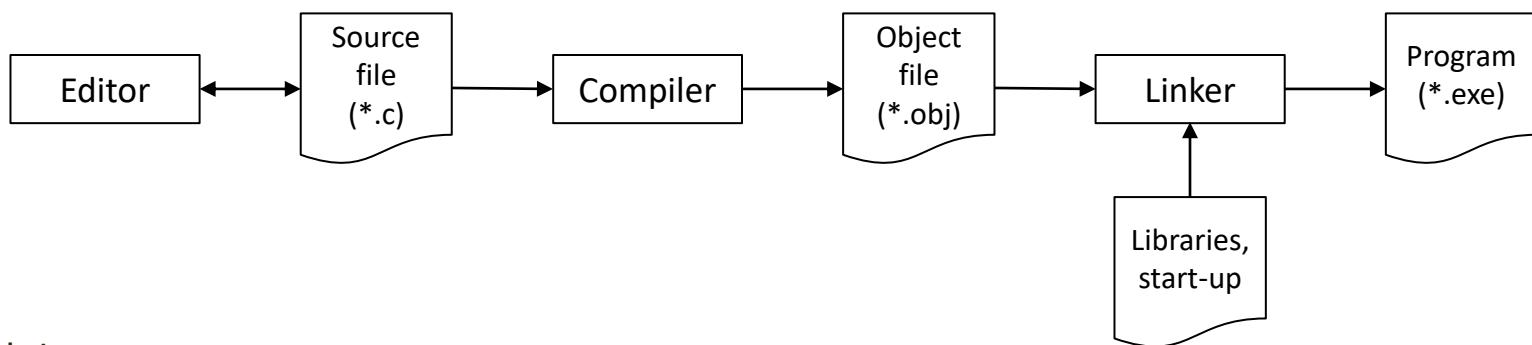
- Bridge the “language gap” between computers and humans
- Humans program in a human-readable programming language (*source code*).
- Source code is translated to computer-readable CPU instructions (*machine code*).

Notes:

- C is a *high-level programming language* (i.e., strong abstraction from computer details).
- C is an *imperative* language consisting of *statements* (i.e., commands) to execute.

Steps in building C programs:

1. Programmer writes source code in an *editor* (→ Source files: *.c)
2. A *compiler* translates source files into machine code (→ Object files: *.obj)
3. A *linker* combines object files into an executable program (→ Executable file: *.exe)



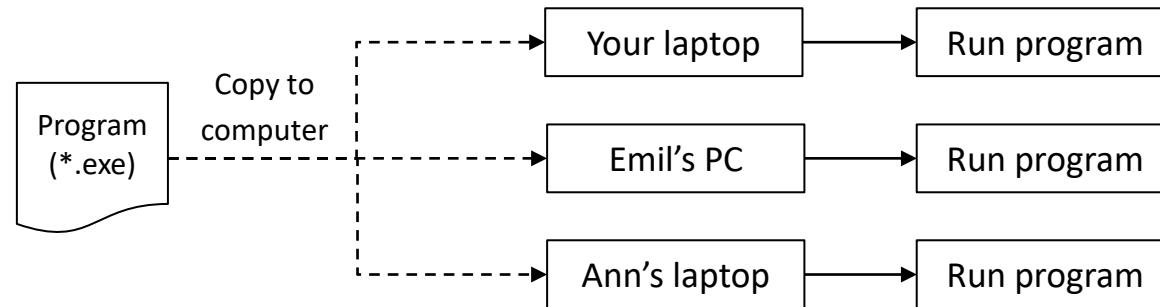
Notes:

- Machine code consists of, e.g., the CPU instructions
- The linker adds the following:
 - Required libraries (e.g., pre-compiled standard functions)
 - Code to make a file executable (*start-up code*)

Executing a C program

Typical approach (for small programs):

1. Program written, compiled, and linked on a developer's computer
2. Final program copied to and executed on user computers

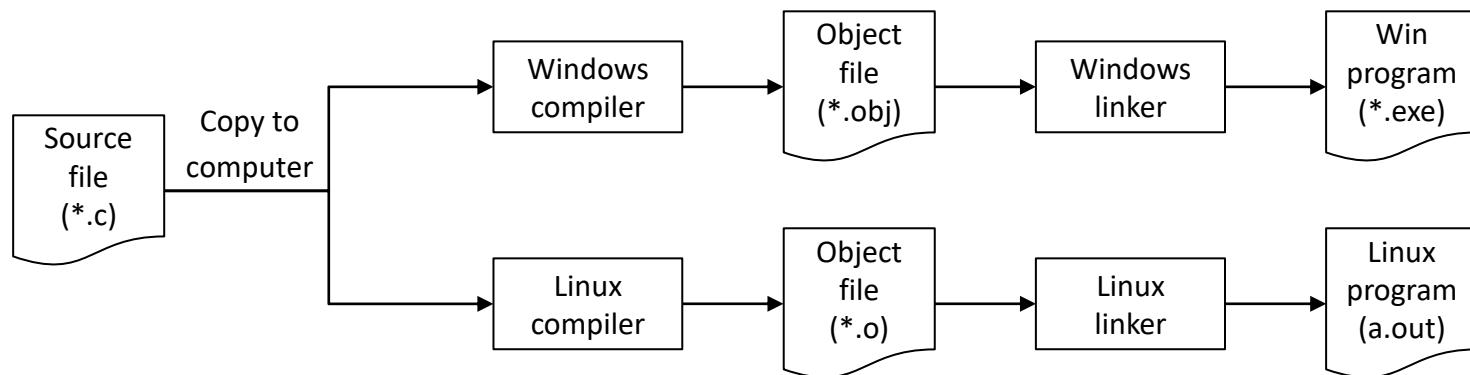


Notes:

- Programs are copied to other computers.
- Users do *not* get source files, but executable files only.

- *Portability*: Transfer a program to a different computer system (*platform*)
 - Machine code differs for different systems
 - A compiler translates source code into machine code for a particular system
- ⇒ Must translate source code specifically for each target system

Example:



Required software & first application

- Development software
- Step-to-step description to create a simple program

- Several compilers and integrated development environments (IDE) exist.
- In this lecture we use *Microsoft Visual Studio 2019/2022* on *Microsoft Windows* systems.
- During installation choose *Desktop Development with C++* (or similar).

Microsoft Visual Studio:

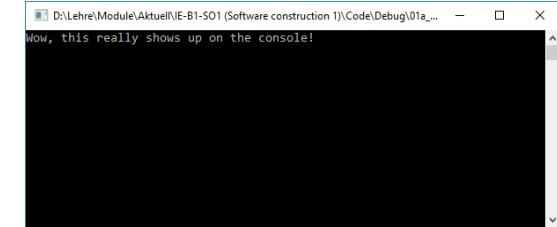
- Professional IDE (contains editor, compiler, linker, debugger, and more)
- Includes *Microsoft Visual C++ (MSVC)* compiler:
 - For programming languages C and C++
 - We restrict to C in this lecture.
- Widespread IDE in software industry
- Download:
 - [Microsoft Azure for education](#) (sign in with your HAW “w account”)
 - You may also use the direct link to [Azure development software](#).
 - Free [Community edition](#) available from Microsoft

Let's create and execute a C program in Visual Studio step-by-step:

1. Create so-called *solution* and *project* in Visual Studio
2. Add source file and write source code for a simple application
3. Compile and link the source code
4. Run the executable program

Notes on *projects*:

- Contain data needed to build a program
- Include files like source code, specific settings, and so on
- We restrict to programs that run in a text *console* window.

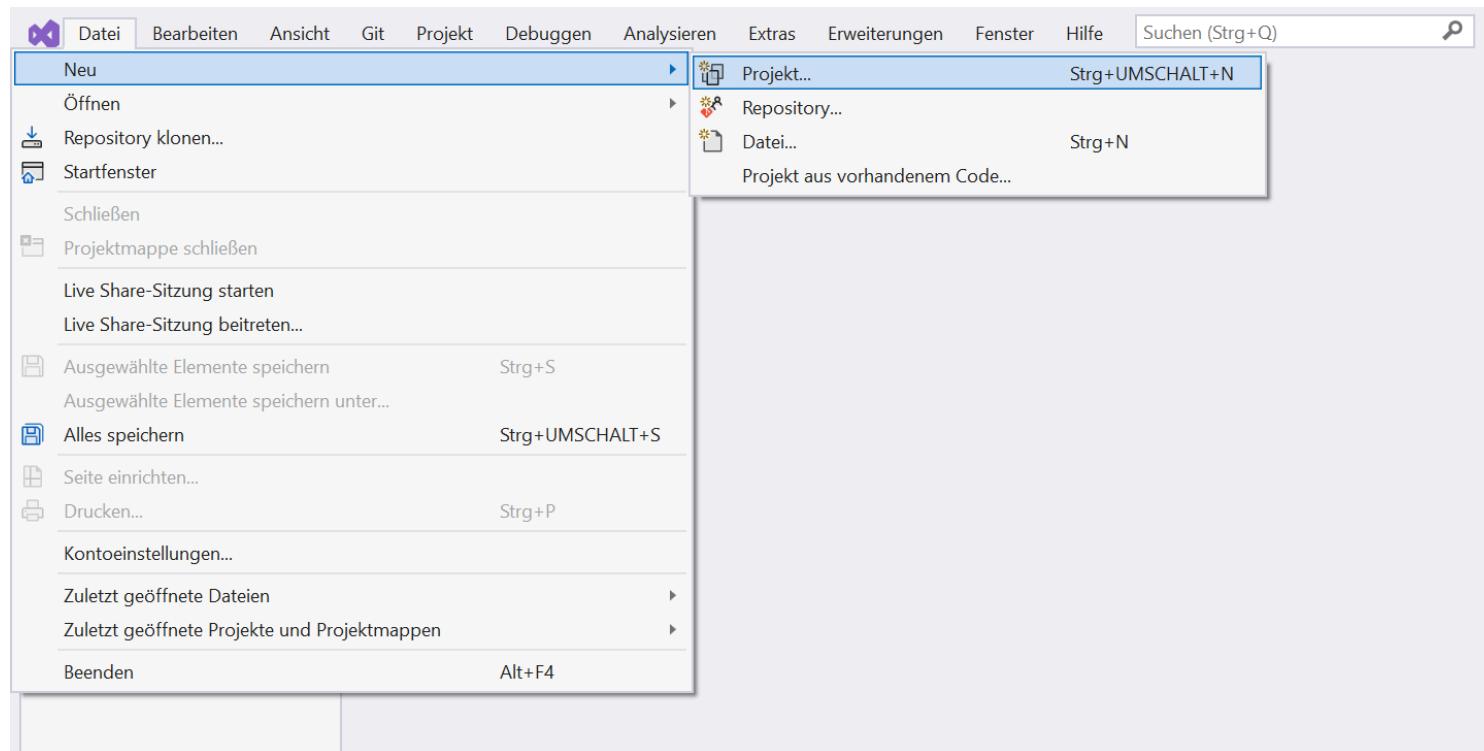


Notes on Visual Studio *solutions*:

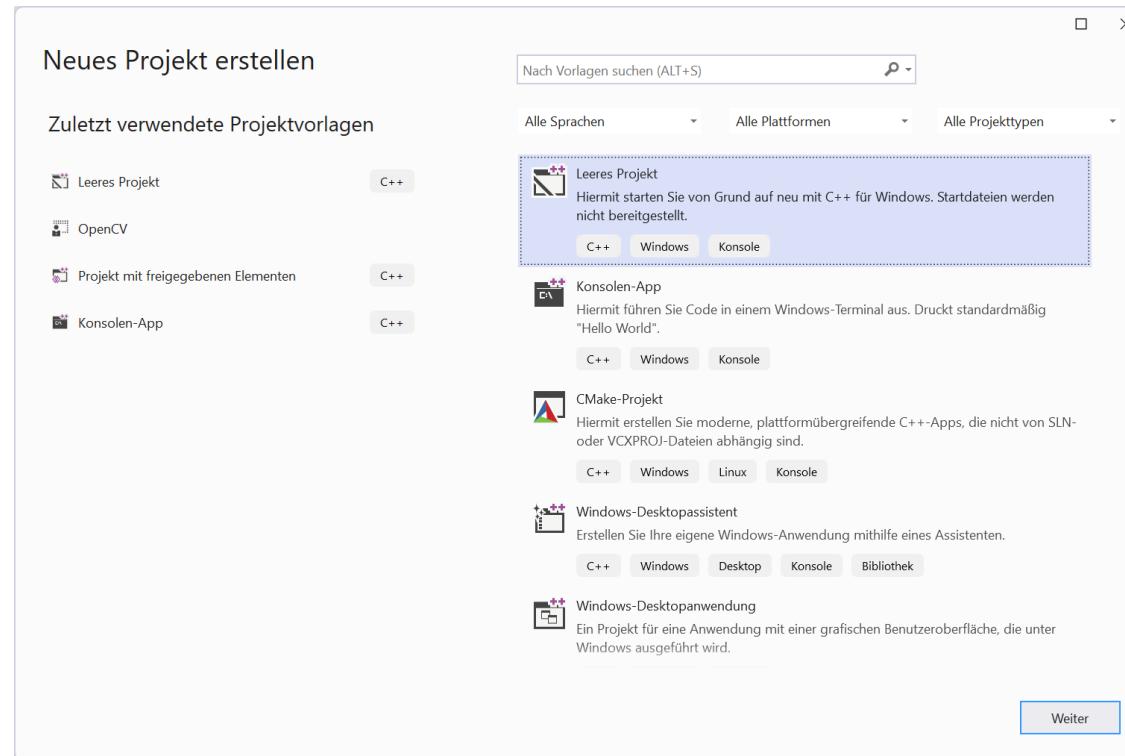
- Can contain several projects
- For each chapter we will create a solution containing *all* related projects.
- Open a solution, e.g., by double-click of the related *.sln file in the Windows Explorer.

Let's write a program:

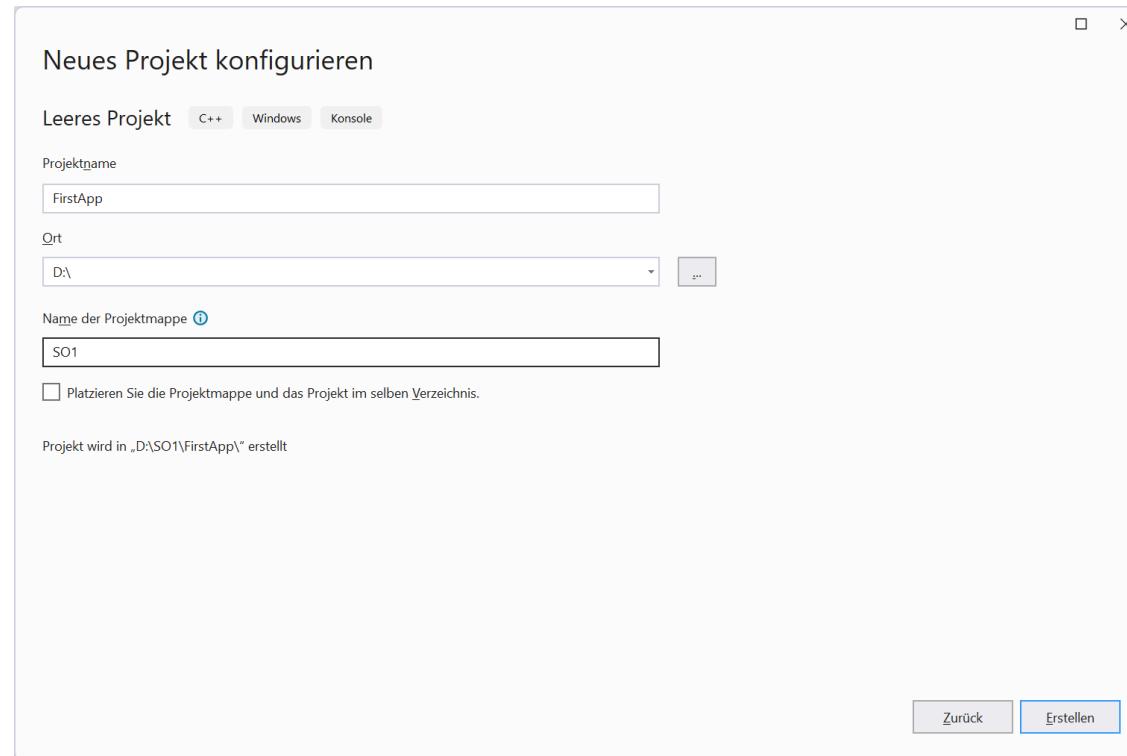
1. Start Visual Studio
2. Select *New Project* (start screen, upper left area, or *File / New* menu)



3. Search for and select *Blank Solution*

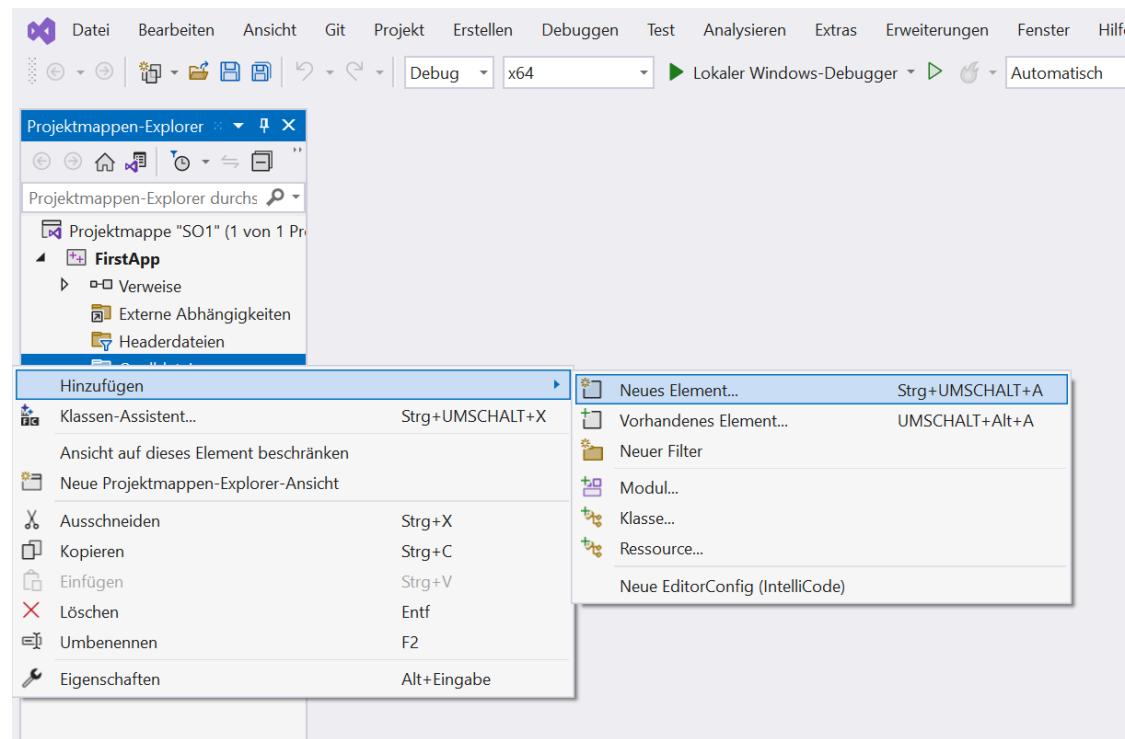


4. Enter project name (e.g., *FirstApp*) and location (e.g., D:\) in file system
5. Uncheck the check box and enter solution name (e.g., *SO1*)



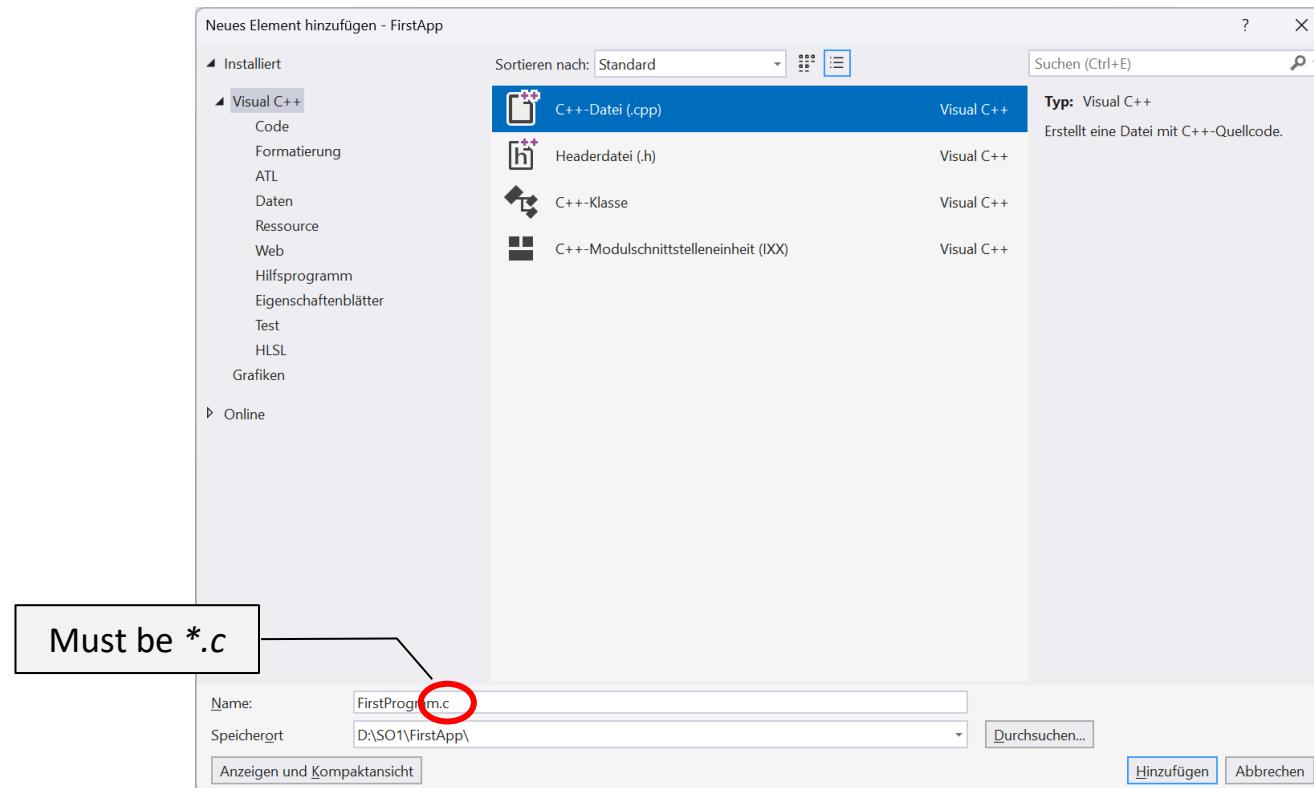
Add a source file

6. Right-click on *Source Files* in solution explorer and select *Add / New Item*



Add a source file

7. Select *C++ File*, enter a name (e.g., *firstProgram.c*), and press the *Add* button



Notes:

- The file extension must be **.c* and not **.cpp*.
- Else the compiler treats the source code as being written in C++.

Write source code

8. Double-click source file in solution explorer, if the (empty) file is not show in the editor.
9. Insert the text below.

The screenshot shows a code editor window titled "FirstProgram.c*" with the file path "FirstApp". The code is as follows:

```
#include <stdio.h>
int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

Annotations with callouts:

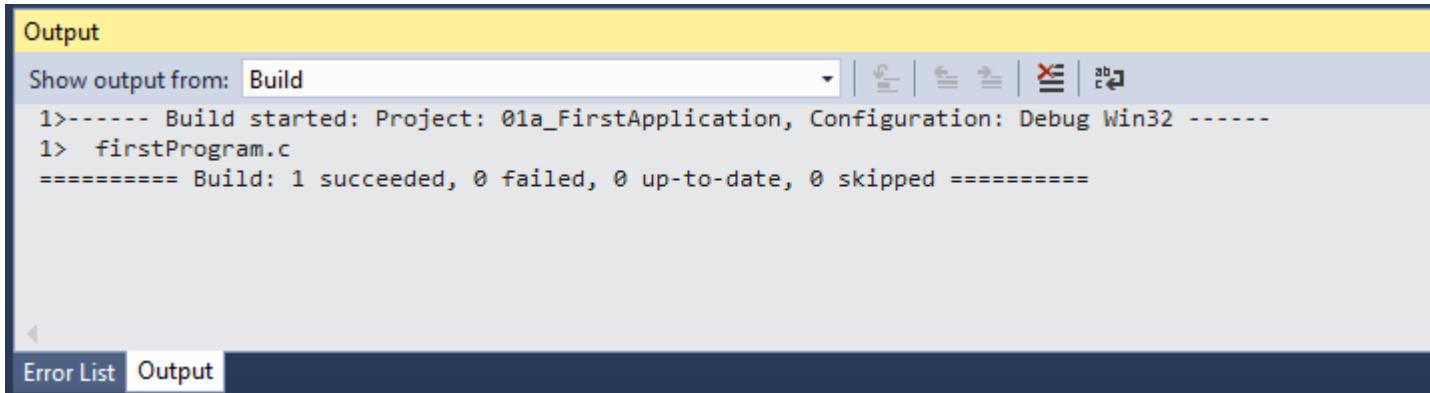
- A box labeled "File name" points to the window title "FirstProgram.c*".
- A box labeled "Line numbers" points to the vertical column of numbers on the left (1 through 9).
- A box labeled "Source code" points to the text area of the code editor.
- A box labeled "Yellow: New code (not yet saved)" points to the line "printf("Wow, this really shows up on the console!\n");".

Notes:

- Do not care about the meaning of the source code at this moment.
- We will come to that in a minute.

Compile and link the source code

10. Compile the source code by selecting *Build / Compile* or *Ctrl-F7*
11. Output area shows progress and should result in *succeeded* status

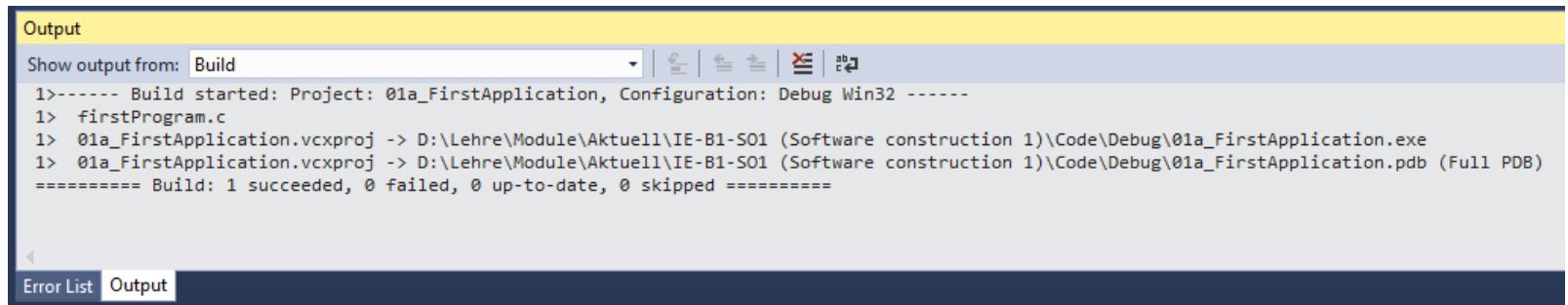


The screenshot shows the 'Output' window from a development environment. The title bar says 'Output'. The dropdown menu says 'Show output from: Build'. The main text area displays the following build log:

```
1>----- Build started: Project: 01a_FirstApplication, Configuration: Debug Win32 -----
1> firstProgram.c
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

At the bottom, there are tabs for 'Error List' and 'Output', with 'Output' being the active tab.

Note: Selecting *Build / Build 01a_FirstApplication* additionally links and creates the *.exe file.



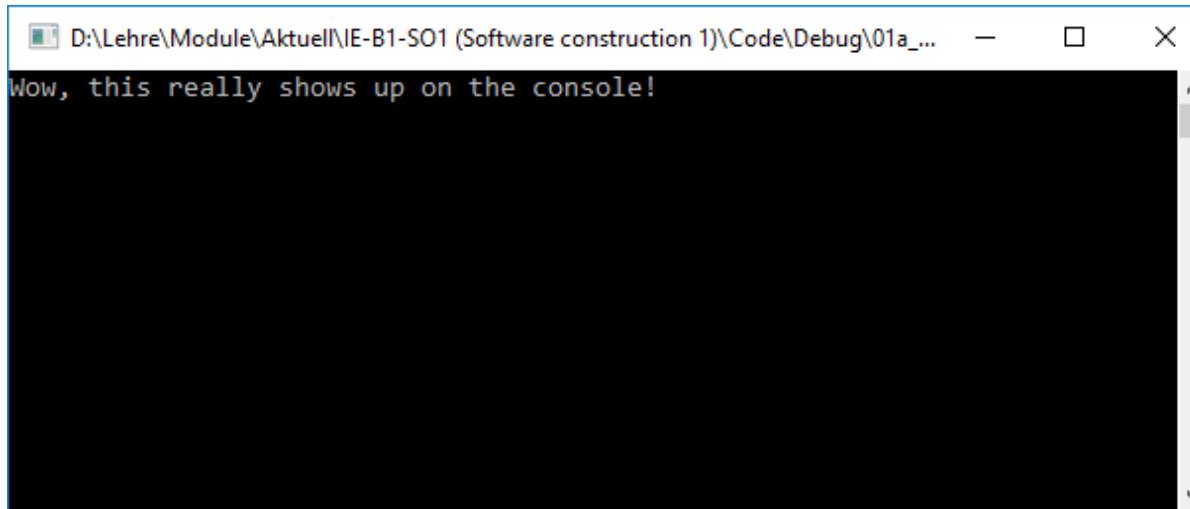
The screenshot shows the 'Output' window from a development environment. The title bar says 'Output'. The dropdown menu says 'Show output from: Build'. The main text area displays the following build log:

```
1>----- Build started: Project: 01a_FirstApplication, Configuration: Debug Win32 -----
1> firstProgram.c
1> 01a_FirstApplication.vcxproj -> D:\Lehre\Module\Aktuell\IE-B1-SO1 (Software construction 1)\Code\Debug\01a_FirstApplication.exe
1> 01a_FirstApplication.vcxproj -> D:\Lehre\Module\Aktuell\IE-B1-SO1 (Software construction 1)\Code\Debug\01a_FirstApplication.pdb (Full PDB)
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

At the bottom, there are tabs for 'Error List' and 'Output', with 'Output' being the active tab.

Run the application

12. Run the application by selecting *Debug / Start Without Debugging* or *Ctrl-F5*
13. A console window displaying some text opens. Press *Enter* to close the window.



Notes:

- Pressing *Ctrl-F5* will automatically compile and link, if required (i.e., no need for *Ctrl-F7*).
- You can execute *01a_FirstApplication.exe* by, e.g., a double-click in the Windows Explorer.

For solutions containing ≥ 2 projects:

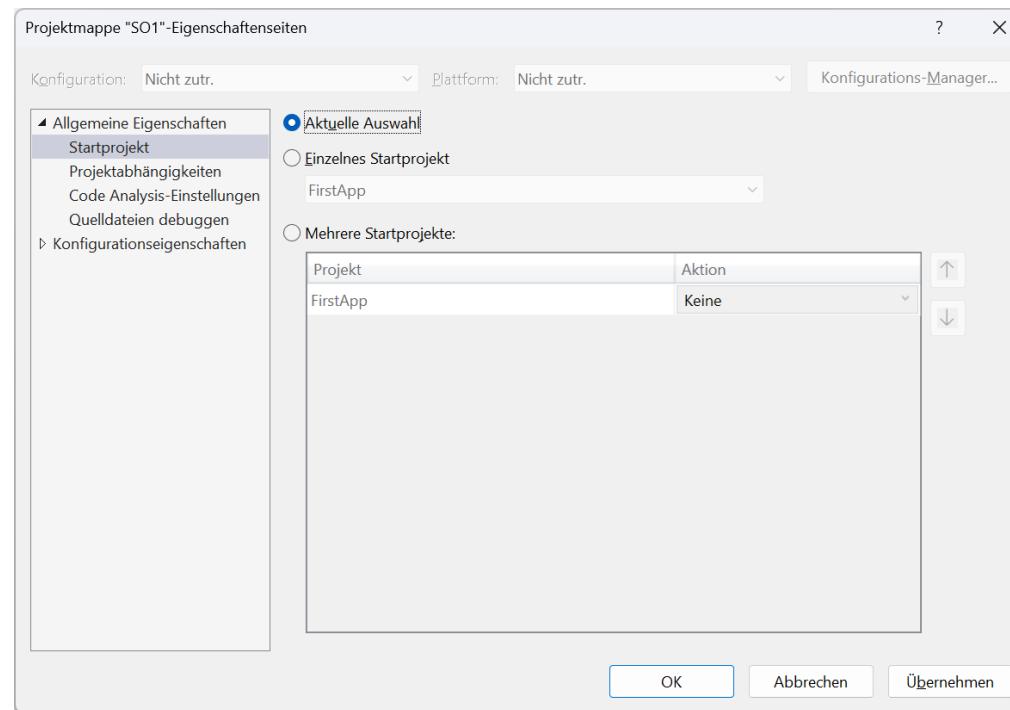
- Visual Studio regards one project to be the startup project.
- The startup project is indicated by **bold letters** in the solution explorer.
- Visual Studio will build and/or execute this project.
- Right-click a project and select *Set as StartUp Project* to change the startup project.

Often more convenient:

- One is editing a source code file in Visual Studio.
- Visual Studio shall build and/or execute the project belonging to this source file.

For this behavior:

1. Right-click on the solution name and select *Properties*
2. Select *Common Properties / Startup Project / Current Selection*



Source code example

- Source code explained in detail

Source code example



What, do you think, is the effect of each line in following sample code?

```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

C programs consist of one or more *functions* (modules).

```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

Declaration with name

Body with statements

- A function consists of:
 1. A *declaration* as first line (with the function's name followed by parentheses)
 2. A *body* containing the code to execute surrounded by braces {...}
- Above we have only one function with name *main*.

The main() function

Every C program has a function with name *main*.

```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

A red rounded rectangle highlights the code block from `printf` to `getchar()`. A brace to the right of the highlighted block is labeled "Executed as program".

- Starting a program ⇒ Operating system executes the *main()* function
- Program terminates when *main()* ends



The main() function

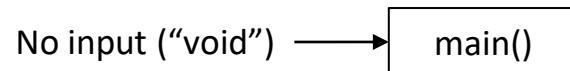
The function below does not receive input data.

```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

} *void: No input*

- Functions can receive input data (*arguments* = values).
- Type of data is specified in parentheses after the function name (*parameters* = variables)
- Here *void* (= empty) indicates that *main()* doesn't expect data from the operating system.



The main() function

The function below returns a value.

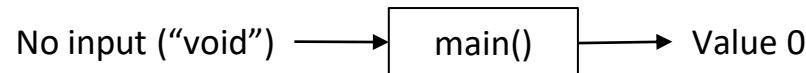
```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

} *int: Returns an integer number*

} *End function by returning 0*

- The keyword *int* indicates that *main()* returns an integer number.
- The *return* statement ends *main()* and passes 0 to the operating system.



The `printf()` function displays text on the console.

```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

} Print on console window

- Name `printf` is followed by parentheses \Rightarrow `printf()` calls a function
- Parentheses contain the text to display on console.
- Texts are included in quotation marks “...”.
- The character `\n` within the text creates a line break (“new line”).

Wait a minute!

- Where is `printf()` defined? There is no such function in our source code!



The *printf()* function is included from a library.

```
#include <stdio.h>
}
int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```

} Declare *printf()* function

- Lines starting with a hash mark ('#') are commands for the *preprocessor*. They are executed *before* compilation into machine code.
- The preprocessor replaces `#include <fileName>` with the contents of file *fileName*.
- File *stdio.h* (standard input/output) declares functions, e.g., for input from the keyboard and output to the display.
- The linker adds the pre-compiled *printf()* function when creating the executable program.

The `getchar()` function

The `getchar()` function waits for the user to press a key (e.g., *Enter*).

```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar(); } Wait for user input
    return 0;
}
```

- When `main()` ends the operating system closes the console window.
- Function `getchar()` waits for the user to press, e.g., the *Enter* key before continuing.
- The function is declared in `stdio.h`.

Think about it:

- What would happen, if we omit `getchar()`?



Some final remarks ...

```
#include <stdio.h>

int main(void)
{
    printf("Wow, this really shows up on the console!\n");
    getchar();
    return 0;
}
```



Statements:

- Commands within a function (e.g., *main()*) are called *statements*.
- Every statement *must* end with a semicolon (';').
- The semicolon is not a separator, but part of the statement.

Braces:

- Braces {...} group the statements of a function.
- You can also use braces within a function body to further group statements.

Literature

The “best” book is the one that you – *personally* – like the most to learn with!

English:

- Stephen Prata: *C Primer Plus*, Addison Wesley
- Robert Martin: *Clean Code*, Prentice Hall

(*> 1000 pages, well-written*)
(*Okay, not C, but brilliant*)

German:

- Thomas Theis: *Einstieg in C*, Galileo Computing
- Peter Prinz: *C – Das Übungsbuch*, mitp

(*Introduction for beginners*)
(*Questions and exercises*)

Introduction to Arduino (out of scope for the exam):

- SparkFun Inventor's Kit Experiment Guide

(*Free PDF download*)

... or any other related book you prefer.

There are more related books than you could carry. ☺



Software Construction 1 (IE1-SO1)

2. Data types



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

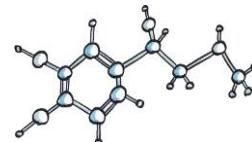
Advanced topics



5. Pointers



6. Memory management

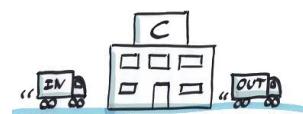


7. Structures



8. Lists and sorting

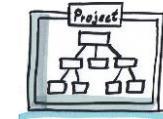
The next steps ...



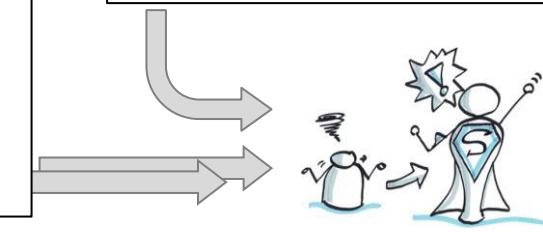
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



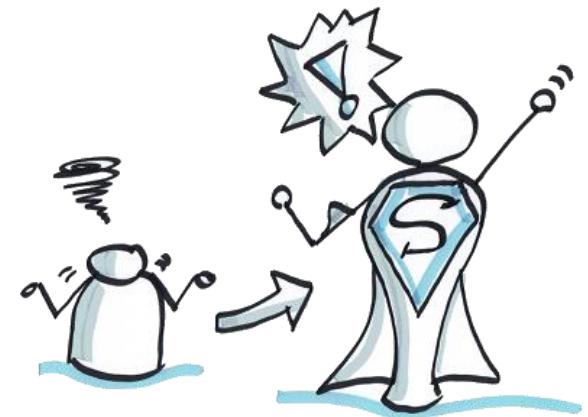
You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *use* data types to represent and manipulate numbers and characters.
- You *display* texts including numerical values in a formatted way on the console.
- You *read* numbers and characters from the keyboard.
- You *apply* comments and rules for programming (*coding style*) to improve the quality and maintainability of source code.



1. Program structure revisited
2. Introducing integer & floating point numbers
 - Numbers
 - Keyboard input
3. Comments & identifiers
4. Data types in detail
 - Numbers and characters
 - Constants
5. Operators
6. Type conversions
7. Coding style

Program structure revisited

- Source code of a small program in detail

Let us recap the general program structure by following minimal program:

```
int main(void)
{
    return 0;
}
```

- Every program has exactly one *main()* function.
- Running the program executes the statements within the braces {...}.

Declaration of input and return types:

- Keyword *void* specifies that no arguments are passed to *main()*.
- Keyword *int* specifies that *main()* ends by returning an integer number.

The *return* statement ends the function:

- The *return* keyword is followed by the number to return (here: 0).
- Value 0 indicates to the operating system that no error occurred during execution.
- Every statement *must* end with a semicolon.

Let us recap the general program structure by following minimal program:

```
int main(void)
{
    return 0;
}
```

Okay, this is a valid C program, but...

- What does the program do?



Let us recap how to print text to the console:

```
#include <stdio.h>

int main(void)
{
    printf("Well, here we go ...\\n");
    return 0;
}
```

- The *printf()* function (“print formatted”) displays text on the console.
- Character \n creates a line break (“new line”).
- Need to add the *include* directive, because *printf()* is not part of the language
- Preprocessor replaces that directive by the contents of file *stdio.h* before compilation

Some more details on *include*:

- By including *stdio.h* the *printf()* function is “known” in a program.
- This does not include the source code for *printf()*. The linker takes the pre-compiled function from a library and links it to the executable.

Let us recap how to keep the console window open:

```
#include <stdio.h>

int main(void)
{
    getchar();
    return 0;
}
```

- Use `getchar()` to wait for users to press a key (e.g., *Enter*).
- Need to add the include statement, because `getchar()` is not part of the language

There is something special about *main()* functions:

- Allowed to omit the *return* statement
- In this case the function implicitly returns 0.

Example:

```
#include <stdio.h>

int main(void)
{
    printf("Well, here we go ...\\n");
}
```

- Allowed for *main()* function, only
- Recommended to explicitly write *return* statement, anyhow

Introducing integer numbers

- Variables for integer numbers
- Print values to the console window
- Arithmetic operations on numbers



We have an issue:

- Printing text samples to console is boring!
- How can a program “remember” data such as numbers?

Solution:

- Store data in memory
- Define *how* the data is coded in memory (*data type*)
- Access the data using a name (*identifier*)



Try to explain the following source code:

```
#include <stdio.h>

int main(void)
{
    int zipCode;

    zipCode = 20099;
    printf("HAW Hamburg\nBerliner Tor 7\n%d Hamburg\n", zipCode);
    getchar();
    return 0;
}
```

Try to explain the following source code:

```
#include <stdio.h>

int main(void)
{
    int zipCode;
    Declare variable

    zipCode = 20099;
    Assign a value

    printf("HAW Hamburg\nBerliner Tor 7\n%d Hamburg\n", zipCode);
    Print to console

    getchar();
    return 0;
}
```

1. Declare a variable:
 - Data type is *int* ⇒ Reserve memory to store an integer number
 - We can access the memory using the identifier *zipCode*
2. Assign the value 20099 (i.e., store 20099 at the memory location referenced by *zipCode*)
3. Print text including the number to console (%d is placeholder for an integer number)

Declaration:

- Variables must be *declared* before using them (i.e., state data type and name).

```
int zipCode;
```

Assignment:

- Subsequently values can be *assigned*.
- The right side of the equal sign is assigned to the left side.

```
int zipCode;  
zipCode = 20099;
```

- Note: $a = a + 7$ allowed \Rightarrow Equal sign has *not* the same meaning as in math!

Definition:

- Variables can be *defined* (i.e., declared *and* initialized with a value) at creation:

```
int zipCode = 20099;
```

Declaration:

- Declare each variable in a separate line:

```
int zipCode;  
int houseNumber;
```

- Alternatively declare variables in a single line (with comma-separated names):

```
int zipCode, houseNumber;
```

Definition:

- Again, declarations may include assignments of initial values.

```
int zipCode = 20099, houseNumber = 7;
```

ISO C90:

- All variables must be declared at the beginning of a *block* surrounded by braces {...}.
- No variable declarations allowed after other kinds of statements (e.g., *printf()*)

ISO C99:

- Variable declarations allowed anywhere in a block

Example:

- Is the following allowed in ISO C90?



```
int main(void)
{
    int zipCode;
    zipCode = 20099;
    int houseNumber = 7;
    return 0;
}
```

Conform to ISO C90:

```
int main(void)
{
    int zipCode;
    int houseNumber = 7;

    zipCode = 20099;
    return 0;
}
```

The diagram illustrates the structure of the provided C90 code. It shows two main categories: 'Declarations' and 'Other statements'. The 'Declarations' category is indicated by a bracket pointing to the first two lines of code, which define variables 'zipCode' and 'houseNumber'. The 'Other statements' category is indicated by a bracket pointing to the assignment statement 'zipCode = 20099'.

- Conform, because all declaration statements are *before* other kinds of statements

Note:

- Regarded good coding practice to declare all variables at beginning of a block
- This gives an overview of all variables used.

- Use `printf()` to print integer numbers to console
- Characters `%d` act as placeholder for an integer (d: decimal)
- Place text (in double quotes) and comma-separated integer value inside parentheses.



Example:

- What is printed to the console?

```
int zipCode = 20099;  
printf("HAW Hamburg\nBerliner Tor %d\n", 7);  
printf("%d Hamburg\n", zipCode);
```

Integer value

Value taken from variable

- Console output:

```
HAW Hamburg  
Berliner Tor 7  
20099 Hamburg
```

More than one integers in a single call of *printf()* :

- To print several integers add %d for each value and comma-separate the values
- Make sure that the number of %d matches the number of values in *printf()*!

Examples:

- Constant value and variable:

```
int zipCode = 20099;  
printf("HAW Hamburg\nBerliner Tor %d\n%d Hamburg\n", 7, zipCode);
```

- Two variables:

```
int houseNumber = 7;  
int zipCode = 20099;  
printf("HAW Hamburg\nBerliner Tor %d\n%d Hamburg\n", houseNumber, zipCode);
```

C defines arithmetic operators for calculating with numbers:

Operator	Operation	Example	Description
+	Addition	$a = b + c$	Adds b and c
-	Subtraction	$a = b - c$	Subtracts c from b
*	Multiplication	$a = b * c$	Multiplies b and c
/	Division	$a = b / c$	Divides b by c
%	Modulus	$a = b \% c$	Remainder when dividing b by c

- Operations + and - have lower priority than *, /, and %.
- Parentheses give priority, e.g.: $a = (b + c) * d$ will calculate $b + c$ first
- Integer division does *not* round to nearest integer, but discards the decimal places.

Numeric examples:

- $20 / 3 \rightarrow 6$ (Mathematically 6.666..., but discards the decimal places .666...)
- $20 \% 6 \rightarrow 2$ ($20 = 3 * 6 + 2 \Rightarrow 6$ fits 3 times into 20 with remainder 2)



Calculate and print following data:

- Number of seconds in a day
- Number of seconds in a week
- Number of full weeks in a calendar year (365 days) and number of remaining days

The console output shall look as follows, with <...> being placeholders for calculated values:

A day has <...> seconds, while a week has <...> seconds.

A year has <...> full weeks with <...> day(s) remaining.

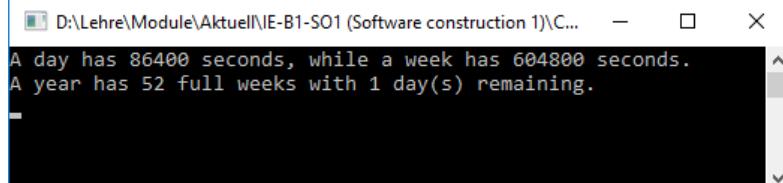
Sample solution:

```
#include <stdio.h>

int main(void)
{
    int secPerDay, secPerWeek;
    int weeksPerYear, remainingDays;

    secPerDay = 24 * 60 * 60;
    secPerWeek = 7 * secPerDay;
    weeksPerYear = 365 / 7;
    remainingDays = 365 % 7;

    printf("A day has %d seconds, while a week has %d seconds.\n",
           secPerDay, secPerWeek);
    printf("A year has %d full weeks with %d day(s) remaining.\n",
           weeksPerYear, remainingDays);
    getchar();
    return 0;
}
```



```
D:\Lehre\Module\Aktuell\IE-B1-SO1 (Software construction 1)\C...
A day has 86400 seconds, while a week has 604800 seconds.
A year has 52 full weeks with 1 day(s) remaining.
```

Exercise: Bank account



- On January 1, there are 1,000.- € on a bank account.
- The rate of return is 2 % a year.
- Calculate and print the account balance after 1, 2, and 3 years.

Exercise: Bank account

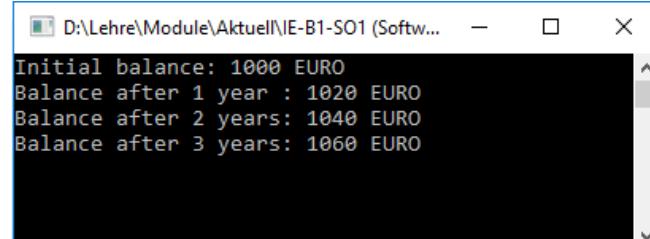
- Did you notice? – So far we cannot calculate an exact solution!
- Reason: We cannot handle decimal places, yet. (But will come to that in a minute ...)

Sample solution (well, sort of ...):

```
#include <stdio.h>

int main(void)
{
    int balance = 1000;
    int rate = 2;

    printf("Initial balance: %d EURO\n", balance);
    balance = (balance * (100 + rate)) / 100;
    printf("Balance after 1 year : %d EURO\n", balance);
    balance = (balance * (100 + rate)) / 100;
    printf("Balance after 2 years: %d EURO\n", balance);
    balance = (balance * (100 + rate)) / 100;
    printf("Balance after 3 years: %d EURO\n", balance);
    getchar();
    return 0;
}
```



```
D:\Lehre\Module\Aktuell\IE-B1-SO1 (Softw... — X
Initial balance: 1000 EURO
Balance after 1 year : 1020 EURO
Balance after 2 years: 1040 EURO
Balance after 3 years: 1060 EURO
```

Introducing floating point numbers

- Numbers with a decimal places

- Data type *float* stores floating point numbers
- These are numbers with decimal places (i.e., real numbers like 7.14).

Printing to console using *printf()*:

- Requires the placeholder %f (just like %d for integer numbers)
- Placeholder %.nf (with n being a number, e.g., %.2f) prints exactly n decimal places

Example:

- What is printed to console?

```
float pi = 3.141592;  
printf("Math constant pi: %f\n", pi);  
printf("Print 3 decimals: %.3f\n", pi);
```

- Console output:

```
Math constant pi: 3.141592  
Print 3 decimals: 3.142
```

Note: Output is rounded



Example:

- What is printed to console?

```
float price = 0.55;  
int units = 3;  
printf("Price per unit: %f EUR\n", price);  
printf("Price per unit: %.2f EUR\n", price);  
printf("Price for %d units: %.2f EUR\n", units, units * price);
```

Multiply *int* and *float*

Result is type *float*

- Console output:

```
Price per unit: 0.550000 EUR  
Price per unit: 0.55 EUR  
Price for 3 units: 1.65 EUR
```

Note:

- You can multiply an integer with a floating point number.
- The result is of type *float*.

Exercise: Bank account



Okay, now let's create a correct solution for our bank account exercise:

- On January 1, there are 1,000.- € on a bank account.
- The rate of return is 2 % a year.
- Calculate and print the account balance after 1, 2, and 3 years.

Exercise: Bank account

Sample solution:

```
#include <stdio.h>

int main(void)
{
    float balance = 1000;
    float rate = 2;

    printf("Initial balance: %.2f EURO\n", balance);
    balance = balance * (100 + rate) / 100;
    printf("Balance after 1 year : %.2f EURO\n", balance);
    balance = balance * (100 + rate) / 100;
    printf("Balance after 2 years: %.2f EURO\n", balance);
    balance = balance * (100 + rate) / 100;
    printf("Balance after 3 years: %.2f EURO\n", balance);
    getchar();
    return 0;
}
```

```
D:\Lehre\Module\Aktuell\IE-B1-SO1 (Software c...
Initial balance: 1000.00 EURO
Balance after 1 year : 1020.00 EURO
Balance after 2 years: 1040.40 EURO
Balance after 3 years: 1061.21 EURO
```

Print 2 decimal places

⇒ If your bank calculates using integers, they owe you 1.21 € after 3 years.



- You have done some running and covered 4.87 km in 29:30 minutes.
- Write a program to calculate and print your average pace in km/h to console.

Sample solution:

```
#include <stdio.h>

int main(void)
{
    float distanceKm = 4.87;
    float timeMinutes = 29.5;
    float timeHours = timeMinutes / 60.0;

    printf("Average pace: %.1f km/h\n", distanceKm / timeHours);
    getchar();
    return 0;
}
```

Note: 30 s = 0.5 min

- By the way, the average pace is 9.9 km/h.



- Calculate the circumference and area of a circle with radius 1.5 units.
- Approximate $\pi \approx 3.141592$.
- For output on console, round all numeric numbers to two decimal places.

Sample solution:

```
#include <stdio.h>

int main(void)
{
    float radius = 1.5, pi = 3.141592;

    printf("Radius      : %.2f units\n", radius);
    printf("Circumference: %.2f units\n", 2.0 * pi * radius);
    printf("Area        : %.2f units^2\n", pi * radius * radius);
    getchar();
    return 0;
}
```



And how would you approach the following?

- Calculate the circumference and area of circles with given radius.
- Radius: 1.5, 2.25, 4.61, and 7.1 units ... and maybe more values?

Some thoughts:

- Do *not* repeat the code for each requested radius!
- Write a program that allows users to *enter* a radius, instead.
⇒ We need user input from the keyboard.

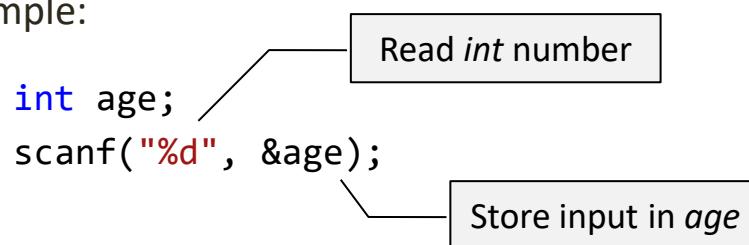
Keyboard input

- Read numbers from the keyboard

The scanf() function

- Reads (“scans”) formatted data (e.g., integer number) from the keyboard
- Parameters:
 - Data type to read (e.g., *int*) specified by a string (i.e., text in double quotes)
 - Variable to store keyboard input in

Example:



Notes:

- Will explain ampersand (&) at variable name in a minute
- Same placeholders for data types *int* and *float* as in `printf()` (e.g., %d for *int*)
- Further text or \n within the string will *not* be printed, e.g.:
 - ✖ `scanf("Enter age: %d\n", &age);`

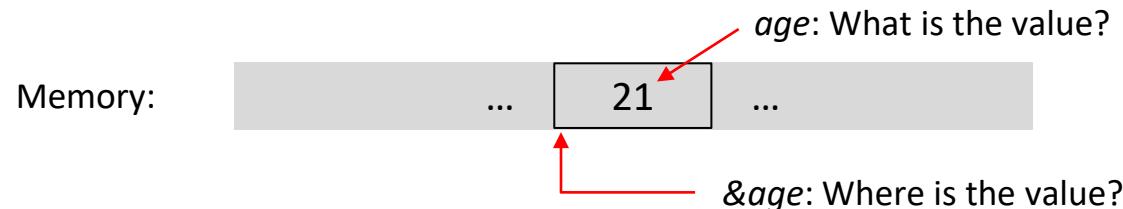
The difference between `age` and `&age`:

- Declaration of variable `age` reserves memory for an integer value
- Identifier `age` used to access and assign the value

Example:

```
int age = 21;  
printf("Age: %d\n", age);
```

- Argument `age` passes the *value* 21, not the location in memory, to `printf()`
- But `scanf()` needs to know, *where* to store the input in memory (i.e., the address).
- The prefix `&` gets the address (i.e., location in memory) of a variable.



Example: How old are you?

Let's look at a complete example:

```
#define _CRT_SECURE_NO_DEPRECATE
#include <stdio.h>

int main(void)
{
    int age;

    printf("Please enter your age: ");
    scanf("%d", &age);
    getchar();

    printf("Age entered: %d\n", age);
    getchar();
    return 0;
}
```

Only for Microsoft compilers

Declares *scanf()*

- The `#define` directive is required for Microsoft's MSVC++ compilers, only.
- Else MSVC++ prevents using `scanf()` for security reasons (potential buffer overflow)
- The directive is typically not printed in the sample codes in the following.

Example: How old are you?

Let's look at a complete example:

```
#define _CRT_SECURE_NO_DEPRECATE
#include <stdio.h>

int main(void)
{
    int age;

    printf("Please enter your age: ");
    scanf("%d", &age);
    getchar(); ——————| Oops ... why that?

    printf("Age entered: %d\n", age);
    getchar();
    return 0;
}
```



What do you think?

- What will happen, if we remove the first `getchar()`?
- Hint: Users end input of age by pressing *Enter*.



- Users shall enter his/her year of birth and the current year
- Program calculates the user's age at the end of the current year
- Example: A user born in 1999 will be 21 years old on December 31, 2020.

Sample solution:

```
#include <stdio.h>

int main(void)
{
    int yearOfBirth, thisYear;

    printf("In which year were you born?: ");
    scanf("%d", &yearOfBirth);
    getchar();

    printf("What year is today?: ");
    scanf("%d", &thisYear);
    getchar();

    printf("\nBy end of %d you will be %d years old.\n", thisYear, thisYear - yearOfBirth);
    getchar();
    return 0;
}
```



Let's come back to our original exercise:

- Users shall enter the radius of a circle.
- Calculate the circumference and area ($\pi \approx 3.141592$).

Sample solution:

```
#include <stdio.h>

int main(void)
{
    float radius;
    float pi = 3.141592;

    printf("Please enter a radius: ");
    scanf("%f", &radius);
    getchar(); → Read float number

    printf("\nRadius      : %.2f units\n", radius);
    printf("Circumference: %.2f units\n", 2.0 * pi * radius);
    printf("Area        : %.2f units^2\n", pi * radius * radius);
    getchar();
    return 0;
}
```



Finally let's revisit the running program:

- Users shall enter the covered distance in km and the time taken in minutes and seconds.
- Calculate the average pace in km/h.

Sample solution:

```
float distanceKm, timeHours;  
int timeMinutes, timeSeconds;  
  
printf("Distance covered: ");  
scanf("%f", &distanceKm);  
getchar();  
printf("Time taken (minutes): ");  
scanf("%d", &timeMinutes);  
getchar();  
printf("Time taken (seconds): ");  
scanf("%d", &timeSeconds);  
getchar();  
  
timeHours = (timeMinutes + timeSeconds / 60.0) / 60.0;  
printf("Average pace: %.1f km/h\n", distanceKm / timeHours);
```

Did you use type *int*?

Congratulations!

Congratulations – you have done a great run through some basics!



You should now be able to write simple, but meaningful programs:

- Declare integer and floating point numbers
- Apply arithmetic operations to them (“calculate”)
- Read numbers from the keyboard and print them to console

Now let's look at things in more detail ...

Comments & identifiers

- Documentation within source code
- Rules for names (e. g., variables)

Compilers ignore the following:

- Preprocessor directives (i.e., lines starting with a hash #)
- Whitespaces (i.e., blank, tab, and new line), if outside a text in double quotes
- Comments



Whitespaces:

- Which of the following must a compiler distinguish? Why?

```
printf("Hi");
printf ("Hi");
printf("H i");
printf( "Hi" );
printf(" Hi");
```



Comments:

- “*Writing comments?! Why should I waste time?*” – What do you think?

- Comments in source code are ignored by the compiler
- Objective is, e.g., to explain and structure the code to improve maintainability

Syntax:

- Texts within slashes and asterisks are comments `(/* ... */)`.
- Two slashes `(// ...)` comment text until the end of the line (ISO C99).

Examples:

```
/* Minimal program with comments.  
   Author: Marc Hensel  
   Lecture: IE-B1-SO1 (Software construction 1)  
*/  
  
int main(void) /* Main function (program entry point) */  
{  
    return 0; // Returns value 0 to operating system  
}
```

Things need a name, e.g.:

- Functions (e.g., *main*, *printf*, *scanf*, *getchar*)
- Variables

Rules for identifiers:

- Sequence of following symbols:
 - Letters of the English alphabet (A-Z, a-z)
 - National special characters (e.g., ä, ö, ü) (ISO C99) – But do *not* use these!
 - Underscore (_)
 - Digits (0-9)
- First letter must not be a digit and should not be underscore (not forbidden, but reserved)
- Must not use a reserved keyword (e.g., *int*, *void*, *return*)
- Identifiers are case sensitive (e.g., *main* ≠ *Main* ≠ *MAIN*).
- Length of local identifiers:
 - Arbitrary length allowed (ISO C99)
 - But distinguished by first 31 (ISO C90) or 63 (ISO C99) characters, only

Reserved keywords (for your reference):

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
void	unsigned	volatile	while



Time for a quiz!

- Which of the following identifiers are permitted?
- Which are not? (Why not?)

✓ `int length;`

✗ `int 3dVolume;`

✓ `int Length;`

✓ `int _3dVolume;`

✓ `int length_cm;`

✗ `int float;`

✓ `int maxLength;`

✗ `int email@;`

✗ `int max-length;`

✓ `int return_int;`

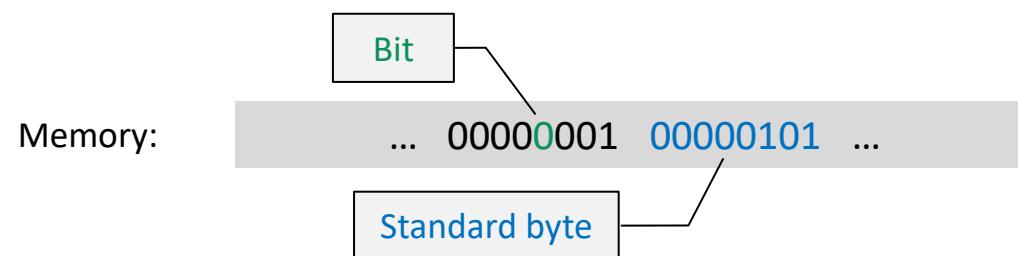
Integer data types

- Representation of integer numbers in memory
- Different types for integer values

- Computers store data as a collection of *bits* (*binary digits*, each with value 0 or 1)
- Can represent a number (e.g., 7 or 2.934) or a character (e.g., 'H')

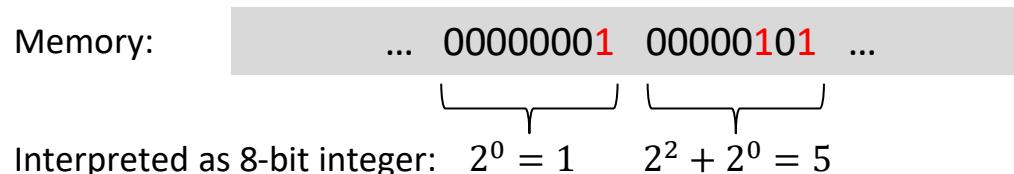
What is a *byte*?

- A collection of 8 bits is commonly called *byte*.
- Note: In C a *byte* is the number of bits for a character, which can be > 8 bits.



Example:

- Integer numbers have no fractional part (i.e., “no dot”)
- The bits represent (from right to left) $2^0, 2^1, 2^2, 2^3, \dots$



Think about it:

- What is the largest integer number representable by 8 bits?
- How many different integer numbers can be represented by 8 bits?

Answers:

- $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$
- Can represent $2^8 = 256$ different numbers (0 ... 255)

Exercise: Binary integers



Let's practice:

- Write the numbers 1 to 16 in 8-bit binary format.

Solution:

Decimal	Binary
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000

Decimal	Binary
9	00001001
10	00001010
11	00001011
12	00001100
13	00001101
14	00001110
15	00001111
16	00010000

Exercise: Binary integers

- Write the equation $45 + 35 = 80$ in 8-bit binary format.
- Denote numbers as being binary by following notation: $(\dots)_2$

Note:

- $45 = 32 + 8 + 4 + 1$
- $35 = 32 + 2 + 1$
- $80 = 64 + 16$

Solution:

$$(00101101)_2 + (00100011)_2 = (01010000)_2$$


 $2^5 + 2^3 + 2^2 + 2^0 = 45$ $2^5 + 2^1 + 2^0 = 35$ $2^6 + 2^4 = 80$



Think about it:

- What is missing to represent potentially all integer numbers required in a program?
- Okay, here is a hint: How would you represent negative numbers in binary format?

Approach for negative numbers:

- Highest bit (*most significant bit, MSB*) represents the mathematical **sign** (+ or -)



Think again:

- What about the value 0?

There should be *only one* code sequence for value 0 (\Rightarrow not +0 and -0):

- Binary code 00000000 represents the decimal value 0
- Binary code 10000000 does *not* represent the decimal value -0

Data range (8-bit):

Binary	Decimal	How many?
00000000	0	1
00000001 ... 01111111	Positive numbers	$2^7 - 1 = 127$
10000000 ... 11111111	Negative numbers	$2^7 = 128$

- 8-bit signed integers are in the range -128 to 127
- Will discuss coding of negative numbers later, but be aware: $(10000001)_2 \neq -1$

Data range for n -bit integers:

- Unsigned: 0 to $2^n - 1$
- Signed: -2^{n-1} to $2^{n-1} - 1$

Standard data types:

Data type	Aliases	Signed	Bits	Specifier <i>printf()</i>	Specifier <i>scanf()</i>
int	signed (int)	yes	≥ 16 bits, typically 1 machine word (e.g., 32 bits for 32-bit processor)	%d	%d
short	short int, signed short (int)	yes	≤ int and ≥ 16 bits	%hd	%hd
long	long int, signed long (int)	yes	≥ int and ≥ 32 bits	%ld	%ld
long long	long long int, signed long long (int)	yes	≥ long and ≥ 64 bits	%lld	%Ld
unsigned	unsigned int	no	Same as int	%u	%u
unsigned short	unsigned short int	no	Same as short	%hu	%hu
unsigned long	unsigned long int	no	Same as long	%lu	%lu
unsigned long long	unsigned long long int	no	Same as long long	%llu	%Lu

- Types (*unsigned*) *long long* introduced with ISO C99
- Format specifiers for *printf()* and *scanf()* functions may differ
- Keyword *signed* has no effect, but makes signed interpretation explicit

- Size of standard integer data types can be different for different machines!
- There exist additional data types that guarantee a specific bit depth.
- Typically:
 - Size of *short* < size of *long*
 - Size of *int* same as size of either *short* or *long* (depending on processor's word size)

Data ranges:

- Select the data type by required range of values
- Calculations for *int* might be faster than for other data types (fits machine word)

Bits	Unsigned	Signed
n	$0 \dots 2^n - 1$	$-2^{n-1} \dots 2^{n-1} - 1$
16	$0 \dots 65,535$	$-32,768 \dots 32,767$
32	$0 \dots 4,294,967,295$	$-2,147,483,648 \dots 2,147,483,647$
64	$0 \dots 18.4 \cdot 10^{18}$	$-9.2 \cdot 10^{18} \dots 9.2 \cdot 10^{18}$



Think about it:

- You want to buy an additional hard disk for your computer (32-bit Windows 10).
- Each address references a memory sector of 512 bytes (Master Boot Record, MBR).
- What is the maximum addressable hard disk / partition size?

Units:

- Strictly speaking:
 - $1 \text{ kB} = 10^3 \text{ bytes} = 1000 \text{ bytes}$ (*kilobyte*)
 - $1 \text{ TB} = 10^{12} \text{ bytes}$ (*terabyte*)
- Still common for hard disks:
 - $1 \text{ kB} = 2^{10} \text{ bytes} = 1024 \text{ bytes}$
 - $1 \text{ TB} = 2^{40} \text{ bytes}$

Solution:

- 32 bits can address 2^{32} different sectors with $512 = 2^9$ bytes, each.
- Maximum addressable size: $2^{32} \cdot 2^9 = 2^{32+9} = 2^{40+1} = 2 \cdot 2^{40} \text{ bytes} = 2 \text{ TB}$



For data types *unsigned short*, *unsigned*, and *unsigned long*:

- Print the maximum value and number of bits used to console.

Notes:

- Maximum values: *USHRT_MAX*, *UINT_MAX*, and *ULONG_MAX* in *limits.h*
- Use function *log2()* in *math.h* to calculate the logarithm

Sample solution:

```
#include <stdio.h>
#include <limits.h>
#include <math.h>

int main(void)
{
    printf("unsigned short: 0 to %hu (%.0f bits)\n", USHRT_MAX, log2(USHRT_MAX));
    printf("unsigned int   : 0 to %u (%.0f bits)\n", UINT_MAX, log2(UINT_MAX));
    printf("unsigned long  : 0 to %lu (%.0f bits)\n", ULONG_MAX, log2(ULONG_MAX));
    getchar();
    return 0;
}
```



For the data type *unsigned short*:

- Print the result of $a = 0 - 1$ to console.
- Print the result of $b = \text{USHRT_MAX} + 1$ to console.

Sample solution:

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("Data range: 0 to %hu\n", USHRT_MAX);
    printf("a = 0 - 1 = %hu\n", 0 - 1);
    printf("b = %hu + 1 = %hu\n", USHRT_MAX, USHRT_MAX + 1);
    getchar();
    return 0;
}
```

Console output:

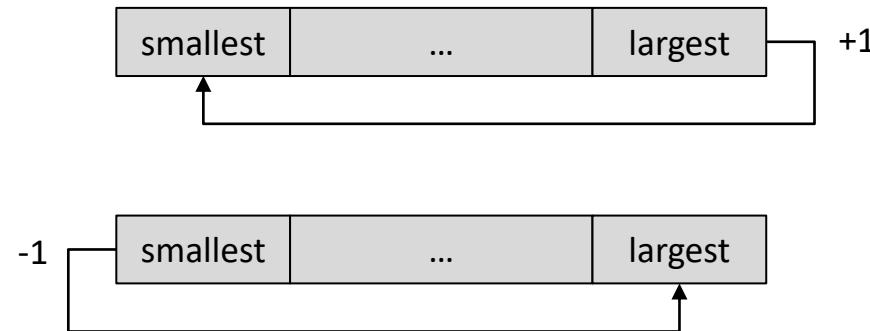
```
Data range: 0 to 65535
a = 0 - 1 = 65535
b = 65535 + 1 = 0
```

Somewhat unexpected?

- Integer *underflow*: Number is too small for the data type
- Integer *overflow*: Numbers is too large for the data type

Wrap around:

- Integer numbers *wrap around* at underflow and overflow.



⇒ Make sure that the result fits the data type!

Character data type

- Representing letters (and other characters)



Imagine you were a computer (... that's difficult, I know):

- How would you store a character (e.g., 'H') in memory?

Approach:

- Cannot store characters directly, but only numbers
- Map each character to an unsigned integer number (*code*)

Example: ASCII

- Code table for U.S.
- 7-bit code (stored as 8-bit unsigned integer)
- Includes Latin letters, digits, and various further symbols (e.g., line break)
- Does not include special national characters (e.g., German, Danish, Greek, Asian)



A quick recap:

- How many different characters can be represented by ASCII?

Data type for characters is *char*:

Data type	Signed	Bits	Format specifier
char	Not specified	Large enough to store character set of machine, typically 8 bits	%c

- Variable declaration as for other data types
- Can assign an integer (e.g., ASCII code) or character enclosed in single quotes

Example:

```
#include <stdio.h>

int main(void)
{
    char capitalH = 'H', newLine = '\n';    // Characters in single quotes
    char smallM = 109;                      // ASCII code for letter 'm'

    printf("%cAW %ca%cburg%c", capitalH, capitalH, smallM, newLine);
    getchar();
    return 0;
}
```

Escape sequences

- *Escape sequences* represent selected non-printable “characters”.
- An escape sequence always begins with a backslash \.

Common escape sequences:

Escape sequence	Meaning
\a	Alert (“beep” sound)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\\	Backslash
\'	Single quote
\”	Double quote



- What are the numeric character codes of digits 0, 1, 2, ...?
- What are the numeric character codes of letter A, B, C, ... and a, b, c, ...?

Sample solution:

```
#include <stdio.h>

int main(void)
{
    printf("Digits\t\t: %d, %d, %d, ..., %d\n", '0', '1', '2', '9');
    printf("Large letters\t: %d, %d, ..., %d\n", 'A', 'B', 'C', 'Z');
    printf("Small letters\t: %d, %d, ..., %d\n", 'a', 'b', 'c', 'z');
    getchar();
    return 0;
}
```

Numeric value

Horizontal tab

Console output:

```
Digits      : 48, 49, 50, ..., 57
Large letters : 65, 66, 67, ..., 90
Small letters : 97, 98, 99, ..., 122
```

Code for '1' ≠ 1

'a' ≠ 'A'



Write a program doing the following:

- Users shall enter a character on the keyboard.
- Print the numeric code of the character entered to console.

Sample solution:

```
#include <stdio.h>

int main(void)
{
    char input;

    /* Get user input */
    printf("Please enter a character: ");
    scanf("%c", &input);
    getchar(); → Character type

    /* Print input and numeric code to console */
    printf("\nEnterd\t\t: '%c'\n", input);
    printf("Numeric code\t: %d\n", input);
    getchar();
    return 0;
}
```

Floating point data types

- Representation of floating point numbers in memory
- Different types for floating point values

Principal approach: $a = \pm 1.m \cdot 2^n$

- Sign
- Fraction $0 \leq m < 1$
- Exponent n

Example for 32-bit floating point:

\pm	Exponent (8 bits)	Fraction (23 bits)
-------	-------------------	--------------------

Think about it:

- What number a is represented by the following data?

-	1	0.570796
---	---	----------

Solution:

- $a = -1.570796 \cdot 2^1 = -3.141592 \approx -\pi$

Good news – the list of floating point types is much shorter than for integers:

Data type	Significant digits	Exponent range	Bits	Specifier printf()	Specifier scanf()
float	≥ 6	At least 10^{-37} to 10^{37}	Not specified, typically 32	%f, %e	%f
double	≥ 10	Same as float	Not specified, typically 64	%f, %e	%lf
long double	\geq double	Not specified	Not specified	%Lf, %Le	%Lf

- All data ranges are signed.
- Specifiers %e and %Le print exponential notation (e.g., -1.6e-19 being $-1.6 \cdot 10^{-19}$).
- Again, size of data types can be different for different machines!
- Type *long double* is typically the same as *double* on MSVC++.

Typical values (IEEE 754 format):

Data type	Bits	Significant digits	Range of absolute numbers (excluding 0)
float	32	≈ 7	$3.4 \cdot 10^{-38} \dots 3.4 \cdot 10^{38}$
double	64	≈ 15	$1.7 \cdot 10^{-308} \dots 1.7 \cdot 10^{308}$

Constants & constants

- Values written directly in source code
- Variables whose values must not change



Oops ... why writing it twice: constants *and* constants?

- Constant values written directly in the code (e.g., number 7)
- “Variables” whose values are constant (i.e. cannot be changed)



Think about it:

- Give practical examples for “variables” with constant value.

Examples:

- Mathematical constants like π and Euler number e
- Physical constants like speed of light, electrical charge q of electron, ...
- Application-related constants like maximum load of an aircraft, ...

To make a variable a constant:

- Add keyword *const* to a data type during declaration
 - Keyword can be placed before or after the data type (recommendation: before).
- ⇒ The value *cannot* be assigned (modified) after the declaration.

Example:

```
#include <stdio.h>
```

```
int main(void)
{
    const float PI = 3.141592;
    // PI = 3.1416;
    printf("Math pi = %f\n", PI);
    getchar();
    return 0;
}
```

Typically in capital letters

Assignment at declaration

Assignment not allowed

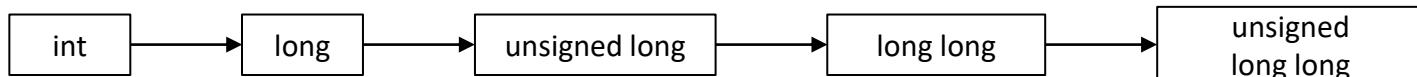
- *Constants* (also called *literals*): Values written directly in source code
- Example:

```
int minAge = 18;
```



C automatically selects the data type:

1. By default stored as *int*
2. Value too large for *int* ⇒ Treated as *long*
3. Value also too large for *long* ⇒ Treated as *unsigned long*
4. ...



Programmer can explicitly set the data type:

- Syntax: Add appropriate suffix to a value

Data type	Suffix
long	L or l
long long	LL or ll
unsigned	U or u
unsigned long	UL or ul
unsigned long long	ULL or ull

- Examples:

```
long a = 23L;  
unsigned long long b = 1024ULL;
```

- Use capital letters, because hard to distinguish small L from digit 1:

```
long a = 23l;
```



Value 23 or 231?

Floating point numbers:

- Number containing a “dot”
- Can be expressed as $a = m \cdot 10^n$ by suffix e or E and exponent

Data types:

- By default treated as *double*
- Suffixes:

Data type	Suffix
float	F or f
long double	L or l

Examples:

```
double a = 2.;  
double b = 0.25;  
float c = 7.1f;  
double milli = 1.0e-3;  
double kilo = 1.0e3;
```



What is printed to the console? Explain.

```
#include <stdio.h>

int main(void)
{
    printf("Math pi: 3.141592654\n");
    printf("Double : %.9f\n", 3.141592654);
    printf("Float  : %.9f\n", 3.141592654f);
    getchar();
    return 0;
}
```

Stored as *float*

Print 9 digits after dot

Sample output:

```
Math pi: 3.141592654
Double : 3.141592654
Float  : 3.141592741
```

Exceeds precision of *float*



What is printed to the console? Explain.

```
#include <stdio.h>

int main(void)
{
    const float CHARGE_E = -1.602177e-19f;

    printf("Electron: q = %f C\n", CHARGE_E);
    printf("Electron: q = %e C\n", CHARGE_E);
    printf("Electron: q = %.1e C\n", CHARGE_E);
    getchar();
    return 0;
}
```

Sample output:

```
Electron: q = -0.000000 C
Electron: q = -1.602177e-19 C
Electron: q = -1.6e-19 C
```

Not enough decimal
places displayed

Operators

- Changing numeric values
- Comparing numeric values
- Logical operators

Arithmetic operators

- We have already introduced arithmetic operators and sign operators.
- Overview including precedence (lower number \Rightarrow higher “priority”):

Operator	Operation	Example	Description	Precedence
+	Sign	$a = +7$	Positive value	2
-	Sign	$a = -7$	Negative value	2
+	Addition	$a = b + c$	Adds b and c	4
-	Subtraction	$a = b - c$	Subtracts c from b	4
*	Multiplication	$a = b * c$	Multiplies b and c	3
/	Division	$a = b / c$	Divides b by c	3
%	Modulus	$a = b \% c$	Remainder when dividing b by c	3

Operands:

- Allowed to have operands of different types (e.g., $float * int$)
- Operation and result is floating point, if at least one of the operands is floating point



What is printed to the console?

```
#include <stdio.h>

int main(void)
{
    int units = 4, maxUnits = 10;
    float price = 0.55;

    printf("float * int: %.2f\n", units * price);
    printf("int / int : %d\n", units / maxUnits);
    printf("int / int : %.1f\n", units / maxUnits);
    printf("float / int: %.1f\n", 4.0 / maxUnits);
    getchar();
    return 0;
}
```

Force output of decimal places

Console output:

```
float * int: 2.20
int / int : 0
int / int : 0.0
float / int: 0.4
```

Integer division discards decimal places

Relational operators

- Compare values of two operands:

Operator	Operation	Precedence
<	Smaller than	6
<=	Smaller than or equal to	6
>	Greater than	6
>=	Greater than or equal to	6
==	Equal to	7
!=	Not equal to	7

Result:

- Result of operators is a logical *true* or *false*
- In C logical values are represented by an integer number:

Logical value	Integer value	Relational operator
false	0	0
true	$\neq 0$	1



What is printed to the console?

```
#include <stdio.h>

int main(void)
{
    int a = 7, b = 4;
    int parentheses = (a > b) == (a <= b);

    printf("true : %d\n", 1 == 1);
    printf("false: %d\n", 1 != 1);
    printf("Parentheses: %d\n", parentheses);
    printf("Precedence : %d\n", a > b == a <= b);
    getchar();
    return 0;
}
```

Console output:

```
true : 1
false: 0
Parentheses: 0
Precedence : 0
```

Logical operators

- We have seen that C represents logical values as integer 0 (*false*) or $\neq 0$ (*true*).
- Logical operators:

Operator	Operation	Precedence
!	Negation (NOT)	2
&&	Conjunction (AND)	11
	Disjunction (OR)	12

Result of operations:

a	b	!a	a b	a && b
0	0	1	0	0
0	1	1	1	0
1	0	0	1	0
1	1	0	1	1

Combined assignments

- Can combine several operations with an assignment, for example:

Operator	Example	Equivalent to	Precedence
<code>+=</code>	<code>a += b</code>	<code>a = a + b</code>	14
<code>-=</code>	<code>a -= b</code>	<code>a = a - b</code>	14
<code>*=</code>	<code>a *= b</code>	<code>a = a * b</code>	14
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>	14
<code>%=</code>	<code>a %= b</code>	<code>a = a % b</code>	14

- What is printed to the console?



```
int a = 1;
```

```
a += 2;
printf("a = %d\n", a);
printf("a = %d\n", a *= 3);
a %= a - 2;
printf("a = %d\n", a);
printf("a = %d\n", a /= 3 * a - 4);
```



```
D:\Lehre\Module\Aktuell>
a = 3
a = 9
a = 2
a = 1
```

- Often needed: Add 1 to a number or subtract 1 from a number

Operators:

- Increment operator `++` increases value of variable by 1
- Decrement operator `--` decreases value of variable by 1
- Can be left (*prefix*) or right (*postfix*) of an integer or floating point variable

Operation	Assignment	Value	Precedence
<code>++a</code>	<code>a = a + 1</code>	$a + 1$	2
<code>a++</code>	<code>a = a + 1</code>	a	2
<code>--a</code>	<code>a = a - 1</code>	$a - 1$	2
<code>a--</code>	<code>a = a - 1</code>	a	2

This is somewhat confusing? – Let's clarify by examples ...

Prefix operator:

```
int a = 4;  
printf("a = %d\n", a);  
printf("Prefix (value) : %d\n", ++a);  
printf("Prefix (assigned): %d\n", a);
```

- Console output:

```
a = 4  
Prefix (value) : 5  
Prefix (assigned): 5
```

- Sequence:

- First, the operator increases the value of the variable ($4 \rightarrow 5$)
- Afterwards, $++a$ is replaced by the value of the variable ($5 \rightarrow \text{printf}()$)

⇒ Read $++a$ from left to right.

Postfix operator:

```
int a = 4;  
printf("a = %d\n", a);  
printf("Postfix (value) : %d\n", a++);  
printf("Postfix (assigned): %d\n", a);
```

- Console output:

```
a = 4  
Postfix (value) : 4  
Postfix (assigned): 5
```

- Sequence:
 - First*, a++ is replaced by the value of the variable (4 → printf())
 - Afterwards*, the operator increases the value of the variable (4 → 5)

⇒ Again, read a++ from left to right.



What is printed to the console?

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    float b = 1.25;

    /* Integer */
    printf("a    : %d\n", a);
    printf("++a : %d\n", ++a);
    printf("a++ : %d\n", a++);
    printf("--a : %d\n", --a);
    printf("a-- : %d\n", a--);

    /* Floating point */
    printf("\nb    : %.2f\n", b);
    printf("++b : %.2f\n", ++b);

    getchar();
    return 0;
}
```

Type conversions

- Changing a value's data type

Example: Average value



Let's calculate the average of two numbers:

- What is printed to the console?

```
int x1 = 1, x2 = 2;  
int count = 2;  
double average;  
  
average = (x1 + x2) / count;  
printf("Values : %d, %d\n", x1, x2);  
printf("Average: %.1f\n", average);
```

- Console output:

```
Values : 1, 2  
Average: 1.0
```



Think about it:

- Oops ... the average value is not correct! Why?
- Fix the issue (using only what we have discussed so far).

- Avoid integer division by operator of type *double*:

```
int x1 = 1, x2 = 2;  
double average;  
  
average = (x1 + x2) / 2.0;  
printf("Values : %d, %d\n", x1, x2);  
printf("Average: %.1f\n", average);
```

- Console output:

```
Values : 1, 2  
Average: 1.5
```



Think about it:

- The average value is correct, *but* ...
- What if we cannot replace an integer operator by a floating point constant?

Changing the data type of a value:

- Place a data type in parentheses before an expression
- ⇒ Changes (“casts”) the expression’s *value* to the specified data type

Example:

```
int x1 = 1, x2 = 2;
int count = 2;
double average = (x1 + x2) / (double) count;
printf("Values : %d, %d\n", x1, x2);
printf("Average: %.1f\n", average);
```



Cast int to double

- Console output:

```
Values : 1, 2
Average: 1.5
```

Note:

- The operator has precedence 1 (i.e., “highest priority”).



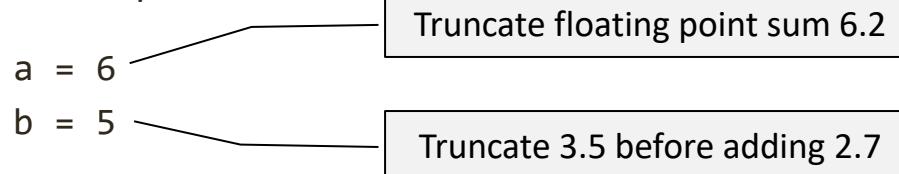
What is printed to the console?

```
#include <stdio.h>

int main(void)
{
    int a = 3.5 + 2.7;
    int b = (int) 3.5 + 2.7;

    printf("a = %d\n", a);
    printf("b = %d\n", b);
    getchar();
    return 0;
}
```

Console output:



Exercise: Part production



Write following program related to manufacturing of parts:

- Users shall enter the number of “good” and “bad” (e.g., broken) parts produced.
- Print the number of bad parts, overall number of parts, and the ratio to the console.

Sample output:

```
Enter number of good parts produced: 793
```

```
Enter number of bad parts produced : 14
```

```
Bad parts: 14 out of 807 (ratio 0.02)
```

Sample solution:

```
#include <stdio.h>

int main(void)
{
    int goodParts, badParts;
    double badRatio;

    /* Get user input */
    printf("Enter number of good parts produced: ");
    scanf("%d", &goodParts);
    getchar();

    printf("Enter number of bad parts produced : ");
    scanf("%d", &badParts);
    getchar();

    /* Calculate and print production performance */
    badRatio = badParts / (double)(goodParts + badParts);
    printf("\nBad parts: %d out of %d (ratio %.2f)\n", badParts, goodParts + badParts, badRatio);

    getchar();
    return 0;
}
```

Coding style

- Conventions how to write code



Let's talk about style:

- What is your association with the term *coding style*?

Note carefully:

- Consists of rules how to write and format source code (e.g., names, indents)
- Is not “correct” or “incorrect”, but *uniform* for all involved developers
 - ⇒ Improved maintainability, programming less error-prone

Note even more carefully:

- Being non-conform to the coding style can influence your grade in the lab exam!

Naming conventions:

- All identifiers (“names”) must be in English.
- Choose meaningful identifiers (e.g., not *temp* or such).
- Variables and functions:
 - Identifiers for variables and functions begin with a small letter.
 - Format identifiers consisting of several words as follows (“CamelCase”):

```
int numberStudents, highScore;  
float gpsLatitude, gpsLongitude;
```

- Constants:
 - Type identifiers for constants in capital letters.
 - Separate words by underscores

```
const float MATH_PI;
```

Indents and spaces:

- Indent blocks and such by a tabulator distance per level.
- Separate logical blocks by an empty line.
- Add a blank (“space”) between, e.g., operands and operators.

```
#include <stdio.h>
```

```
int main(void)
{
    int a = 1;
    printf("Your lucky number is %d.\n", 10 * a - 3);
    return 0;
}
```

The diagram illustrates the application of coding style rules to a C program. A red arrow points to the opening brace of the `main` function. Three callout boxes point to specific parts of the code:

- Empty line**: Points to the blank line after the opening brace of the function.
- Blanks**: Points to the space before the closing parenthesis of the `printf` call.
- Indent**: Points to the four spaces before the first line of code inside the `main` function block.

Software Construction 1 (IE1-SO1)

3. Flow control



Lecture overview

Fundamentals



1. Data types



● 2. Flow control

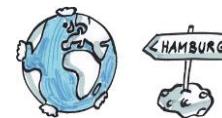


3. Functions



4. Arrays (and strings)

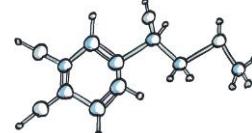
Advanced topics



5. Pointers



6. Memory management



7. Structures



8. Lists and sorting

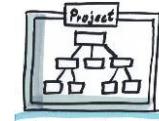
The next steps ...



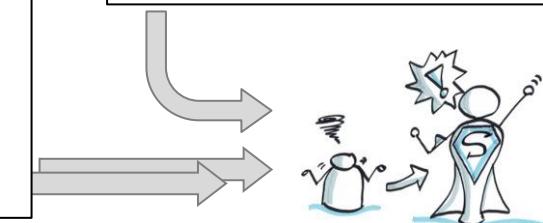
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



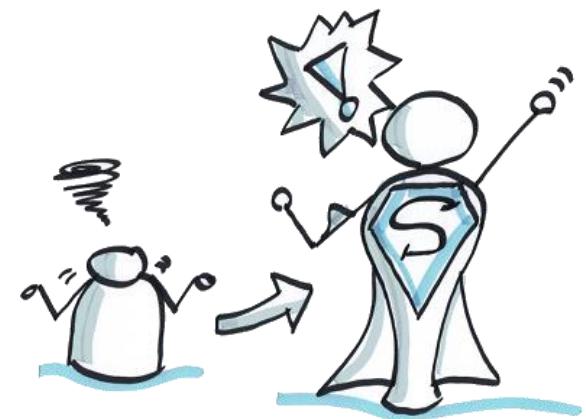
You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *react* to data by executing specific source code depending on the values of variables (e.g., user input or result of calculations).
- You *repeat execution* of parts of a program (e.g., in order to avoid repeating the corresponding source code).



1. Selections & jump statements
 - Conditional selections (if/else)
 - Conditional jumps (switch)
 - Unconditional jump (goto)
2. Loops
 - while-loop
 - do/while-loop
 - for-loop
 - Jump statements (break & continue)

Conditional selections (if/else)

- Execute code block, only if a specific condition is fulfilled

Let's recap some of the examples we have seen so far.

How old are you?



- Users shall enter his/her year of birth and the current year.
- Calculate the user's age at the end of the current year.
- What happens, if you enter 2100 as year of birth and are honest about the current year?

Circle:



- Users shall enter the radius of a circle.
- Calculate the circumference and area.
- What happens, if you enter a radius of -1.5?
- And what happens, if you calculate the square root of that radius?

⇒ Need to react to values of variables (i.e., execute different code depending on value)

Syntax to execute a statement *only if* a specific condition is fulfilled:

```
if (expression)
    statement
```

Expression (“condition”):

- An expression evaluating to a numeric value
- Typically a relational operator (e.g., $a > 5$)
- The value is interpreted as Boolean value (i.e., $0 \Rightarrow false$, any value $\neq 0 \Rightarrow true$).

Statement:

- Any valid C statement (e.g., assignment, function call)
- Can consist of more than 1 statements combined in braces {...} (see next slide)

Statements:

- Consist of an expression and semicolon
- When reaching a semicolon, compilers execute (evaluate) the corresponding expression.

Blocks:

- Braces {...} combine statements to a *block* (*compound statement*).
- When the syntax expects a statement, this can be replaced by a block, e.g. :

```
if (expression)  
    statement
```



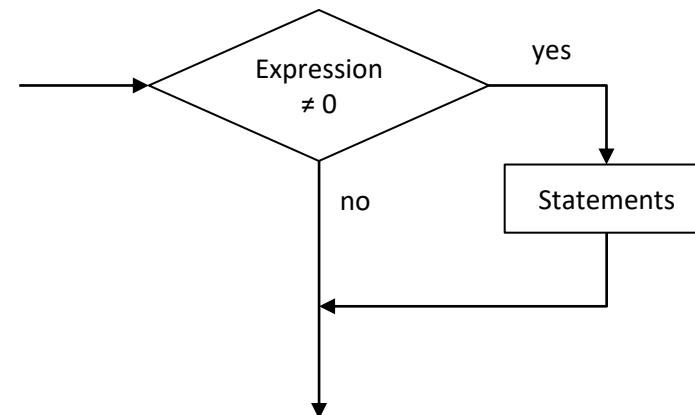
```
if (expression)  
{  
    statement 1  
    statement 2  
    ...  
}
```

No semicolon at end of block

How it works:

- Executes the statement (or block), if the expression is not 0 ($\Rightarrow true$)
- Skips the statement (or block), if the expression is 0 ($\Rightarrow false$)

```
if (expression)
{
    statement 1
    statement 2
    ...
}
...
Expression == 0
```





Write following program:

- Users shall enter two numbers.
- Print the larger of the numbers entered to the console.

Sample solution:

```
float a, b, max;

/* Get user input: 2 numbers */
printf("Enter two comma-separated numbers: ");
scanf("%f,%f", &a, &b);
getchar();
```

Note: Scan two values

```
/* Determine and print maximum value */
max = a;
if (b > a)
{
    max = b;
}
printf("Maximum value entered: %f", max);
```

Braces {...} not required



Let's play "fill the gap":

- Analyze the source code below.
- Add an expression to the *if*-statement, such that the console output is logically correct.

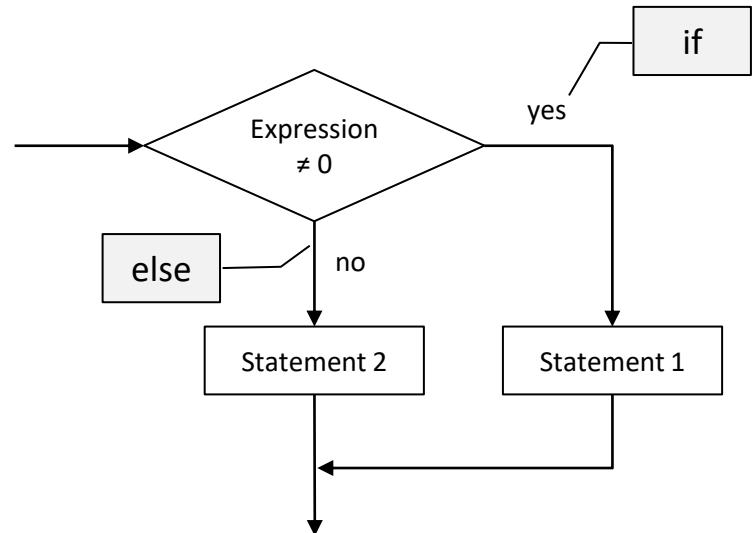
Source code:

```
float a;  
int isNegative;  
  
printf("Please enter a number: ");  
scanf("%f", &a);  
getchar();  
  
isNegative = a < 0;  
if [redacted]  
    printf("You have entered a non-negative number.");
```

You can add statements to execute, if the condition is *not* fulfilled:

```
if (expression)
    statement 1
else
    statement 2
```

- Executes *statement 1*, if expression is not 0
 - Executes *statement 2*, if expression is 0
- ⇒ “Chooses” between two alternatives





Let's adapt a previous exercise:

- Users shall enter his/her year of birth and the current year
- Display the user's age at the end of the year – or a hint that the data does not make sense.

Sample solution:

```
int yearOfBirth, thisYear, ageEndOfYear;

/* Get user input */
printf("In which year were you born?: ");
scanf("%d", &yearOfBirth);
getchar();

printf("What year is today?: ");
scanf("%d", &thisYear);
getchar();

/* Print age or ironic remark */
ageEndOfYear = thisYear - yearOfBirth;
if (ageEndOfYear >= 0)
    printf("\nBy end of %d you will be %d years old.\n", thisYear, ageEndOfYear);
else
    printf("\nHello Marty, are you ready to travel \"Back to the future\"?");
```



Let's adapt another previous exercise:

- Users shall enter the radius of a circle.
- Calculate the circumference and area ($\pi \approx 3.141592$).

Sample solution:

```
float radius;
const float PI = 3.141592f;           Constant (identifier with capital letters)

/* Get user input: radius */
printf("Please enter a radius: ");
scanf("%f", &radius);
getchar();

/* Print area and circumference to console */
if (radius >= 0.0)
{
    printf("\nRadius      : %.2f units\n", radius);
    printf("Circumference: %.2f units\n", 2.0 * PI * radius);
    printf("Area        : %.2f units^2\n", PI * radius * radius);
}
else
    printf("\nThere is no circle with negative radius.");
```

Constant (identifier with capital letters)

Literal of type *float*



- Users shall enter his/her result of an exam graded in 0 ... 15.
- Print, whether the user has failed (0 ... 4), passed (5 ... 15), or entered an invalid grade.

Sample solution:

```
int grade;

/* Get user input: grade in 0 ... 15 */
printf("Please enter your grade: ");
scanf("%d", &grade);
getchar();

/* Print to the console */
if ((grade >= 0) && (grade <= 15))
{
    if (grade >= 5)
        printf("Congratulations, you have passed the exam!\n");
    else
        printf("Sorry, you have failed in the exam.\n");
}
else
    printf("Grade %d?! Please be honest, will you?\n", grade);
```

*if/else-statement nested in
another if/else-statement*

... or like this:

```
int grade;

/* Get user input: grade in 0 ... 15 */
printf("Please enter your grade: ");
scanf("%d", &grade);
getchar();

/* Print to the console */
if ((grade >= 0) && (grade <= 4))
{
    printf("Sorry, you have failed in the exam.\n");
}
else
{
    if ((grade >= 5) && (grade <= 15))
        printf("Congratulations, you have passed the exam!\n");
    else
        printf("Grade %d?! Please be honest, will you?\n", grade);
}
```

... which is equivalent to:

```
int grade;

/* Get user input: grade in 0 ... 15 */
printf("Please enter your grade: ");
scanf("%d", &grade);
getchar();

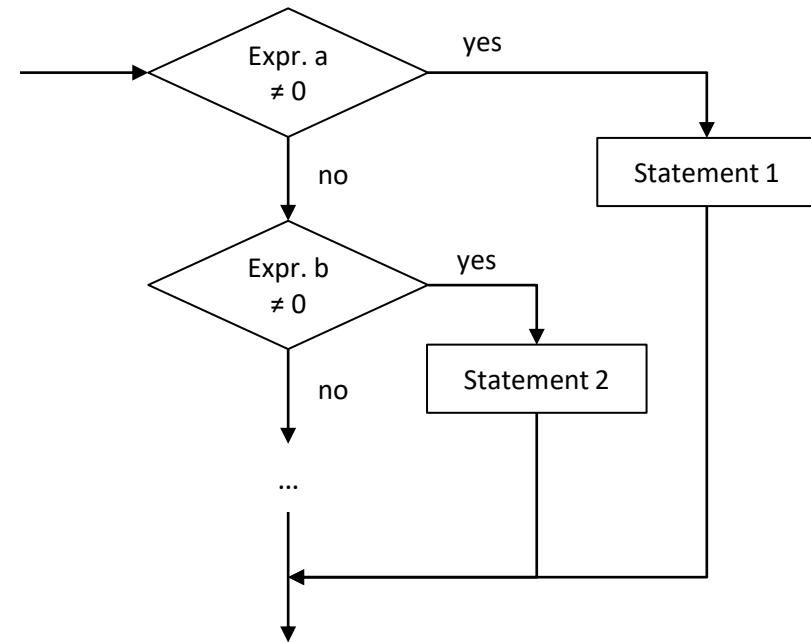
/* Print to the console */
if ((grade >= 0) && (grade <= 4))
    printf("Sorry, you have failed in the exam.\n");
else if ((grade >= 5) && (grade <= 15))
    printf("Congratulations, you have passed the exam!\n");
else
    printf("Grade %d?! Please be honest, will you?\n", grade);
```

- Removed braces that were not required (though useful for better readability)
 - Changed line breaks and indents
- ⇒ Source code has not changed from compiler's point of view, but is easier to read

Multiple if-statements

Compact code to branch program flow by more than one condition:

```
if (expression a)
    statement 1
else if (expression b)
    statement 2
else if (expression c)
    statement 3
...
else
    statement n
```





One final small exercise:

- What is printed to the console?

```
int a = 3;  
printf("a = %d\n", a);  
  
if (a = 1)  
    printf("a is equal to 1.\n");  
else  
    printf("a is not equal to 1.\n");
```

Console output:

```
a = 3  
a is equal to 1.
```

Oops ... a is 1?! Why?

Typical (and difficult to notice) error:

- Expression is an assignment ($a = 1$)
- But should be a comparison ($a == 1$)

Conditional operator ? :

Compact way to express if/else-statement, in case the statements are expressions:

```
expression 1 ? expression 2 : expression 3
```

- Replaced by *expression 2*, if *expression 1* is not 0 ($\Rightarrow true$)
- Replaced by *expression 3*, if *expression 1* is 0 ($\Rightarrow false$)

Example:

- Determine maximum of two values:

```
int a = 3, b = 7;  
int max;  
  
max = (a > b) ? a : b;
```

Conditional jumps (switch)

- Execute specific code blocks depending on the value of a variable

Another syntax to choose from multiple alternatives:

```
switch (integer expression)
{
    case constant 1:
        statements
        break;
    case constant 2:
        statements
        break;
    ...
    default:
        statements
}
```

```
int a = 5;
switch (a)
```

...

```
case 5:
```

...



Jump to label with
same value as *a*

Continue sequential
processing

Integer labels:

- Keyword *switch* is followed by integer (e.g., *int* or *char* variable) in parentheses
- Labels consist of *case* followed by mutually exclusive integer constants.
- Jump to label with same integer value (if exists) and continue from there on

Another syntax to choose from multiple alternatives:

```
switch (integer expression)
{
    case constant 1:
        statements
        break;
    case constant 2:
        statements
        break;
    ...
    default:
        statements
}
```

```
int a = 5;
switch (a)
case 1:
...
case 2:
...
default:
...
```

No matching label
⇒ Jump to *default*

Continue sequential
processing

Default label:

- Jump to *default* label, if no *case* label with same integer value exists
- Continue sequential processing from there on

Another syntax to choose from multiple alternatives:

```
switch (integer expression)
{
    case constant 1:
        statements
        break;
    case constant 2:
        statements
        break;
    ...
    default:
        statements
}
```

```
int a = 5;
switch (a)
    ...
case 5:
    ...
break;
```



break ⇒ Leave
switch-statement

Terminating:

- A *break*-statement terminates the *switch*-statement.
- Processing continues directly after the closing brace } of the *switch*-statement

Another syntax to choose from multiple alternatives:

```
switch (integer expression)
{
    case constant 1:
        statements
        break;
    case constant 2:
        statements
        break;
    ...
    default:
        statements
}
```

- Only *switch* with expression and the braces are required
- All other labels and statements (including *break* and *default*) are optional.
- If exists, the *default*-label must be positioned after all *case*-labels.

Let's replace following *if/else*-statements by an (easier to read) *switch*-statement:

```
int weekday;
printf("Enter a day (1: Monday, 2: Tuesday, ..., 7: Sunday): ");
scanf("%d", &weekday);
getchar();

if (weekday == 1)
    printf("Monday - go to work\n");
else if (weekday == 2)
    printf("Tuesday - go to work\n");
else if (weekday == 3)
    printf("Wednesday - go to work\n");
else if (weekday == 4)
    printf("Thursday - go to work\n");
else if (weekday == 5)
    printf("Friday - go to work\n");
else if (weekday == 6)
    printf("Saturday - go shopping\n");
else if (weekday == 7)
    printf("Sunday - relax\n");
else
    printf("I don't know that day ... \n");
```

Example: Weekday

```
switch (weekday)
{
    case 1:
        printf("Monday - go to work\n");
        break;
    case 2:
        printf("Tuesday - go to work\n");
        break;
    case 3:
        printf("Wednesday - go to work\n");
        break;
    case 4:
        printf("Thursday - go to work\n");
        break;
    case 5:
        printf("Friday - go to work\n");
        break;
    case 6:
        printf("Saturday - go shopping\n");
        break;
    case 7:
        printf("Sunday - relax\n");
        break;
    default:
        printf("I don't know that day ... \n");
}
```



Modify the weekday example:

- Do not print the weekday, but “Go to work”, “Go shopping”, or “Relax”, respectively.

Sample solution:

```
switch (weekday)
{
    case 1:           Runs until break or closing bracket }
    case 2:
    case 3:
    case 4:
    case 5:
        printf("Go to work\n");
        break;
    case 6:
        printf("Go shopping\n");
        break;
    case 7:
        printf("Relax\n");
        break;
    default:
        printf("I don't know that day ... \n");
}
```

Unconditional jump (goto)

- Jump to a specific line of code

We could make this the shortest section of this lecture:

- There is a *goto*-statement ... do not use it (if possible).

With some more detail:

- Define a label: *identifier*: (Similar to *default* in *switch*-statements)
- Jump to a label: *goto identifier*;
- Jumps are unstructured ⇒ Difficult to comprehend ⇒ Error-prone

```
int main(void)
{
    printf("This is printed.\n");
    goto cleanup;
    printf("This is not printed.\n");

    cleanup:
    printf("I am cleaning up.\n");

    getchar();
    return 0;
}
```

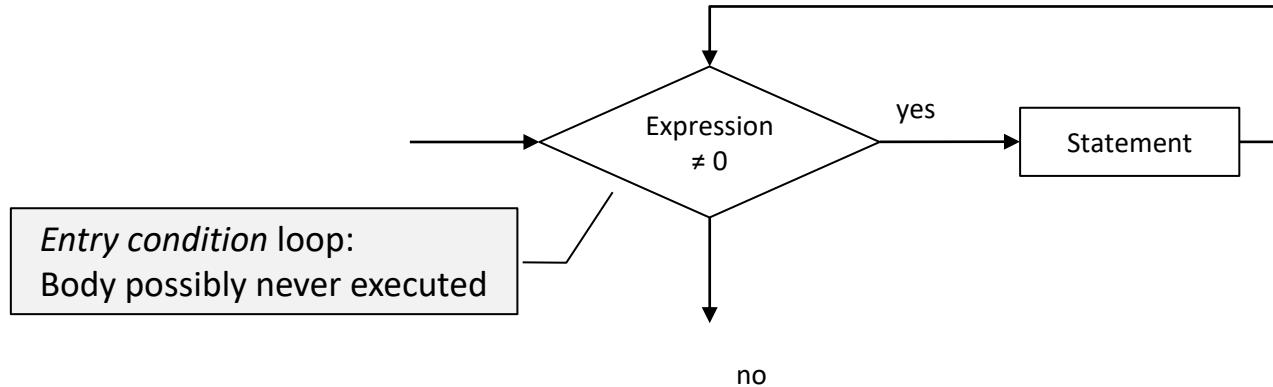
while-loop

- Keep on repeating a code block while a specific condition is fulfilled

Syntax to repeat a statement as long as a condition is fulfilled:

```
while (expression)  
    statement
```

- The same rules for *expression* and *statement* apply as discussed for *if*-statements.
- Executes the statement only if the expression is $\neq 0$ (\Rightarrow true)
- Tests the expression *after completion* of statement (*not* during execution of statement)
- Repeats statement until expression is = 0





What is printed to the console?

```
int a = 20;
```

```
while (a > 0)
{
    printf("a = %2d\n", a);
    a /= 2;
}
```

Display with ≥ 2 places

Integer division

- Console output:

```
a = 20
a = 10
a =  5
a =  2
a =  1
```



Think about it:

- What is the value of variable *a* after the *while*-loop?



Write following program:

- Print “Please give me a floor to stand upon.” to the console.
- Use a *while*-loop to underline the text by ‘-’ characters.
- Note: The *printf()* function returns the number of characters printed.

Sample solution:

```
int numberChars = printf("Please give me a floor to stand upon.\n");

while (numberChars > 1)
{
    printf("-");
    numberChars--;
}
```

Do not underline '\n'



Think about it:

- What happens, if we replace *numberChars--* by *--numberChars*?
- What happens, if we replace *numberChars--* by *numberChars++*?



Write following program:

- Repeatedly ask users to enter a character
- Leave the loop, if users enter 'q' or 'Q'.

Sample solution:

```
char userInput = 0;

while ((userInput != 'q') && (userInput != 'Q'))
{
    printf("Oh no, you're in a loop ('q' or 'Q' to quit): ");
    scanf("%c", &userInput);
    getchar();
}

printf("\nOkay, you've found your way out of the loop.\n");
```

Exercise: Average value



Write following program:

- Repeatedly ask users to enter a number.
- Stop the loop when a user enters a character instead of a number.
- Print to the console the average value of all numbers entered.

Note:

- The `scanf()` function returns the number of inputs correctly read and assigned.

Sample solution:

```
double number;
double sum = 0.0;
int count = 0, inputOK = 1;

/* Get user input: Numbers */
printf("Enter numbers (any character to stop):\n");
inputOK = scanf("%lf", &number);
while (inputOK)
{
    sum += number;
    count++;
    inputOK = scanf("%lf", &number);
}
getchar();      // Last scanf() could not read character
getchar();      // Get enter key of last input

/* Calculate and print mean value */
if (count > 0)
    printf("\nNumbers: %d\nAverage: %.1f\n", count, sum / (double) count);
```

Alternatively control the *while*-loop directly by *scanf()*:

```
double number;
double sum = 0.0;
int count = 0;

/* Get user input: Numbers */
printf("Enter numbers (any character to stop):\n");
while (scanf("%lf", &number))
{
    sum += number;
    count++;
}
getchar();           // Last scanf() could not read character
getchar();           // Get enter key of last input

/* Calculate and print mean value */
if (count > 0)
    printf("\nNumbers: %d\nAverage: %.1f\n", count, sum / (double) count);
```

do/while-loop

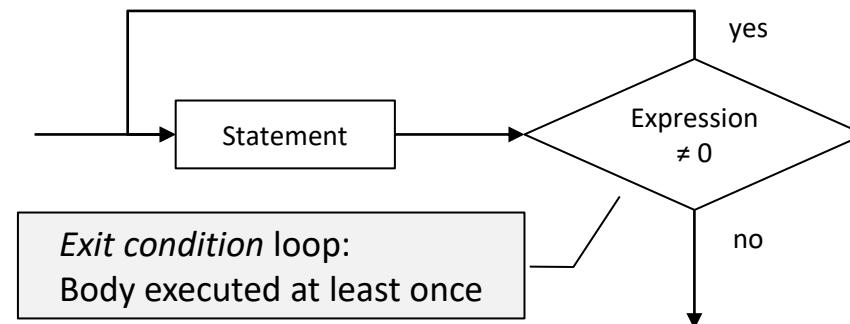
- Make sure the code of a *while*-block is executed at least once

Syntax to repeat a statement as long as a condition is fulfilled:

```
do
    statement
  while (expression);
```

Differences to *while*-loops:

- The statement part is executed before first testing the expression (\Rightarrow at least once).
- A *do/while*-loop ends with a semicolon.



Example:

- Loop until users enter 'q' or 'Q'.

```
char userInput;

/* Wait for user to press 'q' or 'Q' */
printf("Press 'q' or 'Q' to quit.\n");
do
{
    scanf("%c", &userInput);
} while ((userInput != 'q') && (userInput != 'Q'));
```

Don't forget the semicolon.

Notes:

- The brackets {...} are not required, but improve readability.
- Do not forget the semicolon after *while (condition)*.

Example:

```
const int PIN_CODE = 20099;
int pinEntered;

do
{
    printf("Enter 5-digit pin code: ");
    scanf("%d", &pinEntered);
} while (pinEntered != PIN_CODE);
getchar();

printf("\nPin correct. Here are happiness and wealth!\n");
```

for-loop

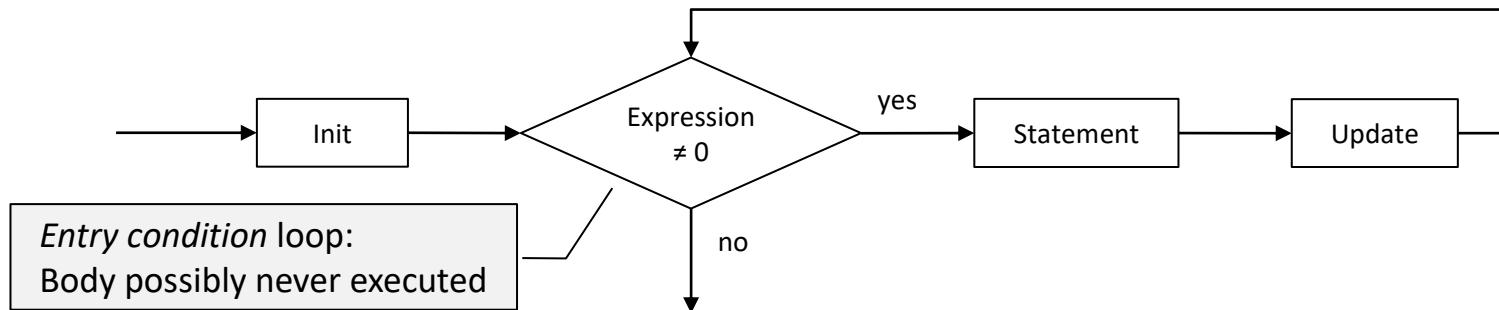
- Keep on repeating a specific code block
- Define the number of repetitions in advance

for-loop

Syntax typically iterating the statement n times (with n known in advance):

```
for (init; expression; update)  
    statement
```

- Processes the *init* statement before the loop
- Processes the *update* after each iteration of a loop's body (*statement*)
- Apart from that just the same as *while-loops*



Example: Summing up numbers



What does the following program do?

```
int sum = 0;  
  
for (int i = 1; i <= 100; i++)  
    sum += i;  
  
printf("Sum: %d\n", sum);
```

Answer:

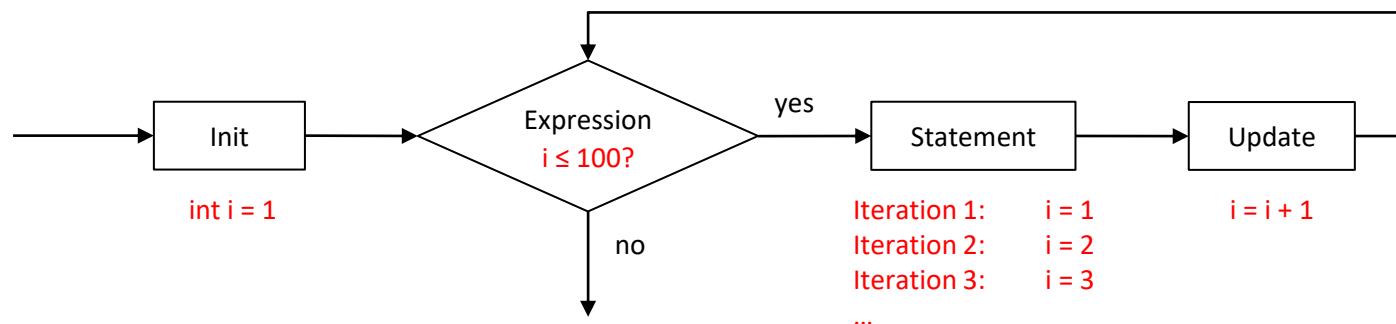
- Sums up all numbers from 1 to 100

Example: Summing up numbers

Variable i for controlling the number of iterations:

```
int sum = 0;  
  
for (int i = 1; i <= 100; i++)  
    sum += i;  
  
printf("Sum: %d\n", sum);
```

- *Init* declares a counter i existing within the *for-loop*, only.
- *Update* increases i after each iteration. \Rightarrow Variable i runs from 1, 2, 3, 4, ...
- *Expression* ends the loop as soon as i is greater than 100.





Do you remember the bank account?

- On January 1, there are 1,000.- € on a bank account.
- The rate of return is 2 % a year.
- Calculate and print the account balance after 1, 2, ..., 15 years.

Sample solution:

```
double balance = 1000.0;
double rate = 2.0;

printf("Initial balance: %.2f EUR\n", balance);
for (int year = 1; year <= 15; year++)
{
    balance += balance * rate / 100.0;
    printf("After %2d years : %.2f EUR\n", year, balance);
}
```

Example: Triangle



What is printed to the console?

```
for (int i = 1; i < 7; i++)  
{  
    for (int j = i; j > 0; j--)  
    {  
        printf("o");  
    }  
    printf("\n");  
}
```

Counter *i* runs from 1 to 6

Counter *j* runs from *i* to 1

Console output:

o
oo
ooo
oooo
ooooo
oooooo



Think about it:

- Replace following *for*-loop by a *while*-loop with the same functionality.

```
int sum = 0;  
for (int i = 1; i <= 100; i++)  
    sum += i;
```

Solution:

```
int sum = 0;  
int i = 1;  
while (i <= 100)  
{  
    sum += i;  
    i++;  
}
```

- Every *for*-loop can be written as *while*-loop:

```
for (init; expression; update)      → init;  
    statement                      while (expression)  
                                    {  
                                    statement  
                                    update;  
                                    }
```

- Every *while*-loop can be written as *for*-loop:

```
while (expression)      → for ( ; expression; )  
    statement          statement
```

Jump statements (break & continue)

- End the execution of a loop as a whole
- End the current iteration of a loop's body

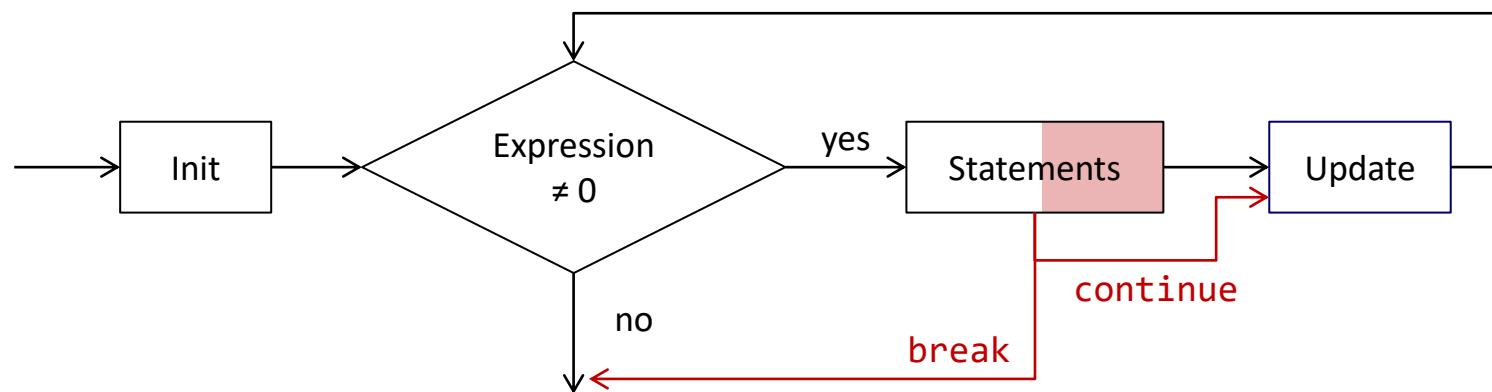
The *break*-statement:

- Immediately terminates a loop
- Proceeds after the loop

The *continue*-statement:

- Immediately terminates the current loop iteration
- Proceeds with *update* (*for*-loop) or *expression* (*while*-loop)

Visualization (*for*-loop):





What is printed to the console?

```
printf("Break (at i == 2):\n");
for (int i = 0; i <= 4; i++)
{
    if (i == 2)
        break;
    printf("i = %d\n", i);
}
```

Console output:

```
Break (at i == 2):
i = 0
i = 1
```



What is printed to the console?

```
printf("Continue (at i == 2):\n");
for (int i = 0; i <= 4; i++)
{
    if (i == 2)
        continue;
    printf("i = %d\n", i);
}
```

Console output:

```
Continue (at i == 2):
i = 0
i = 1
i = 3
i = 4
```



What is the effect of following source code?

```
while (getchar() != '\n')
    continue;
```

Answer:

- Discards (the rest of a) keyboard input until character '\n' *included*
- The *continue*-statement is not required, but improves readability compared to

```
while (getchar() != '\n')
    ;
```

and

```
while (getchar() != '\n');
```

Software Construction 1 (IE1-SO1)

4. Functions



Lecture overview

Fundamentals



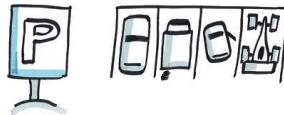
1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

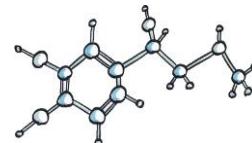
Advanced topics



5. Pointers



6. Memory management

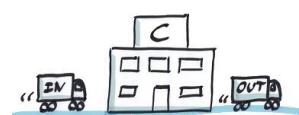


7. Structures



8. Lists and sorting

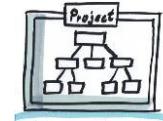
The next steps ...



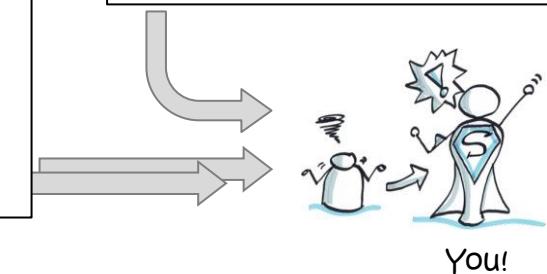
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *organize* tasks in functions in order to structure programs in dedicated and reusable modules.
- You execute and analyze programs step by step in order to identify and correct failures.



1. Introducing simple functions
2. Parameters & arguments
3. Function types & return values
4. Prototyping
5. Recursive functions
6. Introduction to debugging

Introducing simple functions

- Group a block of source code to a “module”

What is a function?

- Unit or module of program code that does a specific task
- Can do actions (e.g., print to screen) or provide values (e.g., math calculations)
- Examples: *printf()*, *scanf()*, *getchar()*

Some advantages:

- Modular programs \Rightarrow Better program structure
- Avoid code repetition within a program
- Reuse functions in other programs

Black-box view:

- Internal implementation is not of interest when using a function
- Interested in *what* a function does, not *how* it does it

A first example:

```
void printHawLogo(void);

int main(void)
{
    printHawLogo();
    getchar();
    return 0;
}

void printHawLogo(void)
{
    for (int i = 0; i < 4; i++)
        printf("oooooooooooo\n      *****\n");
```

Note:

- We will omit *include*-directives (but make sure to have them in your source code)
- The function name *printHawLogo()* is used differently 3 times.

A first example:

```
void printHawLogo(void);

int main(void)
{
    printHawLogo();
    getchar();
    return 0;
}

void printHawLogo(void)
{
    for (int i = 0; i < 4; i++)
        printf("oooooooooo\n      *****\n");
```

Definition

Function definition:

- Header and body with implementation of the functionality
- Common in C to have *main()* on top of all other functions in a source code file

A first example:

```
void printHawLogo(void);

int main(void)
{
    printHawLogo();
    getchar();
    return 0;
}

void printHawLogo(void)
{
    for (int i = 0; i < 4; i++)
        printf("oooooooooo\n      *****\n");
```

The diagram illustrates the relationship between function definitions and calls in C. It shows two boxes: 'Function call' pointing to the line 'printHawLogo();' and 'Definition' pointing to the line 'void printHawLogo(void) { ... }'. A bracket on the right side of the code groups the entire definition under the 'Definition' label.

Function call:

- *Call* (or *invoke*) a function from within a different function in the source code
- In principle an arbitrary number of function calls is allowed in a program.

Example: HAW logo

A first example:

```
void printHawLogo(void);
```

Prototype

```
int main(void)
{
    printHawLogo();
    getchar();
    return 0;
}
```

Function call

```
void printHawLogo(void)
{
    for (int i = 0; i < 4; i++)
        printf("oooooooooooo\n      *****\n");
```

Definition

Prototype (function declaration):

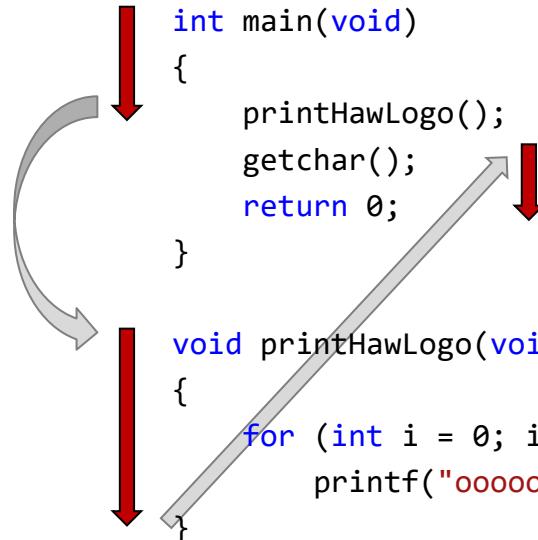
- Makes function “known” to the program (precise: to the code below the declaration)
- Must be stated *before* the first function call

A first example:

```
void printHawLogo(void);

int main(void)
{
    printHawLogo();
    getchar();
    return 0;
}

void printHawLogo(void)
{
    for (int i = 0; i < 4; i++)
        printf("oooooooooo\n      *****\n");
```



Program flow:

- At a function call program flow jumps to the appropriate function.
- At termination of a function the program flow jumps back to the function call.



Write a program with following function:

- Users shall enter a sentence.
- Print cleaned user input to the console.

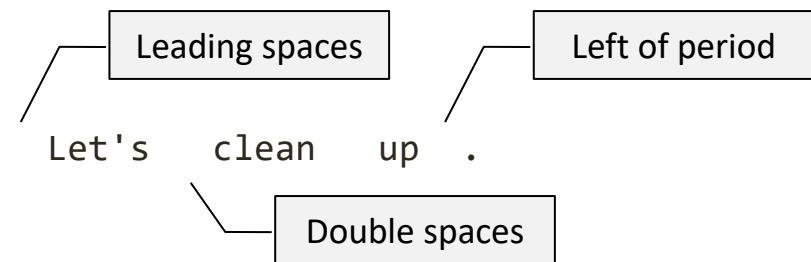
Cleaning rules:

- Not more than one adjacent space character
- No space character at the beginning of the input
- No space character left of a period (i.e., punctuation mark '.')

Example console output:

Enter sentence (cleans spaces):

Let's clean up.



Exercise: Clean text input

```
void cleanInput(void);           Prototype
int main(void)
{
    printf("Enter sentence (cleans spaces): ");
    cleanInput();
    getchar();
    return 0;
}
void cleanInput(void)           Definition
{
    char current, last = '\n';

    while ((current = getchar()) != '\n')
    {
        int isDoubleSpace = (last == ' ') && (current == ' ');
        int isLeadingSpace = (last == '\n') && (current == ' ');

        if ((last == ' ') && (current == '.'))           // Remove space before period
            printf("\b.");
        else if (!isDoubleSpace && !isLeadingSpace)      // Ignore double and leading spaces
            putchar(current);
        last = current;
    }
}
```

Same as `printf("%c", current)`

Parameters & arguments

- Give a function custom values to work with

- A function call can pass values to the called function.
- The function definition declares appropriate *parameters* within the parentheses.
- Terms: *parameters*, *formal parameters*, *formal arguments*

Parameter list:

- Parameters separated by commas
- Each parameter declared as *DataType Identifier*
- Use keyword *void*, if there are no parameters

Example:

```
void printMax(int a, int b)
{
    if (a > b)
        printf("max(%d, %d) = %d\n", a, b, a);
    else
        printf("max(%d, %d) = %d\n", a, b, b);
}
```

Two parameters of type *int*



- Values passed to a function are called *arguments* (or *actual arguments*)
- Can be constants (literals), variables, or expressions

Examples:

Argument of type *double*

- What is printed to the console?

```
int x = 4, y = 7;  
  
printMax(1, 2.25);           // Constants  
printMax(x, y);             // Variables  
printMax(3 * x, y *= x - 2); // Expressions  
printf("y = %d\n", y);
```

- Console output:

```
max(1, 2) = 2  
max(4, 7) = 7  
max(12, 14) = 14  
y = 14
```

Note the difference:

- Arguments: *Values* passed to a function
- Parameters: Local *variables* (i.e., declared only in that function)
- Function call assigns arguments (values) to the corresponding parameters (variables)

Exemplification:

```
int main(void)
{
    int x = 4;
    printMax(x, 7);           Arguments: 4, 7
    ...
}
```

```
void printMax(int a, int b)
{
    a = 4;
    b = 7;
    ...
}
```



Parameters as local variables

- Parameters are *local* variables (i.e., only declared and known in that function).
 - Memory is allocated for the parameters
- ⇒ Parameters and arguments are different variables, even if they have the same name.



What is printed to the console?

```
int main(void)
{
    int a = 3;
    printf("Main: a = %2d\n", a);
    printCube(a);
    printf("Main: a = %2d\n", a);

    getchar();
    return 0;
}

void printCube(int a)
{
    a = a * a * a;
    printf("Cube: a = %2d\n", a);
}
```

Variable *a* as argument

Parameter with same name *a*

Function types & return values

- Get a value (“result”) from a function

Functions can return a value:

- State the *data type* at the very left in the function header
- Statement *return expression*; ⇒ End function and return the value of *expression*

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

The diagram shows a C-style function definition for `max`. A callout box labeled "Declare return type" points to the `int` keyword at the start of the function header. Another callout box labeled "Return value of *a* (ends function)" points to the `return a;` statement inside the `if` block.

Functions without return value:

- State data type as *void* (i.e., “empty”)
- Can end function by *return*-statement without succeeding value

- A function call is replaced by the returned value.
- Hence, a function *has the type* of its return value.

Example:

- Function call is replaced by *int* value \Rightarrow Function is treated as data type *int*



What is printed to the console?

```
int x = 301, y = 126, z;           Used for assignment
int maxValue = max(x, y);

printf("max(%d, %d) = %d\n", x, y, maxValue);    Used as argument
printf("max(%d, %d) = %d\n", y, x/2, max(y, x/2));
z = (max(x, y) > 255) ? 255 : max(x, y);
printf("z = %d\n", z);               Used in expressions
```

Console output:

```
max(301, 126) = 301
max(126, 150) = 150
z = 255
```

Exercise: Maximum of three values



It's time for some "doing":

- Write a function *max3()* that returns the maximum of three *int* values.
- Do *not* declare a variable or use an *if*-statement within the function body.
- Given: Function *max()* to determine the maximum of two *int* values

Hints:

- Call *max()* within function *max3()*.
- Do not forget to add the prototypes on top of *main()*.

Exercise: Maximum of three values

Sample solution:

```
int max(int, int);           → Prototypes
int max3(int, int, int);

int main(void)
{
    int a = 45, b = 35, c = 101;
    printf("max(%d, %d, %d) = %d\n", a, b, c, max3(a, b, c));
    getchar();
    return 0;
}

int max(int a, int b)         → Returns value of operation
{
    return (a > b) ? a : b;
}

int max3(int a, int b, int c) → Returns value of function call
{
    return max(max(a, b), c);
}
```



Explain the functionality of following program:

```
void toCapital(char);

int main(void) {
    char letter;

    do
    {
        printf("Enter small letter: ");
        scanf("%c", &letter);
        getchar();
    } while ((letter < 'a') || (letter > 'z'));

    toCapital(letter);
    printf("Capital letter : %c\n", letter);
    getchar();
    return 0;
}

void toCapital(char letter)
{
    if ((letter >= 'a') && (letter <= 'z'))
        letter += 'A' - 'a';
}
```

Changes local variable *letter*,
but not *letter* in *main()*!

Example: Letters

Corrected program to change a letter from small to capital:

```
char toCapital(char);

int main(void) {
    char letter;

    do
    {
        printf("Enter small letter: ");
        scanf("%c", &letter);
        getchar();
    } while ((letter < 'a') || (letter > 'z'));

    letter = toCapital(letter); _____
    printf("Capital letter : %c\n", letter);
    getchar();
    return 0;
}

char toCapital(char letter)
{
    if ((letter >= 'a') && (letter <= 'z'))
        letter -= 32;
    return letter;
}
```

Assign capital letter

Returns capital letter

Prototyping

- Make known what functions exists
- Help the compiler check that functions are called correctly

- Recap: A function must be “known” (*declared*) before it is first used.

Different ways to declare a function:

1. Place the function definition before its first usage

```
int max(int a, int b)           Function definition
{
    return (a > b) ? a : b;
}

int main(void)
{
    printf("max(%d, %d) = %d\n", 4, 2, max(4, 2));
    getchar();
    return 0;
}                                Function call
```

- Recap: A function must be “known” (*declared*) before it is first used.

Different ways to declare a function:

1. Place the function definition before its first usage
2. State the function’s prototype on top of all function definitions



Prototype

```
int max(int, int);  
  
int main(void)  
{  
    printf("max(%d, %d) = %d\n", 4, 2, max(4, 2));  
    getchar();  
    return 0;  
}  
  
int max(int a, int b)  
{  
    return (a > b) ? a : b;  
}
```

- Recap: A function must be “known” (*declared*) before it is first used.

Different ways to declare a function:

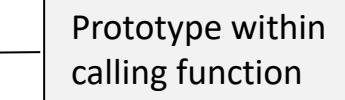
1. Place the function definition before its first usage
2. State the function’s prototype on top of all function definitions
3. State the function’s prototype within the functions making a function call

```
int main(void)
{
    int max(int, int);

    printf("max(%d, %d) = %d\n", 4, 2, max(4, 2));
    getchar();
    return 0;
}

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

Prototype within calling function



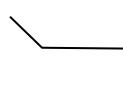
- Recap: A function must be “known” (*declared*) before it is first used.

Different ways to declare a function:

1. Place the function definition before its first usage
2. State the function’s prototype on top of all function definitions
3. State the function’s prototype within the functions making a function call
4. Include the function’s prototype on top of the source file

```
#include <hawMath.h>
int main(void)
{
    printf("max(%d, %d) = %d\n", 4, 2, max(4, 2));
    getchar();
    return 0;
}

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



Header file
with prototype

Keep following structure in source code files:

Include directives

Prototypes

Global variables

main() function

Other functions

Will discuss these later

Recommendations:

- Include header files for functions defined in other source code files
- Write and include header file, if functions can be used in other source code files as well
- State prototypes of functions defined and used within same source code file, only

Must be known to use a function:

- Function name
- Function parameters with data types (\Rightarrow Data type of arguments)
- Return type

Parameter names need not be known, though:

- These are names of variables used *locally inside* the function.
- \Rightarrow Prototypes need not contain parameter names (and may even use differing names)

Examples:

```
int max(int, int);           // No parameter names
int max(int a, int b);       // Same as function definition
int max(int x, int y);       // Different from function definition
```

An empty parameter list is *not* the same as *void*:

- Keyword *void* declares that there are *no* parameters.
- Empty list makes no statement on the parameters ⇒ Accepts any arguments

Recommendations:

- State *void* in prototypes and function definitions, if a function has no parameters.
- State the data types of a function's parameters in its prototype (i.e., no empty list).

Bad example:

```
int max();
```

Empty list allowed
(not recommended)

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

Bad example: Maximum of two (or three?) values



What is printed to the console?

```
int max();  
  
int main(void)  
{  
    printf("max(%d, %d, %d) = %d\n", 4, 2, 5, max(4, 2, 5));  
    getchar();  
    return 0;  
}  
  
int max(int a, int b)  
{  
    return (a > b) ? a : b;  
}
```

Argument 5 not used by *max()*

Console output:

```
max(4, 2, 5) = 4
```

Don't do that in
your math exam ...

Recursive functions

- Let a function call itself

- A *recursive* function calls itself (possibly many times)
- Standard example: Mathematical factorial $n! = \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdot 4 \cdots (n-1) \cdot n$

```
unsigned long long factorial(unsigned long long);

int main(void)
{
    printf("Factorials:\n\n");
    for (int n = 0; n <= 10; n++)
        printf("%2d! = %7llu\n", n, factorial(n));

    getchar();
    return 0;
}

unsigned long long factorial(unsigned long long n)
{
    if (n >= 2)
        return n * factorial(n - 1);
    else
        return 1;
}
```

Recursive function call

- Often the same functionality can be implemented using a loop.



Exercise:

- Implement mathematical factorial $n!$ using a loop.

Sample solution:

```
unsigned long long factorial(unsigned long long n)
{
    unsigned long long fac = 1;
    for (unsigned long long i = 2; i <= n; i++)
        fac *= i;

    return fac;
}
```



Think about it:

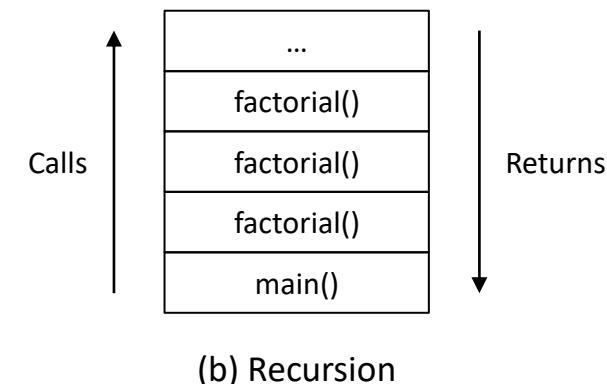
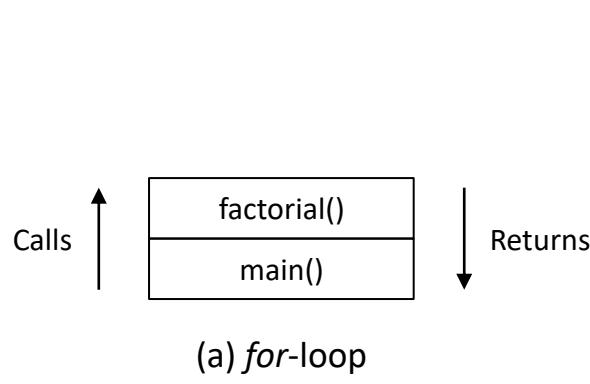
- What are typical disadvantages and advantages of recursive functions?
- What are typical disadvantages of choosing a recursion vs. a loop?

Disadvantages:

- A new set of variables allocated for *each* recursive call (e.g., n in `factorial()`)
- Slower (in particular due to allocation of variables and function call)

Advantages:

- Often an elegant and easy to understand solution





Think about it:

- What is the effect of following program?

```
void recur(void);

int main(void)
{
    recur();
    getchar();
    return 0;
}

void recur(void)
{
    recur();
}
```

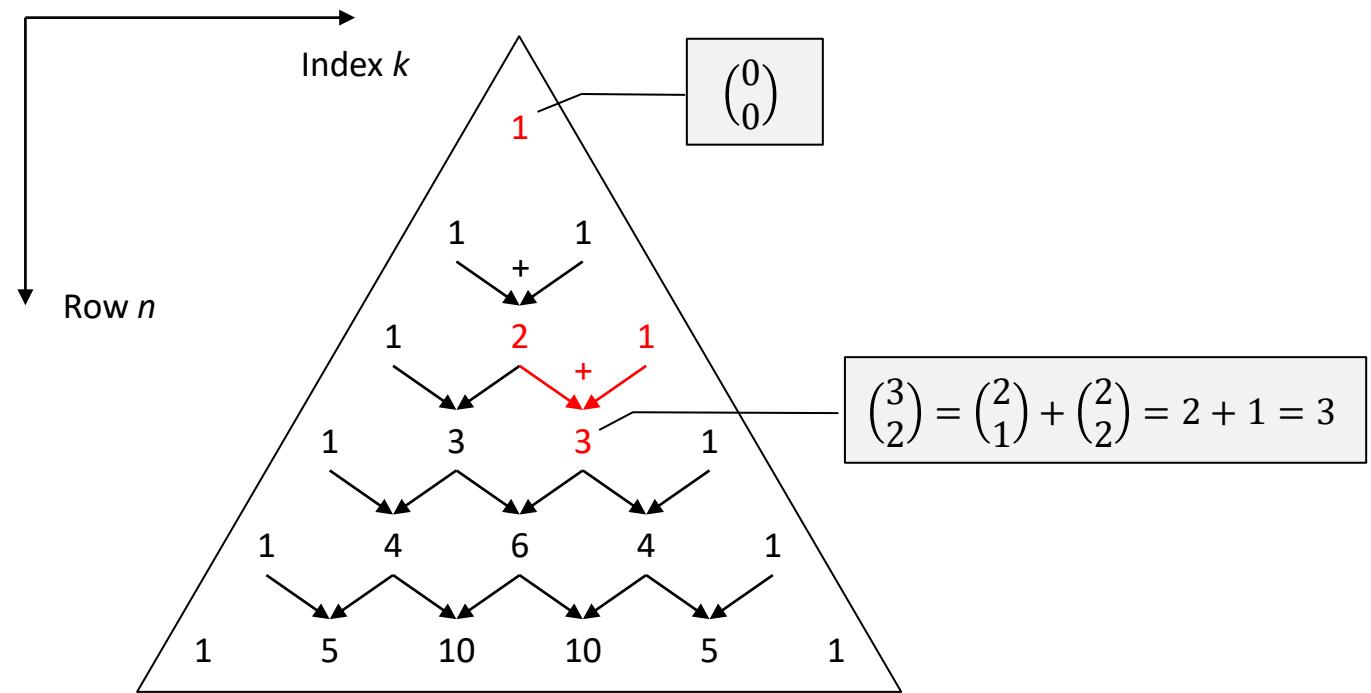
Answer:

- Function *recur()* calls itself “infinitely”, reserving some memory for each function call.
- Eventually there is no free stack memory and the program will crash (*stack overflow*).

Exercise: Pascal's triangle

Pascal's triangle:

- Triangular array of the binomial coefficients $\binom{n}{k}$ with $n \geq 0$ and $0 \leq k \leq n$
- Row n contains coefficients to calculate $(a + b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k$





Let's do some programming:

- Implement a recursive function *binomial()* returning coefficients $\binom{n}{k}$.
- Print the first rows of Pascal's triangle to the console.

Sample output:

```
Enter number of rows: 5
```

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Sample solution:

- Binomial coefficients:

```
int binomial(int n, int k)
{
    if ((k > 0) && (k < n))
        return binomial(n - 1, k - 1) + binomial(n - 1, k);
    else
        return 1;
}
```

- Print certain number of rows of Pascal's triangle:

```
void printPascal(int rows)
{
    for (int n = 0; n < rows; n++)
    {
        for (int k = 0; k <= n; k++)
            printf("%d ", binomial(n, k));
        printf("\n");
    }
}
```

- Main function:

```
int main(void)
{
    int rows;

    /* Get user input (number of rows) and print Pascal's triangle */
    printf("Enter number of rows: ");
    if (scanf("%d", &rows))
        printPascal(rows);
    else
        printf("Invalid input: Not a number.\n");

    /* Discard further characters entered by the user */
    while (getchar() != '\n')
        continue;

    /* Wait for keystroke before closing console window */
    getchar();
    return 0;
}
```

Introduction to debugging

- Tools to find errors in source code
- Step through running programs to observe what's happening

Terms:

- Software bug: Error or fault in source code
 - Debugging: Process of identifying and correcting software bugs

A first, simplified approach:

1. Remove all compiler errors (⇒ Correct syntax errors)
 2. Remove all compiler warnings (⇒ Correct unintended misuse of language)
 3. Establish expected program behavior (⇒ Correct logical errors)

Let's have a look at an example ...



Oh boy (or girl), what is wrong with that code?

```
#include <stdio.h>
#define SIZE 2

int getSumAbs(const double a[SIZE]);

int main(void)
{
    double a[] = { -2.7, 3.1, -4 },
    double sumAbs = getSumAbs(a);
    printf("Sum of absolute values: %.1f\n", sumAbs);
    getchar();
    return 0;
}

double getSumAbs(const double a[SIZE])
{
    double sum = 0.0;
    for (int i = 1; i < SIZE; i++)
        sum = abs(a[i]);
}
```

Step 1: Remove compiler errors

Compiler output:

```
1>----- Build started: Project: 06-01_Debug_Incorrect, Configuration: Debug Win32 -----
1>  buggy.c
1> ...\\buggy.c(20): error C2059: syntax error: 'type'
1> ...\\buggy.c(22): error C2065: 'sumAbs': undeclared identifier
1> ...\\buggy.c(22): warning C4477: 'printf' : format string '%.1f' requires an argument of type 'double',
but variadic argument 1 has type 'int'
1> ...\\buggy.c(28): error C2371: 'getSumAbs': redefinition; different basic types
1> ...\\buggy.c(14): note: see declaration of 'getSumAbs'
1> ...\\buggy.c(32): warning C4013: 'abs' undefined; assuming extern returning int
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

Approach:

- Correct errors one by one and recompile beginning with *the first* in the list
- Other messages might be related to subsequent errors.
- Tip: Double-click on message to “jump” to the corresponding line of code.

Be aware:

- The compiler indicates the first erroneous line.
- The error might be in the line *above!*

Step 2: Remove compiler warnings

Compiler output:

```
1> ...\\sandbox.c(33): warning C4013: 'abs' undefined; assuming extern returning int
1> ...\\sandbox.c(34): warning C4716: 'getSumAbs': must return a value
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

Approach:

- Again, correct warnings and recompile beginning with *the first* in the list

Console output:

- With warnings: Sum of absolute values: -nan(ind)
- With warnings removed: Sum of absolute values: 3.1

“Not a number”

- ⇒ The program runs, but the output is not as expected (sum = 9.8).
⇒ Use Visual Studio’s debugger to analyze the value of variable *sum* step by step.

Step 3: Establish expected program behavior

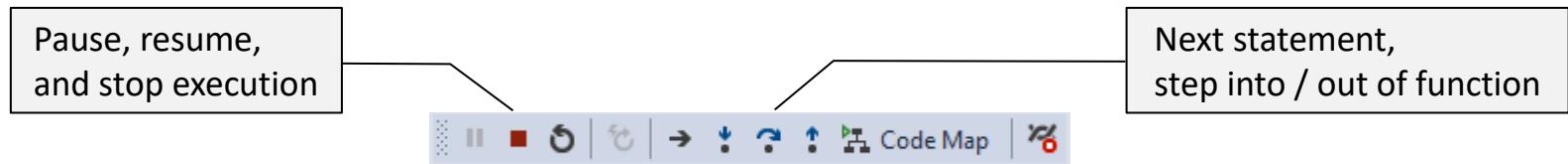
Visual Studio debugger:

- Runs until (next) *breakpoint*
- Click left of line number to set or remove a breakpoint
- Press F5 to compile and run program in debug mode (e.g., show values of variables)

A screenshot of the Visual Studio code editor for a file named "sandbox.c". The code defines a function "getSumAbs" that calculates the sum of absolute values of array elements. A red dot, indicating a breakpoint, is placed on the line number 28. A callout box labeled "Breakpoint" points to this red dot.

```
24     getchar();
25     return 0;
26 }
27
28 /* Get sum of absolute values of array elements */
29 double getSumAbs(const double a[SIZE])
30 {
31     double sum = 0.0;
32     for (int i = 1; i < SIZE; i++)
33         sum = fabs(a[i]);
34     return sum;
35 }
36
```

- Use toolbar icons to run through code step by step, e.g.:



Step 3: Establish expected program behavior

By stepping through `getSumAbs()`:

- The `for`-loop is executed only once (with $i = 1$)
- Oops ... we must start at $i = 0$ and define `SIZE` as 3.

Console output after corrections:

- Still not as expected (sum 4.0 instead of 9.8)
- Debug `for`-loop and see what happens to variable `sum`

The screenshot shows a debugger interface with the following details:

- Code View:** The code for `getSumAbs` is shown. It includes a comment /* Get sum of absolute values of array elements */. The loop starts at `i = 0` and ends at `i < SIZE`. The `sum` variable is initialized to 0.0 and updated to `sum = fabs(a[i])`.
- Breakpoint:** A red dot indicates a breakpoint is set on line 31.
- Autos View:** A table showing variable values:

Name	Value
a	0x006ffb38 {-2.7000000000000002}
a[i]	-4.0000000000000000
i	2
sum	3.1000000000000001
- Annotations:**
 - A callout box points to the assignment statement `sum = fabs(a[i]);` with the text "Assigns absolute instead of adding".
 - A callout box points to the value of `a[i]` in the Autos view with the text "Always absolute value of last array element".

Example: Sum of absolute values

We have corrected a lot of syntactical and logical mistakes:

```
#include <stdio.h>
#define SIZE 2
int getSumAbs(const double a[SIZE]);
int main(void)
{
    double a[] = { -2.7, 3.1, -4 },
    double sumAbs = getSumAbs(a);
    printf("Sum of absolute values: %.1f\n", sumAbs);
    getchar();
    return 0;
}

double getSumAbs(const double a[SIZE])
{
    double sum = 0.0;
    for (int i = 1; i < SIZE; i++)
        sum = abs(a[i]);
}
```

Annotations from top to bottom:

- Header *math.h* missing
- Must be 3
- Incorrect return type
- No semicolon
- Starting at 2nd element
- Assignment
- Function for *int* values

Example: Sum of absolute values

And this is the corrected source code:

```
#include <stdio.h>
#include <math.h>
#define SIZE 3

double getSumAbs(const double a[SIZE]);

int main(void)
{
    double a[] = { -2.7, 3.1, -4 };
    double sumAbs = getSumAbs(a);
    printf("Sum of absolute values: %.1f\n", sumAbs);
    getchar();
    return 0;
}

double getSumAbs(const double a[SIZE])
{
    double sum = 0.0;
    for (int i = 0; i < SIZE; i++)
        sum += fabs(a[i]);
    return sum;
}
```

Software Construction 1 (IE1-SO1)

5. Arrays



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



● 4. Arrays (and strings)

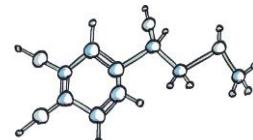
Advanced topics



5. Pointers



6. Memory management

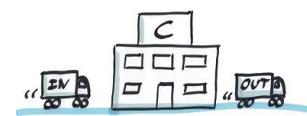


7. Structures



8. Lists and sorting

The next steps ...



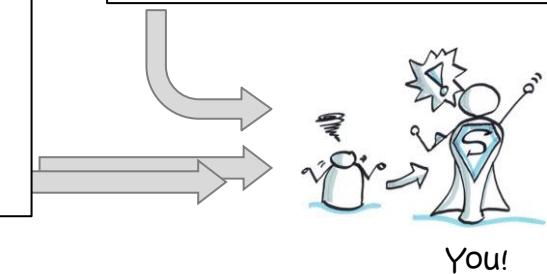
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *organize* data in 1- or 2-dimensional sets containing values of the same data type.
- You *use* character strings, for instance, to represent sentences or other texts.



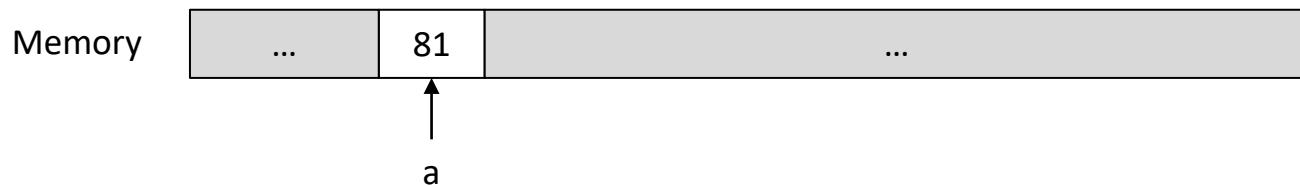
1. Declaration & initialization
2. Sizes & bounds
3. Multidimensional arrays
4. Arrays & functions
5. Character strings

Declaration & initialization

- Create a collection of values
- Assign and read values to/from the collection

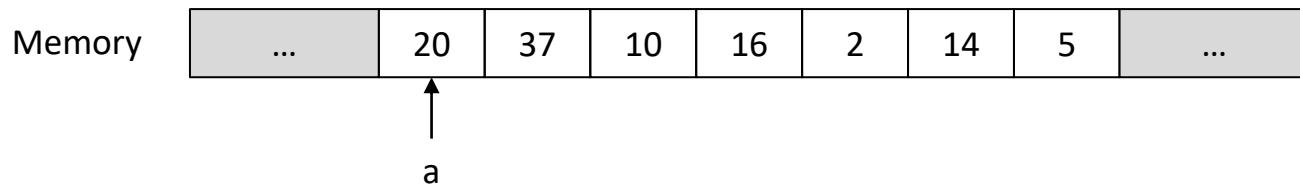
Variable of simple data type:

- Memory reserved for *one* value of a given type
- Declare an identifier (“name”) to access the value
- Example: Variable *a* of type *int*



Array:

- Collection of elements of the same data type
- Declare an identifier (“name”) to access *all* values in a set
- Example: Seven elements of type *int*



Declaration:

- Like a simple variable, but with number of elements (*length, size*) in brackets

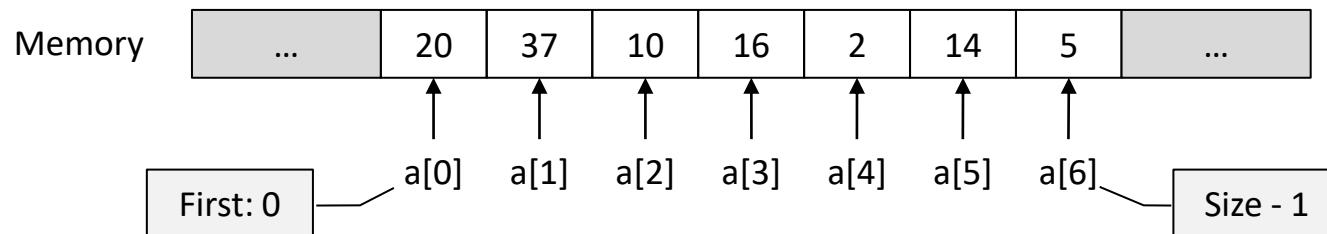
```
int a[7];
```

Access elements:

- Access elements by *index (subscript)* in brackets
- Indices start with 0 (\Rightarrow They run from 0 to *size - 1*).

```
a[0] = 20;  
a[1] = 37;  
a[2] = 10;  
  
...
```

Don't forget. Promise!



Exercise: Hamburg weather



Let's talk about Hamburg's weather:

- Store the average maximum temperature per month in an array.
- Ask users to enter a month (coded as Jan = 1 to Dec = 12).
- Print the corresponding temperature to the console.

Average maximum temperature in Hamburg¹:

Month	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Maximum temperature [°C]	2.7	3.8	7.2	11.9	17.0	20.2	21.4	21.6	18.0	13.3	7.6	4.0

¹ <http://www.wetter.de/klima/europa/deutschland/hamburg-s101470.html> (Visited Oct 2016)

Sample solution:

```
double maxTemp[12];
int month;

/* Initialize array */
maxTemp[0] = 2.7;
maxTemp[1] = 3.8;
maxTemp[2] = 7.2;
maxTemp[3] = 11.9;
maxTemp[4] = 17.0;
maxTemp[5] = 20.2;
maxTemp[6] = 21.4;
maxTemp[7] = 21.6;
maxTemp[8] = 18.0;
maxTemp[9] = 13.3;
maxTemp[10] = 7.6;
maxTemp[11] = 4.0;
```

Initialize array elements

```
/* Print data for month requested by user */
printf("Enter month:\n");
if (scanf("%d", &month) && (month >= 1) && (month <= 12))
    printf("%.1f deg C\n", maxTemp[month - 1]);
else
    printf("Invalid input: Must be in 1 (January) to 12 (December)\n");
```

Evaluated left to right

- Initializing each element by an assignment statement is cumbersome!
- Uninitialized elements can have any value (e.g., the bits that were already in memory)

Much easier:

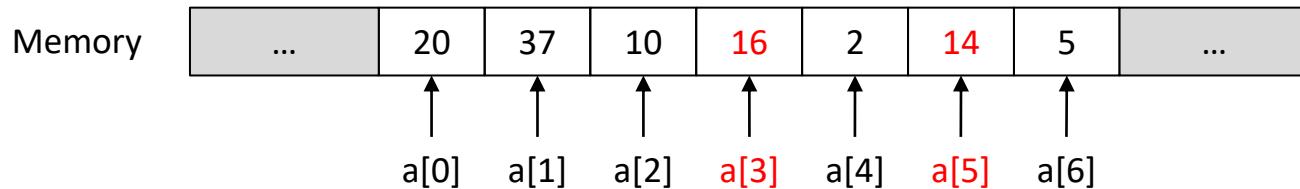
- Assign values as comma-separated list within braces { , , , ...}
- Allowed *only* at declaration of an array variable



What is printed to the console?

```
int a[7] = { 20, 37, 10, 16, 2, 14, 5 };
printf("a[%d] + a[%d]= %d", 3, 5, a[3] + a[5]);
```

Initial values of elements



Exercise: Hamburg weather



Use arrays to determine the following from the table below:

- Average maximum temperature (whole year)
- Average minimum temperature (whole year)
- Which month has the highest difference between average min. and max. temperature?

Average temperatures in Hamburg¹:

Month	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Maximum temperature [°C]	2.7	3.8	7.2	11.9	17.0	20.2	21.4	21.6	18.0	13.3	7.6	4.0
Minimum temperature [°C]	-2.2	-1.8	0.4	3.0	7.2	10.4	12.2	11.9	9.4	6.3	2.5	-0.7

¹ <http://www.wetter.de/klima/europa/deutschland/hamburg-s101470.html> (Visited Oct 2016)

Sample solution:

```
double maxTemp[12] = { 2.7, 3.8, 7.2, 11.9, 17.0, 20.2, 21.4, 21.6, 18.0, 13.3, 7.6, 4.0 };
double minTemp[12] = { -2.2, -1.8, 0.4, 3.0, 7.2, 10.4, 12.2, 11.9, 9.4, 6.3, 2.5, -0.7 };
double sumMax = 0.0, sumMin = 0.0, maxDifference = 0.0;
int monthMaxDifference = 0;

/* Analyze data */
for (int i = 0; i < 12; i++)
{
    sumMax += maxTemp[i];
    sumMin += minTemp[i];

    if (maxTemp[i] - minTemp[i] > maxDifference)
    {
        maxDifference = maxTemp[i] - minTemp[i];
        monthMaxDifference = i + 1;                      // i in 0 .. 11
    }
}

/* Print results to the console */
printf("Average maximum temperature: %.1f deg C\n", sumMax / 12.0);
printf("Average minimum temperature: %.1f deg C\n", sumMin / 12.0);
printf("Largest difference: %.1f deg C (month = %d)\n", maxDifference, monthMaxDifference);
```

Exercise: List too short



Write a sample program to investigate the following:

- What happens, if the initialization list contains less elements than the specified array size?

Sample solution:

Defines only 2 out of 5 values

```
int a[5] = { 7, 4 };
for (int i = 0; i < 5; i++)
    printf("a[%d] = %d\n", i, a[i]);
```

Console output:

```
a[0] = 7
a[1] = 4
a[2] = 0
a[3] = 0
a[4] = 0
```

Initialized from list

Filled with zeros

⇒ Values not defined by the list are initialized with 0.

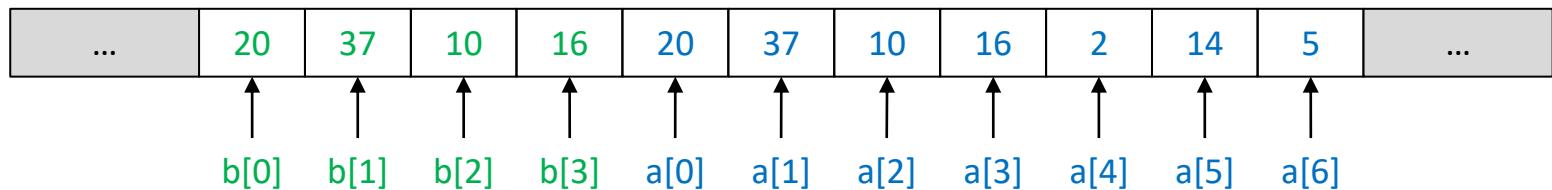
Note:

- You can omit the array size in brackets when initializing with a comma-separated list.
- The number of entries in the list determines the array size.

Depict the arrays (including the value of their elements) defined by following code:

```
int a[] = { 20, 37, 10, 16, 2, 14, 5 };  
int b[] = { 20, 37, 10, 16 };
```

Solution:



- Second list contains 4 elements \Rightarrow Size of array `b` set to 4. (No elements to fill with 0.)
- Sequent declared variables \Rightarrow Typically in adjacent memory locations (“right to left”)

Sizes & bounds

- Number of elements within an array
- What if an index is too small or large?

Standard ANSI C89/ISO C90:

- Array size during declaration must be an expression of constants (literals)
- “Constant variables” (keyword *const*) are *not* allowed.

```
// Okay
int a[12];
int b[128 * 128];

// Not okay
const int size = 12;
*x int c[size];
```

Standard ISO C99/C11:

- Can use a variable to define array size
- Specified in ISO C99, but downgraded to “optional” in ISO C11

```
int n = 12;
int a[n];
```



What is printed to the console?

```
int a[] = { 1, 6, 3, 1, 5, 8, 2, 3 };
int sum = 0;

for (int i = 0; i < 9; i++)
    sum += a[i];
printf("Sum: %d\n", sum);
```

Console output (sample run):

Sum: -858993431



Oops ... why's that?

- Array contains 8 elements ($\Rightarrow a[0]$ to $a[7]$)
- The *for*-loop sums up 9 elements $a[0]$ to $a[8]$, though.

Dilemma:

- Only constant expressions allowed to declare array sizes
- Need *same* constant value at other source code locations (e.g., to control *for-loop*)

Solution using *define* directive:

- Preprocessor directive (similar to `#include`): `#define identifier value`
- The preprocessor replaces every occurrence of *identifier* by *value* before compilation.

```
#define SIZE 8

int main(void)
{
    int a[SIZE] = { 1, 6, 3, 1, 5, 8, 2, 3 };

    for (int i = 0; i < SIZE; i++)
        sum += a[i];
    printf("Sum: %d\n", sum);

    getchar();
    return 0;
}
```

Compiler will read 8, not SIZE

Elegant alternative:

- The *sizeof* operator returns the size in bytes of the item to its right.
- Divide overall size of array by size of an array element \Rightarrow Number of elements in array

Example:

```
int main(void)
{
    int a[] = { 1, 6, 3, 1, 5, 8, 2, 3 };
    int size = sizeof a / sizeof a[0];
    int sum = 0;

    printf("Number of elements: %d\n", size);
    for (int i = 0; i < size; i++)
        sum += a[i];

    printf("Sum: %d\n", sum);

    getchar();
    return 0;
}
```

$$\frac{8 \cdot \text{size of } \text{int}}{\text{size of } \text{int}} = 8$$

Final, but important warning:

- You must make sure array indices are within bounds (0 to size – 1).
- No one else will do it for you.
- Really *no one*. Honestly. No kidding!



Example:

```
int a[4] = { 25, 10, 2, 7 };
printf("a[-1] = %d\n", a[-1]);
printf("a[4] = %d\n", a[4]);
```

Not specified

Notes:

- Not checking the bounds is faster during program execution.
- The C standard does *not specify* how to deal with indices being out of bounds.
⇒ All sorts of “bad things” could happen when using invalid indices.

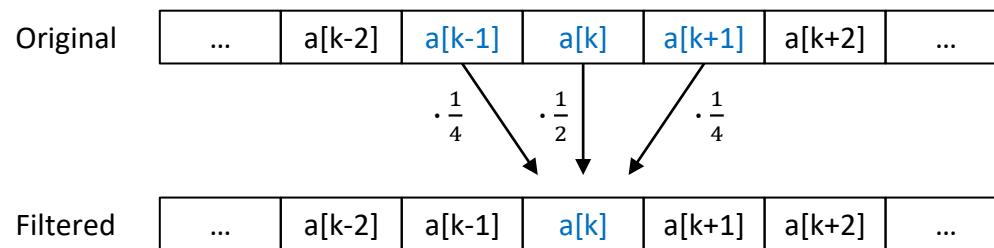
Exercise: Noise reduction (binomial filter)



- Given integer sequence: 95, 91, 211, 97, 89, 96, 94, 3, 91, 94, 92, 96, 93, 97, 94
- The sequence could be, e.g., an audio signal or pixels in a digital image.
- Process the data using the method below and display the result on the console.

Binomial filter:

- Replace each element $a[k]$ by the weighted mean $\frac{1}{4}(a[k - 1] + 2 \cdot a[k] + a[k + 1])$.
- Do not modify the first and the last element (which have only one neighbor).



Sample solution:

```
/* Define arrays */
int raw[15] = { 95, 91, 211, 97, 89, 96, 94, 3, 91, 94, 92, 96, 93, 97, 94 };

/* Apply filters to original data */
binomial[0] = raw[0];                                → Keep first and last element
binomial[size - 1] = raw[size - 1];
for (int i = 1; i <= size - 2; i++)
    binomial[i] = (raw[i - 1] + 2 * raw[i] + raw[i + 1]) / 4;

/* Print arrays to the console */
printf("Original: %3d", raw[0]);
for (int i = 1; i < size; i++)
    printf(", %3d", raw[i]);

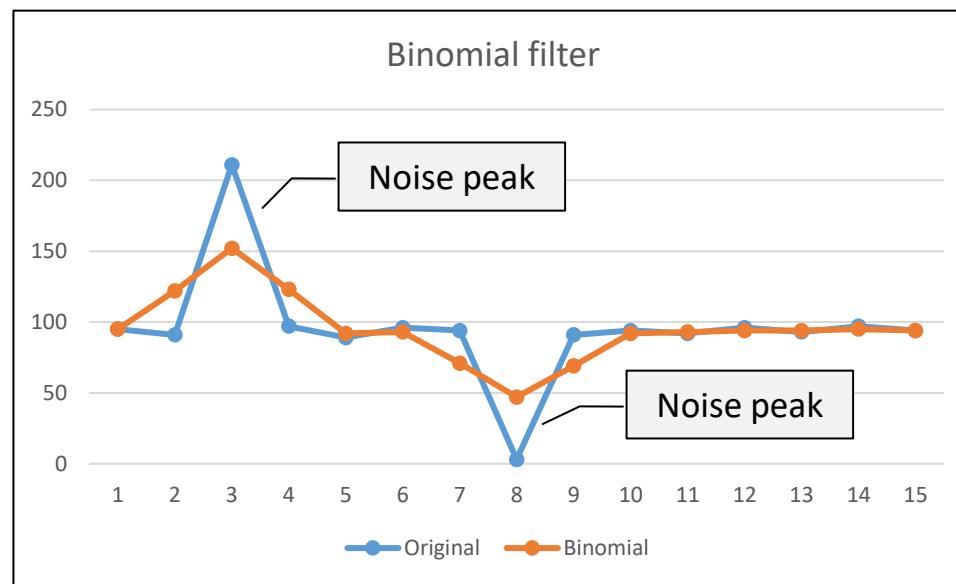
printf("\nBinomial: %3d", binomial[0]);
for (int i = 1; i < size; i++)
    printf(", %3d", binomial[i]);
printf("\n");

Filter other elements
```

Exercise: Noise reduction (binomial filter)

Console output and visualization:

Original: 95, 91, 211, 97, 89, 96, 94, 3, 91, 94, 92, 96, 93, 97, 94
Binomial: 95, 122, 152, 123, 92, 93, 71, 47, 69, 92, 93, 94, 94, 95, 94

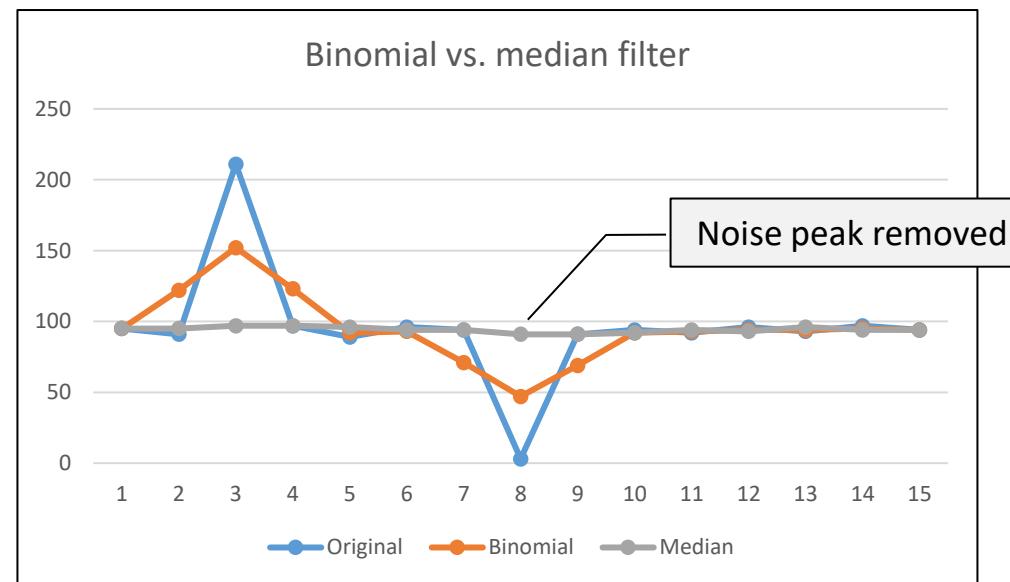


- The strong noise peaks have been reduced, but not removed.
- Do the exercise on median filters (see workbook) to remove noise peaks.

Exercise: Noise reduction (binomial filter)

Console output and visualization when applying a median filter:

Original:	95, 91, 211, 97, 89, 96, 94, 3, 91, 94, 92, 96, 93, 97, 94
Binomial:	95, 122, 152, 123, 92, 93, 71, 47, 69, 92, 93, 94, 94, 95, 94
Median:	95, 95, 97, 97, 96, 94, 94, 91, 91, 92, 94, 93, 96, 94, 94



Multidimensional arrays

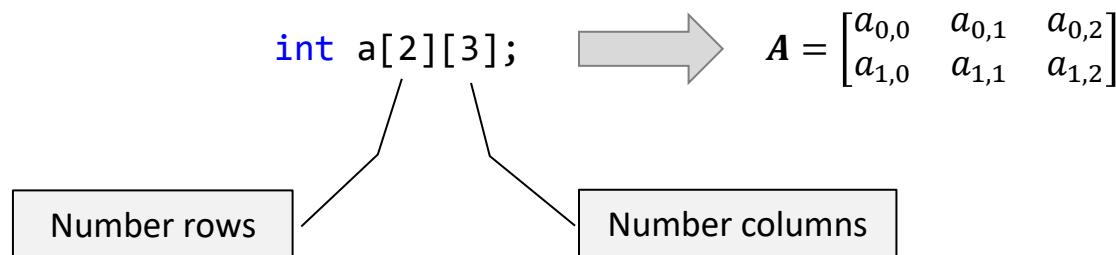
- Data in matrices and higher-dimensional collections

- Allowed to declare arrays with more than 1 dimension
- Example: 2-dimensional matrix $A \in M \times N$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{bmatrix}$$

Syntax:

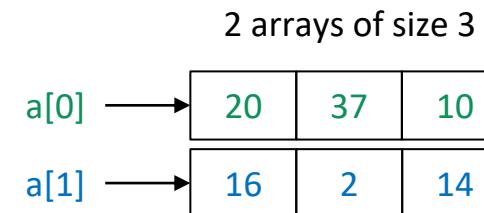
- Add additional brackets for each dimension
- Brackets contain size (declaration) or index (to access elements) for each dimension
- Indices count from 0 to size - 1 for each dimension



Visualization:

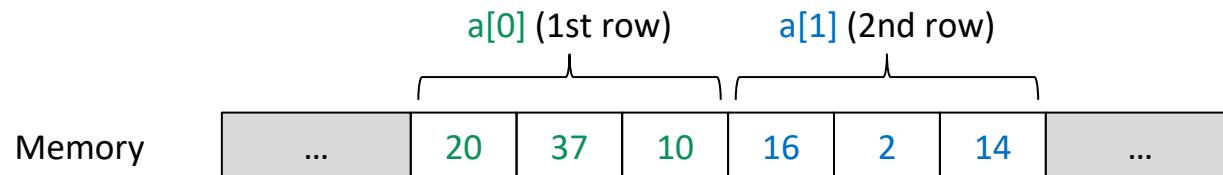
- Array containing arrays as elements (“array of arrays”)
- Example: Array containing 2 elements, each being an *int* array of size 3:

```
int a[2][3];
a[0][0] = 20;
a[0][1] = 37;
a[0][2] = 10;
a[1][0] = 16;
a[1][1] = 2;
a[1][2] = 14;
```



Internal storage in memory:

- Matrix elements are stored row by row from top to bottom



The same rules as for 1-dimensional arrays apply:

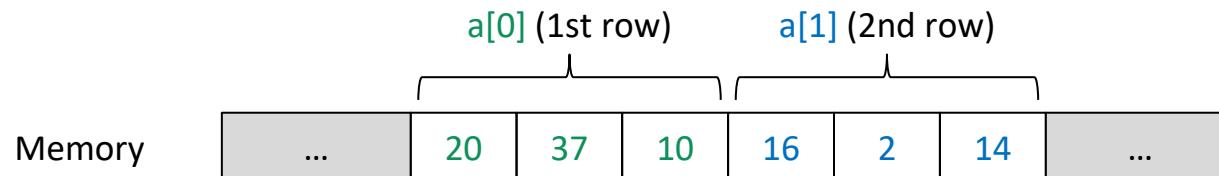
- Can initialize by explicitly assigning a value to every element (see previous slide)
- Can initialize by comma-separated list(s) in braces

Comma-separated lists:

- Can place each dimension in individual pair of braces (but this is not mandatory)
- Must not omit size of inner arrays (i.e., columns) in brackets
- Values not defined by a list are initialized with 0.

Examples:

```
int a[2][3] = { { 20, 37, 10 }, { 16, 2, 14 } };  
int a[2][3] = { 20, 37, 10, 16, 2, 14 };
```





What is printed to the console?

```
#define SIZE_X 3
#define SIZE_Y 2

int main(void)
{
    int a[SIZE_Y][SIZE_X] = { { 20, 37 }, { 16 } };

    for (int y = 0; y < SIZE_Y; y++)
    {
        for (int x = 0; x < SIZE_X; x++)
            printf("%2d  ", a[y][x]);
        printf("\n");
    }

    getchar();
    return 0;
}
```

Console output:

20	37	0
16	0	0



What is printed to the console?

```
#define SIZE_X 3
#define SIZE_Y 2

int main(void)
{
    int a[SIZE_Y][SIZE_X] = { 20, 37, 16 };

    for (int y = 0; y < SIZE_Y; y++)
    {
        for (int x = 0; x < SIZE_X; x++)
            printf("%2d  ", a[y][x]);
        printf("\n");
    }

    getchar();
    return 0;
}
```

Console output:

20	37	16
0	0	0

Exercise: Negative image



Digital image representation:

- A digital image consists of a 2-D matrix of image pixels.
- For 8-bit gray-valued images each pixel is within [0, 255].

Write following program:

- Initialize a 2-D pixel array with some arbitrary values in the range [0, 255].
- Create the negative pixel data by replacing each pixel value $a[m, n]$ by $255 - a[m, n]$.



Negative image



Sample solution:

```
#define SIZE 4

short gray[SIZE][SIZE] = {
    { 0, 6, 2, 8 },
    { 122, 102, 137, 126 },
    { 242, 255, 231, 247 },
    { 15, 21, 16, 8 } };

/* Invert image */
for (int y = 0; y < SIZE; y++)
{
    for (int x = 0; x < SIZE; x++)
        gray[y][x] = 255 - gray[y][x];
}

/* Print image data to the console */
for (int y = 0; y < SIZE; y++)
{
    for (int x = 0; x < SIZE; x++)
        printf("%3hd ", gray[y][x]);
    putchar('\n');
}
putchar('\n');
```

#define SIZE 4

Any numbers

Data type *short*

Arrays & functions

- Passing an array to a function

- Allowed to pass an array to a function
- Need to also pass the number of elements (“size of the array”) as argument

Example:

```
void printArray(int [], int);

int main(void)
{
    int a[] = { 20, 37, 10, 16, 2, 14 };
    printArray(a, 6);
    getchar();
    return 0;
}

void printArray(int a[], int size)
{
    for (int i = 0; i < size; i++)
        printf("%d ", a[i]);
    putchar('\n');
}
```

Diagram illustrating the flow of data between the main function and the printArray function:

- The variable `a` in the `main` function is passed as the first argument to the `printArray` function. This is labeled "Array name as argument".
- The integer value `6` in the `main` function is passed as the second argument to the `printArray` function. This is labeled "Number of elements".
- The parameter `a` in the `printArray` function is labeled "Array type as parameter".

- Just the same for 2-D arrays, but elements are 1-D arrays of fixed size

Example:

```
void printMatrix3(int a[][3], int rows);

int main(void)
{
    int data[][3] = { { 20, 37, 10 }, { 16, 2, 14 } };
    printMatrix3(data, 2);
    getchar();
    return 0;
}

void printMatrix3(int a[][3], int rows)
{
    for (int y = 0; y < rows; y++)
    {
        for (int x = 0; x < 3; x++)
            printf("%2d ", a[y][x]);
        putchar('\n');
    }
}
```

Diagram annotations:

- A callout box points to the parameter `rows` in the `printMatrix3` function definition with the text "List defines number of elements".
- A callout box points to the declaration `int data[][3]` with the text "Array of elements type `int[3]`".
- A callout box points to the parameter `rows` in the `printMatrix3` function call with the text "Number of elements".



Now it is your turn:

- Write a function that determines the sum of all elements in a 1-D array.

Sample solution:

```
int main(void)
{
    int a[6] = { 20, 37, 10, 16, 2, 14 };

    printf("Sum of elements: %d\n", arraySum(a, 6));
    getchar();
    return 0;
}

int arraySum(int a[], int size)
{
    int sum = 0;

    for (int i = 0; i < size; i++)
        sum += a[i];
    return sum;
}
```

Character strings

- Texts and other strings as array of *char* elements

- A *string* is a set of characters.
- Characters can be printable (e.g., “abc”) or non-printable (e.g., escape sequences like ‘\a’).

Data representation:

- Array of type *char[]* containing the *null character* ‘\0’
- Character ‘\0’ is mandatory and marks the end of string
- Character ‘\0’ differs from the digit ‘0’ and has numeric value 0 (e.g., ASCII)

Example:

- The character array below is a string, because it contains ‘\0’.
- The array has 11 elements, the **string** consists of 7 (“Hi ho!\n”) elements plus ‘\0’.



Different ways to declare and initialize a string:

1. String constant (literal) in double quotes
2. Initialize array by comma-separated list of elements in braces { , , , ... }
3. Declare array of type *char[]* and assign elements characters by character

Examples:

```
char hamburg[] = "Hamburg";
```

'\0' implicitly appended

```
char haw[] = { 'H', 'A', 'W', '\0' };
```

'\0' explicitly required

```
char abc[27];
```

```
for (int i = 0; i < 26; i++)  
    abc[i] = (char)('a' + i);  
abc[26] = '\0';
```

'\0' explicitly required



What is printed to the console?

```
char hamburg[] = "Hamburg";
char haw[] = { 'H', 'A', 'W', '\0' };
char abc[42];

for (int i = 0; i < 26; i++)
    abc[i] = (char)('a' + i);
abc[26] = '\0';

printf("hamburg: %2d\n", sizeof(hamburg) / sizeof(char));
printf("haw      : %2d\n", sizeof(haw) / sizeof(char));
printf("abc      : %2d\n", sizeof(abc) / sizeof(char));
```

Parentheses to get
size of a data type

Console output:

hamburg: 8

7 letters + character '\0'

haw : 4

abc : 42

Array size (42) larger
than required (26 + 1)

- Adjacent string literals are concatenated into one string.
- White spaces (i.e., space, tab, new line) between literals are ignored.

Example:

- Following function calls generate the same output on the console:

```
printf("The Globe Sessions\n");
printf("The Globe " "Sessions\n");
printf("The Globe "
      "Sessions\n");
```

- Console output:

```
The Globe Sessions
The Globe Sessions
The Globe Sessions
```

- The `printf()` function accepts the format specifier `%s` for strings.

What is printed to the console?



```
char hamburg[] = "Hamburg";
char haw[] = { 'H', 'A', 'W' };
char abc[30];

for (int i = 0; i < 26; i++)
    abc[i] = (char)('a' + i);

printf("hamburg: %s\n", hamburg);
printf("haw      : %s\n", haw);
printf("abc      : %s\n", abc);
```

Oops ... '\0' is missing!

Console output (sample run):

```
hamburg: Hamburg
haw      : HAW|||||||Hamburg
abc      : abcdefghijklmnopqrstuvwxyz|||||||HAW|||||||Hamburg
```

Prints characters until next '\0' in memory

- The `scanf()` function accepts the format specifier `%s` for strings.
- It reads until the next whitespace (i.e., space, tab, new line), excluded.

Notes:

- Pass the variable's name *without* brackets or ampersand & to `scanf()`.
- Make sure the character array is long enough for the input!

Example:

```
char word[256];  
  
printf("Talk to me, will you?: ");  
scanf("%s", word);
```

No ampersand &



What is printed on console, if a user enters “Marc Hensel” and presses [Enter]?

```
char name[256];
char keystroke;

/* Get user input */
printf("Please enter your name: ");
scanf("%s", name);

/* Print string and keyboard buffer to the console */
printf("Name : %s\n", name);
printf("Buffer: ");
while ((keystroke = getchar()) != '\n')
    putchar(keystroke);
putchar('\n');
```

Console output:

```
Please enter your name: Marc Hensel
Name : Marc
Buffer: Hensel
```



Write following program:

- Given is the string “Laaking aut af the windaw.”
- Replace all occurrences of ‘a’ by ‘o’.

Sample solution:

```
char text[] = "Laaking aut af the windaw.";
int size = sizeof text / sizeof(char);

/* Print original string to the console */
printf("Original: %s\n", text);

/* Replace 'a' by 'o' */
for (int i = 0; i < size; i++)
{
    if (text[i] == 'a')
        text[i] = 'o';
}

/* Print modified string to the console */
printf("Replaced: %s\n", text);
```

Exercise: Find substring



Write following program:

- Users shall enter a word.
- Determine, whether the word is a substring of John Lennon's sentence "Life is what happens while you are busy making other plans."

Notes:

- Compare in a case-sensitive way (e.g., 'a' ≠ 'A').
- Test your program, in particular, for the user inputs "app", "Life", and ".".
- Ask your parents, who John Lennon was.

Exercise: Find substring

Sample solution (implementation of search on next slide):

```
char word[256];
char quote[] = "Life is what happens while you are busy making other plans.";
int size = sizeof quote / sizeof(char);
int subID = -1; // Index of substring in string

/* Get user input */
printf("\%s\n", quote);
for (int i = 0; i < size - 1; i++)
    printf("%d", i % 10);

printf("\n\nEnter word to search for: ");
scanf("%s", word);
while (getchar() != '\n')
    continue;

/* Search for substring */
// See next slide ...

/* Print result to the console */
if (subID >= 0)
    printf("The string contains \"%s\" at index %d.\n", word, subID);
else
    printf("The string does not contain the substring \"%s\" (case-sensitive comparison).\n", word);
```

Value -1 ⇒ Substring not found

Discard rest of keyboard input

Sample solution (implementation of search, not optimized):

```
/* Search for substring */
for (int i = 0; i < size; i++)
{
    // Found first character of substring
    if ((subID < 0) && (quote[i] == word[0]))
        subID = i;

    // Compare character of substring to string
    if (subID >= 0)
    {
        // End of word reached => Substring found
        if (word[i - subID] == '\0')
            break;

        // Different characters
        else if (quote[i] != word[i - subID])
        {
            i = subID;
            subID = -1;
        }
    }
}
```

Note carefully!

Software Construction 1 (IE1-SO1)

6. Pointers



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

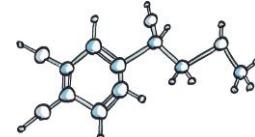
Advanced topics



5. Pointers



6. Memory management

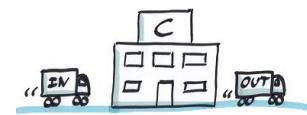


7. Structures



8. Lists and sorting

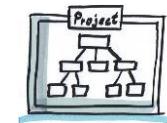
The next steps ...



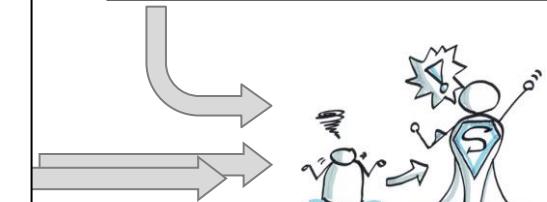
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



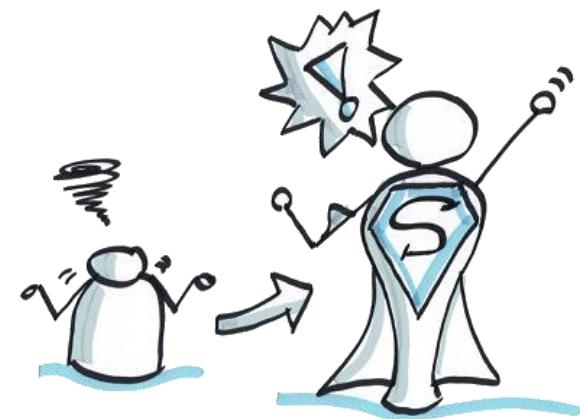
You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *use* memory locations as function arguments in order to receive more than one result (“return values”) from a function.
- You *process* and/or *modify* one-dimensional and two-dimensional arrays in functions.
- You *apply* standard string functions (e.g., for keyboard input or string modifications), not introducing security risks due to unsafe memory access.



1. Pointers as data type
2. Function parameters (call by reference)
3. Arrays & pointer operations
4. Multidimensional arrays
5. Strings

Pointers as data type

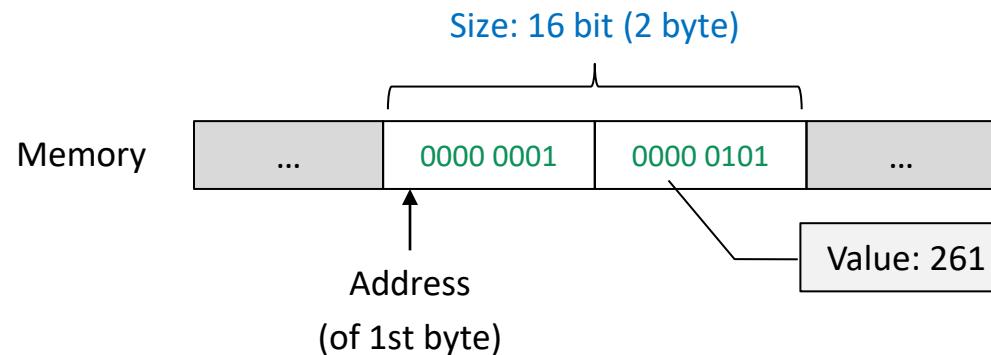
- Pointer variables
- Operations on pointers

Let's recall data storage in memory:

- Location (or *address*) in memory (Where?)
- Memory size (How many bits? Depends on type.)
- Bit pattern (What value? Depends on type.)

Example:

- 16-bit integer of type *short* (\Rightarrow e.g., 2 standard byte)
- Bits interpreted as *short*: $2^8 + 2^2 + 2^0 = 256 + 4 + 1 = 261$



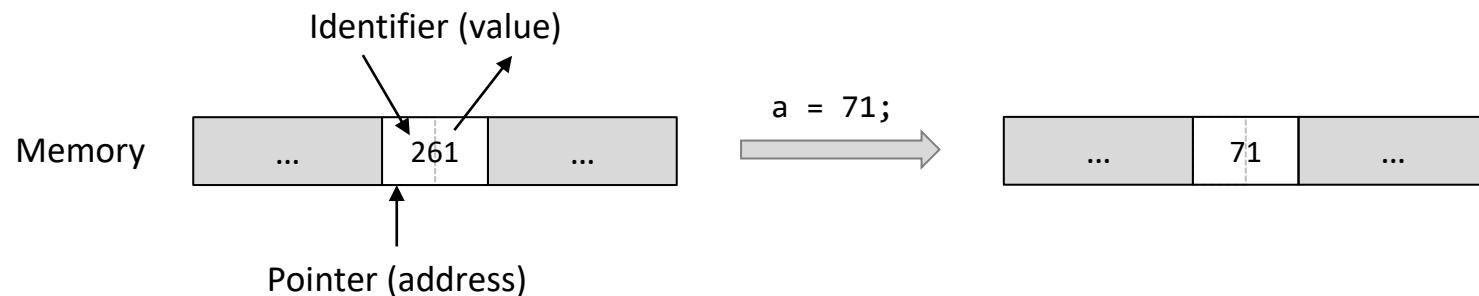
Common variable:

- Identifier to access (assign and read) the stored *value*

```
short a = 261;  
a = 71;  
printf("a = %d\n", a);
```

Pointer:

- Data type for memory *addresses*
- Variable used to store an address (i.e., location) in memory



Pointer variable:

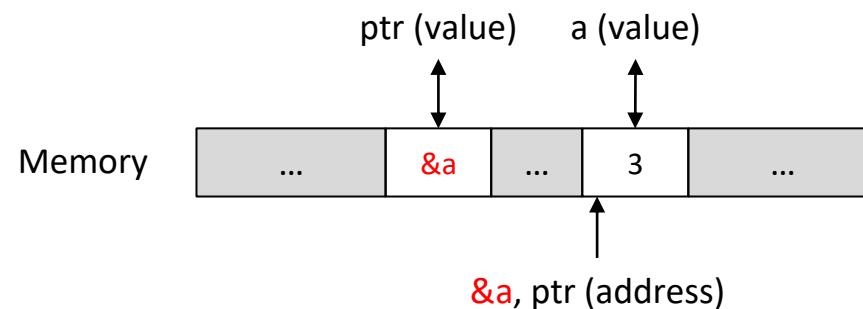
- Declaration: *DataType *identifier*
- Data type declares the size in memory and how to interpret the bits
- Example: Pointer to *int*

```
int *ptr;
```

Address operator &:

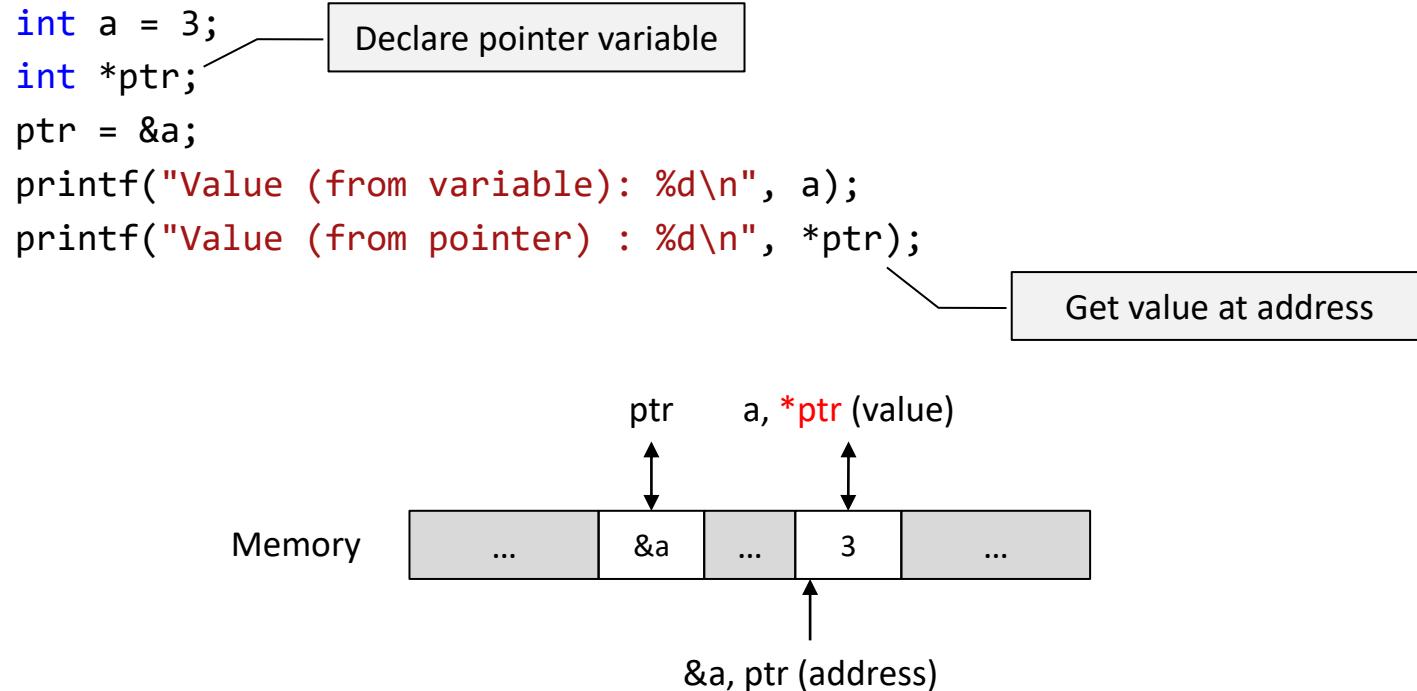
- Address where an element is stored in memory: *&identifier*
- We have already encountered this as argument passed to *scanf()* function.

```
int a = 3;  
int *ptr = &a;
```



Indirection operator *:

- To get the value at a memory location: `*address`
- Also called *dereferencing* operator
- Do not confuse this with an asterisk * used to *declare* a pointer variable.





What is printed to the console?

```
int a = 3;
int *ptr = &a;
int b = *ptr;

printf("a : %d\n", a);
printf("*&a: %d\n", *&a);
printf("b : %d\n\n", b);
```

Console output:

```
a : 3
*&a: 3
b : 3
```

Notes:

- Expression `*&a` is equivalent to `a`.
- Expression `b = *ptr` is equivalent to `b = *&a = a`.



What is printed to the console for the user input “4”?

```
int a = 3;  
int *ptr = &a;  
int b = *ptr;
```

Assign value of *a* to *b*

```
printf("Please enter an integer: ");  
scanf("%d", ptr);  
getchar();
```

Set new value at *address of a*

```
printf("a    : %d\n", a);  
printf("*ptr: %d\n", *ptr);  
printf("b    : %d\n", b);
```

Console output:

```
Please enter an integer: 4  
a    : 4  
*ptr: 4  
b    : 3
```

- Function `printf()` uses the format specifier `%p` for addresses (e.g., pointers)
- Prints addresses in *hexadecimal* number format (see next slide)

Example:

```
int a = 5;  
int *ptr = &a;  
  
printf("Address: %p\n", &a);  
printf("Pointer: %p\n", ptr);  
printf("Value : %d\n", a);
```

Console output (sample run):

```
Address: 0078F79C  
Pointer: 0078F79C  
Value : 5
```

Address in hexadecimal format

Hexadecimal numbers

- Numbers have a basis of 16 digits
- The letters A to F (small or capital) correspond to the values 10 to 15.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Notes:

- Indicate by prefix “0x” or parentheses $(\dots)_{16}$ that a number is in hexadecimal format
- Each digit (values 0 ... 15) represents 4 bits (\Rightarrow 2 digits for an 8-bit byte)

Give it a try:

- What decimal number is represented by $0x02AC = (02AC)_{16}$?
- What bit pattern is represented by $0x02AC = (02AC)_{16}$?
- Solution:

$$(02AC)_{16} = 0 \cdot 16^3 + 2 \cdot 16^2 + 10 \cdot 16^1 + 12 \cdot 16^0 = 684$$

$$(02AC)_{16} = 0000 \textcolor{red}{0010} \textcolor{blue}{1010} \textcolor{green}{1100}$$



What is printed to the console?

```
void printAddress(int);
```

```
int main(void)
{
    int a = 42;
    printf("main()\t\t: %p\n", &a);
    printAddress(a);
    getchar();
    return 0;
}
```

Print address of *a*

```
void printAddress(int a)
{
    printf("printAddress()\t: %p\n", &a);
}
```

Print address of *a*

Console output (sample run):

```
main()      : 004FF80C
printAddress() : 004FF738
```

... but two variables *a*
with different addresses

Function parameters (call by reference)

- Passing addresses of variables to functions
- Functions “returning” more than one value



What is the effect of the *swap()* function?

```
int main(void)
{
    int x = 15, y = 41;
    printf("x = %d, y = %d\n", x, y);
    swap(x, y);
    printf("x = %d, y = %d\n", x, y);
    getchar();
    return 0;
}

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```



Console output:

Before: x = 15, y = 41
After : x = 15, y = 41

No effect, sorry! But why?

Excercise: Swapping values



Modify the previous code so that `swap()` interchanges the values of `x` and `y` in `main()`.

```
int main(void)
{
    int x = 15, y = 41;
    printf("Before: x = %d, y = %d\n", x, y);
    swap(&x, &y); —————— Ampersand & to pass addresses to swap()
    printf("After : x = %d, y = %d\n", x, y);
    getchar();
    return 0;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp; —————— Asterisk * to access values
}
```

Console output:

```
Before: x = 15, y = 41
After : x = 41, y = 15
```

- Can use pointer type parameters to pass memory addresses to a function
- This is known as *call by reference* (i.e., reference to variable).

Term	Passes	Function declaration	Function call
Call by value	Value	<code>void func(int a) {...}</code>	<code>func(x);</code>
Call by reference	Address (pointer)	<code>void func(int *a) {...}</code>	<code>func(&x);</code>

When to use call by value:

- Function uses the values (e.g., of variables), only

When to use call by reference:

- In general, change variables in *calling* function (e.g., *swap()*)
- “Return” more than one value (see next example)
- Pass an array to a function (We will discuss this in a minute.)

Write a function that transforms Cartesian coordinates (x, y) to polar coordinates (r, φ) :

- Radius: $r = \sqrt{x^2 + y^2}$
- Angle (in rad): $\varphi = \text{atan2}(y, x)$

Notes:

- Square root and *atan2()* functions and constant π are declared in *math.h*.
- Mathematical *tan()* is periodic in π .
- The *atan2()* function returns the unique result in $[-\pi, \pi]$.
- Enable π by `#define _USE_MATH_DEFINES` before `#include <math.h>`

Sample output:

```
Cartesian: (x, y) = (2.00, 3.75)
Polar     : (r, phi) = (4.25, 0.34 * pi)
```

Sample solution:

```
#define _USE_MATH_DEFINES      // Enable math constants like pi in math.h
#include <stdio.h>
#include <math.h>

void toPolarCoords(double x, double y, double *r, double *phi);

int main(void)
{
    double x = 2.0, y = 3.75;
    double r, phi;

    toPolarCoords(x, y, &r, &phi);
    printf("Cartesian: (x, y) = (%.2f, %.2f)\n", x, y);
    printf("Polar     : (r, phi) = (%.2f, %.2f * pi)\n", r, phi / M_PI);

    getchar();
    return 0;
}

void toPolarCoords(double x, double y, double *r, double *phi)
{
    *r = sqrt(x * x + y * y);
    *phi = atan2(y, x);
}
```

Arrays & pointer operations

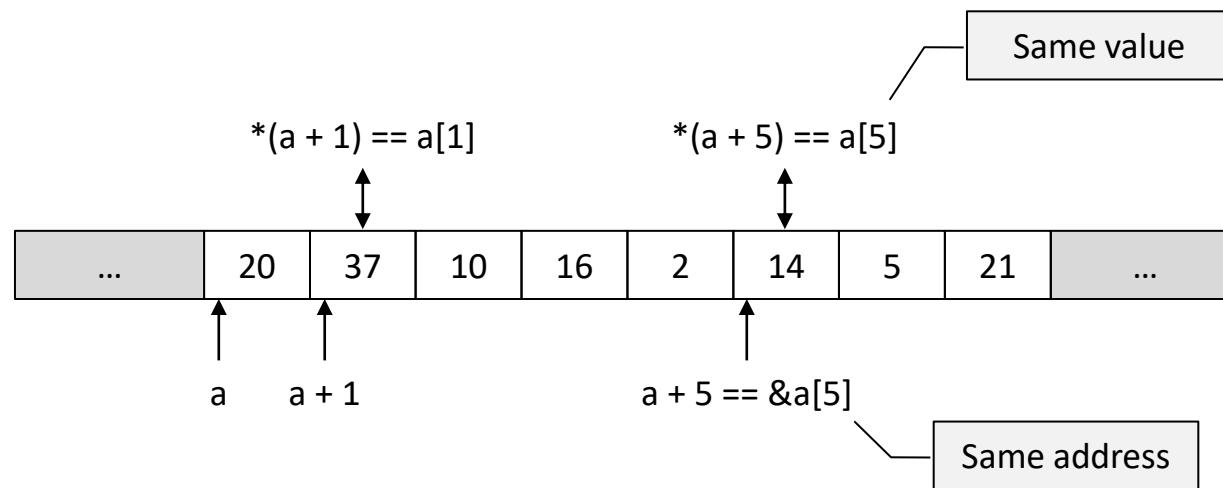
- Accessing array elements by pointers
- Passing arrays to functions

Two fundamental properties:

- An array name is the *address* of its first element ($\Rightarrow a$ is the same as $\&a[0]$).
- Adding 1 to a pointer moves the pointer to the address of the next element.

Example:

```
int a[] = { 20, 37, 10, 16, 2, 14, 5, 21 };
printf("a == &a[0] : %d\n", a == &a[0]);
printf("*a + 1) == a[1] : %d\n", *(a + 1));
printf("*a + 5) == a[5] : %d\n", *(a + 5));
```

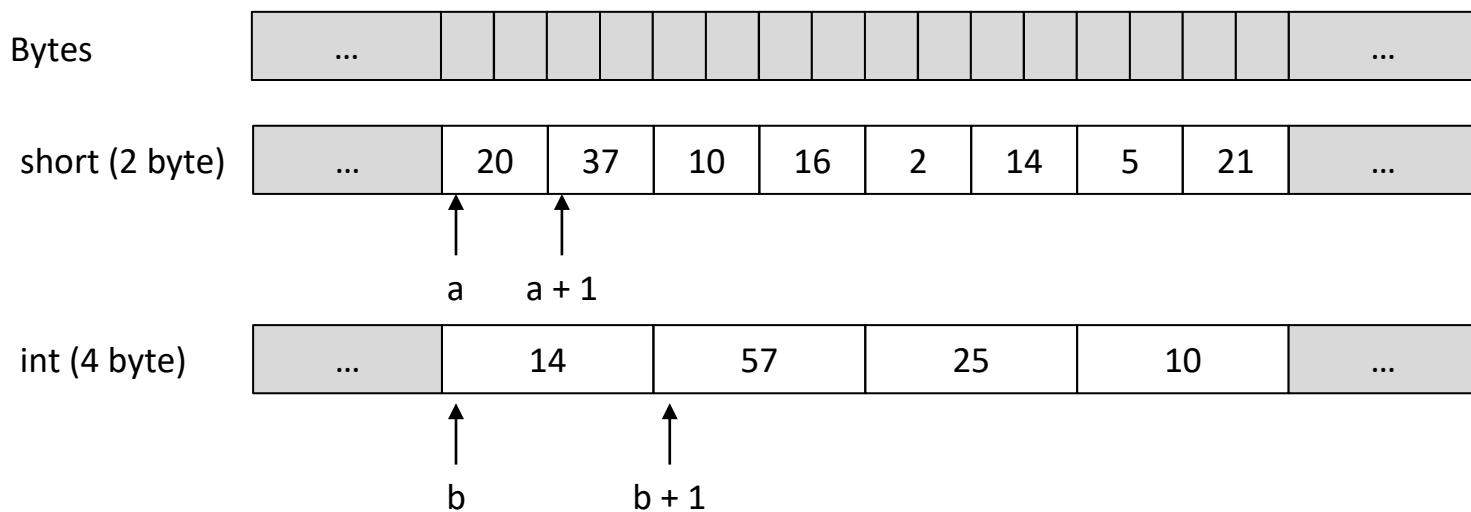


Note carefully:

- Adding 1 to a pointer does *not* increase the address by 1 byte.
- Instead, it increases by the number of bytes of its data type (e.g., 4 byte for *int*).

Example:

```
short a[] = { 20, 37, 10, 16, 2, 14, 5, 21 };  
int b[] = { 14, 57, 25, 10 };
```



- Recall that an array's name is the address of its first element.
- Cannot pass an array as a whole to a function, but a pointer to an array
- Can use pointer (*type **) or array (*type []*) type in parameter declarations

Examples:

```
void func(int *);
```

```
int main(void)
{
    int a[] = { 20, 37, 10 };
    func(a);
    getchar();
    return 0;
}
```

```
void func(int *a)
{
    // Do something ...
}
```

```
void func(int []);
```

```
int main(void)
{
    int a[] = { 20, 37, 10 };
    func(a);
    getchar();
    return 0;
}
```

```
void func(int a[])
{
    // Do something ...
}
```

Prototype

Function call

Definition



What is printed to the console?

```
int main(void)
{
    int a[] = { 20, 1, 6, 4 };
    printf("Sum of all array elements: %d\n", sum(a));
    getchar();
    return 0;
}

int sum(int *a)
{
    int sum = 0;
    int size = sizeof a / sizeof(int);
    for (int i = 0; i < size; i++)
        sum += a[i];
    return sum;
}
```



Console output:

Sum of all array elements: 20

Oops ... what happened?

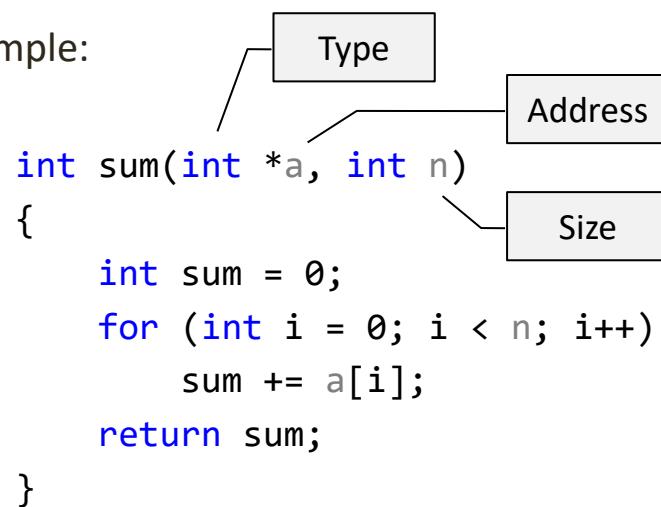
Cause:

- The argument actually is a pointer (even when using declaration *type []*).
- The *sizeof* operator gets the size of the pointer, not the array.

Solution:

- Additionally pass the size of an array to a function.

Example:





Hey, I wanted to make the function more efficient ... but what has gone wrong?!

```
int main(void)
{
    int a[] = { 20, 1, 6, 4 };
    int size = sizeof a / sizeof a[0];
    printf("Sum of all elements (1st call): %d\n", sum(a, size));
    printf("Sum of all elements (2nd call): %d\n", sum(a, size));
    getchar();
    return 0;
}

int sum(int a[], int n)
{
    for (int i = 1; i < n; i++)
        a[0] += a[i];
    return a[0];
}
```

Calling sum() twice

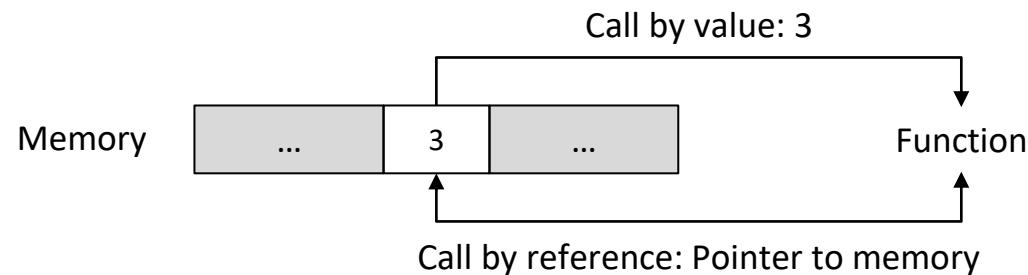
Less variables and iterations

Console output:

```
Sum of all elements (1st call): 31
Sum of all elements (2nd call): 42
```

Recall:

- Call by value: Pass value to function (\Rightarrow Function cannot modify original data)
- Call by reference: Pass variable's address to function (\Rightarrow Can modify original data)



Passing arrays to functions:

- Pass *address* of first array element to function
 - Does not create a copy of the array elements, but a pointer to the *original* array data
- \Rightarrow Function can modify array elements (on purpose or accidentally)

Constant pointers:

- Unlike other data types, keyword *const* does not prevent changing the pointer's value.
- Instead, you cannot modify data pointed to using constant pointers.

Example:

- Can assign a new value to a constant pointer
- Can read array elements using a constant pointer
- Cannot modify array elements using a constant pointer

```
int a[] = { 20, 1, 6, 4 };
int b[] = { 21, 3, 5, 2 };
const int *ptr = a;

ptr = b;
printf("*ptr + 1) = b[1] = %d", *(ptr + 1));
```

```
*(ptr + 1) = 4;
```

Does not compile

- Add keyword *const* to function parameters when the array data shall not be changed
- Function treats array *elements* (not the pointer) as though they were constant

Example:

- Cannot modify array *a* within function *sum()*

```
int sum(const int a[], int n)
{
    int sum = 0;

    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```



It is time for some *doing*:

- Write a function that determines the minimum *and* the maximum value in an array.
- Apply the function to some test data.

And think about the following:

- Note that *if*-statements are costly in terms of computing time.
- How to minimize the number of *if*-statements executed for each array element?

Sample solution (direct approach):

```
int main(void)
{
    int a[] = { 20, 1, 6, 4, 22, 1, 25, 10, 16 };
    int min, max;

    minMax(a, sizeof a / sizeof a[0], &min, &max);
    printf("Minimum: %d\n", min);
    printf("Maximum: %d\n", max);
    getchar();
    return 0;
}

void minMax(const int a[], int n, int *min, int *max)
{
    *min = *max = a[0];
    for (int i = 1; i < n; i++)
    {
        if (a[i] < *min)
            *min = a[i];
        else if (a[i] > *max)
            *max = a[i];
    }
}
```

Do not modify array

Two “return” values

Cannot be min *and* max \Rightarrow else if

Exercise: Minimum and maximum values



Explain this somewhat more sophisticated approach:

```
void minMax(const int a[], int n, int *min, int *max)
{
    // Initialize with first element
    *min = *max = a[0];

    // Run through elements in pairs
    for (int i = (n % 2); i < n - 1; i += 2)
    {
        if (a[i] < a[i + 1])
        {
            if (a[i] < *min)
                *min = a[i];
            if (a[i + 1] > *max)
                *max = a[i + 1];
        }
        else
        {
            if (a[i] > *max)
                *max = a[i];
            if (a[i + 1] < *min)
                *min = a[i + 1];
        }
    }
}
```

Number elements can be even or odd.
⇒ Start at 0 or 1 to include last element.

Only 3 if-statements for 2 elements
⇒ Reduced by ~ 25 %



Let's get familiar with the random generator:

- Create random numbers in 0 to 10.
- Use an array (of size 11) to count how often each value 0 to 10 is generated. (For instance, `a[3] == 17` means that the number 3 was generated 17 times.)
- Print the contents of the array to console.
- Calculate and print the average value of the generated numbers.

Notes:

- Function `rand()` declared in `stdlib.h` generates random integers.
- Initialize the random generator by (include `time.h`):

```
    srand((unsigned) time(NULL));
```

Sample solution:

```
#define HISTO_SIZE 11
#define NUMBERS_GENERATED 11000

int main(void)
{
    int histo[HISTO_SIZE] = { 0 };
    double average;

    // Init random generator
    srand((unsigned)time(NULL));

    // Fill histogram with random numbers and calculate average
    for (int i = 0; i < NUMBERS_GENERATED; i++)
        histo[rand() % HISTO_SIZE]++;
    average = histogramAverage(histo, HISTO_SIZE);

    // Print histogram data and average value to console
    printf("Histogram of random numbers:\n");
    for (int i = 0; i < HISTO_SIZE; i++)
        printf("%2d: %5d\n", i, histo[i]);
    printf("Average value: %.1f\n", average);

    getchar();
    return 0;
}
```

Initialize array with 0

Modulus % 11 \Rightarrow Number in 0 to 10

Exercise: Random numbers

```
double histogramAverage(const int a[], int n)
{
    int sum = 0, count = 0;

    for (int i = 0; i < n; i++)
    {
        sum += i * a[i];
        count += a[i];
    }
    return (double)sum / (double)count;
}
```

Generated a[i] numbers with value i

Console output (sample run):

Histogram of random numbers:

```
0: 1017
1: 980
2: 981
3: 997
4: 966
5: 993
6: 1015
7: 1028
8: 983
9: 1040
10: 1000
Average value: 5.0
```

Let's summarize selected pointer operations, given following array and pointers:

```
int a[] = { 20, 1, 6, 4 };
int *last = &a[3];
int *ptr;
```

Operation	Example	Equivalent to
Assignment	ptr = a;	
Dereferencing (value)	*ptr	a[0]
Address of a pointer	&ptr	
Adding integers	ptr + 3	&a[3]
Subtracting integers	last - 2;	&a[1]
Increment	ptr++;	ptr = ptr + 1;
Decrement	ptr--;	ptr = ptr - 1;

Multidimensional arrays

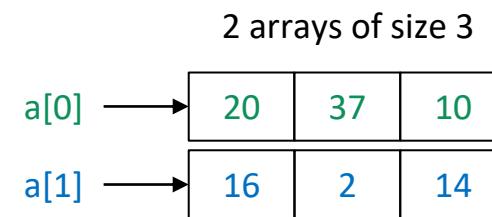
- Accessing data in matrices
- Passing matrices of arbitrary size to functions

Recall:

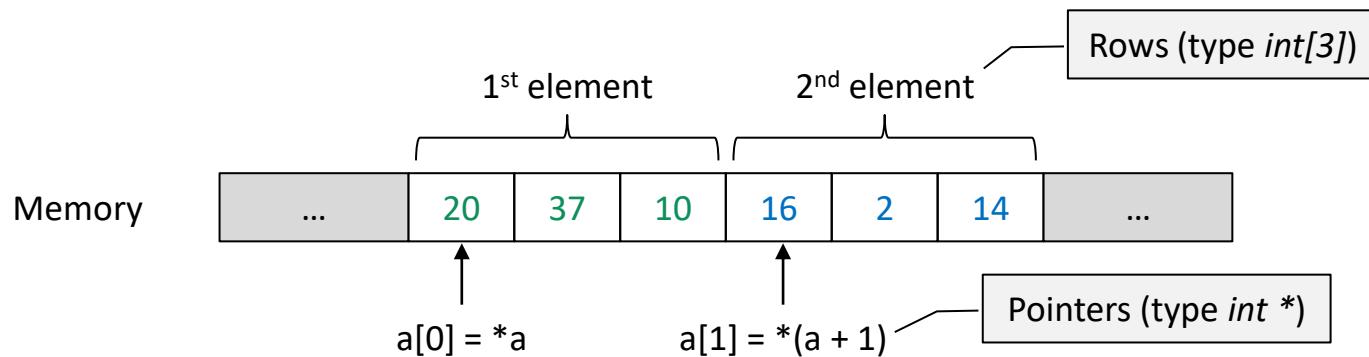
- An 2-D array (“matrix”) is an array containing arrays (“rows”) as elements.
- The name is a *pointer* to the first element (i.e., pointer to array \Rightarrow “pointer to pointer”).

Logical structure:

```
int a[2][3] = {  
    { 20, 37, 10 },  
    { 16, 2, 14 } };
```



Data in memory:

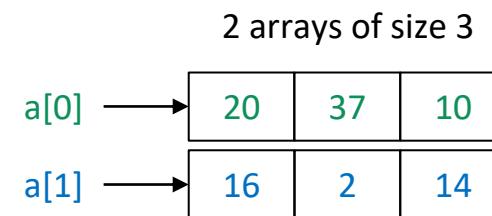


Consequences:

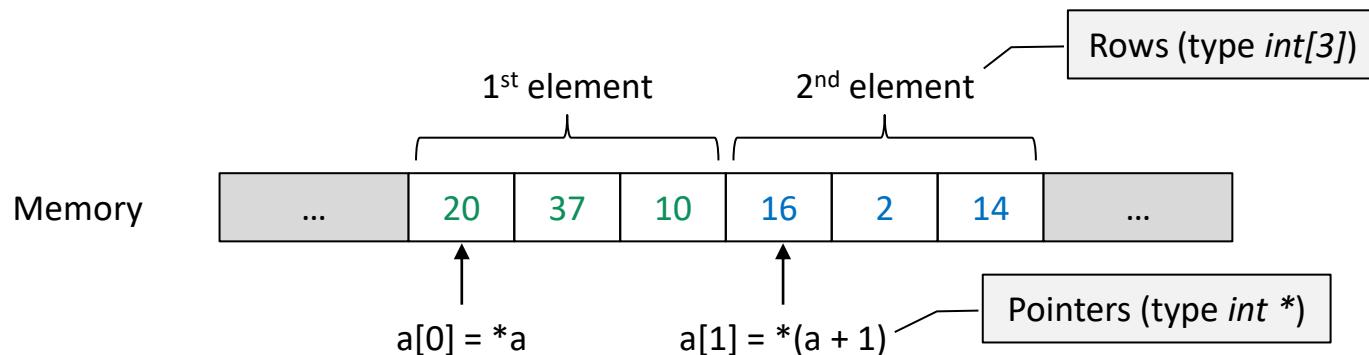
- The “value” $*a$ is the address $a[0]$ of the first row array.
- Expression $**a$ (*double indirection*) gives the value $a[0][0]$ of the first stored element.

Logical structure:

```
int a[2][3] = {  
    { 20, 37, 10 },  
    { 16, 2, 14 } };
```



Data in memory:





What is printed to the console?

```
int a[2][3] = { { 20, 37, 10 }, { 16, 2, 14 } };

printf("*a == a[0]      : %d\n", *a == a[0]);
printf(*(a + 1) == a[1] : %d\n\n", *(a + 1) == a[1]);
printf(**a      : %d\n", **a);
printf(**(a + 1) : %d\n", **(a + 1));
printf(*(*a + 2) : %d\n", *(*a + 2));
```

Console output:

```
*a == a[0]      : 1
*(a + 1) == a[1] : 1
```

a[n] point to rows (*int* arrays of size 3)
⇒ a + 1 increases pointer by 1 row

```
**a      : 20
**(a + 1) : 16
*(*a + 2) : 10
```

Yes, pointers in multidimensional arrays can be confusing, indeed ...

Passing a 2-D array to a function:

- Parameter is a pointer to an array of row arrays of a specific size (⇒ “Array of arrays”)
- Need to pass number of rows to function (like passing number of elements for 1-D array)

Sample declarations:

```
int sum(int a[][3], int rows);  
int sum(int (*a)[3], int rows);
```

Pointer a[] or *a

Object pointed to: *int* array of size 3

Note:

- Asterisk *a declares a pointer, do not confuse with dereferencing operator
- 1-D array: Pass pointer to first element (e.g., *int*)
- 2-D array: Pass pointer to first element (e.g., *int* array)



- Write a function that sums up all elements in a 2-D array.

Sample solution:

```
int main(void)
{
    int a[2][3] = { { 20, 37, 10 }, { 16, 2, 14 } };

    printf("Sum: %d\n", sum(a, 2));
    getchar();
    return 0;
}

int sum(int a[][3], int rows)
{
    int sum = 0;

    for (int row = 0; row < rows; row++)
    {
        for (int column = 0; column < 3; column++)
            sum += a[row][column];
    }
    return sum;
}
```



Think about it:

- What is a huge disadvantage of declarations as the following?

```
int sum(int a[][3], int rows);
```

Answer:

- The number of columns is *fixed* (in this case to 3).
- Need to write a dedicated function for *every* required numbers of columns.

```
int sum_Mx3(int a[][3], int rows);
int sum_Mx4(int a[][4], int rows);
int sum_Mx5(int a[][5], int rows);
```



Think again:

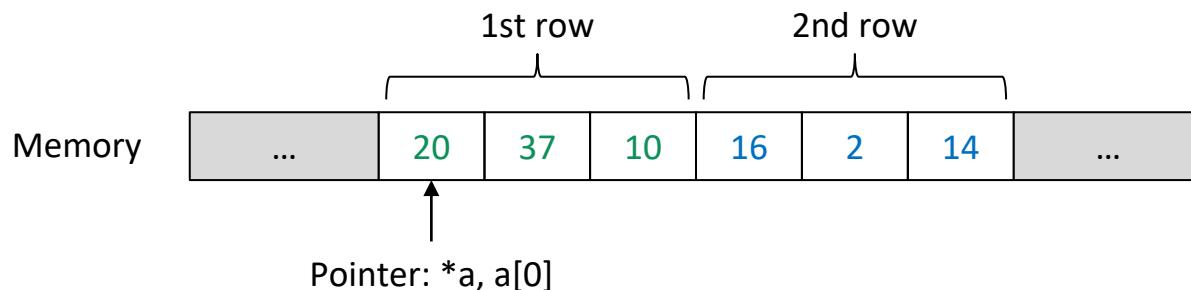
- How could we overcome this issue?

Approach:

- Elements are stored row by row from top to bottom
- Pass pointer `*a` to first element in matrix data (*not* to first row array) to function
- Pass number of rows and columns to determine matrix size (\Rightarrow number of elements)

Visualization (for 2 rows and 3 columns):

```
int a[2][3] = { { 20, 37, 10 }, { 16, 2, 14 } };
```



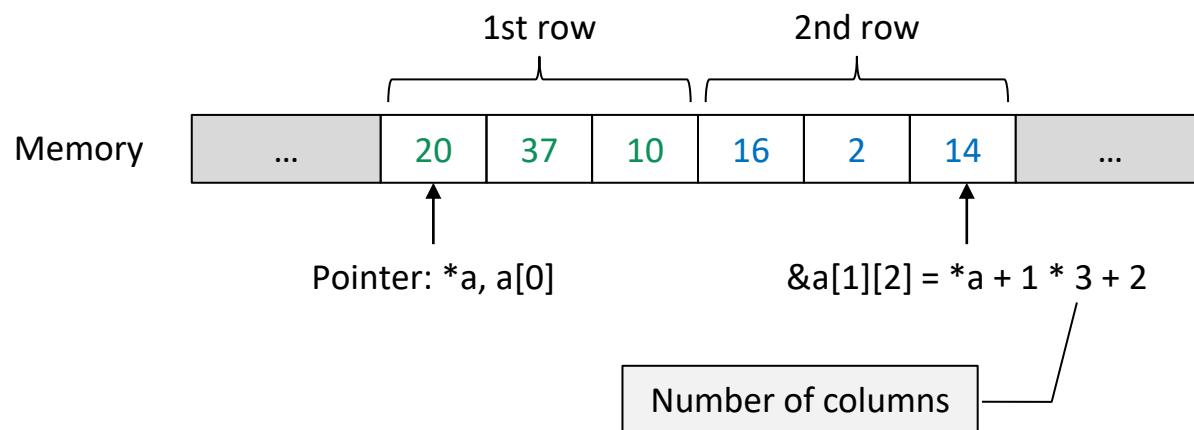


Think about it:

- How to access a specific element $a[y][x]$?
- Given: Pointer $*a$ to first value, number rows, and number columns

Solution:

- Address of y -th row : $*a + y * \text{columns}$
- Address of element $a[y][x]$: $*a + y * \text{columns} + x$



Exercise: Sum of matrix elements



- Write a function that sums up all elements in a 2-D array of variable size.

Sample solution:

```
int main(void)
{
    int a[2][3] = { { 20, 37, 10 }, { 16, 2, 14 } };

    printf("Sum: %d\n", sum(*a, 2, 3));
    getchar();
    return 0;
}

int sum(int *data, int rows, int columns)
{
    int sum = 0;

    for (int i = 0; i < rows * columns; i++)
        sum += data[i];
    return sum;
}
```

Diagram annotations:

- A callout box labeled "Address of first element" points to the expression `*a` in the `printf` statement.
- A callout box labeled "Pointer to 1-D array" points to the parameter `data` in the `sum` function declaration.
- A callout box labeled "Number of elements" points to the parameter `rows * columns` in the `for` loop.



- Write a function that prints a matrix of variable size to the console.

Sample solution:

```
int main(void)
{
    int a[2][3] = { { 20, 37, 10 }, { 16, 2, 14 } };

    printMatrix(*a, 2, 3);
    getchar();
    return 0;
}

void printMatrix(int *data, int rows, int columns)
{
    for (int y = 0; y < rows; y++)
    {
        for (int x = 0; x < columns; x++)
            printf("%2d ", data[y * columns + x]);
        putchar('\n');
    }
}
```

Strings

- Reading strings from the keyboard safely
- Writing strings to the console
- Selected string functions

Functions for reading strings from the keyboard:

Name	Functionality	Notes
scanf()	Reads until next whitespace (excluded)	Potential buffer overflow
gets()	Reads until next '\n'	Discards '\n' Potential buffer overflow \Rightarrow Removed in ISO C11
fgets()	Reads until next '\n', but at most n characters	Stores '\n' in string

Notes:

- All functions are declared in *stdio.h*.
- As discussed before, *scanf()* reads “words”, not lines, until the next ‘\n’.
- Function *gets()* was commonly used, but is unsafe. \Rightarrow Do not use it!
- Although *fgets()* is intended for reading files, it is a safe alternative to *gets()*.



Recall:

- Function `scanf()` reads characters until the next whitespace from keyboard
 - Stores the user input at memory location passed as argument (type `char *`)
- ⇒ Must allocate enough space to hold the user input

Things can go terribly wrong:

- Target `char` array not large enough for input (⇒ *buffer overflow*)
- Pointer to memory location not initialized

```
char *name;           // Can point anywhere in memory!
scanf("%s", name);
```

Consequences:

- Might accidentally overwrite data such as other variables
- Might accidentally overwrite program code using uninitialized pointer
- Might cause the program to crash



Function `gets()`:

- Does not check, if the string fits into the target array
- User input too long \Rightarrow Buffer overflow

```
char name[16];
gets(name);
```

Better *not* enter “Neil A. Armstrong”

How to prevent buffer overflow:

- For `gets()` the programmer cannot \Rightarrow Function removed from the C standard (ISO C11)
- For `scanf()` specify the maximum number of characters to read, e.g.:

```
char name[16];
scanf("%15s", name);
```

Max. 15 characters + '\0'

String input using fgets()

Prototype:

```
char *fgets(char *str, int n, FILE *stream);
```

Returns *str* on success, else *NULL*

Use *stdin* to read from keyboard

Max. *n* characters (incl. '\n' and '\0')

Example:

```
#define SIZE 16

int main(void)
{
    char name[SIZE];

    printf("Please enter your name: ");
    if (fgets(name, SIZE, stdin) != NULL)
        printf("Your input: %s\n", name);

    getchar();
    return 0;
}
```

Corresponding functions for writing strings to the console:

Name	Functionality
printf()	Prints formatted string using specifiers (e.g., %d)
puts()	Prints until next '\0' and adds '\n'
fputs()	Prints until next '\0', but does not add '\n'

Notes:

- All functions are declared in *stdio.h*.
- Pass file pointer *stdout* to *fputs()* to write to the console.

Example:

```
char name[64] = "Mark Watney";  
  
printf("%s\n", name);  
puts(name);  
fputs(name, stdout);
```

Selected string functions declared in *string.h*:

Name	Functionality
strlen()	Length of a string (not including '\0')
strcat()	String concatenation. Does not create a new string, but appends. Potential buffer overflow.
strncat()	String concatenation. Same as <i>strcat()</i> , but limit number of appended characters to prevent overflow.
strcpy()	String copy. Potential buffer overflow.
strncpy()	String copy. Same as <i>strcpy()</i> , but limit number of copied characters to prevent overflow.
strcmp()	String comparison.

Function *sprintf()* declared in *stdio.h*:

- It is basically the same as *printf()*, but writes to a string instead of the console.
- Can be used to convert a number to a string (e.g., 21 to "21").

What is printed to the console?



```
char info[64];
char name[] = "Mark Watney";
int sol = 549;

sprintf(info, "%s left Mars on Sol %d.", name, sol);
puts(info);
```

Target string

Console output:

Mark Watney left Mars on Sol 549.

String-to-number conversions

Selected conversion functions declared in *stdlib.h*:

Name	Functionality
atoi()	Convert string (alphanumeric) to integer type <i>int</i>
atol()	Convert string (alphanumeric) to integer type <i>long</i>
atof()	Convert string (alphanumeric) to floating point type <i>double</i>

What is printed to the console?

```
int hour = atoi("6");
int minutes = atoi("50 minutes");
printf("Time: %02d:%02d\n", hour, minutes);
```

Reads digits from left,
ignores the rest

Console output:

Time: 06:50



- Write a function *stringLength()* returning the length of a string.

Sample solution:

```
int main(void)
{
    char city[64] = "Hamburg";
    printf("Array size      : %d\n", sizeof city / sizeof city[0]);
    printf("String          : %s\n", city);
    printf("String length   : %d (excluding '\\0')\n", stringLength(city));
    getchar();
    return 0;
}

int stringLength(const char *string)
{
    int i = 0;
    while (string[i] != '\0')
        i++;
    return i;
}
```

What if *string* does not contain '\0'?

Software Construction 1 (IE1-SO1)

7. Memory management



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

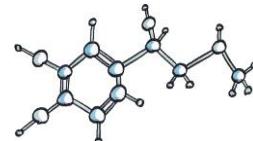
Advanced topics



5. Pointers



6. Memory management

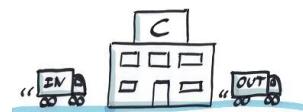


7. Structures



8. Lists and sorting

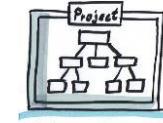
The next steps ...



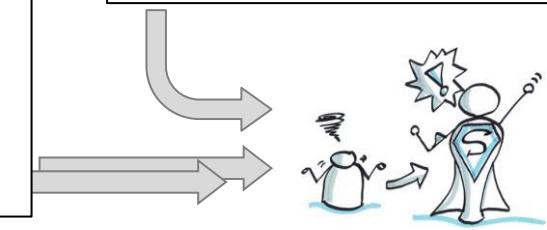
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *use* variables and functions defined in a file in another file in order to better structure your programs.
- You *dynamically create* 1-D and 2-D arrays, whose sizes are not known at compile time, in order to be flexible with respect to the data required in a running program.
- You *dynamically reserve and free* memory as required, in particular memory allocated in a function that is still accessible after the function has ended.



1. Storage classes
2. Dynamic memory allocation
3. Multidimensional arrays

Storage classes

- Where within the source code is a variable known?
- Can variables and functions be used in other files as well?
- How long does a variable exist?

Again, let's distinguish:

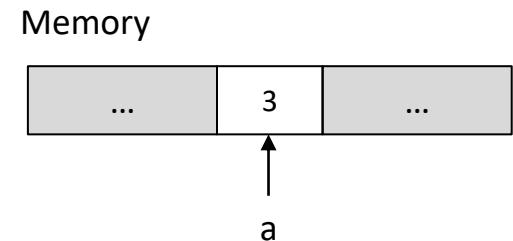
- Memory reserved to store data (e.g., *int* value)
- Identifier (“variable”) to access the data in memory

Source code region:

- Which parts of the source code can use a variable?
- *Scope*: Within, e.g., the complete source file or in a specific function, only?
- *Linkage*: Can other source files use a variable?

Time:

- *Storage duration*: How long does a variable exist in memory?
- As long as the program runs? Within a function, only?



Most important to understand (though there is more):

- Block scope
- File scope

Block scope (*local variables*):

- Recall: A *block* is a section of source code within braces { ... } (e.g., function body).
- Variables known from their declaration until end of the block containing the declaration
- ISO C99 includes loop- and *if/else*-statements, even if these are not within braces.
- Block scope variables are not automatically initialized.

File scope (*global variables*):

- Variables declared *outside* of a function
- These are visible from its declaration to the end of the file.
- File scope variables are automatically initialized to zero.

Example: Block scope

- Variable *sum* is known until the closing brace of the function body.
- Variable *limit* is known until the closing brace of its block.
- Variable *i* is known within the *for*-loop, only (ISO C99).

```
int main(void)
{
    int sum = 0;

    {
        const int limit = 100;
        for (int i = 1; i < limit; i++)
            sum += i;

        printf("Sum 1 to %d: ", limit);
    }
    printf("%d\n", sum);

    getchar();
    return 0;
}
```

The diagram illustrates the scope of variables in the provided C code. The entire code area is highlighted in light green. The innermost block, containing the *for*-loop, is highlighted in light blue. The variable *sum* is shown in a light green box labeled "Scope of sum". The variable *limit* is shown in a light blue box labeled "Scope of limit". The variable *i* is shown in a light blue box labeled "Scope of i".

Example: File scope



Will the following code compile?

```
void knowWho(void);           Function cannot access who
void dontKnowWho(void)
{
}
char *who = "It's me!";       Definition of who

int main(void)                Function can access who
{
    printf("I know who: %s\n", who);
    dontKnowWho();
    knowWho();
    getchar();
    return 0;
}

void knowWho(void)            Function can access who
{
    printf("Me too      : %s\n", who);
}
```

Distinguish for file scope variables:

- *Internal linkage*: Variable is known within one file, only
- *External linkage*: Variable can be accessed in other files of the same program

Syntax:

- By default file scope variables have external linkage (*external variables*).
- Indicate internal linkage by the keyword *static*

Example:

```
int forAnyone = 3;
static int keptPrivate = 5;

int main(void)
{
    // ...

    getchar();
    return 0;
}
```

The diagram illustrates the scope of variables in the provided C code. The variable `forAnyone` is associated with a box labeled "Accessible in other files", indicating it has external linkage. The variable `keptPrivate` and the function `main` are both associated with a box labeled "Known within this file, only", indicating they have internal linkage.

Persistence of a variable in memory:

- When is memory reserved for a variable (\Rightarrow variable starts to exist)?
- When is the memory freed (\Rightarrow variable does not exist anymore)?

Storage duration depending on scope:

- File scope: Exist in memory throughout the program (*static* duration)
- Block scope: Memory is freed at end of the block (*automatic* duration)

Example: Storage duration



Comment on the storage duration of the variables:

```
int max = 0;                                Variable exists throughout program
int main(void)
{
    printf("Enter positive integer numbers (q to quit):\n");
    while (1)
    {
        int input;                            New variable created in each iteration
        if (scanf("%d", &input) == 1)
            max = (max > input) ? max : input;
        else
            break;
    }

    while (getchar() != '\n')
        continue;

    printf("Maximum number entered: %d\n", max);
    getchar();
    return 0;
}
```

Examples



What is the scope, linkage, and storage duration of the variables x, y, and z?

```
int x = 1;  
static int y = 2;  
  
int main(void)  
{  
    int z = 3;  
    getchar();  
    return 0;  
}
```

- File scope (known until end of file)
- External linkage (accessible in other files)
- Static duration (exists as long as the program runs)

- File scope (known until end of file)
- Internal linkage (not accessible in other files)
- Static duration (exists as long as the program runs)

- Block scope (known until end of *main()* function)
- No linkage (not accessible in other files)
- Automatic duration (ends when the function returns)

The same as for global variables:

- By default functions are external (i.e., can be used in other files).
- Keyword *static* restricts a function to the file containing the definition

```
void canCallMe(void);           // Can be used in other files
static void doNotKnowMe(void);   // Restricted to this file
```

External declarations:

- External variables and functions can be used in other files.
- Must declare (using keyword *extern*) that a variable or function is defined in another file

```
extern void canCallMe(void);    // Declare function defined in another file
```

Note on variable and function names:

- Compilers must distinguish first 63 (internal) or 31 (external) characters since ISO C99.
- Compilers must distinguish first 31 (internal) or 6 (external) characters prior to ISO C99.

Example: Cartesian to polar coordinates

File *polarCoords.c*:

- Defines global variable for constant π
- Defines function to transform Cartesian to polar coordinates
- The source file does *not* contain a *main()* function.

```
#include <math.h>
```

External variable (global, without *static*)

```
/* External variable: Math constant pi */
const double PI = 3.14159265358979323846;
```

```
/* External function: Transform 2-D Cartesian to polar coordinates */
```

```
void toPolarCoords(double x, double y, double *r, double *phi)
```

```
{
```

```
    *r = sqrt(x * x + y * y);
    *phi = atan2(y, x);
```

```
}
```

External function (without *static*)

Example: Cartesian to polar coordinates

File *main.c*:

- Declares and uses external variable and function from file *polarCoords.c*

```
#include <stdio.h>
extern const double PI;
extern void toPolarCoords(double x, double y, double *r, double *phi);

int main(void)
{
    double x = 2.0, y = 3.75;      // Input: Cartesian coordinates
    double r, phi;                // Result: Polar coordinates

    toPolarCoords(x, y, &r, &phi);
    printf("Cartesian: (x, y) = (%.2f, %.2f)\n", x, y);
    printf("Polar     : (r, phi) = (%.2f, %.2f * pi)\n", r, phi / PI);

    getchar();
    return 0;
}
```

Declare external variable and function

Use external variable and function

Recommendations



What do you think of the following?

- Global variables make function parameters and return values obsolete.
- If all variables were global, every function could assess them directly.

Clear answer:

- No, do not do that!
- Else source codes tend to be harder to understand, more difficult to maintain, and error-prone (e.g., side effects)

Apply the “need to know rule”:

- Let files and functions access data they need to know to “do their job”, only.
- Nothing more ... nothing, really.

Dynamic memory allocation

- Get memory (e. g., for an array) when it is needed
- Give memory back to the system when it is appropriate

Recall these storage durations:

- *Static data*: Allocate memory when program is loaded and free it when program ends
- *Automatic data*: Allocate memory in block and free it when block ends

But this is not possible so far ...

- Allocate an array where the size is *not known* at compile time
- Allocate memory in a function (i.e., a block) and *return* it to the calling function

Dynamic memory allocation:

- There exist functions to allocate and free memory (declared in *stdlib.h*).
- Gives flexibility to allocate and free memory as required
- Storage duration: *Allocated data* exists from allocation until it is explicitly freed

- Use the function *malloc()* to allocate a specific number of bytes in memory.
- Use the *sizeof* operator to determine the number of bytes for a data type.
- The function returns a pointer to the first byte (or *NULL*, if allocation failed).

Example:

```
int *ptr = (int *)malloc(sizeof(int));  
int *a = (int *)malloc(2 * sizeof(int));
```

New memory for *int* value

New memory for *int* array of size 2

Notes:

- The *malloc()* function does not assign a variable name, but allocates memory, only.
- The type cast is optional in C, but required in C++.

Memory allocation using `calloc()`

- The function `calloc()` allocates memory and sets all *bits* to zero.
- Provide the number of elements and the size (in bytes) of elements as arguments.

Example:

```
int *a = (int *)malloc(2 * sizeof(int));
int *b = (int *)calloc(2, sizeof(int));
```

Number

Overall number of bytes in `malloc()`

Size

```
if ((a == NULL) || (b == NULL))
    exit(EXIT_FAILURE);
```

Terminate program when out of memory

Notes:

- Can end a program by calling `exit()` when memory allocation fails (“out of memory”)
- The `exit()` function and constant `EXIT_FAILURE` are declared in `stdlib.h`.

- Unlike static or automatic data you must explicitly free the memory.
- Call function *free()* with the pointer to the allocated memory
- Make sure the pointer actually points to allocated memory.

Example:

```
int *ptr = (int *)malloc(2 * sizeof(int));  
  
if (ptr != NULL)  
    free(ptr);
```

Important notes:

- Memory not freed (\Rightarrow not reusable) is called *memory leak*.
 - Typically allocated memory is freed automatically when a program ends.
 - But an operating system might *not* guarantee to automatically free dynamic memory.
- \Rightarrow Always, always, always (Did I mention *always*?) free allocated memory!

Example: Dynamic memory allocation



What is printed to the console?

```
int *a = (int *)malloc(2 * sizeof(int));  
int *b = (int *)calloc(2, sizeof(int));
```

Does not initialize

Initializes with zero

```
/* Verify memory has successfully been allocated */  
if (a == NULL) || (b == NULL))  
    exit(EXIT_FAILURE);
```

```
/* Modify and/or print allocated data */  
a[0] = 9;  
printf("a = (%d, %d)\n", *a, *(a + 1));  
printf("b = (%d, %d)\n", b[0], b[1]);
```

a[1] not initialized

```
/* Free allocated memory */  
free(a);  
free(b);
```

Console output (sample run):

```
a = (9, -842150451)  
b = (0, 0)
```

Exercise: String functions



Write your own string functions in a file *myStrings.c*:

- Create and return a copy (clone) of a string
- Create a string being the concatenation of two other strings

Notes:

- Both functions shall allocate and return the memory for the new string.
- Both functions shall protect the input strings from accidental modification.

Exercise: String functions

Create copy (clone) in file *myStrings.c*:

```
int getStringLength(const char *string)
{
    int length = 0;
    while (string[length++] != '\0')
        continue;
    return length;
}

char *newStringClone(const char *string)
{
    int length = getStringLength(string);
    char *clone = (char *)malloc(length * sizeof(char));

    // Copy string contents
    if (clone != NULL)
    {
        for (int i = 0; i < length; i++)
            clone[i] = string[i];
    }
    return clone;
}
```

Helper function

Keyword *const* ⇒ Cannot modify input string

Name with “new” indicates memory allocation
⇒ Need to free memory later on

Create concatenation in file *myStrings.c*:

```
char *newStringConcat(const char *string1, const char *string2)
{
    char *concat;
    int length1 = getStringLength(string1);
    int length2 = getStringLength(string2);
    int i;

    // Allocate memory (Exclude '\0' of first string => - 1)
    concat = (char *)malloc((length1 + length2 - 1) * sizeof(char));

    // Copy contents of both strings (excluding first '\0')
    if (concat != NULL)
    {
        for (i = 0; i < length1 - 1; i++)      // Exclude '\0' => - 1
            concat[i] = string1[i];
        for (int j = 0; j < length2; i++, j++)
            concat[i] = string2[j];
    }
    return concat;
}
```



Increment both index variables

Exercise: String functions

Using the string functions in file *main.c*:

```
extern char *newStringClone(const char *);  
extern char *newStringConcat(const char *, const char *);  
  
int main(void)  
{  
    const char *dolly = "Dolly",  
    const char *sheep = " the sheep";  
    char *concat = newStringConcat(dolly, sheep);  
    char *clone = newStringClone(concat);  
  
    printf("1st string : %s\n 2nd string : %s\n ", dolly, sheep);  
    if (concat != NULL)  
    {  
        printf("Concatenated: %s\n", concat);  
        free(concat);  
    }  
    if (clone != NULL)  
    {  
        printf("Cloned      : %s\n", clone);  
        free(clone);  
    }  
    getchar();  
    return 0;  
}
```

Declare external functions

Functions allocate memory

Do not forget to free memory!

Finally, be aware that a program organizes its memory in four sections (*segments*), the code segment holding the program code and following three segments for variables.

1. Static memory:

- Global variables and string literals are known at compile time (*static duration*).
 - Memory available from loading program until termination
- ⇒ Memory allocation stays as it is for the complete program

2. Stack:

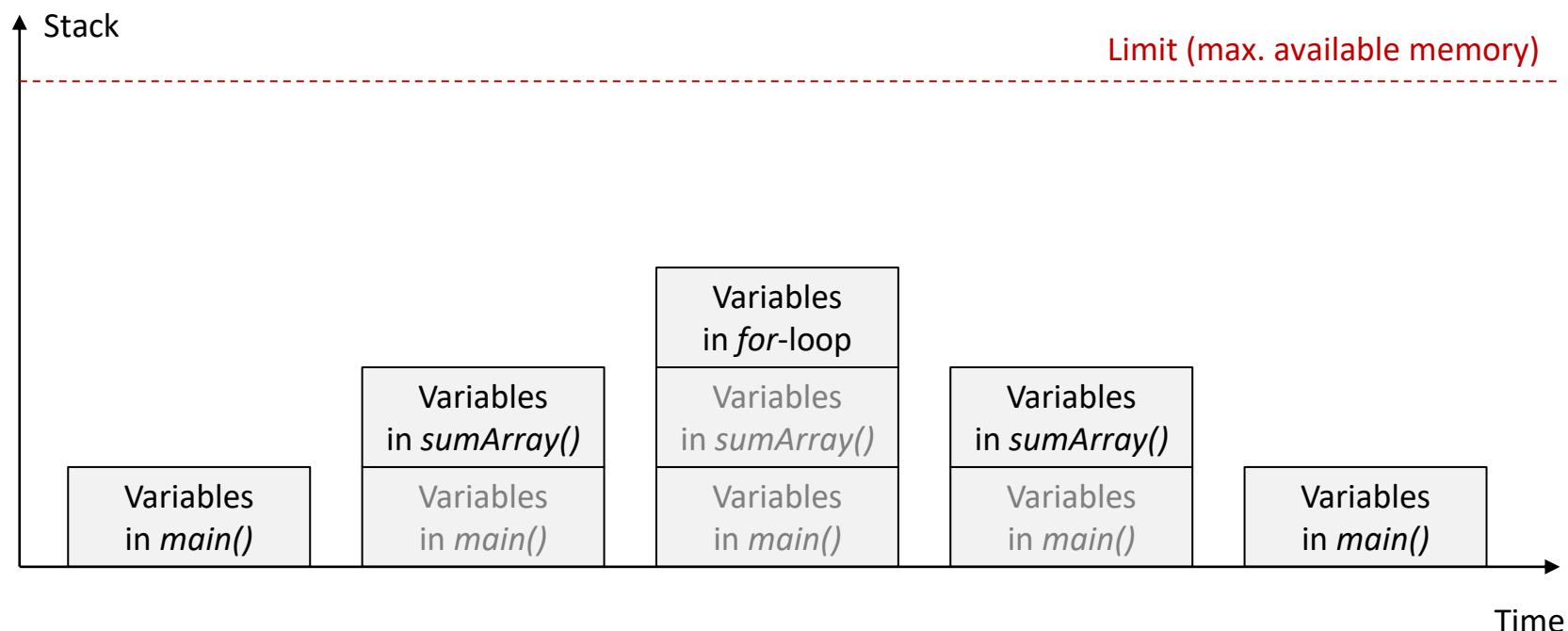
- Block scope variables are allocated when entering the block (*automatic duration*).
 - Memory of all variables of a block is freed at the end of the block.
- ⇒ The *stack* automatically grows (or shrinks) when entering (or leaving) a block.

3. Heap:

- Programmers explicitly allocate and free dynamically allocated memory.
- ⇒ The memory *heap* can end up fragmented.

Visualization of memory stack:

- Stack grows when entering a block (e.g., function)
- Stack shrinks in reverse order when leaving a block



Multidimensional arrays

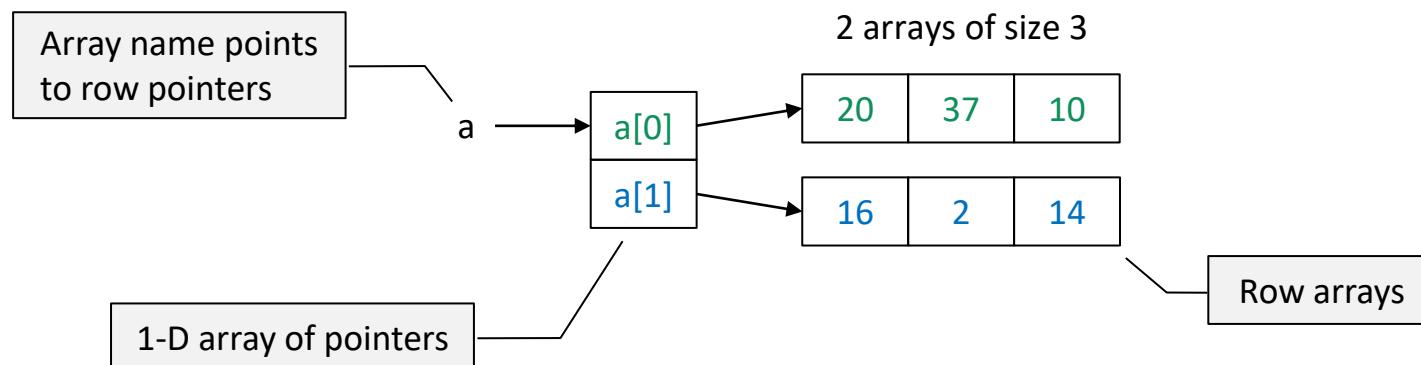
- Allocate memory for matrices
- Allocate and initialize row pointers to allow syntax $matrix[y][x]$

First approach (allocate memory for data, only):

- Allocate 1-D array for 2-D data (with elements stored row by row)
 - Disadvantage: Does not allow access using syntax $a[y][x]$
- ⇒ Let's rather follow the approach below!

Additionally allocate memory for row pointers:

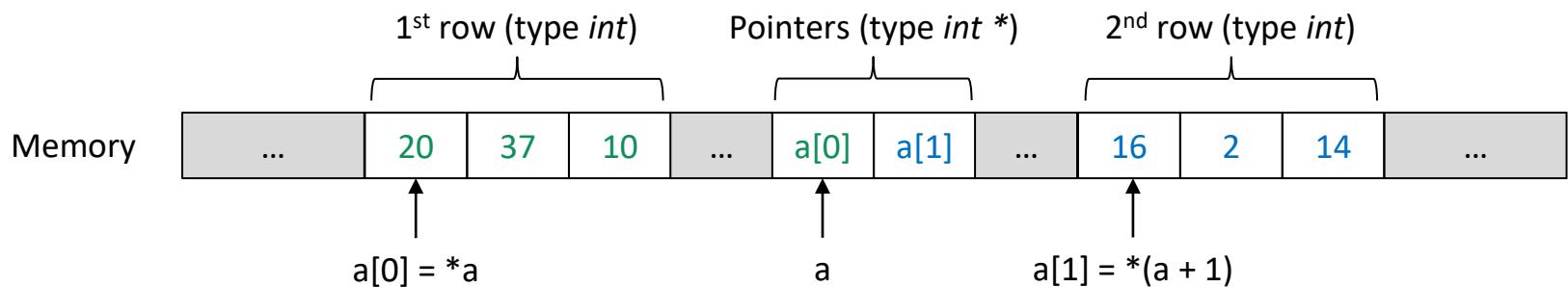
- Allocate memory for the 2-D data
- Allocate and initialize an 1-D array containing pointers to the data rows



Memory allocation for 2-D arrays (row-by-row)

Allocate memory for rows independently:

1. Allocate memory for row pointers
2. Allocate memory for each row and assign it to the row pointers



Advantages:

- Rows can have different number of elements (e.g., triangular 2-D array)
- Works, even if no single area large enough for complete matrix is available in memory

Disadvantages:

- Must free all rows and pointer array independently (else: memory leaks)
- Potentially row data is distributed in memory (\Rightarrow Might increase time to access data)

Example:

```
int **a;
int rows = 3, columns = 4;

/* Allocate memory for row pointers */
if ((a = (int **)malloc(rows * sizeof(int *))) == NULL)
    exit(EXIT_FAILURE);

/* Allocate memory for rows */
for (int y = 0; y < rows; y++)
{
    if ((a[y] = (int *)malloc(columns * sizeof(int))) == NULL)
        exit(EXIT_FAILURE);
}

// Do something ...

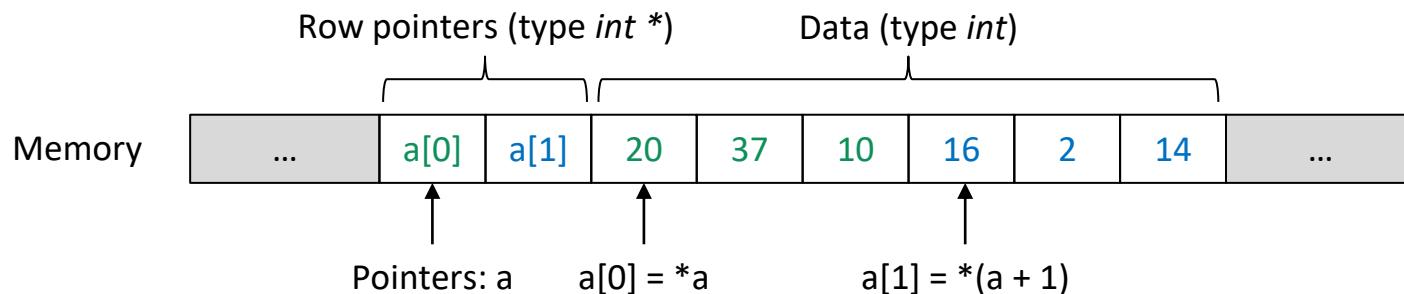
/* Free matrix memory */
for (int y = 0; y < rows; y++)
    free(a[y]);
free(a);
```

Invoke *malloc()* for each row

Invoke *free()* for each row

Allocate memory calling *malloc()* only once:

- Allocate memory for row pointers and rows with one function call
- Initialize pointers to point to rows



Advantages:

- Can free allocated memory calling *free()* only once
- Allocates a connected area in memory (as for *int a[M][N];*)

Now it is your turn:

- Modify the last example accordingly.



Example:

```
int rows = 3, columns = 4;
int **a;                                // Pointer to matrix (i.e., to row pointers)
int *data;                               // Pointer to data in matrix
int pointerBytes, rowBytes;             // Required memory in bytes

/* Allocate memory for matrix */
pointerBytes = rows * sizeof(int *);
rowBytes = columns * sizeof(int);
if ((a = (int **)malloc(pointerBytes + rows * rowBytes)) == NULL)
    exit(EXIT_FAILURE);

/* Initialize pointers to rows */
data = (int *)(a + rows);
for (int y = 0; y < rows; y++)
    a[y] = data + y * columns;

// Do something ...

/* Free matrix memory */
free(a);
```

Invoke *malloc()* only once

Initialize row pointers

Invoke *free()* only once

Software Construction 1 (IE1-SO1)

8. Structures



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

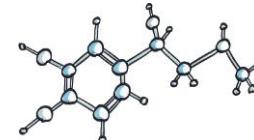
Advanced topics



5. Pointers



6. Memory management

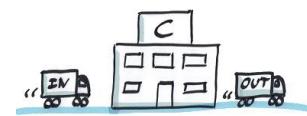


7. Structures



8. Lists and sorting

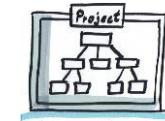
The next steps ...



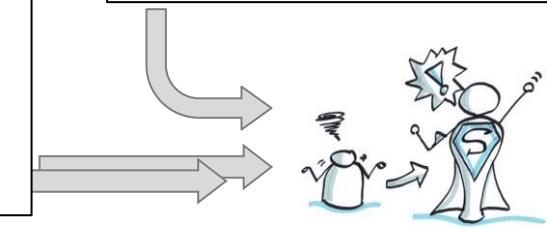
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *group data* logically belonging together in order to better structure your programs.
- You *use symbolic names* for integer constants in order to make your source code easier to understand and maintain.
- You *declare alias names* for existing data types in order to increase the readability of your source code and/or increase portability to other platforms.



1. Structures
 - Fundamentals
 - Arrays & pointers
 - Assignments & functions
2. Enumerations
3. Type definitions

Structures

- Group variables of same or different data types
- Access variables by individual names, not index

- Set of variables can belong together (e.g., vector components, first name and surname)
- Syntax to group data in a *structure*:

```
struct name {  
    // Variable declarations (members, fields)  
};
```

Terms:

- Structure identifier *name* is also called *tag*.
- Variables inside a structure are also called *members* or *fields*.

Notes:

- Structure declarations end with a semicolon.
- Declaration describes how a structure is build up (template or “blueprint”).
- Declaration does *not* allocate memory for variables.

Example:

- Declare a structure representing a calendar date.
- Dates consist of the day of the month, month, year, and a string containing the month.

```
#define MONTH_CHARS 10
                                ↗
                                Longest: "September" (incl. '\0')

int main(void)
{
    struct date {
        int dayOfMonth;
        int month;
        int year;
        char monthName[MONTH_CHARS];
    };

    // ...
}
```

The code defines a structure named 'date' with four members: 'dayOfMonth', 'month', 'year', and 'monthName'. The 'monthName' member is declared as an array of characters with a size of 'MONTH_CHARS', which is defined as 10. A callout box points to this declaration with the text 'Longest: "September" (incl. '\0')'. Another callout box points to the 'struct' keyword with the text 'Tag'.

- A *structure variable* has memory allocated for each member.

Declaring structure variables together with the structure declaration:

```
struct date {  
    int dayOfMonth;  
    int month;  
    int year;  
    char monthName[MONTH_CHARS];  
} birthAlisa;
```

Allowed to omit tag
date in this case

Structure variable

Declaring structure variables after the structure declaration:

```
struct date birthAlisa, birthSarah;
```

- Declares two variables of type *struct date*
- Each variable has memory allocated for three *int* values and a *char* array.

Accessing member variables

- Access member variables using the *member(ship) operator* (dot ‘.’)

```
structureVariable.member
```



Now it is your turn:

- Given are the *struct date* and a structure variable *birthAlisa*.
- Initialize Alisa’s birthday (choose any date) and print it to the console.

Sample solution:

```
/* Declare structure */
struct date {
    int dayOfMonth;
    int month;
    int year;
    char monthName[MONTH_CHARS];
};

/* Define structure variable */
struct date birthAlisa;

/* Initialize member variables */
birthAlisa.dayOfMonth = 16;
birthAlisa.month = 7;
birthAlisa.year = 1991;
strncpy(birthAlisa.monthName, "July", MONTH_CHARS);

/* Print structure data to the console */
printf("Alisa's birthday is in %s (%02d.%02d.%04d).\n",
       birthAlisa.monthName, birthAlisa.dayOfMonth, birthAlisa.month, birthAlisa.year);
```

Assign using member operator

Read using member operator

Analogous to arrays definitions:

- You can initialize the values using a comma-separated list in braces.

```
struct date {  
    int dayOfMonth;  
    int month;  
    int year;  
    char monthName[MONTH_CHARS];  
};
```

Initialize member variables

```
struct date birthAlisa = { 16, 7, 1991, "July" };  
struct date birthSarah = { 7, 9, 1992, "September" };
```

Notes:

- For arrays all elements in the list have the same data type.
- For structures the elements have data types matching the member variables.
- Programmers (\Rightarrow you!) must ensure the data types match.



What is printed to the console?

```
struct date {
    int dayOfMonth;
    int month;
    int year;
    char monthName[MONTH_CHARS];
};

struct date birthAlisa;
struct date birthSarah = { 7, 9, 1992, "September" };

birthAlisa.dayOfMonth = 16;
birthAlisa.month = 7;
birthAlisa.year = 1991;

printf("Alisa's birthday is in %-9s (%02d.%02d.%04d).\n",
    birthAlisa.monthName, birthAlisa.dayOfMonth, birthAlisa.month, birthAlisa.year);
printf("Sarah's birthday is in %-9s (%02d.%02d.%04d).\n",
    birthSarah.monthName, birthSarah.dayOfMonth, birthSarah.month, birthSarah.year);
```

Print ≥ 9 characters, aligned left

Console output (sample run):

Alisa's birthday is in ፲፻፲፭ (16.07.1991).
Sarah's birthday is in September (07.09.1992).

`int ≥ 9 characters, aligned left`

Oops ... *char* array not initialized (no '\0')

- 
- Structures can contain structures (... which can contain structures, which can contain ...)

Let's clarify by following exercise:

- Write a structure *name* containing the first name and the surname of a person.
- Write a structure *birthday* containing the name and the birthday of a person.

Notes on *birthday*:

- The structure shall contain the day, month, and year of the birthday.
- The structure shall contain the structure *name*.

Sample solution:

```
/* Declare structures */
struct name {
    char firstName[NAME_CHARS];
    char surname[NAME_CHARS];
};
```

NAME_CHARS must be defined outside of *main()*

```
struct birthday {
    struct name name;
    int dayOfMonth;
    int month;
    int year;
};
```

Nested structure

```
/* Define and initialize structure variable */
struct birthday birthAlisa = { { "Alisa", "Erhardt" }, 16, 7, 1991 };
```

Initialize nested structure *name*

```
/* Print structure data to the console */
printf("%s %s's birthday is on ", birthAlisa.name.firstName, birthAlisa.name.surname);
printf("%02d.%02d.%04d.\n", birthAlisa.dayOfMonth, birthAlisa.month, birthAlisa.year);
```

Access nested members

Console output:

Alisa Erhardt's birthday is on 16.07.1991.

Arrays & pointers

- Arrays of structure variables
- Pointers to structure variables

- You can create arrays of structure variables (just the same as for other data types).

Example:

```
/* Declare structure */
struct name {
    char firstName[NAME_CHARS];
    char surname[NAME_CHARS];
};

/* Define and initialize array of structure variables */
struct name people[SIZE] = {
    { "Alisa", "Erhardt" },
    { "Sarah", "Heitmann" },
    { "Tobias", "Neuner" }
};

/* Print structure data to the console */
for (int i = 0; i < SIZE; i++)
    printf("%s %s\n", people[i].firstName, people[i].surname);
```

The code illustrates the declaration, initialization, and access of an array of structure variables. The first part defines a structure named 'name' with two character arrays for first and last names. The second part declares an array 'people' of size 'SIZE' and initializes it with three elements containing names. The third part prints each element of the array to the console.

- The name of a structure variable is *not* the address of the structure variable.
- You can declare pointers to structure variables.
- Access members using the *indirect member(ship) operator*:

```
structPointer->member
```

Example:

```
struct name {  
    char firstName[NAME_CHARS];  
    char surname[NAME_CHARS];  
};  
  
struct name alisa = { "Alisa", "Erhardt" };  
struct name *ptr = &alisa;  
  
printf("ptr->firstName : %s\n", ptr->firstName);  
printf("(ptr).firstName : %s\n", (*ptr).firstName);
```

Variable name ≠ address
⇒ Requires address operator &



What is printed to the console?

```
struct name {  
    char firstName[NAME_CHARS];  
    char surname[NAME_CHARS];  
};  
  
struct name people[SIZE] = { { "Alisa", "Erhardt" }, { "Sarah", "Heitmann" } };  
struct name *ptr = &people[0];  
  
printf("Using pointer      : %s %s\n", ptr->firstName, ptr->surname);  
printf("people[0].firstName : %s\n", people[0].firstName);  
printf("people[0].firstName[0] : %c\n", people[0].firstName[0]);
```

Console output:

```
Using pointer      : Alisa Erhardt  
people[0].firstName : Alisa  
people[0].firstName[0] : A
```

Assignments & functions

- Assigning structure variables
- Passing structure variables to functions
- Returning structures from functions

- You can assign a structure variable to another structure variable.
- This copies the values of the member variables.



What is printed to the console?

```
struct name {  
    char firstName[NAME_CHARS];  
    char surname[NAME_CHARS];  
};  
  
struct name alisa = { "Alisa", "Erhardt" };  
struct name copy = alisa;  
copy.firstName[0] = 'E';  
printf("Original      : %s\n", alisa.firstName);  
printf("Modified copy : %s\n", copy.firstName);
```

Copies *char* arrays

Console output:

```
Original      : Alisa  
Modified copy : Elisa
```

Function parameters:

- You can pass a structure variable to a function.
- Distinguish passing a structure variable or a pointer:

Parameter type	Sample prototype	Can read original data?	Can modify original data?
Pointer	<code>void func(struct name *person);</code>	yes	yes
Constant pointer	<code>void func(const struct name *person);</code>	yes	no
Value	<code>void func(struct name person);</code>	no (copy)	no (copy)

Returned data:

- A function can return a structure variable.
- Distinguish returning the values (⇒ creates copy) or a pointer accordingly.



What is printed to the console?

```
struct name {  
    char firstName[NAME_CHARS];  
    char surname[NAME_CHARS];  
};  
  
void printAndModifyName(struct name *person);  
  
int main(void)  
{  
    struct name alisa = { "Alisa", "Erhardt" };  
  
    printAndModifyName(&alisa);  
    printf("%s %s\n", alisa.firstName, alisa.surname);  
    getchar();  
    return 0;  
}  
  
void printAndModifyName(struct name *person)  
{  
    printf("%s %s\n", person->firstName, person->surname);  
    person->firstName[0] = 'E';  
}
```

Declare struct before functions

... because here it must be known

Pointer type ⇒ Access original data



And now?

```
struct name {
    char firstName[NAME_CHARS];
    char surname[NAME_CHARS];
};

void printAndModifyName(struct name person);

int main(void)
{
    struct name alisa = { "Alisa", "Erhardt" };

    printAndModifyName(alisa);
    printf("%s %s\n", alisa.firstName, alisa.surname);
    getchar();
    return 0;
}

void printAndModifyName(struct name person)
{
    printf("%s %s\n", person.firstName, person.surname);
    person.firstName[0] = 'E';
}
```

Value type ⇒ Copy of original data



And now?

```
struct name {
    char firstName[NAME_CHARS];
    char surname[NAME_CHARS];
};

struct name printAndModifyName(struct name person);

int main(void)
{
    struct name alisa = { "Alisa", "Erhardt" };
    alisa = printAndModifyName(alisa);
    printf("%s %s\n", alisa.firstName, alisa.surname);
    getchar();
    return 0;
}

struct name printAndModifyName(struct name person)
{
    printf("%s %s\n", person.firstName, person.surname);
    person.firstName[0] = 'E';
    return person;
}
```

Assigns returned value

Value type ⇒ Copy of original data

Returns value

Enumerations

- Use human-readable names (e. g., for colors or months)



When is Alisa's birthday?

```
const char *monthNames[12] = {  
    "January", "February", "March", "April", "May", "June",  
    "July", "August", "September", "October", "November", "December" };  
  
struct date {  
    int dayOfMonth;  
    int month;  
    int year;  
} birthAlisa = { 16, 7, 1991 };  
  
printf("Alisa's birthday is in %s.\n", monthNames[birthAlisa.month]);
```

Suppose it is a day in July

Console output:

Alisa's birthday is in August.

Oops, this is one month too late!

- Enumerations define a set of symbolic names for *integer constants*.
- The names are defined in a comma-separated list.
- By default the values are ascending with the first constant having the value 0.



What is printed to the console?

```
enum {  
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,  
    JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER  
};  
  
const char *monthNames[12] = {  
    "January", "February", "March", "April", "May", "June",  
    "July", "August", "September", "October", "November", "December" };  
  
for (int month = JANUARY; month <= DECEMBER; month++)  
    printf("%-10s: %d\n", monthNames[month], month);
```

Define enumeration constants:
JANUARY = 0, FEBRUARY = 1, ...

Constants are integer values

Example: Calendar month

Improved initialization of month using enumeration constant:

```
enum {
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
    JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER };

const char *monthNames[12] = {
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December" };

struct date {
    int dayOfMonth;
    int month;           Still ambiguous
    int year;           Unambiguous
} birthAlisa = { 16, JULY, 1991 };

printf("Alisa's birthday is in %s.\n", monthNames[birthAlisa.month]);
```

Console output:

Alisa's birthday is in July.

Correct month

Example: Calendar month

Using the enumeration as data type (*enumerated type*):

```
enum month { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER };

const char *monthNames[12] = {
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December" };

struct date {
    int dayOfMonth;
    enum month month;
    int year;
} birthAlisa = { 16, JULY, 1991 };

printf("Alisa's birthday is in %s.\n", monthNames[birthAlisa.month]);
```

Declare name (tag)

Unambiguous

Notes:

- The member *month* of type *enum month* is in fact a variable of type *int*.
- An enumeration provides easy-to-understand names (being constant values).

Recall:

- By default the first constant has value 0.
- Subsequent constants have ascending values.

But:

- You can assign specific values to enumeration constants.

Example (status of a hardware component):

```
enum detectorStatus {  
    DETECT_INTERNAL_ERROR = -1,    // -1  
    DETECT_READY,                // Adding 1 => 0  
    DETECT_NOT_READY = 16,        // 16  
    DETECT_DO_CALIBRATION       // Adding 1 => 17  
};
```

Type definitions

- Improve readability by new names (“aliases”) for existing data types
- Introduce names for a defined bit-depth of, for instance, integers

- Type definitions create a new name (*alias*) for an exiting type.
- Syntax: *typedef dataType alias;*

Example:

```
// Define enumeration for boolean values
enum bool {
    FALSE = 0,
    TRUE = 1
};

// Declare alias
typedef enum bool boolean;

// Use alias as type
boolean isUnderstood = TRUE;
```

The diagram illustrates the use of a type alias. It shows three code snippets. The first snippet defines an enumeration named 'bool' with values 'FALSE' and 'TRUE'. The second snippet declares an alias 'boolean' for the enumeration 'bool'. The third snippet uses the alias 'boolean' to declare a variable 'isUnderstood' with the value 'TRUE'. Brackets from the first two snippets point to a box labeled 'Declare alias boolean', and a bracket from the third snippet points to a box labeled 'Use type boolean'.

- You can combine an enumeration or structure declaration with a type definition.
- No need to assign a tag (i.e., name) to the structure or enumeration.

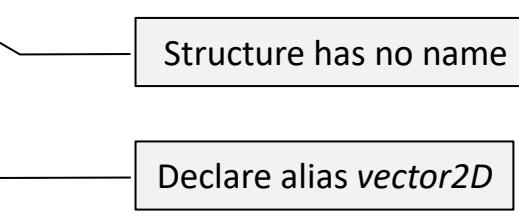
Example:

```
// Define structure and alias
typedef struct {
    double x;
    double y;
} vector2D;           
```

Structure has no name

```
// Use alias as type
vector2D a;
a.x = 1.0;
a.y = 2.7;
```

Declare alias *vector2D*





Let's do some programming!

Types:

- Declare a type *month* defining constants *JANUARY* to *DECEMBER* for calendar months.
- Declare a type *birthday* containing the day of the month, month, and year as well as a string for the person's name.

Functions:

- Write a function *getMonthName()* returning the corresponding string for a *month*.
- Write a function *printBirthday()* printing a person's name and day of birth, stored in type *birthday*, to the console.
- Create and initialize a variable of type *birthday* and invoke the functions above to print the name and day of birth to the console.

Sample console output:

Alisa was born on July 16, 1991.

Constants and types (sample solution):

```
typedef enum {
    JANUARY = 10, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
    AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
} month;

typedef struct {
    const char *name;
    int dayOfMonth;
    month month;
    int year;
} birthday;
```

Does not begin with 0.
Oh no! Will it crash?!

Main function (sample solution):

```
int main(void)
{
    birthday birthAlisa = { "Alisa", 16, JULY, 1991 };
    printBirthday(birthAlisa);
    getchar();
    return 0;
}
```

Other functions (sample solution):

```
/* Get string representation of month constant */
const char *getMonthName(month month)
{
    if ((month >= JANUARY) && (month <= DECEMBER))
    {
        const char *monthNames[12] = {
            "January", "February", "March", "April", "May", "June",
            "July", "August", "September", "October", "November", "December" };
        return monthNames[month - JANUARY];
    }
    else
        return NULL;
}

/* Print birthday to the console */
void printBirthday(birthday day)
{
    printf(
        "%s was born on %s %d, %d.\n",
        day.name, getMonthName(day.month), day.dayOfMonth, day.year);
}
```

Type *month* is in fact *int*

Works even if JANUARY ≠ 0



Typical application:

- Recall that the size of data types may vary between platforms.
- How to ensure that, e.g., an integer variable has exactly 16 bits?

Solution:

- Define types mapping to the correct data types of the specific platform.
- The mapping might differ for another platform.

```
/* Signed integers */
typedef char int8;                      // 8-bit
typedef short int16;                     // 16-bit
typedef int int32;                       // 32-bit
typedef long long int64;                 // 64-bit

/* Unsigned integers */
typedef unsigned char uint8;             // 8-bit
typedef unsigned short uint16;            // 16-bit
typedef unsigned uint32;                  // 32-bit
typedef unsigned long long uint64;        // 64-bit
```

Software Construction 1 (IE1-SO1)

9. Lists & sorting



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

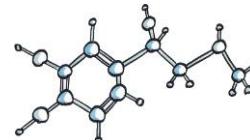
Advanced topics



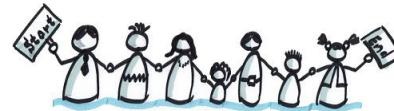
5. Pointers



6. Memory management

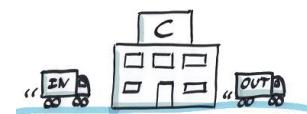


7. Structures



8. Lists and sorting

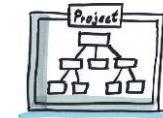
The next steps ...



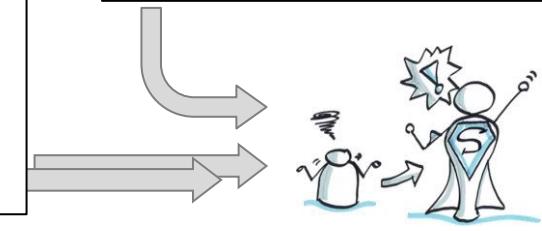
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



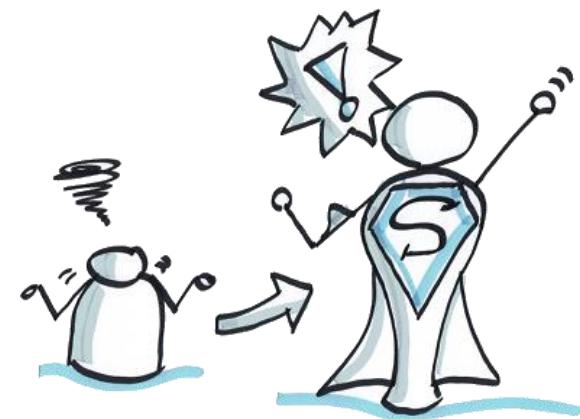
You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *have practiced* concepts discussed so far by applying them to more complex tasks.
- You *connect* variables by pointers in order to create lists of variables that can grow and shrink as required.
- You *sort* sets of data according to own sorting criteria.



1. Linked lists
2. Sorting
 - Bubble sort
 - Single-pass approach
 - Quicksort
 - Linked list (exercise)

Linked lists

- Create collections of elements linked with their prior and/or next element
- Add and remove elements at arbitrary positions



Let's apply! (After all, what does the "A" in "HAW" stand for ...?)

Task overview:

- A bar offers several types of drinks served at the tables.
- Service staff shall get handheld devices to keep track of the ordered drinks.
- Develop and implement related data structures and functions.

Data:

- Offered drinks: water, soft drink, juice, beer, coffee, tea
- Need to track: table number, type of drink, quantity for each ordered type
- Allocate memory for actually placed orders, only.

Functionality:

- Print orders to the console (all orders in system or all orders for a specific table)
- Add an order (⇒ Guests make new order)
- Remove all orders of a specific table (⇒ Guests pay and leave bar)
- Remove all orders in the system

Sample console output:

Taking some orders ...

Table | Order

Table	Order
1	Soft drink (2x)
1	Water (1x)
4	Beer (4x)
6	Tea (2x)
1	Coffee (1x)

Printing orders for table 6, only ...

Table | Order

Table	Order
6	Tea (2x)

Removing orders for table 1 ...

Table | Order

Table	Order
4	Beer (4x)
6	Tea (2x)

Removing all orders ...

Table | Order

Table	Order

Exercise: Have a drink



Let's approach this task step by step:

- Define how to represent and store the required data.
- Write down your approach on paper (or on a laptop, if available).

Recall:

- Offered drinks: water, soft drink, juice, beer, coffee, tea
- Need to track: table number, type of drink, quantity for each ordered type
- Allocate memory for actually placed orders, only.



But ...

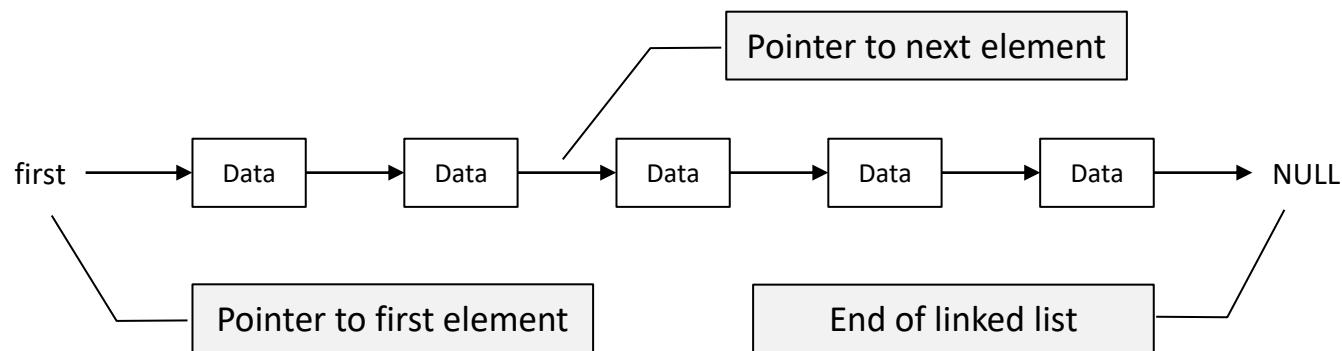
- The number of orders changes as guests come and go.
 - Orders might be removed at any position in the set of orders.
- ⇒ How to deal with that?

Enumeration:

- Enumerate the types of drinks (\Rightarrow Easier to read)

Linked list:

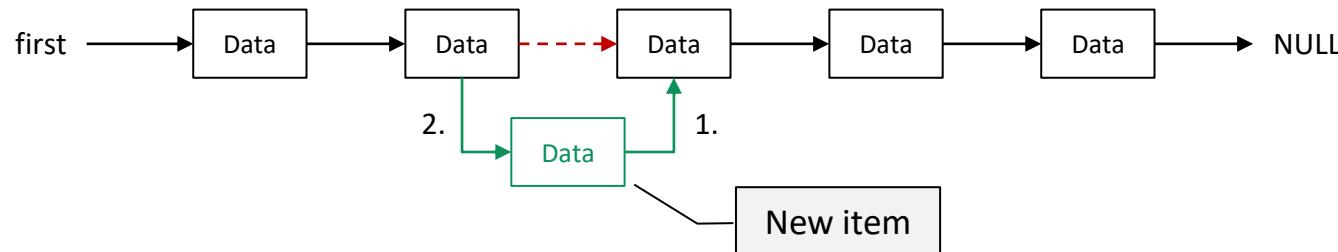
- Group data in a structure (here: a single order).
 - Data contains a pointer $*next$ to the next data element in the list (here: the next order).
 - Pointer $NULL$ indicates the end of the list (i.e., there is no next data element)
- \Rightarrow List of orders can grow and shrink.



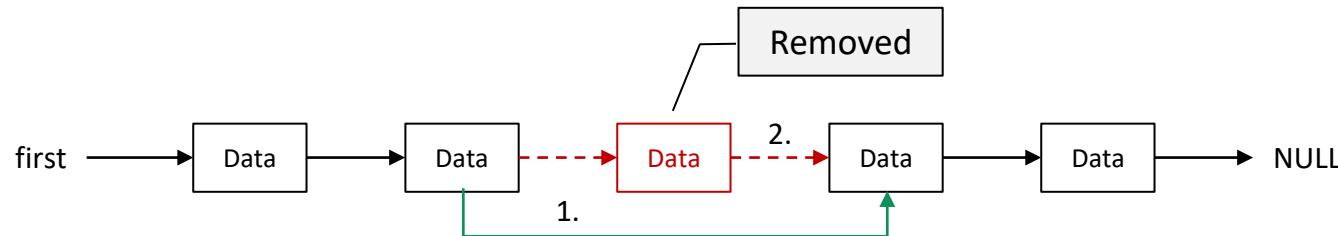
- Empty list:

first → NULL

- Adding an item (*element, node*):



- Removing an item (*element, node*):





- Implement the data structure for an order
- Sample solution:

```
/* Enumerated types */  
typedef enum {  
    WATER,  
    SOFT_DRINK,  
    JUICE,  
    BEER,  
    COFFEE,  
    TEA  
} drink;
```

Enumerated type for drinks

```
/* Structured types */  
typedef struct orderItem {  
    int table;  
    drink drinkID;  
    int quantity;  
    struct orderItem *next;  
} orderItem;
```

Pointer to next list node

Structured type for an order

Example: Have a drink

To get basic understanding of how lists work, first:

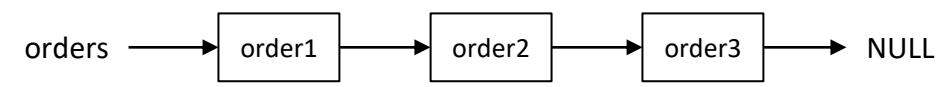
- This is how a list of three orders could be created.
- Functions follow on the next slides. (Exception: Ignore `printOrders()` for the moment.)

```
int main(void)
{
    orderItem *orders = NULL;                                /* List: Pointer to 1st element */
    orderItem *order1 = newOrderItem(1, WATER, 2);          /* Orders (not yet in list) */
    orderItem *order2 = newOrderItem(1, SOFT_DRINK, 1);
    orderItem *order3 = newOrderItem(6, TEA, 2);

    /* Add the orders to the list */
    orders = order1;
    order1->next = order2;
    order2->next = order3;

    /* Print orders to the console and free allocated memory */
    printOrders(orders);
    freeOrderList(orders);
    getchar();
    return 0;
}
```

NULL \Rightarrow List is empty



```
graph LR; orders --> order1[order1]; order1 --> order2[order2]; order2 --> order3[order3]; order3 --> NULL[NULL]
```

Example: Have a drink

- Allocate and initialize a new order structure:

```
orderItem *newOrderItem(int table, drink drinkID, int quantity)
{
    orderItem *newItem = (orderItem *)malloc(sizeof(orderItem));

    if (newItem != NULL)
    {
        newItem->table = table;
        newItem->drinkID = drinkID;
        newItem->quantity = quantity;
        newItem->next = NULL;
    }
    return newItem;
}
```

- Free allocated memory of all order structures in a list:

```
orderItem *freeOrderList(orderItem *orders)
{
    orderItem *node = orders;

    while (node)
    {
        orderItem *freeNode = node;
        node = node->next;
        free(freeNode);
    }
    return NULL;
}
```

Exercise: Have a drink



Now it becomes slightly more complicated:

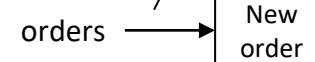
- Implement a function to add a new order to the list
- Notice the difference for empty and non-empty lists

Adding to an empty list:

- Assign new order's address to list pointer (e.g., variable *orders* in *main()*)

```
orderItem *orders = NULL;
```

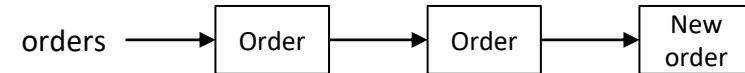
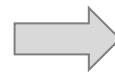
orders → NULL



Change pointer variable in *main()*

Adding to a non-empty list:

- Assign new order's address to pointer in last node



Exercise: Have a drink

```
orderItem *addOrder(orderItem **listPtr, int table, drink menuID, int count)
{
    orderItem *listNode = *listPtr;
    orderItem *newNode;

    // Allocate and initialize new node
    if ((newNode = (orderItem *)malloc(sizeof(orderItem))) != NULL)
    {
        newNode->table = table;
        newNode->drinkID = menuID;
        newNode->quantity = count;
        newNode->next = NULL;
    }

    // Append node to end of list
    if (listNode)
    {
        while (listNode->next)
            listNode = listNode->next;
        listNode->next = newNode;
    }
    else
        *listPtr = newNode;
}

return newNode;
}
```

Pointer to pointer variable in calling function (e.g., *main()*)

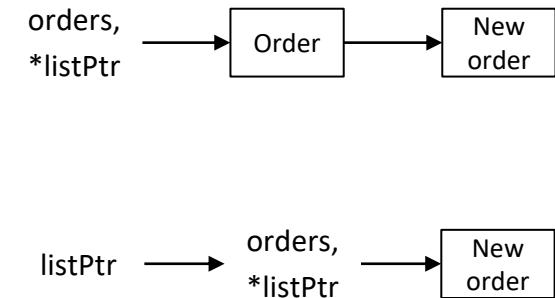
listPtr → orders,
*listPtr

New order

List is not empty

Find last node

Empty list ⇒ Assign to list pointer in calling function



Sample *main()* function:

```
int main(void)
{
    orderItem *orders = NULL;                                Initialize empty list

    /* Fill list with some orders */
    printf("Taking some orders ... \n");
    addOrder(&orders, 1, SOFT_DRINK, 2);                    &orders is pointer to pointer
    addOrder(&orders, 1, WATER, 1);
    addOrder(&orders, 4, BEER, 4);
    addOrder(&orders, 6, TEA, 2);
    addOrder(&orders, 1, COFFEE, 1);
    printOrders(orders, 0);                                 Printing not yet implemented

    /* Print the orders for a specific table, only */
    printf("Printing orders for table 6, only ... \n");
    printOrders(orders, 6);

    getchar();
    return 0;
}
```



Now let's add the missing print function *printOrders()*:

- Calling functions pass the table number for which to print the orders.
- Prints all orders if table is 0.

Recall the sample output:

```
Taking some orders ...
```

```
Table | Order
```

```
-----+-----
```

```
1 | Soft drink (2x)
```

```
1 | Water (1x)
```

```
4 | Beer (4x)
```

```
6 | Tea (2x)
```

```
1 | Coffee (1x)
```

```
Printing orders for table 6, only ...
```

```
Table | Order
```

```
-----+-----
```

```
6 | Tea (2x)
```

Supporting function returning a string representation of drinks:

```
/* Get corresponding string for enumerated drink constants */
const char *drinkToString(const drink id)
{
    switch (id)
    {
        case WATER:
            return "Water";
        case SOFT_DRINK:
            return "Soft drink";
        case JUICE:
            return "Juice";
        case BEER:
            return "Beer";
        case COFFEE:
            return "Coffee";
        case TEA:
            return "Tea";
        default:
            return "Unknown item";
    }
}
```

Notice usage of enumerated type

Why are there no *break*-statements?

Printing orders to the console:

```
/* Print orders for a specific table (or all orders for table = 0) to the console */
void printOrders(const orderItem *orders, int table)
{
    printf("Table | Order\n");
    printf("-----+-----\n");

    if (orders != NULL)
    {
        do
        {
            if ((table == 0) || (table == orders->table))
                printf(
                    "%5d | %-10s (%dx)\n",
                    orders->table,
                    drinkToString(orders->drinkID),
                    orders->quantity);
        } while ((orders = orders->next) != NULL);
        putchar('\n');
    }
}
```

The diagram illustrates the control flow of the `printOrders` function. It starts with a box labeled "Print all orders" connected by an arrow to the beginning of the `do` loop. Inside the loop, an arrow points from the `if` condition to a box labeled "Next order exists?". Another arrow points from the `orders = orders->next` assignment to a box labeled "Move to next order".

Exercise: Have a drink



- Finally let's provide functions to remove orders (complete list or all orders for a table).
- Recall the sample output:

Taking some orders ...

Table | Order

-----+-----

1		Soft drink	(2x)
1		Water	(1x)
4		Beer	(4x)
6		Tea	(2x)
1		Coffee	(1x)

Removing orders for table 1 ...

Table | Order

-----+-----

4		Beer	(4x)
6		Tea	(2x)

Removing all orders ...

Table | Order

-----+-----

Exercise: Have a drink

```
void removeOrders(orderItem **listPtr, int table)
{
    orderItem *prior, *node;

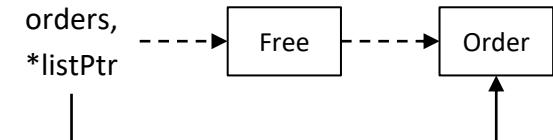
    // Remove matching nodes at beginning of list
    while (*listPtr && ((*listPtr)->table == table))
    {
        node = *listPtr;
        *listPtr = (*listPtr)->next;
        free(node);
    }
    prior = *listPtr;      // No matching table in this item

    // Run through rest of list
    while (prior && (node = prior->next))
    {
        // Remove item, if table matches
        if (node->table == table)
        {
            prior->next = node->next;
            free(node);
        }
        prior = prior->next;
    }
}
```

Pointer to list pointer
in calling function

listPtr → orders,
*listPtr

First element ⇒ Assign next element
to list pointer in calling function



Other elements ⇒ Update pointers

Don't forget to free dynamic memory

Empty complete list:

```
/* Empty list of orders */
orderItem *freeOrderList(orderItem *orders)
{
    orderItem *node = orders;

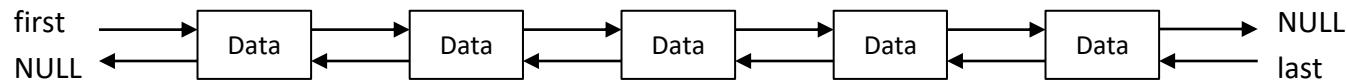
    while (node)
    {
        orderItem *freeNode = node;
        node = node->next;
        free(freeNode);
    }
    return NULL;
}
```

This has been quite a bit of work. Enjoy your drink!

Finally, some further *abstract data types* (without examples or exercises).

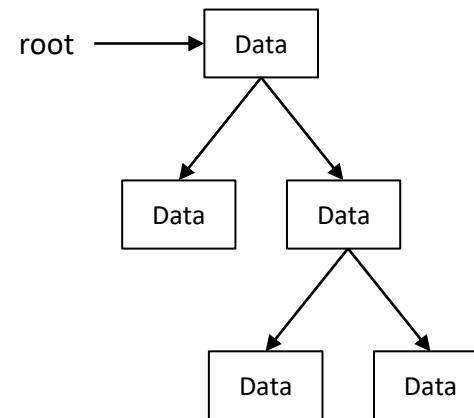
Double linked lists:

- Pointer to next *and* prior element (\Rightarrow navigate through list in both directions)



Binary tree:

- Pointer to two successors



Sorting using bubble sort

- Sort data (e. g., numbers in ascending order)
- Elaborate the standard method “bubble sort”

- Okay, we have helped the service staff manage the orders, but ...
- It helps them to have the orders sorted by table and type of drink.

Sorting:

- Various methods (with specific advantages/disadvantages) exist
- Frequently needed in all sorts of software applications

Examples:

- Email software (e.g., order inbox)
- Financial software (e.g., order invoices by customer and/or date)
- Image processing (e.g., reduce impulse noise by median filters)
- Image rendering (e.g., render objects by layer)
- Operating system (e.g., run threads by priority)
- And so much more ...



Think about it for a minute:

- How would you (as a computer) sort data (e.g., an array of *int* values)?
- What would be your approach?
- Be precise, formulate the *exact* processing steps.

In the following, we will ...

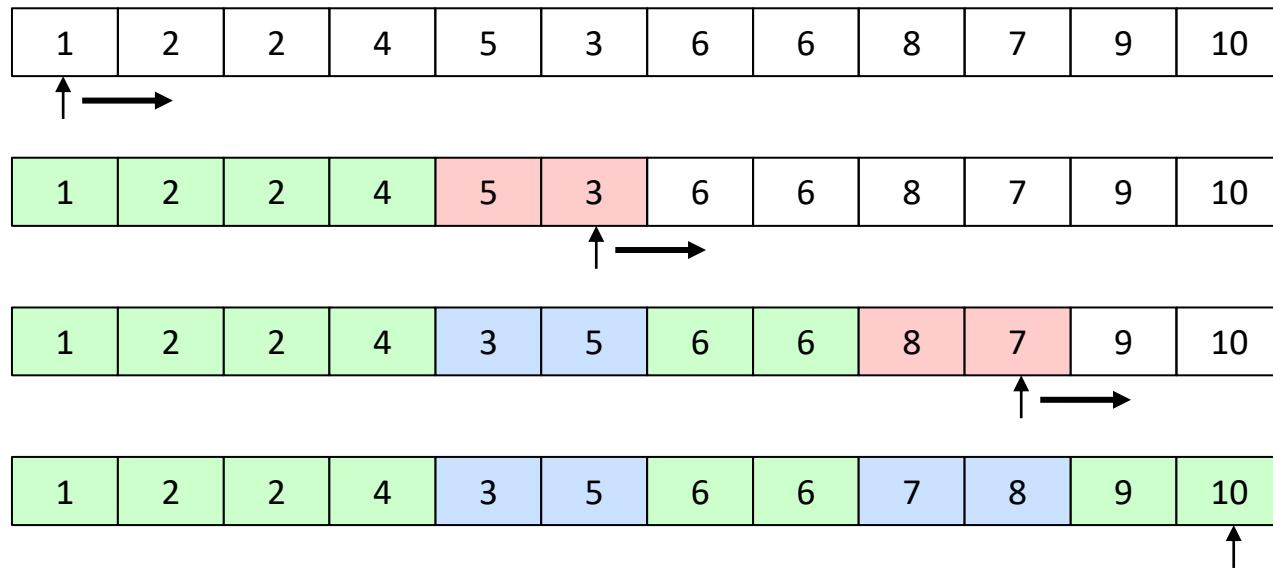
- Introduce *bubble sort* (very slow, but easy to understand)
- Introduce a variation of bubble sort (not quite as slow, but still easy to understand)
- Introduce *quicksort* (faster, but more difficult to understand)
- Help the service staff sort their orders

Principle idea of bubble sort (can be optimized):

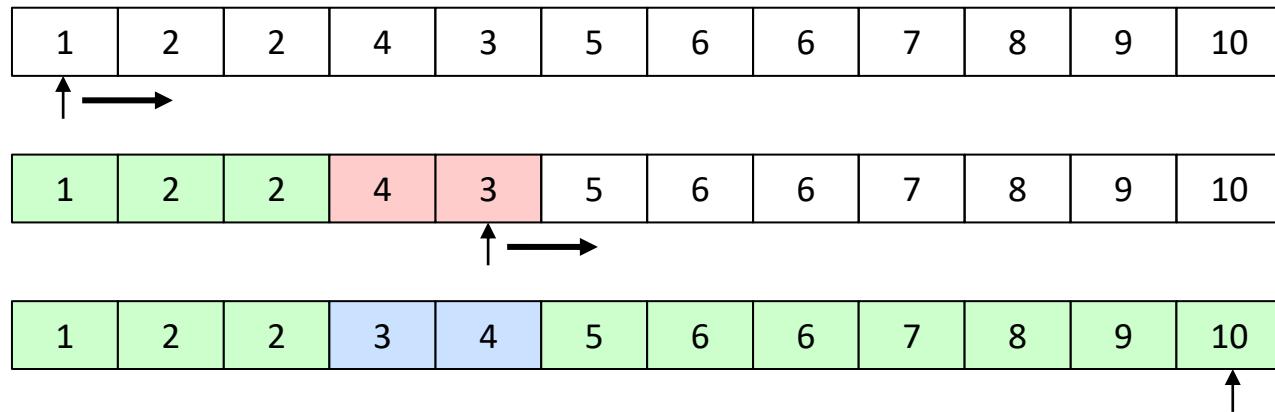
1. Run through data from first to last element
2. If a value is smaller than its left neighbor, swap these two values
3. Repeat until no values were swapped in an iteration through the data

Example:

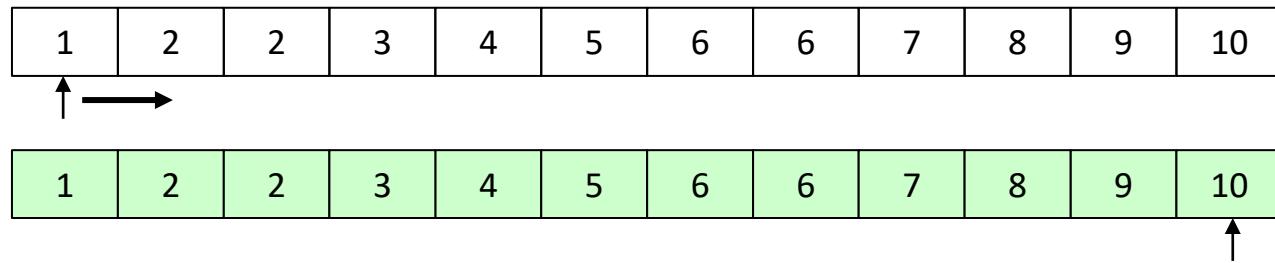
- Iteration 1:



- Iteration 2:



- Iteration 3:





- Implement a bubble sort function for *int* arrays.
- Sample solution:

```
void bubbleSort(int a[], int size)
{
    int isSwapped;

    do
    {
        isSwapped = 0;

        for (int i = 1; i < size; i++)
        {
            if (a[i] < a[i - 1])
            {
                int swap = a[i];
                a[i] = a[i - 1];
                a[i - 1] = swap;
                isSwapped = 1;
            }
        }
    } while (isSwapped);
}
```

Incorrect order

Swap values

Do another iteration

Main function:

```
#define COUNT 25           // Number of random values to sort
#define MAX_VAL 10          // Maximum random value

int main(void)
{
    int bubble[COUNT];

    /* Initialize random values */
    srand((unsigned)time(NULL));
    for (int i = 0; i < COUNT; i++)
        bubble[i] = rand() % MAX_VAL + 1;

    /* Sort array */
    bubbleSort(bubble, COUNT);

    /* Print data to the console */
    for (int i = 0; i < COUNT; i++)
        printf("%3d ", bubble[i]);
    printf("\n");

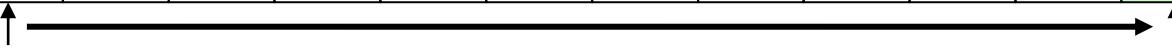
    getchar();
    return 0;
}
```

Note:

- 1st iteration moves the largest value to the final position (⇒ last element)

1	2	9	4	5	3	10	6	8	7	7	8
---	---	---	---	---	---	----	---	---	---	---	---

1	2	4	5	3	9	6	8	7	7	8	10
---	---	---	---	---	---	---	---	---	---	---	----



- 2nd iteration moves the 2nd largest value to final position (⇒ second to last element)

1	2	4	3	5	6	8	7	7	8	9	10
---	---	---	---	---	---	---	---	---	---	---	----



- And so on ...

1	2	4	3	5	6	8	7	7	8	9	10
---	---	---	---	---	---	---	---	---	---	---	----



- ⇒ Move stop index one position left after each iteration
⇒ Decreases runtime of method

Stop index



- Modify the bubble sort function to use a decreasing stop index.
- Sample solution:

```
void bubbleSort(int a[], int size)
{
    int isSwapped;
    int lastID = size - 1;

    do
    {
        isSwapped = 0;
        for (int i = 1; i <= lastID; i++)
        {
            if (a[i] < a[i - 1])
            {
                int swap = a[i];
                a[i] = a[i - 1];
                a[i - 1] = swap;
                isSwapped = 1;
            }
        }
        lastID--;
    } while (isSwapped);
}
```

Decreasing stop index

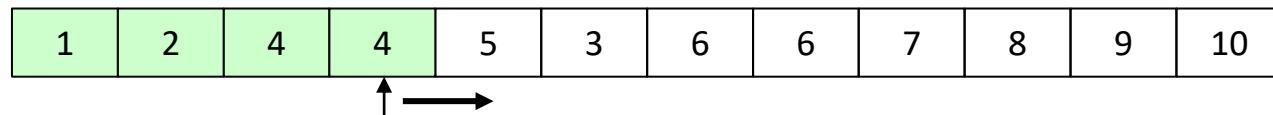
Sorting using a single-pass approach

- Elaborate approaches to speed-up “bubble sort”

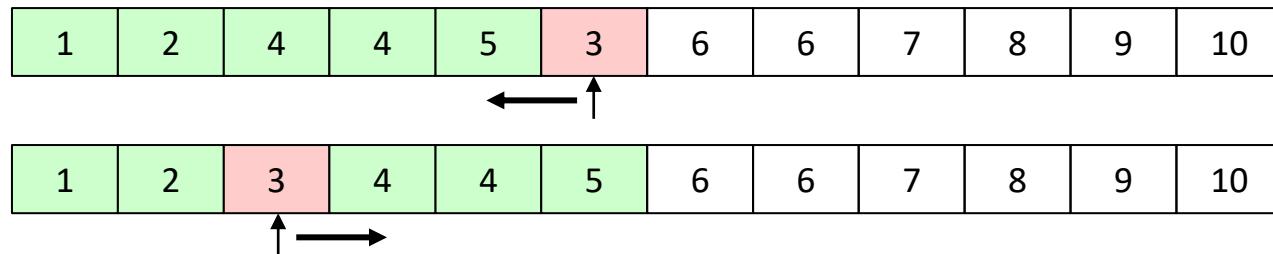
Approach

Principle idea of a single-pass approach for sorting:

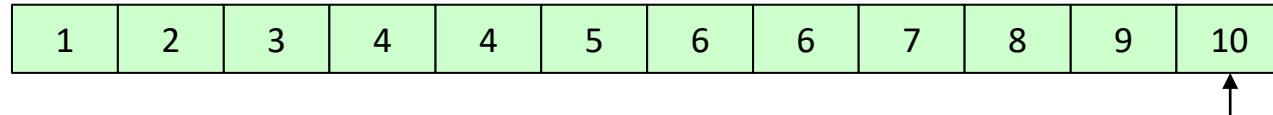
1. Run through data from left to right
2. Proceed as long as values are sorted correctly



3. If a value is smaller than its left neighbor, *move it left until sorted correctly*



4. Repeat steps 2 and 3 until the last element has been processed



Detailed example

Let's visualize the steps in detail using following integer data:

2	3	4	1	6	5
---	---	---	---	---	---

1. Run through the data from left to right
2. Compare each element to its left neighbor (pairwise comparison)

2	3	4	1	6	5
---	---	---	---	---	---

↑ →

2	3	4	1	6	5
---	---	---	---	---	---

↑ →

2	3	4	1	6	5
---	---	---	---	---	---

← ↑

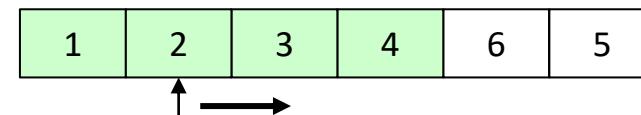
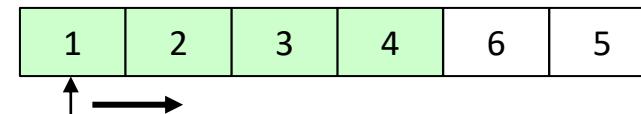
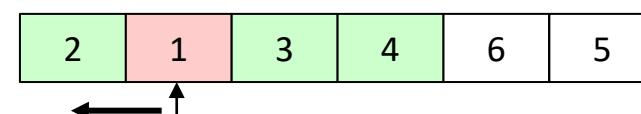
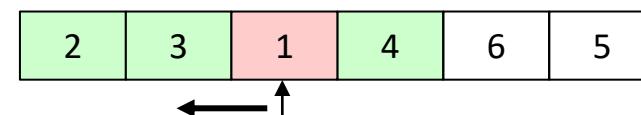
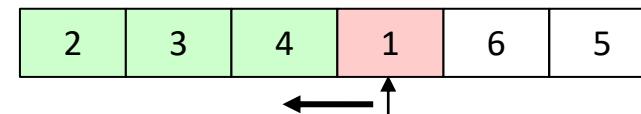
Incorrect order ($1 < 4$)

Detailed example

3. If pair of values is not in correct order:

a) Swap values (\Rightarrow Establish correct order)

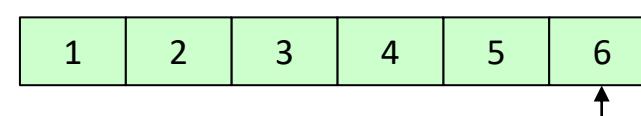
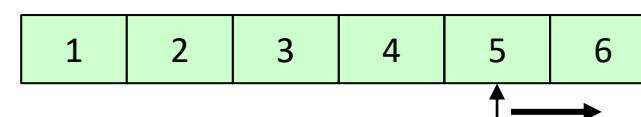
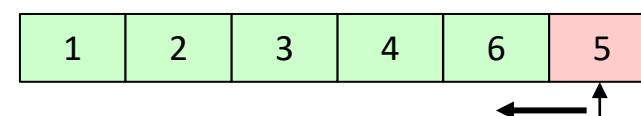
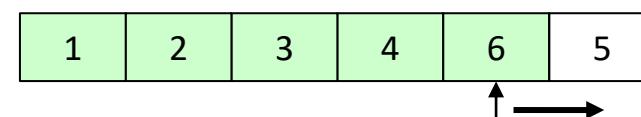
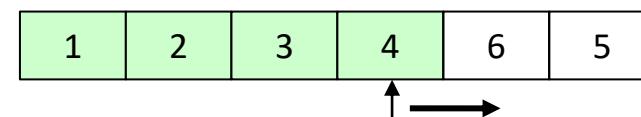
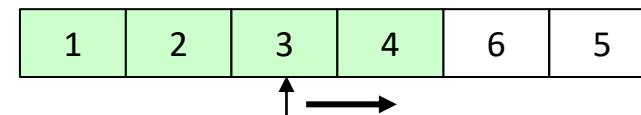
b) Move index one element to left (to compare the smaller value to its left neighbor)



See exercises at end of slide set for a smarter way
 $\Rightarrow \sim 50\%$ runtime

Detailed example

4. When the index reaches the last element, the data is sorted.





- Implement the single-pass method in a function for sorting *int* arrays.
- Sample solution:

```
void singlePassSort(int a[], int size)
{
    for (int i = 1; i < size; i++)
    {
        // Swap neighboring values in incorrect order
        if (a[i] < a[i - 1])
        {
            // Swap values
            int swap = a[i];
            a[i] = a[i - 1];
            a[i - 1] = swap;

            // Move back to compare smaller value to its left neighbor
            if (i > 1)
                i -= 2;
        }
    }
}
```

Effectively $i = i - 1$
(because of $i++$)

Sorting using quicksort

- Get to know a fast sorting method based on “divide and conquer”

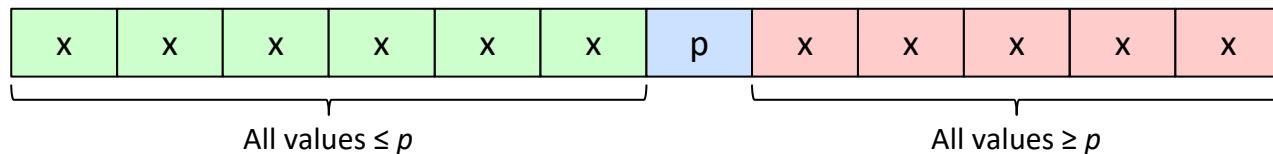
Divide & conquer:

1. Select last element as so-called *pivot* element with value p

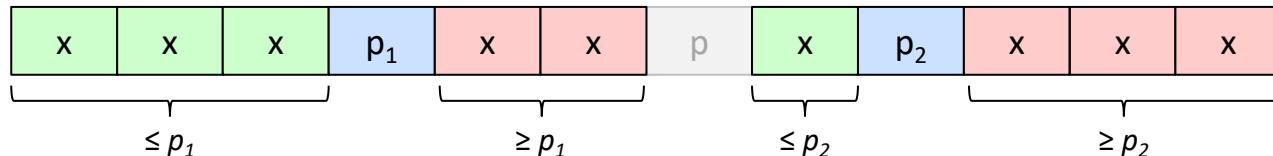


2. Re-order values so that there are two subarrays:

- All values left of pivot are $\leq p$.
- All values right of pivot are $\geq p$.



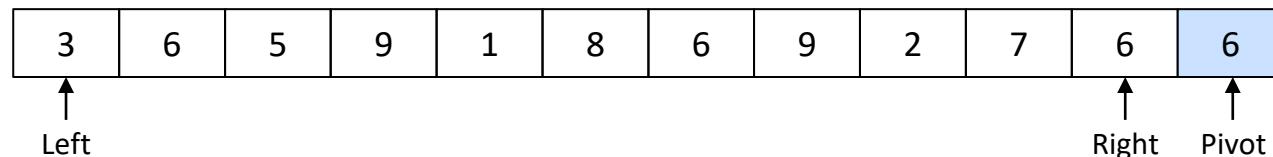
3. Apply method recursively to subarrays (until subarrays contain only 1 element)



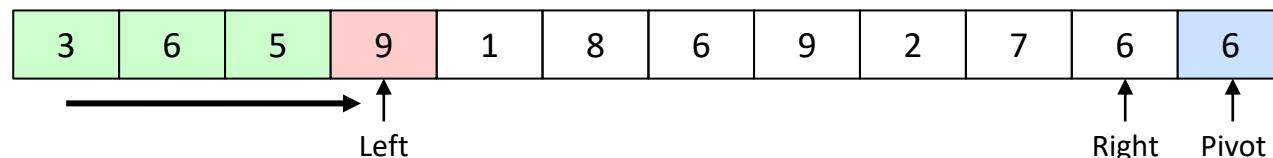
Detailed method and example

Let's visualize the recursion steps using integer data:

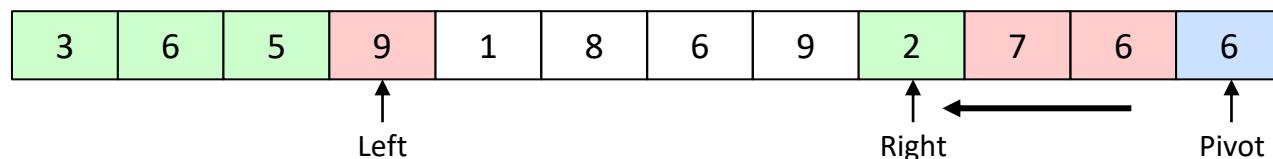
1. Initialize the pivot element and a left and right index



2. Move left index as long as value is \leq 6 (pivot value) and $left \leq right$

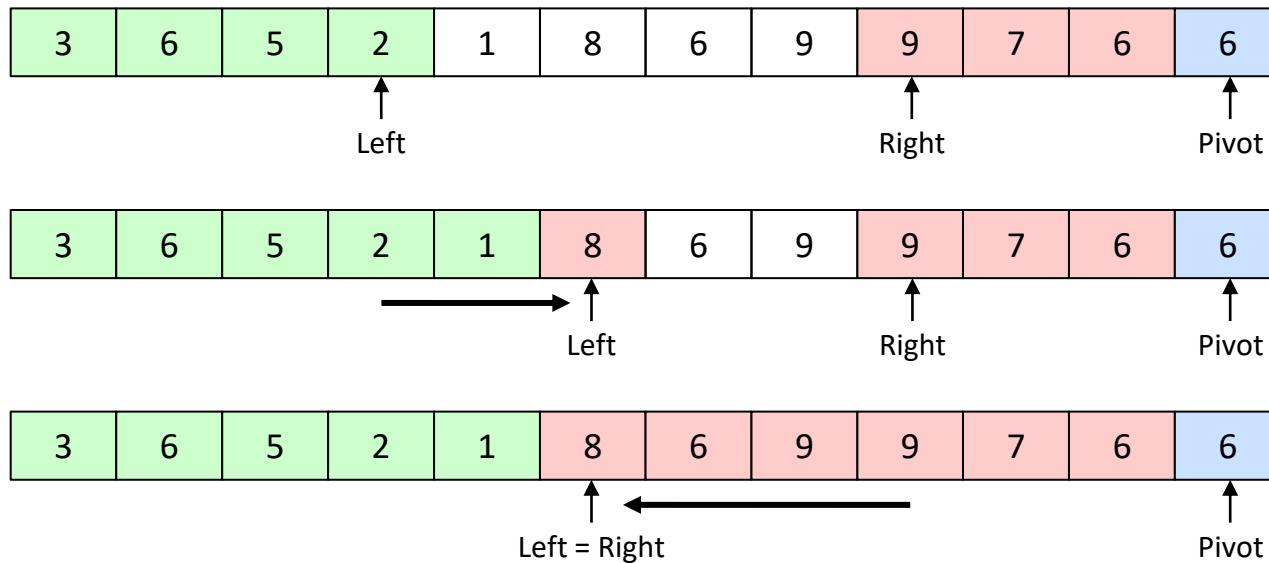


3. Move right index as long as value is \geq 6 (pivot value) and $right > left$

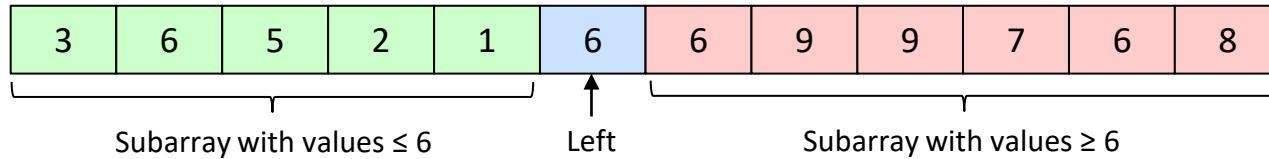


Detailed method and example

4. While $\text{left} < \text{right}$ swap the respective values and repeat steps 2 to 4



5. Swap the pivot value and the left value and apply method to subarrays





- Implement a quicksort function for *int* arrays.
- Compare the runtimes of quicksort, bubble sort, and the single-pass approach applied to the same random data.

Measuring the runtime:

- Function *clock()* returns the elapsed processor clock ticks since program start
- Manifest constant *CLOCKS_PER_SEC* corresponds to the clocks per second
- Both are declared in the standard library *time.h*.

Example (determine time taken in ms):

```
clock_t start;
unsigned timeInMs;

start = clock();           Start "time"
// Do something ...
timeInMs = (unsigned)((clock() - start) * 1000) / CLOCKS_PER_SEC;    Convert to ms
                                                               Stop "time"
```

Exercise: Integer array

```
void quicksort(int a[], int size)
{
    /* Sort >= 3 elements */
    if (size > 2)
    {
        int pivot = size - 1;                      // Choose last element (size - 1) as pivot
        int left = 0, right = size - 2;             // Rest of data is in indices [0, size - 2]
        int swap;

        // Until left and right meet ...
        while (left < right)
        {
            // Move left index as far to the right as possible
            while ((a[left] <= a[pivot]) && (left <= right))
                left++;

            // Move right index as far to the left as possible
            while ((a[right] >= a[pivot]) && (left < right))
                right--;

            // Swap values at left and right indices, if the indices have not yet met
            if (left < right)
            {
                swap = a[left];
                a[left] = a[right];
                a[right] = swap;
            }
        }

        // Continued on next slide ...
    }
}
```

```
void quicksort(int a[], int size)
{
    /* Sort >= 3 elements */
    if (size > 2)
    {
        // ... (see previous slide)

        // Indices have met (left >= right) AND (a[left] >= a[pivot]).
        // Swap a[pivot] and a[left] => Pivot value at final position in sorted array
        swap = a[left];
        a[left] = a[pivot];
        a[pivot] = swap;

        // All values left of a[left] <= pivot value
        // All values right of a[left] >= pivot value
        // => Sort these subarrays (by recursively function call)
        quicksort(a, left);
        quicksort(&a[left + 1], size - (left + 1));
    }

    /* Sort 2 elements by comparison of values */
    else if ((size == 2) && (a[0] > a[1]))
    {
        int swap = a[0];
        a[0] = a[1];
        a[1] = swap;
    }
}
```

Console output for sample run:

```
Initializing 10000 random values in [0, 100] ...
Applying sorting functions to the same values ...
```

```
Time taken (bubble sort): 563 ms
Time taken (single-pass): 445 ms
Time taken (quicksort) : 4 ms
```

Guess why it is called *quicksort* ...

Final remarks on quicksort:

- The standard library contains a function *qsort()* implementing a variant of quicksort.
- Needs another *function* implementing the sorting criteria (comparison) as argument
- We do not discuss function pointers as arguments in this lecture.

Sorting a linked list (exercise)

- Combine the previous sections by applying sorting to linked lists

Exercise: Have a drink



Finally, we come back to our original motivation for sorting:

- Let's enhance the software to manage orders at a bar.
- Add and apply a function to sort orders by table and type of drink.

Sample console output:

Taking some orders ...

Table | Order

-----+-----		
6	Tea	(2x)
1	Beer	(3x)
1	Soft drink	(2x)
6	Water	(3x)
1	Water	(1x)
4	Beer	(4x)
1	Coffee	(1x)
1	Soft drink	(1x)



Sorting orders ...

Table | Order

-----+-----		
1	Water	(1x)
1	Soft drink	(2x)
1	Soft drink	(1x)
1	Beer	(3x)
1	Coffee	(1x)
4	Beer	(4x)
6	Water	(3x)
6	Tea	(2x)

Exercise: Have a drink

```
void sortOrders(orderItem **listPtr)
{
    orderItem **ppNode = listPtr;           // Address of pointer to current node
    orderItem *node = *listPtr;             // Current node: *ppNode
    orderItem *next;                      // Next node: node->next

    while (node && (next = node->next)) —————— Stop when next == NULL
    {
        int isSwapTable = node->table > next->table;
        int isSwapDrink = (node->table == next->table) && (node->drinkID > next->drinkID);

        if (isSwapTable || isSwapDrink)      /* If current and next incorrect order */
        {
            *ppNode = next;                // Swap current and next nodes
            node->next = next->next;
            next->next = node;

            ppNode = listPtr;              // Start again at beginning of list
            node = *listPtr;
        }
        else                                /* Else move to next element */
        {
            ppNode = &node->next;
            node = node->next;
        }
    }
}
```

Incorrect order?

Software Construction 1 (IE1-SO1)

10. Input & output



Fundamentals



1. Data types



2. Flow control



3. Functions

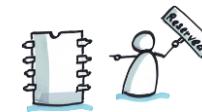


4. Arrays (and strings)

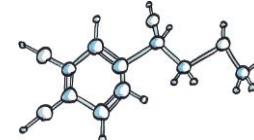
Advanced topics



5. Pointers



6. Memory management

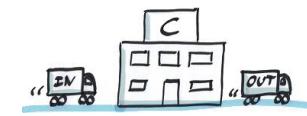


7. Structures



8. Lists and sorting

The next steps ...



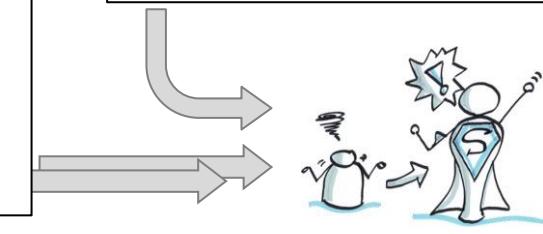
● 9. Input and output



10. Bit operations (“magic”)



11. Project and preprocessor



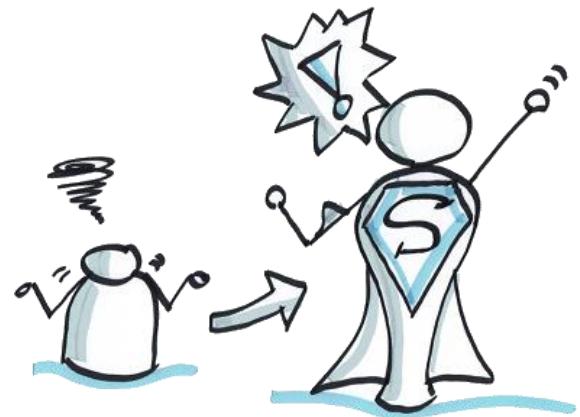
You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *print* text formatted according to your needs to the console.
- You *read* text formatted according to your needs from the keyboard buffer.
- You *write and read* text files.



1. Console output revisited
2. Keyboard input revisited
3. Reading & writing text files

Console output revisited

- The *printf()* function in detail

- We have already been using `printf()` and `scanf()` for a while.
- Let's briefly round off these I/O (= input/output) functions.

Syntax:

```
printf(controlString, item1, item2, ...);
```

Control string:

- Contains characters to be printed and *conversion (or format) specifiers* %...
- Conversion specifiers specify how to convert data into displayable form
- Use %% to print the percentage character %.

Items:

- One item for each conversion specifier
- Must match the type of the conversion specifier

Selected conversion letters:

Specifier	Output format
%c	Character
%s	Character string (until first occurrence of \0)
%d, %i	Integer (signed decimal)
%u	Integer (unsigned decimal)
%o	Integer (unsigned octal)
%x, %X	Integer (unsigned hexadecimal)
%e, %E	Floating-point number (exponential notation aeb or aEb for $a \cdot 10^b$)
%f	Floating-point number (decimal notation)
%g, %G	Floating-point number (exponential or decimal, depending on value)
%p	Pointer

What is printed to the console?

```
printf("Octal      : 27 =  %o\n", 27);
printf("Hexadecimal: 27 =  %x\n", 27);
```

Prefix 0 indicates octal.
Prefix 0x indicates hexadecimal

- Conversion specifiers can contain modifiers.
- Place modifiers between % and the conversion letter.
- For ≥ 2 modifiers, these must be in same order than in table below

Selected modifiers:

Modifier	Meaning	Examples
<i>Flags</i>	See next table	See next table
<i>Integer</i>	Minimum width	%4d, %20s
<i>.Integer</i>	Precision: <ul style="list-style-type: none">▪ Integer: Minimum number of digits (adds leading zeros)▪ Floating-point: Number of decimal places (rounded)▪ String: Maximum number of characters	%.4d, %.5u %.1f, %.3e %.10s
hh	Print an integer as type (<i>unsigned</i>) <i>char</i>	%hhd, %hhu
h	Print an integer as type (<i>unsigned</i>) <i>short</i>	%hd, %hu
l	Print an integer as type (<i>unsigned</i>) <i>long</i>	%ld, %lu
ll	Print an integer as type (<i>unsigned</i>) <i>long long</i> (ISO C99)	%lld, %llu
L	Print a floating-point as type <i>long double</i>	%Lf, %Le
z	Print an integer as type <i>size_t</i> as returned by <i>sizeof</i> (ISO C99)	%zd

Modifier flags:

Flag	Meaning	Examples
-	Align to the left	%-10s
+	Display positive values with a plus sign	%+6.1f
Space	Display positive values with a leading space, but without plus sign	% 6.1f
#	<ul style="list-style-type: none">▪ Octal integer: Add prefix 0▪ Hexadecimal integer: Add prefix 0x or 0X▪ Floating-point: Force to print dot	%#0 %#4x, %#.4x %#.0f, %.0e
0	Pad numeric values with leading zeros	%04d

What is printed to the console?

```
printf("Octal      : %#o\n", 27);
printf("Hexadecimal : %#x\n", 27);
printf("Minimum width : %#6x\n", 27);
printf("Integer      : %#.4x\n", 27);
printf("Floating-point: %+#05.0f\n", 7.1);
```

Return value:

- Number of printed characters (including “invisible” characters like \n)
- Negative number, if an error occurred

Return type:

- Data type may differ between systems (e.g., *unsigned* or *unsigned long*)
- ⇒ Returns type *size_t* defined in *stdio.h*

Keyboard input revisited

- The *scanf()* function in detail

- Converts keyboard input (i.e., character strings) into various data types

Syntax:

```
scanf(controlString, item1, item2, ...);
```

Control string:

- Specifiers %... specify destination data types
- String can contain further characters to specify expected input format

Items:

- Pointers to destination variables
- Typically prefix & to a variable's name (Not for array names, which are pointers.)
- Must match specified data types in the control string

Selected conversion letters:

Specifier	Interpretation of input
%c	Character
%s	Character string Reads from first non-whitespace character until next whitespace, excluded.
%d, %i	Integer (signed decimal)
%u	Integer (unsigned decimal)
%o	Integer (signed octal)
%x, %X	Integer (signed hexadecimal)
%f, %e, %g, %F, %E, %G	Floating-point number (without further modifier as type <i>float</i>) All specifiers accept decimal and exponential notation.
%p	Pointer
%n	Number of characters read by <i>scanf()</i> (provide pointer to <i>int</i> variable)

- Specifier %n does not scan user input, but stores the overall number characters read

Conversion specification modifiers

- Conversion specifiers can contain modifiers.
- Place modifiers between % and the conversion letters.
- For ≥ 2 modifiers, these must be in same order than in table below

Selected modifiers:

Modifier	Meaning	Examples
*	Suppress assignment Read data from the buffer, but do not store it in a variable.	%*d
Integer	Maximum width Stop reading at maximum width or at first whitespace.	%20s
hh	Read an integer as type (<i>unsigned</i>) <i>char</i>	%hhd, %hhu
h	Read an integer as type (<i>unsigned</i>) <i>short</i>	%hd, %hu
l	▪ Integer: Read as type (<i>unsigned</i>) <i>long</i> ▪ Floating-point: Read as type <i>double</i>	%ld, %lu %lf, %le, %lg
ll	Read an integer as type (<i>unsigned</i>) <i>long long</i> (ISO C99)	%lld, %llu
L	Read a floating-point as type <i>long double</i>	%Lf, %Le, %Lg
z	Read an integer as type <i>size_t</i> as returned by <i>sizeof</i> (ISO C99)	%zd, %zo

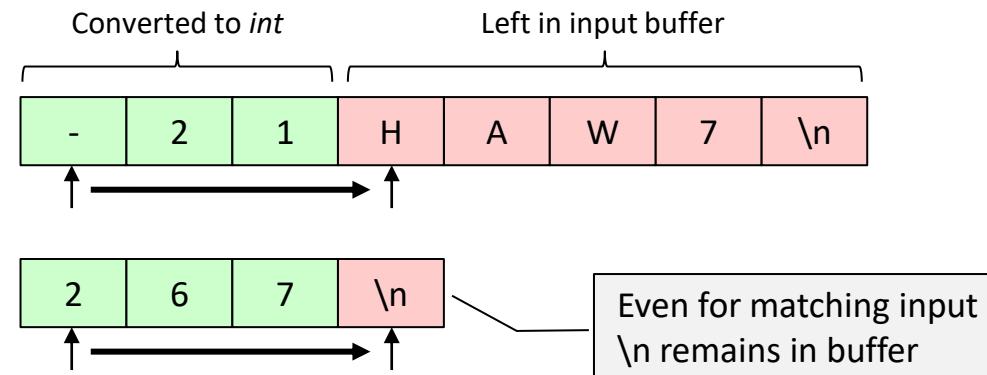
Line buffer:

- System buffers keyboard input
- Input not available to a program (i.e., `scanf()`) before users press enter key (= '\n')

Scanning the buffer:

- Function takes characters from the buffer as long as these match the expected input
- Sample code and user input:

```
int x;  
scanf("%d", &x);
```



- Function ignores whitespaces (i.e., new line, tab, space) between fields
 - Exception: %c reads next character (including whitespace characters)
- ⇒ Input may have leading spaces and whitespaces between data.

Example:

```
int x, y, z;
printf("Enter integer values x, y, and z: ");
scanf("%d%d%d", &x, &y, &z);
printf("(x, y, z) = (%d, %d, %d)\n", x, y, z);
```

Sample user input and console output:

Enter integer values x, y, and z: 2

3 4

(x, y, z) = (2, 3, 4)

New line (removed from buffer)

Spaces in input

- Additional characters in the control string define the expected input format.
- These are *not* printed to the console.

Example:

- Reading comma-separated *int* values:

```
int x, y;
printf("Enter point <x>, <y>: ");
scanf("%d , %d", &x, &y);
printf("(x, y) = (%d, %d)\n", x, y);
```

- Different sample input resulting in x = 1 and y = 2:

Enter point <x>, <y>: 1,2

Enter point <x>, <y>: 1, 2

Enter point <x>, <y>: 1,
2

Return value:

- Number of items successfully read or 0
- *EOF* when system-dependent *end of file* character has been detected

Exercise: Reading calendar days



- Scan for calendar days formatted *dd.mm.yyyy* (with day of month, month, and year).
- Sample solution:

```
int dayOfMonth, month, year;
char *monthAbbr[] = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

/* Read user input and print to the console */
printf("Enter calendar day (dd.mm.yyyy): ");
while (scanf("%2d.%2d.%4d", &dayOfMonth, &month, &year) == 3)
{
    // Print to the console
    if ((month >= 1) && (month <= 12))
        printf("You have entered %s %.2d, %d.\n", monthAbbr[month - 1], dayOfMonth, year);
    else
        printf("Month must be within 1 to 12.\n");

    // Clear keyboard buffer and prompt for next input
    while (getchar() != '\n')
        continue;
    printf("Next day (q to quit): ");
}

/* Clear keyboard buffer */
while (getchar() != '\n')
    continue;
```

Reading & writing text files

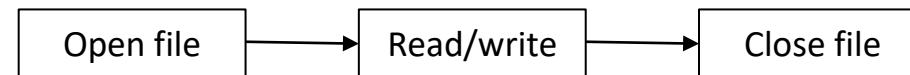
- A file is a reserved area in storage (e.g., hard disk) with an assigned name.
- In C a file is a sequence (*stream*) of bytes.
- Can read/write bytes (*binary mode*) or characters (*text mode*)
- Related functions are declared in *stdio.h*.
- In the following we discuss text files, only.

File paths using Visual Studio:

- Typically programs read/write files in the same directory as the executable file.
- Debug build: Working directory set in project properties (*Debugging/Working Directory*)

Principle file access:

1. Open a file (existing file or create new one)
2. Read and/or write data
3. Close file



Function `fopen()` opens a file for reading and/or writing:

```
FILE *fopen(const char *fileName, const char *mode);
```

Parameters:

- The file name may include the path (using “\\” as separator on Windows systems).
- For text files the mode is one of the following:

Mode	Read / write	File modification
r	Reading	None
w	Writing	Overwrites contents
a	Writing	Appends to contents
r+	Reading and writing	Reads from beginning, overwrites contents
w+	Reading and writing	Deletes contents when opening file
a+	Reading and writing	Reads from the beginning, but appends to contents

Caution: Some modes *overwrite* existing files!

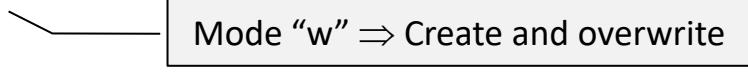
Except for mode “r”, all modes create a new file, if the file does not exist.

Return value:

- Returns a file pointer of type *FILE* or *NULL* (if file could not be opened)
- *FILE* does not point to the file, but a structure containing information about the file

Example:

```
const char *fileName = "sampleFile.txt";
FILE *file = fopen(fileName, "w");
if (file != NULL)
    printf("File opened: \"%s\"\n", fileName);
else
    printf("Could not open file: \"%s\"\n", fileName);
```



Mode "w" ⇒ Create and overwrite

Function `fclose()` closes an open file:

- Pass the file pointer as argument
- The function flushes write buffers, if not empty
- Makes files accessible by other programs again

Notes:

- Always, always, always (... did I already mention *always*?) close open files.
- Function `exit()` automatically closes all open files before terminating the program

Example:

```
FILE *file = fopen("sampleFile.txt", "w");
if (file != NULL)
{
    // Write data to the file ...

    fclose(file);
}
```

Write functions for text files:

Function	Writes	Corresponding function for <i>stdout</i>
<code>int putc(int, FILE *)</code>	Character	Same as <i>putchar()</i>
<code>int fprintf(FILE *, const char *, ...)</code>	Formatted string	Same as <i>printf()</i>
<code>int fputs(const char *, FILE *)</code>	String	Same as <i>puts()</i> , but does not append a newline character \n

Example:

```
FILE *file = fopen("sampleFile.txt", "w");
if (file != NULL)
{
    fputc('W', file);
    fprintf(file, "eather forecast: %.1f C and sunny.\n", 21.233);
    fputs("Go out and have some fresh air.\n", file);

    fclose(file);
}
```

Read functions for text files:

Function	Reads	Corresponding function for <i>stdin</i>
<code>int getc(FILE *)</code>	Character	Same as <code>getchar()</code>
<code>int fscanf(FILE *, const char *, ...)</code>	Formatted input	Same as <code>scanf()</code>
<code>char *fgets(char *, int, FILE *)</code>	String	Same as <code>gets()</code> , but keeps the newline character <code>\n</code> and reads at most <i>n</i> characters. (See slide set on pointers)

End of file:

- Function `getc()` returns *EOF* when trying to read past end of file
- Function `fgets()` returns *NULL* when reaching end of file without characters to read



- Write some text (including line breaks) to a text file.
- Read the text as character strings from the file and print it to the console.

Sample solution:

```
#define FILE_NAME "sampleFile.txt"
#define READ_SIZE 16

int main(void)
{
    FILE *file;
    char string[READ_SIZE];

    /* Write text file */
    if ((file = fopen(FILE_NAME, "w")) == NULL)
        exit(EXIT_FAILURE);

    fputs("HAW Hamburg\nBerliner Tor 7\n20099 Hamburg\n", file);
    fclose(file);

    // Reading and printing (see next slide) ...
}
```

Mode "w" ⇒ Create and overwrite

Sample solution (continued):

```
#define FILE_NAME "sampleFile.txt"
#define READ_SIZE 16

int main(void)
{
    FILE *file;
    char string[READ_SIZE];

    // Writing to file (see previous slide) ...

    /* Read and print text file */
    if ((file = fopen(FILE_NAME, "r")) == NULL)
        exit(EXIT_FAILURE);

    while (fgets(string, READ_SIZE, file) != NULL)
        printf(string);
    fclose(file);

    getchar();
    return 0;
}
```

Mode “r” ⇒ Read (if exists)

Maximum number to read

End of file reached



- Modify the previous exercise so that single characters are read.

Sample solution:

```
int main(void)
{
    FILE *file;
    char character;

    // Writing to file (see previous slide) ...

    /* Read and print text file */
    if ((file = fopen(FILE_NAME, "r")) == NULL)
        exit(EXIT_FAILURE);

    while ((character = getc(file)) != EOF)
        putchar(character);
    fclose(file);

    getchar();
    return 0;
}
```

Mode “r” ⇒ Read (if exists)

End of file reached



Think about it:

- How to let a function print the same text either to a file or to the console?

Example:

```
int main(void)
{
    /* Write address to text file */
    FILE *file = fopen("sampleFile.txt", "w");
    printAddress(file);
    fclose(file);

    /* Write address to the console */
    printAddress(stdout);
    getchar();
    return 0;
}

void printAddress(FILE *stream)
{
    fputs("HAW Hamburg\nBerliner Tor 7\n20099 Hamburg\n", stream);
}
```

File pointer *stdout* ⇒ Console

Software Construction 1 (IE1-SO1)

11. Bit operations



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

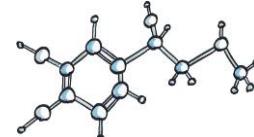
Advanced topics



5. Pointers



6. Memory management

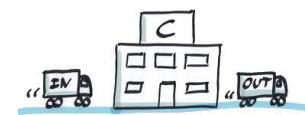


7. Structures



8. Lists and sorting

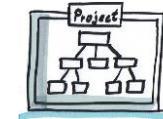
The next steps ...



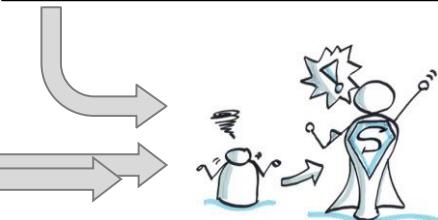
9. Input and output



● 10. Bit operations ("magic")



11. Project and preprocessor



You!

● Location

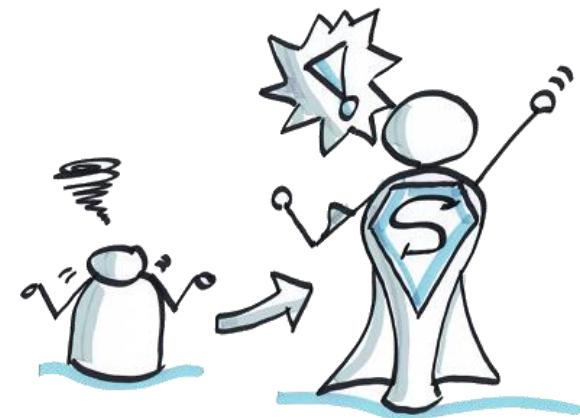
What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You can do all the fancy bit magic that many people using other languages cannot!

Okay, somewhat more precise ...

- You *access and manipulate* individual bits, for instance, to control hardware components or minimize memory for storage and/or data transmission.
- You *shift* bit patterns in memory to the left and/or the right, for instance, to implement fast mathematical operations.



1. Logical bitwise operators
2. Bit shifting

Logical bitwise operators

- Logical operators applied to *each* bit position independently
- Typical tasks (such as setting and reading bit flags)

Recall bits & bytes:

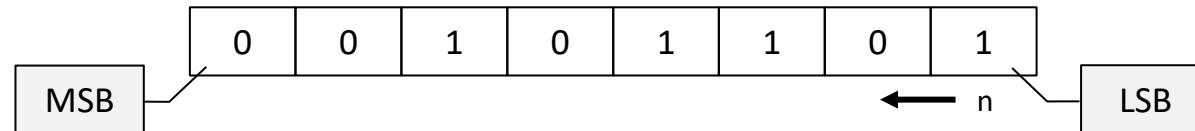
- Data in memory is represented by *bits* (*binary digits*, value 0 or 1).
- Commonly a *byte* contains 8 bits (also called an *octet*).
- The C standard defines a byte as number of bits used for the data type *char*.

Unsigned integers:

- Bit n (reading right to left, starting with index 0) represents decimal value 2^n
- *Low-order bit* (or *least significant bit, LSB*): Bit representing lowest value
- *High-order bit* (or *most significant bit, MSB*): Bit representing highest value

Example:

- Decimal: $(00101101)_2 = 2^5 + 2^3 + 2^2 + 2^0 = 32 + 8 + 4 + 1 = 45$
- Hexadecimal: $(00101101)_2 = 0x((0010)_2(1101)_2) = 0x2D$



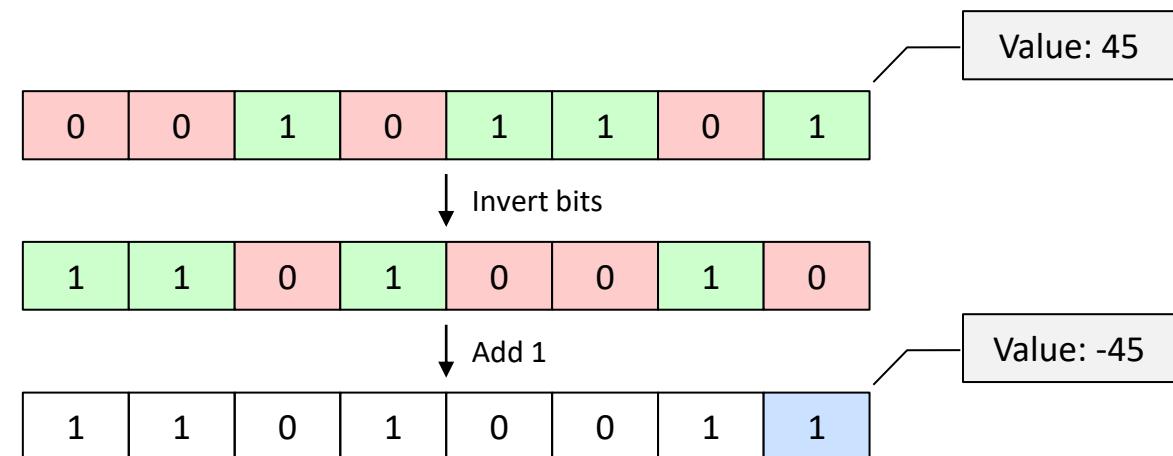
Signed integers as *two's complement*:

- Wide-spread representation, though bit representation can differ for platforms
- Highest bit MSB determines sign (negative number for MSB = 1)

Changing the sign $x \rightarrow -x$:

1. Invert each bit (i.e., 0 → 1 and 1 → 0)
2. Add 1 (i.e., bit pattern 0 … 01)

Example:

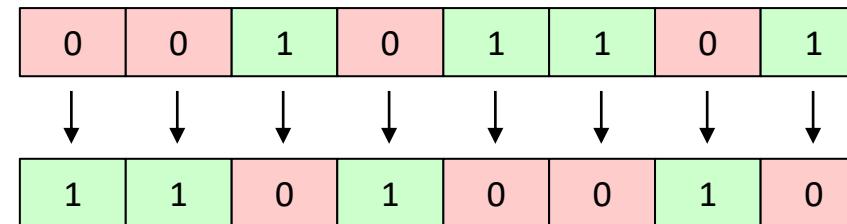


Logical bitwise operations:

- Work on all integer data types (including *char*)
- Operate on each bit, independently of other bits
- Note that all bit operations are *fast*.

Negation operator \sim (*one's complement*):

- Inverts bits ($0 \rightarrow 1$ and $1 \rightarrow 0$)
- Example:



Exercise: Two's complement



- Change the sign of an integer number twice, applying bitwise negation.
- Use a 1-byte signed integer as data type.

Sample solution:

```
signed char x = 45;  
signed char y = ~x + 1;
```

Does not modify x

```
printf("Change sign twice (two's complement):\n");  
printf("% hhd -> % hhd\n", x, y);  
printf("% hhd -> % hhd\n", y, ~y + 1);
```

Leave space for sign

Console output:

Change sign twice (two's complement):

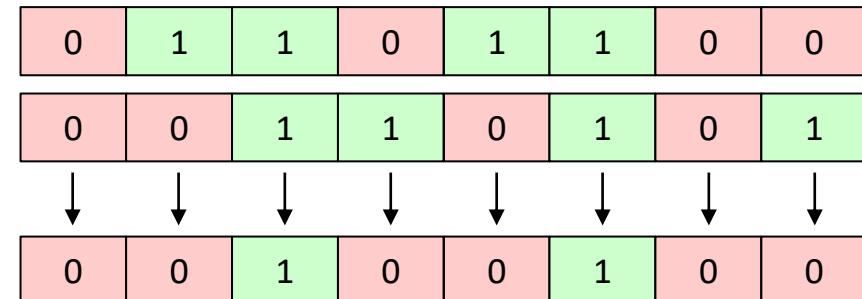
45 -> -45

-45 -> 45

AND operator &:

- Bit is 1, if corresponding bits of both operands are 1

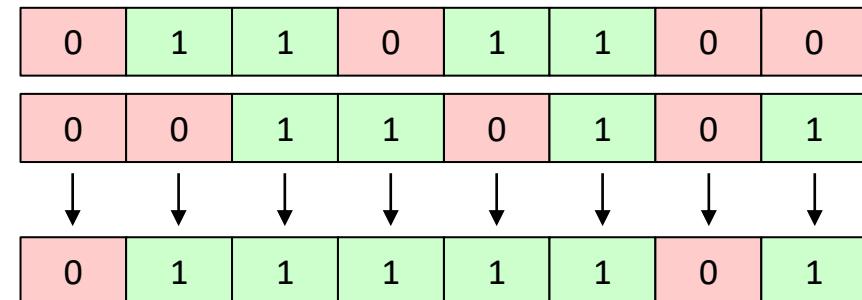
```
unsigned char a = 0x6c;  
unsigned char b = 0x35;  
unsigned char bitAnd = a & b;
```



OR operator |:

- Bit is 1, if at least one of the corresponding bits of the operands is 1

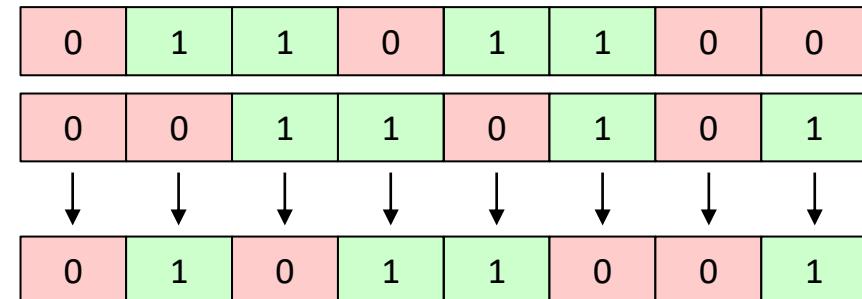
```
unsigned char a = 0x6c;  
unsigned char b = 0x35;  
unsigned char bitOr = a | b;
```



Exclusive OR operator (XOR) ^:

- Bit is 1, if exactly one corresponding bit of the operands is 1

```
unsigned char a = 0x6c;  
unsigned char b = 0x35;  
unsigned char bitXor = a ^ b;
```



Assignment operators:

- Assignment operators <op>= exist for all binary operations.

```
unsigned char a1 = 0x6C;  
unsigned char a2 = 0x6C;  
unsigned char a3 = 0x6C;
```

```
a1 &= 0x35;  
a2 |= 0x35;  
a3 ^= 0x35;
```

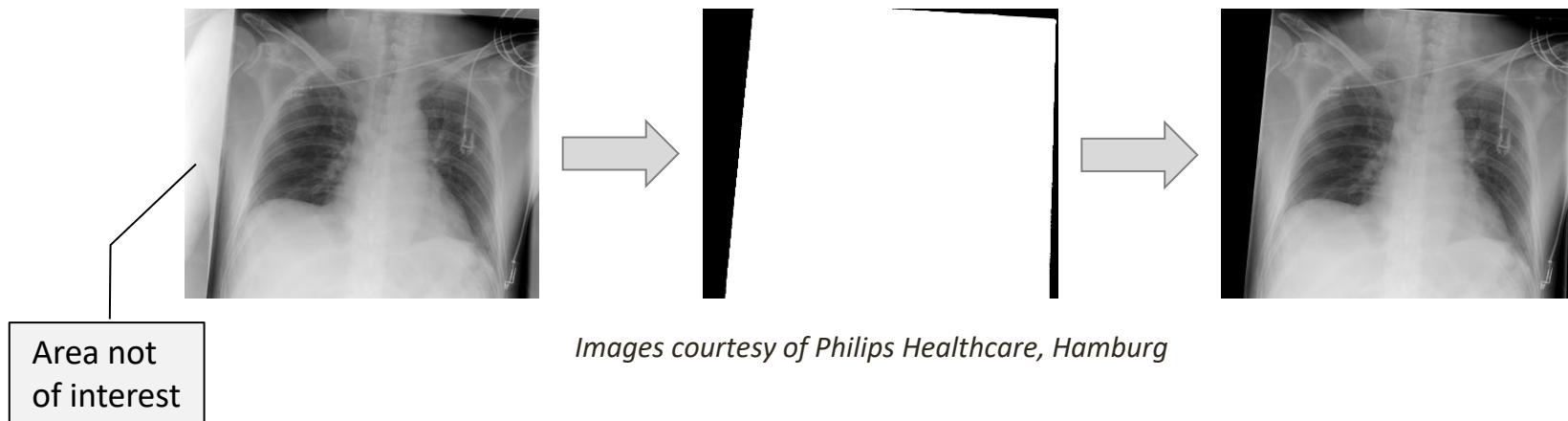
Same as a1 = a1 & 0x35;

Controlling hardware:

- Define bytes where the bits (*flags*) control hardware components
- Example: Speaker enabled (or disabled), if specific bit is 1 (or 0)

Binary digital images:

- Pixels in *binary* images have values 1 or 0, only.
- Each byte representing 8 pixels saves memory compared to using a byte for each pixel
- Example: Binary image to mask region of interest in medical X-ray image



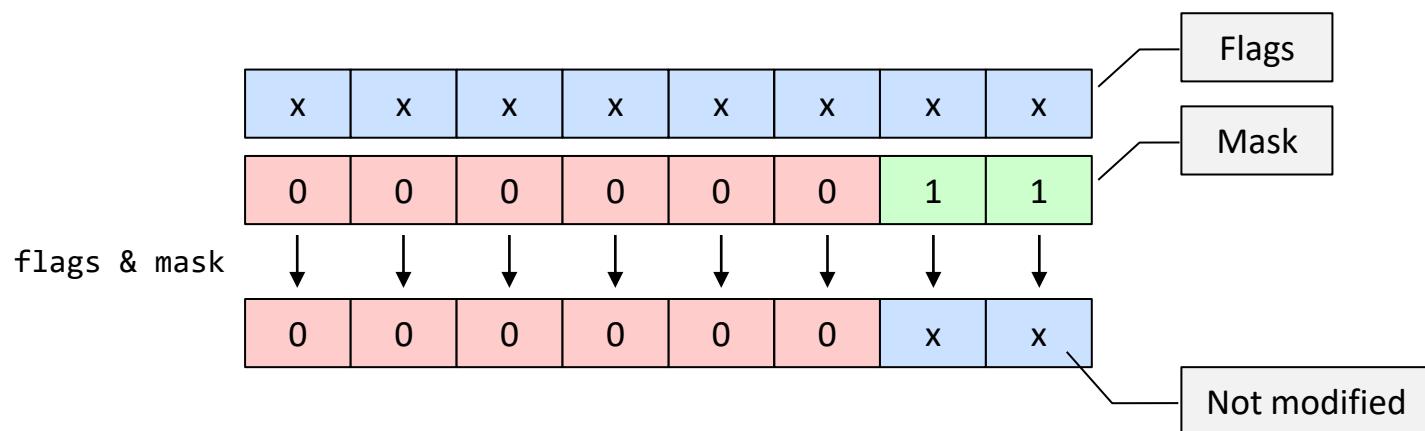
In the following:

- Objective to modify or analyze bits (*flags*) in a byte
- Will use another byte (*mask*) for the modification or analysis



Masking:

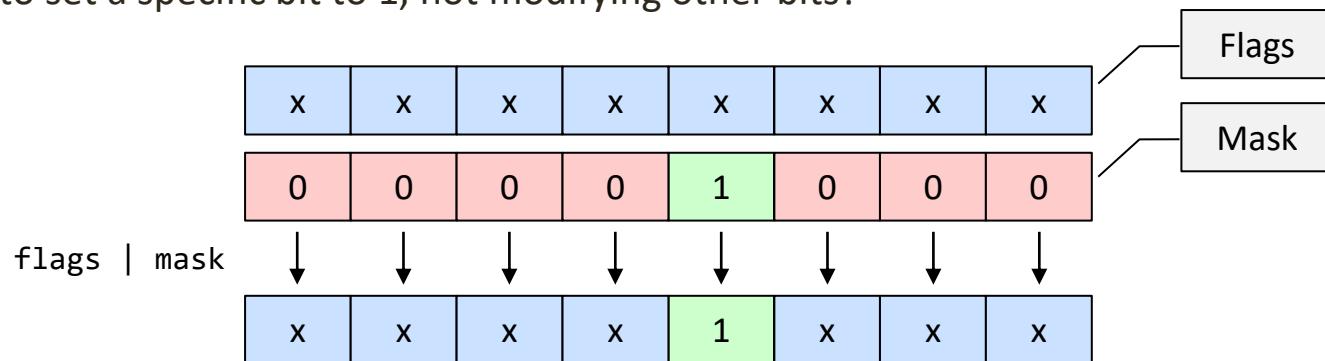
- How to hide (i.e., show as 0) all bits except where the mask is 1?
- Answer: Apply AND-operation: `flags & mask`





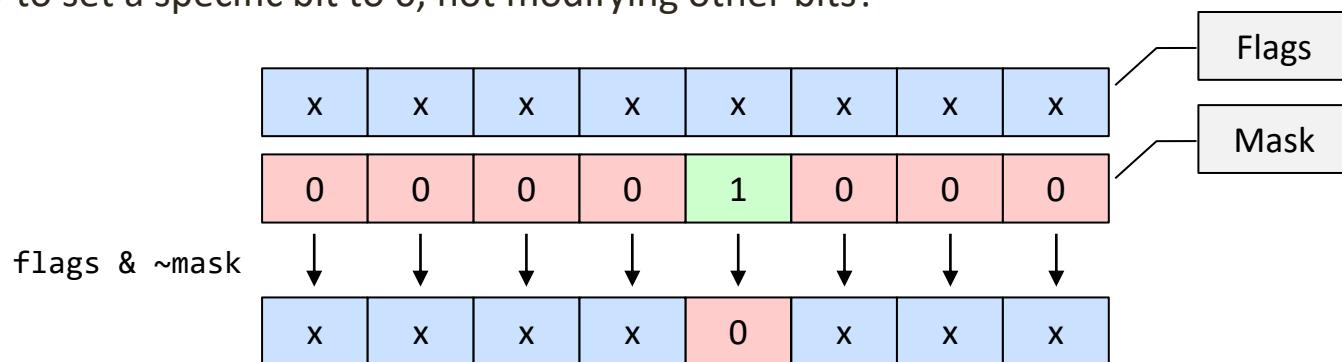
Setting bits (turning bits “on”):

- How to set a specific bit to 1, not modifying other bits?



Clearing bits (turning bits “off”):

- How to set a specific bit to 0, not modifying other bits?

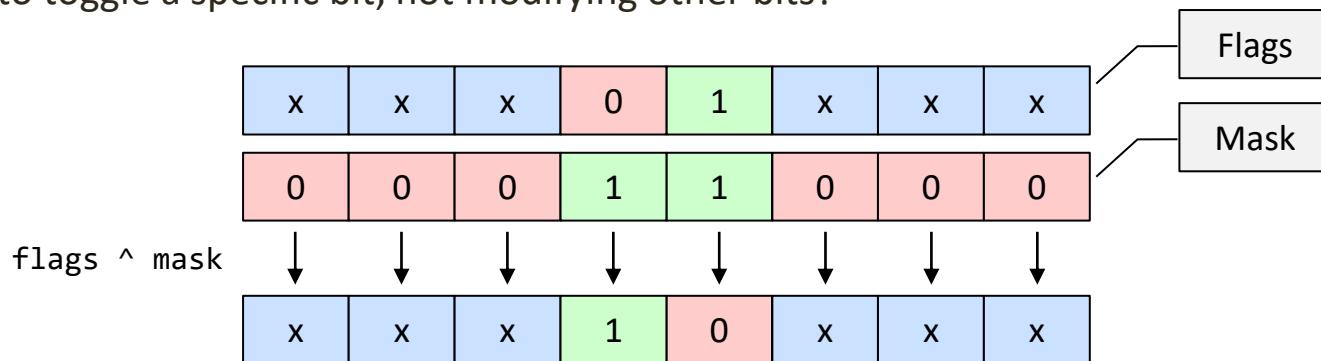


Typical tasks



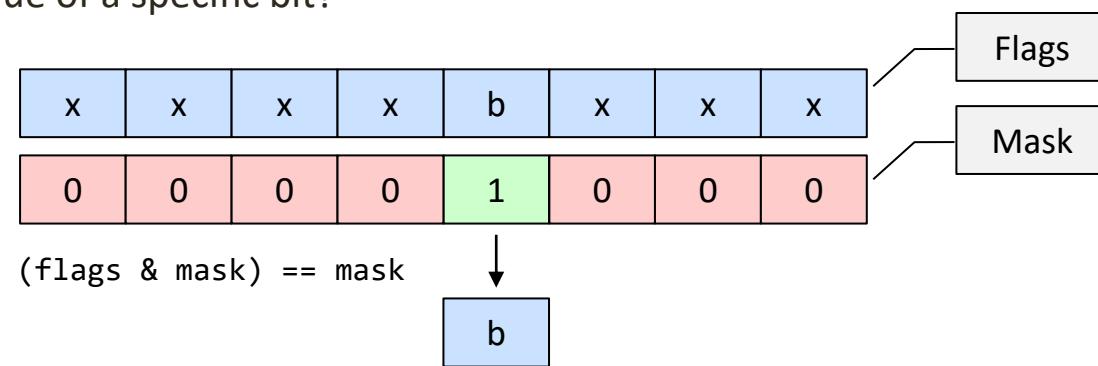
Toggling bits ($0 \rightarrow 1$ and $1 \rightarrow 0$):

- How to toggle a specific bit, not modifying other bits?



Checking bit values:

- How to get the value of a specific bit?





Time for some doing:

- Implement a function that prints the lowest byte of an integer number to the console.

Sample solution:

```
void printLowByteBinary(int a)
{
    char string[9];
    unsigned char mask = 1;

    /* Create string */
    for (int i = 7; i >= 0; i--, mask *= 2)
        string[i] = ((a & mask) == mask) ? '1' : '0';
    string[8] = '\0';

    /* Print to the console*/
    printf("%s", string);
}
```

Initial mask: 00000001

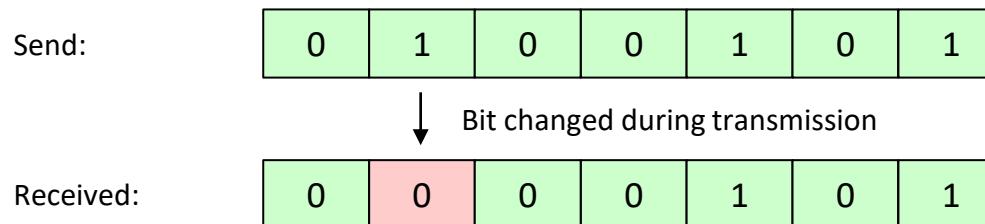
Move 1 one bit to the left

Exercise: Parity bit



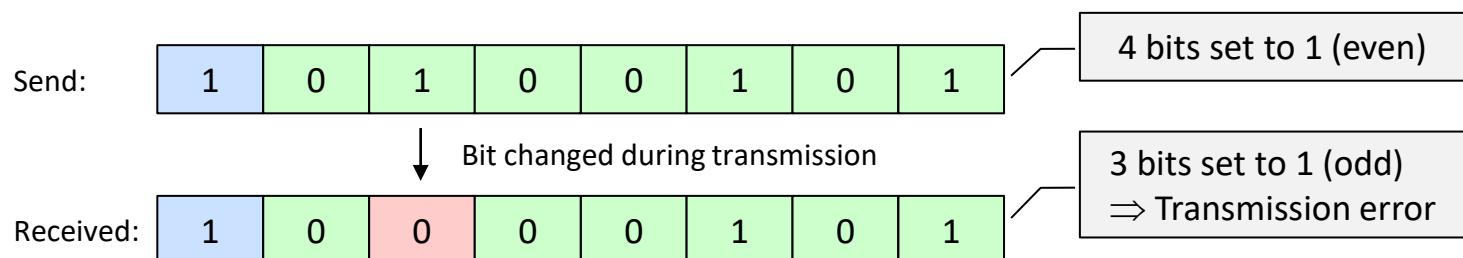
The issue:

- Data can get corrupted (i.e., changed) during transmission (e.g., via WLAN).



Parity bit:

- Add an additional bit to the data and set bit such that sum of 1's is even
- If sum of bits is *not even* at receiver, data was corrupted during transmission





Implement setting a parity bit for 7-bit data:

- The 7 lowest bit of an 8-bit byte (type *unsigned char*) contain user data.
- Set the high-order bit such that the overall number of 1's in the byte is even.

Sample solution:

```
unsigned char addParityMSB(unsigned char a)
{
    int count = 0;
    unsigned char mask = 1, maskMSB = 128;           Mask high-order bit: 10000000

    /* Count number of 1's in lowest 7 bits of bit pattern */
    for (int i = 0; i < 7; i++, mask *= 2)
        count += ((a & mask) == mask);            Move 1 by one bit to the left

    /* Set MSB to either 1 (count odd) or 0 (count even) */
    if ((count % 2) == 0)
        a &= ~maskMSB;                         Set MSB to 0
    else
        a |= maskMSB;                          Set MSB to 1

    return a;
}
```

Bit shifting

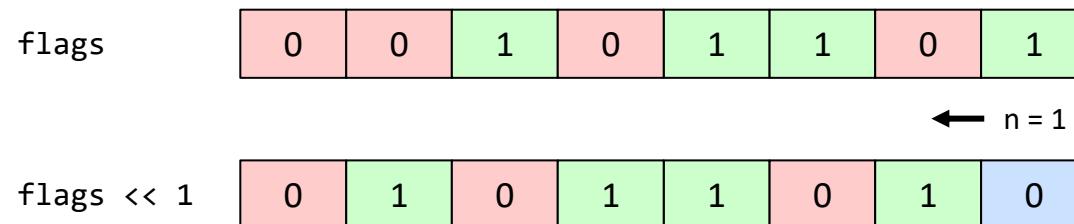
- Shift bit patterns to the left or right in memory
- Implement very fast multiplications and divisions by powers of 2

Shift to the left

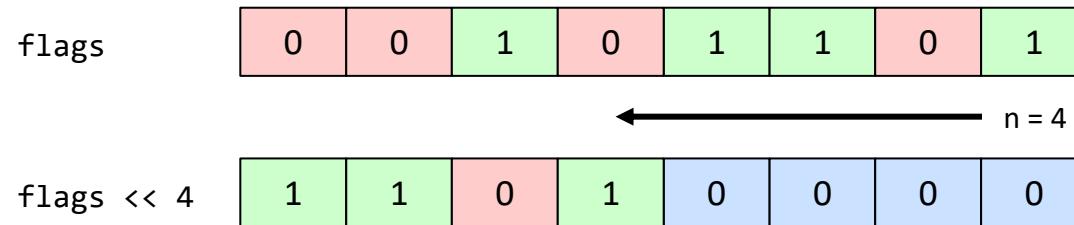
- Operator `<<` (and `<=` with assignment) shifts bits by n places to the left
- Vacant positions are filled with zeros.
- Bits moved “out of the area” are lost.

Examples:

- $n = 1$:



- $n = 4$:

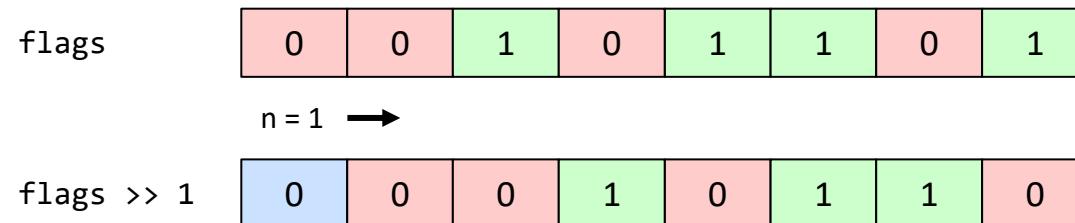


Shift to the right

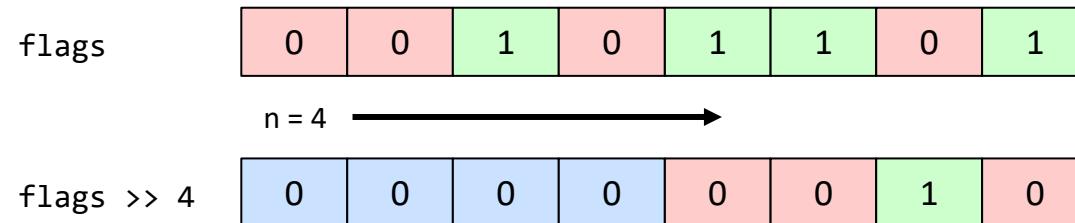
- Operator `>>` (and `>>=` with assignment) shifts bits by n places to the right
- Vacant bits 0 for unsigned and *machine-dependent* (e.g., value of sign bit) for signed types
- Bits moved “out of the area” are lost.

Examples:

- $n = 1$:



- $n = 4$:





Think about it:

- Bit shifting can be done very fast by processors.
- How to use shifting for fast mathematical multiplication and integer division?

Answer:

- Shifting bits n positions to the left \Rightarrow Multiplication with 2^n
- Shifting bits n positions to the right \Rightarrow Division by 2^n

Examples:

```
printf(" 2 * 357 = %4d\n", 357 << 1);
printf("    2^10 = %4d\n", 1 << 10);
printf("3000 / 4 = %4d\n", 3000 >> 2);
```



Digital images in RGBA color space:

- Each pixel represented by 4 values *red*, *green*, *blue*, and *alpha*
- Alpha represents transparency (i.e., background “shining through”)
- Each value is within 0 to 255 (i.e., represented by 8 bits).

Implementation:

- Each pixel shall be stored in a 32-bit unsigned integer.
- From highest to lowest byte: red, green, blue, and alpha

Example:

- RGBA = (11, 21, 61, 3)
- ⇒ Bit pattern: 0000 1011 0001 0101 0011 1101 0000 0011 (= 0xOB153D03)
- ⇒ Integer value: 185,941,251

Pixel value (32-bit)	0000 1011	0001 0101	0011 1101	0000 0011
	Red = 11	Green = 21	Blue = 61	Alpha = 3



Now it is your turn:

- Define a type *uint32* being a 32-bit unsigned integer.
- Implement a function to set a *uint32* pixel from the values for red, green, blue, and alpha.

Sample solution:

```
typedef unsigned uint32;

uint32 toRGB(A(uint32 red, uint32 green, uint32 blue, uint32 alpha)
{
    return (red << 24) | (green << 16) | (blue << 8) | alpha;
}
```

red << 24	0000 1011	0000 0000	0000 0000	0000 0000
green << 16	0000 0000	0001 0101	0000 0000	0000 0000
blue << 8	0000 0000	0000 0000	0011 1101	0000 0000
alpha	0000 0000	0000 0000	0000 0000	0000 0011

Exercise: RGBA color images



We also need functions to invert this step:

- Implement functions to decompose a *uint32* pixel into red, green, blue, and alpha.

Sample solution:

```
#define BYTE_MASK 0xff;
```

```
uint32 getRed(uint32 pixel)
{
    return pixel >> 24;
}
```

```
uint32 getGreen(uint32 pixel)
{
    return (pixel >> 16) & BYTE_MASK;
}
```

0000 0000	0000 0000	0000 0000	1111 1111
-----------	-----------	-----------	-----------

>> 24 →

0000 0000	0000 0000	0000 0000	0001 0101
-----------	-----------	-----------	-----------

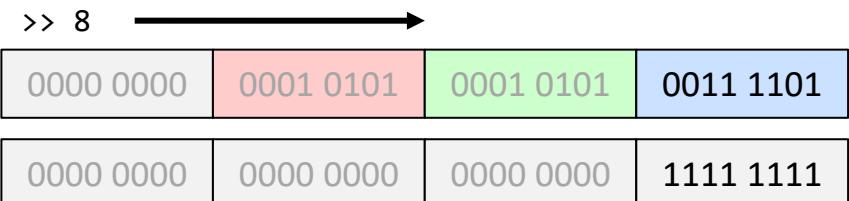
>> 16 →

0000 0000	0000 0000	0001 0101	0001 0101
0000 0000	0000 0000	0000 0000	1111 1111

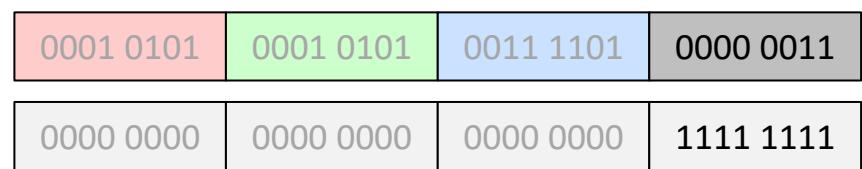
Exercise: RGBA color images

Sample solution (continued):

```
uint32 getBlue(uint32 pixel)
{
    return (pixel >> 8) & BYTE_MASK;
}
```



```
uint32 getAlpha(uint32 pixel)
{
    return pixel & BYTE_MASK;
}
```



Sample solution (continued):

```
int main(void)
{
    uint32 pixelLow = 0x00010203;      // RGBA = (0, 1, 2, 3)
    uint32 pixelHigh = 0xfcfdfeff;    // RGBA = (252, 253, 254, 255)

    /* Set pixel data */
    printf("Set RGBA pixel:\n(11, 21, 61, 3) -> %#.8x\n\n", toRGBA(11, 21, 61, 3));

    /* Decompose color channels */
    printf("Decompose RGBA pixel:\n");
    printf("%#.8x -> (%u, %u, %u, %u)\n", pixelLow, getRed(pixelLow),
           getGreen(pixelLow), getBlue(pixelLow), getAlpha(pixelLow));
    printf("%#.8x -> (%u, %u, %u, %u)\n", pixelHigh, getRed(pixelHigh),
           getGreen(pixelHigh), getBlue(pixelHigh), getAlpha(pixelHigh));

    getchar();
    return 0;
}
```

Software Construction 1 (IE1-SO1)

12. Projects & preprocessor



Lecture overview

Fundamentals



1. Data types



2. Flow control



3. Functions



4. Arrays (and strings)

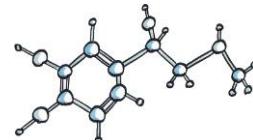
Advanced topics



5. Pointers



6. Memory management

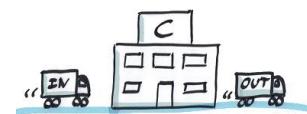


7. Structures



8. Lists and sorting

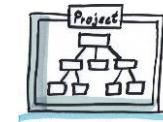
The next steps ...



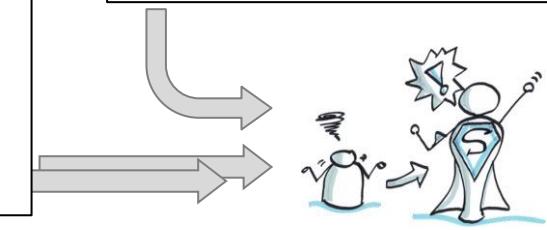
9. Input and output



10. Bit operations ("magic")



11. Project and preprocessor



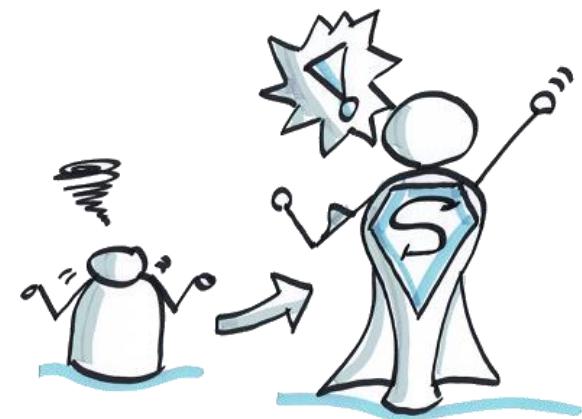
You!

● Location

What we want to achieve in this chapter

You shall become “slightly more of an engineer” – meaning you *can do* more!

- You *use macros* to implement function-like code that runs faster than corresponding functions.
- You *structure* projects in files dedicated to specific tasks or functionalities, by this increasing the readability of projects and re-use of functionality.
- And ... ***be proud of yourself***. We have covered all topics to be discussed in this course.



1. The define directive revisited
2. File inclusion & header files
3. Selected standard libraries

Preprocessor:

- Runs before compilation (⇒ The compiler “sees” the code *after* preprocessing.)
- Controlled by directives starting with #
- Preprocessor directives must be the first code in their line of code

What we have seen so far:

- Manifest constants (*#define*)
- File inclusion (*#include*)

In the following we will revisit these and introduce header files.

The define directive revisited

- Use human-readable constants
- Use fast function-like macros

- Define symbolic constants (*macros*)
- Syntax: `#define macro body`
- Preprocessor replaces occurrences of constants by corresponding value (*body*)

Examples:

```
#define SIZE 15
#define PI 3.14159265358979323846
#define NAME "HAW Hamburg"

int main(void)
{
    int histogram[SIZE] = { 0 };

    printf("2 * pi = %f\n", 2 * PI);
    printf("You are here: %s\n", NAME);

    getchar();
    return 0;
}
```

- Macros can act similar to functions with arguments.
- Provide comma-separated argument names in parentheses
- Parenthesize argument names in body and body as a whole to avoid mistakes

Example:

```
#define SQUARE(X) ((X) * (X))

int main(void)
{
    int a = 6;
    printf("6^2      = %d\n", SQUARE(a));
    printf("(7 - 1)^2 = %d\n", SQUARE(7 - 1));
    getchar();
    return 0;
}
```

Think about it:

- What is printed to the console?
- What would result using the incorrect macro `#define SQUARE(X) (X * X)`?

And another example:

```
#define MEAN(X,Y) (((X) + (Y)) / 2)

int main(void)
{
    int a = 6;
    printf("mean(%d, %d)      = %d\n", a, a + 3, MEAN(a, a + 3));
    printf("4.0 / mean(%d, %d) = %.1f\n", 1, 3, 4.0 / MEAN(1, 3));
    getchar();
    return 0;
}
```

Think again:

- What is printed to the console?
- What would result using the incorrect macro `#define MEAN(X,Y) ((X) + (Y)) / 2`?

File inclusion & header files

- Organize source code in several files
- Re-use functions in other projects

- The preprocessor replaces `#include` directives by contents of the specified file.
- We have been using this right from the beginning (e.g., `#include <stdio.h>`).

Be aware:

- It is allowed to use angle brackets `<...>` or double quotes “`...`”.
- The respective behavior is not part of the C standard, but compiler-dependent.

```
#include <name.h>
#include "name.h"
```

Typically:

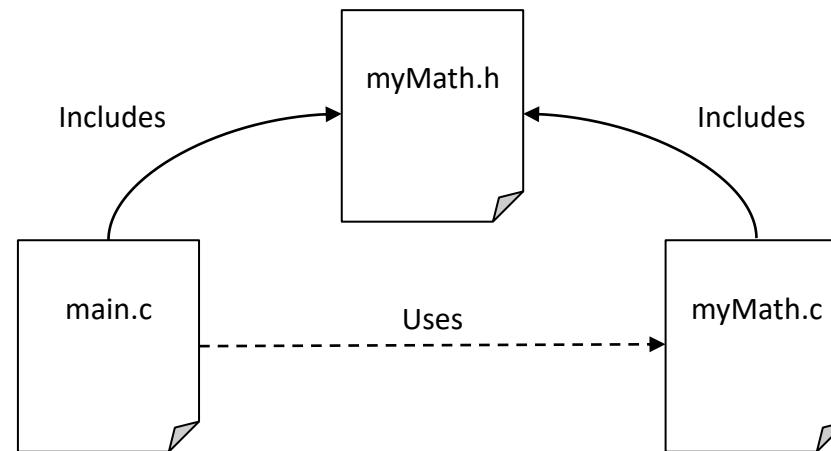
- Using `<...>` searches the system's or IDE's include directories for the file.
- Using “`...`” searches the working directory (e.g., location of source code), first.

Header files

- Do not contain executable code, but information needed to compile functions
- Examples: Manifest constants, structures, type definitions, function prototypes
- Use file extension *.h and include header files where required
- Functions prototypes in headers are implicitly extern (i.e., even without keyword `extern`).

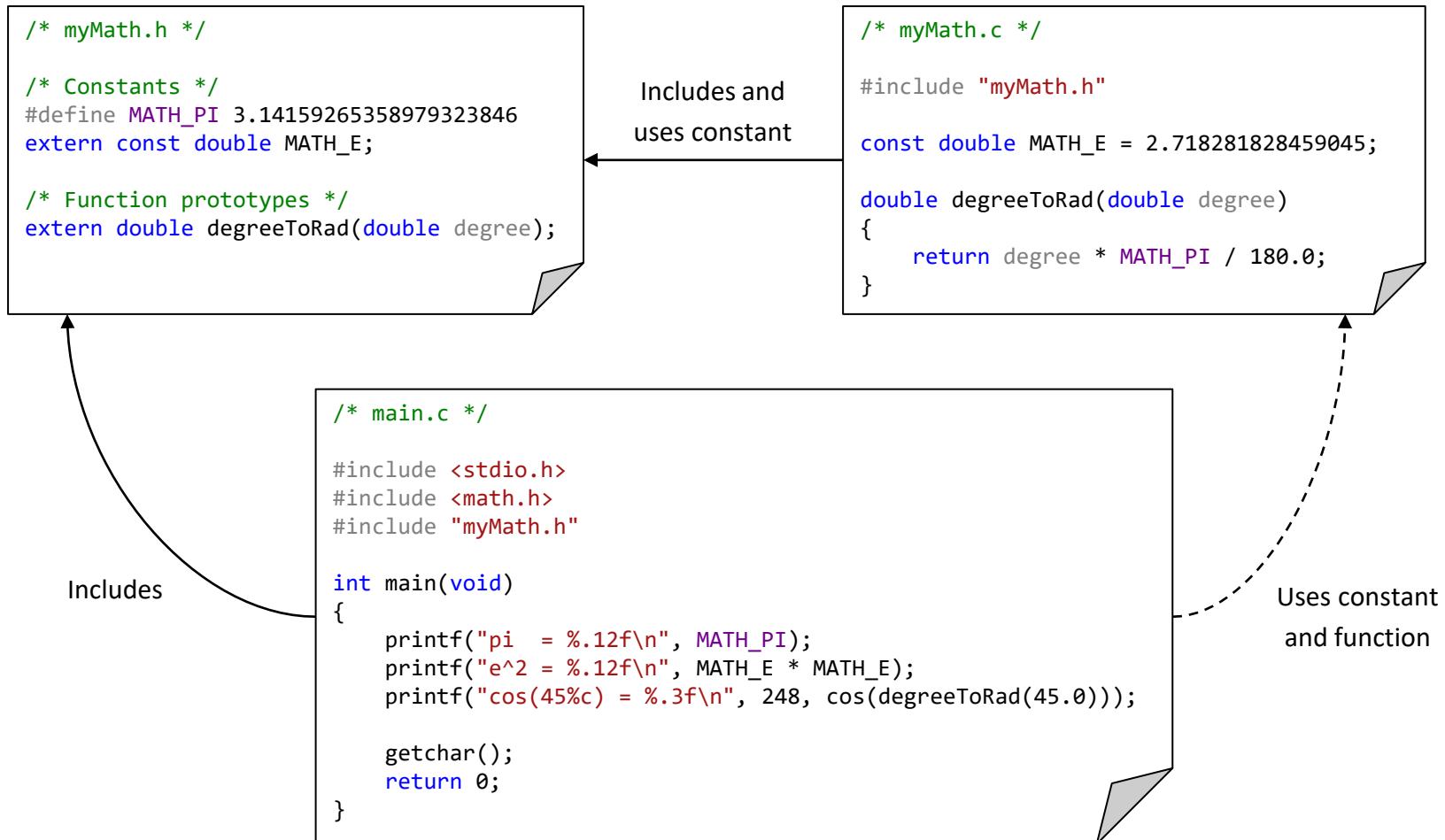
Example:

- Math functions and constant global variable in source file `myMath.c`
- Corresponding prototypes and manifest constant (`#define`) in header file `myMath.h`
- Files `myMath.c` and `main.c` using functions and constants include `myMath.h`



Header files

Example:

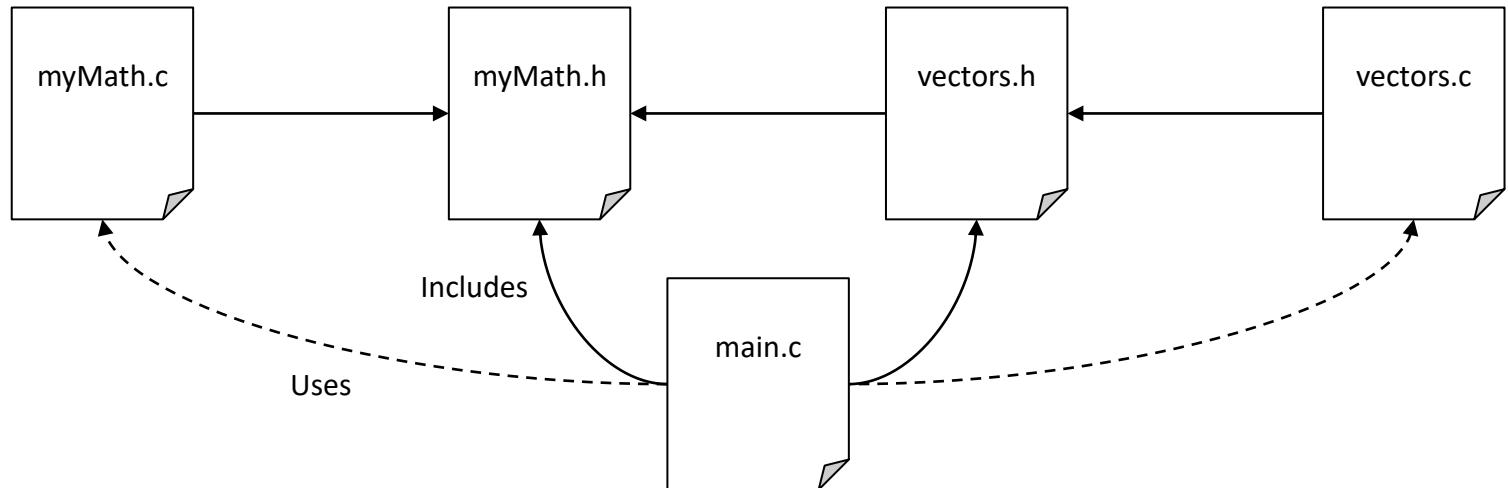


Header files



Finally, there is one more thing:

- What do you think of the dependencies depicted below?



Multiple inclusions:

- File *main.c* includes *myMath.h* twice (directly and indirectly by including *vector.h*).
 - Cannot avoid multiple inclusions in complex software
- ⇒ Let's see how to resolve this using the preprocessor ...

- Directives `#if`, `#else`, and `#endif` allow conditional compilation of “blocks”.
- The preprocessor removes source code where the condition is not fulfilled.

Example:

```
#define MARKETING_MODE 0          Set to 1 to have Marketing's position

int main(void)
{
    #if MARKETING_MODE == 1        Line removed before compilation
        printf("Our product is nice, shiny ... and the BEST in the world!!!\n");
    #else
        printf("Okay, personally I wouldn't buy it.\n");
    #endif

    getchar();
    return 0;
}
```

Now let's deal with multiple inclusions:

- Can use `#define` to define a symbolic name, even without a value
- Directives `#ifdef` and `#ifndef` exist for conditions “if defined” and “if not defined”.

Example for file *myMath.h*:

```
#ifndef MY_MATH_H_  
#define MY_MATH_H_
```

Defines MY_MATH_H_ at first inclusion.
Ignores following code at further inclusions.

```
/* Constants */  
#define MATH_PI 3.14159265358979323846  
extern const double MATH_E;
```

```
/* Function prototypes */  
extern double degreeToRad(double degree);  
extern double radToDegree(double rad);
```

Keyword *extern* not required

```
#endif
```

Selected standard libraries

Selected standard libraries

- Many frequently required functions are already implemented in standard libraries.
- Below are some selected files where you could start exploring the library:

Header	Description
assert.h	Check expressions (e.g., for debugging)
ctype.h	Character handling (e.g., transform to lowercase or uppercase letter, check for a character being a letter or digit)
math.h	Mathematical functions
stdio.h	Input and output functions
stdlib.h	General utilities (e.g., memory management, program termination, string to number conversion, random numbers, sorting)
string.h	String functions and copying memory ranges (e.g., to copy all values of an array with one function call)
time.h	Date and time functions

Software Construction 1 (IE1-SO1)

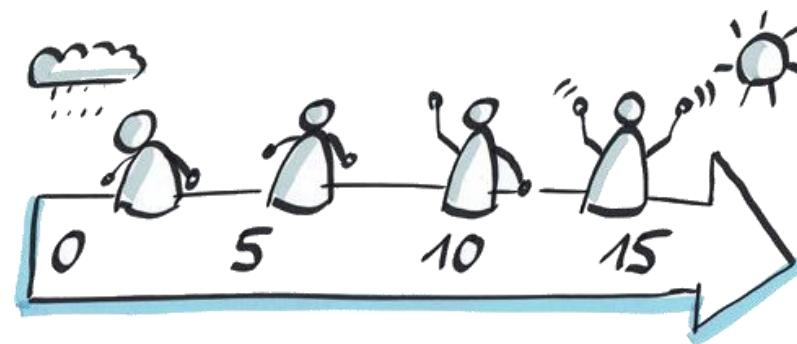
Coding style



- Consists of rules how to write and format source code (e.g., names, indents)
- Is not “correct” or “incorrect”, but uniform for all involved developers
 - ⇒ Improved maintainability, programming less error-prone

Note carefully:

- Being non-conform to the coding style can influence your grade in the lab exam!



Identifiers:

- All identifiers (“names”) must be in English.
- Choose meaningful identifiers.

Variables and functions:

- Identifiers for variables and functions begin with a small letter.
- Begin each new word with a capital letter (“*CamelCase*”), *not* using underscores:

```
int numberStudents, highScore;  
float gpsLatitude, gpsLongitude;
```

Constants:

- Type identifiers for constants in capital letters.
- Separate words by underscores

```
const float MATH_PI;
```

Opening and closing braces { ... }:

- Braces are always in a new line.
- Braces are followed by a new line.

Example:

```
int a = 4, b = 7;  
int max;
```

```
if (a > b)  
{  
    max = a;  
}  
else  
{  
    max = b;  
}
```

Indentation:

- Indent blocks and such by a tabulator per level.

Spaces:

- Separate preprocessor directives, global variables, prototypes, and functions by a line.
- Separate several functions by an empty line
- Separate logical blocks by an empty line.
- Add a blank (“space”) between, e.g., operands and operators.

The diagram shows a snippet of C code with various parts highlighted by red arrows and boxes:

```
#include <stdio.h>
int main(void)
{
    int a = 1;
    printf("Your lucky number is %d.\n", 10 * a - 3);
    return 0;
}
```

- A red arrow points to the first empty line after the include directive, with a box labeled "Empty line".
- A red arrow points to the empty line before the opening brace of the main function, with a box labeled "Empty line".
- A red arrow points to the space between the closing brace and the final return statement, with a box labeled "Blanks".
- A red arrow points to the four spaces preceding the first line of code inside the main function block, with a box labeled "Indentation".

Keep following structure in source code files:

Include directives

Prototypes

Global variables

main() function

Other functions

Software Construction 1 (IE1-SO1)

Simple image processing framework (Chooks)



1. Building the project
2. Using the application
3. Adding own image processing methods

Building the project

- Yes, *chook* means chicken, but *Chooks* ends with ‘s’ for a reason.
- The application uses the C++ *wxWidgets* library (e.g., for GUI and image file input).
- Sorry, you will have to install and build *wxWidgets* on your system.

Tested configuration:

- *wxWidgets 3.1.1* (installer file *wxMSW-3.1.1-Setup.exe*)
- Visual Studio Enterprise 2017
- Windows 10 (64-bit)

To avoid confusion:

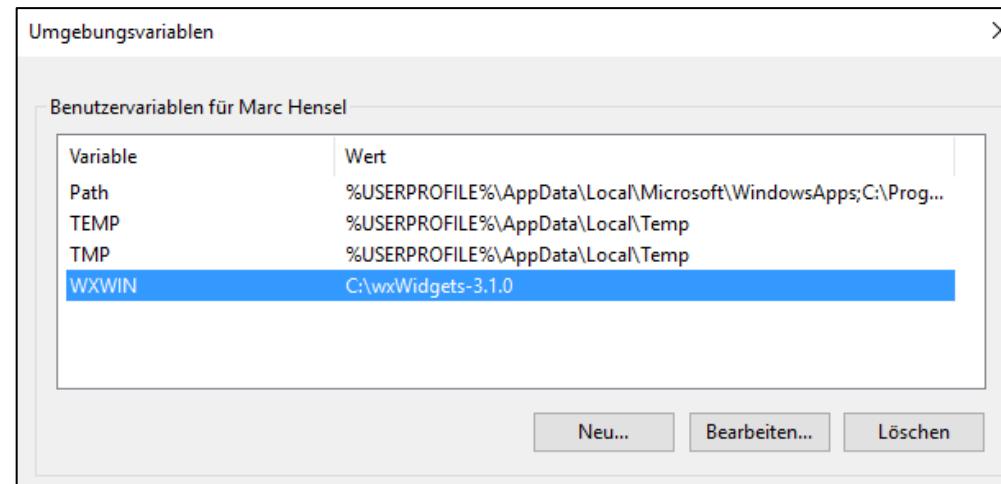
- The application is written in C++.
- You can provide image processing methods in standard C, though.
- Yes, *chook* really means chicken (in particular in Australia and New Zealand).

1. Installation:

- Download the Windows installer from <http://www.wxwidgets.org/downloads/>.
- Install the library directly to *C:* (e.g., *C:\wxWidgets-3.1.1*).

2. Setting the environment:

- Create an environment variable *WXWIN*. (Search Windows for “environment variable”.)
- The variable’s value must be wxWidgets’ root directory (e.g., “*C:\wxWidgets-3.1.1*”).
- Visual Studio uses this environment variable to find wxWidgets.



3. Building wxWidgets:

- Go to the sub directory *build\msw* (e.g., *C:\wxWidgets-3.1.1\build\msw*).
- Open the Visual Studio solution matching your version (e.g., *wx_vc15.sln* for VS2017).
- Select a configuration in the toolbar (e.g., *Debug* for *Win32* or *x86*) and build the solution.
- Build at least *Release* and *Debug* for *Win32* (*x86*).



Notes:

- Be aware that both, 32-bit and 64-bit systems, can run *Win32 (x86)* builds.
- Be patient, building takes some time ... have a coffee or two.

- Visual Studio projects must be configured to use wxWidgets.
- The following steps shall help you configure new Visual Studio projects.
- You can skip the steps for Chooks, as the project is already configured accordingly.

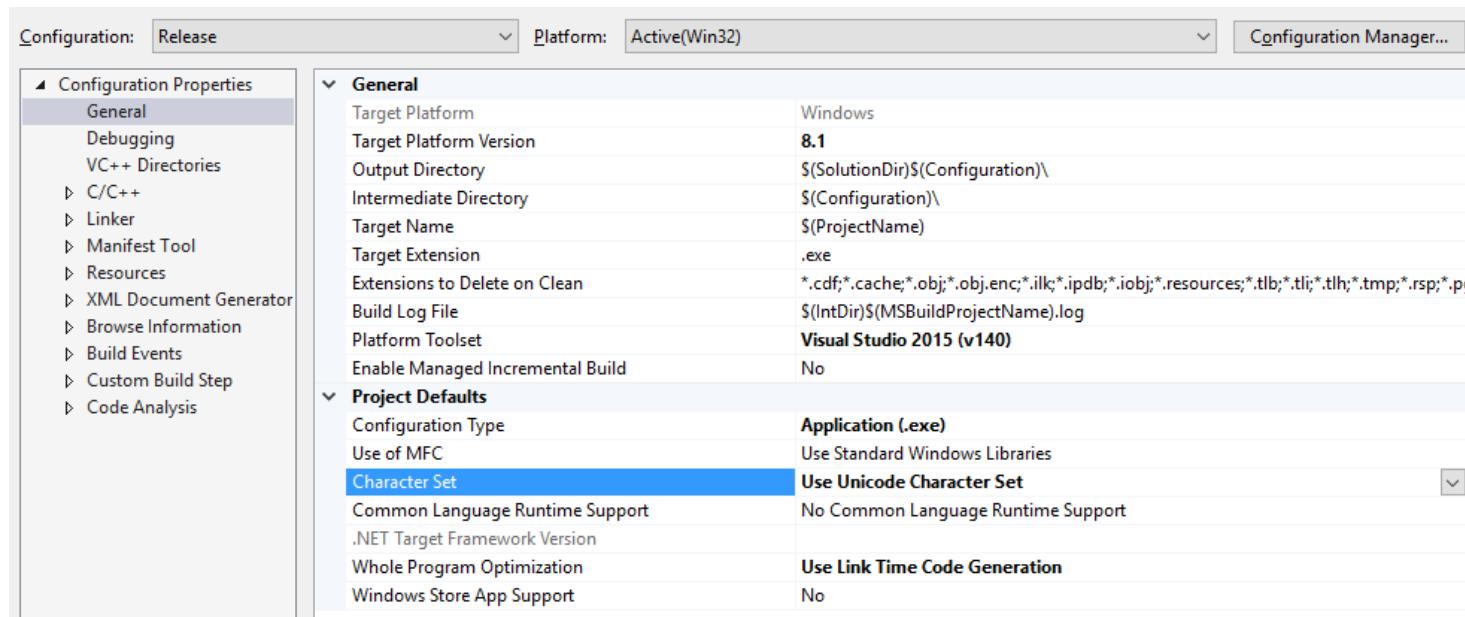
Creating new Visual Studio projects:

1. Create a new project of type *Visual C++ / General / Empty Project*.
2. Right-click the project in the Solution Explorer and select *Properties*.
3. Select the configuration at the top (e.g., *Debug for Win32*).

Follow the steps on the next slides and press the *Apply* button after every modification ...

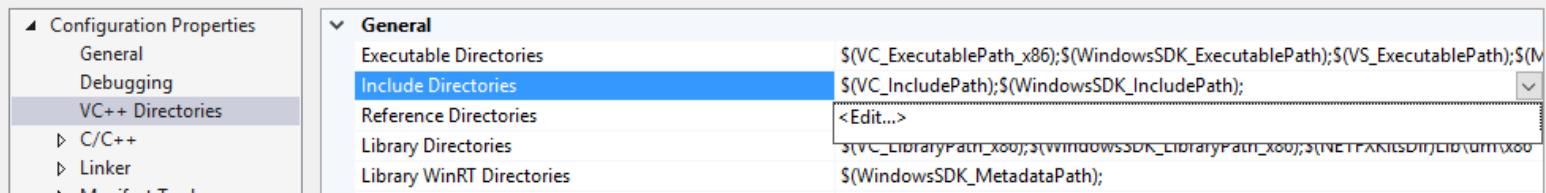
Setting the character set:

4. Set *Configuration Properties / General / Character Set* to “Use Unicode Character Set”.

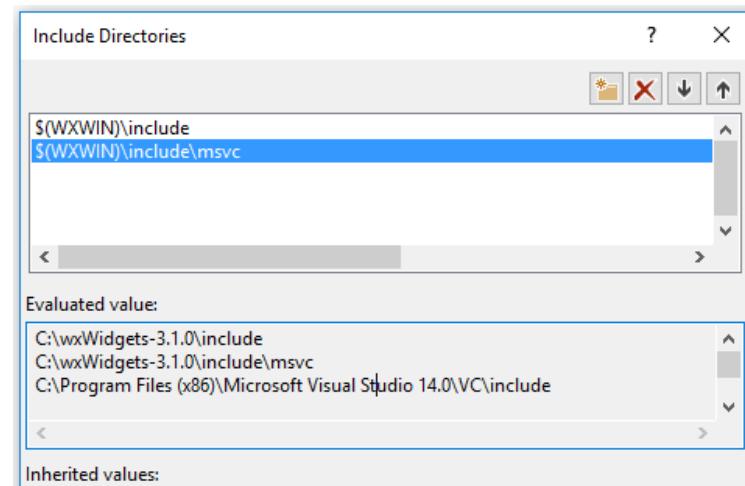


Adding the path to include files:

5. Select *Configuration Properties / VC++ Directories / Include Directories*.
6. Select <Edit ...> from the drop down list at the far right.

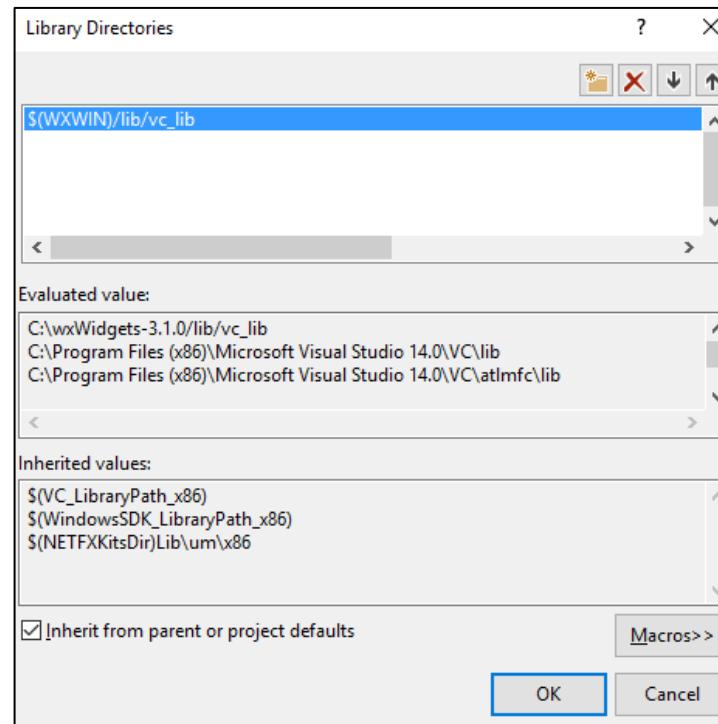


7. Click into the empty area and add `$(WXWIN)\include` and `$(WXWIN)\include\msvc`.

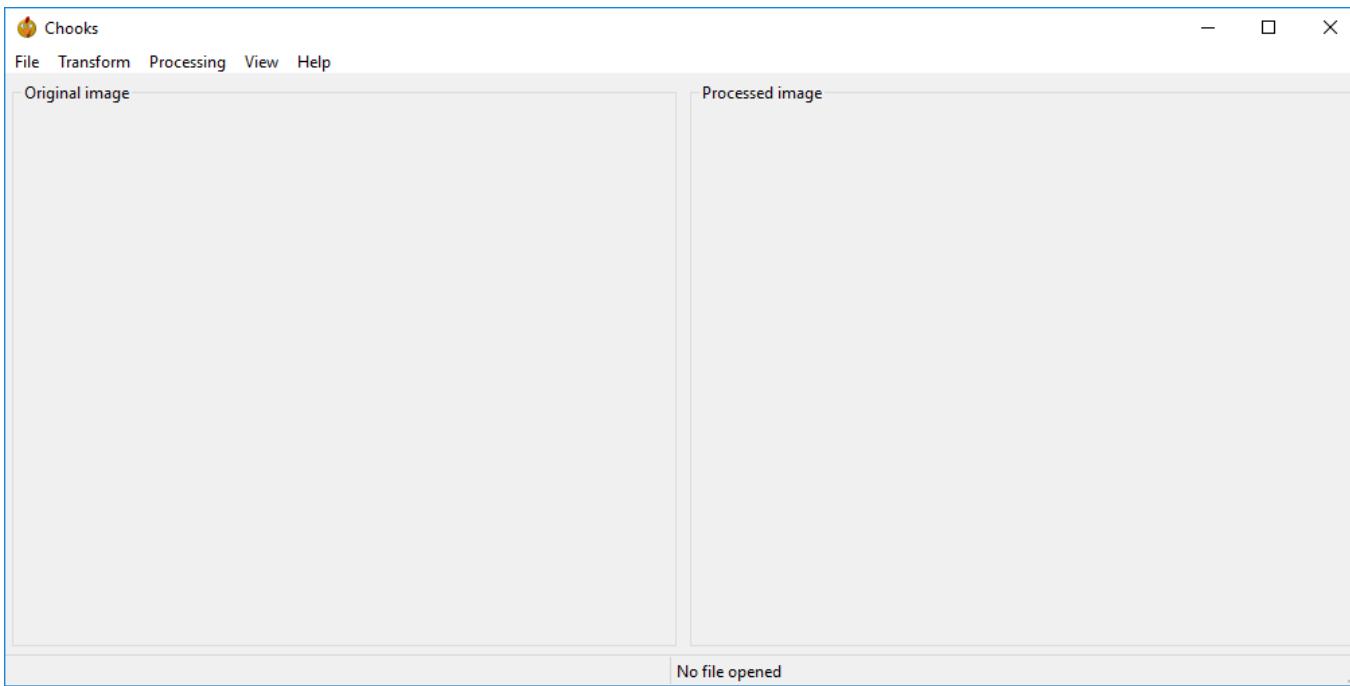


Adding the path to object files:

8. Select *Configuration Properties / VC++ Directories / Library Directories*.
9. Select <Edit ...> from the drop down list at the far right.
10. Add `$(WXWIN)/lib/vc_lib` (for Win32) or `$(WXWIN)/lib/vc_x64_lib` (for 64-bit x64).



- Open the provided Visual Studio solution.
- Select the configuration to build (e.g., *Debug* for *Win32*).
- Build the project and start the application by pressing *Ctrl-F5*.
- On success, following application window appears:



Using the application

Opening an image file

- Open a file using the *File* menu or drag & drop a file from the Explorer.
- The original image is displayed at the left, the processed image at the right.
- The processed image is initially the same as the unprocessed grayscale image.

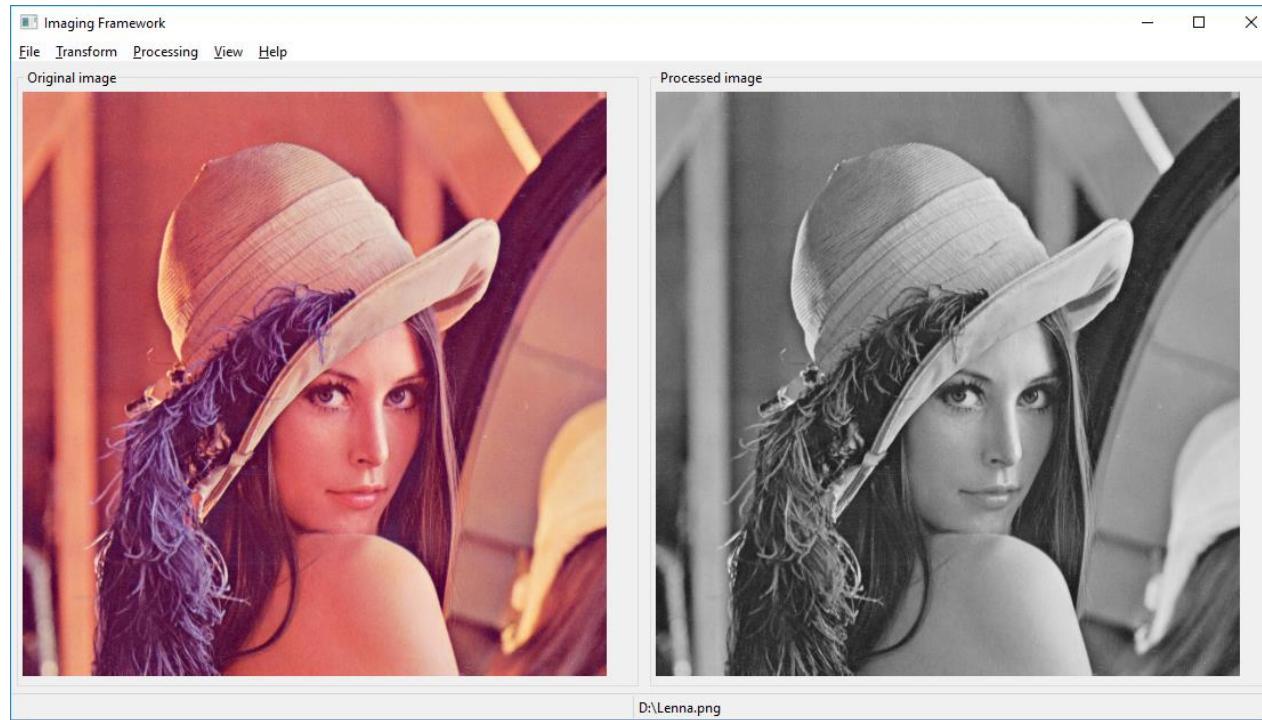


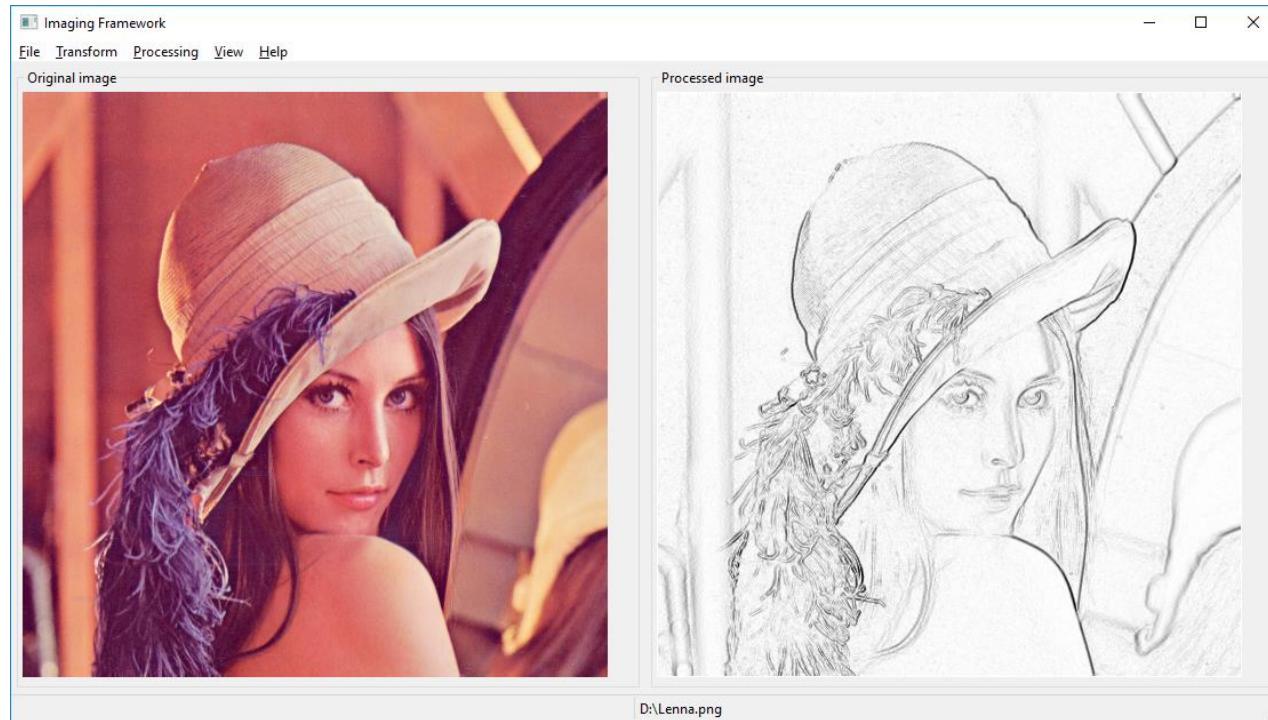
Image display

- Select *View / Original as grayscale* to toggle color \leftrightarrow grayscale of the unprocessed image.
- Use the *Transform* menu to rotate or mirror the images.
- Be aware of the keyboard shortcuts (stated at the corresponding menu items).



Image processing

- The *Processing* menu contains some simple image processing methods.
- Own methods can be added to the *Processing / Plugins* submenu (see next section).
- A menu item and keyboard shortcut (Ctrl-Z) exists to reset all image processing steps.

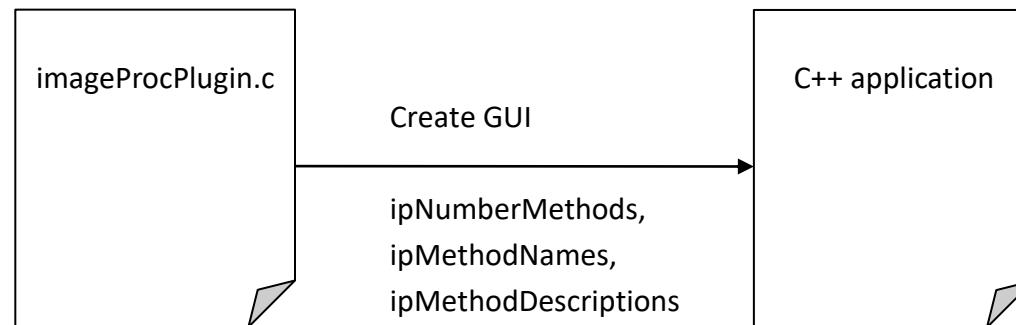


Adding own image processing methods

- Yes, the application is written in C++.
- You can add image processing methods in standard C, though.
- For this, the *only* source file to modify is *imageProcPlugin.c*.

The application uses *imageProcPlugin.c* to create the GUI:

- How many methods (*ipNumberMethods*)?
- Names of the methods to display in the processing menu (*ipMethodNames*)
- Descriptions of the methods to display in the status bar (*ipMethodDescriptions*)

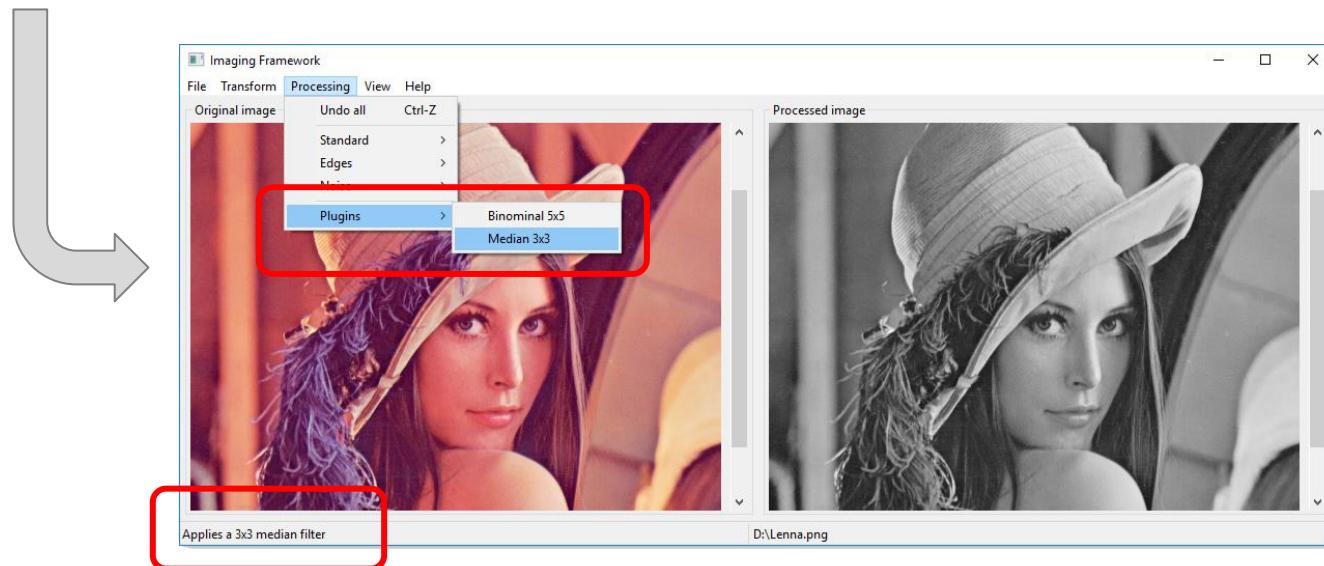


Example:

```
const int ipNumberMethods = 2;

const char * const ipMethodNames[] = {
    "Binomial 5x5",
    "Median 3x3" };

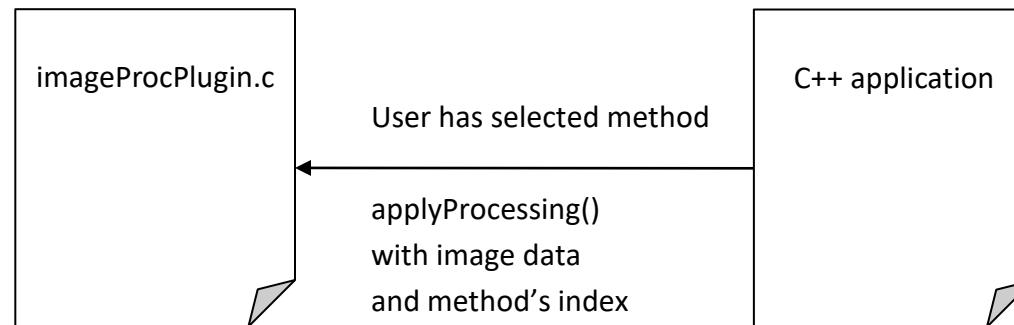
const char * const ipMethodDescriptions[] = {
    "Applies a 5x5 binomial filter",
    "Applies a 3x3 median filter" };
```



Apply processing

The application uses *imageProcPlugin.c* to apply image processing:

- Calls function *applyProcessing()* when a user selects a method in the *Plugins* menu
- Passes image data to the function (2-D array of pixel values, image width and height)
- Passes to function, *which* of the methods has been selected (integer index)



Example:

```
void applyProcessing(unsigned char** grayData, int width, int height, int index)
{
    switch (index)
    {
        case 0:
            // Source code to apply for binomial filter ...
            break;
        case 1:
            // Source code to apply for median filter ...
            break;
    }
}
```

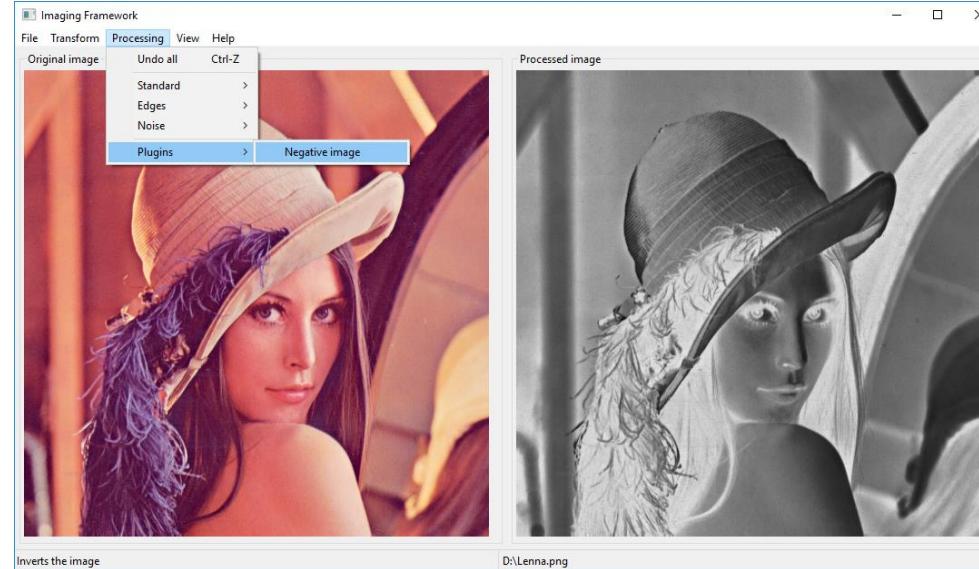
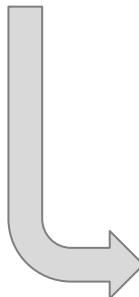
Notes:

- The image data is passed as 2D-array of 8-bit *unsigned char*.
- Each value represents a grayscale pixel in 0 to 255.
- When *applyProcessing()* returns, the application sets the processed image to the pixel values in *grayData*.

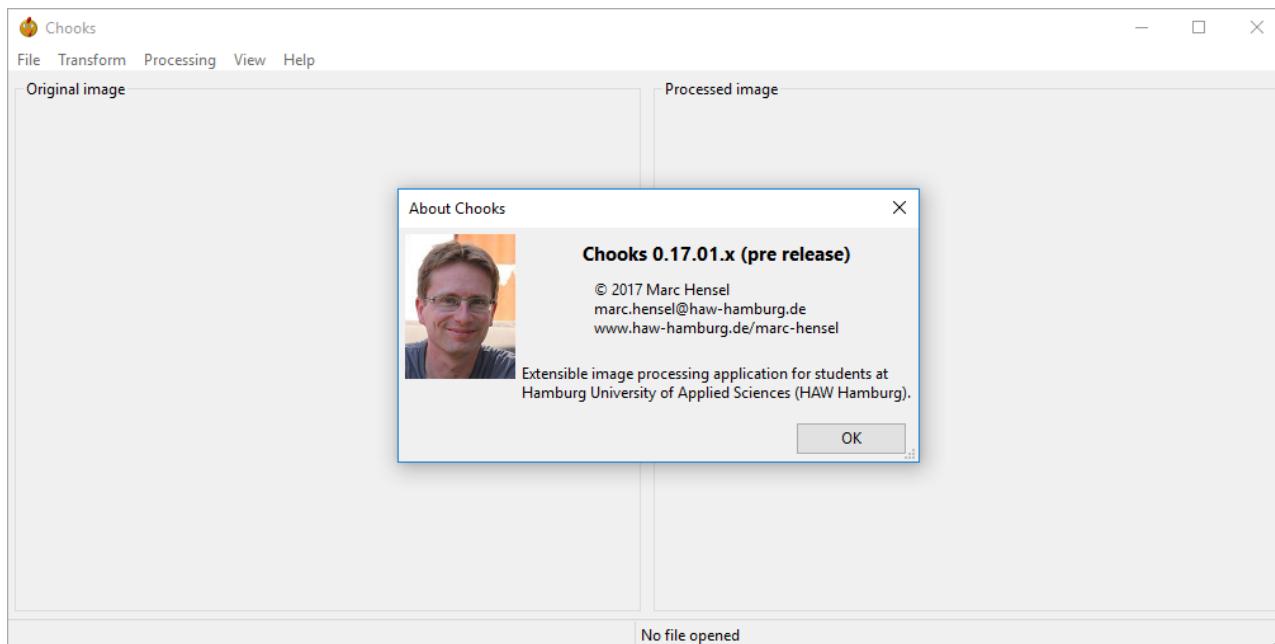
Example: Simple image processing method

```
const int ipNumberMethods = 1;
const char * const ipMethodNames[] = { "Negative image" };
const char * const ipMethodDescriptions[] = { "Inverts the image" };

void applyProcessing(unsigned char** grayData, int width, int height, int index)
{
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
            grayData[y][x] = 255 - grayData[y][x];
    }
}
```



Now it is your turn



Who said software development isn't fun?

Enjoy! 