Algorithmics and Programming

Loriane Leclercq

loriane.leclercq@ec-nantes.fr

Overview

Section 1

References

References

- Freely inspired by Vincent Toure's first year ALGPR course, mainly for the algorithm part.
- C documentation online or directly in the section 3 of the manual on Linux based systems using command: man 3 function_name
- Wikipedia for all C norms about floats, and memory representations

Section 2

Content

Content

- Lecture + tutorials : 10+4h
- Lab : 16h
- Exam : 2h

Content

- Algorithmics and pseudocode
- Programming in C

Resources

https://box.ec-nantes.fr/index.php/s/EpLMZpeY8rjcyGx

On hippocampus, course named “Algorithmics and programming”.

Section 3

Introduction

Algorithmics and programming

Why learn algorithmic when we want to know how to program:

Algorithmics and programming

Why learn algorithmic when we want to know how to program:

- A good program needs a good algorithm

Algorithmics and programming

Why learn algorithmic when we want to know how to program:

- A good program needs a good algorithm
- Algorithms are not specific to a single programming language, a single type of computer,...

Algorithmics and programming

Why learn algorithmic when we want to know how to program:

- A good program needs a good algorithm
- Algorithms are not specific to a single programming language, a single type of computer,...
- Know the (non-)feasibility of a task

Algorithmics and programming

Why learn algorithmic when we want to know how to program:

- A good program needs a good algorithm
- Algorithms are not specific to a single programming language, a single type of computer,...
- Know the (non-)feasibility of a task
- Know maximal and minimal performances to expect from a program

Algorithmics

An *algorithm* is a sequence of instruction which, given an environment, provides the steps to follow to obtain a result.

Algorithmics

An *algorithm* is a sequence of instruction which, given an environment, provides the steps to follow to obtain a result.

It divides the problem into:

Algorithmics

An *algorithm* is a sequence of instruction which, given an environment, provides the steps to follow to obtain a result.

It divides the problem into:

- basic operations

Algorithmics

An *algorithm* is a sequence of instruction which, given an environment, provides the steps to follow to obtain a result.

It divides the problem into:

- basic operations
- loops

Algorithmics

An *algorithm* is a sequence of instruction which, given an environment, provides the steps to follow to obtain a result.

It divides the problem into:

- basic operations
- loops
- adequate data structures

Abstract Machine

Each machine is different by its:

Abstract Machine

Each machine is different by its:

- processor (X64,X86,..)

Abstract Machine

Each machine is different by its:

- processor (X64,X86,..)
- basic instructions set

Abstract Machine

Each machine is different by its:

- processor (X64,X86,..)
- basic instructions set
- architecture

Abstract Machine

Each machine is different by its:

- processor (X64,X86,..)
- basic instructions set
- architecture
- memory setting

Abstract Machine

Each machine is different by its:

- processor (X64,X86,..)
- basic instructions set
- architecture
- memory setting

We need a *abstract* model to reason about all this machines:
the Turing Machine and the Abstract Machine

A first model

The *Turing Machine (TM)*:

A first model

The *Turing Machine (TM)*:

- sequential

A first model

The *Turing Machine (TM)*:

- sequential
- infinite memory

A first model

The *Turing Machine (TM)*:

- sequential
- infinite memory
- finite program

A first model

The *Turing Machine (TM)*:

- sequential
- infinite memory
- finite program
- memory access is too restrictive

A first model

The *Turing Machine (TM)*:

- sequential
- infinite memory
- finite program
- memory access is too restrictive
- instruction set is too low level (operates at the level of symbol syntactically)

A first model

The *Turing Machine (TM)*:

- sequential
- infinite memory
- finite program
- memory access is too restrictive
- instruction set is too low level (operates at the level of symbol syntactically)

Excellent for mathematical reasoning, but the memory access is too restrictive and the instruction set is too low level because it operates at the level of the symbol syntactically.

A more suitable model

The *Abstract Machine (AM)*:

A more suitable model

The *Abstract Machine (AM)*:

- sequential
- infinite memory
- finite program

A more suitable model

The *Abstract Machine (AM)*:

- sequential
- infinite memory
- finite program
- finite number of simultaneously accessible memory locations marked by names (variables)

A more suitable model

The *Abstract Machine (AM)*:

- sequential
- infinite memory
- finite program
- finite number of simultaneously accessible memory locations marked by names (variables)

Together with an *abstract programming language* that is composed of:

A more suitable model

The *Abstract Machine (AM)*:

- sequential
- infinite memory
- finite program
- finite number of simultaneously accessible memory locations marked by names (variables)

Together with an *abstract programming language* that is composed of:

- objects with semantics (integers, characters, reals, etc) represented by symbols

A more suitable model

The *Abstract Machine (AM)*:

- sequential
- infinite memory
- finite program
- finite number of simultaneously accessible memory locations marked by names (variables)

Together with an *abstract programming language* that is composed of:

- objects with semantics (integers, characters, reals, etc) represented by symbols
- classical operators (addition, division, concatenation, etc)

A more suitable model

The *Abstract Machine (AM)*:

- sequential
- infinite memory
- finite program
- finite number of simultaneously accessible memory locations marked by names (variables)

Together with an *abstract programming language* that is composed of:

- objects with semantics (integers, characters, reals, etc) represented by symbols
- classical operators (addition, division, concatenation, etc)
- classical structures (loops, conditions, functions, etc)

Formal languages

A *formal language* is

Formal languages

A *formal language* is

- described by a formal grammar (symbols, rules, axioms)

Formal languages

A *formal language* is

- described by a formal grammar (symbols, rules, axioms)
- a set of words (sequence of symbols) produced by these production rules

Formal languages

A *formal language* is

- described by a formal grammar (symbols, rules, axioms)
- a set of words (sequence of symbols) produced by these production rules

Example: Simplest integer arithmetic operations:

Formal languages

A *formal language* is

- described by a formal grammar (symbols, rules, axioms)
- a set of words (sequence of symbols) produced by these production rules

Example: Simplest integer arithmetic operations:

$$
\begin{aligned}
E &\rightarrow E + T \;\mid\; T \\
T &\rightarrow T * N \;\mid\; N \\
N &\rightarrow 0 \mid 1 \mid \cdots \mid \infty \mid (\; E \;)
\end{aligned}
$$

Formal languages

A *formal language* is

- described by a formal grammar (symbols, rules, axioms)
- a set of words (sequence of symbols) produced by these production rules

Example: Simplest integer arithmetic operations:

$$
\begin{aligned}
E &\to E + T \;\mid\; E - T \mid\; T \\
T &\to T * N \;\mid\; T/N \mid\; N \\
N &\to 0 \mid 1 \mid \cdots \mid \infty \mid (\; E \;)
\end{aligned}
$$

Programming language

We use different types of languages, depending on the different “treatments” to run the program:

Programming language

We use different types of languages, depending on the different "treatments" to run the program:

- compiled languages

Programming language

We use different types of languages, depending on the different “treatments” to run the program:

- compiled languages
- interpreted languages

Programming language

We use different types of languages, depending on the different “treatments” to run the program:

- compiled languages
- interpreted languages
- use of virtual machine

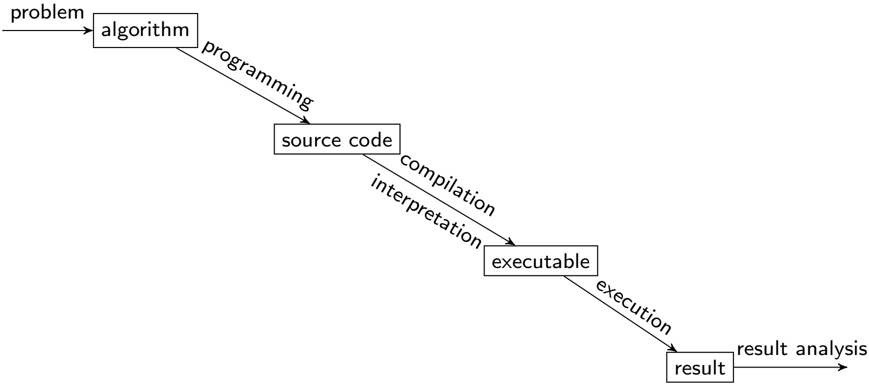
Programming language

We use different types of languages, depending on the different “treatments” to run the program:

- compiled languages
- interpreted languages
- use of virtual machine

We will explain this in details later in the part where we will talk about computer architecture.

Steps


problem
algorithm
programming
source code
compilation
interpretation
executable
execution
result
result analysis

In this course

Two aspects:

In this course

Two aspects:

- Algorithms: the theoretical part

In this course

Two aspects:

- Algorithms: the theoretical part

- programming: the practical part using C language

In this course

Two aspects:

- Algorithms: the theoretical part
  - transferable/reproducible for any programming language: a given algorithm can be *implemented* in many different programming languages
- programming: the practical part using C language

In this course

Two aspects:

- Algorithms: the theoretical part
  - transferable/reproducible for any programming language: a given algorithm can be *implemented* in many different programming languages
  - useful to evaluate the *complexity* of a program or a problem (i.e. the number of basic instructions necessary to obtain a result in the worst case scenario).
- programming: the practical part using C language

In this course

Two aspects:

- Algorithms: the theoretical part
  - transferable/reproducible for any programming language: a given algorithm can be *implemented* in many different programming languages
  - useful to evaluate the *complexity* of a program or a problem (i.e. the number of basic instructions necessary to obtain a result in the worst case scenario).
- programming: the practical part using C language
  - implementing given algorithms in C

In this course

Two aspects:

- Algorithms: the theoretical part
  - transferable/reproducible for any programming language: a given algorithm can be *implemented* in many different programming languages
  - useful to evaluate the *complexity* of a program or a problem (i.e. the number of basic instructions necessary to obtain a result in the worst case scenario).
- programming: the practical part using C language
  - implementing given algorithms in C
  - choosing the right data structure to solve a given problem

In this course

Two aspects:

- Algorithms: the theoretical part
  - transferable/reproducible for any programming language: a given algorithm can be *implemented* in many different programming languages
  - useful to evaluate the *complexity* of a program or a problem (i.e. the number of basic instructions necessary to obtain a result in the worst case scenario).
- programming: the practical part using C language
  - implementing given algorithms in C
  - choosing the right data structure to solve a given problem
  - choosing the right structure for a program, distribute functions and definitions in different meaningful files.

In this course

Two aspects:

- Algorithms: the theoretical part
  - transferable/reproducible for any programming language: a given algorithm can be *implemented* in many different programming languages
  - useful to evaluate the *complexity* of a program or a problem (i.e. the number of basic instructions necessary to obtain a result in the worst case scenario).
- programming: the practical part using C language
  - implementing given algorithms in C
  - choosing the right data structure to solve a given problem
  - choosing the right structure for a program, distribute functions and definitions in different meaningful files.

The goal is not only to teach you C but to make you able to learn a new programming language by yourself.

In this course

Two aspects:

- Algorithms: the theoretical part
  - transferable/reproducible for any programming language: a given algorithm can be *implemented* in many different programming languages
  - useful to evaluate the *complexity* of a program or a problem (i.e. the number of basic instructions necessary to obtain a result in the worst case scenario).
- programming: the practical part using C language
  - implementing given algorithms in C
  - choosing the right data structure to solve a given problem
  - choosing the right structure for a program, distribute functions and definitions in different meaningful files.

The goal is not only to teach you C but to make you able to learn a new programming language by yourself.

And Python at the end of this course ;)

Section 4

Algo

What's an algorithm and why it's useful

Let's see different definitions of an algorithm that we can find in the literature:

What's an algorithm and why it's useful

Let's see different definitions of an algorithm that we can find in the literature:

- A process used to solve a problem in a given environment.
- A procedure to obtain a result in a finite number of operations.
- From Wikipedia : "A finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation."

What's an algorithm and why it's useful

Let's see different definitions of an algorithm that we can find in the literature:

- A process used to solve a problem in a given environment.

- A procedure to obtain a result in a finite number of operations.

- From Wikipedia : "A finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation."

The environment of an algorithm is composed of:

- the physical or virtual environment
- the object, result or thing to which this algorithm corresponds
- the possible operations
- the result

What's an algorithm and why it's useful

Let's see different definitions of an algorithm that we can find in the literature:

- A process used to solve a problem in a given environment.

- A procedure to obtain a result in a finite number of operations.

- From Wikipedia : "A finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation."

The environment of an algorithm is composed of:

- the physical or virtual environment
- the object, result or thing to which this algorithm corresponds
- the possible operations
- the result

Think of it as a recipe or a group vacation organization.

Example 1: your morning routine

What you did this morning, in which order ?

- Wake up.
- Take a shower or eat breakfast or both or neither.
- Put on clothes.
- Go to the streetcar, bus or take your bike, your car, or come on foot,...

Example 2: Cake

For example the environment for baking a cake consists of:

- the environment: food, cooking utensils, oven
- the object: a recipe
- the operations: cut, mix, hit, pour, etc
- the result: a cake

Example 2: Cake

For example the environment for baking a cake consists of:

- the environment: food, cooking utensils, oven
- the object: a recipe
- the operations: cut, mix, hit, pour, etc
- the result: a cake

Since I have no way to bake a cake in this room, let's do some additions, it's almost the same !

Example 3: integer addition

$$\begin{array}{r} 655729560532 \\ +\quad 759587203523745 \\ \hline \end{array}$$

Remember when you used to add up as a kid.

Example 3: integer addition

$$\begin{array}{r} 65572956053{\color{red}2} \\ + \quad 75958720352374{\color{red}5} \\ \hline \end{array}$$

From right to left:

Example 3: integer addition

$$\begin{array}{r} 655729560532 \\ + \quad 759587203523745 \\ \hline 7 \end{array}$$

From right to left:
$2 + 5 = 7$

Example 3: integer addition

$$
\begin{array}{rr}
 & 6557295605\color{red}{3}2 \\
+ & 759587203523\color{red}{7}\color{red}{4}5 \\
\hline
 & 7
\end{array}
$$

From right to left:

Example 3: integer addition

$$\begin{array}{rr} & 6557295605\textcolor{red}{3}2 \\ + & 7595872035237\textcolor{red}{4}5 \\ \hline & \textcolor{red}{7}7 \end{array}$$

From right to left:
$3 + 4 = 7$

Example 3: integer addition

$$\begin{array}{rr} & 655729560532 \\ + & 759587203523745 \\ \hline & 77 \end{array}$$

From right to left:
$5 + 7 = 12 \geq 10$

Example 3: integer addition

$$
\begin{array}{rr}
 & \overset{\color{red}{1}}{}\\[-4pt]
 & 65572956 0{\color{red}{5}}32 \\
+ & 7595872035 23{\color{red}{7}}45 \\
\hline
 & {\color{red}{2}}77
\end{array}
$$

From right to left:
$5 + 7 = 12 \geq 10$

Example 3: integer addition

$$
\begin{array}{rr}
 & \overset{\color{red}1}{} \\
 & 65572956{\color{red}0}532 \\
+ & 759587203523{\color{red}3}745 \\
\hline
 & 277
\end{array}
$$

From right to left:
$1 + 0 + 3 = 4$

Example 3: integer addition

$$\begin{array}{rr} & \overset{1}{} \\ & 6557295605 32 \\ + & 759587203523745 \\ \hline & 4277 \end{array}$$

From right to left:
$1 + 0 + 3 = 4$

Example 3: integer addition

$$
\begin{array}{c@{\quad}ccccccccccccccc}
 & & & & 1&1& & & 1&1& & & 1& & & \\
 & & {}^{1}&{}^{1}&6&5&5&7&2&9&5&6&0&5&3&2 \\
+ & 7&5&9&5&8&7&2&0&3&5&2&3&7&4&5 \\ \hline
 & 7&6&0&2&4&2&9&3&3&0&8&4&2&7&7
\end{array}
$$

Example 3: integer addition

$$
\begin{array}{cr}
 & {}^{1\,1}\quad{}^{1\,1}\quad{}^{1}\quad \\
 & {}^{11}655729560532 \\
+ & 759587203523745 \\
\hline
 & 760242933084277
\end{array}
$$

What's the algorithm to compute $a + b$?

Example: algorithm for integer addition with regrouping (or carrying over)

Algorithm 1 Addition algorithm

procedure $\text{ADDITION}(a, b, n)$ ▷ The addition of a and b assuming they have n digits and are written $a = a_{n-1}...a_0$ and $b = b_{n-1}...b_0$

    Integer $R \leftarrow 0$

    for i from 0 to $n - 1$ **do**

        $s_i \leftarrow a_i + b_i + R$

        $R \leftarrow 0$

        if $s_i \geq 10$ **then**

            $R \leftarrow s_i/10$

            $s_i \leftarrow s_i \mod 10$

        end if

    end for

    $s_n \leftarrow R$

    return $S = s_n...s_0$

end procedure

Exercise

Write an algorithm to compute the Fibonacci numbers and the GCD (greatest common divisor).

Solution Fibonacci

Algorithm 2 Fibonacci algorithm (recursive)

```
procedure FIBONACCI(n)                    ▷ Return the n-th Fibonacci number
    if n == 0 OR n == 1 then
        return n
    end if
    return Fibonacci(n − 1) + Fibonacci(n − 2)
end procedure
```

Solution Fibonacci

Algorithm 4 Fibonacci algorithm (recursive)

```
procedure FIBONACCI(n)                    ▷ Return the n-th Fibonacci number
    if n == 0 OR n == 1 then
        return n
    end if
    return Fibonacci(n − 1) + Fibonacci(n − 2)
end procedure
```

Algorithm 5 Fibonacci algorithm (without recursion)

```
procedure FIBONACCI(n)                    ▷ Return the n-th Fibonacci number
    Integer last_u, current_u, next_u
    if n == 0 OR n == 1 then
        return n
    end if
    last_u ← 0, current_u ← 1
    for i from 2 to n do
        next_u ← current_u+last_u
        last_u ← current_u
        current_u ← next_u
    end for
    return current_u
end procedure
```

Solution GCD

To compute the GCD we use Euclidean algorithm. The idea is to do an integer division of a and b and if the remainder is non nul, repeat the process with b and r.

Algorithm 6 Euclidean algorithm

```
procedure EUCLID(a,b)          ▷ Return the GCD of a and b assuming a ≥ b > 0.
    Integer q ← a/b            ▷ "/" is a division with remainder (Euclidean division)
    Integer r ← a − b ∗ q
    if r == 0 then
        return b
    end if
    return Euclid(b, r)
end procedure
```

Algorithm efficiency

The *complexity* of an algorithm corresponds to the amount of resources required to run it. It mesures its efficiency in terms of:

Algorithm efficiency

The *complexity* of an algorithm corresponds to the amount of resources required to run it. It mesures its efficiency in terms of:

- time requirement: the number of elementary operations executed during the computation. Using usual units of time is too dependent on the choice of computer and the technological advances.

Algorithm efficiency

The *complexity* of an algorithm corresponds to the amount of resources required to run it. It mesures its efficiency in terms of:

- time requirement: the number of elementary operations executed during the computation. Using usual units of time is too dependent on the choice of computer and the technological advances.
- memory requirement: the size of computer memory needed.

Algorithm efficiency

The *complexity* of an algorithm corresponds to the amount of resources required to run it. It mesures its efficiency in terms of:

- time requirement: the number of elementary operations executed during the computation. Using usual units of time is too dependent on the choice of computer and the technological advances.
- memory requirement: the size of computer memory needed.

The complexity of a problem is the complexity of the best algorithm that solve this problem.

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

- linear: $O(n)$,

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

- linear: $O(n)$,
- quadratic: $O(n^2)$,

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

- linear: $O(n)$,
- quadratic: $O(n^2)$,
- exponential: $O(2^n)$,

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

- linear: $O(n)$,
- quadratic: $O(n^2)$,
- exponential: $O(2^n)$,
- doubly exponential: $O(2^{2^n})$,

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

- linear: $O(n)$,
- quadratic: $O(n^2)$,
- exponential: $O(2^n)$,
- doubly exponential: $O(2^{2^n})$,
- k-exponential: $O(2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}})$ with a tower of k exponential,

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

- linear: $O(n)$,
- quadratic: $O(n^2)$,
- exponential: $O(2^n)$,
- doubly exponential: $O(2^{2^n})$,
- k-exponential: $O(2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}})$ with a tower of k exponential,
- elementary recursive if it is $O(2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}})$ for some arbitrary tower of exponential and

Complexity

We say that an algorithm is *linear* in n if for any entry of size n it uses a number of elementary operations linear in n, i.e. at most $O(n) = \{an + b \mid a, b \in \mathbb{N}\}$ operations.

Same for:

- linear: $O(n)$,
- quadratic: $O(n^2)$,
- exponential: $O(2^n)$,
- doubly exponential: $O(2^{2^n})$,
- k-exponential: $O(2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}})$ with a tower of k exponential,
- elementary recursive if it is $O(2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}})$ for some arbitrary tower of exponential and
- nonelementary otherwise (see Ackermann function)

Data

There are two types of data:

Data

There are two types of data:

- variables whose value can change during the program

Data

There are two types of data:

- variables whose value can change during the program
- constants whose value is set at their creation

- Data declaration:

```
Type name
```

- Data declaration:

```
Type name
```

- variable values:

```
Integer sum
Real x,y
```

- constant values:

```
Integer NB_ANT
Real X0,Y0
```

Basic operations

- assignment: variable $\leftarrow$ value

Basic operations

- assignment: variable $\leftarrow$ value
- basic arithmetic operators: $+, -, *, /, \mod, \ldots$

Basic operations

- assignment: variable $\leftarrow$ value
- basic arithmetic operators: $+, -, *, /, \mod, \ldots$
- logical operators: AND, OR, $\top$, $\perp$

Basic operations

- assignment: variable $\leftarrow$ value
- basic arithmetic operators: $+, -, *, /, \mod, \ldots$
- logical operators: AND, OR, $\top$, $\perp$
- comparison operators: $==, <, >, <=, >=$

Bloc of instructions

Algorithm are structured in blocks by control structures:

Bloc of instructions

Algorithm are structured in blocks by control structures:

- selection statements (conditional execution):

Bloc of instructions

Algorithm are structured in blocks by control structures:

- selection statements (conditional execution):

- iterative execution (iteration statements):

Bloc of instructions

Algorithm are structured in blocks by control structures:

- selection statements (conditional execution):

- iterative execution (iteration statements):

- call of procedure:

```
your_procedure(arg1,arg2,...)
```

Bloc of instructions

Algorithm are structured in blocks by control structures:

- selection statements (conditional execution):
  - branch or jump:

```
match expression with {
  case1: sequence1
  case2: sequence2
  default: default_sequence
}
```

- iterative execution (iteration statements):

- call of procedure:

```
your_procedure(arg1,arg2,...)
```

Bloc of instructions

Algorithm are structured in blocks by control structures:

- selection statements (conditional execution):
  - branch or jump:

```
match expression with {
  case1: sequence1
  case2: sequence2
  default: default_sequence
}
```

  - conditional branch:

```
if condition then sequence1 else sequence2
```

- iterative execution (iteration statements):

- call of procedure:

```
your_procedure(arg1,arg2,...)
```

Bloc of instructions

Algorithm are structured in blocks by control structures:

- selection statements (conditional execution):
  - branch or jump:

```
match expression with {
  case1: sequence1
  case2: sequence2
  default: default_sequence
}
```

  - conditional branch:

```
if condition then sequence1 else sequence2
```

- iterative execution (iteration statements):
  - while loop:

```
while condition do sequence
```

- call of procedure:

```
your_procedure(arg1,arg2,...)
```

Bloc of instructions

Algorithm are structured in blocks by control structures:

- selection statements (conditional execution):
  - branch or jump:

    ```
    match expression with {
      case1: sequence1
      case2: sequence2
      default: default_sequence
    }
    ```

  - conditional branch:

    ```
    if condition then sequence1 else sequence2
    ```

- iterative execution (iteration statements):
  - while loop:

    ```
    while condition do sequence
    ```

  - for loop:

    ```
    for iterations do sequence
    ```

- call of procedure:

  ```
  your_procedure(arg1,arg2,...)
  ```

Choice of data structure

Store different data in a single structure to:

Choice of data structure

Store different data in a single structure to:

- structure data

Choice of data structure

Store different data in a single structure to:

- structure data
- simplify data processing

Choice of data structure

Store different data in a single structure to:

- structure data
- simplify data processing

Depending on what type of data:

Choice of data structure

Store different data in a single structure to:

- structure data
- simplify data processing

Depending on what type of data:

- any type of data: record

Choice of data structure

Store different data in a single structure to:

- structure data
- simplify data processing

Depending on what type of data:

- any type of data: record
- same type: lists or tree

Choice of data structure

Store different data in a single structure to:

- structure data
- simplify data processing

Depending on what type of data:

- any type of data: record
- same type: lists or tree

We'll see these structures in details later, directly in C.

Section 5

Architecture (and Operating system)

Operating system

An *operating system (OS)* is the system of software that controls the use of resources (computer hardware, software) for computer programs.

Operating system

An *operating system (OS)* is the system of software that controls the use of resources (computer hardware, software) for computer programs.

The OS receives requests for:

Operating system

An *operating system (OS)* is the system of software that controls the use of resources (computer hardware, software) for computer programs.

The OS receives requests for:

- memory allocation or access,

Operating system

An *operating system (OS)* is the system of software that controls the use of resources (computer hardware, software) for computer programs.

The OS receives requests for:

- memory allocation or access,
- computation resources from GPU (or any processor),

Operating system

An *operating system (OS)* is the system of software that controls the use of resources (computer hardware, software) for computer programs.

The OS receives requests for:

- memory allocation or access,
- computation resources from GPU (or any processor),
- input/output,

Operating system

An *operating system (OS)* is the system of software that controls the use of resources (computer hardware, software) for computer programs.

The OS receives requests for:

- memory allocation or access,
- computation resources from GPU (or any processor),
- input/output,
- file manipulation, ...

Operating system

An *operating system (OS)* is the system of software that controls the use of resources (computer hardware, software) for computer programs.

The OS receives requests for:

- memory allocation or access,
- computation resources from GPU (or any processor),
- input/output,
- file manipulation, ...

The OS has to schedule all these tasks in order to execute them properly.

Operating system

Examples

- Unix with as command line user interface, the scripting language bash

Operating system

Examples

- Unix with as command line user interface, the scripting language bash

- Microsoft Windows: use PowerShell (good luck with that !)

Operating system

Examples

- Unix with as command line user interface, the scripting language bash
  - BSD (FreeBSD)
- Microsoft Windows: use PowerShell (good luck with that !)

Operating system

Examples

- Unix with as command line user interface, the scripting language bash
  - BSD (FreeBSD)
  - Linux (Ubuntu, Debian, Fedora, Arch Linux, Linux Mint)
- Microsoft Windows: use PowerShell (good luck with that !)

Operating system
Examples

- Unix with as command line user interface, the scripting language bash
  - BSD (FreeBSD)
  - Linux (Ubuntu, Debian, Fedora, Arch Linux, Linux Mint)
  - MacOS
- Microsoft Windows: use PowerShell (good luck with that !)

Programming languages

Which programming languages do you know ? Which do you use ?

Programming languages

imperative	functional
Fortran, C, C++, C#, Java	Common Lisp, Scheme, OCaml, Haskell

Programming languages

imperative	functional
Fortran, C, C++, C#, Java	Common Lisp, Scheme, OCaml, Haskell

Common classification :

Programming languages

imperative	functional
Fortran, C, C++, C#, Java	Common Lisp, Scheme, OCaml, Haskell

Common classification :

- imperative vs functional languages

Programming languages

imperative	functional
Fortran, C, C++, C#, Java	Common Lisp, Scheme, OCaml, Haskell

Common classification :

- imperative vs functional languages
- scripting languages vs compiled languages vs interpreted languages

Programming languages

imperative	functional
Fortran, C, C++, C#, Java	Common Lisp, Scheme, OCaml, Haskell

Common classification :

- imperative vs functional languages
- scripting languages vs compiled languages vs interpreted languages
- object-oriented

Programming languages

imperative	functional
Fortran, C, C++, C#, Java	Common Lisp, Scheme, OCaml, Haskell

Common classification :

- imperative vs functional languages
- scripting languages vs compiled languages vs interpreted languages
- object-oriented
- high-level vs low-level

Programming languages

imperative	functional
Fortran, C, C++, C#, Java	Common Lisp, Scheme, OCaml, Haskell

Common classification :

- imperative vs functional languages
- scripting languages vs compiled languages vs interpreted languages
- object-oriented
- high-level vs low-level

Programming-language theory : design, implementation, analysis, characterization, and classification of programming languages.

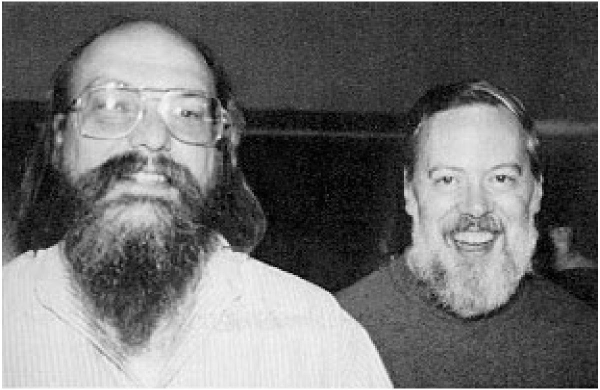
Programming languages

A classification

kind	description	pros	contra	examples
compiled	whole program translated by a *compiler* into binary code	fast	everything recompiled after every minor change	Fortran, C, C++, Objective-C, Go
interpreted	each instruction is simulated by an *interpretor*	less fast	less sensitive to minor changes	Bash, Python, Ruby, JavaScript, PHP
virtual machine (VM)	a VM simulate a generic computer	using a high-level compiled language	need a specific VM and compiler for each architecture	Java, Scala, C#

C language

Creation in 1972 by Dennis Ritchie and Ken Thompson at Bell Laboratories.
(from right to left)



Several successive norms: ANSI C, C99, C11, C17, (C23?), Embedded C
Several influenced languages: C++, C#, Java, Python, PHP,...

C standard library

The standard library should be included in every program you code, using:
#include<stdlib.h>

stdlib library contains procedures for:

C standard library

The standard library should be included in every program you code, using:
#include<stdlib.h>

stdlib library contains procedures for:

- dynamic memory management: malloc, free,...

C standard library

The standard library should be included in every program you code, using:
#include<stdlib.h>

stdlib library contains procedures for:

- dynamic memory management: malloc, free,...
- system communications: exit, abort,...

C standard library

The standard library should be included in every program you code, using:
#include<stdlib.h>

stdlib library contains procedures for:

- dynamic memory management: malloc, free,...
- system communications: exit, abort,...
- conversion from string to numeric values: atoi,...

C standard library

The standard library should be included in every program you code, using:
#include<stdlib.h>

stdlib library contains procedures for:

- dynamic memory management: malloc, free,...
- system communications: exit, abort,...
- conversion from string to numeric values: atoi,...
- random numbers: rand,...

Section 6

Basics

Examples file

The file basics_ex.c contains examples for this section, you can comment the parts that correspond to other examples to make the output more clear.

And the file helloWorld.c contains the first hello world program.
https://box.ec-nantes.fr/index.php/s/EpLMZpeY8rjcyGx

First simple program: hello world

In a file helloWorld.c write:

```
#include <stdio.h>

int main(){
  printf("Hello, World!\n");
  return 0;
}
```

First simple program: hello world

In a file helloWorld.c write:

```
#include <stdio.h>

int main(){
  printf("Hello, World!\n");
  return 0;
}
```

Compiling it with the command gcc helloWorld.c will give an executable with a random name (usually a.out).

First simple program: hello world

In a file helloWorld.c write:

```
#include <stdio.h>

int main(){
  printf("Hello, World!\n");
  return 0;
}
```

Compiling it with the command gcc helloWorld.c will give an executable with a random name (usually a.out).

To choose the name of the executable, use the option −o name and the command: gcc −o helloworld helloWorld.c

First simple program: hello world

In a file helloWorld.c write:

```
#include <stdio.h>

int main(){
  printf("Hello, World!\n");
  return 0;
}
```

Compiling it with the command gcc helloWorld.c will give an executable with a random name (usually a.out).

To choose the name of the executable, use the option –o name and the command:
gcc –o helloworld helloWorld.c

Finally you can run the program with: ./helloworld

Data

- Data declaration:

In pseudocode:

```
Type Name
```

In C:

```c
type name;
```

Data

- Data declaration:

In pseudocode:

```
Type Name
```

In C:

```c
type name;
```

- variable values:

In pseudocode:

```
Integer Sum
Real x,y
```

In C:

```c
int sum;
float x,y;
```

Data

- Data declaration:

In pseudocode:

```
Type Name
```

In C:

```
type name;
```

- variable values:

In pseudocode:

```
Integer Sum
Real x,y
```

In C:

```
int sum;
float x,y;
```

- constant values:

In pseudocode:

```
Integer NB_ANT
Real X0,Y0
```

In C:

```
const int NB_ANT;
const float X0,Y0;
```

An other way to define constants

We can also use **#define** which is a *preprocessor directive*. The syntax is:

```
#define token value
```

This is a substitution of all occurrences of “token” by “value” that will take place at the beginning of the compilation, before anything else.

Basic operators

- assignment: change value of a variable

In pseudocode:

```
VariableName←value
VariableName←variable
VariableName← expression
```

In C:

```
variable=value;
variable=variable;
variable=expression;
```

Basic operators

- assignment: change value of a variable

In pseudocode:

```
VariableName←value
VariableName←variable
VariableName← expression
```

In C:

```
variable=value;
variable=variable;
variable=expression;
```

- mathematic operators: $+, -, *, /, \mod, \ldots$

Basic operators

- assignment: change value of a variable

In pseudocode:	In C:

VariableName←value
VariableName←variable
VariableName← expression
	variable=value;
	variable=variable;
	variable=expression;

- mathematic operators: $+, -, *, /,$ mod , ...

- logical operators:

In pseudocode:	In C:
AND, OR, $\top$, $\bot$	&&, || , 1 , 0

Basic operators

- assignment: change value of a variable

In pseudocode:

```
VariableName←value
VariableName←variable
VariableName← expression
```

In C:

```
variable=value;
variable=variable;
variable=expression;
```

- mathematic operators: $+, -, *, /,\ \bmod, \ldots$

- logical operators:

In pseudocode:

AND, OR, $\top$, $\bot$

In C:

```
&&, ||, 1, 0
```

- comparison operators: $==, <, >, <=, >=$

Instructions

A function is a sequence of instructions that can be structured in *bloc of instructions*.

- branch or jump:

In pseudocode:

```
match expression with {
  case1: sequence1
  case2: sequence2
  default: default_sequence
}
```

In C:

```
switch (expression){
  case case1 : sequence1
  case case2 : sequence2
  default : default_sequence
}
```

- branch or jump:

In pseudocode:

```
match expression with {
   case1: sequence1
   case2: sequence2
   default: default_sequence
}
```

In C:

```
switch (expression){
  case case1 : sequence1
  case case2 : sequence2
  default : default_sequence
}
```

- conditional branch/expressions:

In pseudocode:

```
if condition
then sequence1
else sequence2
```

In C:

```
if (boolean_expression){
  sequence1
} else {
  sequence2
}
```

Loops

Iteration statements

- while loop:

In pseudocode:

```
while condition
do sequence
```

In C:

```
while (condition){
  sequence
}

do{
  sequence
}
while (condition);
```

Loops

Iteration statements

- while loop:

In pseudocode:

```
while condition
do sequence
```

In C:

```
while (condition){
  sequence
}

do{
  sequence
}
while (condition);
```

- for loop:

In pseudocode:

```
for iterations
do sequence
```

In C:

```
for (initialization;test;update){
  sequence;
}

initialization;
while(test){
  sequence;
  update;
}
```

Input and output

Printing and scanning numbers:

```
int printf(const char* format, ...)
int scanf(const char* format, ...)
```

Input and output

Printing and scanning numbers:

```
int printf(const char* format, ...)
int scanf(const char* format, ...)
```

Examples:

```
int integer=13;
float real=3.14159;
printf("This prints an integer %d and a real %f\n",integer,real);
```

Input and output

Printing and scanning numbers:

```
int printf(const char* format, ...)
int scanf(const char* format, ...)
```

Examples:

```
int integer=13;
float real=3.14159;
printf("This prints an integer %d and a real %f\n",integer,real);

printf("This waits for you to write an integer and then a float:"
scanf("%d%f",&integer,&real);
printf("Your numbers are %d and %f\n",integer,real);
```

Read and write in file

We use the stdio library to manage file operations, for that you add this line at the beginning of your file:

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

Read data

```
FILE *input;
input=fopen(path,''r'');
if(input==NULL){
  fprintf(stderr,"\n ERROR : Impossible to
                          read the file %s\n", path);
  exit(1);
}
int max_length_line=1024;
char *line=(char*)malloc(max_length_line);
char *buff;

/* get content line by line */
line=fgets(line,max_length_line,input);
while(!feof(input)){
  buff=strtok(line,separator); // usualy `` '' or ``;''

  /* data initial treatment here */

  line=fgets(line,max_length_line,input);
}
fclose(input);
```

Write data

```
FILE *output;
output=fopen(path,''w'');
if(output==NULL){
  fprintf(stderr,"\n ERROR : Impossible to
                                    open the file %s\n", path);
  exit(1);
}

/* write content at ones */
fprintf(output,"%s",content); // with content a string

fclose(output);
```

Write data

```
FILE *output;
output=fopen(path,''w'');
if(output==NULL){
  fprintf(stderr,"\n ERROR : Impossible to
                                   open the file %s\n", path);
  exit(1);
}

/* write content at ones */
fprintf(output,"%s",content); // with content a string

fclose(output);
```

In case we want to write content in several calls to fprintf, we can use:

```
output=fopen(path,"a");
```

Write data

```
FILE *output;
output=fopen(path,''w'');
if(output==NULL){
  fprintf(stderr,"\n ERROR : Impossible to
                                        open the file %s\n", path);
  exit(1);
}

/* write content at ones */
fprintf(output,"%s",content); // with content a string

fclose(output);
```

In case we want to write content in several calls to fprintf, we can use:

```
output=fopen(path,"a");
```

Be careful: this will not erase file content before writing.

Section 7

Types

Why types ?

- Store data efficiently

Why types ?

- Store data efficiently
- manipulate different kinds of data without ambiguity.
  For example the string "10" and the number 10 without confusion (and the number 10 in not the same in binary, in least (most) significant bit first (LSb/MSb),...)

Why types ?

- Store data efficiently
- manipulate different kinds of data without ambiguity.
  For example the string "10" and the number 10 without confusion (and the number 10 in not the same in binary, in least (most) significant bit first (LSb/MSb),...)
- be sure that a function is well-formed: the type returned correspond to the type declared.

Data declaration

type name;

Declaration

Take the pointer of an existing and initialized variable:

```
int n = 72;
int* p = &n;
```

Declare a pointer:

```
int* n; // for an integer
char* c; // for a character
int** p; // a pointer of a pointer of integer
type* p; // a pointer of something of type "type"
```

Built-in types

In C

keyword	type	size (bits)	format specifier
char	character	8	%c
short	integer	16	%hd, %hi
int	integer	16/32/64 depending on the architecture (N for N-bit processor)	%i, %d, %o, %x or %X
float	real floating-point single-precision	usually 32	%f
double	real floating-point double-precision	usually 64	%lf
since C99 _Bool	boolean	same as int	%i or %d

and modifiers *signed*, *unsigned*, *short* and *long*

There is fixed-width integer types int8_t, int16_t, int32_t,...

All numbers are encoded:

All numbers are encoded:

- in binary

All numbers are encoded:

- in binary
- using a bit for the sign (if not unsigned)

All numbers are encoded:

- in binary
- using a bit for the sign (if not unsigned)
- are limited due to integer overflow or rounding problems

Strings as arrays

There is no built-in type for strings in C.

There are two ways to declare a string as an array of characters:

Use **#include** <string.h> to have access to functions strcpy ,strcmp ,...

Strings as arrays

There is no built-in type for strings in C.

There are two ways to declare a string as an array of characters:

- ```
  char * str;
  ```

Use **#include** <string.h> to have access to functions strcpy ,strcmp ,...
  ```

Strings as arrays

There is no built-in type for strings in C.

There are two ways to declare a string as an array of characters:

- ```
  char * str;
  ```
- ```
  char str[];
  ```
  You can define a fixed length string for this kind of declaration using:
  ```
  char str[size];
  ```

Use **#include** <string.h> to have access to functions strcpy ,strcmp ,...

Record

A *record* allows you to group data of different types in a single structure:

```
struct name{
  type1 member1;
  type2 member2;
  ...
};
```

Declare a structure variable or a pointer to a structure:

```
struct name n1,*n2;
struct name{
  ...
} n1,*n2;
```

Access a parameter of a structure:

```
n1.member1;
```

Access a parameter of a pointer to a structure:

```
n2->member1;
(*n2).member1
```

Union

A *union* allows you to store **only one** member of a set of possibly different types in a single structure:

```
union name{
  type1 member1;
  type2 member2;
  ...
};
```

Declare a union variable or a pointer to a union:

```
struct name n1,*n2;
struct name{
  ...
} n1,*n2;
```

Access a parameter of a union:

```
n1.member1;
```

Access a parameter of a pointer to a union:

```
n2->member1;
(*n2).member1;
```

Array

```
type name[size];
```

Different ways to declare and initialize an array:

```
int array1[4]={3,8,5,0};
int array2[]={3,8,5,0};
int array3[4];
array3[0]=3;
array3[1]=8;
array3[2]=5;
array3[3]=0;
```

Multidimensional arrays:

```
type name[size1][size2][size3]...;
```

Arrays as pointers:
&x[i] is the same as x+i and
x[i] is the same as *(x+i)

New types

In general, you declare a new type using:

```
typedef type name;
```

But if your type is a structure, you will need to use an alias to call your new type, it can be the same as the name of your type.

```
typedef struct name{
  type1 member1;
  type2 member2;
} alias;
```

Section 8

Data structures using pointers

Call by reference

Since a function cannot return more than one value, the idea is to use pointers to values as arguments to be able to “modify” some values.

Call by reference

Since a function cannot return more than one value, the idea is to use pointers to values as arguments to be able to "modify" some values.

To modify a value name of type type, we use a pointer of this type:

```
void function(type * name, other_args){
  ...
  *name=new_value;
  ...
}
```

Call by reference

Since a function cannot return more than one value, the idea is to use pointers to values as arguments to be able to “modify” some values.

To modify a value name of type type, we use a pointer of this type:

```
void function(type * name, other_args){
   ...
   *name=new_value;
   ...
}
```

For example, to switch 3 integers, we can use the function:

```
void switch_numbers(int *i1, int *i2, int *i3){
  int j=*i1;
  *i1=*i2;
  *i2=*i3;
  *i3=j;
}
```

This example is in the file call_ref_ex.c

Array

Little game: try to print `arr[i]` and `i[arr]` for an array `arr` and a position i.

All examples about arrays and multidimensional arrays are in the file `array_ex.c`.

Multidimensional array

How to get the size of a multidimensional array type array [size1][size2][size3] ?

If array is initialized, we can compute the first dimension size1 using:

```
sizeof(array)/sizeof(array[0]);
```

the second dimension size2 using:

```
sizeof(array[0])/sizeof(array[0][0]);
```

and the third dimension size3 using:

```
sizeof(array[0][0])/sizeof(type);
```

Singly linked list

A (singly linked) *list* is a sequence of data structures connected with links.

A cell of a list is a structure:

```
struct element{
  type data;
  struct element *next;
};
```

We can define a type with this structure:

```
typedef struct element{
  type data;
  struct element *next;
} element;
```

An example of list of persons with a function to print the list and to add and remove a data are in the file list_ex .c.

Doubly linked list

A *doubly linked list* is a sequence of data structures connected with links in both directions for previous and next cells.

A cell of a list is a structure:

```
struct element{
  type data;
  struct element *next, *prev;
};
```

We can define a type with this structure:

```
typedef struct element{
  type data;
  struct element *next, *prev;
} element;
```

An example of doubly linked list of persons with a function to print the list and to add and remove a data are in the file doubly_list_ex .c.

Binary trees

Binary trees are a really useful way to store ordered data and search for them efficiently in the tree.

The basic node of a tree is the structure:

```
typedef struct node{
  type data;
  struct node *left, *right;
}node;
```

An example of binary tree of persons is in the file `tree_ex.c` with a function to add a new node.

For a type of data, we will need a comparison function. For example for persons:

```
typedef struct person{
   ...
}person;


/* return 1 if p1 is greater than p2 in the chosen order,
   0 otherwise */
int bigger_person(person p1,person p2){
  return ...
}
```

Binary trees

Add a new node

```c
int addNode(node** root, person p){
  node *new_node=(node*)malloc(sizeof(node));

  node *new_root=*root;
  node *tmp_node;

  new_node->data=p;
  new_node->left=NULL;
  new_node->right=NULL;

  if(*root==NULL) *root=new_node;
  else{
    while(new_root!=NULL){
      tmp_node=new_root;
      if(bigger_pers(p,tmp_node->data)){
        new_root=new_root->right;
        if(new_root==NULL) tmp_node->right=new_node;
      }else{
        new_root=new_root->left;
        if(new_root==NULL) tmp_node->left=new_node;
      }
    }
  }return 0;
}
```

```
node* recAddNode(node* root, person p){
  if(root==NULL){
    node *new_node=(node*)malloc(sizeof(node));
    new_node->data=p;
    new_node->left=NULL;
    new_node->right=NULL;
    return new_node;
  }
  if(bigger_pers(p,root->data))
    root->right=recAddNode(root->right,p);
  else
    root->left=recAddNode(root->left,p);
  return root;
}
```

Section 9

Bitwise operations, floating point

Floating-point arithmetic

- trade-off between range and precision.

Floating-point arithmetic

- trade-off between range and precision.
- use arithmetic representation of real numbers:
  $(sign) significant \times base^{exposant}$

Floating-point arithmetic

- trade-off between range and precision.
- use arithmetic representation of real numbers: $(sign)significant \times base^{exposant}$
- The length of the significand determine the precision. Using precision p, the significand s encoded in base b has the value: $\frac{s}{b^{p-1}} \times b^p$

Floating-point arithmetic

- trade-off between range and precision.
- use arithmetic representation of real numbers:
  $(sign) significant \times base^{exposant}$
- The length of the significand determine the precision. Using precision p, the significand s encoded in base b has the value: $\frac{s}{b^{p-1}} \times b^p$
- examples in standard-form scientific notation (in base 10):
  $7947234 = 7.947234 \times 10^6$
  $0.000598346 = 5.98346 \times 10^{-4}$

Floating-point arithmetic

- trade-off between range and precision.
- use arithmetic representation of real numbers:
  $(sign) significant \times base^{exposant}$
- The length of the significand determine the precision. Using precision p, the significand s encoded in base b has the value: $\frac{s}{b^{p-1}} \times b^p$
- examples in standard-form scientific notation (in base 10):
  $7947234 = 7.947234 \times 10^6$
  $0.000598346 = 5.98346 \times 10^{-4}$
- How to add these two numbers ?

Floating-point arithmetic

- trade-off between range and precision.
- use arithmetic representation of real numbers:
  $(sign) significant \times base^{exposant}$
- The length of the significand determine the precision. Using precision p, the significand s encoded in base b has the value: $\frac{s}{b^{p-1}} \times b^p$
- examples in standard-form scientific notation (in base 10):
  $7947234 = 7.947234 \times 10^6$
  $0.000598346 = 5.98346 \times 10^{-4}$
- How to add these two numbers ?
  Usually, we rewrite them using the same power of 10

Floating-point in C

In C, floats and double follows the IEEE 754 standard.

Three versions IEEE 754-1985, then IEEE 754-2008 and the new IEEE 754-2019 (under the name IEC 60559:2020). Next projected revision in 2028.

Float and double

- floats follows the single-precision floating point format

Float and double

- floats follows the single-precision floating point format
- doubles follows the double-precision floating-point format

Float and double

- floats follows the single-precision floating point format
- doubles follows the double-precision floating-point format
- long doubles follows either quadruple-precision floating-point format, or non-IEEE "double-double" or the same as double.

IEEE 754 single precision standard binary32

Over 32 bits:

IEEE 754 single precision standard binary32

Over 32 bits:

- sign bit n: 1 bit, the most significant (on the left)

IEEE 754 single precision standard binary32

Over 32 bits:

- sign bit n: 1 bit, the most significant (on the left)
- exponent width e: 8-bit unsigned integer from 0 to 255, the zero is placed at 127

IEEE 754 single precision standard binary32

Over 32 bits:

- sign bit n: 1 bit, the most significant (on the left)
- exponent width e: 8-bit unsigned integer from 0 to 255, the zero is placed at 127
- significand precision s: 23 remaining bits

IEEE 754 single precision standard binary32

Over 32 bits:

- sign bit n: 1 bit, the most significant (on the left)
- exponent width e: 8-bit unsigned integer from 0 to 255, the zero is placed at 127
- significand precision s: 23 remaining bits

When the exponent is valid the formula for a float value is:

$$(-1)^n \times 2^{e-127} \times (1.s)$$

IEEE 754 single precision standard binary32

Over 32 bits:

- sign bit n: 1 bit, the most significant (on the left)
- exponent width e: 8-bit unsigned integer from 0 to 255, the zero is placed at 127
- significand precision s: 23 remaining bits

When the exponent is valid the formula for a float value is:

$$(-1)^n \times 2^{e-127} \times (1.s)$$

the all-zero and all-one exponents are used for special values.

The all-zero and all-one exponents

Special exponents values:

The all-zero and all-one exponents

Special exponents values:

- 11111111 is used either for NaN (Not a number) to display an abnormality when the significand is non-zero (quiet or signalling NaN depending on the first bit of the significand) or for infinity if the significand is all-zero ($\pm\infty$ depending on the sign bit).

The all-zero and all-one exponents

Special exponents values:

- 11111111 is used either for NaN (Not a number) to display an abnormality when the significand is non-zero (quiet or signalling NaN depending on the first bit of the significand) or for infinity if the significand is all-zero ($\pm\infty$ depending on the sign bit).
- 00000000 is ether ± 0 if the significand is 0 or subnormal numbers otherwise. The subnormal numbers follow the formula :

$$(-1)^n \times 2^{-126} \times (0.s)$$

Float examples

Type	Sign	Actual expnt	Biased expnt	Exponent field	Fraction field	value
Zero	0	-126	0	0000 0000	000 0000 0000 0000 0000 0000	0.0
Negative Zero	1	-126	0	0000 0000	000 0000 0000 0000 0000 0000	-0.0
(Minus) One	*	0	127	0111 1111	000 0000 0000 0000 0000 0000	± 1.0
Smallest denormalized(d)	*	-126	0	0000 0000	000 0000 0000 0000 0000 0001	$\pm 1,4 \times 10^{-45}$
Middle d	*	-126	0	0000 0000	100 0000 0000 0000 0000 0000	$\pm 5.88 \times 10^{-39}$
Largest d	*	-126	0	0000 0000	111 1111 1111 1111 1111 1111	$\pm 1.18 \times 10^{-38}$
Smallest normalized (n)	*	-126	1	0000 0001	000 0000 0000 0000 0000 0000	$\pm 1.18 \times 10^{-38}$
Largest n	*	127	254	1111 1110	111 1111 1111 1111 1111 1111	$\pm 3.4 \times 10^{38}$
Infinity	*	128	255	1111 1111	000 0000 0000 0000 0000 0000	$\pm\infty$
Not a number	*	128	255	1111 1111	non-zero	NaN

bitwise operations in C

There are two types of bitwise operations in C:

- logical: &, |, ^, $\sim$
- shifts: $<<$, $>>$

bitwise operations in C

There are two types of bitwise operations in C:

- logical: &, |, ^, ~
- shifts: <<, >>

Also available in C++ and other C-family languages

bitwise operators: logical operations

- bitwise AND: &
- bitwise NOT: ~
- bitwise inclusive OR: |
- bitwise XOR: ^

bitwise operators: logical operations

- bitwise AND: &

bit a	bit b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

- bitwise inclusive OR: |

- bitwise NOT: $\sim$

- bitwise XOR: ^

bitwise operators: logical operations

- bitwise AND: &

bit a	bit b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

- bitwise inclusive OR: |

bit a	bit b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

- bitwise NOT: ~

- bitwise XOR: ^

bitwise operators: logical operations

- bitwise AND: &

bit a	bit b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

- bitwise inclusive OR: |

bit a	bit b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

- bitwise NOT: $\sim$

  unary one's complement

bit	$\sim$ a
0	1
1	0

- bitwise XOR: ^

bitwise operators: logical operations

- bitwise AND: &

bit a	bit b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

- bitwise inclusive OR: |

bit a	bit b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

- bitwise NOT: ~

  unary one's complement

bit	~ a
0	1
1	0

- bitwise XOR: ^

bit a	bit b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

logical equivalent for bitwise operators

Bitwise	**Logical**
a & b	a && b
a | b	a || b
a ˆ b	a != b
∼ a	!a

Bitwise operators work on bits and perform bit by bit operations. Whereas Logical operators are used to make a decision based on multiple conditions.

bit shifts

Example: $(10663)_{10} = (0010100110100111)_2$ in 16 bits (2 Bytes)

bit shifts

Example: $(10663)_{10} = (0010100110100111)_2$ in 16 bits (2 Bytes)

- right shift: $>>$

$10663 >> 0$	0010100110100111	10663
$10663 >> 1$	0001010011010011	$10663/2$
$10663 >> 2$	0000101001101001	$10663/4$
$10663 >> 3$	0000010100110100	$10663/8$
$10663 >> n$	...	$10663/2^n$
$10663 >> -1$	0101001101001110	$10663 * 2 = 10663 << 1$

bit shifts

Example: $(10663)_{10} = (0010100110100111)_2$ in 16 bits (2 Bytes)

- right shift: $>>$

$10663 >> 0$	0010100110100111	10663
$10663 >> 1$	0001010011010011	$10663/2$
$10663 >> 2$	0000101001101001	$10663/4$
$10663 >> 3$	0000010100110100	$10663/8$
$10663 >> n$	...	$10663/2^n$
$10663 >> -1$	0101001101001110	$10663 * 2 = 10663 << 1$

- left shift: $<<$

$10663 << 0$	0010100110100111	10663
$10663 << 1$	0101001101001110	$10663 * 2$
$10663 << 2$	1010011010010000	$10663 * 4$
$10663 << 3$	0100110100000000	$10663 * 8$
$10663 << n$	...	$10663 * 2^n$
$10663 << -1$	0001010011010011	$10663/2 = 10663 >> 1$

bit shifts

Example: $(10663)_{10} = (0010100110100111)_2$ in 16 bits (2 Bytes)

- right shift: $>>$

$10663 >> 0$	0010100110100111	10663
$10663 >> 1$	0001010011010011	$10663/2$
$10663 >> 2$	0000101001101001	$10663/4$
$10663 >> 3$	0000010100110100	$10663/8$
$10663 >> n$	...	$10663/2^n$
$10663 >> -1$	0101001101001110	$10663 * 2 = 10663 << 1$

- left shift: $<<$

$10663 << 0$	0010100110100111	10663
$10663 << 1$	0101001101001110	$10663 * 2$
$10663 << 2$	1010011010010000	$10663 * 4$
$10663 << 3$	0100110100000000	$10663 * 8$
$10663 << n$	...	$10663 * 2^n$
$10663 << -1$	0001010011010011	$10663/2 = 10663 >> 1$

Remark: $10663 << -n = 10663 >> n$ because $10663 * 2^{-n} = 10663/2^n$

Section 10

Compilation, Makefile, how to get a working program

compiler

What's a compiler ?

compiler

What's a compiler ?

A computer software that translates (compiles) instructions of the source code written in a high-level language into a set of low-level machine-language instructions that can be understood by a digital computer's CPU.

compiler

What's a compiler ?

A computer software that translates (compiles) instructions of the source code written in a high-level language into a set of low-level machine-language instructions that can be understood by a digital computer's CPU.

Basically compilers are programs translating programs so that they can be executed...but they must also be translated beforehand.

GNU Compiler Collection (gcc)

The gcc compiler is:

GNU Compiler Collection (gcc)

The gcc compiler is:

- a free software

GNU Compiler Collection (gcc)

The gcc compiler is:

- a free software
- an optimizing compiler

GNU Compiler Collection (gcc)

The gcc compiler is:

- a free software
- an optimizing compiler
- supporting various programming languages: C, C++, Objective-C, Go, Ada, D, Fortran,...

About gcc

- first released in 1987

About gcc

- first released in 1987
- roughly 15 million lines of code in 2019

About gcc

- first released in 1987
- roughly 15 million lines of code in 2019
- one of the biggest free program in existence

About gcc

- first released in 1987
- roughly 15 million lines of code in 2019
- one of the biggest free program in existence
- important role in the growth of free software as a tool and an example

About gcc

- first released in 1987
- roughly 15 million lines of code in 2019
- one of the biggest free program in existence
- important role in the growth of free software as a tool and an example
- in 1990, it supported 30 computer architectures, was outperforming several vendor compilers, and was used commercially by several companies

gcc command syntax

The order of arguments in the command is:
gcc [options] [source_files] [object_files] [−o output_file]

with:

gcc command syntax

The order of arguments in the command is:
gcc [options] [source_files] [object_files] [−o output_file]

with:

- options

  the different options, for example: −lm to add the maths.h library, −I*path* to specify a path to search for the included files, −Wall for all warnings mode, −O*level* for the level of optimization of the code size and execution time...

gcc command syntax

The order of arguments in the command is:
gcc [options] [source_files] [object_files] [−o output_file]

with:

- options
  the different options, for example: −lm to add the maths.h library, −I*path* to specify a path to search for the included files, −Wall for all warnings mode, −O*level* for the level of optimization of the code size and execution time...

- source_files
  the .c files

gcc command syntax

The order of arguments in the command is:
gcc [options] [source_files] [object_files] [−o output_file]

with:

- options
  the different options, for example: −lm to add the maths.h library, −I*path* to specify a path to search for the included files, −Wall for all warnings mode, −O*level* for the level of optimization of the code size and execution time...

- source_files
  the .c files

- object_files
  the .o files

gcc command syntax

The order of arguments in the command is:
gcc [options] [source_files] [object_files] [−o output_file]

with:

- options
  the different options, for example: −lm to add the maths.h library, −I*path* to specify a path to search for the included files, −Wall for all warnings mode, −O*level* for the level of optimization of the code size and execution time...

- source_files
  the .c files

- object_files
  the .o files

- −o output_file
  the name of the output produced by the command

What's a makefile

The make command automatically determine which files of a program needs to be recompiled and in which order, using rules given in the Makefile

Example working makefile

Files

We have three files hello_main.c and hello_fct.c with header file is hello_fct.h, we will construct different makefiles to automatically recompile them.

```c
// hello_main.c

#include<stdio.h>
#include "hello_fct.h"

int main(){

  printHello();

  return 0;

}
```

```c
// hello_fct.c

#include<stdio.h>
#include "hello_fct.h"

void printHello(){
  printf("Hello world !\n");
}
```

```c
// hello_fct.h

#ifndef HELLO_FCT_H
#define HELLO_FCT_H

void printHello();

#endif
```

Example working makefile

First Makefile

The command to compute these files is:

```
gcc -c -o hello_main.o hello_main.c -I.
gcc -c -o hello_fct.o hello_fct.c -I.
gcc -o hello hello_main.o hello_fct.o -I.
```

Example working makefile

First Makefile

The command to compute these files is:

```
gcc -c -o hello_main.o hello_main.c -I.
gcc -c -o hello_fct.o hello_fct.c -I.
gcc -o hello hello_main.o hello_fct.o -I.
```

- 1st Makefile:

```
hello: hello_main.c hello_fct.c
        gcc -o hello_out hello_main.c hello_fct.c -I.
```

The option -I. is for gcc to look for the included files in the current folder (.)

The command to compute these files is:

```
gcc -c -o hello_main.o hello_main.c -I.
gcc -c -o hello_fct.o hello_fct.c -I.
gcc -o hello hello_main.o hello_fct.o -I.
```

- 1st Makefile:

```
hello: hello_main.c hello_fct.c
	gcc -o hello_out hello_main.c hello_fct.c -I.
```

  The option -I. is for gcc to look for the included files in the current folder (.)

- 2nd Makefile:

```
CC=gcc
CFLAGS=-I.

hello: hello_main.c hello_fct.c
	$(CC) -o hello_out hello_main.c hello_fct.c
```

More complicated Makefiles

- 3rd Makefile:

```
CC=gcc
CFLAGS=-I.
DEPS=hello_fct.h

%.o: %.c (DEPS)\(CC) -c -o $@ $< $(CFLAGS)

 hello: hello_main.o hello_fct.o
        $(CC) -o hello_out hello_main.o hello_fct.o
```

More complicated Makefiles

- 3rd Makefile:

```
CC=gcc
CFLAGS=-I.
DEPS=hello_fct.h

%.o: %.c (DEPS)\(CC) -c -o $@ $< $(CFLAGS)

hello: hello_main.o hello_fct.o
        $(CC) -o hello_out hello_main.o hello_fct.o
```

- 4th Makefile

```
CC=gcc
CFLAGS=-I.
DEPS=hello_fct.h
OBJ=hello_main.o hello_fct.o

%.o: %.c (DEPS)\(CC) -c -o $@ $< $(CFLAGS)

hello: $(OBJ)
        $(CC) -o $@ $^ $(CFLAGS)

make clean:
        rm *.o
```

Complete Makefile

```makefile
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)

ODIR=obj
LDIR =../lib

LIBS=-lm

_DEPS = hello_fct.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = hello_main.o hello_fct.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
	$(CC) -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:
	rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Optimization flag

There are different options to optimize compiled C code using gcc:

Optimization flag

There are different options to optimize compiled C code using gcc:

- $-$O*level* for different level of optimization (the higher, the less execution time but the most compilation time and also memory usage)

Optimization flag

There are different options to optimize compiled C code using gcc:

- –O*level* for different level of optimization (the higher, the less execution time but the most compilation time and also memory usage)
- –Os optimization for code size

Optimization flag

There are different options to optimize compiled C code using gcc:

- −O*level* for different level of optimization (the higher, the less execution time but the most compilation time and also memory usage)
- −Os optimization for code size
- −Ofast O3 that optimizes for speed disregarding exact standard's compliance (for example for math calculations accuracy)

Optimization flag

There are different options to optimize compiled C code using gcc:

- −O*level* for different level of optimization (the higher, the less execution time but the most compilation time and also memory usage)

- −Os optimization for code size

- −Ofast O3 that optimizes for speed disregarding exact standard's compliance (for example for math calculations accuracy)

All details are in the doc for gcc using: gcc −Q [−O<number>] −−help=optimizers

Some other options

Among a lot of other options:

Some other options

Among a lot of other options:

- —Wall for all warnings mode

Some other options

Among a lot of other options:

- –Wall for all warnings mode
- –std=norm to control the dialect of C/C++/... for example 'c11' or 'iso9899:2011' for the C11 norm

Some other options

Among a lot of other options:

- −Wall for all warnings mode
- −std=norm to control the dialect of C/C++/... for example 'c11' or 'iso9899:2011' for the C11 norm
- −l*library* (lowercase L), −L*dir* and −I*dir* (uppercase i) to link respectively library file, directory of library files and directory of header files

Some other options

Among a lot of other options:

- −Wall for all warnings mode
- −std=norm to control the dialect of C/C++/... for example 'c11' or 'iso9899:2011' for the C11 norm
- −l*library* (lowercase L), −L*dir* and −I*dir* (uppercase i) to link respectively library file, directory of library files and directory of header files
- −g*level* to generate different level of information for GDB debugging

Some other options

Among a lot of other options:

- −Wall for all warnings mode
- −std=norm to control the dialect of C/C++/... for example 'c11' or 'iso9899:2011' for the C11 norm
- −l*library* (lowercase L), −L*dir* and −I*dir* (uppercase i) to link respectively library file, directory of library files and directory of header files
- −g*level* to generate different level of information for GDB debugging
- −fPIC to generate Position Independent Code, meaning it is not dependent on being located at a specific address in order to work. (for example jumps are position dependent: jump from instruction 5 to 11 will be from $CURRENT to $CURRENT+6)

Section 11

Software architecture

Good coding practices

- code indentation: choose a style (and stay on it)

Good coding practices

- code indentation: choose a style (and stay on it)
- meaningful naming

Good coding practices

- code indentation: choose a style (and stay on it)
- meaningful naming
- add comments to add context

Good coding practices

- code indentation: choose a style (and stay on it)
- meaningful naming
- add comments to add context
- reduce code duplication by abstracting code into functions

Good coding practices

- code indentation: choose a style (and stay on it)
- meaningful naming
- add comments to add context
- reduce code duplication by abstracting code into functions
- "low coupling, high cohesion": files as independent as possible, related code close to each other inside a file

Good coding practices

- code indentation: choose a style (and stay on it)
- meaningful naming
- add comments to add context
- reduce code duplication by abstracting code into functions
- "low coupling, high cohesion": files as independent as possible, related code close to each other inside a file
- choose a convention and follow it for indentation, naming,...

Indentation style

- K&R style (Kernighan & Ritchie Style):

```
int main()
{
  if (x == 2) {
    try_this();
  } else {
    finish_this();
  }
  end();
}
```

- Whitesmiths style

```
int main()
    {
    if (x == 2)
        {
        try_this();
        }
    else
        {
        finish_this();
        }
    end();
    }
```

- GNU style

```
int
main()
{
  if (x == 2)
    {
      try_this();
    }
  else
    {
      finish_this();
    }
  end();
}
```

- Pico style (unpopular in C)

```
int main()
{ if (x == 2)
  { try_this();
    and_this(); }
  else
  { finish_this(); }
  end(); }
```

Naming convention

In C and C++, identifiers are mostly lowercase:

- in C standard library: abbreviated names, e.g. isalnum for a function testing if a char is alphanumeric

Naming convention

In C and C++, identifiers are mostly lowercase:

- in C standard library: abbreviated names, e.g. isalnum for a function testing if a char is alphanumeric
- in C++ standard library: underscore as a word separator, e.g. out_of_range for a type of exception used when attempting to access elements out of defined range.

Naming convention

In C and C++, identifiers are mostly lowercase:

- in C standard library: abbreviated names, e.g. isalnum for a function testing if a char is alphanumeric
- in C++ standard library: underscore as a word separator, e.g. out_of_range for a type of exception used when attempting to access elements out of defined range.
- macros are only in uppercase letters and underscore

Naming convention

In C and C++, identifiers are mostly lowercase:

- in C standard library: abbreviated names, e.g. isalnum for a function testing if a char is alphanumeric
- in C++ standard library: underscore as a word separator, e.g. out_of_range for a type of exception used when attempting to access elements out of defined range.
- macros are only in uppercase letters and underscore
- names containing double underscore or beginning with an underscore and a capital letter are reserved for implementation (compiler, standard library), e.g. is__reserved or _Reserved

Naming convention

In C and C++, identifiers are mostly lowercase:

- in C standard library: abbreviated names, e.g. isalnum for a function testing if a char is alphanumeric
- in C++ standard library: underscore as a word separator, e.g. out_of_range for a type of exception used when attempting to access elements out of defined range.
- macros are only in uppercase letters and underscore
- names containing double underscore or beginning with an underscore and a capital letter are reserved for implementation (compiler, standard library), e.g. is__reserved or _Reserved
- in C try to use two_words instead of twoWords

Header file

- files with extension .h

Header file

- files with extension .h
- contains C function declarations (prototypes), macro definitions (using **#define**), data structures and other header file inclusion (using **#include**).

Header file

- files with extension .h
- contains C function declarations (prototypes), macro definitions (using **#define**), data structures and other header file inclusion (using **#include**).
- can be shared between several source files.

Header file

- files with extension .h
- contains C function declarations (prototypes), macro definitions (using **#define**), data structures and other header file inclusion (using **#include**).
- can be shared between several source files.
- are of two types: the files that the programmer writes and the files that come with your compiler.

Header file

- files with extension .h
- contains C function declarations (prototypes), macro definitions (using **#define**), data structures and other header file inclusion (using **#include**).
- can be shared between several source files.
- are of two types: the files that the programmer writes and the files that come with your compiler.
- are included using the preprocessing directive **#include**: the first type is included using: **#include** " header_file .h" and the second type using: **#include** <header_file.h>.

Header file

- files with extension .h
- contains C function declarations (prototypes), macro definitions (using **#define**), data structures and other header file inclusion (using **#include**).
- can be shared between several source files.
- are of two types: the files that the programmer writes and the files that come with your compiler.
- are included using the preprocessing directive **#include**: the first type is included using: **#include** " header_file .h" and the second type using: **#include** <header_file.h>.
- are the place where you search for information about functions and data structure without having to read the .c files.

Once-only headers

If the header file is included twice, the compiler will process its contents twice and raise an error. To prevent this, we surround the content of the file with:

```
#ifndef HEADER_FILE
#define HEADER_FILE

// content of the header file header_file.h

#endif
```

Debugging

A multistep process:

- identify a problem

Debugging

A multistep process:

- identify a problem
- isolate the source of the problem

Debugging

A multistep process:

- identify a problem
- isolate the source of the problem
- correct the problem or determinate a way to work around it

Debugging

A multistep process:

- identify a problem
- isolate the source of the problem
- correct the problem or determinate a way to work around it
- test the correction or workaround and repeat the process if it is not sufficient

Debugging

A multistep process:

- identify a problem
- isolate the source of the problem
- correct the problem or determinate a way to work around it
- test the correction or workaround and repeat the process if it is not sufficient

Most IDE have an integrated debugger, otherwise use gdb

gdb

- GNU Debugger

gdb

- GNU Debugger
- for several languages (C, C++,...)

gdb

- GNU Debugger
- for several languages (C, C++,...)
- inspect the program at a certain point during execution

gdb

- GNU Debugger
- for several languages (C, C++,...)
- inspect the program at a certain point during execution
- add −g option to the compile command to enable built-in debugging support

```
#include<stdio.h>

int main(){

   int x;
   int a=x;
   int b=x;
   int c=a+b;
   printf("%d\n",c);
   return 0;
}
```

Compiled using gcc −g −o debug debug.c.

If an object is not initialized explicitly, its value is indeterminate, where the indeterminate value is either an unspecified value or a trap representation.

gdb

Commands

- run or r: execute the program from start to end

gdb

Commands

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint
- next or n: executes next line of code, but as single instruction (don't dive into functions)

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint
- next or n: executes next line of code, but as single instruction (don't dive into functions)
- step or s: go to next instruction, diving into functions

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint
- next or n: executes next line of code, but as single instruction (don't dive into functions)
- step or s: go to next instruction, diving into functions
- list or l: displays the code

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint
- next or n: executes next line of code, but as single instruction (don't dive into functions)
- step or s: go to next instruction, diving into functions
- list or l: displays the code
- print *var* or p *var*: displays the stored value of a variable (watch *var* displays any change on this variable)

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint
- next or n: executes next line of code, but as single instruction (don't dive into functions)
- step or s: go to next instruction, diving into functions
- list or l: displays the code
- print *var* or p *var*: displays the stored value of a variable (watch *var* displays any change on this variable)
- quit or q: exits out of gdb

gdb

Commands

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint
- next or n: executes next line of code, but as single instruction (don't dive into functions)
- step or s: go to next instruction, diving into functions
- list or l: displays the code
- print *var* or p *var*: displays the stored value of a variable (watch *var* displays any change on this variable)
- quit or q: exits out of gdb
- clear : clear all breakpoints

- run or r: execute the program from start to end
- break *file_name:nbr* or b *file_name:nbr*: sets breakpoint to line *nbr* from file *file_name*
- disable : disable a breakpoint
- enable: enable a disabled breakpoint
- next or n: executes next line of code, but as single instruction (don't dive into functions)
- step or s: go to next instruction, diving into functions
- list or l: displays the code
- print *var* or p *var*: displays the stored value of a variable (watch *var* displays any change on this variable)
- quit or q: exits out of gdb
- clear : clear all breakpoints
- continue or c: continue normal execution

Section 12

Conclusion

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables
  - loops and conditions

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables
  - loops and conditions
  - use of pointers

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables
  - loops and conditions
  - use of pointers
- good coding practices
  - split your code into functions that you can use from different places

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables
  - loops and conditions
  - use of pointers
- good coding practices
  - split your code into functions that you can use from different places
  - structure your program into files with corresponding

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables
  - loops and conditions
  - use of pointers
- good coding practices
  - split your code into functions that you can use from different places
  - structure your program into files with corresponding
  - choose wisely your data structure (for example lists if you need to insert datas, arrays if not)

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables
  - loops and conditions
  - use of pointers
- good coding practices
  - split your code into functions that you can use from different places
  - structure your program into files with corresponding
  - choose wisely your data structure (for example lists if you need to insert datas, arrays if not)
  - remember that other people need to understand your code (including the future you from 2 years from now)

Conclusion

- algorithms allows us to know for any language:
  - the feasibility of a problem,
  - the complexity, the expected time to have an answer,
  - the correction of a solution
- C language
  - typed variables
  - loops and conditions
  - use of pointers
- good coding practices
  - split your code into functions that you can use from different places
  - structure your program into files with corresponding
  - choose wisely your data structure (for example lists if you need to insert datas, arrays if not)
  - remember that other people need to understand your code (including the future you from 2 years from now)
  - for these two years, your teachers might need to understand it easily