



Informatics II

Tutorial Session 1

Tuesday, 21st of February 2020

Introduction to the C Language

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

14.15 – 15.45

Y35-F-32



Agenda

- Organisation and Expectations
- Introduction to the C Language



Organisation and Expectations

- Contact Information
- Language in Tutorials
- Lab Schedule Overview
- Exams and Grading
- My Thoughts on the Tutorials
- Your Expectations



Contact Information

Christoph Vogel

christoph.vogel@uzh.ch

Webpage for the tutorials and additional materials:

<http://www.physik.uzh.ch/~vogelch/algodat>

If you have questions:

- Preferably ask me right away during the exercise sessions.
- Send me an email.
- Use the OLAT forum.

Language in the Tutorials

- 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 **written artifacts** (including these slides, obviously) will be in **English** (and some of them will possibly also be available in German).



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



Lab Schedule Overview

Date	No.	Room	Exercise	Content	
Fri, 21.02.	1	Y35-F-32	Ex1	Introduction to C, exercise 1	
Fri, 28.02.	2	Y35-F-32	Sol1, Ex2	Recursion, ...	
Fri, 06.03.	3	Y35-F-32	Sol2, Ex3	Asymptotic Complexity and Correctness, ...	
Fri, 13.03.	4	Y35-F-32	Sol3, Ex4	Divide and Conquer (Merge Sort), Recurrences	
Fri, 20.03.	5	Y35-F-32	Sol4, Ex5	Heap Sort, Quick Sort	← Mon, 23.03. Midterm I
Fri, 27.03.	6	Y35-F-32	Sol5, Ex6	Pointers, Linked Lists	
Fri, 03.04.	7	Y35-F-32	Sol6, Ex7	ADTs (Stack, Queue)	
tbd	8	tbd	Sol7, Ex8	Binary Trees, Binary Search Trees	← Spring break
Fri, 24.04.	9	Y35-F-32	Sol8, Ex9	Red-Black Trees	
tbd	10	tbd	Sol9, Ex10	Hash Tables, Hash Functions	← Mon, 27.04. Midterm II
Fri, 08.05.	11	Y35-F-32	Sol10, Ex11	Dynamic Programming	
Fri, 15.05.	12	Y35-F-32	Sol11, Ex12	Graphs (DFS, BFS)	
Fri, 22.05.	13	Y35-F-32	Sol12	tba	← Wed, 27.05. Final Exam



Exams and Grading

- No mandatory attendance in lectures nor tutorials nor midterms (but missing them may be at your own peril, in case of exams, grade 1 will result).
- No mandatory hand-in of exercises, no correction and no grading of exercises. Sample solutions for all exercises will be provided.

Final exam

60% to 100% of your grade

Midterms

2 instances

0% to 40% of your grade

Labs

13 sessions

voluntary, recommended

Exercises

12 instances

voluntary, recommended



Exam Regulations

- There will be **two midterm exams** and a **final exam**. All of them will be **on paper**.
 - At each exam for Informatics II (Midterms, Final Exam) you are allowed to use one A4 sheet with notes (both sides, hand written / printed / photocopied), a pocket calculator without text storage / memory (e.g. TI-30 XII B/S). More details will follow.
- The **midterm dates** are as follows (more information will be distributed as the midterms approach):
 - Midterm I: Monday, March 23, 12:15 – 13:45 h
 - Midterm II: Monday, April 27, 12:15 – 13:45 h
- Authorative information about the **final exam** will be provided directly by the deanery, see <https://www.oec.uzh.ch/de/studies/general/exams/assessment.html>.



Exam Grading

- All exams will be **graded from 1 to 6** using quarter grades (and applying «round half» up as a tie-breaking rule, e.g. a results of 3.875 and higher will round up to 4).
- The midterms are an **opportunity to improve** your grade but completely **optional**. If you visit and achieve a grade which is better than your final exam grade, then the midterms will count 20% each towards your course grade. If the grade received in a midterm is worse than your grade in the final exam, the respective midterm will simply be ignored.
 - If you are **absent** in one or two **midterms**, you'll receive **grade 1** for this exam, regardless of the reasons (justifiable due to illness or military service or simple no-show).
 - There will be **no repetition exams**.
 - I strongly recommend to attend the midterms.

Exams and Grading Calculation: Examples



Alice



Bob



Mallory



Lucky



Oscar

<i>Midterm 1:</i>	5.50	3.00	4.00	4.25	6.00
<i>Midterm 2:</i>	3.00	1.00 (absent)	4.25	4.25	6.00
<i>Final exam:</i>	4.50	5.00	3.50	3.50	2.25
<i>Weighted result:</i>	4.70	5.00	3.75	3.80	3.75
<i>Course grade:</i>	4.75 (pass)	5.00 (pass)	3.75 (fail)	4.00 (pass)	3.75 (fail)

Note: Experience from previous semester shows that extreme cases like Bob or Oscar do not tend to happen.



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 **exams** 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. Distributed materials should be sufficient for learning the contents; if you like to learn from books, the suggested reading (CLRS) is recommendable in my opinion.



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. At the same time, learning a new programming language can be time consuming and/or frustrating.

Important note especially for students repeating this lecture: In this year, a **much strong emphasis** will be placed on **learning the C language**. In particular, you will be **required** mandatorily to **provide solutions in exams as C code** to be awarded full points (just giving pseudocode will not be sufficient).



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.



My Thoughts on the Tutorials

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, ...

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

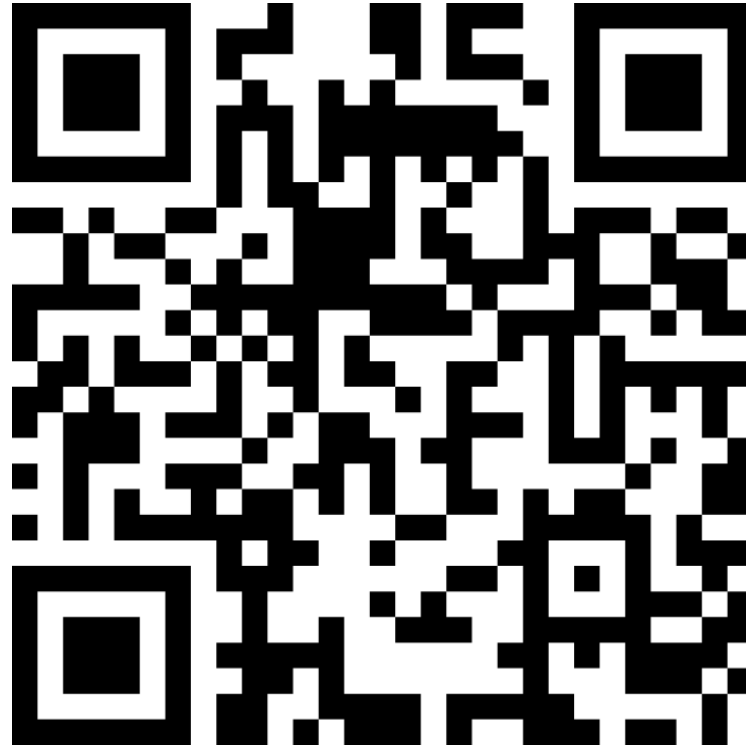
- My goal is to create true additional value and that it will be worth your time attending the lab sessions.
- I'll try to design the tutorials such that you will never leave the session and say that it was a waste of your time. If you think that I did not achieve this, please speak to me and say what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

I tend to overshoot time. I'll really try to stay within the time frame.



Your Expectations

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



Results: <https://app.klicker.uzh.ch/sessions/public/5e4e4945982331000b74a7a0>



Typical Layout of a Tutorial Session

Proposal for a prototypical tutorial session:

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



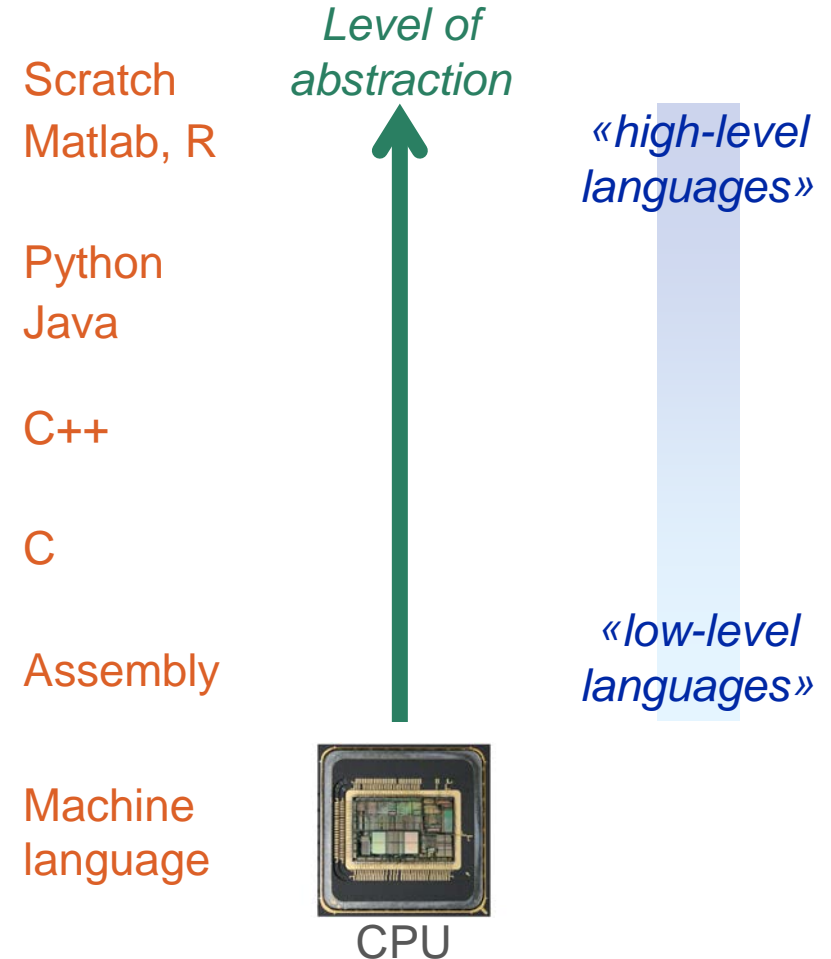
Introduction to C – Basics

- 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.

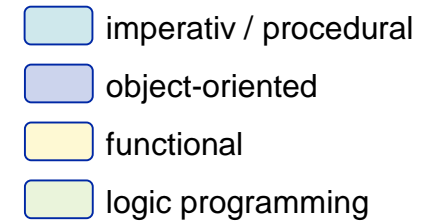
C as a Programming Language

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

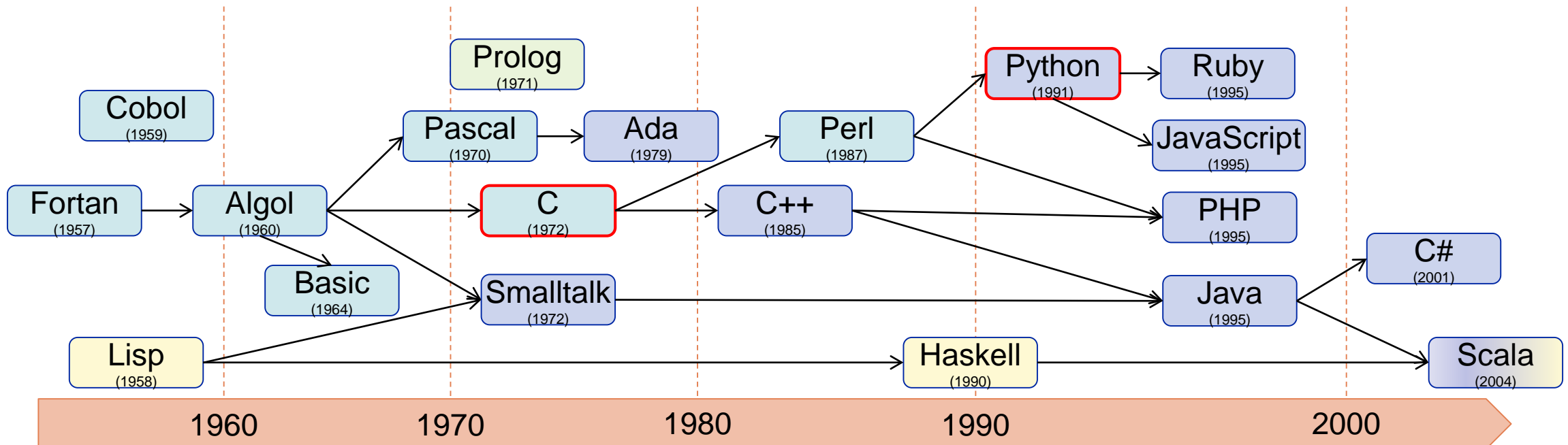


C as a Programming Language

Many modern programming languages are strongly influenced syntactically by C.



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):

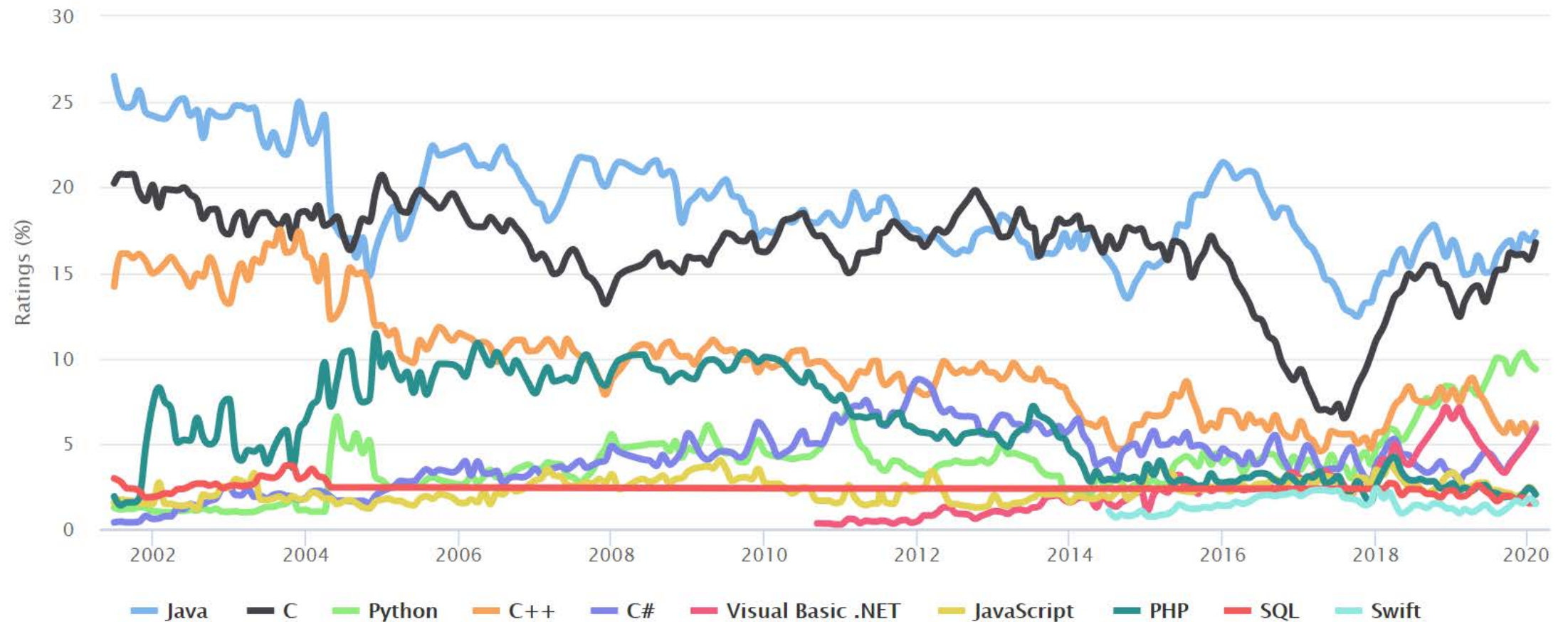
- Operating systems (kernels), e.g. macOS, Windows, Unix, Linux,
- Java Virtual Machine, Python Internals (CPython),
- Database Management Systems, e.g. PostgreSQL, MySQL,
- Embedded Systems (washing machines, Mars rovers, ...),
- High Performance Computing,
- ...



PostgreSQL



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

```
1  /* my first program in C */
2  #include <stdio.h>
3
4  int main() {
5      int number;
6
7      printf("Enter an integer:  >> ");
8      scanf("%d", &number);
9
10     printf("Your input was %d\n", number);
11
12     if (number > 0 && number % 2 == 0) {
13         printf("Your input was valid.\n");
14     } else {
15         printf("Your input was not valid.\n");
16     }
17
18     return 0;
19 }
```

Compile: gcc myFirstProgram.c -o myFirstProgram

Execute: ./myFirstProgram

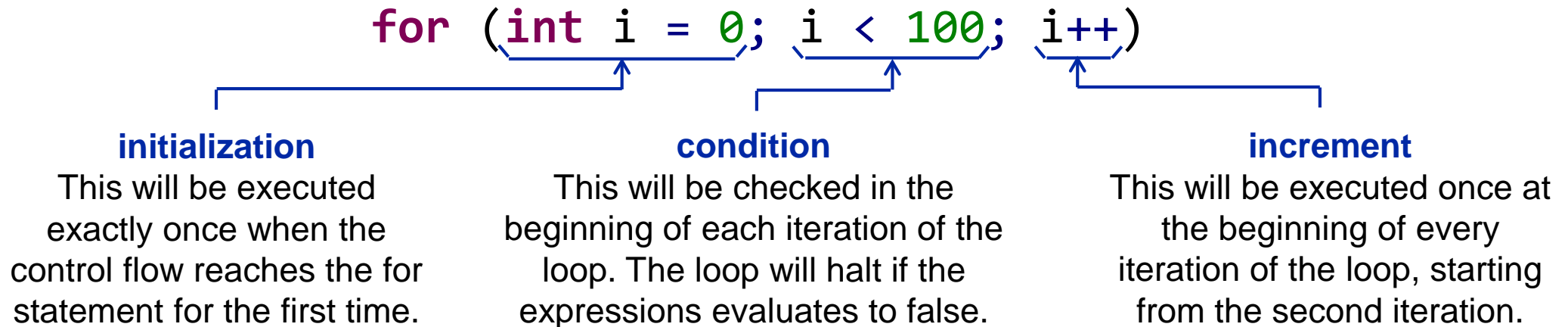


Python vs. C: Syntax

- 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.
- ...

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**.



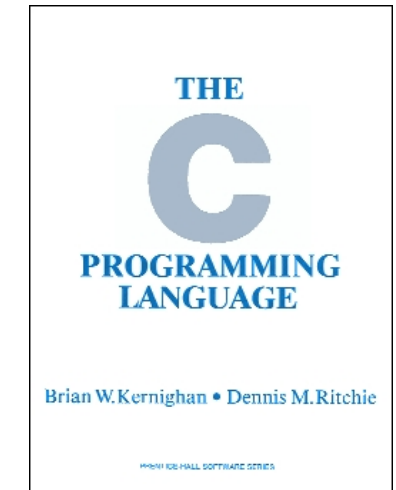
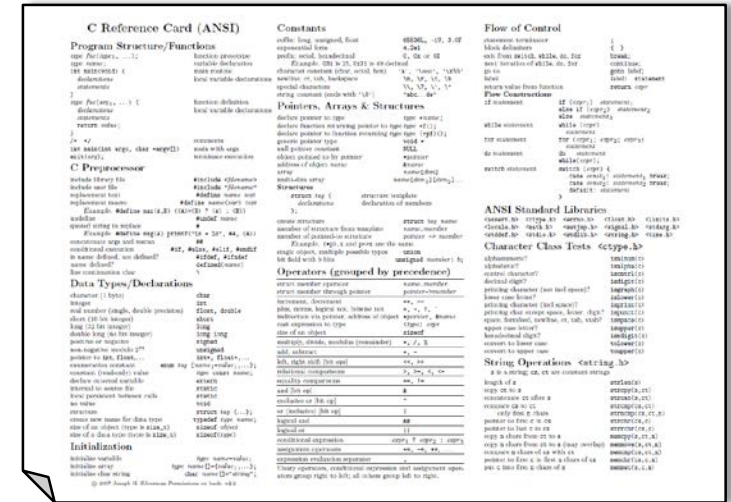
- All three parts of the header can be arbitrary expressions in principle. It is possible to have complex statements within the head of the for loop (use it hesitantly because it can get difficult to understand).
- Any of the three parts of the header can be omitted (the semicolons need to stay).

C vs Python: Loops

	Python	C
<i>Count up from 0 to 99 one by one:</i> 0, 1, 2, ..., 99	<pre>for i in range(0, 100): #...</pre>	<pre>for (int i = 0; i < 100; i++) { /* ... */ }</pre>
<i>Count down from 1 to 0 one by one:</i> 99, 98, 97, ..., 0	<pre>for i in range(100, -1, -1): #...</pre>	<pre>for (int i = 99; i > -1; i--) { /* ... */ }</pre>
<i>Count up from 0 to 99 with a step width of 3:</i> 0, 3, 6, ..., 99	<pre>for i in range(0, 100, 3): #...</pre>	<pre>for (int i = 0; i < 100; i += 3) { /* ... */ }</pre>
<i>Iterate all elements of a data structure:</i>	<pre>list_of_words = ["hi", "my", "friend"] for word in list_of_words: #...</pre>	<p>(a bit more complicated; we'll see later how this is done in C; there is no «foreach» like expression)</p>

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



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.



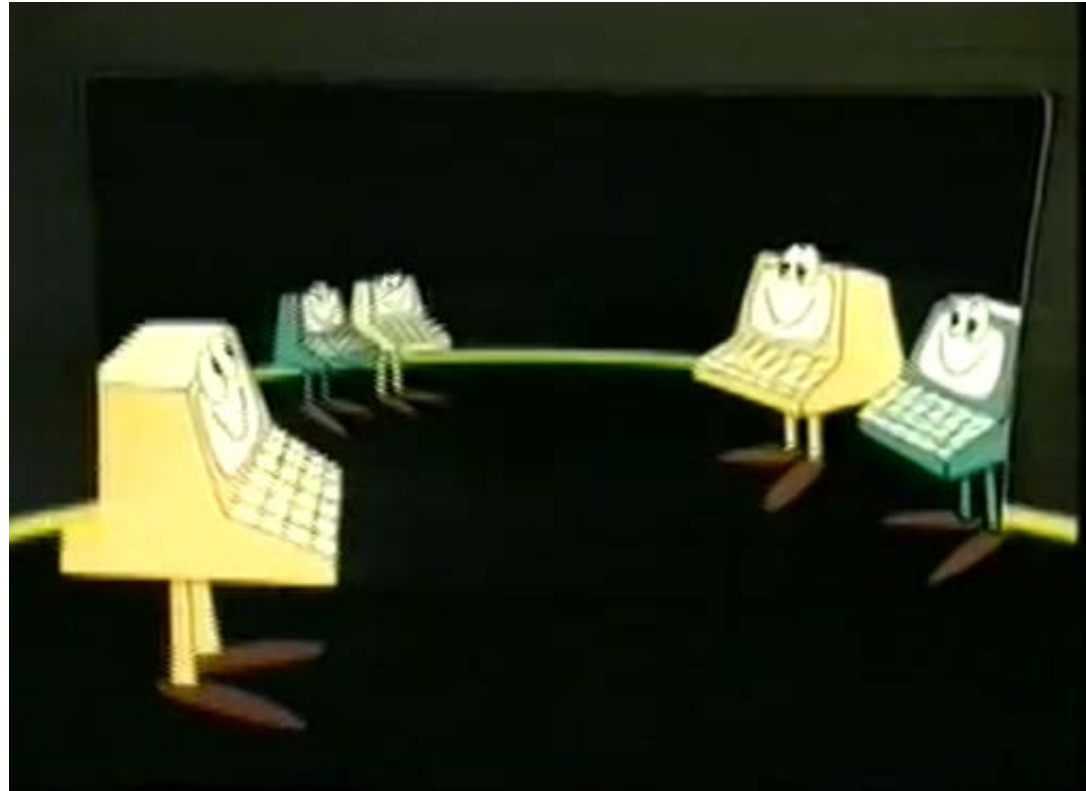


Compiler and Interpreter

- 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).

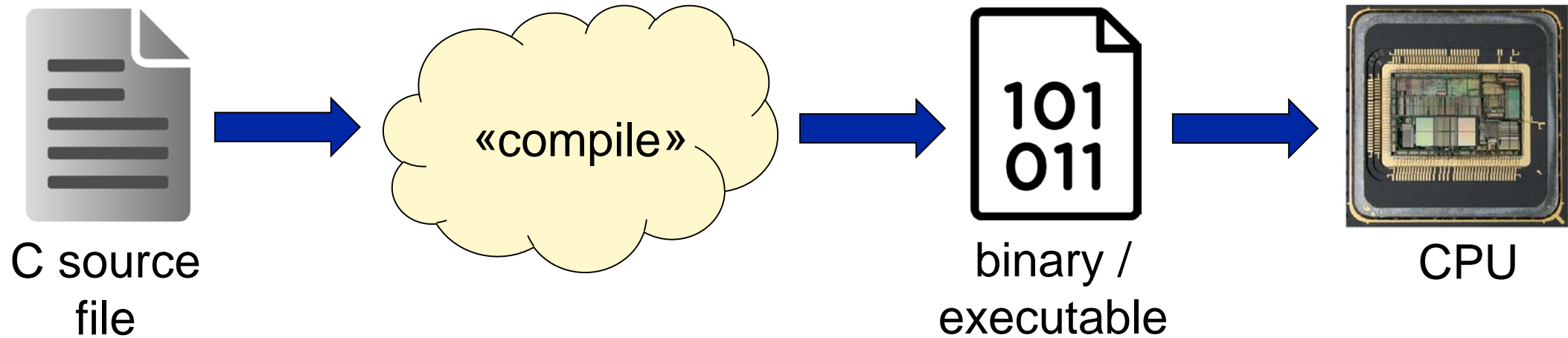
Compiler and Interpreter



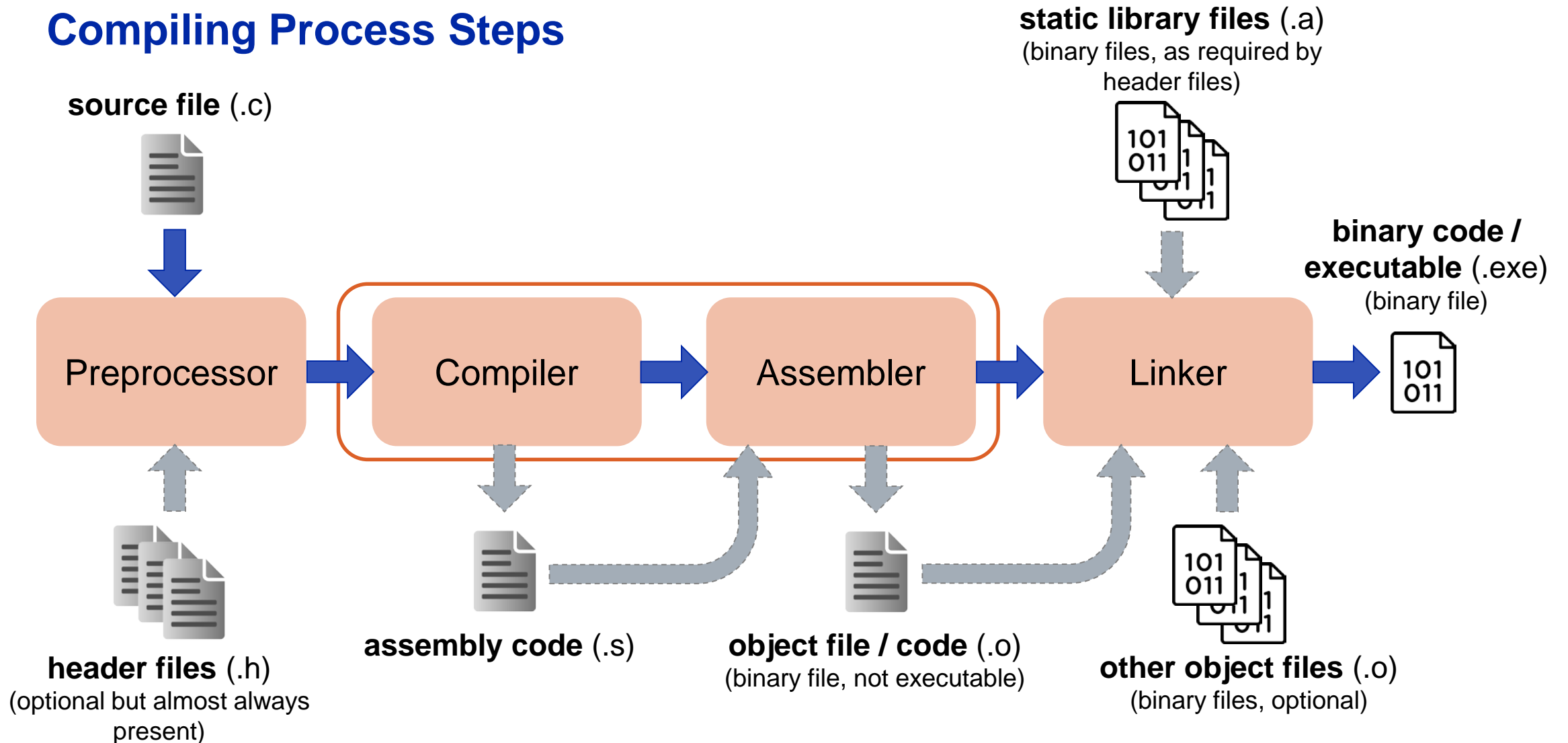
from the Canadian television series «Bits and Bytes», episode 6, from the year 1983

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

A Closer Look at the Compiling Process







Compiling Process Steps



Compiling on the Console

On the console, C code can be translated into an executable binary using statements similar the 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](#).

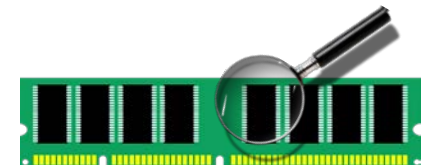
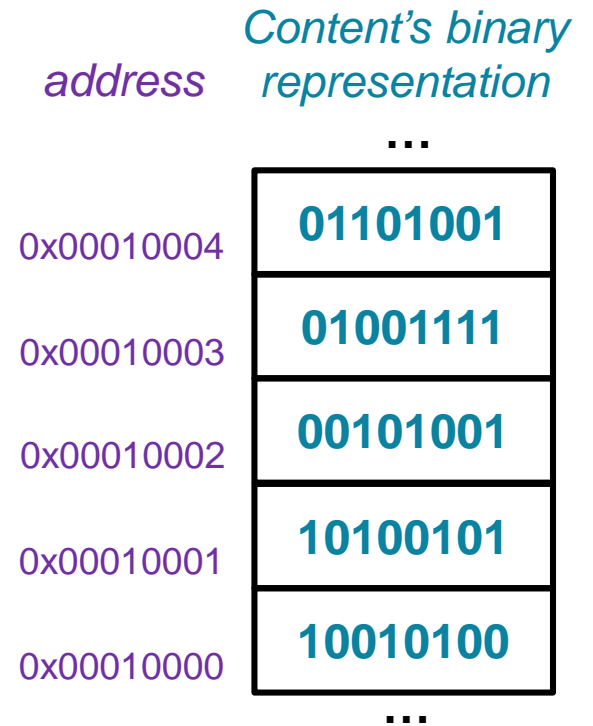
Memory in C

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

Today's computer hardware will typically have **byte addressable** memory architecture. This means that conceptually 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 typically 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.





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

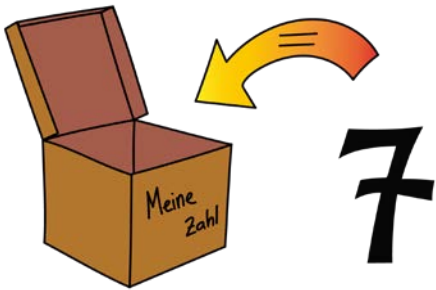
- Data Types and Their Sizes
- ASCII Table
- Variable Declaration and Initialization, Memory Allocation
- Type Conversions, Casts
- Operators
- Qualifiers

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.

In C code, each variable has a unique name which makes it easier to refer to the reserved memory.

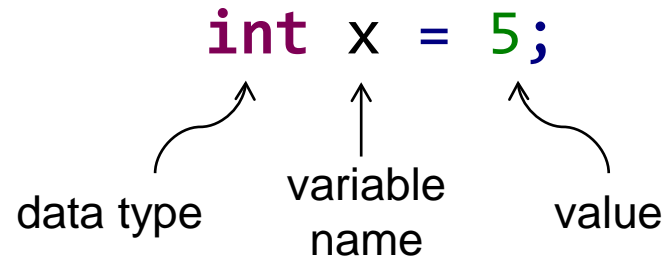


Data Types, Variable Declaration and Initialization

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:

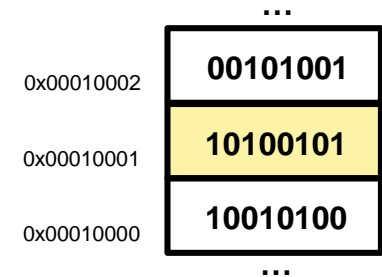
`int x = 5;`

data type variable name value



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).



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 has 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 is 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?

```
1  #include <stdio.h>
2
3  int main() {
4      int a = 100000;
5      int b = 200000;
6      int c = a * b;
7
8      printf("a * b = %d", c);
9      return 0;
10 }
```

- A: Does not compile
- B: Run time error
- C: $a * b = 20000000000$
- D: $a * b = -1474836480$

Data Types in C: Danger Zone

Be careful when using floating point types.

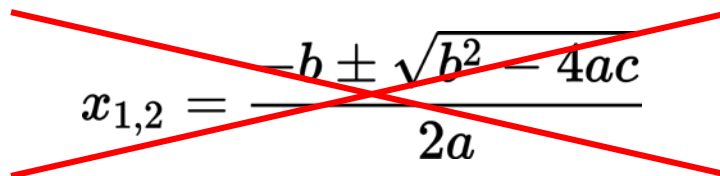
- *Never* make comparisons using floating point types (at least without controlling for roundoff errors).
- Do not use floating point types to represent currencies.

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

```
1  #include <stdio.h>
2  int main() {
3      float x = 0.0;
4      for (int i = 0; i < 10000; i++) {
5          x = x + 0.1;
6      }
7      printf("Result: %f", x);
8      return 0;
9  }
```

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.


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

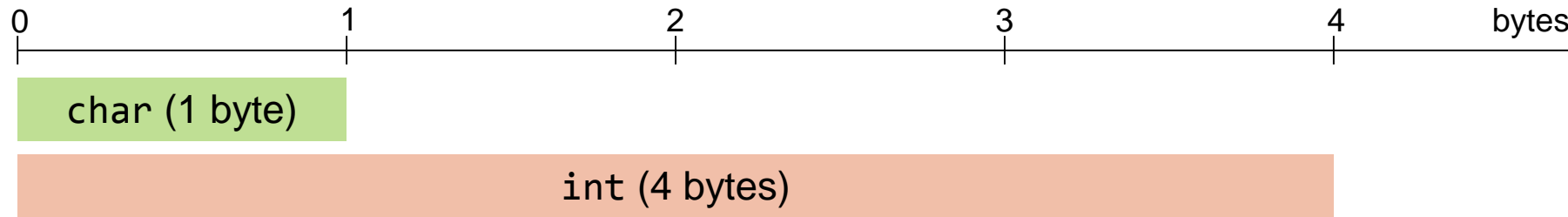
Data Types: Bad Examples

- 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?

```
1  int a = 0;  
2  char b = 'A';  
3  a = b;  
4  printf("%d", a);  
5  a = -2147424447;  
6  b = a;  
7  printf("%c", b);
```

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.
- 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 losing information.

Example of an explicit down cast:

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

Operators in C

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?

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

- 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).

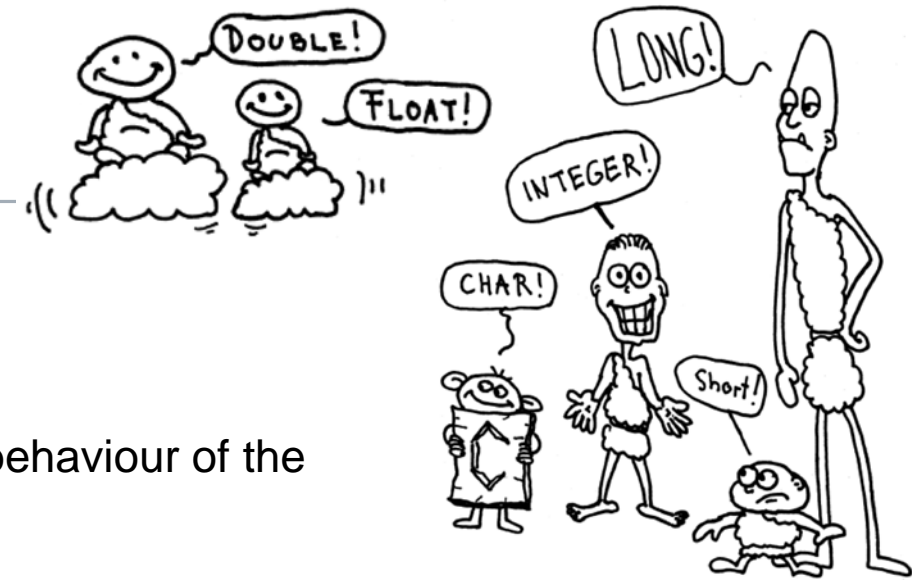


Type Conversion and Operator Precedence: Example

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: $3.0 / 2 + 3 / 2 * 4.0$



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 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
```



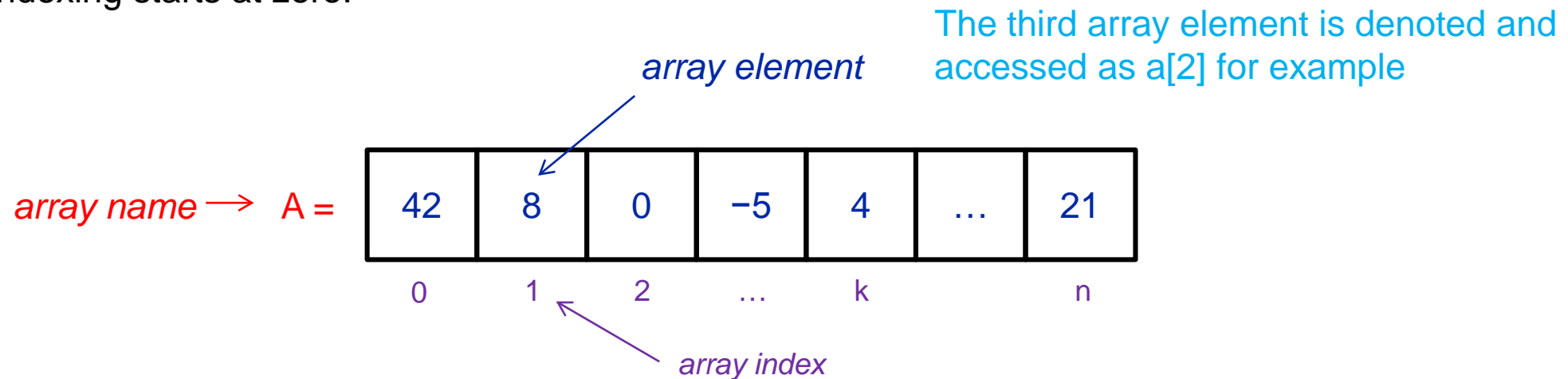

Introduction to C – Arrays, Strings, Functions, Input/Output

- 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.



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 */
```

Examples of accessing array elements:

```
int palindromicDate[] = {20, 2, 20};
int month = palindromicDate[1];
int someArray [17];
someArray[13] = 77;
someArray[14] = someArray[13];
const int ARRAY_SIZE = 42; /* needs to be constant */
float floatArray[ARRAY_SIZE];
```

myotherArray =

42	11	9
----	----	---

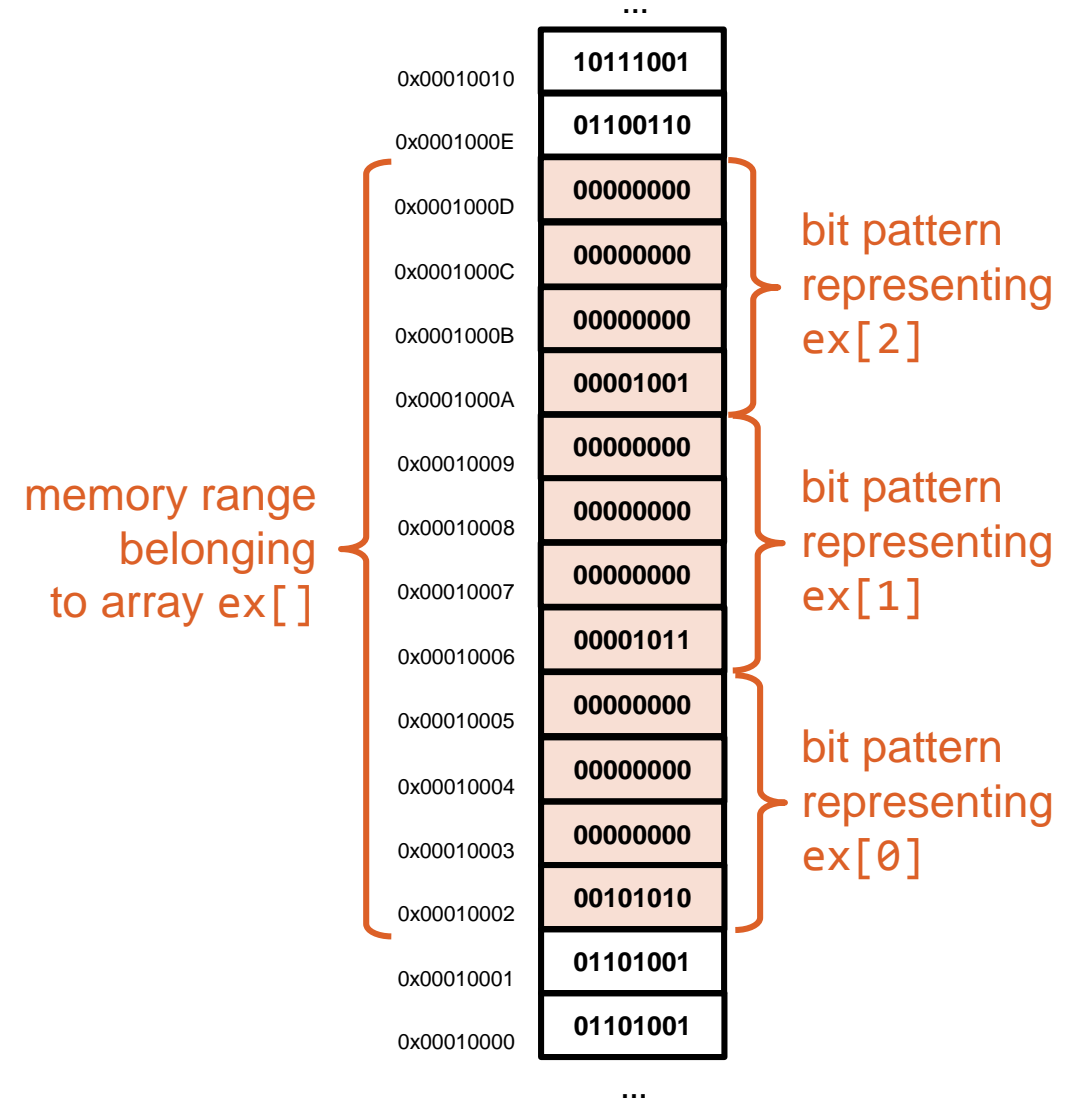
 0 1 2

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 k^{th} 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 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.



Arrays: Example

Consider the following C code fragment:

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

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.

Strings

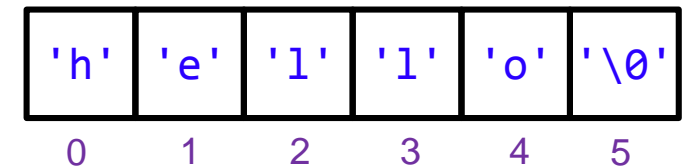
There is no dedicated string data type in C. Instead, strings are represented in C by **arrays of characters**. These arrays are expected to be **null terminated** (if this is not the case, bad things will happen in general).

Example of string declaration and initialization:

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

...or equivalently (and much more conveniently):

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



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:

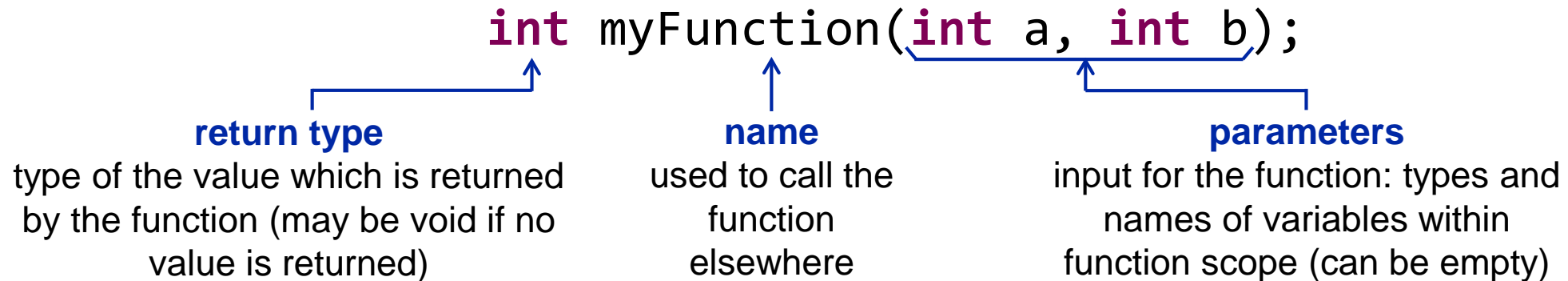
- Only double quotes are allowed when initializing a string like in the second example above.
- Note that strings are thus *not* immutable in C (in contrast to Python).

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 be declared before used, but not actually defined (implemented); this declaration is called a **function prototype**.

Example of a function declaration:

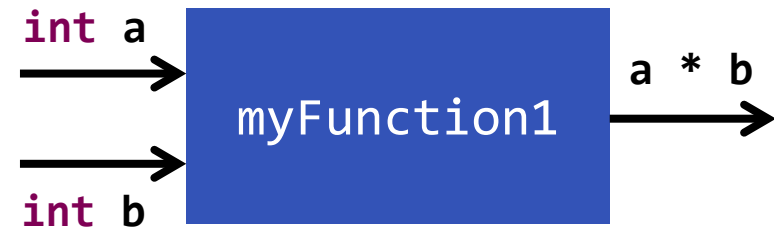


(The name, return type, together with the ordered list of parameter types is called the **signature** of the function.)

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) {  
    int y = x + 42;  
    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[])`

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



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

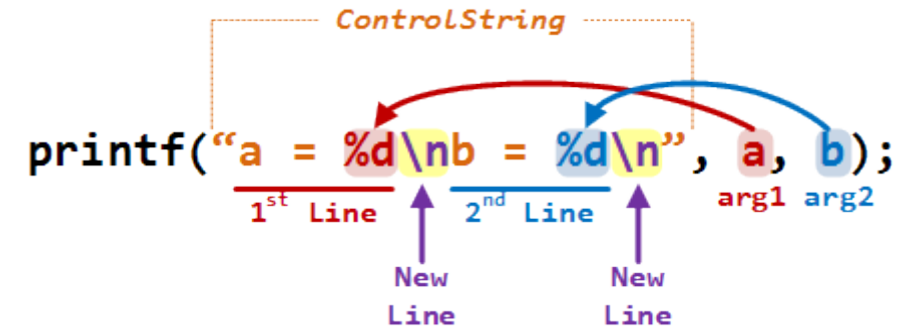
Input and Output

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

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

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!



The diagram illustrates the components of a `printf` format string. The string is `"a = %d\nb = %d\n"`. A dashed orange box labeled *ControlString* encompasses the entire string. The string is divided into two parts by a red line: the first part, `"a = %d\n"`, is labeled *1st Line* in red, and the second part, `"b = %d\n"`, is labeled *2nd Line* in blue. Two purple arrows point to the `\n` characters in each part, both labeled *New Line*. The arguments `a` and `b` are labeled *arg1* and *arg2* respectively. Red and blue curved arrows connect the `%d` placeholders to their corresponding arguments.

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

Printf: Format String

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



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
- Call by value, call by reference
- Side effects
- Asserts
- Stack memory segment, stack frames
- Dynamical memory allocation, heap memory segment
- Multidimensional arrays
- Buffer overflows
- Testing



Preview on Exercise 1

- Task 1: Determining String Length by Foot
- Task 2: Palindromes
- Task 3: Counting the Number of Occurrences of Integers in an Array
- Task 4: Sorting Even and Odd Integers in an Array Separately

Exercise 1 – Task 1: Determining String Length By Foot

A string with whitespaces can be read from the console as follows:

```
char input[100];  
scanf ("%[^\\n]*c", input);
```

(Alternatively, you may assume that the input string will not contain any whitespaces).

- Remember that strings in C are character arrays which are null terminated (having the character '\\0' at the end).
- Think about how big a character array has to be in order to enable the storage of a string with 1000 characters.
- Think about potential special cases (e.g. empty string).



Exercise 1 – Task 2: Palindromes

- You need to include the `stdbool.h` header file in order to use “bool” as a type.
- You may, again, assume that the string will be no longer than 1000 characters.
- You can read in the input string as in task 1.
- It can be assumed that the solution should be case sensitive, i.e. «Anna» as an input will be evaluated *not* to be a palindrome.
- Think about possible special cases, e.g. the empty string. Note that the empty string is usually considered as a palindrome (it’s the same when read from back and from front) and so is a single letter string.
- This task can be solved iteratively or recursively. As an supplemental exercise, you may try to implement a solution both ways.

Exercise 1 – Task 3: Counting the Occurrences of Integers

A series of integers can be read into an array in the way requested in the task description as follows:

```
int a[1000];
int i = 0;
while (scanf("%d", &a[i]) == 1) {
    i++;
}
scanf("%*s");
```

With the above code snippet, the reading will continue as long as the input can be parsed as an integer value. At the end, the variable `i` will contain the number of integers which have been read. The additional call of `scanf` after the while loop is there to clean up the input buffer. Also, note that there needs to be an ampersand (&) before the variable of the array here (in contrast to tasks 1 and 2).



Exercise 1 – Task 3: Counting the Occurrences of Integers

- You may assume that arrays A and B each contain at most 1000 integers.
- Remember that you have to inform the function about the size of the input arrays, i.e. the prototype of the function has to have the following form (or similar): `void countPairs(int A[], int nA, int B[], int nB)`
- You only have to print the resulting pairs, no need to store them in memory.
- As always, think about possible special cases and the behaviour of your program when they apply.



Exercise 1 – Task 4: Sorting Even and Odd Integers Separately

- You can read in the integers the same way as in task 3.
- Again, you can assume and define an upper size for the input array.



**Universität
Zürich** UZH

Institut für Informatik

Roundup

- Summary
- Outlook
- Questions



Summary

In today's introduction to the C language, we have seen:

- 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 and 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



Outlook on Next Friday's Lab Session

Next tutorial: Friday, 28.02.2020, Y35-F-32

Topics:

- poss. additional introductory hints to the C language
- review of exercise 1
- preview to exercise 2
- poss. repetition on recursion
- poss. repetition of basic sorting (Bubblesort, Selectionsort, Insertionsort)
- ... (your wishes)



Questions?



Thank you for your attention.