Programming Languages

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se

Purpose of this lecture and of the ”Advanced Topics”

- The purpose of these slides is to have a basic knowledge of key concepts in programming, giving you some references or starting points.
- Some topics are discussed in depth in another set of slides called ”Advanced Topics”.
  - Some of the advanced topics are also **small clips**, max 20 min.
  - You will be recommended to **watch** or **read** about the topic before a lecture or a tutorial, so that you have a better understanding about what is going to be discussed

Outline

- Part A: Programming languages
  - Brief history and classification
  - Programming paradigms
  - Compilation and Interpretation
  - Features of C++ and Bash
- Part B: Algorithms and Programming

Part A: Programming languages

- Goals:
  - Understanding what a programming language is
  - Understanding the path from code to machine executable applications

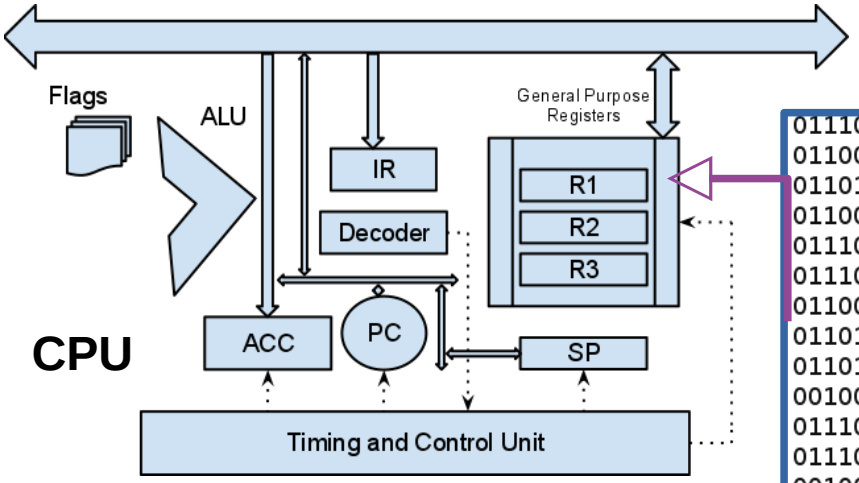
This part is important to understand the differences between the code you write and the resulting program.

Programming languages: A brief history

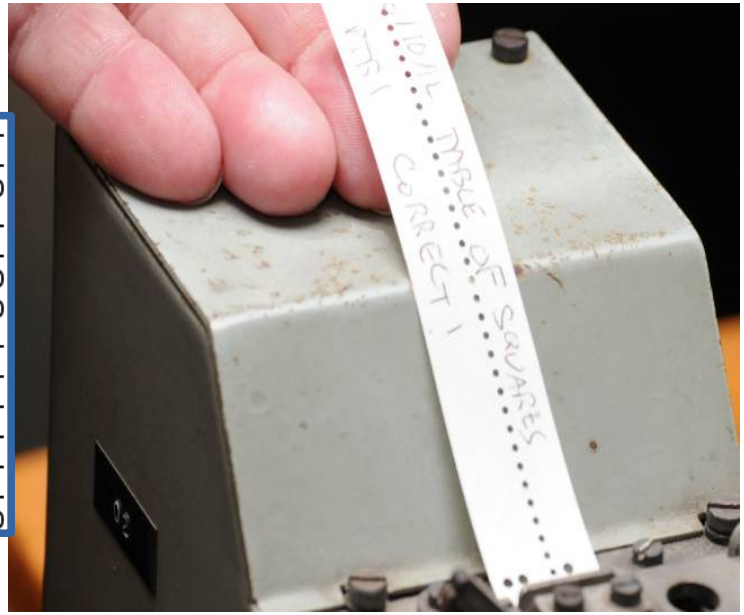
Modern classification of programming languages is based on generations. As generation increases, the languages are closer to the human way of expressing concepts.

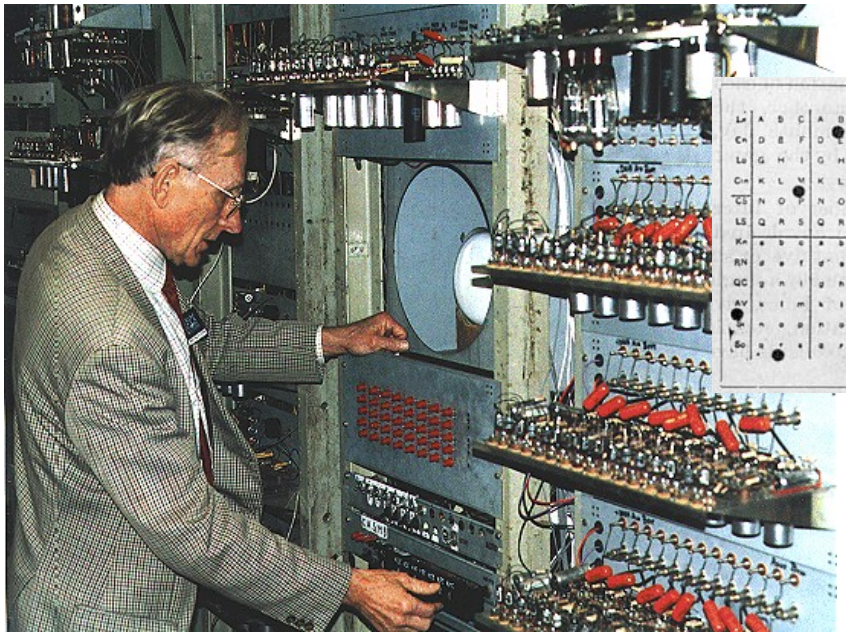
- 1st generation. **Machine code** language. This includes punchboards and **binary code**. Machine dependent.
- 2nd generation. **Assembly** or instruction-based languages. Still used in embedded programming, but through 3rd generation ones. Machine dependent. Hard to use for complex things.
- 3rd generation. Also called **High-Level** programming languages. Mostly use **English** to describe commands. **Machine independent. General Purpose: you can use them for EVERYTHING.**
  These include: C, C++, C#, Java, Javascript, Python, Bash, PHP, Pascal, Fortran...
- 4th generation. **Domain specific** languages. Report or Form generator, or Data manipulation. Examples: Mathematica, Matlab, SPSS, R (statistics). Targeted to a specific set of tasks.
- 5th generation. Mathematical or logical languages. Solving problem by specifying constraints, **without focusing on the algorithm**. Mainly used in artificial intelligence research. Examples: Prolog, NetLogo. Very narrow scope.

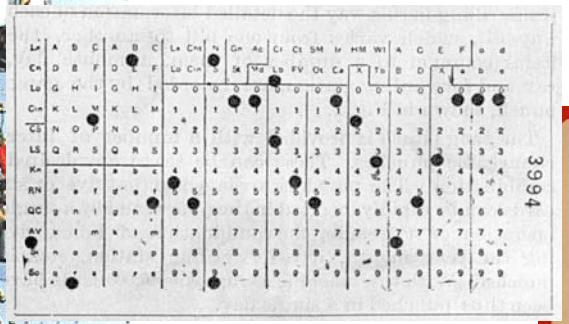
1st generation: Machine Language

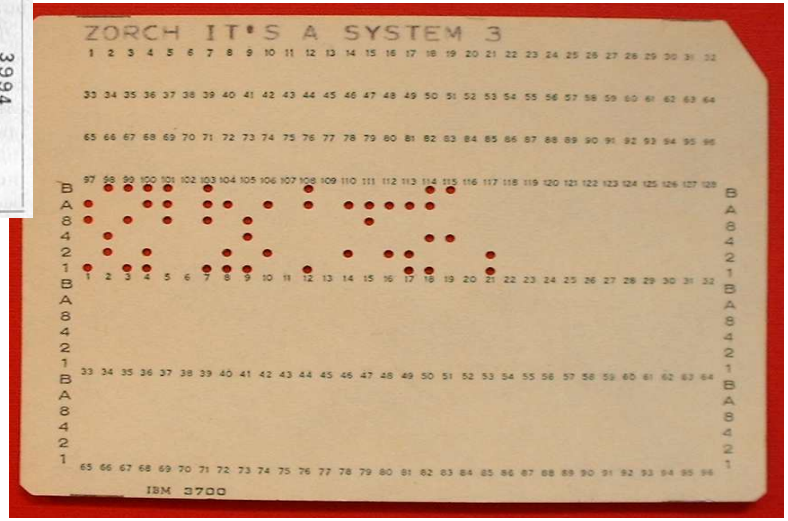



```
01110011 01100101 01110010 01
01100101 01110010 00100000 01
01101000 01100001 01110100 00
01100100 01101001 01110011 01
01110010 01101001 01100010 01
01110100 01100101 01110011 00
01100001 01101110 01111001 00
01101001 01101110 01100011 01
01101101 01101001 01101110 01
00100000 01101101 01100101 01
01110011 01100001 01100111 01
01110011 00100000 01110100 01
00100000 01100001 01101100 01
00001101 00001010 00100000 00
```









1st generation: Machine Language

- Minimal instructions set in **binary code**: binary sequences corresponding to operations like move, read, sum, multiply
- Programming done via **switches** or **punch cards** as in the pictures in the previous slide.
  - Direct edit of computer components such as CPU Registers, Memory Pointers, Start of Program Counter.
  - Direct programming, **not portable =** specific to a machine, code cannot be reused.

Why binary?

- Digital circuits are based on mapping **voltage** to **information**
- Measuring voltage can be error-prone, so one must minimize the error
- Years of engineering studies showed that the safest choice is either to have three voltage states or two
- Two proved to be safest and easiest to handle as the number of circuits on a circuit board grows: they interfere with each other! (magnetic fields etc)
- Modern computing sets the voltage difference to be $\mp$5V
- Mapping: $\mp$5V = 0, 0V = 1 (yeah, I know, misleading. But there are practical reasons for it. We don't have to care.)

Mapping things to binary

- In a computer, a sequence of bits contained in a memory chunk can be one of:
  - **Boolean expression:**
    - 1 = True , 0 = False
      but even the opposite in some cases!
    - binary *strings* of true and false:
      1001 = true false false true
  - Number or **Value**
    - Example: 11110000
    - A binary string as a value like the one above can be mapped to anything. It can be a number, a character, a sequence of characters... it all depends on how the current instruction or running program wants to interpret it
  - Operation or **Instruction**
    - There are circuits in the CPU that interpret sequences of bits into operations such as addition, transfer to memory, comparison ...
    - One instruction can be made of multiple sequences of bits.

  - The way for a computer to distinguish among those depends on where in the computing cycle the information is accessed.

Digital circuits are discrete (countable)

- **Digitization** is the process of transforming what is continuous (infinite) into something **discrete (finite)** with electronic devices.
- A dreadful consequence of having a finite set of countable memory components representing information is that **there is a finite set of numbers we can represent.**
- **Problems:**
  - What happens when the result of an operation **exceeds the finite representation** space?
  - How do one represents **negative** numbers?
  - How do we represent **fractions/irrational** numbers/**periodic** numbers/**complex** numbers?
  - How do we represent the concept of **infinity**?

Issues with limited representation overflow example

Real world arithmetics:

```
  9 +
  1 =
-----
 10
```

1 is the carry. One simply adds it on the left.

Real world

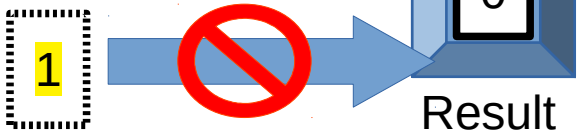
Result from the fictious computer:

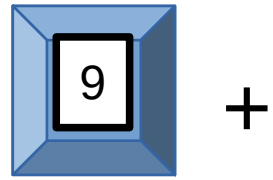
9 + 1 = 0 ???

DIGITIZATION

Imagine a fictious computer system that is made out of **boxes** and **tiles**.

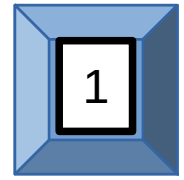
- **Tiles** are numbered from **0 to 9**.
- Each box is a **memory location** and can only contain **one tile at time.**
- The system has three boxes, two for the operands and one for the result.
  - There is no space in a box to add the carry.





Operand 1

+



Operand 2

=

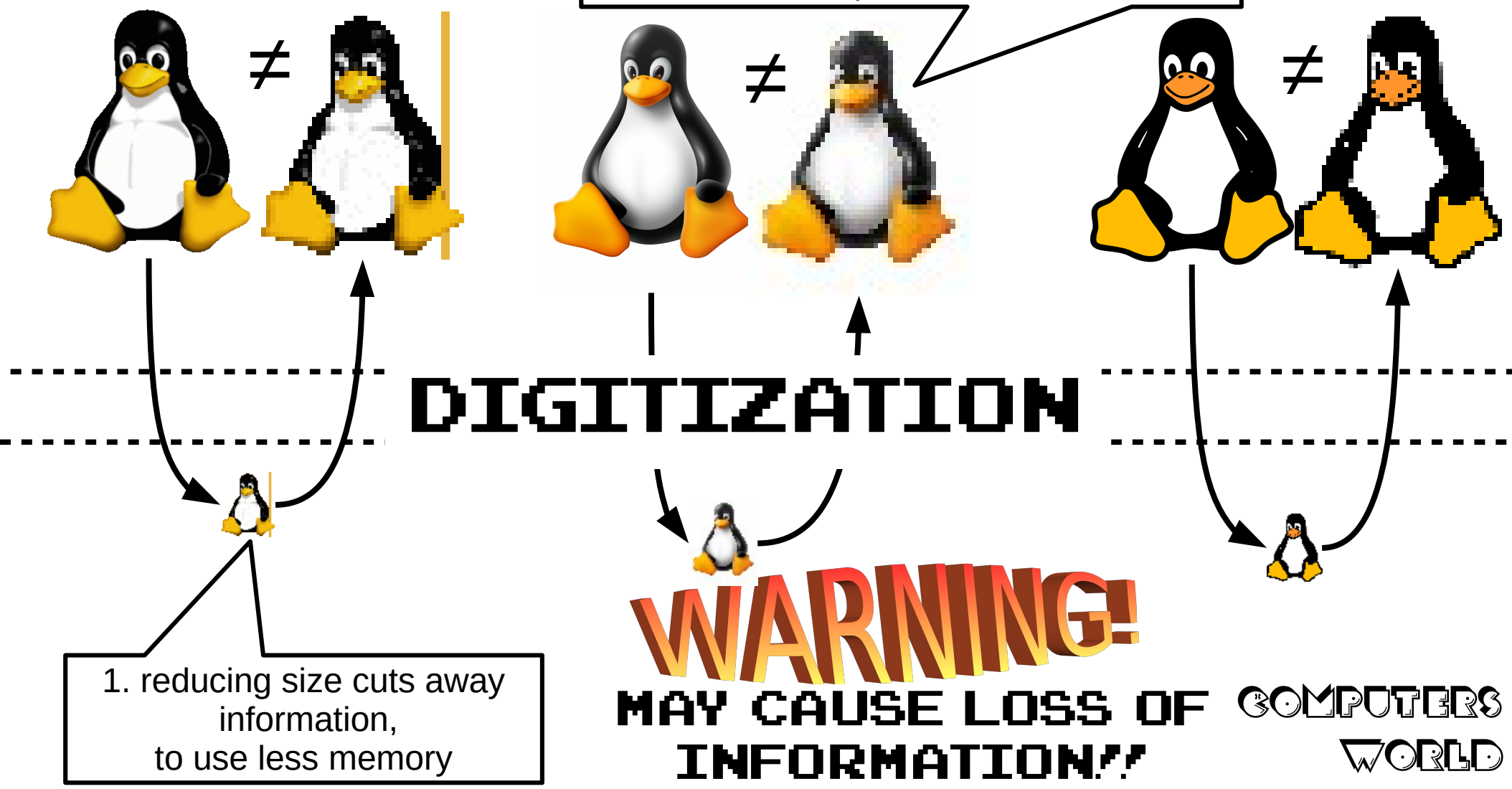
This is called **overflow**, as in the result is **outside** the range of representable numbers.

It also causes the observer to see a symbol among the representable numbers as if the numbering restarted (0 after 9).

COMPUTERS WORLD

Issues with limited representation scaling example

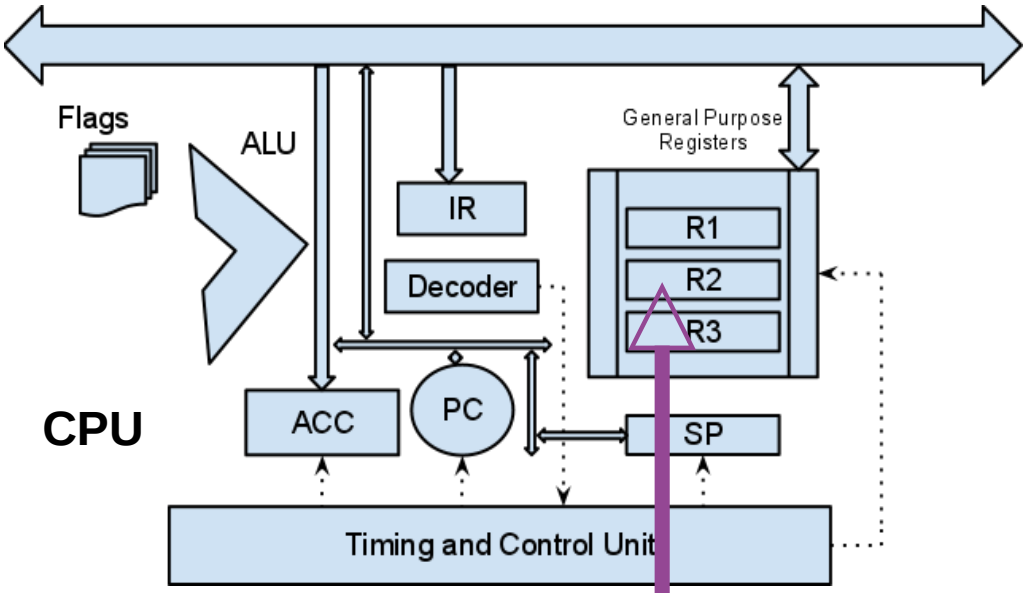
Scaling is the process with which the size of an image is reduced or enlarged



2nd generation: Assembly Code

```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32   PROC          ; procedure begins here
        CMP   AX,97   ; compare AX to 97
        JL    DONE    ; if less, jump to DONE
        CMP   AX,122  ; compare AX to 122
        JG    DONE    ; if greater, jump to DONE
        SUB   AX,32   ; subtract 32 from AX
DONE:   RET           ; return to main program
SUB32   ENDP          ; procedure ends here
```

FIGURE 17. Assembly language

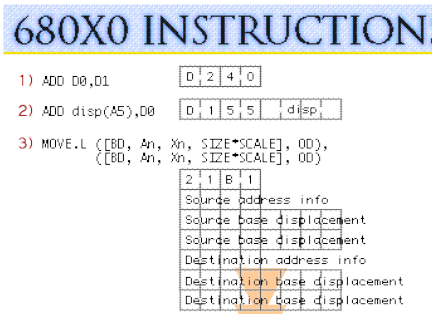


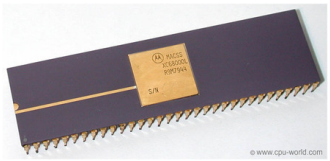
Assembler

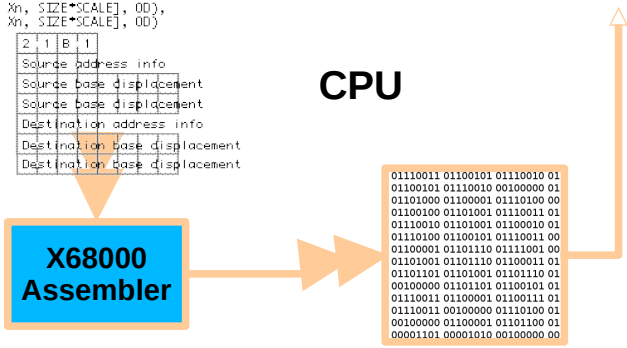
```
01110011 01100101 01110010 01
01100101 01110010 00100000 01
01101000 01100001 01110100 00
01100100 01101001 01110011 01
01110010 01101001 01100010 01
01110100 01100101 01110011 00
01100001 01101110 01111001 00
01101001 01101110 01100011 01
01101101 01101001 01101110 01
00100000 01101101 01100101 01
01110011 01100001 01100111 01
01110011 00100000 01110100 01
00100000 01100001 01101100 01
00001101 00001010 00100000 00
```

2nd generation: Assembly Code and Microcode

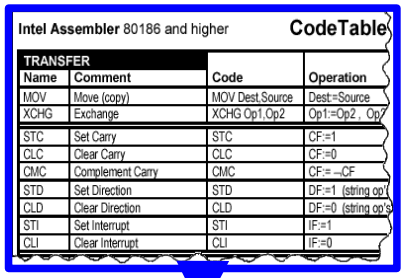
Motorola

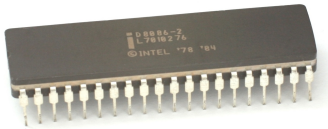


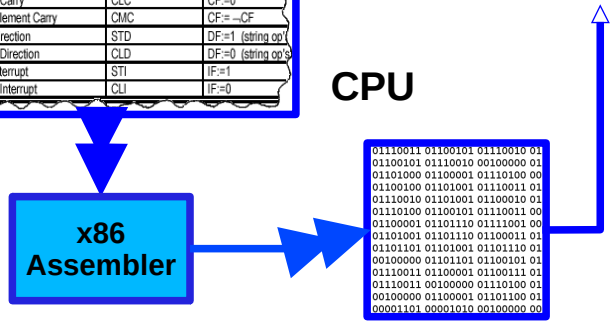




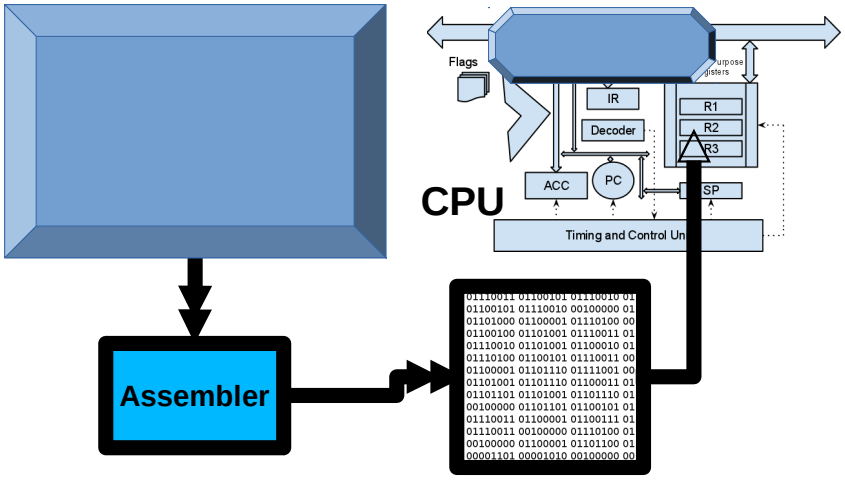
Intel







Other Architecure



Not Portable!

2nd generation: Assembly Code and Microcode

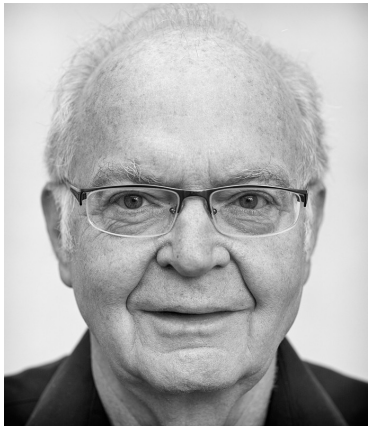
- Each instruction is represented by an **opcode** and its **arguments**.
- A more human readable language is introduced, **assembly**, that maps each opcode and arguments to a human readable syntax.
  - The program used to code is called **assembler**, takes in input a sequence of assembly statements and translates them into binary code
- New CPUs emerge that contain a more complex instruction set called **microcode**, stored physically in a ROM inside the CPU: a single instruction can do more than a single operation. Different assembly for different architectures.
  - **Not portable**: code can only be used for a specific machine.
- Used for home computers, nowadays for small devices.

Live example: https://schweigi.github.io/assembler-simulator/

3rd generation: Human-oriented

- **Algorithm oriented**: the user translates an algorithm into language commands
- Introduces programming *paradigms*:
  - **Imperative**
  - **Object Oriented**
  - Functional
  - ... more!
- Introduces various **translation to machine language** methods:
  - Compiled
  - Interpreted
  - Bytecode interpreted

Grace Hopper
1959 invents COBOL



Donald Knuth
1970 Writes
“The Art of Computer Programming”

Imperative languages

- Programming style that describes computation in terms of a **program state** and **statements** that **change** the program state.
- Adheres to the *separation of code and data* principle.
- Examples: C, FORTRAN, Python, Bash
- Example:

```
printf ("%s \n", "Hello World!");
```

Code

Data

Object-oriented languages

- A computer program is a **collection of objects** that act on each other.
- Each object is capable of **sending and receiving messages** and **processing data**. Each object is independent.
- An object is a ‘black box’ which sends and receives messages, and consists both of **code** (computer instructions) and **data** (information which these instructions operate on).

Breaks the *separation of code and data* principle.

- Examples: Java, C++, Python

Object-Oriented Languages Concepts

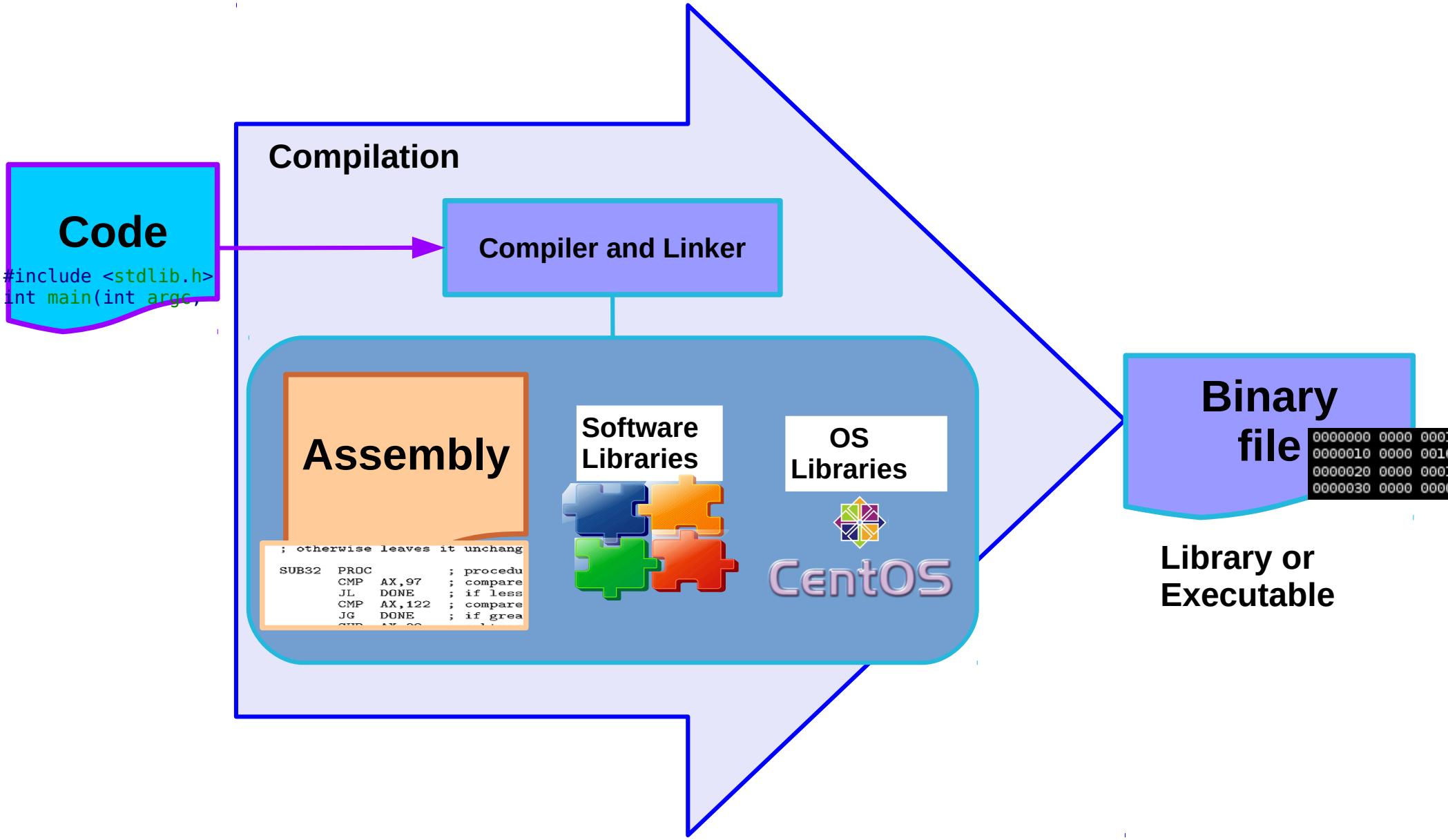
- A **Class** is a piece of code that *defines* the object's:
  - **properties** (also called **attributes**) usually data that the object can handle or carries
  - **Methods** (Usually **functions** or **procedures**) that is usually code used to *modify* the properties of the object or other objects.
- An **instance** of a class is an object, that is, something that a computer can run.
- Classes can **inherit** properties and methods from each other. If *class A is inherits from class B*, B is said to be the **parent** class of the ***child*** class A.
- Classes can **override** properties and methods that belong to their parent class by reusing the same names of properties and methods
- More practical stuff during C++ tutorials

From code to machine language

- The process with which *source code* becomes a *binary file* that can be *executed* by a computer is called **compilation**.
- The result of a compilation is also called a **build**.
- We will detail it later in the course, but we can summarize its main steps with this *algorithm*:

1. Transform the *source code* into assembly code.
2. Enrich the assembly code by *linking* it to assembly code or binary code offered by the Operating System and external libraries, to manage hardware (memory, access to devices…) or functionalities provided by other programmers
3. Produce an *executable binary file* in machine language.

From code to machine language



Compiled vs Interpreted

In this course we will see two types of languages:

- **Compiled**: developer needs to **manually run the compiler**
  - steep learning curve
  - lots of freedom on how to manage the memory contents
  - usually good for high performance and precision
  - good for serious calculations
- **Interpreted**: it **either does the compilation** for for the developer, or just **uses precompiled software** to do its job
  - easy to learn
  - limited or no memory management
  - very limited ability to tweak and customize performance and precision
  - good for automation and prototyping

To every language its purpose

- Most programming languages were invented **for a purpose**. That is actually where they **shine**
- Most programming languages *can* do things they were **not** meant for.
  Usually the result is **very sad**.
- When you choose a programming language, **make sure that it fits the task you want it to do.**
- In the next two slides there is a summary of the good and bad of the two we will use, BASH and C++.

Bash

Features:

- Interpreted
- Runs commands, executables
- Imperative paradigm
- Not explicitly typed
- No memory pointers: only environment

Preferred use:

- Scripting
- Automation of command tasks
- Combine several commands

Pros:

- Use existing commands to do tasks
- Lots of community experience
- Very low learning curve
- Very intuitive approach

Cons:

- Not portable; code depends on installed software
- Lack of types might cause unexpected results
- No memory management, only environment variables might cause scope issues: all variables are global!
- Not rich in native datastructures, that are hard to use and very rarely used in practice

C++

Features:

- Compiled
- Imperative paradigm
- Object oriented paradigm
- Types and type creation
- Templating
- Memory Pointers
- Based on standards

Preferred use:

- System development
- Embedded devices
- Low-level coding, i.e. hardware drivers
- Performance

Pros:

- Very efficient
- Empowers C with objects, allowing extending existing code
- Can directly use Assembly
- Lots of community experience
- Good debugging tools
- Good coding environments
- Control on the code preprocessor (for efficiency)

Cons:

- Requires deep knowledge of pointers and memory handling – developer has to free memory by herself
- Has high learning curve
- Not suitable for fast prototyping
- Hard to foresee runtime errors at compile time
- Control on the code preprocessor (hard to debug and understand

Outline

- Part A: Programming languages
- Part B: Algorithms and Programming
  - What is an algorithm
  - From algorithm to code
    - ingredients of programming
    - pseudocode examples
    - concepts and tools

Part B: Algorithms and Programming

- Goals:
  - Understanding the coding/programming process
  - Understanding the concepts and tools involved

This part is important to understand what we will do during the tutorials.

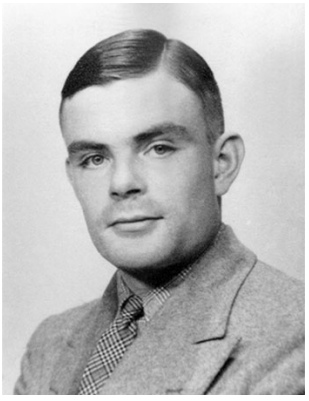
General concepts in programming

- **Programming** is the process of writing a **computer program,** that is, *translating an idea* into something that can be **executed** by a computer.
- This *translation* happens in several steps and, like a recipe for cooking a meal, one needs to understand the *ingredients* and how to *mix/cook* them.
- The *idea* usually takes the form on an **algorithm**.

Ingredients of programming: What is an **algorithm**?

- A **finite sequence of instructions** to carry out a task or solve a problem.
- An algorithm can be written in natural language or in mathematical terms.
- The term is derived from the name of the Persian scholar Al-Khwarizmi.

Ada Lovelace,
First programmer
in history



Alan Turing, Alonso Church
Hypothesis on computability
Theoretical foundations of computing

Algorithm example

1. Ask the user to input two numbers

2. Sum the two numbers

3. Print the sum on screen

Ingredients of programming: Code

- Code or *source code*
  - *source* because is the information from which a program is generated.
  - Is a **structured description** of an algorithm, it determines what a program will do
  - It is usually stored in digital format on one or more **files**
  - The description is usually done via a **programming language**
    - It is called **language** because one must respect several *grammar rules*, like in spoken or written natural human languages.
  - It can refer to other programs or program components, often called **libraries**

Ingredients of programming: Code example

Code might look weird at first. But there is a strive to make it human-readable. Consider the following example of **C** code, what do you think it does?

```
printf ("%s \n", "Hello World!");
```

Ingredients of programming: Code example

Yes, it `prints` on screen the text *string*

`Hello World!`

Let's analyze the components of the language **statement**:

```
printf ( "%s \n", "Hello World!" );
```

Issues a command:
function or procedure `printf();`

Command argument:
two function arguments

1. Formatting information:
   - "`%s \n`" means "I want you to print a string (`%s`) and then go to next line (`\n`)
2. Content information:
   "`Hello World!`" is the actual thing to print.

Grammar syntax:
<function name>**(**<argument or parameter>**);**

WARNING:
NOT A MATHEMATICAL FUNCTION!!!! : it has a domain, codomain and range, but it also has **side effects:** it changes the state of a machine.

From algorithm to code

- The **translation of an algorithm into code**, using a programming language, is called **implementation**
- The transition between an algorithm and and its implementation can have an intermediate representation that is still human readable, which mixes natural language and programming language. This is often called **pseudo-code**.
  - Writing pseudo-code is one of the best techniques to implement an algorithm, although it can be time consuming.

Pseudocode and code example

***Algorithm* in Pseudo code:**

1. Ask the user for 2 numbers:
   - print (`echo`) a message
   - use the `read` command to gather the input

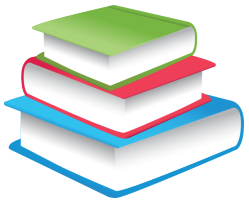
2. Sum the 2 numbers using a for loop
   - initialize the variable `sum`
   - `for` each number in input `do`:
     `sum` = `sum` + number

3. Print (`echo`) the sum on screen

***Implementation* in Bash source code:**

```
#!/bin/bash
# sum2num.sh : sum two numbers


# read input from user
echo "Enter two numbers separated by spaces, then press Enter"
# read stores data in the special variable REPLY by default
read;


# initialize sum
sum=0
# calculate sum by summing up each number separated by space
for value in $REPLY; do
        sum=$(($sum + $value))
done


# print output to screen
echo "The sum of the two numbers is $sum"
```

Ingredients of programming: Libraries



- **Libraries** are code written by other people, that can be used to write other code, so that one does not have to rewrite everything from scratch
- Typically one needs to specify in the source code what libraries are going to be used.
- Two kinds:
  - **User libraries**, like scientific libraries to calculate Fourier transform or solve equations;

  - **System libraries,** specific to the operating system, that are required to allow your program to talk to the operating system where it will run.

CentOS

Ingredients of programming: Tools

- **Text editors** and/or **IDE** (Integrated development environments) to help the developer writing code.
- **Compilers** and **Interpreters** are software that allows you to *convert* your *code* into *machine language*. They are language-specific.
  Examples:
  - `gcc, g++` for C, C++
  - `python, python3` for Python (various versions)
  - `bash` for the Bash command line interpreter
- **Build tools** that help the coder preparing all the software and libraries required for its program to compile. Examples are:
  - `make` and its configuration files `Makefiles`
  - `cmake` and its configuration tool `ccmake`
- **Package managers** to download and install user libraries independently from the operating system
  - `npm` for JavaScript
  - `conda, anaconda, pip, virtualenv` for Python

Ingredients of programming: Data

- Often provided by the user
- NOT code, but *used* by code to do things
- Carries **information**, most likely understandable by a scientist.
- **Input data**: provided in input **to** the code to process information.
  - Example: the formatting information `"%s \n"`, and the text string `”Hello World!”`
- **Output data**: the result of the code execution, that will be generated as output **from** the code execution.
  - Example: the output string `Hello World!`

Separation of Code and Data principle

- Code is information about **logic**, **arithmetics** and **algorithms**.
  - One can think of it like a mathematical function, that defines a domain and co-domain in generic terms.
- Data is information that is **to be read, processed, written**.
  - **Input** data **should be left untouched and not modified**. Think about is as a science fact or empirical/experimental data.
    - One does modify it in memory while running a program, but the changes should never be written back to the original data (would pollute science facts!)
  - **Output** Data is usually **the result** of something code did on it. For ease of use, it might be represented the same way as Input Data.

Code and data highlighted

***Algorithm* in Pseudo code:**

1. Ask the user for 2 numbers:
   - print (`echo`) a message
   - use the `read` command to gather the input

2. Sum the 2 numbers using a for loop
   - initialize the variable `sum`
   - `for` each number in input `do`:
     `sum` = `sum` + number

3. Print (`echo`) the sum result on screen

***Implementation* in Bash source code:**

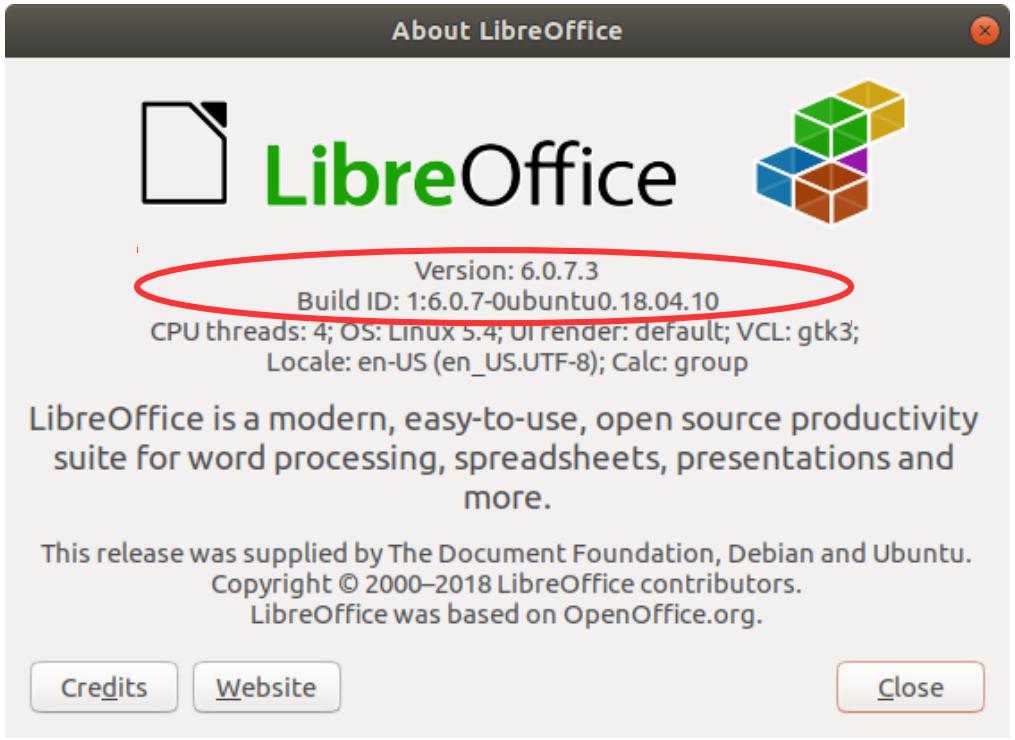
```
#!/bin/bash
# sum2num.sh : sum two numbers


# read input from user
echo "Enter two numbers separated by spaces, then press Enter"
# read stores data in the special variable REPLY by default
read;


# initialize sum
sum=0
# calculate sum by summing up each number separated by space
for value in $REPLY; do
        sum=$(($sum + $value))
done


# print output to screen
echo "The sum of the two numbers is $sum"
```

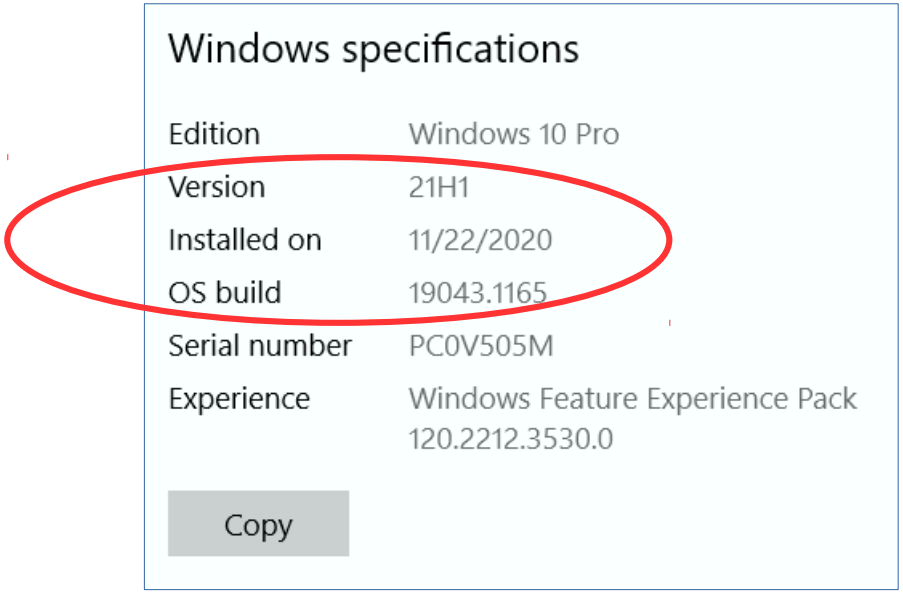
If I input the numbers 3 and 4 to obtain the result 7, then for that specific *execution* of the algorithm/program the input data is 3, 4 and the output data is 7

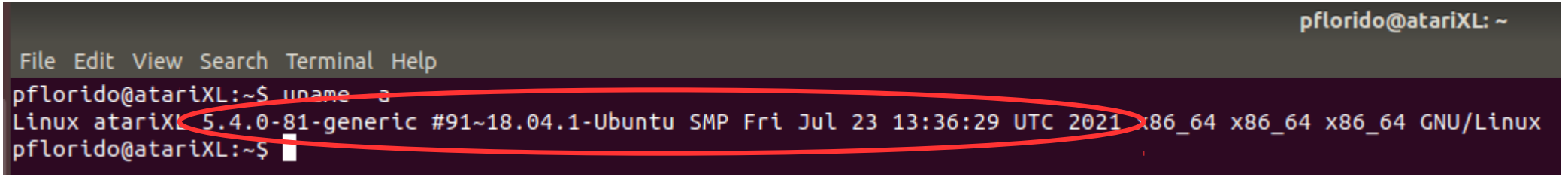
Ingredients of programming: the **build**

- Putting together source code, libraries and operating system libraries to generate a program that can *run* in a computer is called a **build**.
- A minimum build process for C++ it's the **compilation** that we have seen earlier. We will use it later in the course.
  A build process is usually automated by bash scripts or other tools. It can include other tasks happening before and after the compilation, for example:
  - Downloading required libraries
  - Checking that all required libraries are present
  - Selecting functionalities that may or may not be included in the build
  - Running some basic tests on the compiled software
- Sometimes software includes a *build number* to identify exactly when and how the software was created.
  - Check the "about" menu of software you know to discover build numbers! Sometimes they contain the date of the build.

Examples of build numbers







Ingredients of Programming Software Build ≠ Version

- A Software **Version** identifies a **fixed set of functionalities** the software will offer. There can be multiple builds for the same version of a software.
  - For example a version X of a software that runs both on Windows and Linux has at least a Windows build and a Linux build, because they use different OS libraries.
- A build is the practical step of ”putting things together”, **its requirements come from the features the software at a given Version X** should contain
  - For example, version 5 of the Zoom client supports *virtual backgrounds*, and this support must be built independently on Windows, Linux and Mac.

Ingredients of Programming Comments and Documentation

- Software becomes quickly complicated and hard to maintain. Sometimes it is not possible to read all the code. Sometimes the code is not even accessible due to legal or intellectual property reasons. Having information in human language is therefore very important. Some rules of thumb:
- Write **inline comments** in your code that explain what the code is doing.
- Adhere to **good standards** regarding **coding styles** and **text formatting**.
  - These are not just for sharing, it's also for you to remember what the code does after many years that you've developed it.
- Keep track of the "big picture" by writing **Documentation** that describes
  - overall architecture
  - common usage and coding patterns
- **ChangeLog** to keep track of important changes (Hint: See example in the last page of the course manual!)

Ingredients of Programming testing and reviewing

- **Testing** is the process of writing *additional code* that tests a functionality or a limitation of your code
  - There are different kind of tests one can run. For a list, see https://www.ibm.com/topics/software-testing
  - What if I try to sum negative numbers in the bash example? Try to ”break your own code” to identify possible faults – We will do with your project!
- **Reviewing** is the process of inspecting the code, usually by some other programmer, to get feedback about the quality of code
  - Improvements
  - Possible bottlenecks
  - Comments on whether is easy or not to understand the code. Code that is difficult to read is more likely to generate bugs and makes it hard to solve them.

Ingredients of Programming
Complexity and Optimization

- **Algorithm complexity** is a theoretical tool to help estimating
  - Time complexity: how much time it takes for a program to end
    - Knowing when a program will end is a known **undecidable** problem, called ”the halting problem” (see https://brilliant.org/wiki/halting-problem/ for a mathematical discussion)
  - Space complexity: how much memory a program will use
    - This is actually easier to estimate.
  - See https://towardsdatascience.com/algorithmic-complexity-101-28b567cc335b
- **Optimization** is the process of identifying bottlenecks or high complexity issues above and finding workarounds to make a program faster or use less memory.
  - It is usually better to optimize at a late stage of development. Trying to write optimal code from the beginning rarely helps understanding the code bottlenecks – instead it tends to create unnecessary limitations.

We don’t really have time to discuss these topics in this course, but they’re very important to save time and money.

Ingredients of Programming
Modern Collaboration and Automation

- Tools used by a community to develop software.
  - Versioning systems, e.g. git that we will see in this course
  - Continuous integration tools: automation that builds your code at every change
  - Deployment and testing suites: tools that install and test your code
  - Common errors and code proofing tools: tools that help making your code better

- More on Oxana's slides

Golden rules for a scientific programmer

(1) **Never trust the computer**, but trust your scientific intuition

- The digitization problem: a computer has limited precision

(2) Keep your **code simple** and **functionalities separate** in your code

- Write and test each functionality
- Will help you figure out what is wrong

(3) Write many (significant) **comments**

- Science is knowledge sharing: others will read your code sooner or later

(4) Don't blame the sysadmin unless you're absolutely sure it's their fault! ;-)

References

- A brief history of computing
  http://ludwig.lub.lu.se/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat01310a&AN=lovisa.003214669&lang=sv&site=eds-live&scope=site

Pictures references (not complete)

- http://www.jegerlehner.ch/intel/
- http://www.cpu-world.com/CPUs/68000/
- http://en.wikipedia.org/wiki/X86