Universität Zürich^UZH

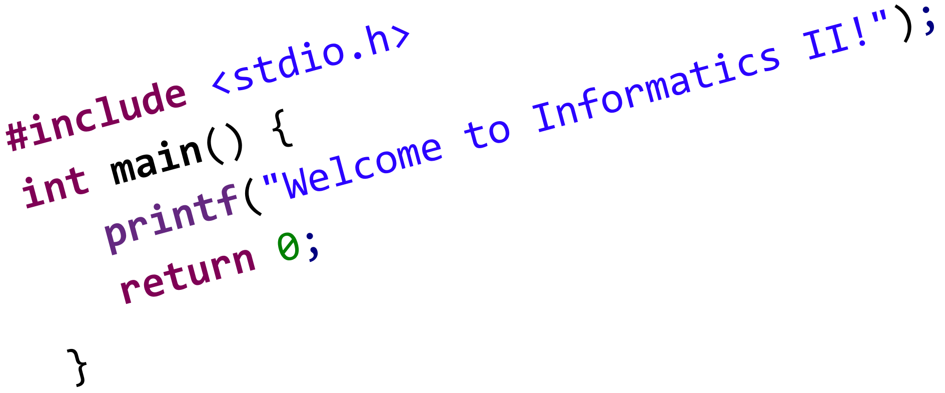
Institut für Informatik

Informatics II

Tutorial Session 1

Thursday, 23rd of February 2022

Introduction to the C Language



```c
#include <stdio.h>
int main() {
    printf("Welcome to Informatics II!");
    return 0;

}
```

14.00 – 15.45

BIN 0.B.06

Agenda

– Administrivia and Expectations

– Introduction to the C Language

Goals for Today

In this introduction to the C language, you should learn at least:

- What the main characteristics of C language are and how it differentiates itself to Python
- Data types in C and the dangers which come with them
- Arrays in C and how to work with them
- Functions in C and how to work with them
- Simple input and output from / to console in C

- Goals: what you will do / learn today
- Pseudo code vs. C code
- How to work with C (repl.it)

Administrivia and Expectations

- Contact Information
- Remote Teaching: Philosophy and Rules
- Structure of Labs
- Your Expectations
- Lab Schedule Overview
- Exams and Grading

Contact Information

Christoph Vogel

christoph.vogel@uzh.ch

If you have questions:

- Preferably ask me right away during the exercise sessions. Just interrupt me at any time.
- Use the OLAT forum.
- Send me an email.

Language, Materials, Recordings

- Teaching language in the tutorials will be English (unless everybody understands German). You may always ask questions in German; don’t hesitate to get to me with questions.

- All materials from the lab (including these slides) will be provided on OLAT (materials folder).

- This lab is not recorded, participation is only possible on-site. Lab 6 on Fridays is done online and will be recorded.

«Turmbau zu Babel» by Pieter Bruegel the Elder (c. 1563)
Kunsthistorisches Museum, Vienna

Bits and Pieces of This Course

- No mandatory attendance in lectures nor tutorials (but missing them is at your own peril).

- No mandatory hand-in of exercises, no correction and no grading of exercises. Sample solutions for all exercises will be provided at the end of the respective week (usually on Fridays after the last lab).

- There will be no midterms this year, your grade will depend entirely on your performance in the final exam. The final exam will be an open book online remote exam (Wednesday, 1st of June 2022, 14.00 – 15.30 h, 90 minutes).

Final exam
100% of your grade

Labs
13 sessions
voluntary, recommended

Exercises
12 (+1) instances
voluntary, recommended

Lab Schedule Overview

Date	No.	Exercise	Content
Wed, 23.02.	1	(Ex0)	Introduction to C
Wed, 02.03.	2	Ex1	Basic Sorting
Wed, 09.03.	3	Ex2	Recursion
Wed, 16.03.	4	Ex3	Tba
Wed, 23.03.	5	Ex4	Tba
Wed, 30.03.	6	Ex5	Tba
Wed, 06.04.	7	Ex6	Tba
Wed, 13.04.	8	Ex7	Tba
Wed, 20.04.	–		Spring Break, *no lab*
Wed, 27.04.	9	Ex8	Tba
Wed, 04.05.	10	Ex9	Tba
Wed, 11.05.	11	Ex10	Tba
Wed, 18.05.	12	Ex11	Tba
Wed, 25.05.	13	Ex12	Tba

About this Course: Contents

- The contents of this course are (in parts) hard. It’s perfectly normal to struggle. If you don’t struggle, you’re probably not trying hard enough (or you’re a genius).

- The exam will probably be hard, too. Usually you will not need a high percentage of reachable points to pass (though it’s relatively easy to reach a low score, unfortunately). To get points you really need to understand stuff. Memorizing things or superficial/cursory knowledge is not sufficient.

- The wealth of materials presented in the lecture is quite big. Try to stay up to date. If you get lost, it will be painful or impossible to catch up.

About this Course: Topics

- The main focus of this course are algorithms and data structures. For passing this course it is crucial that you focus on understanding really them thoroughly. For that, it is imperative that you work on the exercises as well as sample tasks from earlier exams. Just attending the lectures alone will most probably not be sufficient to pass the exam, neither will just looking at the solutions.

- Learning the C programming language is a secondary goal of this course. Programming is also a good way to get an understanding of the algorithms discussed in class.

- In exams, you will be required to be able to provide solutions as C or as pseudo code (but you're probably not able to choose). This year, a stronger emphasis will be put on writing and understanding pseudo code. You need to be able to read and write both C code and pseudo code and both representations might show up in the final exam.

Previous Knowledge Expected From You

I will assume that you already know basic concepts of programming (e.g. what a for loop and a while loop is, what an if/else statement does, (what a variable is), what literals and variable scopes are etc. pp.). to the extent of what you should have learned in Informatics I (or any other introductory programming course).

If this is not the case, please speak to me.

SRF SPORTLIVE
SLOPESTYLE FINAL
7

How To Pass This Course

- Practicing is key! Spend several hours a week (sic!) coding in C.
  - I might be able to help you understand topics, I might be able to show you how to approach problems. But I most certainly can magically induce practice into your brain. Most of this practice needs to happen outside of this room.
  - Being able to read C code does not mean one can write C code.
  - Only looking at solutions is not sufficient. You need to work on them yourself.
  - Understanding solutions is not sufficient. You need to be able to produce them from scratch yourself.
- Don’t fall behind. Start coding in C today.

- A dire warning:

DO NOT POSTPONE LEARNING TO A LATER TIME. IT WILL NOT WORK.

My Thoughts and Goals Regarding the Tutorials

Since attendance is not mandatory, it is completely up to you whether you choose to be here.

In my view, the tutorials should be an opportunity to…

- practice, strengthen and deepen the understanding of the topics from the lecture,
- get a feedback and check on your progress of learning,
- meet other people, connect, share, discuss, …

My goals:

- I'll strive to create true additional value and that it will be worth your time attending the lab sessions.
- I'll try to design the tutorials such that you will never leave the session and say that it was a waste of your time. If you think that I did not achieve this, please speak to me and say what I should change.

Your Expectations

https://www.klicker.uzh.ch/algodat

Tentative Structure of Exercise Sessions

1. Intro, news and administrative stuff
2. Discussion of previous exercise
3. Recitation of topics of the current exercise
4. Short coffee break
5. Additional examples and exercises on current lecture
6. Summary and open questions

Introduction to C – Lab

Some Remarks on Tooling

- Make sure that you have a possibility to write C code. Writing C code yourself is – with all frustrations it comes with – the only way you really will learn C.
- Try to spend most of your time with actual coding. If you’re not happy with a particular IDE or setting, just take another one.

Tooling: How to work with C?

There are two main ways:

- Text editor and command line interface (CLI)
- Integrated Development Environment (IDE)
- (Online IDE)

- Windows: Use Windows Subsystem for Linux (WSL2) to compile and run C code.

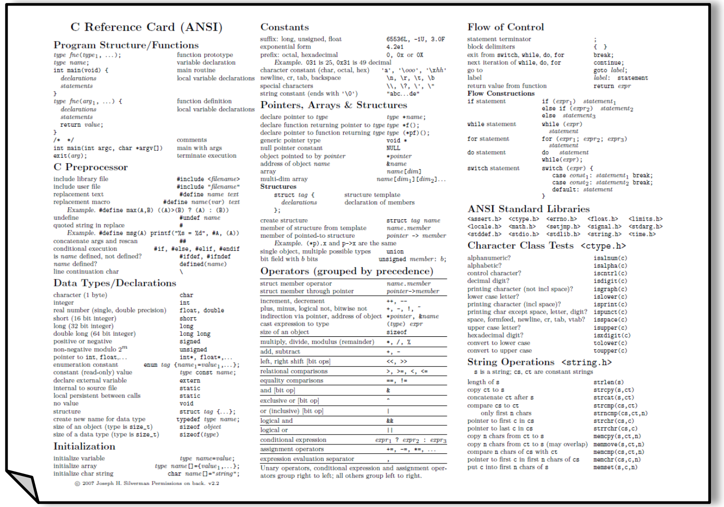
Hints and suggestions can be found on OLAT in the wiki of our lab.

Resources on C Programming

- C Reference Card by Joseph H. Silverman

- There are many, many books on C. The most famous one is probably the work by the creators of C (the «Bible of C»):
  - «The C Programming Language» by Brian W. Kernighan and Dennis M. Ritchie

- There are also many, many online resources with tutorials, interactive exercises etc. There are also browser-based compilers which can could use if other approaches fail.



C Reference Card (ANSI)

Program Structure/Functions

`type fnc(type1, ...);`	function prototype
`type name;`	variable declaration
`int main(void) {`	main routine
`  declarations`	local variable declarations
`  statements`	
`}`	
`type fnc(arg1, ...) {`	function definition
`  declarations`	local variable declarations
`  statements`	
`  return value;`	
`}`	
`/*  */`	comments
`int main(int argc, char *argv[])`	main with args
`exit(arg);`	terminate execution

C Preprocessor

include library file	`#include <filename>`
include user file	`#include "filename"`
replacement text	`#define name text`
replacement macro	`#define name(var) text`
Example. `#define max(A,B) ((A)>(B) ? (A) : (B))`	
undefine	`#undef name`
quoted string in replace	`#`
Example. `#define msg(A) printf("%s = %d", #A, (A))`	
concatenate args and rescan	`##`
conditional execution	`#if, #else, #elif, #endif`
is *name* defined, not defined?	`#ifdef, #ifndef`
name defined?	`defined(name)`
line continuation char	`\`

Data Types/Declarations

character (1 byte)	`char`
integer	`int`
real number (single, double precision)	`float, double`
short (16 bit integer)	`short`
long (32 bit integer)	`long`
double long (64 bit integer)	`long long`
positive or negative	`signed`
non-negative modulo 2^m	`unsigned`
pointer to `int`, `float`,...	`int*, float*,...`
enumeration constant	`enum tag {name1=value1,...};`
constant (read-only) value	`type const name;`
declare external variable	`extern`
internal to source file	`static`
local persistent between calls	`static`
no value	`void`
structure	`struct tag {...};`
create new name for data type	`typedef type name;`
size of an object (type is `size_t`)	`sizeof object`
size of a data type (type is `size_t`)	`sizeof(type)`

Initialization

initialize variable	`type name=value;`
initialize array	`type name[]={value1,...};`
initialize char string	`char name[]="string";`

© 2007 Joseph H. Silverman Permissions on back. v2.2

Constants

suffix: long, unsigned, float	`65536L, -1U, 3.0F`
exponential form	`4.2e1`
prefix: octal, hexadecimal	`0, 0x or 0X`
Example. `031` is 25, `0x31` is 49 decimal	
character constant (char, octal, hex)	`'a', '\ooo', '\xhh'`
newline, cr, tab, backspace	`\n, \r, \t, \b`
special characters	`\\, \?, \', \"`
string constant (ends with `'\0'`)	`"abc...de"`

Pointers, Arrays & Structures

declare pointer to *type*	`type *name;`
declare function returning pointer to *type*	`type *f();`
declare pointer to function returning *type*	`type (*pf)();`
generic pointer type	`void *`
null pointer constant	`NULL`
object pointed to by *pointer*	`*pointer`
address of object *name*	`&name`
array	`name[dim]`
multi-dim array	`name[dim1][dim2]...`

Structures

`struct tag {`	structure template
`  declarations`	declaration of members
`};`	

create structure	`struct tag name`
member of structure from template	`name.member`
member of pointed-to structure	`pointer -> member`
Example. `(*p).x` and `p->x` are the same	
single object, multiple possible types	`union`
bit field with *b* bits	`unsigned member: b;`

Operators (grouped by precedence)

struct member operator	`name.member`
struct member through pointer	`pointer->member`
increment, decrement	`++, --`
plus, minus, logical not, bitwise not	`+, -, !, ~`
indirection via pointer, address of object	`*pointer, &name`
cast expression to type	`(type) expr`
size of an object	`sizeof`
multiply, divide, modulus (remainder)	`*, /, %`
add, subtract	`+, -`
left, right shift [bit ops]	`<<, >>`
relational comparisons	`>, >=, <, <=`
equality comparisons	`==, !=`
and [bit op]	`&`
exclusive or [bit op]	`^`
or (inclusive) [bit op]	`|`
logical and	`&&`
logical or	`||`
conditional expression	`expr1 ? expr2 : expr3`
assignment operators	`+=, -=, *=, ...`
expression evaluation separator	`,`

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	`;`
block delimiters	`{ }`
exit from `switch`, `while`, `do`, `for`	`break;`
next iteration of `while`, `do`, `for`	`continue;`
go to	`goto label;`
label	`label: statement`
return value from function	`return expr`

Flow Constructions

```
if statement        if (expr1) statement1
                    else if (expr2) statement2
                    else statement3
while statement     while (expr)
                      statement
for statement       for (expr1; expr2; expr3)
                      statement
do statement        do  statement
                    while(expr);
switch statement    switch (expr) {
                       case const1: statement1 break;
                       case const2: statement2 break;
                       default: statement
                    }
```

ANSI Standard Libraries

`<assert.h>`	`<ctype.h>`	`<errno.h>`	`<float.h>`	`<limits.h>`
`<locale.h>`	`<math.h>`	`<setjmp.h>`	`<signal.h>`	`<stdarg.h>`
`<stddef.h>`	`<stdio.h>`	`<stdlib.h>`	`<string.h>`	`<time.h>`

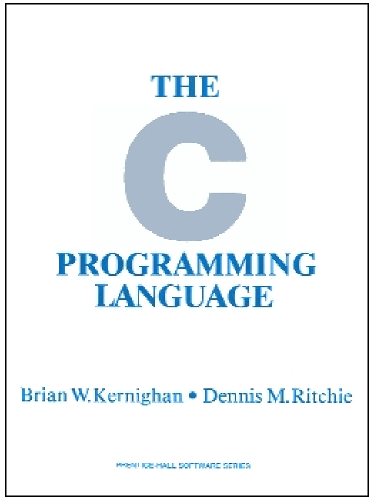
Character Class Tests `<ctype.h>`

alphanumeric?	`isalnum(c)`
alphabetic?	`isalpha(c)`
control character?	`iscntrl(c)`
decimal digit?	`isdigit(c)`
printing character (not incl space)?	`isgraph(c)`
lower case letter?	`islower(c)`
printing character (incl space)?	`isprint(c)`
printing char except space, letter, digit?	`ispunct(c)`
space, formfeed, newline, cr, tab, vtab?	`isspace(c)`
upper case letter?	`isupper(c)`
hexadecimal digit?	`isxdigit(c)`
convert to lower case	`tolower(c)`
convert to upper case	`toupper(c)`

String Operations `<string.h>`

`s` is a string; `cs`, `ct` are constant strings

length of `s`	`strlen(s)`
copy `ct` to `s`	`strcpy(s,ct)`
concatenate `ct` after `s`	`strcat(s,ct)`
compare `cs` to `ct`	`strcmp(cs,ct)`
only first `n` chars	`strncmp(cs,ct,n)`
pointer to first `c` in `cs`	`strchr(cs,c)`
pointer to last `c` in `cs`	`strrchr(cs,c)`
copy `n` chars from `ct` to `s`	`memcpy(s,ct,n)`
copy `n` chars from `ct` to `s` (may overlap)	`memmove(s,ct,n)`
compare `n` chars of `cs` with `ct`	`memcmp(cs,ct,n)`
pointer to first `c` in first `n` chars of `cs`	`memchr(cs,c,n)`
put `c` into first `n` chars of `s`	`memset(s,c,n)`



THE C PROGRAMMING LANGUAGE

Brian W.Kernighan • Dennis M.Ritchie

PRENTICE HALL SOFTWARE SERIES

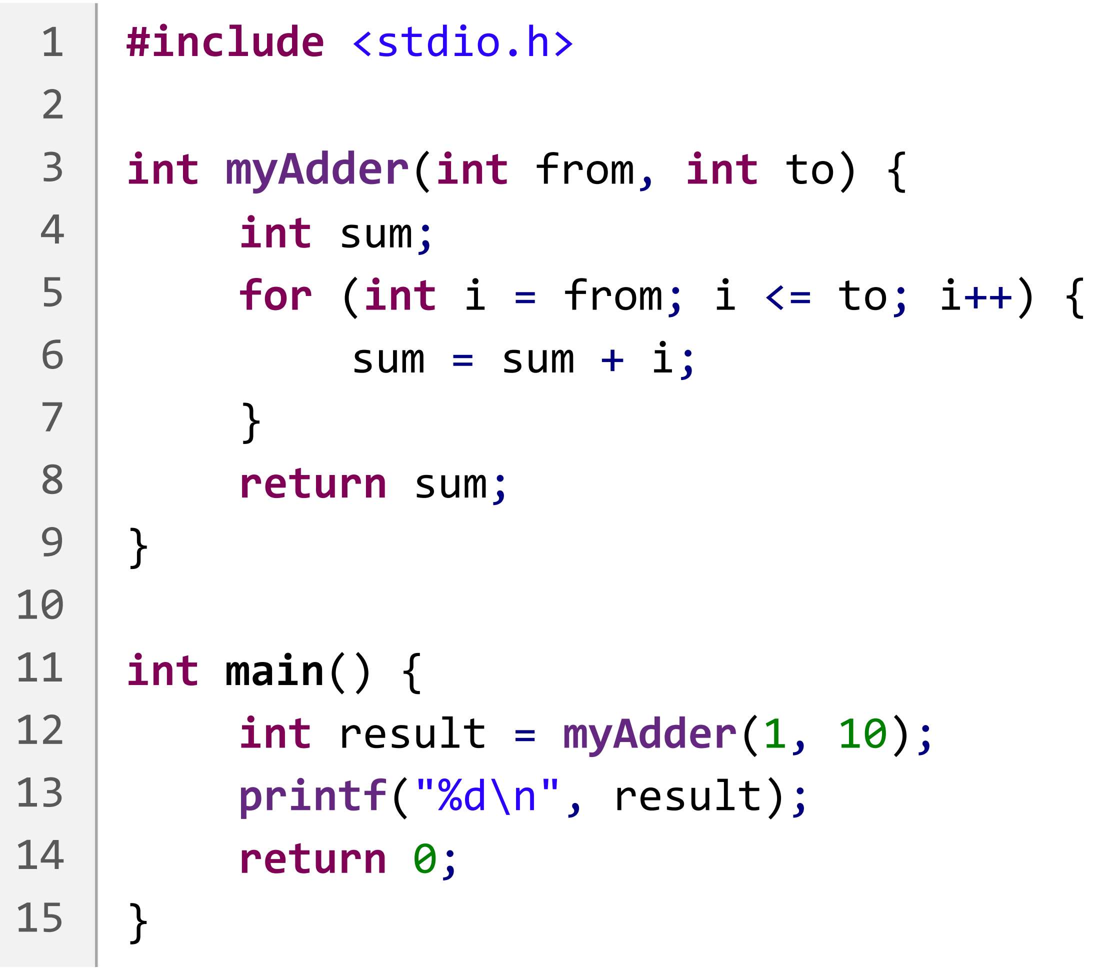
Example Problem

Tasks:

a) If you are able to run C code on your computer: Please copy and paste the following code snippet and execute it.

b) What is the result printed to the console by the program?

c) Think about what the code does and whether the result meets your expectations.



```c
#include <stdio.h>

int myAdder(int from, int to) {
    int sum;
    for (int i = from; i <= to; i++) {
        sum = sum + i;
    }
    return sum;
}

int main() {
    int result = myAdder(1, 10);
    printf("%d\n", result);
    return 0;
}
```

Additional Coding Exercise: Run-Length Encoding (RLE)

Write a simple run-length encoding (RLE) function in C which operates on a given string.

For example, for the input "AAAAABBBBCCAAABDDDDDCC", it should output "A5B4C2A3BD5C2".

Introduction to C – Basics

- C as a Programming Language
- Python vs. C: General Overview
- Example C Program, Python vs. C: Syntax
- How to work with C?
- Compiler vs. Interpreter
- Compiling Process
- Memory in C

Note: Due to time constraints, it is not possible to provide a complete and/or systematic introduction to the C language during the tutorial sessions. This is only a very brief first introduction to get us started. I will try to successively provide the necessary knowledge as it is needed.

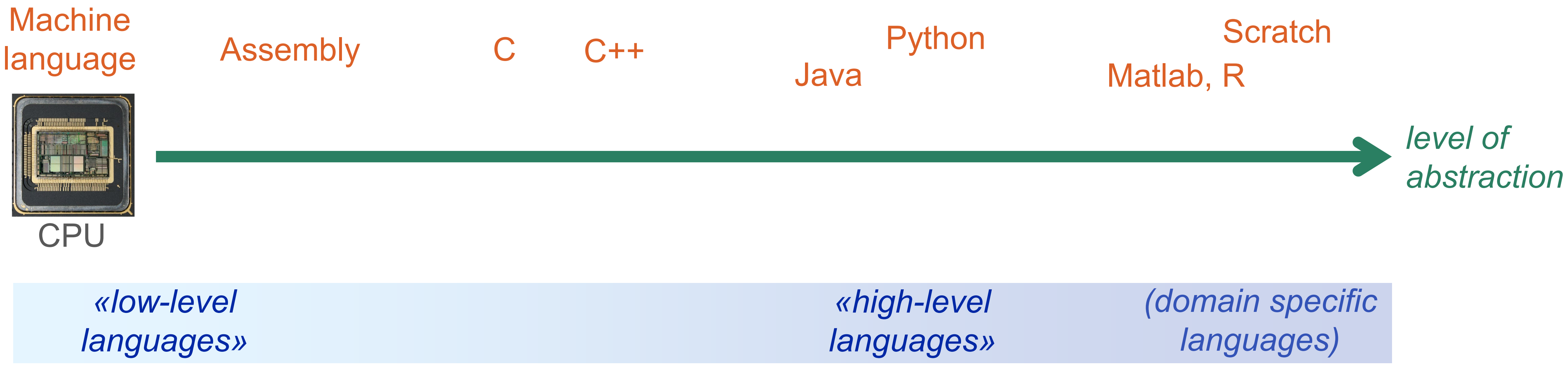
Learning Objectives

In this introduction to the C language, you should learn:

- How the compiler works and turns a piece of C code into a sequence of zeros and ones which can be fed to the CPU an will instruct it to do what has been written in abstract C language
- What the difference between a compiler and an interpreter is
- What the main characteristics of C language are and how it differentiates itself to Python
- Why it is important to have a concept of physical memory in mind when programming in C
- Data types in C and the dangers which come with them
- Arrays in C and how to work with them
- Functions in C and how to work with them
- Simple input and output from / to console in C

C as a Programming Language

C is a relatively low-level language. This comes with performance benefits.



Machine language

Assembly

C

C++

Java

Python

Matlab, R

Scratch

CPU

level of abstraction

«low-level languages»

«high-level languages»

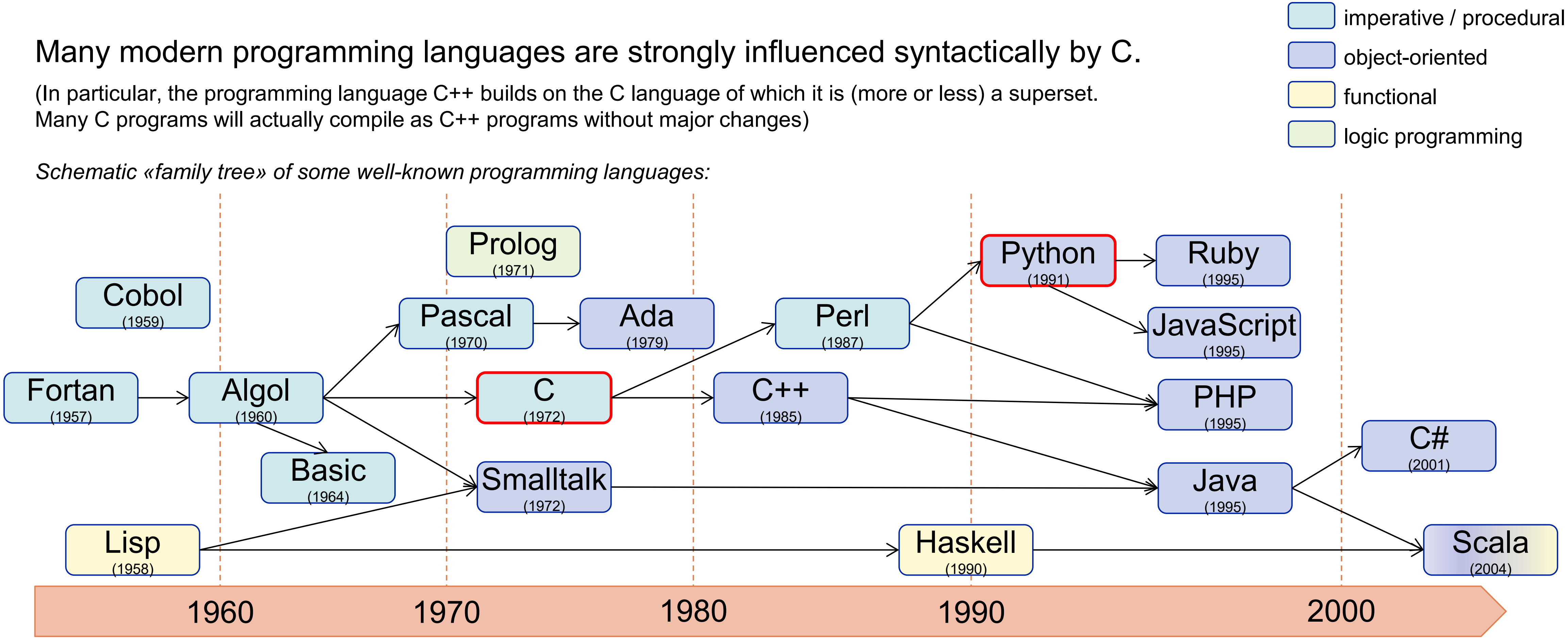
(domain specific languages)

C as a Programming Language

Many modern programming languages are strongly influenced syntactically by C.

(In particular, the programming language C++ builds on the C language of which it is (more or less) a superset. Many C programs will actually compile as C++ programs without major changes)

Schematic «family tree» of some well-known programming languages:



Applications Using C

C/C++ is a very wide-spread, cross-platform (portable) and multi-purpose language.

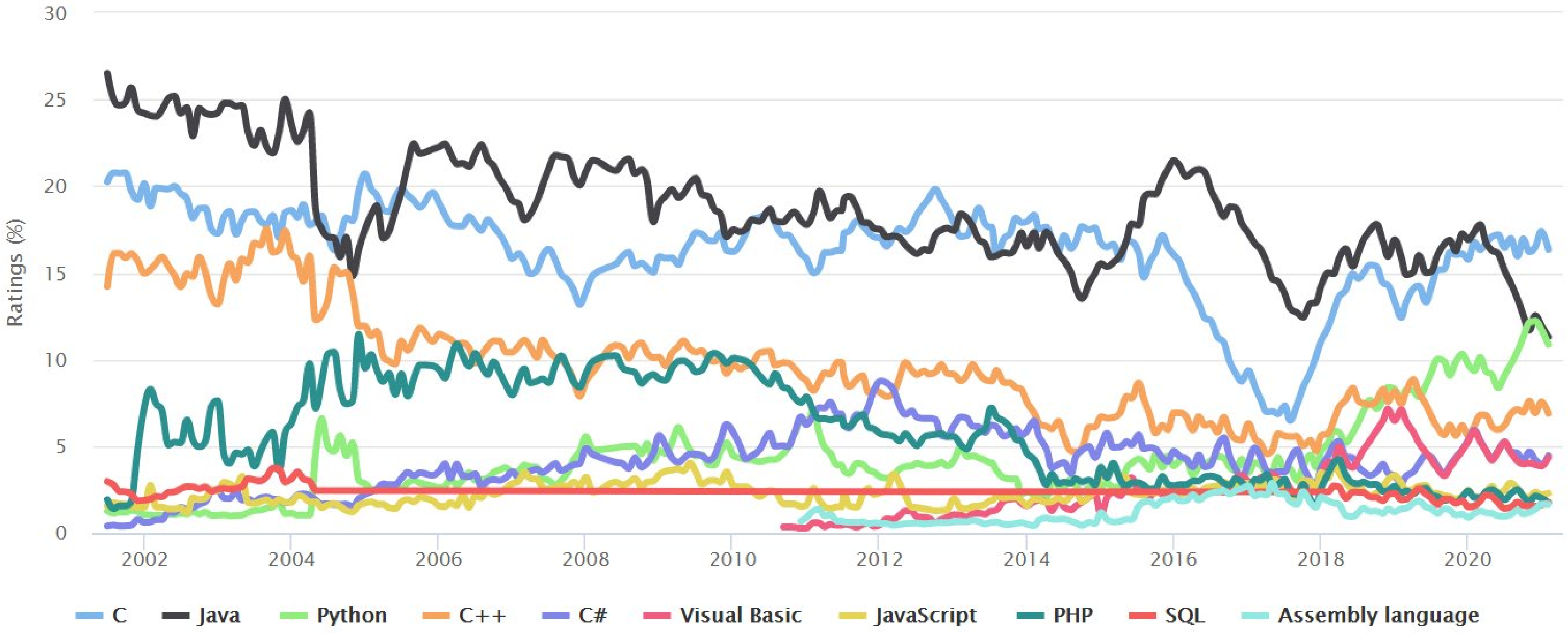
Examples of usage (arbitrary selection):

- Operating systems (kernels), e.g. macOS, Windows, Unix, Linux,
- Java Virtual Machine, Python Internals (CPython),
- Database Management Systems, e.g. PostgreSQL, MySQL,
- Embedded Systems (washing machines, Mars rovers, …),
- High Performance Computing,
- …

PostgreSQL

MySQL™

Usage and Spread of C: TIOBE Index



Python vs. C: General Overview

- C is distinctively more low-level than Python.

- C uses static type checking, meaning that you as programmers have to specify *before runtime* what data type a variable will have and this data type will then stick unchangeably to that variable. Python in contrast uses dynamic type checking.

- C is *not* an object-oriented programming language:
  - C has no classes.
  - C programs are sets of functions. Natively, there are only limited ways to structure large amounts of code.

Python vs. C: General Overview

- C code is (usually) compiled while Python is (usually) interpreted.

- Python is the more modern language: C was released in 1972, Python in 1991.

- There are various syntactical differences (some of which might take some time to get used to).

A Sample Program in C

```
/* my first program in C */
#include <stdio.h>

int main() {
   int number;

   printf("Enter an integer:  >> ");
   scanf("%d", &number);

   printf("Your input was %d\n", number);

   if (number > 0 && number % 2 == 0) {
      printf("Your input was valid.\n");
   } else {
      printf("Your input was not valid.\n");
   }

   return 0;
}
```

Compile: `gcc myFirstProgram.c -o myFirstProgram`

Execute: `./myFirstProgram`

Python vs. C: Syntax

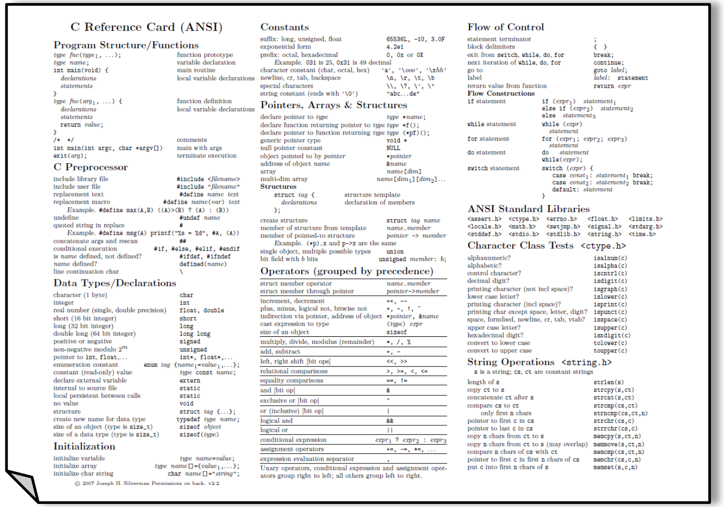
- Every statement in C has to end with a semicolon (`;`);
- Operators &&, `||` and `!` (instead of `and`, `or` and `not` as in Python)
- Blocks are indicated with curly braces (`{`, `}`); additional whitespaces are ignored (but should be properly set either way).
- There is no colon (`:`) after function signatures and `if`, `else`, `while`, `for` statements.
- Comments are indicated with `/*`, `*/` (and `//`) for comments (instead of # as in Python).
- Libraries are made available using `#include` (instead of `import` as in Python).
- Functions have return types (e.g. `int`) but there is no `def` keyword.
- The function `printf` is used for output to the console (and not `print` as in Python).
- There is a function `scanf` instead of `input` / `raw_input` as in Python.
- …

Resources on C Programming

- C Reference Card by Joseph H. Silverman

- There are many, many books on C. The most famous one is probably the work by the creators of C (the «Bible of C»):
  - «The C Programming Language» by Brian W. Kernighan and Dennis M. Ritchie

- There are also many, many online resources with tutorials, interactive exercises etc.



C Reference Card (ANSI)

Program Structure/Functions

```
type fnc(type1, ...);               function prototype
type name;                          variable declaration
int main(void) {                    main routine
  declarations                      local variable declarations
  statements
}
type fnc(arg1, ...) {               function definition
  declarations                      local variable declarations
  statements
  return value;
}
/*  */                              comments
int main(int argc, char *argv[])    main with args
exit(arg);                          terminate execution
```

C Preprocessor

include library file	`#include <filename>`
include user file	`#include "filename"`
replacement text	`#define name text`
replacement macro	`#define name(var) text`
Example. `#define max(A,B) ((A)>(B) ? (A) : (B))`	
undefine	`#undef name`
quoted string in replace	`#`
Example. `#define msg(A) printf("%s = %d", #A, (A))`	
concatenate args and rescan	`##`
conditional execution	`#if, #else, #elif, #endif`
is *name* defined, not defined?	`#ifdef, #ifndef`
name defined?	`defined(name)`
line continuation char	`\`

Data Types/Declarations

character (1 byte)	`char`
integer	`int`
real number (single, double precision)	`float, double`
short (16 bit integer)	`short`
long (32 bit integer)	`long`
double long (64 bit integer)	`long long`
positive or negative	`signed`
non-negative modulo 2^m	`unsigned`
pointer to `int`, `float`,...	`int*, float*,...`
enumeration constant	`enum tag {name1=value1,...};`
constant (read-only) value	`type const name;`
declare external variable	`extern`
internal to source file	`static`
local persistent between calls	`static`
no value	`void`
structure	`struct tag {...};`
create new name for data type	`typedef type name;`
size of an object (type is `size_t`)	`sizeof object`
size of a data type (type is `size_t`)	`sizeof(type)`

Initialization

initialize variable	`type name=value;`
initialize array	`type name[]={value1,...};`
initialize char string	`char name[]="string";`

© 2007 Joseph H. Silverman Permissions on back. v2.2

Constants

suffix: long, unsigned, float	`65536L, -1U, 3.0F`
exponential form	`4.2e1`
prefix: octal, hexadecimal	`0, 0x or 0X`
Example. `031` is 25, `0x31` is 49 decimal	
character constant (char, octal, hex)	`'a', '\ooo', '\xhh'`
newline, cr, tab, backspace	`\n, \r, \t, \b`
special characters	`\\, \?, \', \"`
string constant (ends with `'\0'`)	`"abc...de"`

Pointers, Arrays & Structures

declare pointer to *type*	`type *name;`
declare function returning pointer to *type*	`type *f();`
declare pointer to function returning *type*	`type (*pf)();`
generic pointer type	`void *`
null pointer constant	`NULL`
object pointed to by *pointer*	`*pointer`
address of object *name*	`&name`
array	`name[dim]`
multi-dim array	`name[dim1][dim2]...`

Structures

```
  struct tag {          structure template
    declarations        declaration of members
  };
```

create structure	`struct tag name`
member of structure from template	`name.member`
member of pointed-to structure	`pointer -> member`
Example. `(*p).x` and `p->x` are the same	
single object, multiple possible types	`union`
bit field with *b* bits	`unsigned member: b;`

Operators (grouped by precedence)

struct member operator	`name.member`
struct member through pointer	`pointer->member`
increment, decrement	`++, --`
plus, minus, logical not, bitwise not	`+, -, !, ~`
indirection via pointer, address of object	`*pointer, &name`
cast expression to type	`(type) expr`
size of an object	`sizeof`
multiply, divide, modulus (remainder)	`*, /, %`
add, subtract	`+, -`
left, right shift [bit ops]	`<<, >>`
relational comparisons	`>, >=, <, <=`
equality comparisons	`==, !=`
and [bit op]	`&`
exclusive or [bit op]	`^`
or (inclusive) [bit op]	`|`
logical and	`&&`
logical or	`||`
conditional expression	`expr1 ? expr2 : expr3`
assignment operators	`+=, -=, *=, ...`
expression evaluation separator	`,`

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	`;`
block delimiters	`{ }`
exit from `switch`, `while`, `do`, `for`	`break;`
next iteration of `while`, `do`, `for`	`continue;`
go to	`goto label;`
label	`label: statement`
return value from function	`return expr`

Flow Constructions

```
if statement        if (expr1)  statement1
                    else if (expr2)  statement2
                    else  statement3
while statement     while (expr)
                      statement
for statement       for (expr1; expr2; expr3)
                      statement
do statement        do  statement
                    while(expr);
switch statement    switch (expr) {
                       case const1: statement1 break;
                       case const2: statement2 break;
                       default: statement
                    }
```

ANSI Standard Libraries

```
<assert.h>  <ctype.h>  <errno.h>   <float.h>   <limits.h>
<locale.h>  <math.h>   <setjmp.h>  <signal.h>  <stdarg.h>
<stddef.h>  <stdio.h>  <stdlib.h>  <string.h>  <time.h>
```

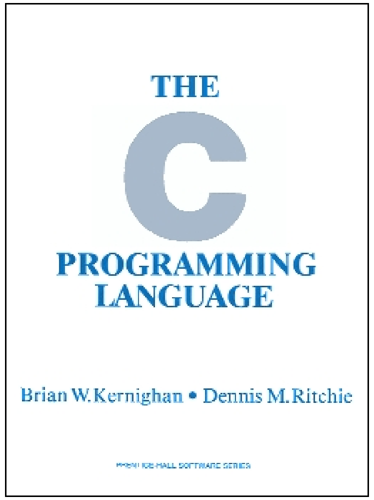
Character Class Tests <ctype.h>

alphanumeric?	`isalnum(c)`
alphabetic?	`isalpha(c)`
control character?	`iscntrl(c)`
decimal digit?	`isdigit(c)`
printing character (not incl space)?	`isgraph(c)`
lower case letter?	`islower(c)`
printing character (incl space)?	`isprint(c)`
printing char except space, letter, digit?	`ispunct(c)`
space, formfeed, newline, cr, tab, vtab?	`isspace(c)`
upper case letter?	`isupper(c)`
hexadecimal digit?	`isxdigit(c)`
convert to lower case	`tolower(c)`
convert to upper case	`toupper(c)`

String Operations <string.h>

`s` is a string; `cs`, `ct` are constant strings

length of `s`	`strlen(s)`
copy `ct` to `s`	`strcpy(s,ct)`
concatenate `ct` after `s`	`strcat(s,ct)`
compare `cs` to `ct`	`strcmp(cs,ct)`
only first `n` chars	`strncmp(cs,ct,n)`
pointer to first `c` in `cs`	`strchr(cs,c)`
pointer to last `c` in `cs`	`strrchr(cs,c)`
copy `n` chars from `ct` to `s`	`memcpy(s,ct,n)`
copy `n` chars from `ct` to `s` (may overlap)	`memmove(s,ct,n)`
compare `n` chars of `cs` with `ct`	`memcmp(cs,ct,n)`
pointer to first `c` in first `n` chars of `cs`	`memchr(cs,c,n)`
put `c` into first `n` chars of `s`	`memset(s,c,n)`


THE
C
PROGRAMMING
LANGUAGE
Brian W.Kernighan • Dennis M.Ritchie
PRENTICE HALL SOFTWARE SERIES

How to work with C?

There are two main ways:

- Text editor and command line interface (CLI)
- Integrated Development Environment (IDE)
- (Virtual machine, Docker container)
- (Online IDE)

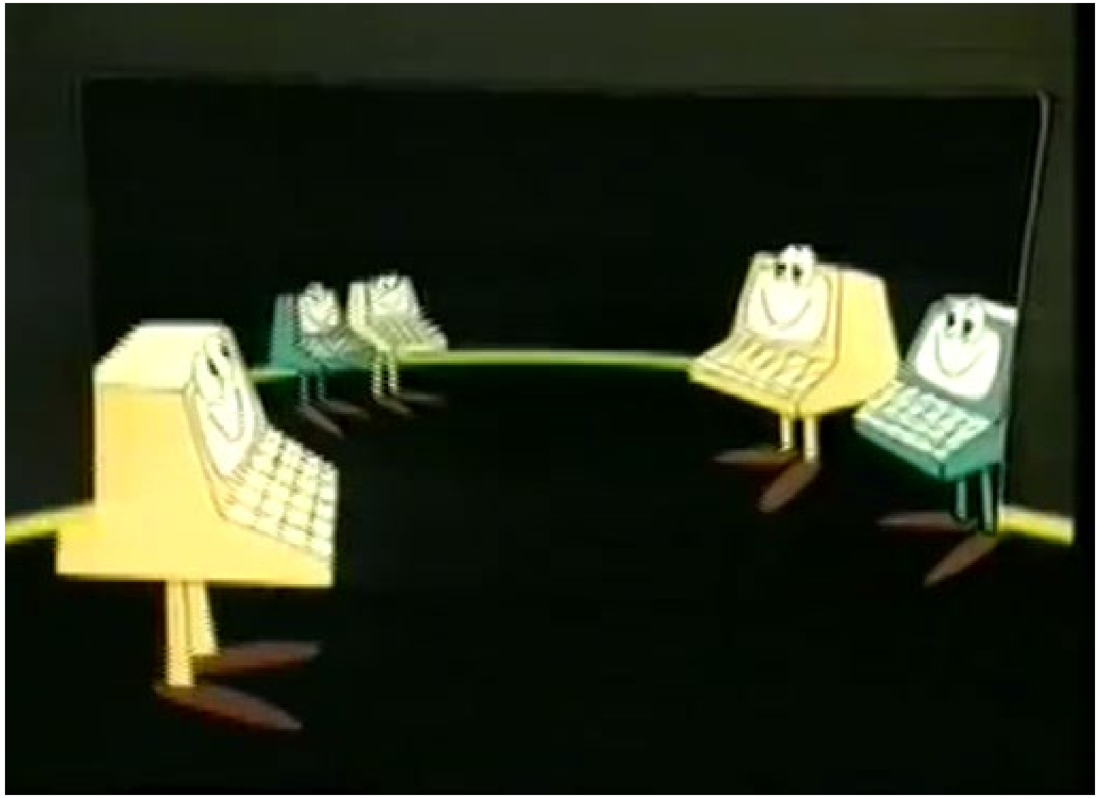
Hints and suggestions can be found on OLAT (folder «Useful Docs» → «C Programming») and on the web page for the tutorials.

Compiler and Interpreter

- C is usually compiled.
- Python is usually interpreted.

Note that a C program can also be run on an interpreter and a Python program can be compiled for execution (both being quite unusual, though).

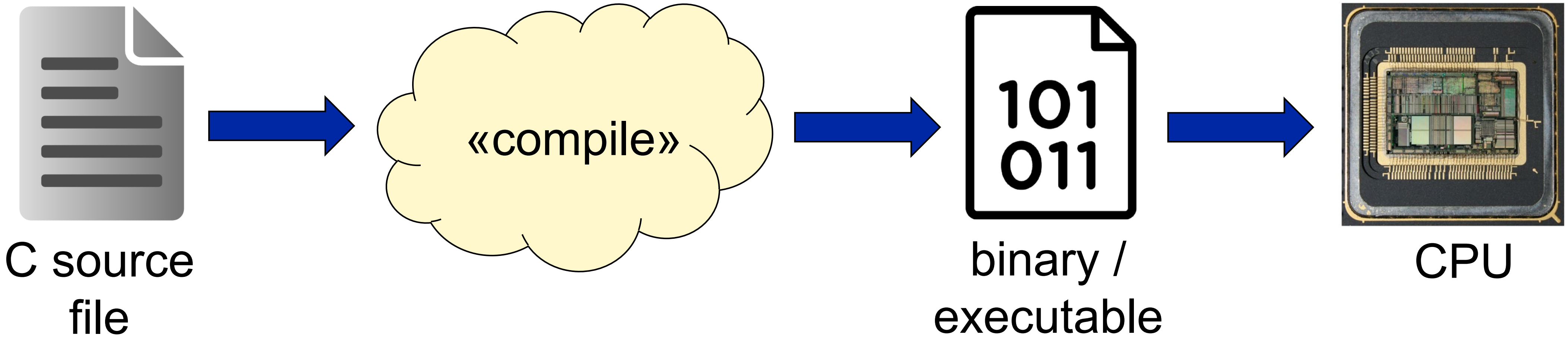
Compiler and Interpreter



from the Canadian television series «Bits and Bytes», episode 6, from the year 1983

https://www.youtube.com/watch?v=_C5AHaS1mOA

A Closer Look at the Compiling Process



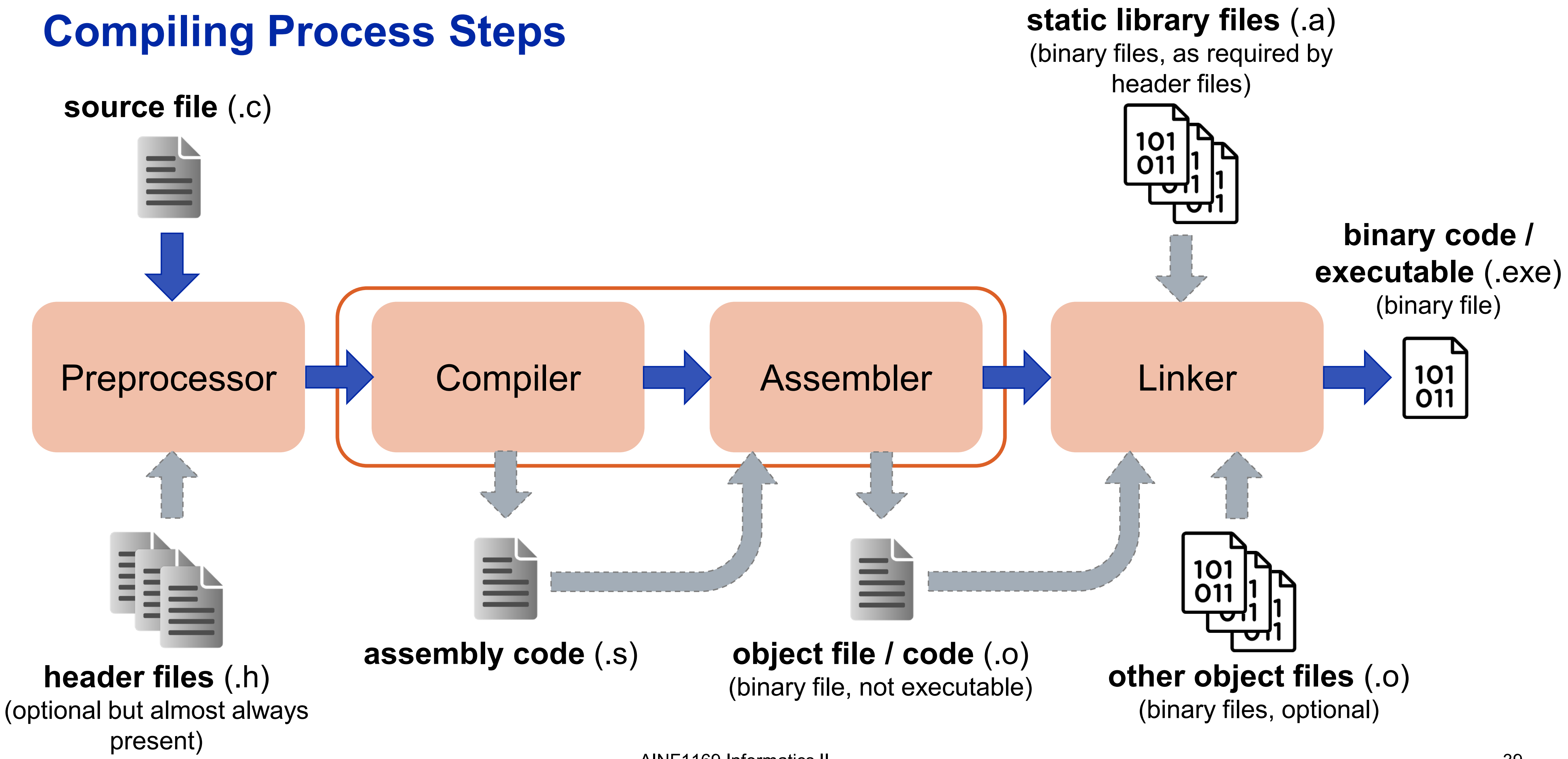
C source file

«compile»

binary / executable

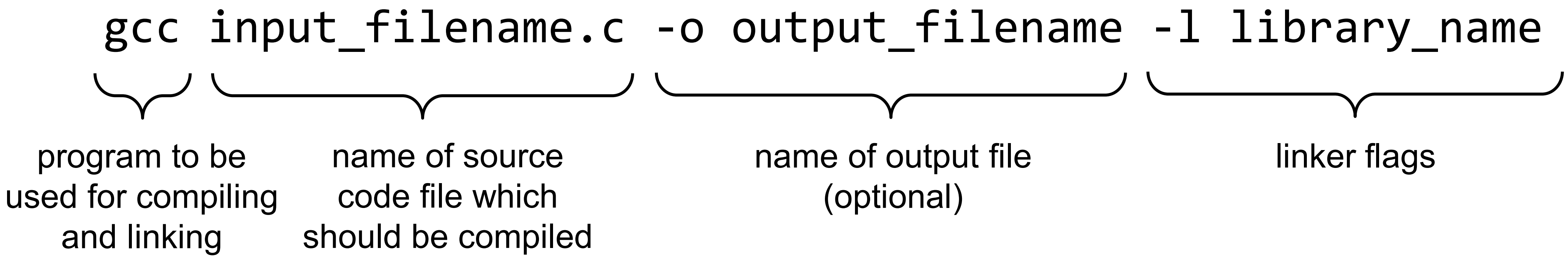
CPU

Compiling Process Steps


static library files (.a)
(binary files, as required by header files)
source file (.c)
binary code / executable (.exe)
(binary file)
Preprocessor
Compiler
Assembler
Linker
101
011
header files (.h)
(optional but almost always present)
assembly code (.s)
object file / code (.o)
(binary file, not executable)
other object files (.o)
(binary files, optional)

Compiling on the Console

On the console, C code can be translated into an executable binary using statements similar to the one below:



```
gcc input_filename.c -o output_filename -l library_name
```

program to be used for compiling and linking

name of source code file which should be compiled

name of output file (optional)

linker flags

There are more compiler flags available which might be helpful. In particular, the `-Wall` flag can be applied to enforce that all warning messages are printed.

Memory in C

The single most important difference between Python and C may be the treatment of memory.

In C you get more control over and responsibility for memory, whereas in Python this is hidden from you beneath a layer of abstraction (you’re programming «closer to the bare metal»).

To understand C and write C code, it is therefore necessary to understand memory.

Memory in C

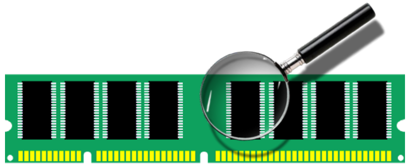
In a strongly simplified model we can consider the memory in a computer to be a sequence of boxes, each having a distinct address and each able to contain certain «things» (represented in the form of the symbols 0 and 1).

Today’s computer hardware will almost always have a byte addressable memory architecture. This means that there is a separate address for every byte (8 bit) of memory. (For performance reasons, the hardware will not actually read and write single bytes, though.)

The memory address in a contemporary computer will typically be a 32 bit number or a 64 bit number. They are usually written as hexadecimal numbers (starting with `0x`).

In graphical representations like in the one on the right, memory addresses are mostly increasing from bottom to top.

address	*Content’s binary representation*
	…
0x00010004	**01101001**
0x00010003	**01001111**
0x00010002	**00101001**
0x00010001	**10100101**
0x00010000	**10010100**
	…



Outlook: Advanced Concepts when Working with C

Since this is only a very short initial introduction, there were many things omitted which are not needed at the moment for the exercises but would be important in a full introduction and necessary when working with C in a productive manner. The list below contains some of these topics as a reference, so you might look it up yourself if you're interested:

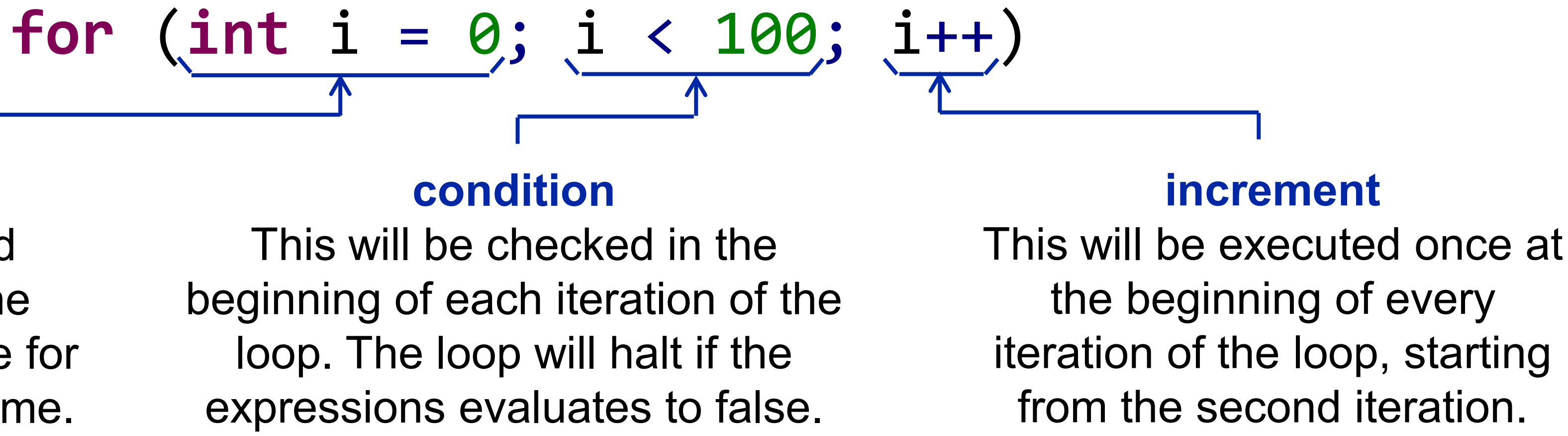
- Debugging, debuggers (e.g. gdb)
- Environment variables
- Preprocessor directives and macros
- Make files
- Memory segments
- Creating your own static libraries
- C variants and standards
- …

Introduction to C – Data Types, Variables, Conversions, Operators

- Basic Control Structures: Loops, Increment / Decrement Operator
- Variable Declaration and Initialization, Memory Allocation
- Data Types an Their Sizes
- ASCII Table
- Type Conversions, Casts
- Operators and Operator Precedence
- Qualifiers, Constants

Remarks on Loops: for Loops in C

A `for` loop consists of an header and a body. The header has three parts separated by semicolons: initialization, condition and increment.



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

initialization
This will be executed exactly once when the control flow reaches the for statement for the first time.

condition
This will be checked in the beginning of each iteration of the loop. The loop will halt if the expressions evaluates to false.

increment
This will be executed once at the beginning of every iteration of the loop, starting from the second iteration.

- All three parts of the header can be arbitrary expressions in principle. It is possible to have complex statements within the head of the for loop (use it hesitantly because it can get difficult to understand).
- Any of the three parts of the header can be omitted (the semicolons need to stay).

C vs Python: Loops

	Python	C

Count up from 0 to 99 one by one:
0, 1, 2, …, 99
	`for i in range(0, 100):`
	`    #...`
		`for (int i = 0; i < 100; i++) {`
		`        /* ... */`
		`}`

Count down from 99 to 0 one by one:
99, 98, 97, …, 0
	`for i in range(99, -1 , -1):`
	`    #...`
		`for (int i = 99; i > -1; i--) {`
		`        /* ... */`
		`}`

Count up from 0 to 99 with a step width of 3:
0, 3, 6, …, 99
	`for i in range(0, 100, 3):`
	`    #...`
		`for (int i = 0; i < 100; i += 3) {`
		`        /* ... */`
		`}`

Iterate all elements of a data structure:	`list_of_words = ["hi", "my", "friend"]`
	`for word in list_of_words:`
	`    #...`
		(a bit more complicated; we'll see later how this is done in C; there is no «foreach» like expression)

Loops: `while`

The header of the `while` loop is executed…

– at least once,

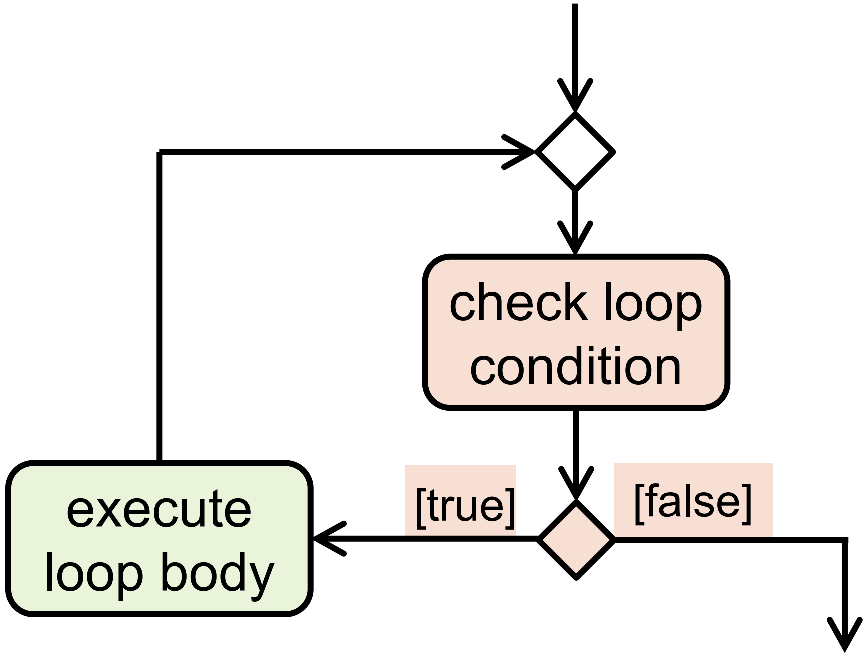
– one more time than the body of the loop.

C code:

```
while (/* loop condition */) {
    /* loop body */
}
```

Pseudocode:

```
while <condition> do
    <loop body>
```



Loops: `for`

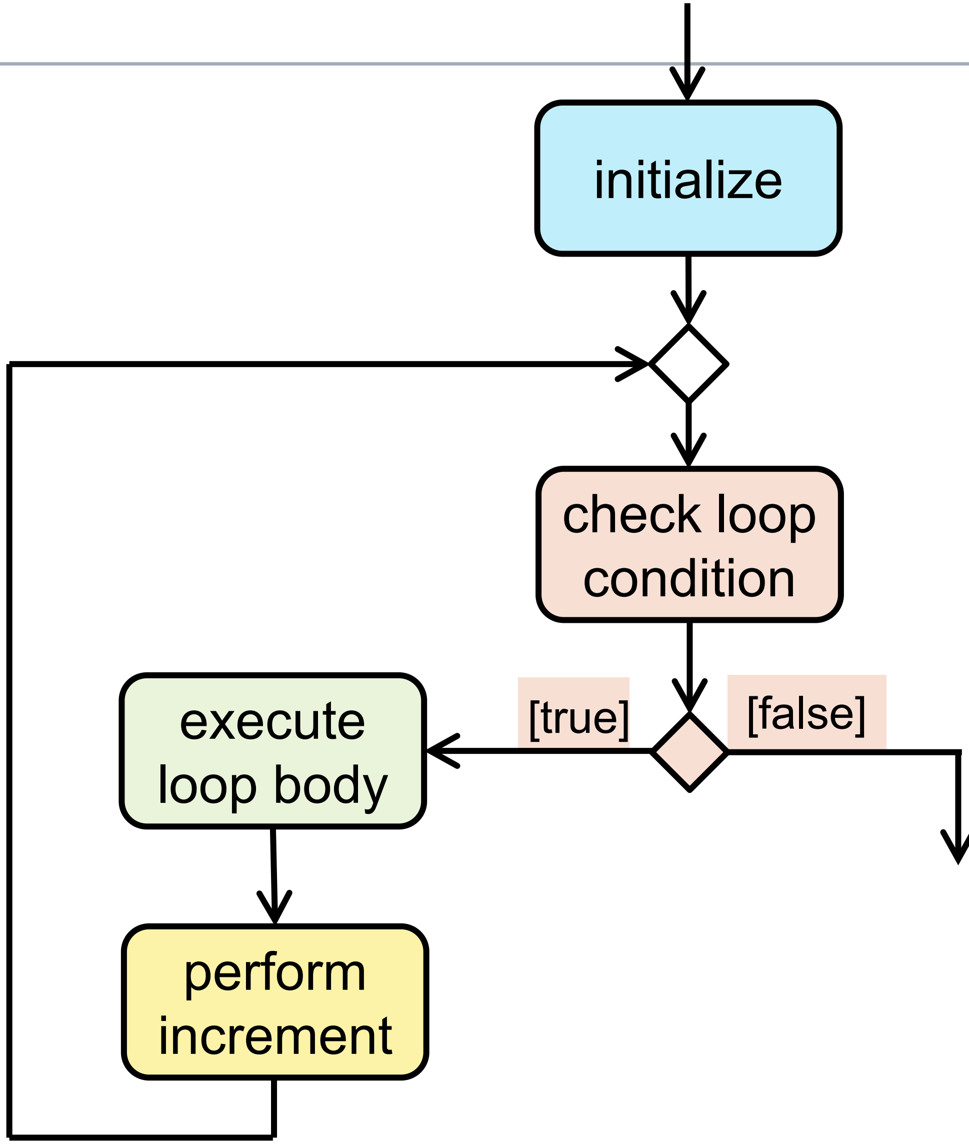
A `for` loop consists of an header and a body. The header has three parts (separated by semicolons in C code): initialization, condition and increment.

The initialization is executed exactly once.

The loop exit condition check is executed…

- at least once,
- one more time than the body of the loop and the increment.

```
for (/*ini*/; /*con*/; /*inc*/) {
    /* loop body */
}
```



Increment and Decrement (Pre / Post)

If the value of an integer variable `x` needs to be incremented in C, this can be done by just reassigning using `x = x + 1` or `x += 1`. There are also two convenient operators for this purpose:

- **pre-increment operator**: **`x++`**
- **post-increment operator**: **`++x`**

These operators have (apart from being shorter) the advantage that they can be applied in places where an assignment like `x = x + 1` could not be used, for example when printing a value: `printf("%d", x++)`. In this case, it would be necessary to perform the increment on a separate line.

These two operators are not working in exactly the same manner. As their names indicate, the

- pre-increment operator will increment the value of the variable first, before it is used within the larger code context, and the
- post-increment operator will use the current, unchanged value of the variable first within the code and only then increment it.

Thus, you can think of pre-incrementation and post-incrementation as two-step processes.

Preliminaries: Pre-Increment and Post-Increment (and Decrement)

pre-increment

++x

increment – use

```
x = ++i;    ≙    i = i + 1;
                 x = i;
```

```
int x = 42;
printf("%d, ", ++x);
printf("%d", x);
```

will print 43, 43

post-increment

x++

use – increment

```
x = i++;    ≙    x = i;
                 i = i + 1;
```

```
int x = 42;
printf("%d, ", x++);
printf("%d", x);
```

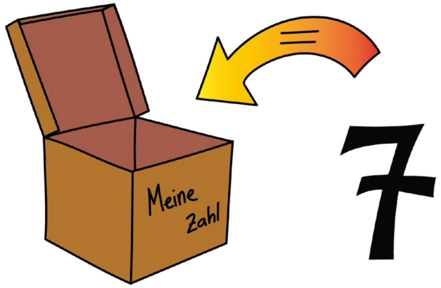
will print 42, 43

Variables, Allocation of Memory

When you declare a variable, the compiler will translate this into machine language which will reserve at a certain position in memory a certain amount of memory needed to store the variable.

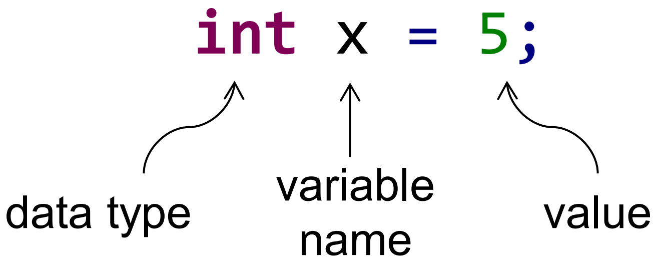
This process is called allocating memory.

Since we (usually) do not want to work with the actual memory address, variables in C code will instead get a (unique) name which makes it easier to refer to the reserved memory. (At run time, these names will have been completely replaced by memory addresses, though.)



Data Types, Variable Declaration and Initialization

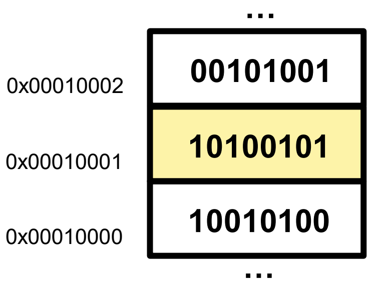
Since C applies static typing, you have to state what data type a variable should have when you declare the variable (in contrast to Python), for example:



```
int x = 5;
```

Depending on the data type, a respective, *fixed* amount of memory will be reserved when the program runs (e.g. 1 byte for a `char`) and the address where it starts will be remembered. The data type of a variable cannot be changed, so neither can the amount of memory.

This comes with several consequences, e.g. when performing operations with variables of different data types (see later).



	...
0x00010002	00101001
0x00010001	10100101
0x00010000	10010100
	...

Variable Declaration and Initialization

- Declaration: specify name and reserve required memory according to type, e.g. `int myVariable;`
- Initialization: set value, e.g. declaration and initialization at once: `int myVariable = 42;`

- In C, a variable can only be used after it has been declared (the same applies for functions).
- Beware: The value of an uninitialized variable in a function will not (necessarily) be set to a default value (meaning: is not guaranteed to be zero). The value currently present at the respective memory position could just be used as the initial value of the variable if not specifically set otherwise by the programmer. Therefore always explicitly initialize variables used in functions if this is relevant to the control flow.

Data Types in C

- C applies static type checking. This means that you as a programmer have to define what data type a variable has and this is fixed at compile time and cannot change at runtime. C also applies strong typing which means that it restrictive about what is allowed when adding, multiplying or assigning variables with different data types.

- There is a relatively constricted list of native data types in C. In fact, there are only four of them:

  `char`, `int`, `float`, `double`

- Note that (originally) there is (was) no dedicated boolean type in C. Instead, zero values (`0`) are treated as false and everything else is regarded as true. The `stdbool.h` header file might be used to help out, though (see later).

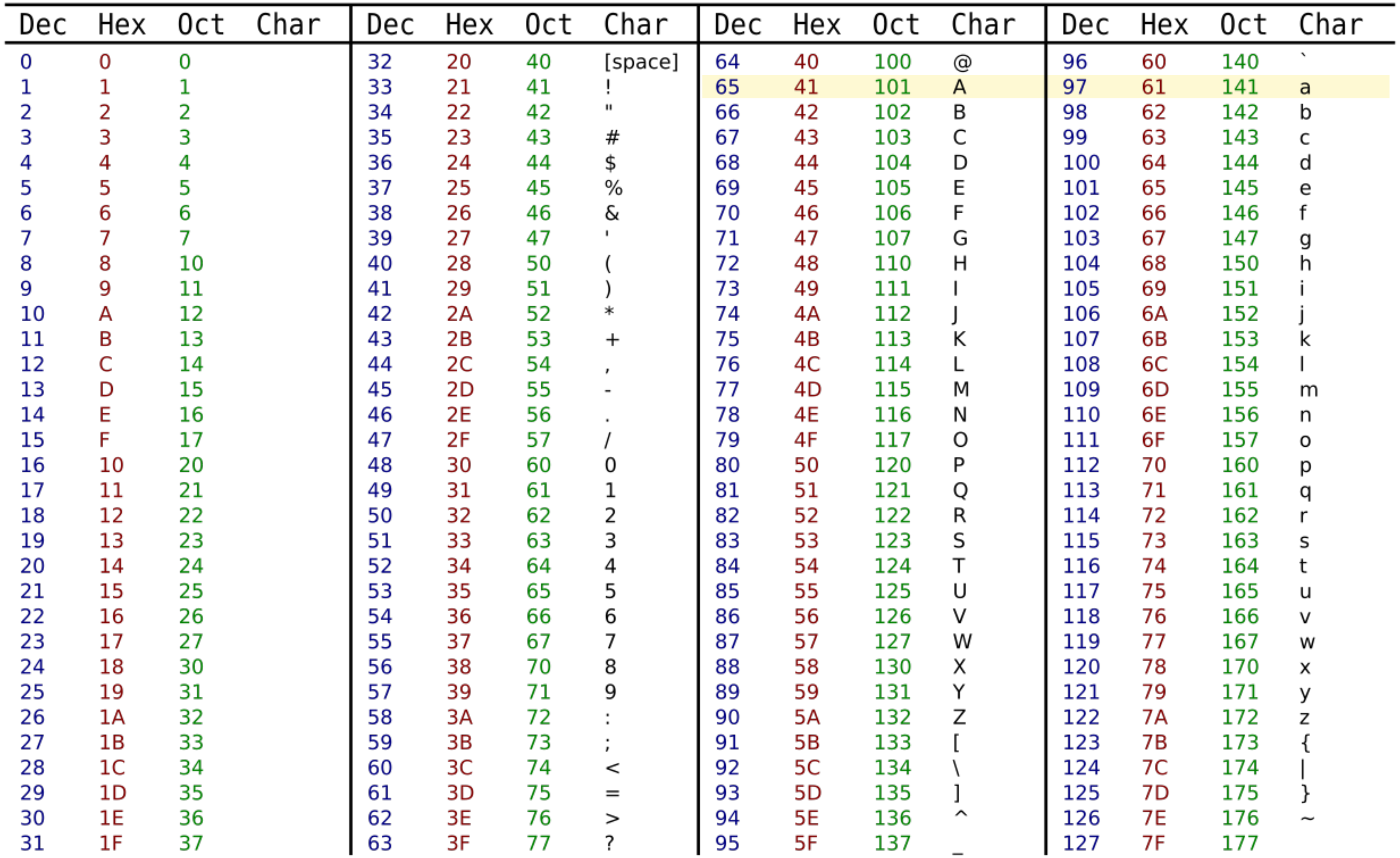
- Natively, there are no lists, tuples or dictionaries in C. Strictly speaking, there are also no strings as such… These structures can be built in C «by hand», of course. How this can be done is a topic of this course amongst others.

Data Types in C

Keyword	Typical size*	Value range*	Meaning
`char`	1 byte	256 different values / characters	Single character; encoded according to ASCII table
`int`	4 bytes	−2,147,483,648 (-2^{31}) to +2,147,483,647 ($+2^{31}-1$)	Integer
`float`	4 bytes	ca. $\pm 1.2 \cdot 10^{-38}$ to $\pm 3.4 \cdot 10^{38}$ (about 6 decimal places of precision)	Decimal number
`double`	8 bytes	ca. $\pm 2.2 \cdot 10^{-308}$ to $\pm 1.8 \cdot 10^{308}$ (about 15 decimal places of precision)	Decimal number

* Actual sizes and respective value ranges depend on the platform / compiler. In order to deal with this, there also are support types with some guarantees on this (e.g. fixed-width integer types like `int8_t`, `int32_t` defined in `stdint.h`).

Characters, ASCII Table, Uppercase/Lowercase



Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

← difference between uppercase and lowercase characters 'A' and 'a' is 32 in decimal system. This applies to all letters up until 'z' / 'Z'.

Note that character literals in C can only be defined using single quotes, e.g.

`char myChar = 'X';` ✓ valid

`char myChar = "X";` ✗ wrong

Data Types in C: Example

What will most probably be the output of the following C program?

```c
#include <stdio.h>

int main() {
	int a = 100000;
	int b = 200000;
	int c = a * b;

	printf("a * b = %d", c);
	return 0;
}
```

A: Does not compile

B: Run time error

C: a * b = 20000000000

D: a * b = -1474836480

Answer D is correct: There will be an overflow of the integer variable.

Data Types in C: Danger Zone

Be careful when using floating point types.

– *Never* make comparisons using floating point types (at least without controlling for roundoff errors).

– Do not use floating point types to represent currencies.

Example: What will be the output of this code snippet?

```c
#include <stdio.h>
int main() {
     float x = 0.0;
     for (int i = 0; i < 10000; i++) {
         x = x + 0.1;
     }
     printf("Result: %f", x);
     return 0;
}
```

The adjacent code snippet will produce the following output:

```
Result: 999.902893
```

This differs from the output that one probably might have expected (1000) because the value 0.1 cannot be represented exactly in binary and thus there will be a small difference in every iteration which will successively pile up.

Data Types in C: More Pitfalls

- Mathematical laws do not necessarily apply, e.g. the associative property will not necessarily always hold in C programming:

  `(a + b) + c == a + (b + c)` *can* be false (e.g. for a = 0.006f, b = 0.0006f and c = 0.0007f on most systems)

  `(a * b) * c == a * (b * c)` *can* be false.

- The order in which calculations are done may be important.

- Just applying formulas you know from Mathematics as they are, may be a bad idea. Example: using the usual form of the quadratic formula («Mitternachtsformel») to calculate the roots of a quadratic equation is a bad idea in general since it could fail in edge cases.

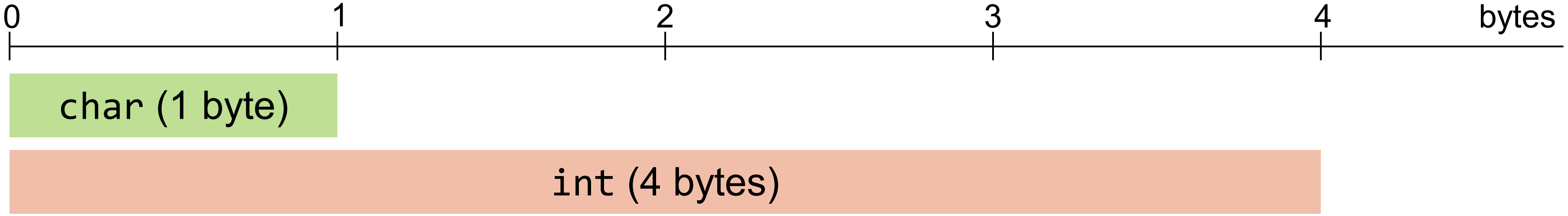
$$\xcancel{x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}}$$

Data Types: Bad Examples

- Failure of MM-104 Patriot missile in 1991 (Gulf War): 28 deaths due to a floating point rounding error
- Maiden flight of Ariane 5 in 1996: financial loss of about 290 million Euros due to erroneous type conversion / overflow

Type Conversion

When a variable is declared, an amount of memory will be allocated (reserved) according to the data type. Since some data types have more memory assigned than others, there potentially could be a problem when one tries to assign two variables of different data types or perform an operation in which variables of different data types are involved.

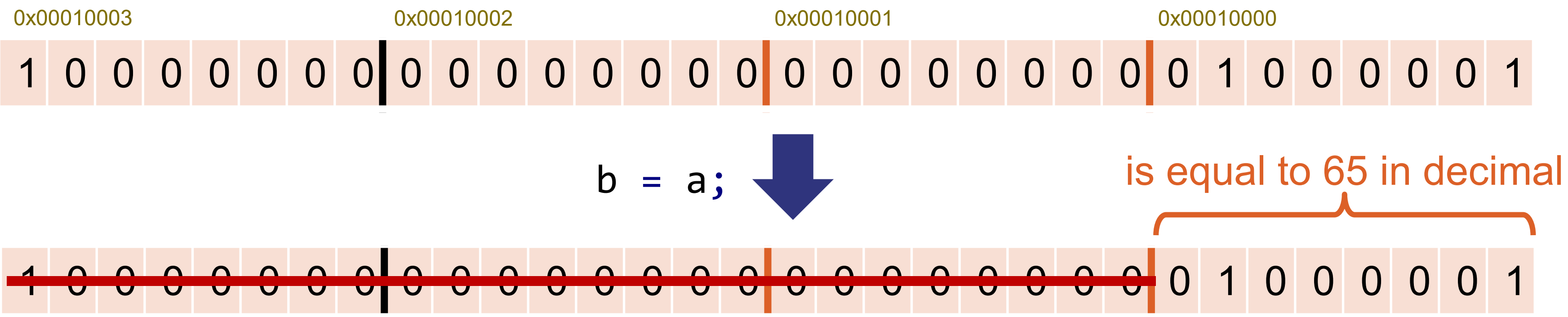


For example, if you assign a variable of type `int` (typically 4 bytes) to a variable of type `char` (1 byte), all bits which do not fit into the target variable will just be discarded (cut off) without warning.

Type Conversion: Example

Consider the following code fragment. What will be printed to the console?

```
int a = 0;
char b = 'A';
a = b;                 /* no compiler error */
printf("%d", a);       /* prints 65 */
a = -2147424447;       /* in binary: 10000000 00000000 11100111 01000001 */
b = a;                 /* no compiler error */
printf("%c", b);       /* prints A */
```



1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 1 0 0 0 0 0 1

b = a;

is equal to 65 in decimal

~~1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0~~ | 0 1 0 0 0 0 0 1

Type Conversion, Cast

- Conversion can occur / be done implicitly (automatically by the compiler) or explicitly (forced by the programmer).
- An implicit conversion occurs when an operation is done with different data types (e.g. `3.14 * 5`).
- Implicit conversions will always be done «upwards», meaning towards the data type which occupies more memory space (i.e. `char` → `int` → `float` → `double`).
- An explicit / forced conversion is called a **cast**.

  Example of a cast: `int myVariable = (int)(10.0 / 3.0);`
- A cast towards a data type with less memory is called a down cast. This kind of cast will (usually / potentially) result in loosing information.

  Example of an explicite down cast:

```
double x = 1.2;
int sum = (int)x + 1;
printf("sum = %d", sum); // prints 2
```

Operators in C: Integer Division and Exponentiation

There are some important differences between operator behaviour between Python and C:

- Integer division won’t be implicitly converted to float.

  Example: What will be the output of the following code snippet?

```c
int a = 4;
int b = 3;
double x = a / b;
printf("output: %f\n", x);
```

The output will be: `output: 1.000000`

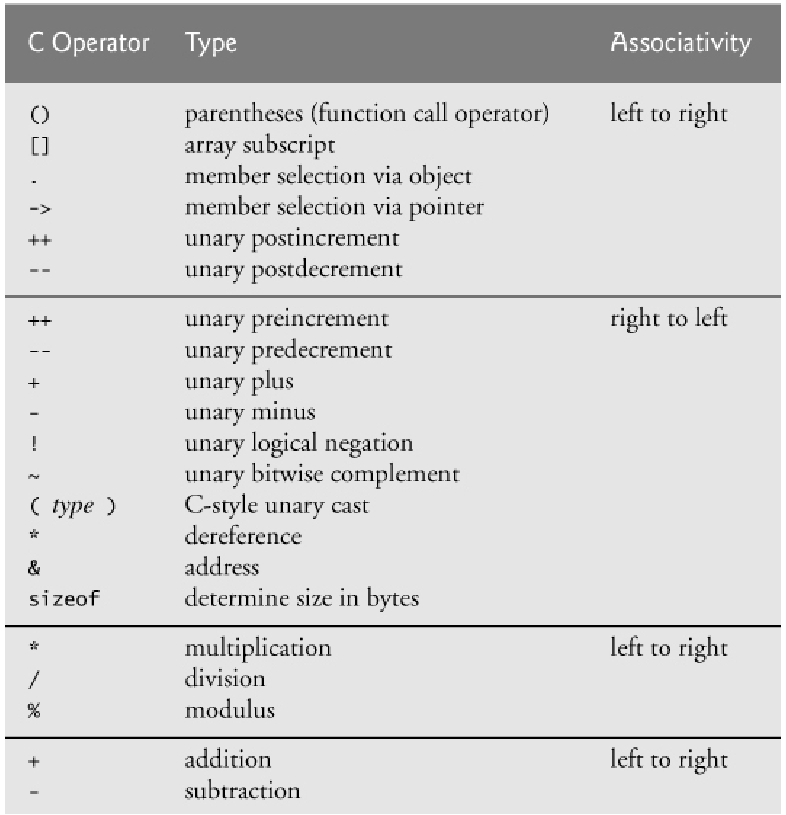
This is because in line 4 there is an integer division and the value of x will be 1.

- i.e. for integers, the division (`/`) operator in C is semantically equivalent to the // operator in Python.

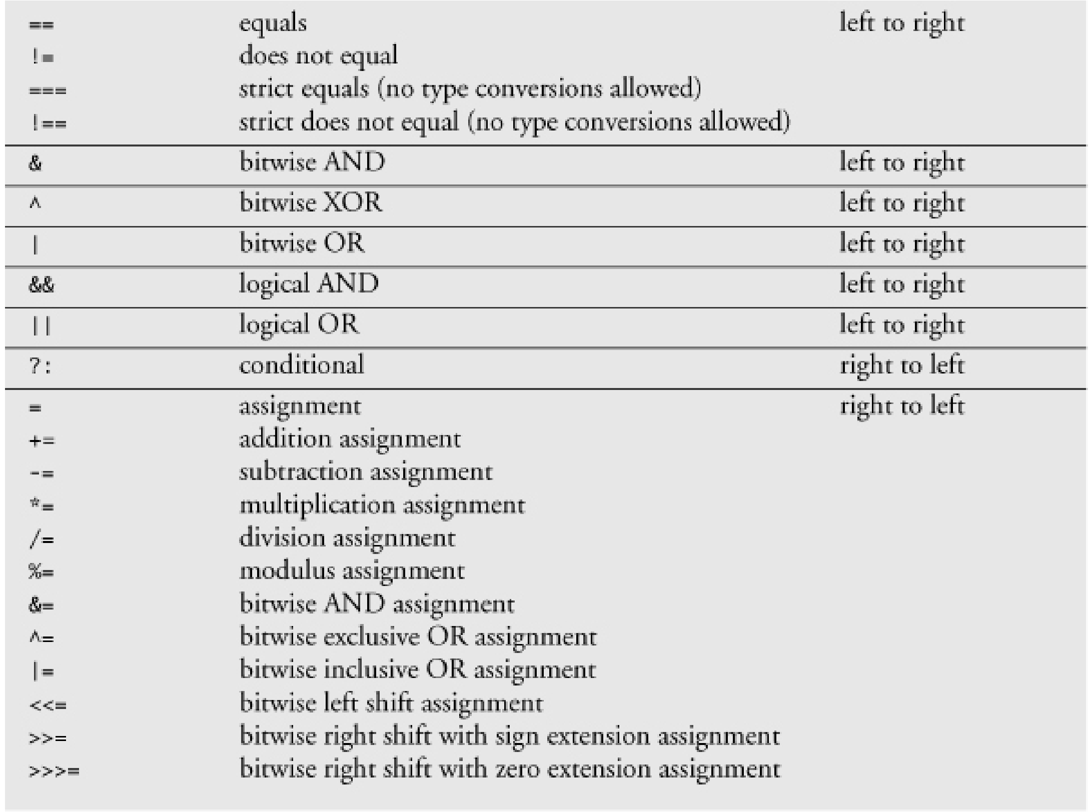
- In C, there is no `**` operator for exponentiation (you may use the function `pow` from the `math` library).

Operators in C: Precedence

The operator precedence in C is more or less identical to operator precedence in Python.



C Operator	Type	Associativity
`()`	parentheses (function call operator)	left to right
`[]`	array subscript	
`.`	member selection via object	
`->`	member selection via pointer	
`++`	unary postincrement	
`--`	unary postdecrement	
`++`	unary preincrement	right to left
`--`	unary predecrement	
`+`	unary plus	
`-`	unary minus	
`!`	unary logical negation	
`~`	unary bitwise complement	
`(` *type* `)`	C-style unary cast	
`*`	dereference	
`&`	address	
`sizeof`	determine size in bytes	
`*`	multiplication	left to right
`/`	division	
`%`	modulus	
`+`	addition	left to right
`-`	subtraction	



`==`	equals	left to right
`!=`	does not equal	
`===`	strict equals (no type conversions allowed)	
`!==`	strict does not equal (no type conversions allowed)	
`&`	bitwise AND	left to right
`^`	bitwise XOR	left to right
`|`	bitwise OR	left to right
`&&`	logical AND	left to right
`||`	logical OR	left to right
`?:`	conditional	right to left
`=`	assignment	right to left
`+=`	addition assignment	
`-=`	subtraction assignment	
`*=`	multiplication assignment	
`/=`	division assignment	
`%=`	modulus assignment	
`&=`	bitwise AND assignment	
`^=`	bitwise exclusive OR assignment	
`|=`	bitwise inclusive OR assignment	
`<<=`	bitwise left shift assignment	
`>>=`	bitwise right shift with sign extension assignment	
`>>>=`	bitwise right shift with zero extension assignment	

Type Conversion and Operator Precedence: Example

What is the type of the result of the following expressions in C? Evaluate them step by step and take into account operator precedence.

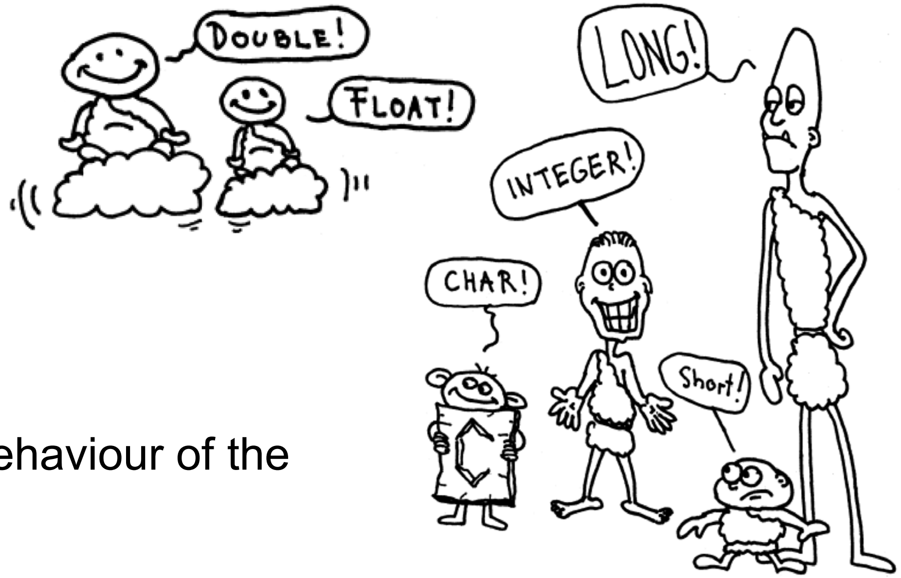
A: `4 + 1 * 2`

B: `3.0 / 2 + 3 / 2 * 4.0`

Example A yields an integer value of 6 (operator * has a higher precedence than the operator +).

Example B yields a type `double` with value 5.5. Sequence of evaluations:

```
3.0 / 2 + 3 / 2 * 4.0
  1.5   + 3 / 2 * 4.0
  1.5   +   1   * 4.0
  1.5   +     4.0
        5.5
```



More on Data Types in C: Qualifiers

Data types can be combined with qualifiers that will change the size / behaviour of the respective variable:

- `unsigned`
- `short`, `long`
- `const`

Example: `const unsigned short int myInteger;` declares an unsigned integer value with reduced memory size (2 bytes on most systems).

Constants

The `const` qualifier can be used to indicate that the value of a variable *should not* change. Although there are situations where it might nevertheless be possible to change it, so there is *no absolute guarantee* that the value actually *cannot* be changed. The C preprocess and the `#define` directive can be used alternatively to get constants that actually cannot change:

```
#define MY_CONSTANT 42.
```

An Example of Constants: The `stdbool.h` Header File

As we have seen, there is no boolean type in C natively. For convenience, there is the header file `stdbool.h` though, which can be included using

```
#include <stdbool.h>
```

and allows you to code as if there was a type bool in C.

This is achieved in the `stdbool.h` header file through simple precompiler directives replacing all occurrences of the string «true» in your code through 1 and replacing all occurrences of the string «false» through 0:

```
#define true 1

#define false 0
```

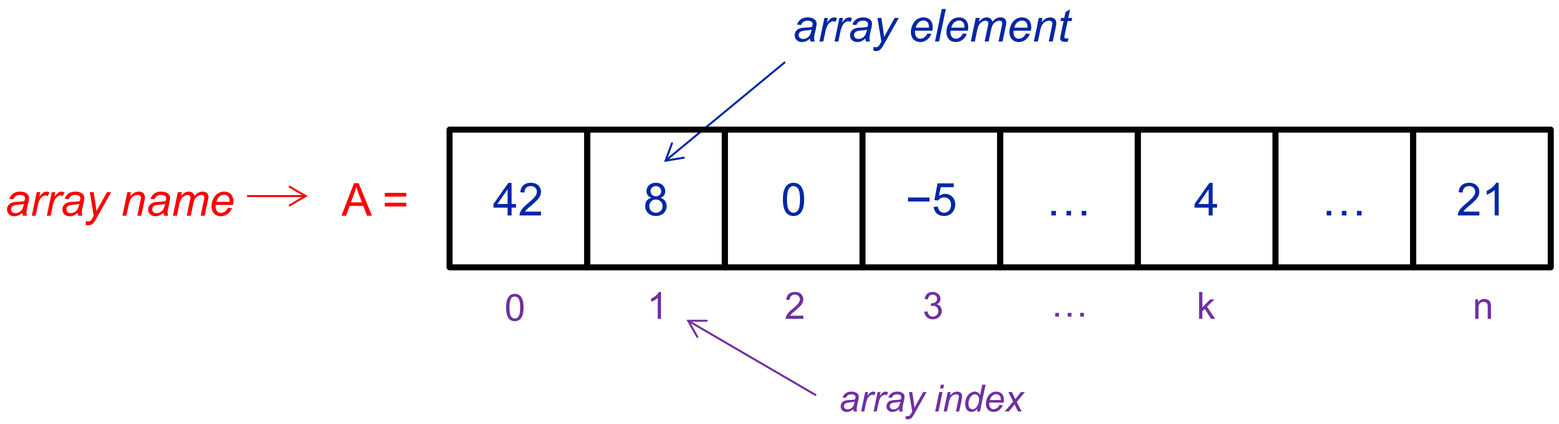
Introduction to C – Arrays, Strings, Functions, Input/Output

- Arrays
- «Strings»
- Functions
- Input and Output

Arrays: Introduction

Natively, there are no lists, tuples or dictionaries in C (strictly speaking, there are not even strings…). Instead, C offers arrays as basic data structure for storing simple collections. An array is a named and indexed collection of data items which are all of the same data type. They have the following properties:

- Entries are called «elements».
- Indexing starts at zero.



array element

array name → A =

42	8	0	−5	…	4	…	21
0	1	2	3	…	k		n

array index

The third element of the adjacent array is denoted and accessed as A[2] for example.

Arrays: Comparison to Lists in Python

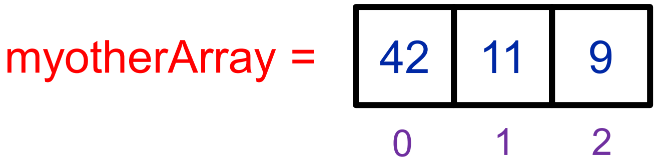
Arrays are on the first sight quite similar to lists in Python but there are some important differences:

- An array in C cannot grow or shrink, it has a fixed size which is set at compile time and cannot be changed during runtime. Once the array has been declared, its size is fixed and cannot be changed anymore.
- All elements an array have to be of the same type.
- Arrays provide no access control. Beware of out of bounds errors with arrays!
- There is no generally applicable method for retrieving the length of an array once it has been declared. (There is a workaround using the `sizeof` operator which cannot always be applied and / or there is a possibility to have a special value signaling the end of an array. The most famous example of the latter are strings as we will see later.)
- There is no equivalent to the negative indices used for string slicing in Python (where -1 is the last position, -2 the second last and so forth).

Arrays: Syntax / Declaration

Declaration and initialization examples:

```
int myFirstArray[3]; /* declare an integer array of size 3; content undefined */
double myInitializedArray[3] = {56.0, 23.0, 35.2};
int myOtherArray[] = {42, 11, 9}; /* size is automatically determined by compiler */
```



myotherArray =

42	11	9
0	1	2

Examples of accessing array elements:

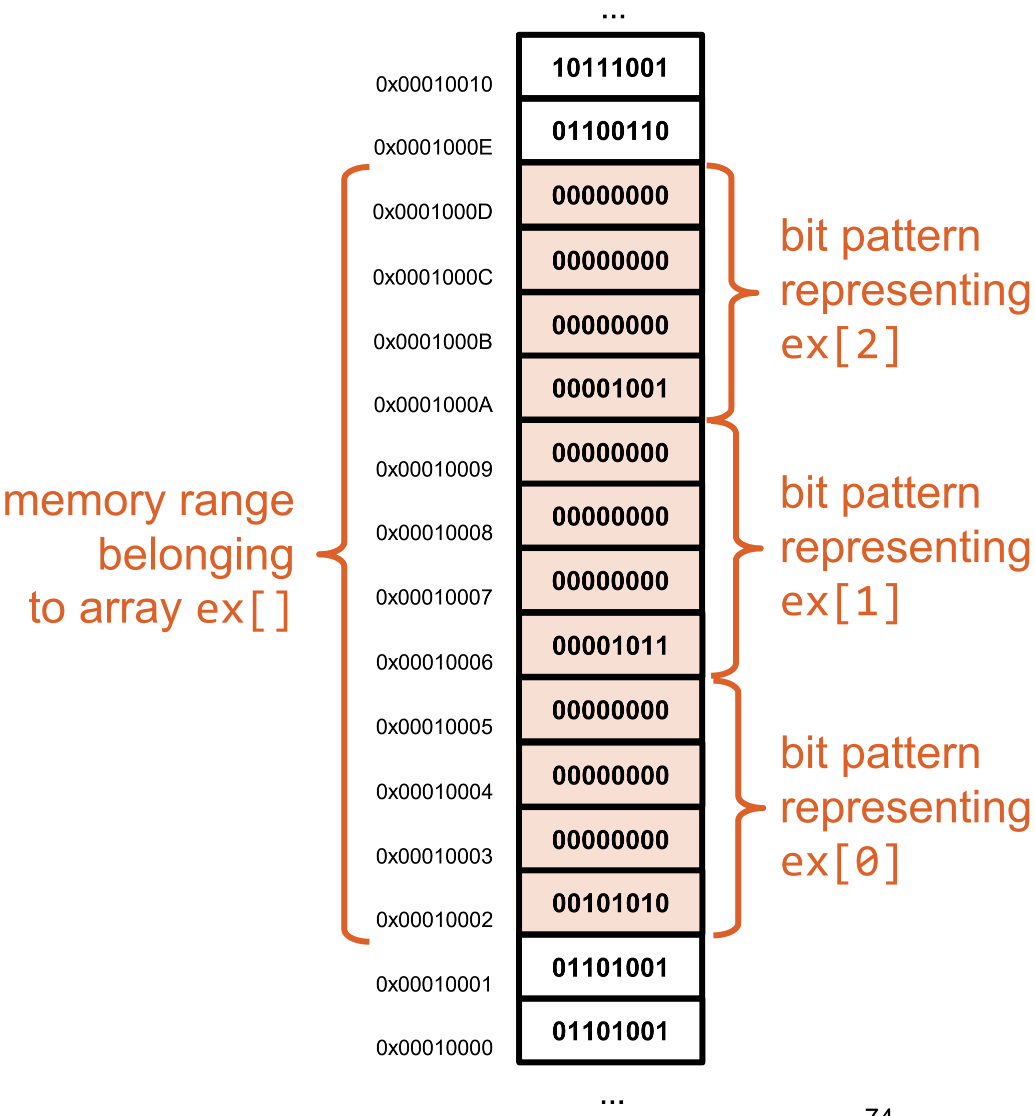
```
int myDate[] = {25, 2, 2021};
int month = myDate[1];
int someArray [17];
someArray[13] = 77;
someArray[14] = someArray[13];
const int ARRAY_SIZE = 42; /* needs to be constant */
float floatArray[ARRAY_SIZE];
```

Arrays in Memory

An array can conceptually be seen as a continuous section in memory (a portion of memory address space).

```
int ex[] = {42, 11, 9};
```

When an array of a certain data type is declared, memory is allocated for the number of elements times the width of the data type (in the example image: 3 elements with 4 bytes each for the integer data type). To get to the kth element we just have to remember the starting address (in the example: 0x00010002) and add k times the width according to the data type. (We will go into a more precise description when pointers were introduced in the middle of the semester.) This also explains why the size cannot be changed later on and the strange behaviour that might happen upon out of bound access.



	...
0x00010010	10111001
0x0001000E	01100110
0x0001000D	00000000
0x0001000C	00000000
0x0001000B	00000000
0x0001000A	00001001
0x00010009	00000000
0x00010008	00000000
0x00010007	00000000
0x00010006	00001011
0x00010005	00000000
0x00010004	00000000
0x00010003	00000000
0x00010002	00101010
0x00010001	01101001
0x00010000	01101001
	...

memory range belonging to array ex[]

bit pattern representing ex[2]

bit pattern representing ex[1]

bit pattern representing ex[0]

Arrays: Remarks

Arrays are *not* assignable, i.e. you *cannot* simply put the content of one array into another as follows:

```
char myChar[100];
char myOtherChar[] = {'H', 'E', 'L', 'L', 'O', '\0'};
myChar = myOtherChar; /* error, arrays are not assignable */  ✘ wrong!
```

If you need to put the content of one array into another, you could do it by hand and copy it element by element using a loop. There is the `memcpy()` function from the cstring library which can handle that for you efficiently (and the `strcpy()` function, also from the cstring library for strings).

Arrays: Example

Consider the following C code fragment:

```
int myArray[3] = {1, 2, 3};
int test = myArray[42];
```

Statement E is correct.

When accessing the 42nd element of the array with `myArray[42]`, the value which is found 42 integers sizes away from the start of `myArray`.

There is no access control in C arrays. It is therefore very important to make sure to not run over the boundaries of an array. (Off-by-one errors in loops involving arrays are a common source of errors.)

Which of the following statements regarding this fragment is true?

A: The compiler will throw an error.

B: There definitively will be an error message during runtime.

C: There will be no error message but the program will definitively crash when it is executed.

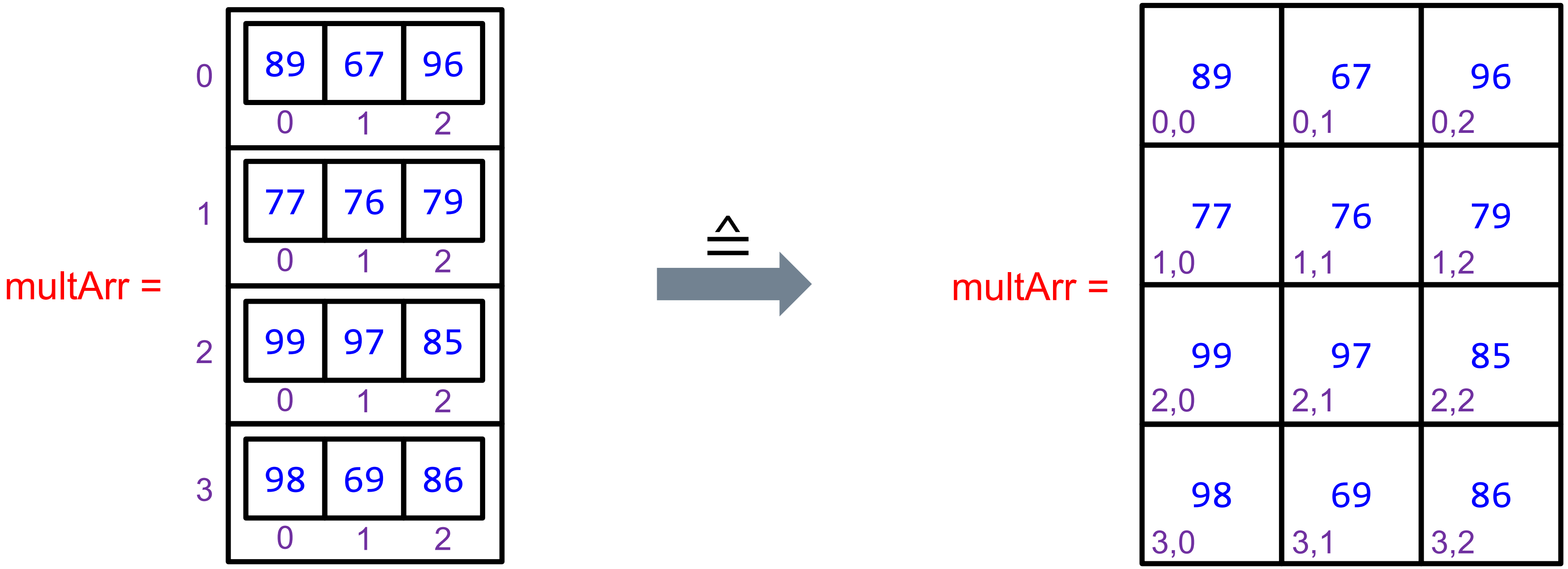
D: The IDE will certainly warn the user when he or she is typing these lines.

E: It can not be said what will happen when this code is executed. The behaviour is undefined and depends upon the current state of memory at the time of execution of the program.

Multi-dimensional Arrays: Structure

C has only one-dimensional arrays, but an element can contain any other thing (thus also arrays) which allows to simulate multi-dimensional arrays (nested arrays). Below is an example of a two-dimensional array:

```
int multArr[4][3];
```



89 0,0	67 0,1	96 0,2
77 1,0	76 1,1	79 1,2
99 2,0	97 2,1	85 2,2
98 3,0	69 3,1	86 3,2

Multi-dimensional Arrays: Code

Multi-dimensional arrays can be declared and initialized as follows:

```
int multArr[4][3];
multArr[0][0] = 89;
multArr[0][1] = 67;
multArr[0][2] = 96;
multArr[1][0] = 77;
multArr[1][1] = 76;
multArr[1][2] = 79;
multArr[2][0] = 99;
multArr[2][1] = 97;
multArr[2][2] = 85;
multArr[3][0] = 98;
multArr[3][1] = 69;
multArr[3][2] = 86;
```

rows index *columns index*

≙

number of rows *number of columns*

```
int multArr[4][3] = {
    {89, 67, 96},
    {77, 76, 79},
    {99, 97, 85},
    {98, 69, 86} };
```

≙

multArr =

89 0,0	67 0,1	96 0,2
77 1,0	76 1,1	79 1,2
99 2,0	97 2,1	85 2,2
98 3,0	69 3,1	86 3,2

Strings

There is no dedicated string data type in C. Instead, strings are represented in C by arrays of characters.

These arrays are expected to be null terminated (if this is not the case, bad things will happen in general).

Example of string declaration and initialization:

```
char myFirstString[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

…or equivalently (and much more conveniently):

```
char mySecondString[] = "hello";
```

'h'	'e'	'l'	'l'	'o'	'\0'
0	1	2	3	4	5

Note that the sample char array `myString` from above has length 6, i.e. it has one element more than the word «hello» has characters because of the null terminator. Further remarks:

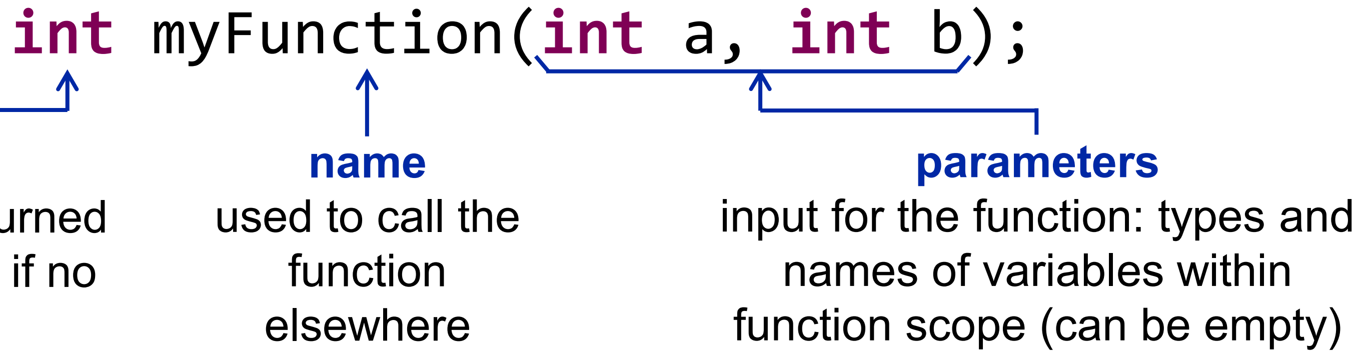
- Only double quotes are allowed when initializing a string like in the second example above.
- Note that strings are thus *not* immutable in C (in contrast to Python).

Functions

In C, functions have a return type (type which the return value needs to have). If a function should not return a value, there is the special type `void` (meaning nothing is returned).

In C, functions have to be declared before they can be used at another point in the code («forward declaration»). Note: Functions only need to be declared before used, but not actually defined (implemented); this declaration is called a function prototype.

Example of a function declaration:



```
int myFunction(int a, int b);
```

return type
type of the value which is returned by the function (may be void if no value is returned)

name
used to call the function elsewhere

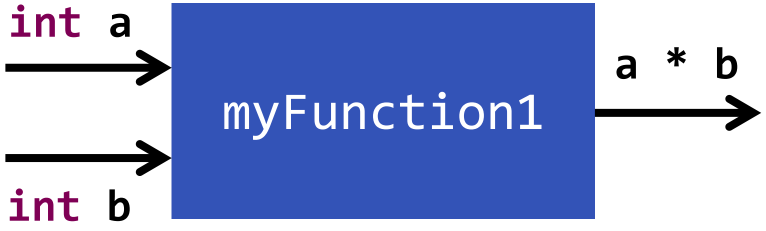
parameters
input for the function: types and names of variables within function scope (can be empty)

The name, return type, together with the ordered list of parameter types is called the signature of the function. Note that C does *not* support function overloading, i.e. it is *not* possible that there are two functions with the same name but different signatures (a typical feature provided by object-oriented languages).

Functions

Example of a function definition with return value:

```
int myFunction1(int a, int b) {
    return a * b;
}
```



Example of a function definition without return value:

```
void myFunction2(double x) {
    double y = x + 42.0;
    printf("%f", y);
}
```

The `main` Function

There is a special function named `main` which will be used as an entry point (start of control flow).

You will encounter several different signatures for the `main` function, e.g.

`int main()` or `int main(void)` or `int main(int argc, char* argv[])`

Some standards will also allow the main function to have void as return type, i.e. `void main()` or similar.

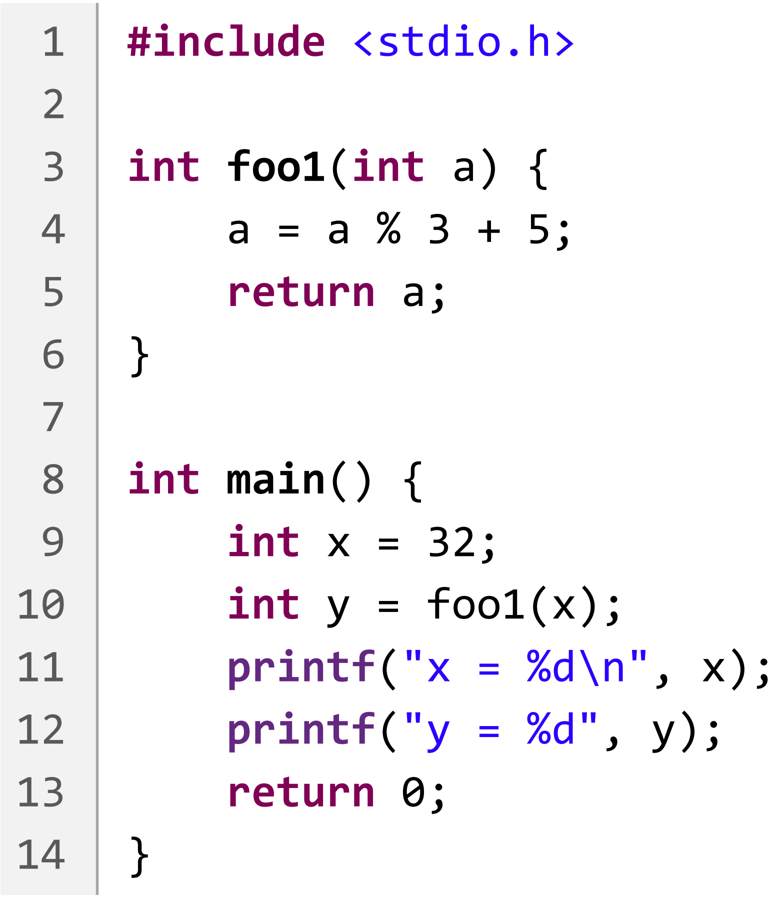
If the main function has return type `int` (e.g. `int main()`), there is the convention that it will return 0 to signal to the operating system (and other programs which have called it) that the code has terminated normally. Return values different from 0 indicate that the program has terminated erroneously (where different values may be used for different types of errors).

Functions and Memory

In memory, there is a special section where function calls are stored (kind of a «todo list» for function calls); this is called the «stack»; we’ll have closer look at this later on in the course.

Call By Value and Call By Reference (Pass By Value and Pass By Reference)

Consider the following code snippet: What will be the output?



```c
#include <stdio.h>

int foo1(int a) {
    a = a % 3 + 5;
    return a;
}

int main() {
    int x = 32;
    int y = foo1(x);
    printf("x = %d\n", x);
    printf("y = %d", y);
    return 0;
}
```

Call By Value and Call By Reference

Consider the following code snippet: What will be the output?

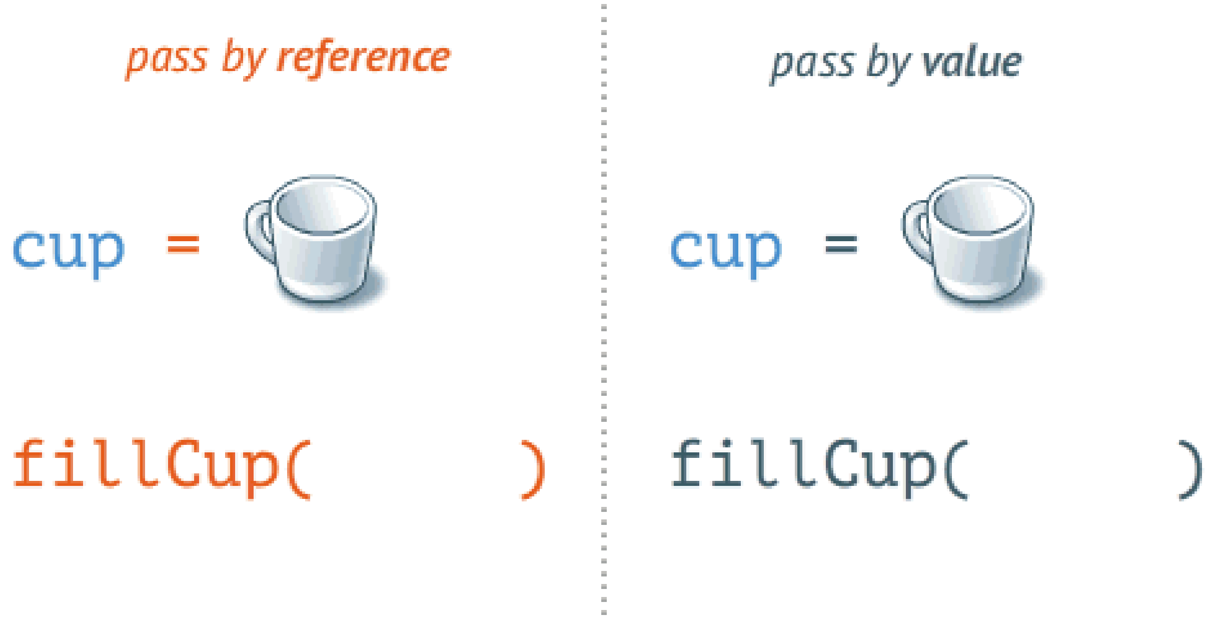
```c
#include <stdio.h>

int foo2(int a[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        a[i] = a[i] % 3 + 5;
        sum += a[i];
    }
    return sum;
}

```

```c
int main() {
    int x[] = {32, 33, 34};
    int y = foo2(x, 3);
    printf("x = %d, %d, %d\n", x[0], x[1], x[2]);
    printf("y = %d", y);
    return 0;
}
```

Call By Value and Call By Reference



(from https://www.mathwarehouse.com/programming/passing-by-value-vs-by-reference-visual-explanation.php)

Overview of Some Useful C Standard Libraries

C comes with a set of standard libraries with commonly used and helpful functions.

Some libraries which might be useful:

`cstdio`	required to perform input / output operations
`cstdlib`	general utility library
`cmath`	mathematical functions
`cstring`	functions for handling strings
`ctime`	time library

To use libraries within a program, the respective header files (.h) are included using the following syntax (for the example of cmath) which should be done at the top of the file (to honour forward declaration):

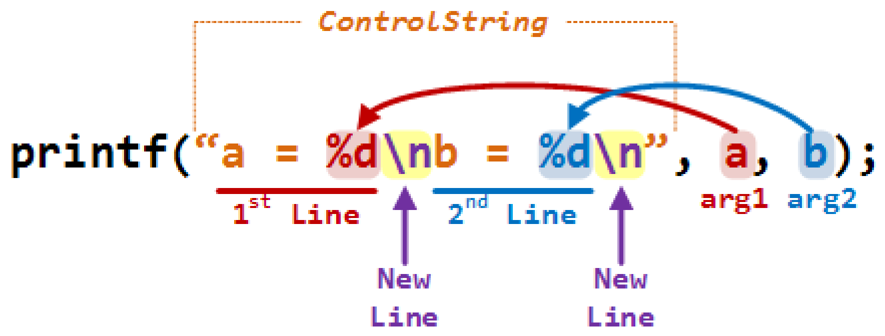
```
#include <math.h>
```

(Remark: Instead of angle brackets (<, >) as used above, there is also a syntax with quotes (") instead. If the latter is used, the compiler will look in the current directory for the given header file before going to the system paths.)

Input and Output

The `printf()` function is used for output.

- Is declared in `stdio.h`.
- Is used quite similarly to the `print()` function in Python.



```
printf(“a = %d\nb = %d\n”, a, b);
```

The `scanf()` function can be used for input.

- Is declared in `stdio.h`.
- Returns number of read entities on success, returns −1 otherwise.
- Very susceptible to buffer overflows; *do not use in real-world applications!*

Printf: Format String

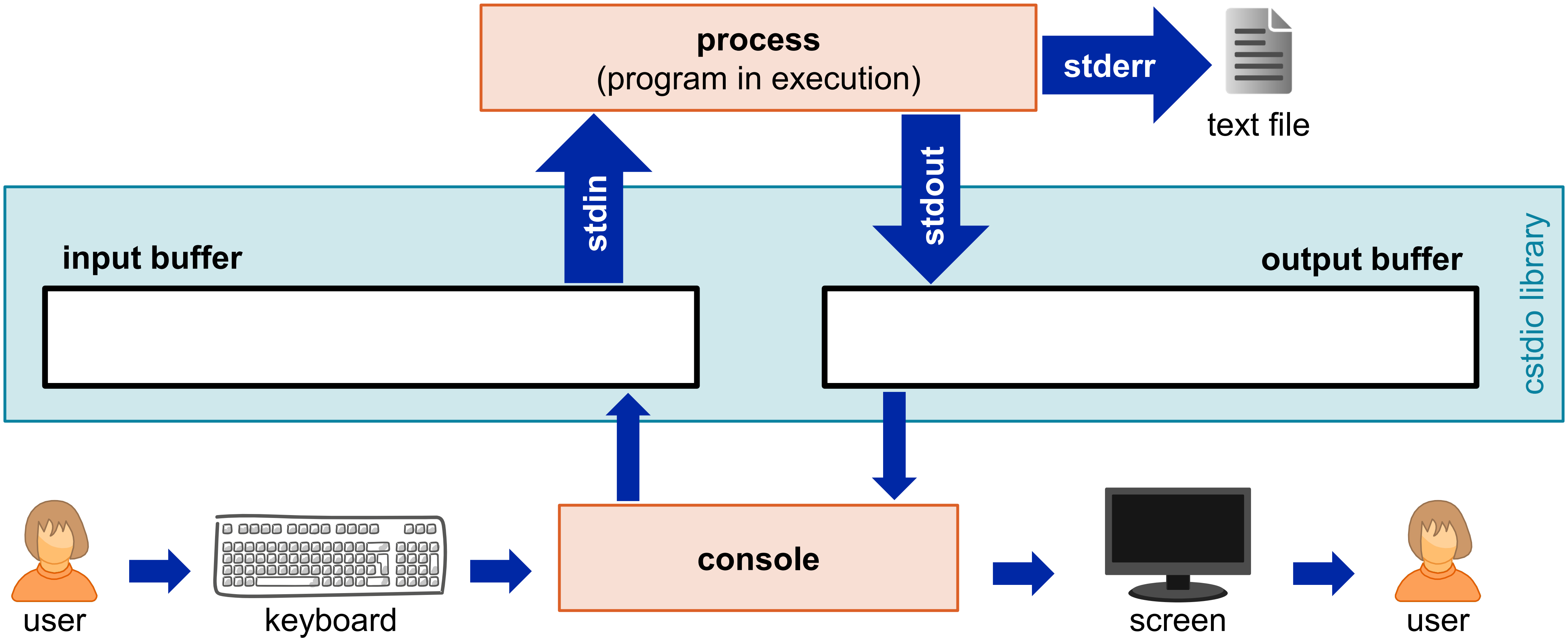
Non-exhaustive assortment of some commonly used format strings for the `printf` function:

Specifier	Description	Suitable Types
%c	single ASCII character	`char`
%d	decimal number	`int`
%e	scientific notation	`float`, `double`
%f	floating point number	`float`
%lf	floating point number	`double`
%s	strings	`char` array

Input / Output: Streams, Files, Buffers

- Programs need a way to communicate with the outside world. When a program is in execution (is a process), there are at start-up automatically several predefined connections established for this purpose. These are called data streams and have the following names / abbreviations:
  - stdin (standard input; for writing conventional input)
  - stdout (standard output; for writing conventional output)
  - stderr (standard error output; for writing diagnostic output)
- These streams are (usually) attached to the user’s terminal – and through this to the keyboard for inputs and the screen for outputs.
- These data streams are buffered streams. A buffer is a temporary storage containing data waiting to be processed (e.g. displayed to the screen or fed into a program).
- When stdin and stdout streams come from keyboard / go to screen (as it will be usually the case), they are line buffered which means that the content of the buffer will be transmitted to the respective process as a block when a new-line character (`'\n'`) is encountered.

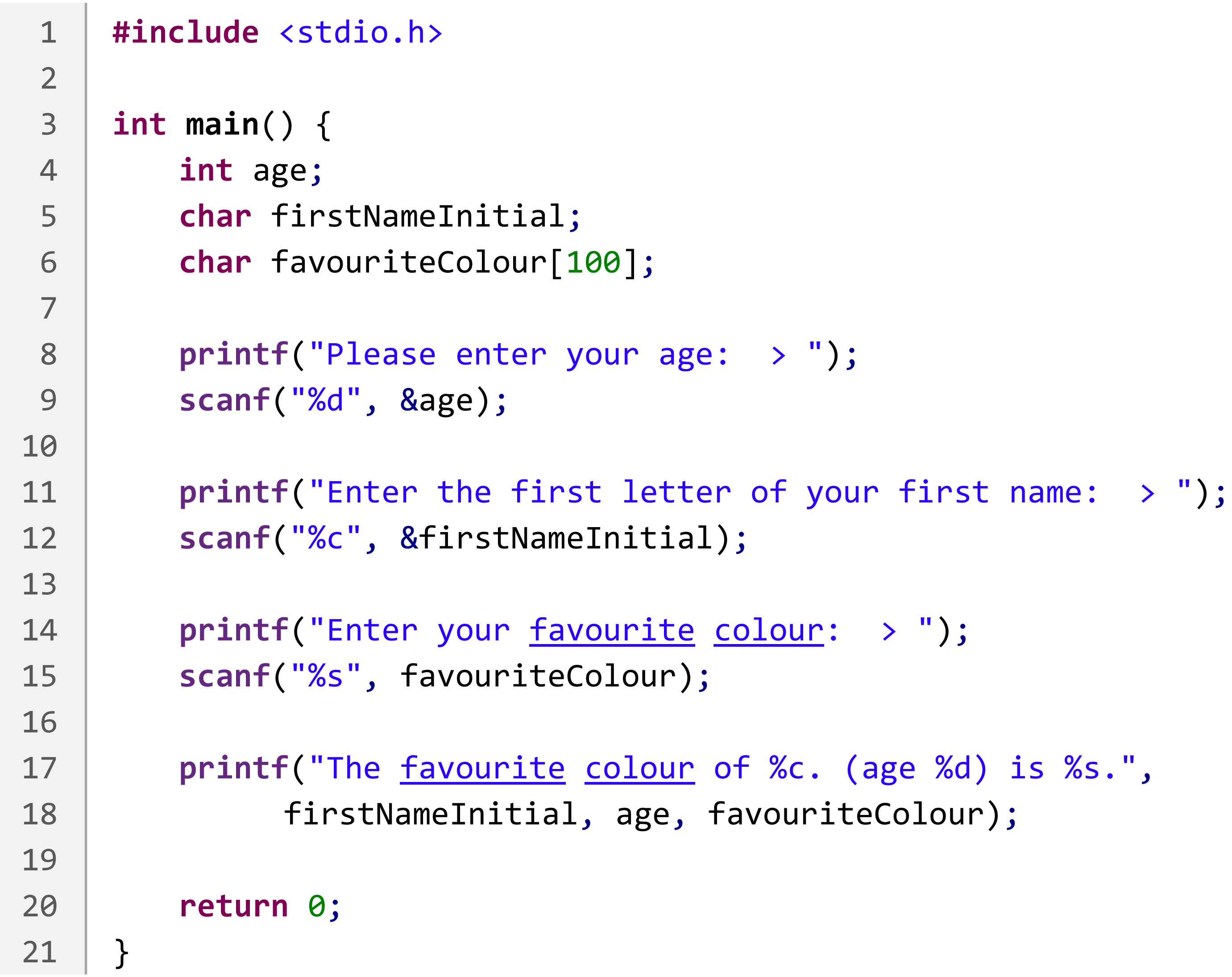
Input / Output: Streams, Files, Buffers



Input / Output Issue Example

When scanf is applied more than once in succession an issue with potentially surprising behaviour might arise.

Consider the following example and explain why it will *not* work as intended:



```c
#include <stdio.h>

int main() {
    int age;
    char firstNameInitial;
    char favouriteColour[100];

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

    printf("Enter the first letter of your first name:  > ");
    scanf("%c", &firstNameInitial);

    printf("Enter your favourite colour:  > ");
    scanf("%s", favouriteColour);

    printf("The favourite colour of %c. (age %d) is %s.",
          firstNameInitial, age, favouriteColour);

    return 0;
}
```

Outlook: Advanced Concepts when Working with C

Here are some topics which were not (yet) discussed in this short introduction. We will visit some of them later in the semester:

- Pointers, structs, enums
- Side effects
- Asserts
- Stack memory segment, stack frames
- Dynamical memory allocation, heap memory segment
- Buffer overflows
- Testing

Preview on Exercise 1

Tips and Tricks When Devising Algorithms

- In the beginning, think about a solution on a high level and write it down as a (pseudocode) blueprint which you can later use when you're actually implementing it in C.
- Make drawings!
- Use pens or your fingers to think about iterative algorithms operating over arrays.
- Make a table to analyze loops, keeping track of iteration variables and other relevant values.
- Don't forget to think about potential special cases. Test your code thorougly.
- Execute your code regularly, do not write lots of code and only then try out. Make baby steps in the beginning!

Exercises: General Technical Remark

Use a seperate file for each task. Don’t try to have include parts from other files.

Wrap-Up

- Summary
- Feedback
- Outlook
- Questions

Wrap-Up

- Summary

- Please take 5 minutes to answer these questions:
  - What were the most important points / insights of today's exercise session for you?
  - What things remained unclear or were confusing? About what do you want to know more or needs clarification?
  - What I wanted to say anyway…

Outlook on Next Thursday’s Lab Session

Next tutorial: Wednesday, 02.03.2022, 14.00 h, BIN 0.B.06

Topics:

- Review of Exercise 1
- Further Intro to C (?)
- Basic Sorting
- Preview to Exercise 2
- …
- … (your wishes)

Questions?

Thank you for your attention.