Thomas Grenier

Computer Science 1

Thomas Grenier

M.Sc IMESI

Summary

International Master in Embedded Systems and Medical Image Engineering

IMESI

Computer Science

Computer Science 1&2

Modules overview

Introduction

- Why computer science modules?
  - Fundamentals of embedded systems and digital processing
  - Refresher courses!

➔ 1st semester:

- C/C++ and object programming (UML)
- Algorithms for discrete mathematics & operational research (and many other things!)

➔ 2nd semester:

- Real time programming
- Network : TCP/IP, client/server application, …

Introduction

- Who ?
  - For CS1: coordinator J. Fondrevelle
    - *Thomas Grenier*, Assistant Professor, Ph.D
      - Lab: Creatis
      - Teaching: INSA, Génie Electrique (Electrical Engineering)
    - *Julien Fondrevelle*, Assistant Professor, Ph.D
      - Lab: LIESP-Ampère
      - Teaching: INSA, Génie Industriel
  - For CS2: coordinator T. Glatard
    - *Arnaud Lelevé*, Assistant Professor, Ph.D
      - Lab: Ampère
      - Teaching : INSA Génie Industriel
    - *Tristan Glatard*, CNRS CR, Ph.D
      - Lab: Creatis
      - Research area: Grid of computers

Organization

- CS1 : 3 ECTS
  - Lectures : 30 hours
  - Project : 5 hours advising plus 4-week individual work
  - Written examination: 2 hours

- CS2: 2 ECTS (2nd semester…)
  - Lectures: 20 hours
  - Project: ~5 hours
  - Written examination: 2 hour

Schedule CS1

- Weeks 39-43: Algorithms and C++
  - T. Grenier
- Weeks 43-45: Discrete mathematics and operation research
  - J. Fondrevelle
- Weeks 43-48: Project
  - J. Fondrevelle
- Week 04: exam
  - J. Fondrevelle

Questions ?

Computer Science 1

Introduction

M.Sc IMESI

Summary

Introduction

1. Why studying algorithms ?
   a) Definitions and interests of algorithms design
   b) *Pseudocode*
   c) Efficiency
   d) Correctness
2. Why learning C++ ?
   a) Languages history and future
   b) Object programming language
   c) C++

1.Why studying algorithms ?

   a) Definitions and interests of algorithms design
   b) P*seudocode*
   c) Efficiency
   d) Correctness

1.a – Definitions and interests

- Algorithm

  *Any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output***

➔ Tool for solving well-specified computational problem

1.a – Definitions and interests

- Example
  - Sort a sequence of numbers into non-decreasing order
    - ***Input***: sequence of ***n*** numbers $\{a_1, a_2, \ldots a_n\}$
    - ***Output***: permutation (reordering) $\{a'_1, a'_2, \ldots a'_n\}$ of the input sequence such as $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

- Exercise:
  - Give an algorithm that computes the average value of ***n*** given numbers.
  - Give an algorithm that computes the median value of ***n*** given numbers.

1.a – Definitions and interests

- Conclusions of the exercise
  - How to specify an algorithm?
  - Is my algorithm correct ?
  - Is my algorithm better than another one ?
  - What kind of problems are solved by algorithms?
  - By the way…is it really interesting to study algorithms?
    - Computers are faster and faster!
    - Memory is cheaper and cheaper!

1.a – Definition and interests

How to specify an algorithms?

- Specifying an algorithm
  - Describe it in French or English or …
  - As computer program
  - As hardware design
  - …
  - ➔ *The only requirement is that the specification must provide a precise description of the computational procedure to be followed*
  - ➔ *But… How to convey the essence of the algorithm*
    - ➔ *Concisely?*
    - ➔ *Without issues of software engineering?*

➔ *We will use a dedicated language : **pseudocode***

1.a – Definitions and interests

Is my algorithm correct ?

- Correctness of an algorithm - definitions
  - An algorithm is said ***correct*** if, for every input instance, it halts with the correct output
  - A correct algorithm ***solves*** the given computational problem
- How to demonstrate an algorithm is correct?
  - ➔ We often use a ***loop invariant*** (similar to mathematical induction)
  - ➔ Sometimes difficult to use… (strange loop, recurrence)

1.a – Definitions and interests

Is my algorithm better than another one ?

- Algorithm analysis
  - Meaning: “predicting the resources that the algorithm requires”
    - ***Memory***, communication bandwidth, ***computational time***, …
  - ➔ By analyzing several algorithms for a problem, a most efficient one can be easily identified
- We generally focus on the computational time
  - The running time of an algorithm depends on the input and the size of the input !
  - ➔ Algorithms efficiency
    - ➔ ***Worst case***, best case, average case analysis
    - ➔ ***Order of growth*** of the running time function of the input size

1.a – Definitions and interests

What kind of problems are solved by algorithms?

- Problems **solved** by algorithms
  - All ? theoretically right (but practically wrong cause of bounded resources…)

  ➔ Use of approximate/convergent algorithms and “incorrect” (with controlled error rate) algorithms
- Challenging problems are linked to
  - Optimization (minimization of a cost function $f(\mathbf{x})$): industrial, logistic, mathematical problems
  - Manipulation of large numbers (related to input size): DNA, chess, cosmology, cryptography, web search engine…
  - Real time

1.a – Definitions and interests

Is it really interesting to study algorithms?

- Guess in few years… Computers were infinitely fast and computer memory were free?
  - Computational time is null, no space memory limit…
  - All right but:
    - Does your process stop?
    - With the correct output ?

  ➔ You have to demonstrate it
- In real world …
  - Computer are fast but not infinitely fast
  - Memory is cheap but not free
  - Energy?
  - Cost?

  ➔ The most efficient algorithms is the best compromise

1.b – Pseudocode

How to specify an algorithm?

- *Pseudocode* to specify an algorithm
  - Example: compute the average of an array

```
AVERAGE(A)
1  ▷ Initialize sum at the first value of A : A[1].
2  sum ← A[1]
3  for j ← 2 to length[A]
4  ▷ Sum each value of A in sum : A[2 .. length[A]].
5        do sum ← sum + A[j]
6  return sum / length[A]
```

→ Indentation
→ Comment

1.b – Pseudocode

- Pseudocode conventions
  - Variables
    - Assignment is : ←
    - Local to the given procedure
    - Have the *right* type (described using mathematical notation)
    - Array elements are accessed by specifying the array name followed by the index in brackets. The first array element is at index 1.
    - Compound data are typically organized into **objects**, which are composed of **attributes** or **fields**. Objects and arrays are treated as pointer.
  - → Examples:

A[1] ← A[2] * A[1]
size ← *length[*A]

1.b – Pseudocode

- Pseudocode conventions
  - Tests: **if-then**, **if-then-else**
    - Indentation indicates block structure!
  - Examples

```
SIMPLE-IF-THEN-TEST(a, b)
1  if a > b
2     then
3             temp ← a
4             a ← b
5             b ← temp
```

```
SIMPLE-IF-THEN-ELSE-TEST(a, b)
     ▷ Test if a is greater than b.
1  if a > b
2     then return TRUE
3     else  return FALSE
```


1.b – Pseudocode

- Pseudocode conventions
  - Loop : **for**, **while**, **repeat-until** (do-while)
    - Indentation indicates block structure!
  - Examples

```
SIMPLE-FOR()
1  for i ← 1 to 10
2        do
3           print " number : " i
```

```
SIMPLE-WHILE()
1  i ← 1
2  while i ≤ 10
3        do
4           print " number : " i
5           i ← i + 1
```

```
SIMPLE-REPEAT()
1  i ← 1
2  repeat
3           print " number : " i
4           i ← i + 1
5     until i > 10
```

1.b – Pseudocode

- Pseudocode conventions
  - Procedure parameters are passed by value!
    - Modification of a value parameter in procedure is not seen by the calling procedure
    - Modification of an object field (or an array element) is seen by the calling procedure (object are passed using pointer…)
  - Example

```
PARAMETERS-TO-PROCEDURE(A, b)
     ▷ the next assignment is not visible to the calling procedure
1    b ← 1
     ▷ the next assignment is visible to the calling procedure
2    A[i] ← 1
```


1.b – Pseudocode

- Pseudocode conventions
  - ***short circuiting*** *of Boolean operators (‘and’, ‘or’ …)*
    - Evaluate the expression “x *and* y”: first evaluate *x*, and then evaluate *y* to determine the whole expression only if *x* is evaluated to True.
    - Evaluate the expression “x *or* y”: first evaluate x, and then evaluate y only if x is evaluated to False.
  - Examples

```
SHORT-CIRCUITING(A, i, y)
     ▷ Cause of short circuiting we don't worry about what happens
     ▷ when we try to evaluate A[i] when i is NIL or out of range
1    if i ≠ NIL and A[i] = y
2       then
3                ...
4    if i ≤ lenght[A] and A[i] = y
5       then
6                ...
```

1.c – Efficiency

- Example: 2 algorithms for a same problem
  - i.e. insertion and merge sort
  - Computational time of these algorithms depends on the input size *n*
    - The first algorithm (insertion sort) takes time roughly equal to $c_1.n^2$ to sort *n* numbers
    - The second algorithm (merge sort) takes time roughly equal to $c_2.n.log_2(n)$ to sort *n* numbers
  - Insertion sort runs on a fast computer A (1 billion instructions per second) and is coded by the world's craftiest programmer. Resulting code requires $2n^2$ instructions to sort *n* numbers
  - Merge sort runs on a slower computer B (ten million instructions per second) and is coded by an average programmer using high level language with inefficient compiler. Resulting code requires $50n.log(n)$ instructions to sort *n* numbers
  - ➔ Give the computational times of each algorithm if n = {1 000, 38 000,1 000 000}

1.c – Efficiency

- Example: 2 algorithms for a same problem
  - Computer A : $\frac{2n^2 \text{instructions}}{10^9 \text{instructions/second}}$
  - Computer B : $\frac{50n\log_2(n) \text{instructions}}{10^6 \text{instructions/second}}$

n	1000	38000	1 000 000	10 000 000
A (in seconds)	0.002	2.888	2000	200000
B (in seconds)	0.050	2.890	99.65	1163

1.c – Efficiency

- Find n=38000 is difficult!

$$\frac{2n^2}{10^9} = \frac{50n.\log_2(n)}{10^6}$$

$$n^2 = 25.10^3.n.\log_2(n)$$

$$n = 25.10^3.\log_2(n)$$

$$\log(n) = \frac{\log(2)}{25.10^3}.n$$

$$W(x) = \sum_{n=1}^{\infty} \frac{(-n)^{n-1}}{n!}.x^n$$

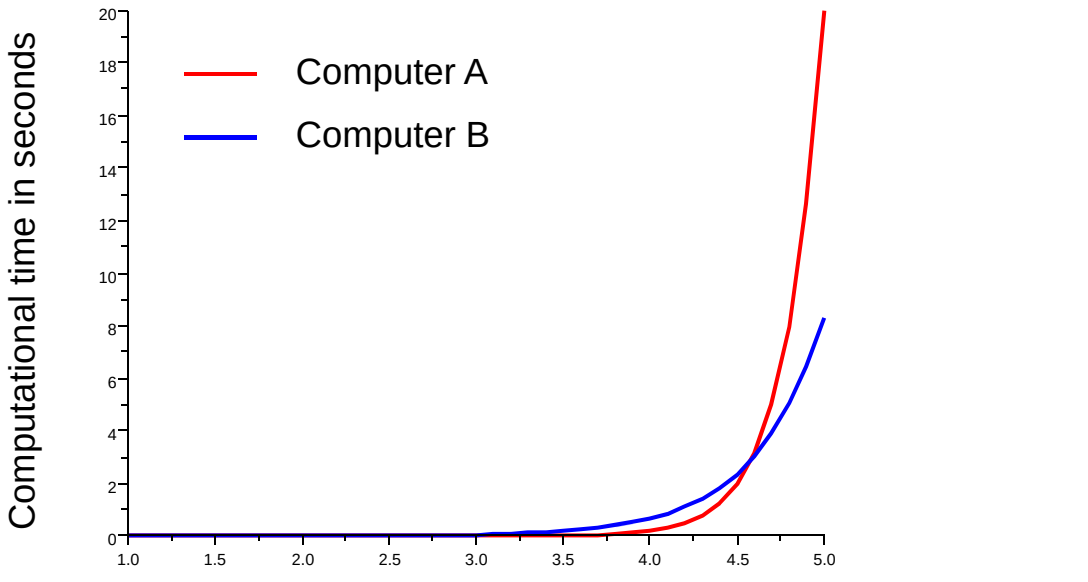
Lambert's transcendental equation

$\log(n) = v.n^{\beta}$ is $n = \exp\left(-\frac{W(-\beta.v)}{\beta}\right)$ With W the Lambert function

$$-25.W(-1/25)$$

1.c – Efficiency

- Example: 2 algorithms for a same problem



1.c – Efficiency

- Definition : Running time of an algorithm $T(n)$
  - On a particular input, that's the number of executed primitive operations (or 'steps')
    - *step* should be defined as machine-independent as possible
    - Particular input can be: best case, worst case or average case
  - For pseudocode
    - a constant amount of time is required to execute each line of pseudocode:
      - The execution of the i^{th} line takes time $\boldsymbol{c_i}$
    - Running time of the algorithm is given by:

$$T(n) = \sum_{i=1}^{number\ of\ lines} c_i.(number\ of\ times\ line\ i\ is\ executed)$$

1.c – Efficiency

- Pseudocode example

	cost	times
AVERAGE(A)		
1 ▷ Initialize sum at the first value of A : $A[1]$.	0	1
2 $sum \leftarrow A[1]$	c_2	1
3 **for** $j \leftarrow 2$ **to** $length[A]$	c_3	n
4 ▷ Sum each value of A in sum : $A[2\,..\,length[A]]$.	0	n-1
5 **do** $sum \leftarrow sum + A[j]$	c_5	n-1
6 **return** $sum\ /\ length[A]$	c_6	1

$$T(n) = c_2 + n.c_3 + (n-1).c_5 + c_6$$
$$= n(c_3 + c_5) + c_2 - c_5 + c_6$$

1.c – Efficiency

- Order of growth
  - To simplify computational time analysis:
    - All c_i are equal to a constant time

$$\rightarrow T(n) = a.n + b$$

    - Consider only the leading term of formula, since the lower-order terms are relatively insignificant for large *n*

$$\rightarrow T(n) = a.n$$

- Asymptotic notation
  - We asymptotically bound the function $T(n)$
    - Asymptotic upper bound: *O*-notation $\rightarrow T(n) = O(n)$
    - Asymptotic upper and lower bounds: Θ-notation

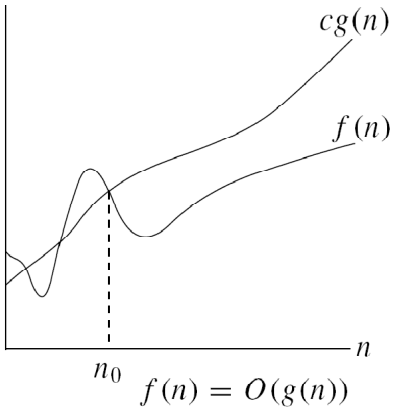
$$\rightarrow T(n) = \Theta(n)$$

Which is stronger than O(n)

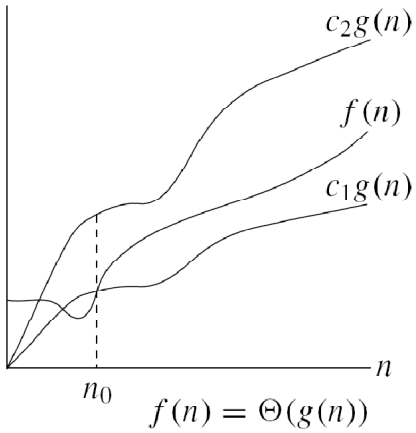
1.c – Efficiency

- Two classical asymptotic notations

$O(n)$ "big-oh"



$\Theta(n)$ "Theta"



$$O(g(n)) = \{f(n) : \exists c\ , n_0 > 0 | 0 \le f(n) \le cg(n), \forall n \ge n_0\}$$

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 | 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n \ge n_0\}$$

1.c – Efficiency

- Exercise
  - Find the best-case and the worst-case running time of Insertion-Sort
  - Give the order of growth

→ Let t_j be the number of times the 'while' instruction is executed at iteration j

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          i ← j − 1
4          while i > 0 and A[i] > key
5               do A[i + 1] ← A[i]
6                  i ← i − 1
7          A[i + 1] ← key
```


1.c – Efficiency

- Exercise solution

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*[A]	c_1	n
2 **do** *key* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

1.c – Efficiency

- Solution

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Best case running time: A is already sorted → t_j =1 for *j=2..n*

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Worst case running time: A is in reverse sorted order → $t_j = j$ for *j=2..n*

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

1.d – Correctness

- We often use ***Loop invariant*** to understand why an algorithm gives the correct answer
  - *In computer science, a predicate that, if true, will remain true throughout a specific sequence of operations, is called (an) **invariant** to that sequence. (Wikipedia)*
- Proof of correctness is trivial… or very hard
  - Average proof of correctness is trivial
  - Insertion-Sort proof of correctness is also trivial!

1.d – Correctness

- To use **loop invariant** to prove correctness, we must show three things about it:
  - **Initialization**: It is true prior the first iteration of the loop
  - **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration
  - **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
- The **invariant** must be correctly defined

1.d – Correctness

- Example **Loop invariant** for Insertion-Sort
  - ➔ Invariant : the subarray A[1..j-1] is sorted
  - **Initialization**: before the first iteration j=2.
    - The subarray A[1] is sorted!
  - **Maintenance**:
    - Problem of the inner ***while*** loop
      - use another loop invariant,
      - or simply note that the ***while*** loop moves A[j-1], A[j-2],…by one position to the right until proper position for key is found
  - **Termination**:
    - The outer 'for' loop ends when j>n, this occur when j=n+1
    - Therefore n= j-1
    - Thus the subarray A[1..j-1] consists of the elements originally in A[1..n] but in sorted order

2.Why studying C++ ?

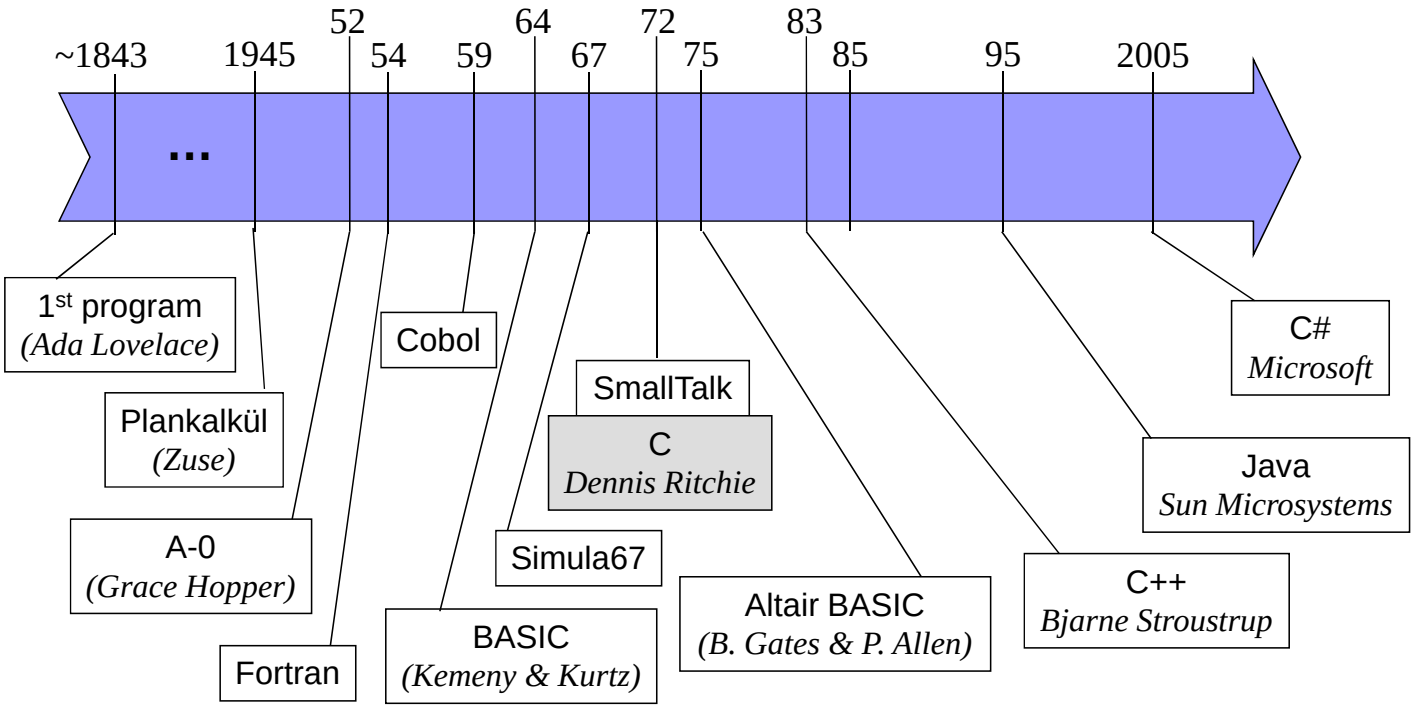
a) Languages history and future
b) Object programming language
c) C++ ?

2.a – Languages history and future

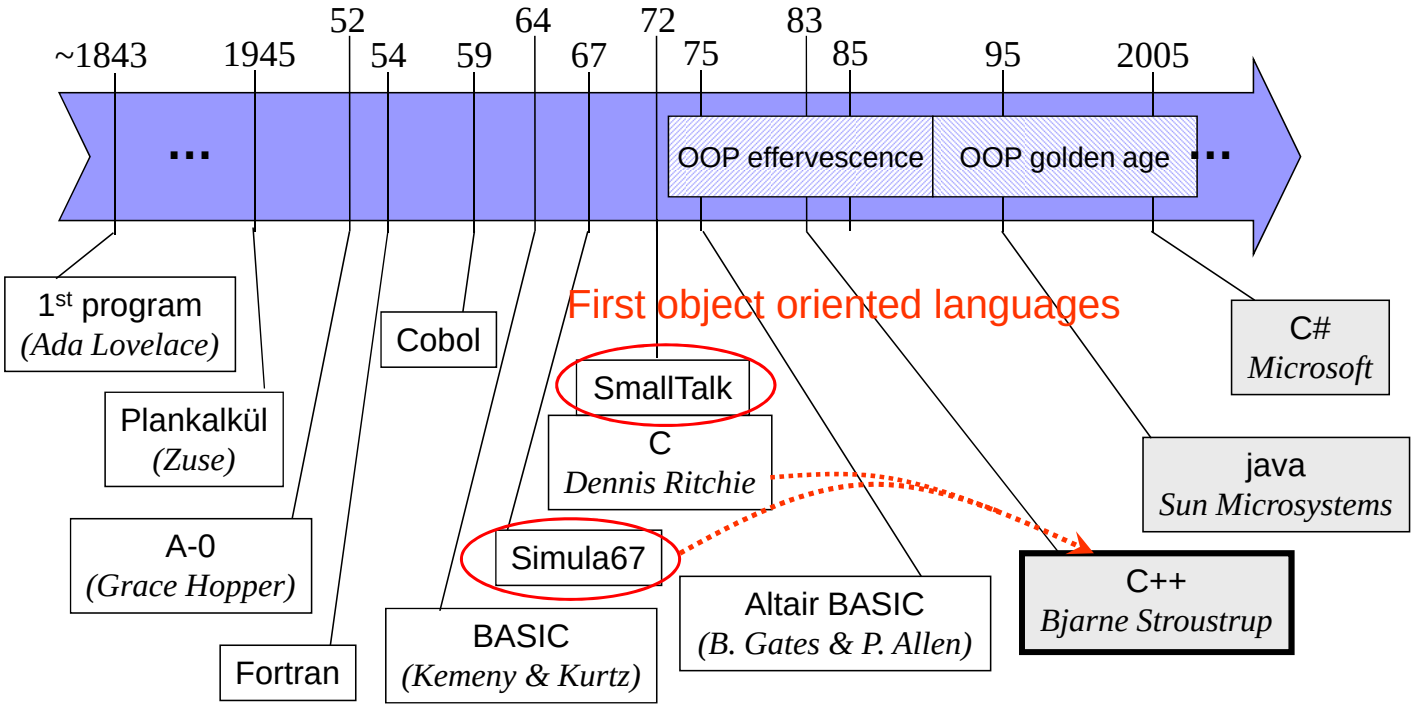
- When was the first program written ?

Ada Lovelace
1815-1852

2.a – Languages history and future



2.a – Languages history and future



OOP: oriented-object programming

2.a – Languages history and future

- What kind of language for the future?
  - Based on object-oriented language or on new paradigms
  - Very high level language
    - Hardware abstraction (like java)
    - Data type abstraction (cf. python)
    - Full object communication, reflection (ie. objectiveC)
    - Intelligent operators (*matlab*)
  - Visual (3D) programming language
    - Link objects and/or functions together (Access, Simulink)
    - No syntax language for algorithms (LabView)

2.b – Object-oriented languages

- The most famous are C++ and java
- Now new standards of each language include object oriented capabilities
  - Cobol, Basic, matlab, fortran, Perl , PHP…
- In real-world
  - OOP can be used to translate from real-world phenomena to program elements (and vice versa)
- *Specify* OO Program → Modeling
  - UML: Unified Modeling Language

2.b – Object oriented languages

- Fundamental paradigms (top 3)
  - **Encapsulation**
    - ***fields*** and ***methods*** are merged into ***class***

    ➔ ***Object*** (or ***instance***) is a pattern (exemplar) of class
  - **Inheritance**
    - ***Subclasses*** are more specialized versions of a class, which ***inherit*** attributes and behaviors from their parent classes, and can introduce their own fields and methods
  - **Polymorphism**
    - Polymorphism allows the programmer to treat ***derived*** class members just like their parent class' members

2.c – C++

- Characteristics of C++
  - Combination of both high and low level language features
  - Statically typed
  - **Object oriented language**
  - Procedural programming
  - Data abstraction (or at least this possibility can be offered…)
  - Generic programming (template)
  - Operators overloading
  - …

New standard is C++0x … promising for the future

2.c – C++

- About the C++ syntax
  - Similar to C (useful for low level programming)
  - Similar to many languages (java, C#, Pascal, matlab)
  - Not so far from pseudocode ☺
    - The first array element is at index 0 …

INSA LYON

International Master in Embedded Systems and Medical Image Engineering

IMESI

Computer Science 1

Sorting Problems

M.Sc IMESI

Summary

Sorting algorithms

- Definition of the sorting problem
  - Sort a sequence of numbers into non-decreasing order
    - ***Input***: sequence of ***n*** numbers $\{a_1, a_2, \ldots a_n\}$
    - ***Output***: permutation (reordering) $\{a'_1, a'_2, \ldots a'_n\}$ of the input sequence such as $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
  - The input sequence is usually a *n*-element array
  - Numbers to be sorted are rarely isolated values
    - Part of data collection called a ***record***
    - Each record contains a ***key***, which is the value to be sorted
    - The remainder of the record consists of ***satellite data***

Sorting algorithms

- Why sorting?
  - Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms
    - Easy to understand
    - Inherent in many applications
    - Used as subroutine
      - (graphical layered objects render, …)
    - There is a wide variety of sorting algorithms
      - Large set of techniques are used (memory, data structure, recurrence)
      - Application of correctness and efficiency demonstrations

List of sorting algorithms

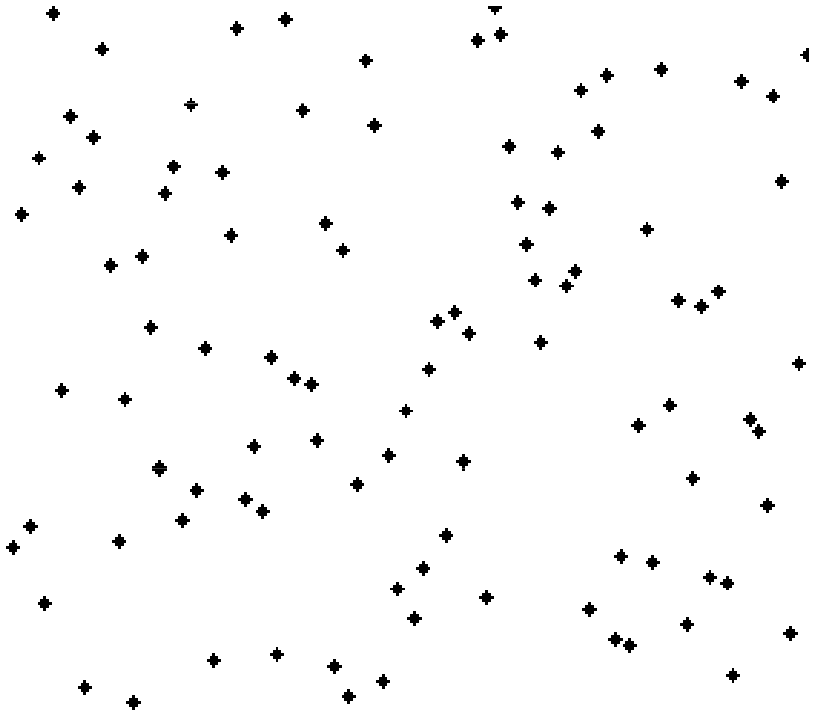
- More than 20 good sorting algorithms
  - Wikipedia → list of sorting algorithm
- Popular sorting algorithms
  - ***Bubble-Sort***
  - ***Insertion-Sort, Selection-Sort***
  - Shell-Sort
  - ***Merge-Sort, Quick-Sort***
  - Heapsort
  - ***Counting-Sort***, BucketSort,

} IntroSort

Bubble-Sort algorithm

- Method
  - The algorithm works as follows:
    - The algorithm starts at the beginning of the data set.
    - It compares the first two elements, and if the first one is greater than the second one, it swaps them.
    - It continues doing this for each pair of adjacent elements to the end of the data set.
    - And then it starts again with the first two elements, repeating until no swaps have occurred on the last pass.
- Write the Bubble-Sort algorithm
- *Show* the correctness of the Bubble-Sort algorithm
- Give the time complexity (worst and best case)

Bubble-Sort algorithm, *animation*



Bubble-Sort algorithm *While*

```
WHILE-BUBBLE-SORT(A)
1  swap ← TRUE
2  while swap = TRUE
3        do swap ← FALSE
4           for i ← 1 to length[A] − 1
5                 do if A[i] > A[i + 1]
6                       then exchange A[i] ↔ A[i + 1]
7                            swap ← TRUE
```

➔Correctness :

➔ the ***for*** inner loop improve the sorting sequence of A

➔The j^{th} highest value of A moves to j^{th} index at the j^{th} execution of this for loop

➔The process ends when no permutation occurs :

➔each A[i] is less or equal than A[i+1]

➔ Best case: already sorted array $T(n) = O(n)$

➔Worst case: invert sorted array $T(n) = O(n^2)$

➔Last element has to move to the first index

➔ Memory: (defined later) $M(n) = O(1)$

Bubble-Sort algorithm, *For*

- Another version of *Bubblesort*

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2       do for j ← length[A] downto i + 1
3            do if A[j] < A[j − 1]
4                 then exchange A[j] ↔ A[j − 1]
```

➔ Give the time complexity (worst and best case)
➔ Prove the correctness of the *Bubblesort* algorithm

Bubble-Sort algorithm, *For*

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2       do for j ← length[A] downto i + 1
3            do if A[j] < A[j − 1]
4                 then exchange A[j] ↔ A[j − 1]
```

$$\begin{aligned}\sum_{i=1}^{n}(n-i) &= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i \\ &= n^2 - \frac{n(n+1)}{2} \\ &= n^2 - \frac{n^2}{2} - \frac{n}{2} \\ &= \frac{n^2}{2} - \frac{n}{2}.\end{aligned}$$

Loop invariant: At the start of each iteration of the **for** loop of lines 2–4, $A[j] = \min\{A[k] : j \le k \le n\}$ and the subarray $A[j..n]$ is a permutation of the values that were in $A[j..n]$ at the time that the loop started.

Loop invariant: At the start of each iteration of the **for** loop of lines 1–4, the subarray $A[1..i-1]$ consists of the $i-1$ smallest values originally in $A[1..n]$, in sorted order, and $A[i..n]$ consists of the $n-i+1$ remaining values originally in $A[1..n]$.

Insertion-Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

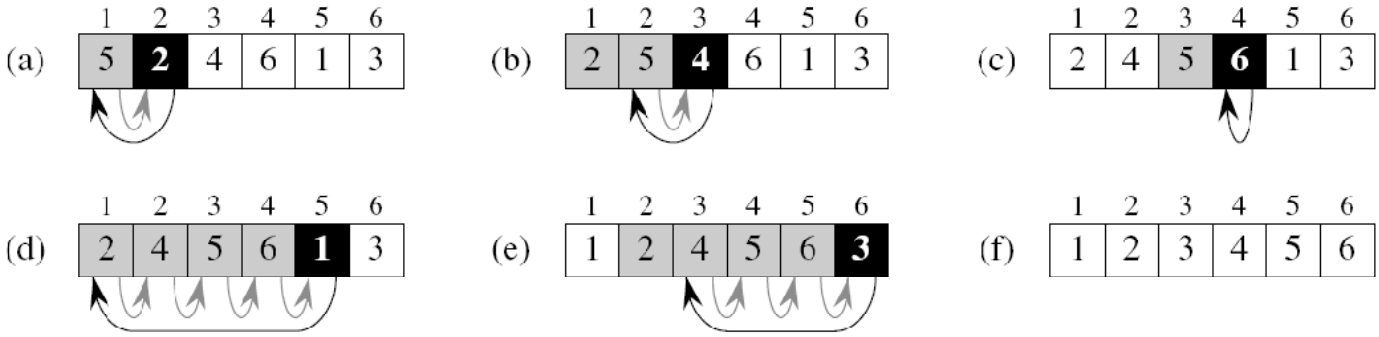
➔ Time complexity $T(n) = O(n^2)$

➔ Memory $M(n) = O(1)$

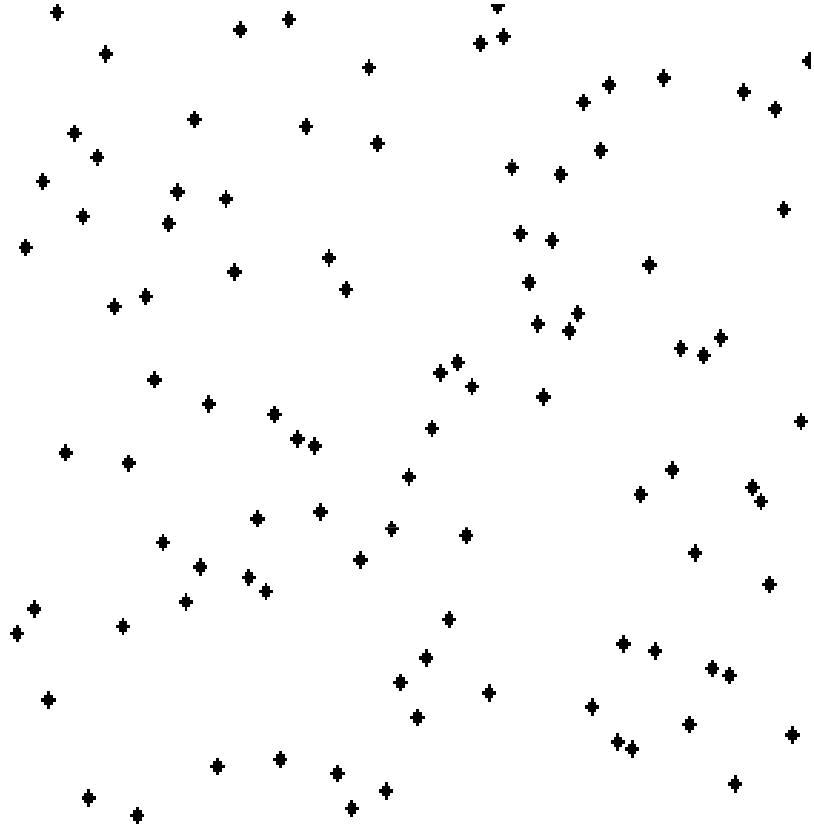
"Memory" denotes the amount of auxiliary storage needed beyond that used by the array itself

Insertion-Sort

- How this algorithm works



Insertion-Sort, *animation*



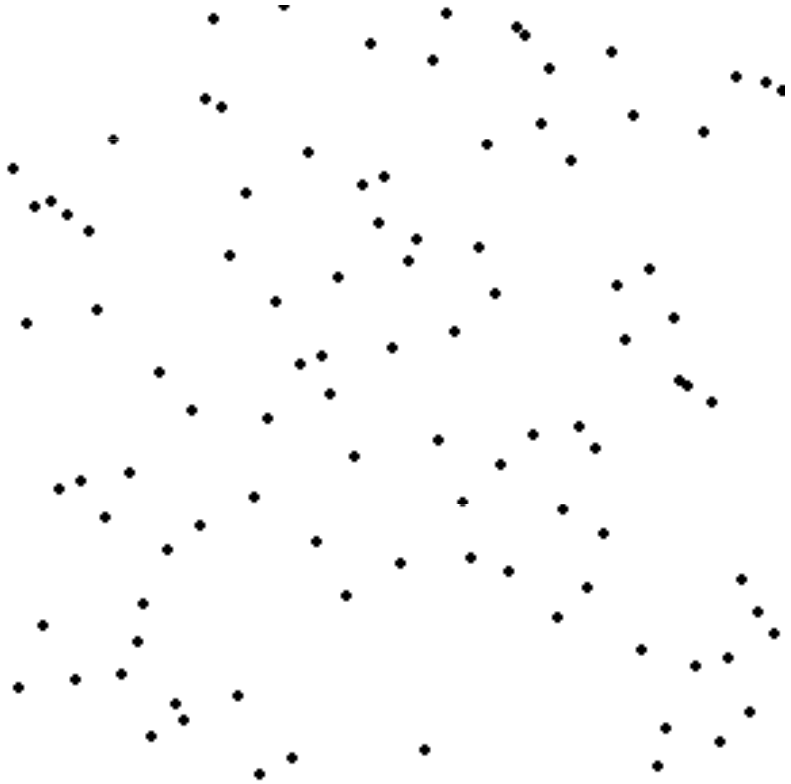
Selection-Sort algorithm

- Method
  - The algorithm works as follows:
    - Find the minimum value in the array
    - Swap it with the value in the first position
    - Repeat the steps above for remainder of the array (starting at the next position)

➔ Write the Selection-Sort algorithm

➔ Give the time and memory complexities of this algorithm

Selection-Sort algorithm, *animation*



Selection-Sort algorithm

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                 then smallest ← i
       exchange A[j] ↔ A[smallest]
```

Not-In-Place Selection-Sort algorithm

- Give a sorting algorithm using 2 different arrays based on Selection-Sort algorithm
- Give the time complexity
- Give the memory complexity
- Compare these values to values of the In-place version

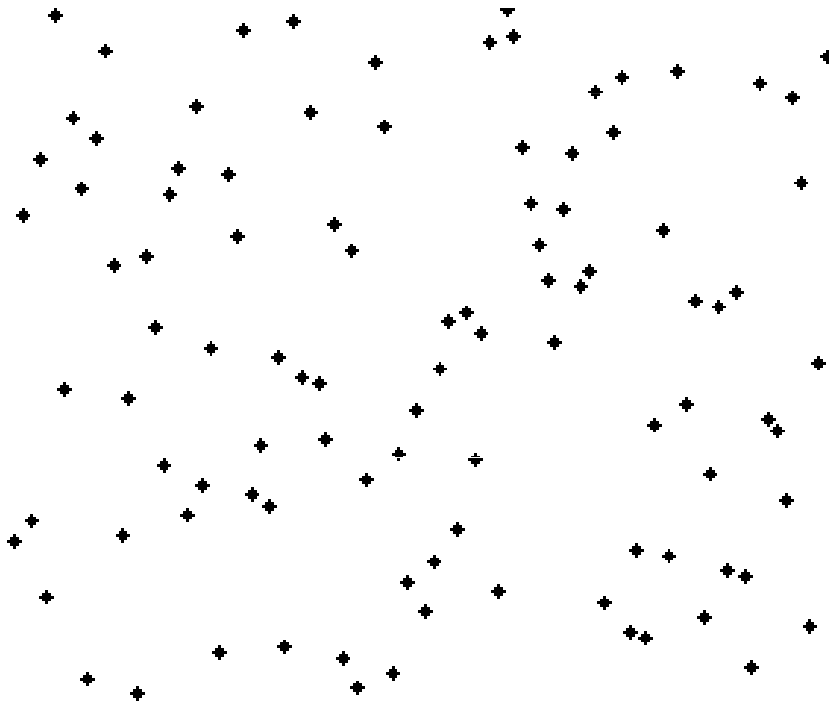
Merge-Sort algorithm, *recursive*

- Recursive algorithms
  - To solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems
  - These algorithms typically follow a ***Divide and conquer*** approach
    - ***Divide*** the problem into a number of subproblems
    - ***Conquer*** the subproblems by solving them recursively. If the subproblem sizes are small enough, just solve the subproblems in a straightforward manner
    - ***Combine*** the solutions of the subproblems into the solution of the original problem

Merge-Sort algorithm, *recursive*

- For Merge-Sort
  - **Divide** the n-element sequence to be sorted into ***two*** subsequences of n/2 elements
  - **Conquer**: Sort the two subsequences recursively using Merge-Sort
  - **Combine**: Merge the two sorted subsequences to produce the sorted answer
  - ➔Write the Merge(A,p,q,r) algorithm which merges two sorted sequences (A[p..q] and A[q+1..r]).

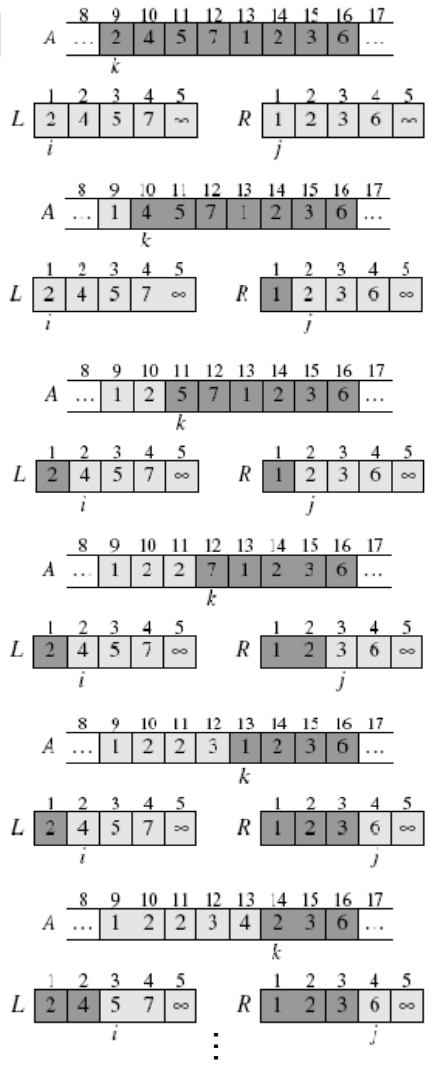
Merge-Sort algorithm, *animation*



Merge-Sort algorithm, *recursive*

- Write the Merge(A,p,q,r) algorithm which merges two sorted subarrays (A[p..q] and A[q+1..r]),and then forms a single sorted subarray that replaces the current subarray (A[p..r])
  - Hints : create 2 subarrays


```
MERGE(A, p, q, r)
 1  create arrays L[1 .. q − p + 1] and R[1 .. r − q]
    ▷ First : copy the A array in the subarrays L and R
 2  for Li ← 1 to Length[L]
 3        do L[Li] ← A[Li + p − 1]
 4  for Ri ← 1 to Length[R]
 5        do R[Ri] ← A[Ri + q]
    ▷ Second : merge L and R to A (sorted!)
 6  Li ← 1
 7  Ri ← 1
 8  for Ai ← p to r
 9        do if Li ≤ Length[L] and Ri ≤ Length[R]
10              then if L[Li] ≤ R[Ri]
11                      then A[Ai] ← L[Li]
12                           Li ← Li + 1
13                      else A[Ai] ← R[Ri]
14                           Ri ← Ri + 1
15              else
                      ▷ End of subarrays
16                    if Li ≤ Length[L]
17                      then A[Ai] ← L[Li]
18                           Li ← Li + 1
19                    if Ri ≤ Length[R]
20                      then A[Ai] ← R[Ri]
21                           Ri ← Ri + 1
```



Merge-Sort algorithm, *recursive*

- Write the Merge-Sort(*A,p,r*) algorithm which sort recursively the array *A[p..r]*
  - If p ≥ r the subarray contains at most one element and then it is already sorted
  - Otherwise, the divide step computes an index q that partition A[p..r] into 2 subarrays
    - A[p..q] containing $\lceil n/2 \rceil$ elements
    - A[q+1..r] containing $\lfloor n/2 \rfloor$ elements

Merge-Sort algorithm, *recursive*

- Write the Merge-Sort(*A,p,r*) algorithm which recursively sort the array *A[p..r]*

```
MERGE-SORT(A, p, r)
1  if p < q
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Merge-Sort algorithm, *recursive*

- Running time analysis
  - Intuitively
    - Merge procedure takes O(n)
    - Merge-Sort procedure calls Merge procedure $\log_2(n)$ times

  ➔ Merge-Sort algorithms running time is $O(n.\log_2(n))$

  - Master theorem approach

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + C(n) + D(n) & \text{otherwise} \end{cases}$$

Straightforward solution

Number of subproblems

Subproblems depending value

Combine time

Divide time

Merge-Sort algorithm, *recursive*

- Master theorem

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + C(n) + D(n) & \text{otherwise} \end{cases}$$

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Merge-Sort algorithm, *recursive*

- Master theorem applied to Merge-Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(1) & \text{if } n > 1 \end{cases}$$

thus $$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + c.n & \text{if } n > 1 \end{cases}$$

then $$\Theta(n^{\log_2(2)}) = \Theta(n) = c.n \quad \Rightarrow \quad T(n) = \Theta(n \log_2(n))$$

The constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps

Merge-Sort algorithm, *recursive*

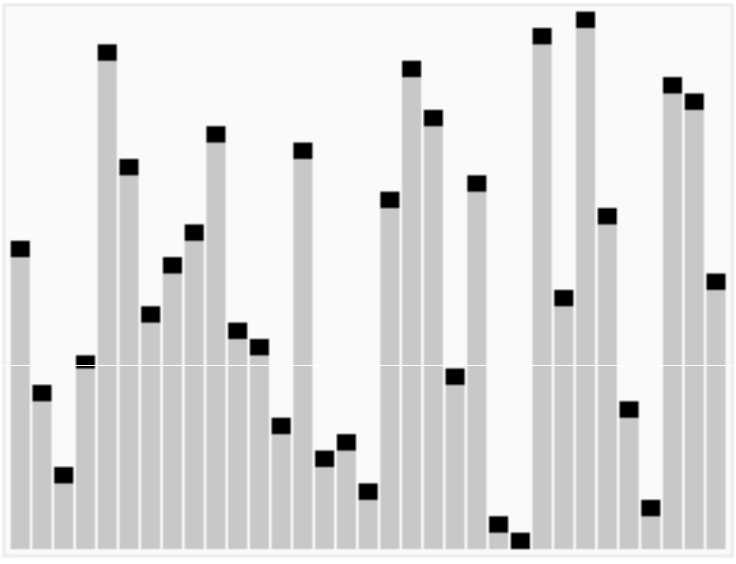
- Memory needed by the Merge-Sort algorithm?
  - The A array (normal…)
  - At the last step the summed size of subarrays is *n*

➔ M(n)=O(n)

Quicksort

- Method
  - Quicksort sorts by using a divide and conquer strategy to divide a array into two sub-arrays
  - The steps are:
    - Pick an element, called a ***pivot***, from the array
    - Reorder the array so that all elements which are smaller than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go whatever the way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
    - Recursively sort the sub-array of smaller elements and the sub-list of greater elements
  - The base cases of the recursion are either array of size zero or of size one, which are always sorted

Quicksort, *animation*



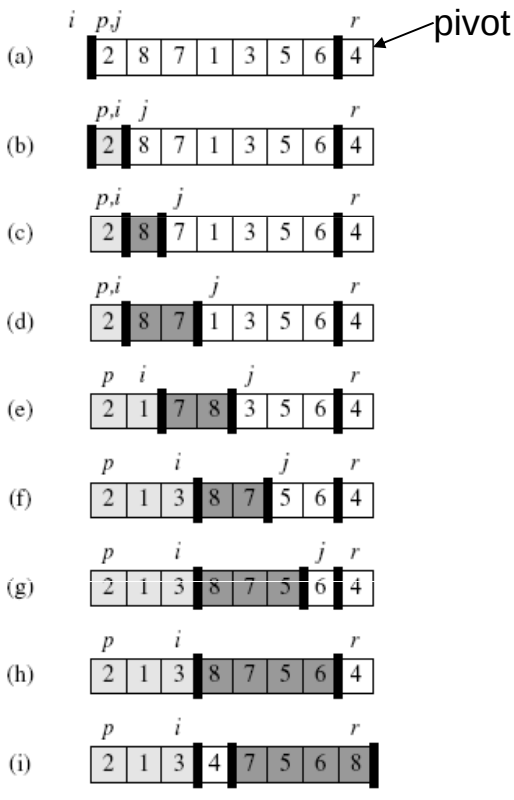
Quicksort

- Write the quicksort algorithm
- Compare time complexity of quicksort with merge-sort
- Compare memory used by quicksort to memory used by merge-sort

Quicksort algorithm

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```



Partition procedure

Sorting in O(n)

- Game…
  - Students vs. teacher

INSA LYON

International Master in Embedded Systems and Medical Image Engineering

IMESI

Computer Science 1

Bases of C++ Syntax

M.Sc IMESI

Summary

Bases of C++ syntax

1. C/C++ Syntax
2. Complete source
   - *main* function
   - Includes
   - Preprocessor directives
   - File association (.cpp and .h)
3. From source code to executable
   - Compilation and Linkage
   - Debugging
4. Exercises

1- C/C++ Syntax overview

- Comment
  - // only one line of comments
  - /* … more than one line of comments */
- Instruction delimiting
  - Each instruction ends by a semicolon '**;**'
  - **{** …**}**: the two braces delimit a block of instructions

1- C/C++ Syntax overview

- Data Type (standard)

Type	Syze	Range of values	Precision
`bool`	1 Byte	*false* or *true*	-
`char`	1 Byte	−128 to 127	-
`unsigned char`	1 Byte	0 to 255	-
`short`	2 Bytes	−32768 to 32767	-
`unsigned short`	2 Bytes	0 to 65535	-
`int`	4 Bytes	−2147483648 to 2147483647	-
`unsigned (int)`	4 Bytes	0 to 4294967295	-
`long`	4 Bytes	−2147483648 to 2147483647	-
`unsigned (long)`	4 Bytes	0 to 4294967295	-
`float`	4 Bytes	$+/-3.4*10^{-38}$ to $3.4*10^{38}$	6 digits
`double`	8 Bytes	$+/-1.7*10^{-308}$ to $1.7*10^{308}$	15 digits
`long double`	10 Bytes	$+/-1.2*10^{-4932}$ to $1.2*10^{4932}$	18 digits

1- C/C++ Syntax overview

- Variable declaration *and initialization*
  - Normal variable
    DataType VariableName *[, Var2]* ;
    DataType VariableName(*construction parameters*);
    DataType VariableName = … ;
  - Pointer
    DataType***** VariableName;
    DataType***** VariableName = …;
  - Reference (& : ampersand)
    DataType **&**VariableName = Variable;
  - Static array (not dynamic)
    DataType VariableName[Number of elements];
    DataType VariableName[] = { list of elements, comma separated};
  - Dynamic array
    DataType VariableName* = **new** *DataType*[*Number of elements*];
    … // instructions
    delete[] *VariableName* ; // memory free

Optional !

```
char ch;
int i,j;
double a(3.1415);
float one = 1.0;

int* _i;
double* _pt = &a;

double& ref = a;

int tab_int[10];
double tab[]=
  {1, 1.5, 2, 2.5, 3, 3.5};
```

1- C/C++ Syntax overview

- Operators
  - Many operators (cf. next table)
    - arithmetic, logical, access, …
  - 17 levels of priority
    - First operator with the highest priority are executed

```
int a = 3 + 6 / 2;     // a = 6
```

  - Associative property
    - How to execute to operator with the same priority

```
int b = 3 * 4 / 2 * 3;     // b = 18
int c = 3 * 4 / ( 2 * 3); // c = 2
```

'*' and '/' have the same priority level (13)

Prio.	Opér.	Signification	Associativité	Exemple
17	::	Résolution de portée (unaire)	droite à gauche	
17	::	Résolution de portée (binaire)	droite à gauche	
16	()	Parenthèse de	gauche à droite	MaFonction(x,y)
16	()	Construction d'objet	gauche à droite	MaClass(x,y)
16	[]	Indexation de tableau	gauche à droite	Tab[i]
16	.	Sélection de composant	gauche à droite	objet.methode();
16	->	Sélection de composant	gauche à droite	this->methode();
15	()	conversion de type explicite	droite à gauche	(double) x; double(x);
15	sizeof	Taille en octets	droite à gauche	sizeof(int);
15	&	Fournit l'adresse mémoire	droite à gauche	&a
15	*	Déférentiation	droite à gauche	
15	~	Négation binaire	droite à gauche	~a;
15	!	Négation logique	droite à gauche	!a;
15	+	Addition (unaire)	droite à gauche	+a;
15	-	Soustraction (unaire)	droite à gauche	-a;
15	++	Incrément	droite à gauche	++a;
15	--	Décrément	droite à gauche	--a;
15	new	Allocation mémoire	droite à gauche	double *t=new;
15	delete	Dé-allocation mémoire	droite à gauche	delete[] t;
14	.*	Sélection de composant		
14	->*	Sélection de composant (pointeur)	gauche à droite	
13	*	Multiplication	gauche à droite	a*b;
13	/	Division	gauche à droite	a/b;
13	%	Modulo	gauche à droite	a%b;
12	+	Addition (binaire)	gauche à droite	a+b;
12	-	Soustraction (binaire)	gauche à droite	a-b;

Prio.	Opér.	Signification	Associativité	Exemple
11	>>	Décalage des bits à droite	gauche à droite	a>>2;
11	<<	Décalage des bits à gauche	gauche à droite	a<<2;
10	>	Test strictement supérieur	gauche à droite	a>2;
10	>=	Test supérieur ou égal	gauche à droite	a>=2;
10	<	Test strictement inférieur	gauche à droite	a<2;
10	<=	Test inférieur ou égal	gauche à droite	a<=2;
9	==	Test d'égalité	gauche à droite	if(choix=='o')
9	!=	Test de non égalité	gauche à droite	if(pt!=NULL)
8	&	ET binaire	gauche à droite	a=1&3; //1
7	^	OU exclusif binaire	gauche à droite	a=1^3; //2
6	|	OU binaire	gauche à droite	a=1|2; //3
5	&&	ET logique	gauche à droite	if(a=1&&b<3)
4	||	OU logique	gauche à droite	if(a=1||b<3)
3	? :	Opérateur de condition (ternaire)	gauche à droite	
2	=	Affectation simple |	droite à gauche	a=b=2;
2	+=	Affectation combinée +	droite à gauche	a+=2; //a=a+2
2	-=	Affectation combinée −	droite à gauche	a-=2; //a=a-2
2	*=	Affectation combinée *	droite à gauche	a*=2; //a=a*2
2	/=	Affectation combinée /	droite à gauche	a/=2; //a=a/2
2	%=	Affectation combinée %	droite à gauche	a%=2; //a=a%2
2	<<=	Affectation combinée <<	droite à gauche	a<<=2; //a*=4
2	>>=	Affectation combinée >>	droite à gauche	a>>=1; //a/=2
2	&=	Affectation combinée &	droite à gauche	a&=1;
2	^=	Affectation combinée ^	droite à gauche	a^=0xFF;
2	|=	Affectation combinée |	droite à gauche	a|=0xFE;
1	,	Séquence d'expressions	gauche à droite	int a,b,c;

1- C/C++ Syntax overview

- Examples
  - Playing with variables and pointers

```
int a = 3; //
int* pt; // pointer declaration
pt = &a; // put address of a
         //     into pt
*pt = 4; // assign 4 to a!
```

  - Mathematical operators
  - Logical operators

1- C/C++ Syntax overview

- Condition
  - Boolean : true or false (==0)
  - Use logical operators to combine conditions

```
(a>=0) && (a<=10)
```

- Tests
  - if, if/else
  - switch/case (only for integer values)

1- C/C++ Syntax overview

- If, if/then

```
if ( condition )
  {
  ...   // These instructions are executed
        //  if condition is true
  }
[ else    //  else: optional
  {
  ...   // These instructions are executed
        //  if condition is false
  }
]
```

These brackets means that this block is optional

1- C/C++ Syntax overview

- Switch case

```
switch ( variable )
  {
  case constante_1 :       //  don't forget the ":" !
    [
      // These instructions are executed
      //  if variable == constante_1
      ]
  [ break ; ]
  case constante_2:
    [ // These instructions are executed
        //  if variable == constante_1        ]
  [ break ; ]
  ...
  [ default:
    // These instructions are executed
    // if no cases above is true
    ]
  }
```


1- C/C++ Syntax overview

- C/C++ loops
  - for
  - while
  - do while

1- C/C++ Syntax overview

- for

```
for ( [initializations]; [condition]; [post-instructions] )
        {
  // These instructions are executed while the condition is true
        }
```

Principle of the **for** loop : 3 steps

1) Initializations
2) *condition* is evaluated
3) if *condition* is true
   then
      the bloc of instructions is executed
      the post-instructions are executed
      the loop continues, goes to step 2)
   else the loop ends

1- C/C++ Syntax overview

- while

```
while ( [condition] )
  {
  // These instructions are executed while the condition is true
  }
```

- do … while

```
do
  {
  // These instructions are executed once
  // and as long as the condition is true
  }
while ( [condition] );
```

1- C/C++ Syntax overview

- Functions
  - Declaration
  - Definition
  - Parameters to/from functions
    - return parameter
    - Copy
    - Pointer/array
    - Reference

1- C/C++ Syntax overview

- Functions declaration (or function prototype)
  - It is used to inform the compiler that the function exists
  - specify the function's name, arity (number of arguments), argument types and return type

```
ReturnType FunctionName( [Type Argument1, [Type Argument2...]] );
```

  - Examples

```
void Print();
double Square(double x);
double Mean( double *Tab, int Size);
```

1- C/C++ Syntax overview

- Functions definition
  - Supply the function body
  - Notice that formal parameters are local variables which are assigned values from the arguments when the function is called

```
ReturnType FunctionName( [Type Argument1, [Type Argument2...]] )
{
ReturnType Variable;
// executed instructions

return Variable; //if ReturnType is not void
                 // The type of Variable must be the same type as ReturnType
}
```


1- C/C++ Syntax overview

- Functions definition
  - Examples

```
void Print()
  { cout<< "Bonjour les étudiants d'IMESI" << endl; }
double Square(double x)
  { return x*x; }
double Mean( double *Tab, int Size)
{
  double mean = 0;
  for( int i=0; i<Size; i++)
    mean += Tab[i];
  return mean/Size;
}
```

1- C/C++ Syntax overview

- Functions parameters
  - Formal parameters
    - By default, argument values are simply copied to the formal parameter variables at the time of the call. This type of parameter passing is called *pass-by-value*.
  - Pointer parameters
    - A *pointer parameter* is indicated by the star (*) that precedes the parameter. The argument value must be an address (a pointer or an address of (&) a variable)
  - Reference perameters
    - A *reference parameter* is indicated by the ampersand (&) that precedes the parameter name. The compiler will then pass the *memory address* of the actual parameter, not the value. A formal reference parameter may be used as a normal variable
  - Return parameter

1- C/C++ Syntax overview

- Function parameter: *pass-by-copy*

```
void fonction(double a)
{
  a = 3;
}
```

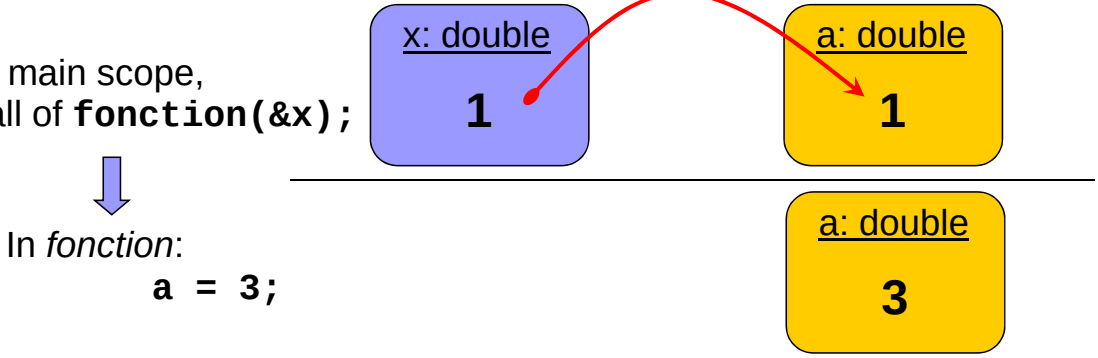
```
void main(void)
{
  double x = 1;
  fonction(x);
  cout << x;
}
```

→ x equal to 1

In memory:

Copy the value

In main scope,
call of **`fonction(&x);`**



In *fonction*:

`a = 3;`

➔ The value of x is NOT modified by the function

1- C/C++ Syntax overview

- Function parameter: *pass-by-address*

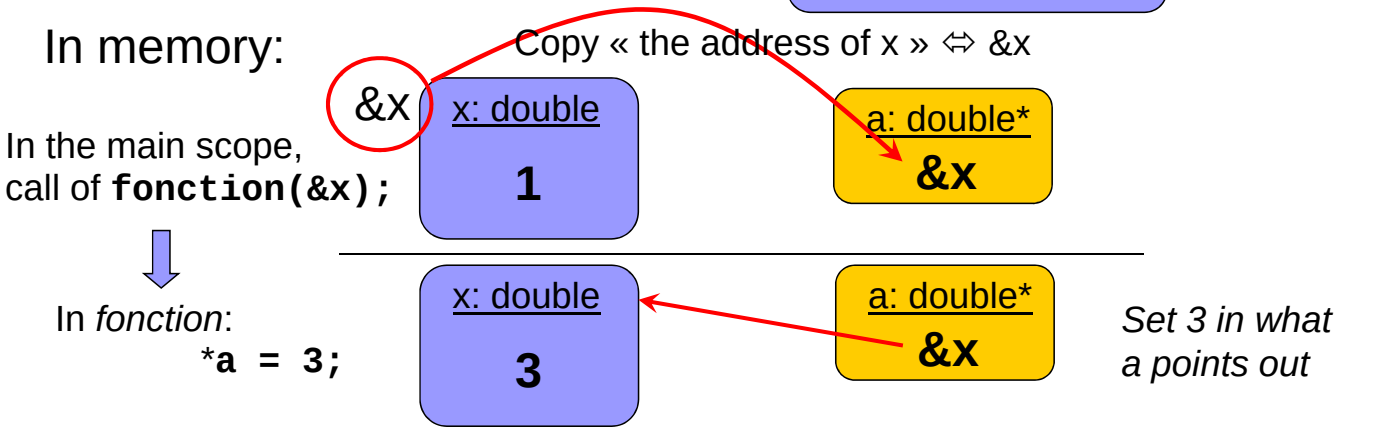
```
void fonction(double *a)
{
  *a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(&x);
  cout << x;
}
```

→ x equal to 3

In memory:

Copy « the address of x » ⇔ &x



➔ The value of x IS modified by the function

1- C/C++ Syntax overview

- Function parameter: *pass-by-address*

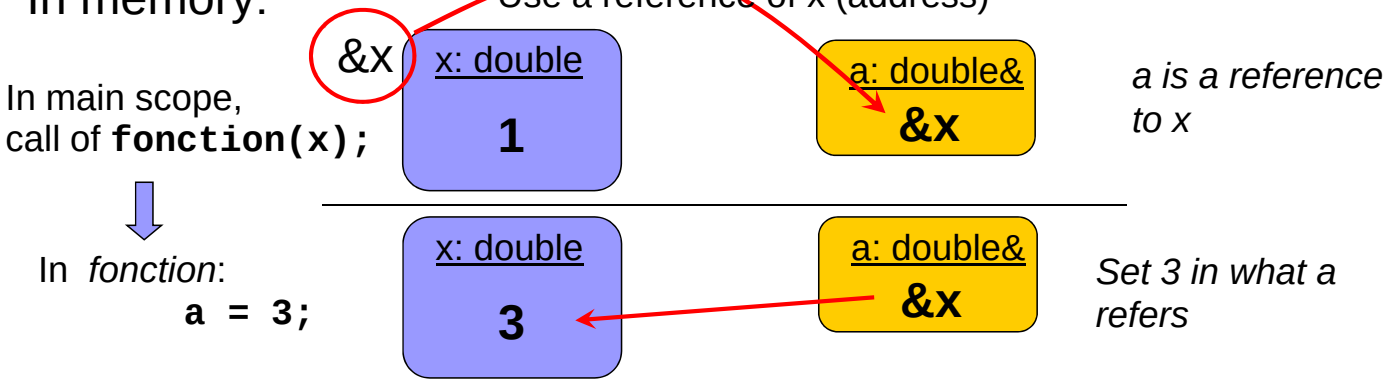
```
void fonction(double &a)
{
  a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(x);
  cout << x;
}
```

→ x equal 3

In memory:

Use a reference of x (address)



➔ The value of x IS modified by the function

1- C/C++ Syntax overview

- Sum up (or *résumé*)

	by-copy	***by-pointer***	***by-reference***
speed	slow	fast	fast
syntax and use	very easy	be carefull!	easy
array takedown	no	yes	no
memory use per param.	size of the object	size of pointer =	size of reference
parameter modification	no	yes	yes

1- C/C++ Syntax overview

- Function parameter: *return parameter*

```
double Square(double a)
{ double result;
  result = a*a;
  return result;
}
```

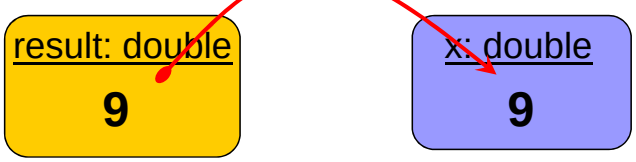
```
void main(void)
{
  double x;
  x=Square(3);
  cout << x;
}
```

→ x equal 9

In memory:

Copy the value

At the end of **`Square(x);`**



➔The value is copied in x

➔Remember that the local variable *result* is destroyed at the end of function *Square*()

1- C/C++ Syntax overview

- Standard lib
  - iostream (C++), Input Output stream
    - ***#include <iostream>*** and ***using namespace std;***
      - cout << something; // prints something on the screen
      - cin >> variable; // stops and waits a value from keyboard, this value is stored in *variable*
      - *cout << endl; // equivalent to "\n": new line*
  - Math
    - ***#include <math.h>*** and compile argument : -lm
      - sqrt(x): return the square root of x
      - sin(x), cos(x),… : trigonometric function
  - *Objects and classes ? later!*

2- Sources

- The *main* function
  - The main function is the main program
  - The C++ environment calls main
    - your program must never call main
  - Parameters:
    - The main function must return type int, the value return by main is passed to the host environment (0⇔EXIT_SUCCESS)
    - It can be called with no arguments or with two arguments

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

  - When the main function ends (or returns), all static objects are destroyed in the reverse order of their construction, and the program terminates.

2- Sources

- Include directive
  - The #include directive includes the entire contents of a standard header or source file.
  - Two forms:

```
#include "myheader.h"
#include <standardheader.h>
```

  - The only difference between both expressions is the places (directories) where the compiler is going to look for the file
    - when the file name is specified between double-quotes, the file is searched first in the same directory that includes the file containing the directive. (if it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files).
    - When the file name is enclosed between angle-brackets <>, the file is searched directly where the compiler is configured to look for the standard header files.

These default directories are specified either by configuration files or by environment variables

2- Sources

- Preprocessor directives
  - The preprocessing step occurs before the main compilation step
  - Preprocessor directives
    - To define and undefine macros,
    - To establish regions of conditional compilation,
    - To include other source files,
    - To control the compilation process
  - Preprocessor directives obey different syntax rules from the rest of the language
  - C++ understands *only* 12 directives

2- Sources

- Preprocessor directives

#define			
#elif	#else	#endif	#error
#if	#ifdef	#ifndef	#include
#line			
#pragma			
#undef			

Examples

```
#ifndef _MY_HEADER_FILE_
  #define _MY_HEADER_FILE_

  // all functions declaration
  // (or class definition...)

#endif
```

```
#ifndef __cplusplus
  #error A C++ compiler is required
#endif
```

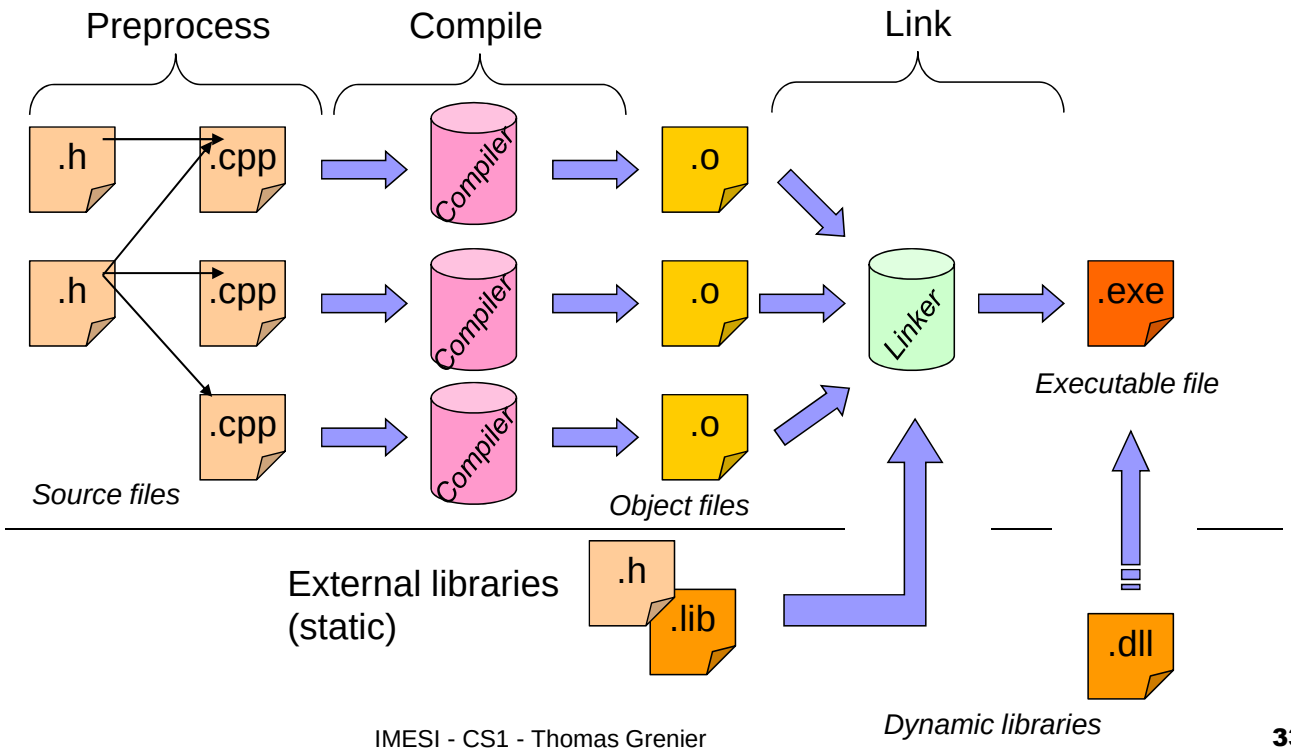
```
#define getmax(a,b) ((a)>(b)?(a):(b))
```


2- Sources

- Standard extension of files
  - Header files: .h .hpp .hxx
    - Contains the functions (and classes) declarations
    - Contains the classes definitions
  - Source files: .cpp .cxx
    - Contains the functions definition
  - Template source files (.txx)
    - Contains the definition of template functions

3- From source files to executable

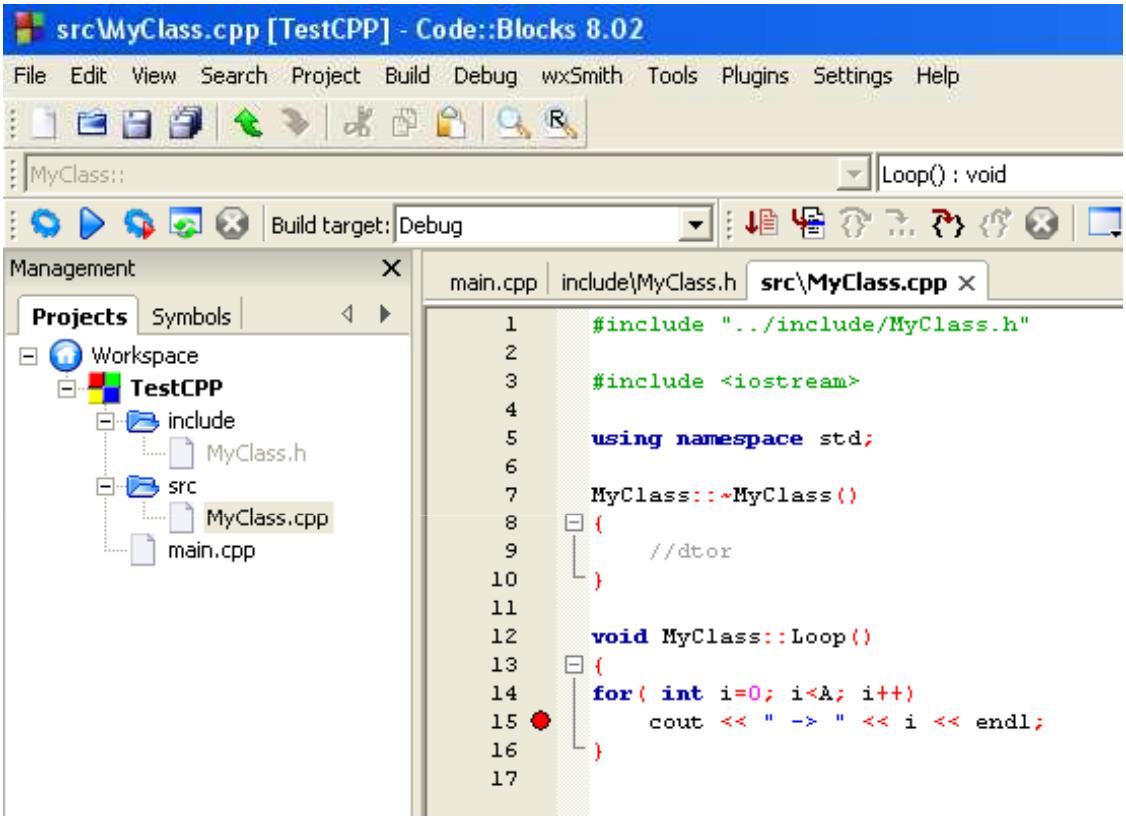
- Compilation and linkage



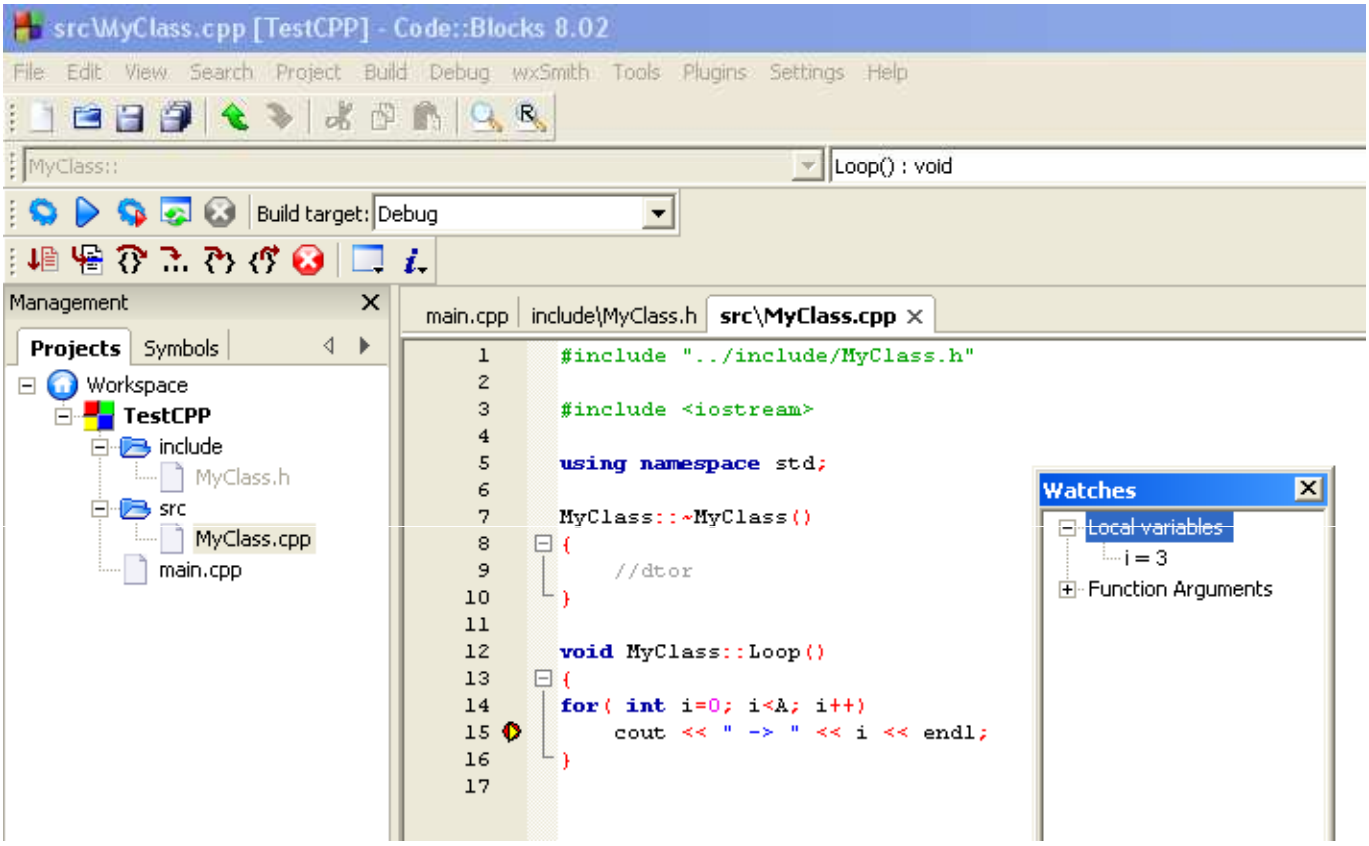
3- From source files to executable

- Debugging
  - To execute a program step by step, watch variables values, …
    - → understand what is going wrong
    - → or if all is alright! (memory, loops, function calls…)
  - Tools : debugger
    - Breakpoints
    - Watches variables

3- From source files to executable



3- From source files to executable



4- Examples

```cpp
// File: main.cpp
#include <iostream>

#include "myfunction.h"

using namespace std;

int main()
{
  int a;
  cout << "Entrer une valeur" << endl;
  cin >> a;

  Print(a);

  return 0;
}
```

```cpp
//File: myfunction.h
#ifndef _MY_FUNCTION_
#define _MY_FUNCTION_

void Print( int a);

#endif
```

```cpp
//File: myfunction.cpp
#include "myfunction.h"
#include <iostream>

using namespace std;
void Print(int a)
{
int i;
for( i=0; i<a; i++)
  cout<< "Hello !" << i << endl;
}
```


4- Examples

```cpp
// standard macro names
#include <iostream>
using namespace std;
int main()
{
  cout << "This is the line number " << __LINE__;
  cout << " of file " << __FILE__ << ".\n";
  cout << "Its compilation began on the" << __DATE__;
  cout << " at " << __TIME__ << ".\n";
  cout << "The compiler gives a __cplusplus value of " << __cplusplus;
  return 0;
}
```

4- Examples

```
void quickSort(int tableau[],
               int p, int r)
{
int q;
if(p<r)
  {
  q = partitionner(tableau, p, r);
  quickSort(tableau, p, q);
  quickSort(tableau, q+1, r);
  }
}
```

```
int partitionner(int tableau[],
                 int p, int r)
{
int pivot = tableau[p];
int i = p-1, j = r+1;
int temp;
while(1)
  {
  do
    j--;
  while(tableau[j] > pivot);

  do
    i++;
  while(tableau[i] < pivot);
  if(i<j)
    {
    temp = tableau[i];
    tableau[i] = tableau[j];
    tableau[j] = temp;
    }
  else
    return j;
  }
return j;
}
```

INSA LYON

International Master in Embedded Systems and Medical Image Engineering

IMESI

Computer Science 1

Sorting with *linked list*

M.Sc IMESI

Summary

Sorting with *linked list*

- Element insertion/removal in an array
  - Computational time
- How to insert quickly (O(1)) a value in an array?
  - By using a *list*
  - ➔ How to create a list ? How to use it?
- Exercises:
  - Playing with list
  - Sorting algorithm with linked list
    - Efficiency of insertion sort with linked list

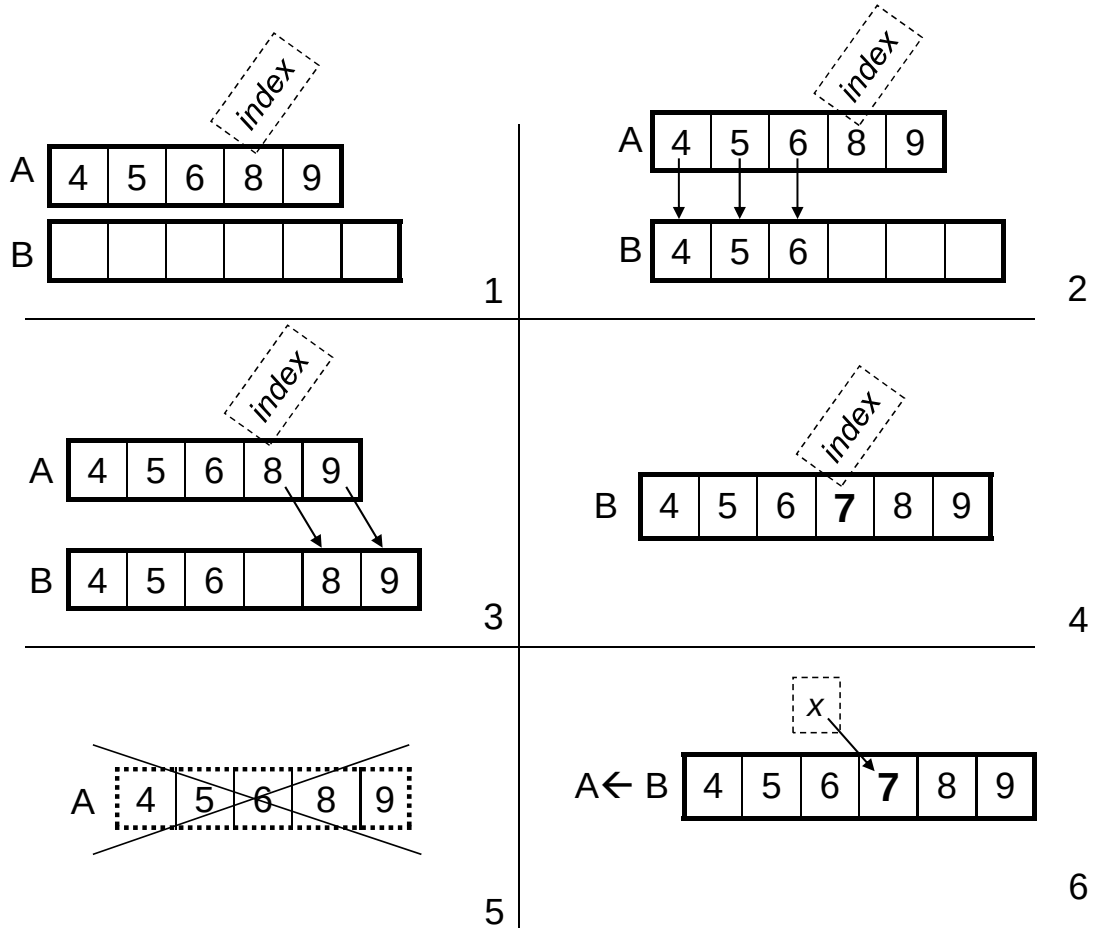
1- Inserting/removing array elements

- Adding a new element
  - Pseudocode

```
INSERTION(A, x, index)
    ▷ Create a new array B of size Length[A] + 1.
1   CREATE(B, Length[A]+1)
    ▷ Copy elements from A to B
2   for i ← 1 to index
3       do B[i] ← A[i]
    ▷ Copy elements from A to B with a shift
4   for i ← index to Length[A]
5       do B[i + 1] ← A[i]
6   B[index] ← x
7   DESTROY(A)
8   A ← B
```

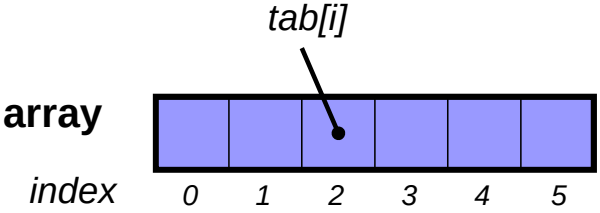
  - Asymptotical running time: O(n)

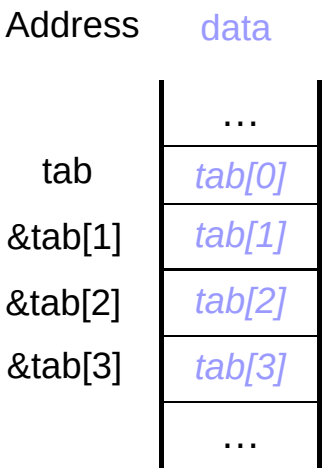
index=4
x=7



2- Linked list

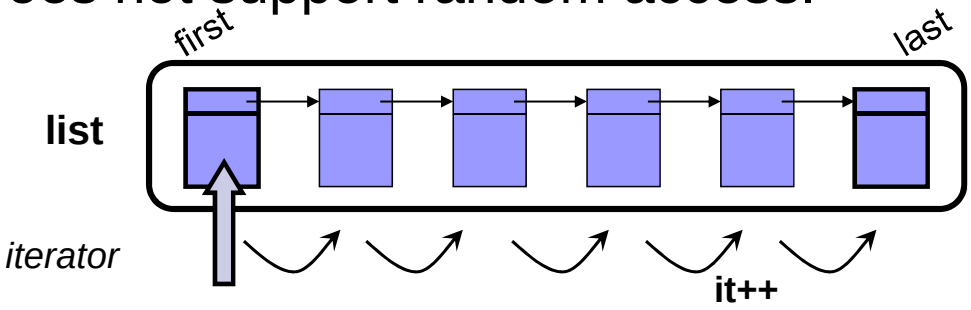
- Arrays
  - Are continuous sequence in memory
  - Support random access





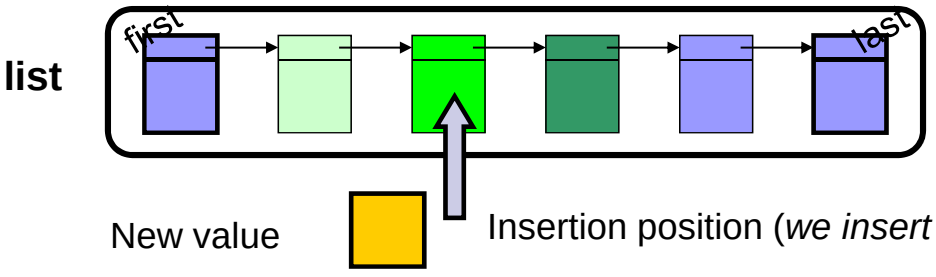
2- Linked list

- List
  - Is a sequence container that has constant performance when adding to or removing from any point in the container.
  - Supports bidirectional iterators,
    - you can use as many iterators as you need
  - Does not support random access.



2- Linked list

- (after)Insertion of a new value in a list

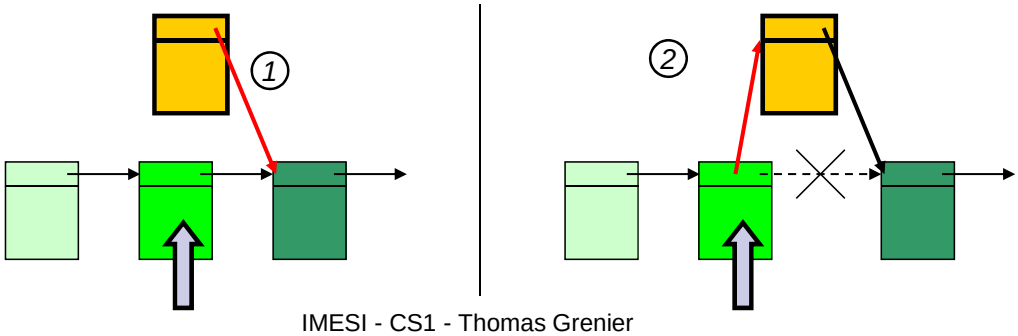


New value

Insertion position (*we insert **after** this position…*)

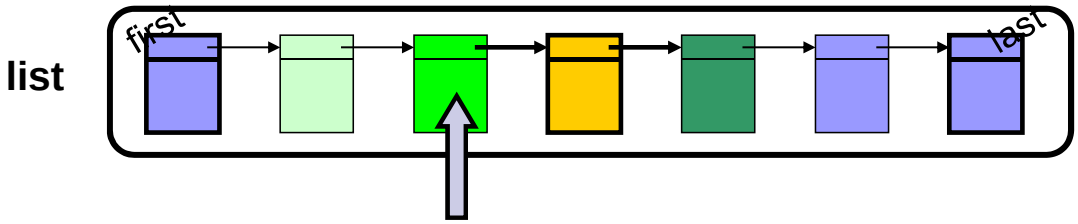
New value

New *element*



2- Linked list

- After-Insertion of a new value in a list



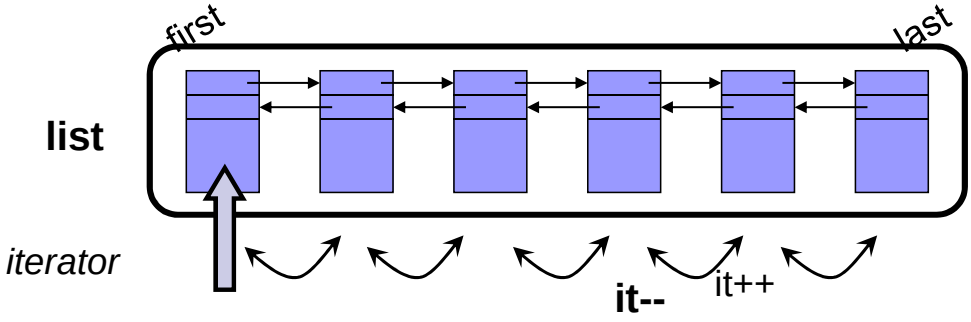
  - Asymptotical running time: O(1)
    - (not depending of n!)

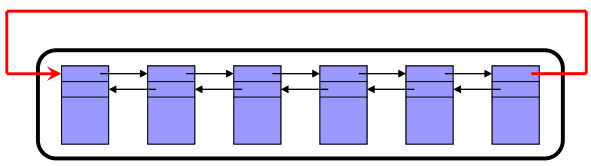
➔ What is the asymptotical running time of the before-insertion algorithm ?

  - (the new element is inserted before the iterator position)

2- Linked list

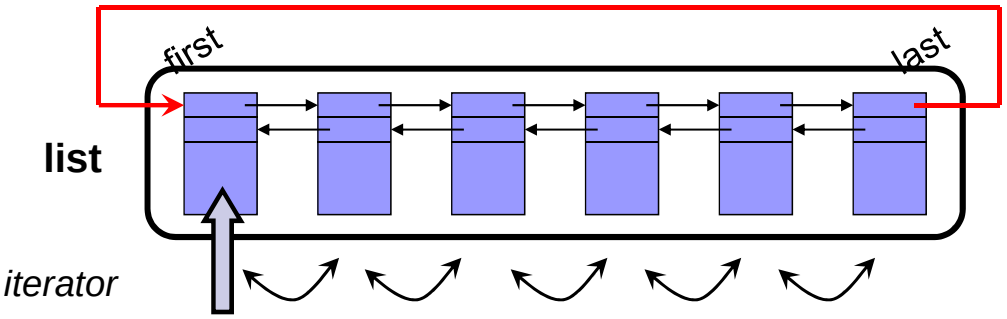
- Doubly linked list (*the obvious choice!*)

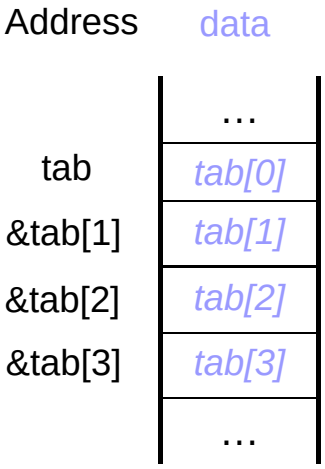


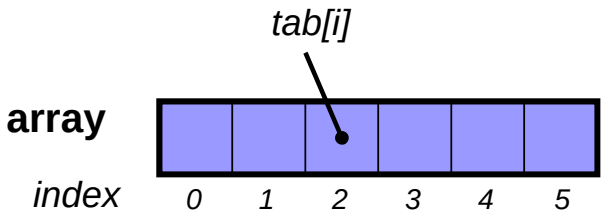


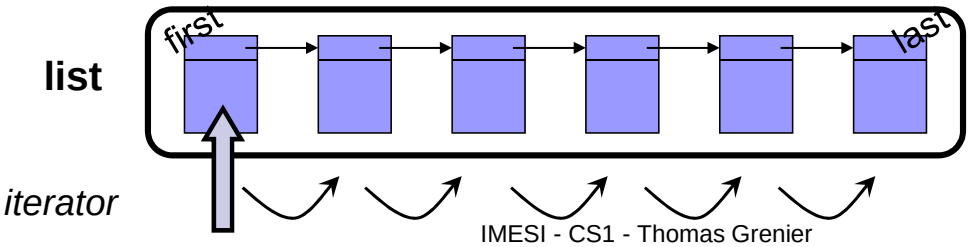
  - Many versions
    - Chained list
  - Many applications
    - Stacks (lifo), queues (fifo)

figures



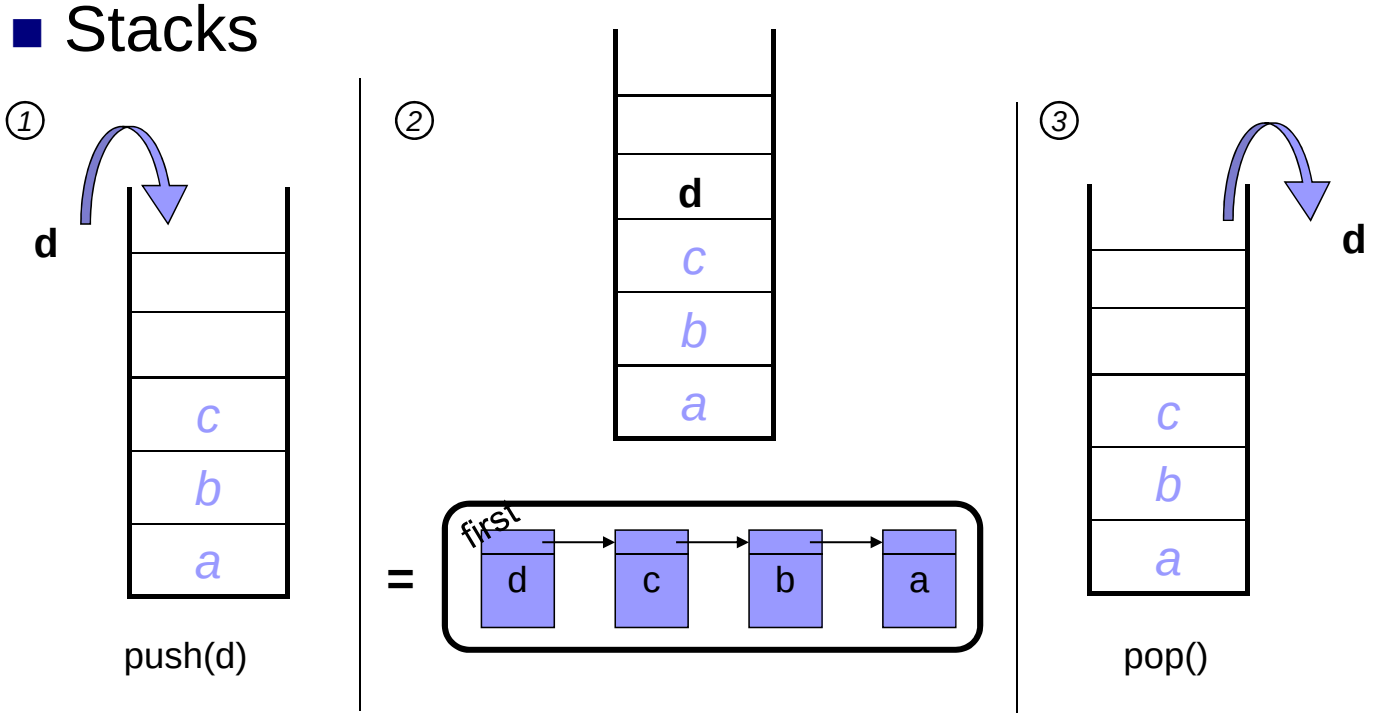






2- Linked list

- Stacks

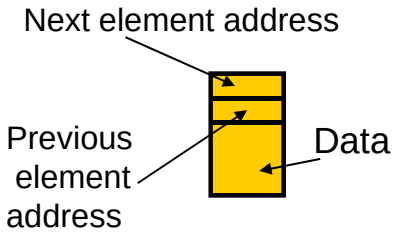


➔Push_front(b)
➔Add a new element in the front of the list

➔Pop_front()
➔Remove the first element of the list

2- Linked list

- How to create a list ?
  - We need **Element** that contains
    - The data
    - The addresses of next (and previous) element
  - We need **List** structure that contains
    - The number of elements (the size of the list)
    - Addresses of the first and the last elements
    - **Operations** like: insert, remove an element



2- Linked list

- Encapsulation of fields and methods

  ➔ Classes

UML

Element
-Data: double -Next: Element -Previous: Element
+GetData(): double +SetData(d: double): void +SetNext(n: Element): void +GetNext(): Element +SetPrevious(p: Element): void +GetPrevious(): Element +IsValid(): bool

List
-First: Element -Last: Element -Size: int
+Begin(): Element +End(): Element +Insert(a: double, pos: Element): void +InsertLast(a: double): void +Erase(pos: Element): double +Clear(): void +Empty(): bool +GetSize(): int <<create>>+List() <<destroy>>+List()

2- Linked list

- Encapsulation of fields and methods

➔ Classes

C++

```
#ifndef _ELEMENT_H
#define _ELEMENT_H

class Element
{
private:
  double Data;
  Element* Next;
  Element* Previous;
public:
  double GetData();
  void SetData(double d);
  void SetNext(Element* n);
  Element* GetNext();
  void SetPrevious(Element* p);
  Element* GetPrevious();
};

#endif  //_ELEMENT_H
```

```
#ifndef _LIST_H
#define _LIST_H

#include "Element.h"
class List
{
private:
  Element* First;
  Element* Last;
  int Size;
public:
  Element* Begin();
  Element* End();
  void Insert(double a, Element* pos);
  void List::InsertLast(double a);
  double Erase(Element* &pos);
  void Clear();
  bool Empty();
  int GetSize();
  List();
  ~List();
};

#endif  // LIST_H
```


2- Linked list

- Encapsulation of fields and methods

➔ Classes

Sources files?... later

C++

2- Linked list

- How to use it ? Example

```
List list;
list.Insert(1, list.Begin() );
list.Insert(2, list.Begin() );
list.Insert(3, list.Begin() );
list.Insert(4, list.Begin() );
list.InsertLast(0);

Element *it;//iterator
for( it=list.Begin(); it != 0; it=it->GetNext() )
  cout<< it->GetData() << " ";
cout << endl;   // 4 3 2 1 0;

it=list.Begin();
it=(*it).GetNext(); // <=> it->GetNext()
cout<< "removed value: " << list.Erase( it ) << endl;
   // 3
cout<< "removed value: " << list.Erase( it ) << endl;
   // 2
```


3- Exercises

- Write an algorithm verifying that a list is sorted
  - Efficiency of this solution
- Insertion sort using a doubly linked list
  - Write this algorithm
  - Give the efficiency of this algorithm
  - Are lists useless?

INSA LYON

International Master in Embedded Systems and Medical Image Engineering

IMESI

Computer Science 1

C++ Classes and UML

M.Sc IMESI

Summary

C++ Classes and UML

- Simple example: Class Complexe
  - What do we need to manipulate complex numbers?
  - C++: header and source files
  - Step by step example of use
  - Exercises
- Bases of OOP (UML and C++)
  - Fields and methods
    - Declaration, definition, visibility, multiplicity
    - Constructors and destructor
  - Inheritance
  - Operators overload

1- Simple Example: Class Complexe

- Complex numbers

such $z_1, z_2 \in \mathbb{C}$

Example : $z_1 = 1 + j$

$z_2 = 2 - 4j$

mathematics | *implementation*

$z_i = a_i + jb_i$

2 data are merged

C struct

Object Oriented Programming

z_1 and z_2 come from the same algebraic structure (complex)

z_1 and z_2 are two independent complex numbers

The main interest of complex numbers: they are a field (+,x)

$z_3 = z_1 + z_2 = 3 - 3j$

$z_4 = z_1 \times z_2 = 6 - 2j$

➔ Algebraic structure with dedicated functions

OOP definitions:

- z_1, z_2, z_3 and z_4 are **objects**
- $\mathbb{C}$ is a **class**

1- Simple Example: Class Complexe

- Complex numbers in C and in C++

C Language

Data

```
typedef struct
{ double Reel;
  double Imag;
} Complexe;
```

Functions

```
Complexe Somme(Complexe z1, Complexe z2)
{
Complexe s;
  s.Reel = z1.Reel + z2.Reel;
  s.Imag = z1.Imag + z2.Imag;
return s;
}
```

C++ Language

```
class Complexe
{
public:
  double Reel;
  double Imag;
```

Fields

constructor

```
  Complexe() {Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Methodes

1- Simple Example: Class Complexe

- Complex numbers in C++

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

```
#include <iostream>
…
…
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

Objects

Lazy programming

1- Simple Example: Class Complexe

- How does the `main` function work?

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

1) 3 *Complexe* objects (z1, z2, z3) are created
   Call of the appropriate object constructor

In memory :

z1	z2	z3
Reel = 0	Reel = 0	Reel = 0
Imag = 0	Imag = 0	Imag = 0

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```


1- Simple Example: Class Complexe

- How does the `main` function work?

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

1) 3 *Complexe* objects (z1, z2, z3) are created
   Call of the appropriate object constructor

2) Modification of z1 and z2 fields
   Direct access to values (`public`…)

In memory :

z1	z2	z3
Reel = 1	Reel = 2	Reel = 0
Imag = 1	Imag = -4	Imag = 0

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1- Simple Example: Class Complexe

- How does the `main` function work?

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1) 3 *Complexe* objects (z1, z2, z3) are created
   Call of the appropriate object constructor
2) Modification of z1 and z2 fields
   Direct access to values (`public`…)
3) The `Plus` function is called from z1 with the z2 parameter, the result is stored in z3

In memory :

1- Simple Example: Class Complexe

- How does the `main` function work?

3) The `Plus` function is called from z1 with the z2 parameter, the result is stored in z3

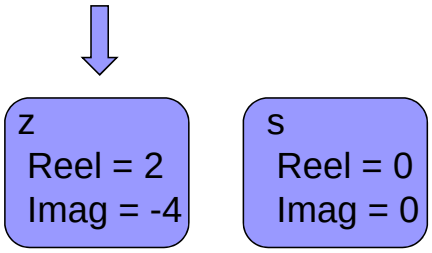
```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

```
z3 = z1.Plus(z2);
```

- In memory, when *Plus* is called

```
z3 = z1.Plus(z2);
```

- Local variables in *Plus* function



1- Simple Example: Class Complexe

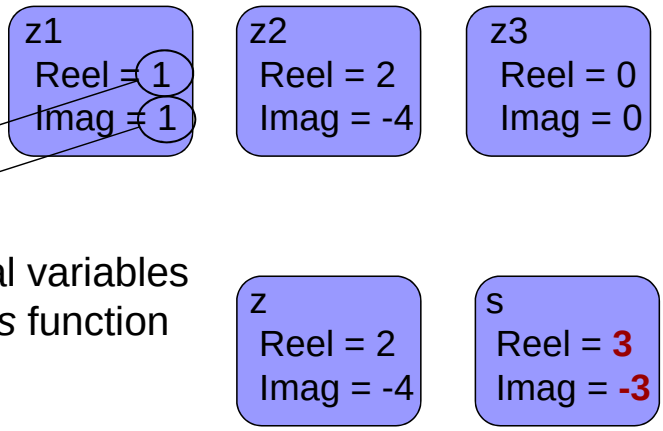
- How does the `main` function work?

3) The `Plus` function is called from z1 with the z2 parameter, the result is stored in z3

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

- In memory, when *Plus* is called

```
z3 = z1.Plus(z2);
```



- Local variables in *Plus* function

1- Simple Example: Class Complexe

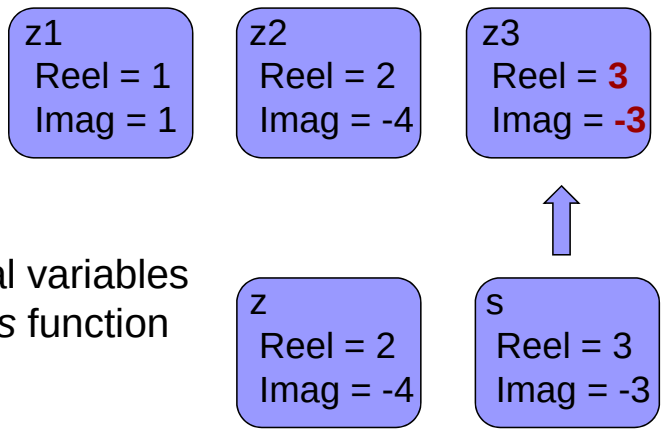
- How does the `main` function work?

3) The `Plus` function is called from z1 with the z2 parameter, the result is stored in z3

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

- In memory, when *Plus* is called

```
z3 = z1.Plus(z2);
```



- Local variables in *Plus* function

1- Simple Example: Class Complexe

- How does the `main` function work?

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
 return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + s.Imag;
     return s;
   }
};
```

1) 3 *Complexe* objects (z1, z2, z3) are created
   Call of the appropriate object constructor
2) Modification of z1 and z2 fields
   Direct access to values (`public`…)
3) The `Plus` function is called from `z1` with the `z2` parameter, the result is stored in z3
4) z3 is printed:

```
3
-3j
```

In memory :

z1	z2	z3
Reel = 1	Reel = 2	Reel = 3
Imag = 1	Imag = -4	Imag = -3

1- Simple Example: Class Complexe

- Exercise
  - Add the following operations in Complexe class:
    - absolute value (or *modulus* or *magnitude*)
    - conjugation
    - subtraction, multiplication, and division
  - Add the polar to cartesian conversion

Conversion from the polar form to the Cartesian form

$z = x + j.y$

$$x = r\cos\varphi$$
$$y = r\sin\varphi$$

Conversion from the Cartesian form to the polar form

$$r = \sqrt{x^2 + y^2}$$
$$\varphi = \arg(z) = \operatorname{atan2}(y, x)$$

•Addition: $(a+bi)+(c+di)=(a+c)+(b+d)i$

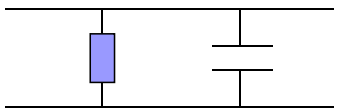
•Subtraction: $(a+bi)-(c+di)=(a-c)+(b-d)i$

•Multiplication: $(a+bi)(c+di)=ac+bci+adi+bdi^2=(ac-bd)+(bc+ad)i$

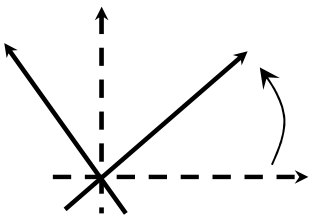
•Division: $\frac{(a+bi)}{(c+di)}=\left(\frac{ac+bd}{c^2+d^2}\right)+\left(\frac{bc-ad}{c^2+d^2}\right)i\,,$

1- Simple Example: Class Complexe

- Exercise with Complex numbers
  - Compute the complex impedance of a RC parallel circuit: R=50 Ohm, C=10nF, if
    - f= 50hz and f=[1, 10000]Hz



  - Give the new position of 2D vectors after a 45° rotation



Some answers

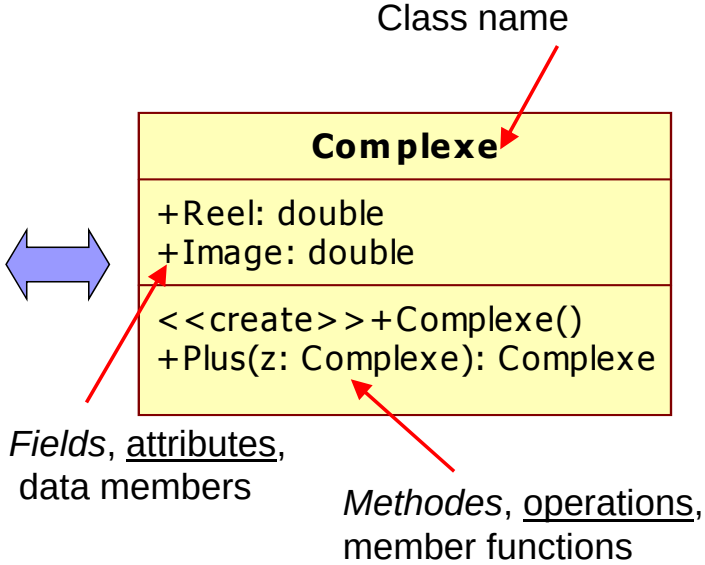
```
cout << " Question 1 : z for f= 50 Hz R // C"
  << " R=50 Ohms and C=10nF" << endl;
Complexe z = Complexe(1)
                /
  Complexe(1.0/50.0, 0.00000001 * 2 * 3.141592654 * 50) ;
cout << " z= ";
z.AffichePolarDegre();
cout << endl;

cout << endl;
Complexe w(1);
cout << " Question 2 : 45 degrees rotation of z=";
w.Affiche();
cout << endl;
Complexe r;

r.SetPolarCoordDegre( 1, 45 );
cout << endl;
(r * w).AffichePolarDegre();
```


2- OOP bases

- Complex numbers, C++ and UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implementation



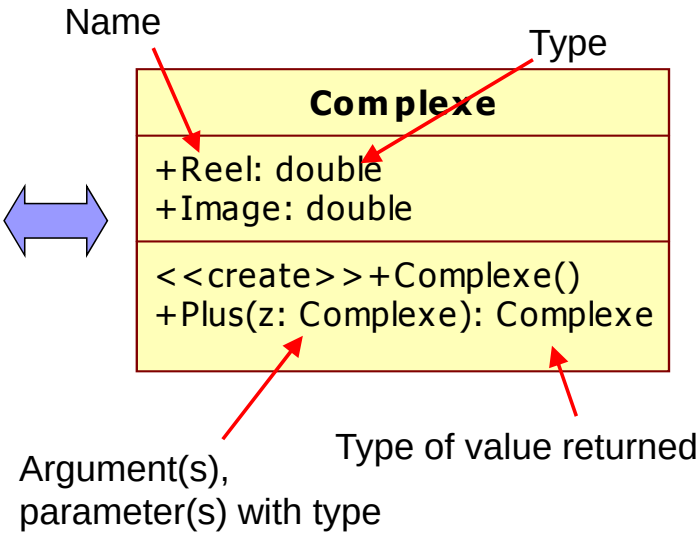
Modeling

2- OOP bases

- Complex numbers, C++ and UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implementation



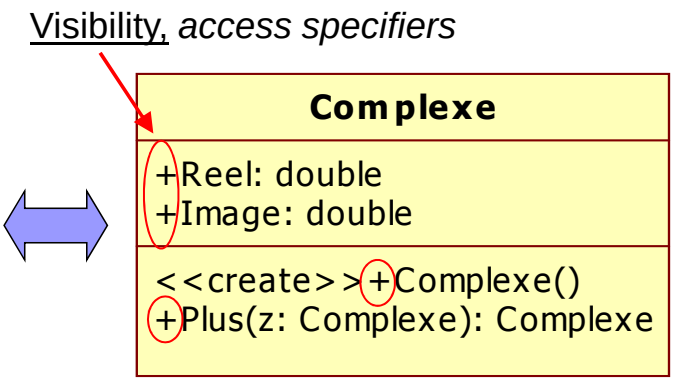
Modeling

2- OOP bases

- Complex numbers, C++ and UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implementation



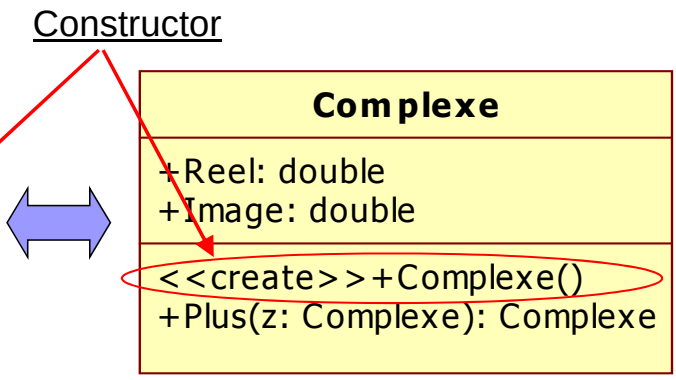
Modeling

2- OOP bases

- Complex numbers, C++ and UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

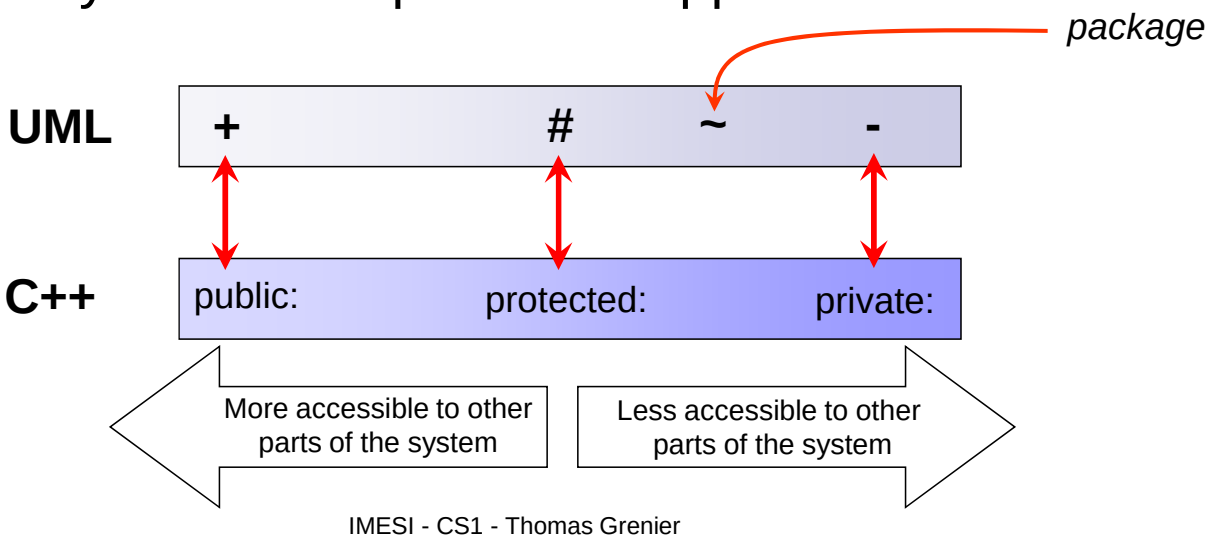
Implementation



Modeling

2- OOP bases – Visibility

- Attributes and operations access specifiers: restrict who can access a member
  - 4 UML-supported visibility types
  - Only 3 access specifiers supported in C++



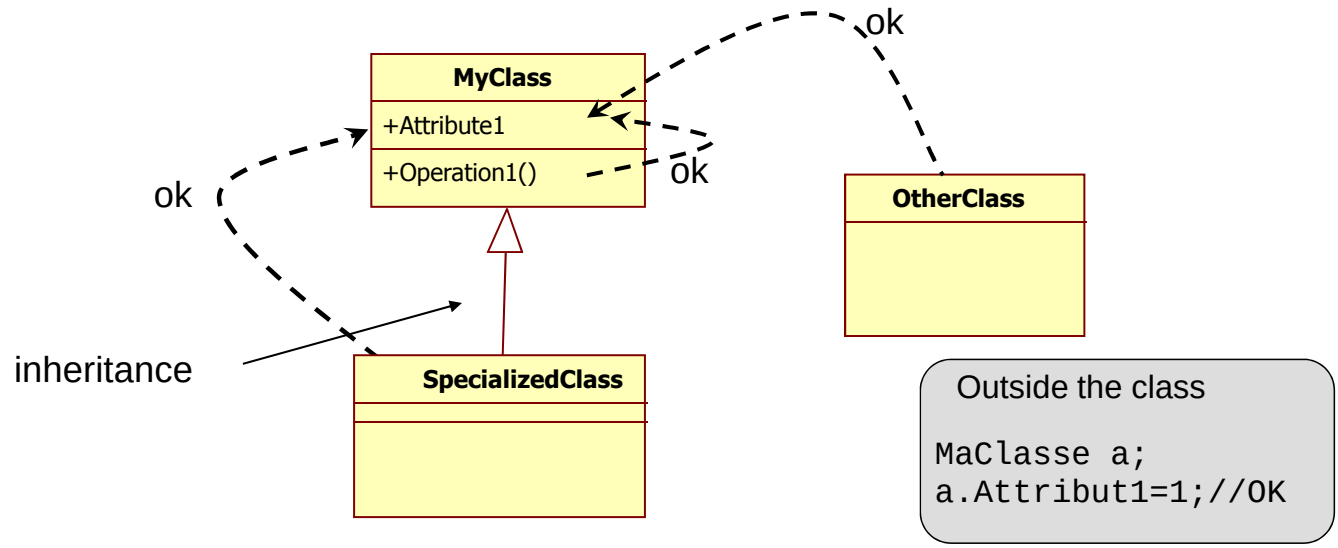
2- OOP bases – Visibility

- *public* visibility (+)
  - Anyone have access to a public member
- *protected* visibility (#)
  - Only the class, derived classes, and friends have access to protected members
- *private* visibility (-)
  - Only the class and friends have access to private members

➔ In a class definition, the default access for members and base classes is private

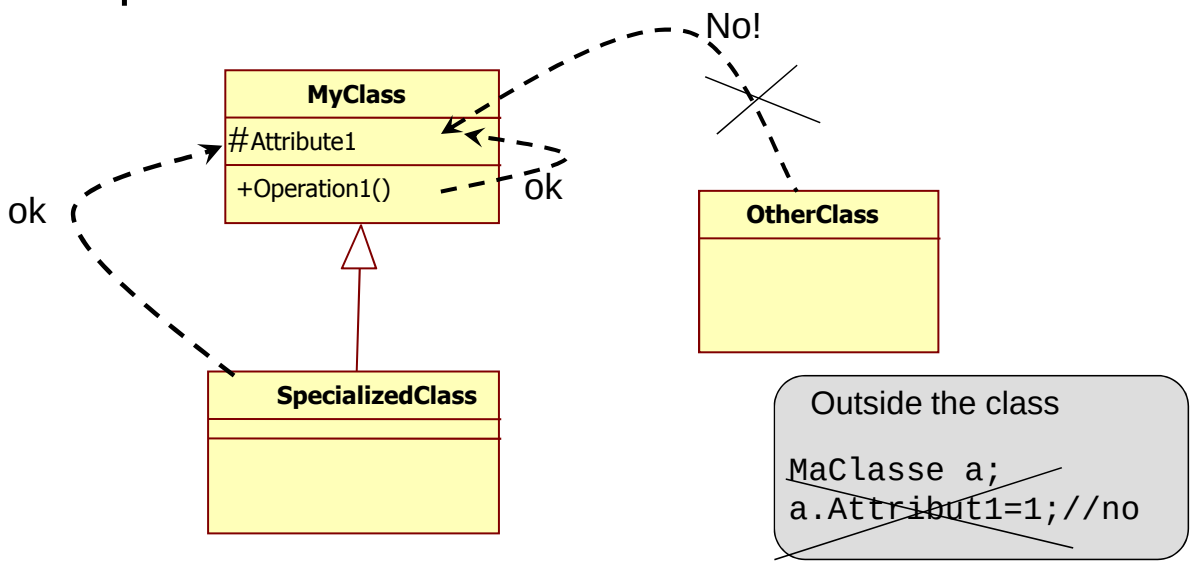
2- OOP bases – Visibility

- *public* visibility (+)
  - Anyone can access a public member



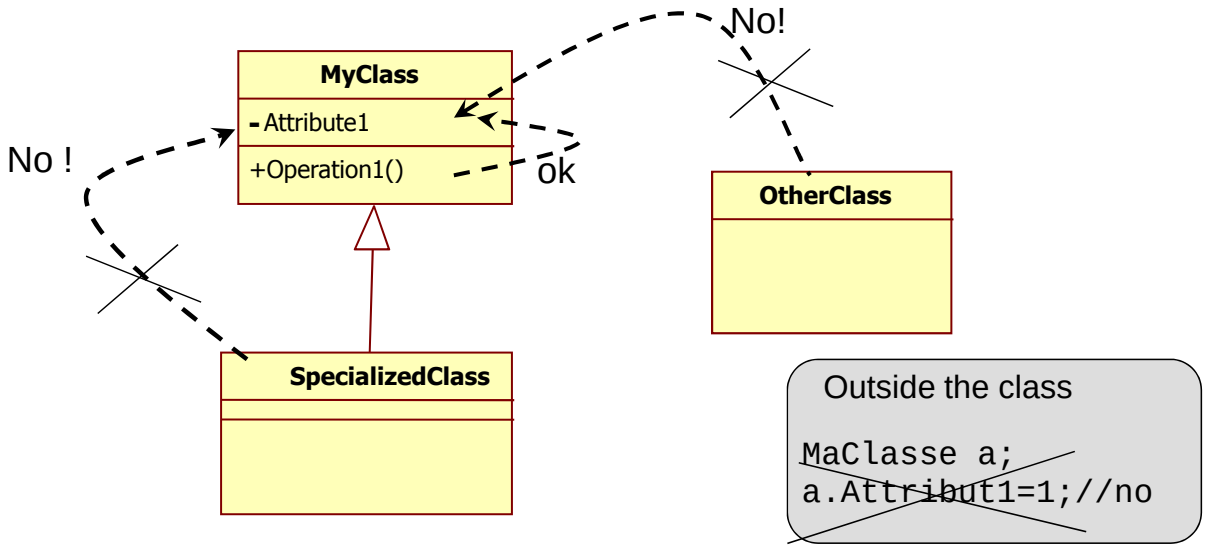
2- OOP bases – Visibility

- *protected* visibility (#)
  - Only the class, derived classes, and friends can access protected members



2- OOP bases – Visibility

- *private* visibility (-)
  - Only the class and friends can access private members



2- OOP bases – Visibility

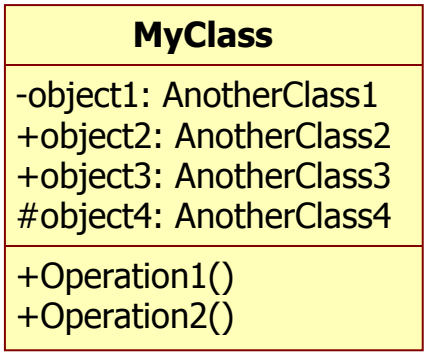
- To have public attributes or not to have public attributes? That is the question
  - the use of public attributes: opening a class's attributes to the rest of the system is like exposing your house to any person off the street without requiring him to check with you before entering. There is just as much potential for abuse.
- It's usually best to avoid public attributes
- Exceptions
  - when the attribute is a constant
  - when the attribute is not important for the class works and it is also not secret!

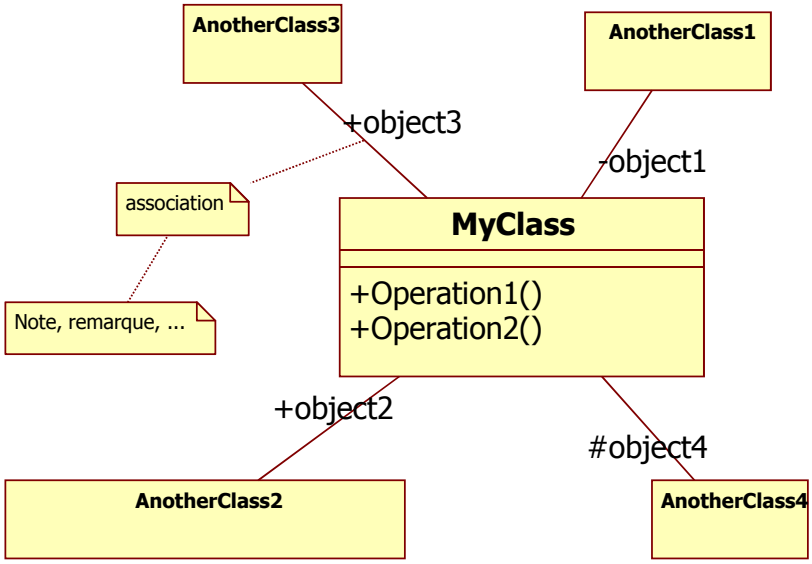
2- OOP bases – Name and Type

- Members: Name and Type
  - Name
    - can be any set of characters,
    - but two members in the same class can not have the same name.
    - ➔ Make sure that the name accurately describes what is being named
  - Type
    - The type of attribute can vary depending on how the class will be implemented in your system
    - it is usually either a class, such as string, or a primitive type, such as an int, double,…

2- OOP bases – Attributes

- *Inline Attributes* **or** *Attributes by association*





2- OOP bases – Multiplicity

- An attribute could represent any number of objects of its type
  - This is like declaring that an attribute is an array
- Multiplicity
  - Allows you to specify that an attribute actually represents a collection of objects, and it can be applied to both inline and attributes by association

2- OOP bases – Multiplicity

- The ***trackbacks***, comments, and authors attributes all represent collections of objects.
- The ***** at the end of the ***trackbacks*** and comments attributes specifies that they could contain any number of objects of the **TrackBack** and **Comment** class, respectively.
- The ***authors*** attribute is a little more constrained since it has specified that it contains between one and five authors.

2- OOP bases – Multiplicity

- The ***entries*** attribute (introduced using an association) has two multiplicity properties specified at either end of the association
  - A ***** at the **BlogEntry** class end of the association indicates that any number of **BlogEntry** objects will be stored in the ***entries*** attribute within the **BlogAccount** class.
  - The **1** specified at the other end of the association indicates that each **BlogEntry** object in the ***entries*** attribute is associated with one and only one **BlogAccount** object.

2- OOP bases – constructors

- C++ constructors
  - **A constructor is used to initialize an object**
  - A constructor's name is the same as the class name
  - A constructor cannot have a return type, and you cannot return a value
  - A constructor is executed when an object is created
    - Variable declaration or call of operator ***new***
    - You never call a constructor directly
  - 4 forms of constructor
    - User constructors (construct new object with some user parameters)
      - Complexe z1(2,3);
    - Copy constructor (constructs a new object from a previous one)
      - Complexe z2(z1);
    - Default constructor (constructor without parameters)
      - Complexe z3();
    - hidden constructor (when none constructor is declared in your class, the compiler adds one)
      - Element p;

2- OOP bases – constructors

- C++ constructors examples

2- OOP bases – destructor

- C++ destructor
  - **A destructor is used to finalize an object**
  - The name of a destructor is a tilde (~) followed by the class name
  - A destructor cannot have a return type, and you cannot return a value
  - A destructor is executed when an object is destroyed
    - For local variable : automatically called at the end of life (neither for pointers nor references)
    - Executed from a *delete* call
  - A destructor has none parameters

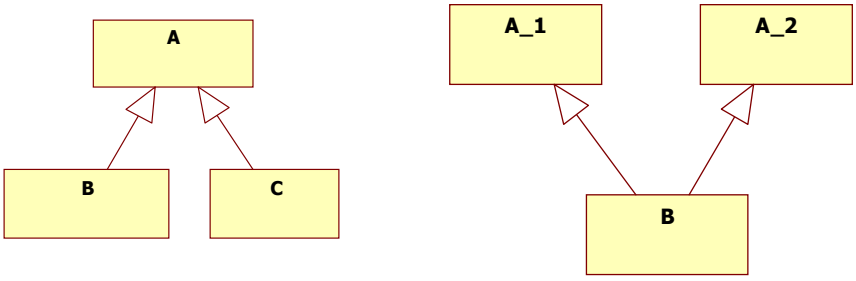
    ➔Only one destructor per class!
  - If you don't write the destructor, one (doing nothing) is added

2- OOP bases – constructors

- C++ destructors examples

2- OOP bases – Generalization (Inheritance)

- What is Generalization (or inheritance)?
  - A very important concept in object-oriented design
  - Generalization refers to the ability of one class (child class) to *inherit* the identical functionality of another class (super class, base class), and then the new class adds new functionalities to its base classes



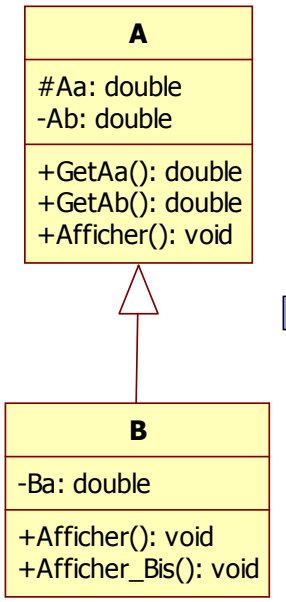
2- OOP bases – Generalization (Inheritance)

- Definitions
  - A class with at least one base class is said to be a *derived class*.
  - A derived class inherits all the data members and member functions of all of its base classes
  - A class's immediate base classes are called *direct base classes*.
    - Their base classes are *indirect base classes*.
    - The complete set of direct and indirect base classes is sometimes called the *ancestor classes*.
  - A class can inherit from zero or more *base classes*.

2- OOP bases – Generalization (Inheritance)

- Example

UML



C++

```
class A
{
protected:
  double Aa
private:
  double Ab;
public:
  A() {};
  double GetAa()
   {return Aa;}
  double GetAb()
   {return Ab;}
  void Afficher()
   { cout<< Aa << Ab;}
};
```

```
class B : public A
{
private:
  double Ba;
public:
  B() {};
  void Afficher()
   {
   cout << Aa << GetAb();
   cout << Ba << endl;
   }
 void Afficher_Bis()
   {
   A::Afficher();
   cout << Ba << endl;
   }
};
```


2- OOP bases – Generalization (Inheritance)

- Order of constructors execution
  - The base class constructor is executed before the constructor of the derived class
  - From the derived class constructor, you can pass some parameters to the base class constructor. If none parameter is passed, the default constructor of the base is executed

```
class A
{
double X;
public:
  A()
   { cout << "A:defaut" << endl;}
  A(double a)
   { X=a;
     cout << "A:General" << endl;}
};
```

```
class B: public A
{
public:
  B()
   { cout << "B:defaut" << endl;}
  B(double a):A(a)
   { cout << "B:General" << endl;}
};
```

```
void main()
{
B ob1; //X=?
B ob2(3);//X=3
}
```

Result:

A:defaut
B:defaut

2- OOP bases – Generalization (Inheritance)

- Order of constructors execution
  - The base class constructor is executed before the constructor of the derived class
  - From the derived class constructor, you can pass some parameters to the base class constructor. If none parameter is passed, the default constructor of the base is executed

```
class A
{
double X;
public:
  A()
   { cout << "A:defaut" << endl;}
  A(double a)
   { X=a;
     cout << "A:General" << endl;}
};
```

```
class B: public A
{
public:
  B()
   { cout << "B:defaut" << endl;}
  B(double a):A(a)
   { cout << "B:General" << endl;}
};
```

```
void main()
{
B ob1; //X=?
B ob2(3);//X=3
}
```

Result:
A:General
B:General

2- OOP bases – Generalization (Inheritance)

- Destructors order
  - The derived class destructor is executed before the base class destructor

```
class Array
{
double *Tab;
int Size;
public:
  Array(int size=1)
        :Size(size)
   { Tab = new double[Size]; }
  double GetElement(int i) const
   { return Tab[i]; }
  void SetElement(int i, double v)
   { Tab[i] = v; }
  int GetSize() const
   { return Size; }
  ~Array() // destructor
   { delete[] Tab;
  cout << "TD:Destructor"<< endl;
   }
};
```

```
class VecteurZero:
        public Array
{
public:
  VecteurZero(int size)
        :Array(size)
    {for( int i=0; i<GetSize(); i++)
        SetElement(i, 0);
    }
  ~VecteurZero() //useless destructor
   {cout << "VZ:Destructor"<< endl;}
};
```

```
void main()
{
VecteurZero a(10);
a.SetElement( 0, 1+a.GetElement(0));
}
```

2- OOP bases – operators overload

- Operator+
  - In class Complexe:

    Complexe operator+(const Complexe &z) { … }
  - In main function:

    Complexe z1, z2, z3;

    z3 = z1 + z2; // ⇔ z3 = z1.operator+(z2);

- operator-, operator/, operator*, …
  - idem

2- OOP bases – operators overload

- operator=
  - Automatically added by the compiler if not defined in your class
  - In class Complexe:

```
Complexe& operator=(const Complexe &z) {
  if( this != &z)
        {
        // copy the z object into the current object (this)
        }
  return *this;
  }
```

  - In main function

    Complexe z1, z2, z3;

    z3 = z1 = z2; // ⇔ z3.operator=(z1.operator=(z2));

2- OOP bases – operators overload

- operator[]
  - In class Array (of double):

```
double& Array::operator[](int index)
  { // first you can test the value of index
  return A[i];
  }
```

  - In main function

```
Array a;
a[0] = 1;
cout << a[0];
```


2- OOP bases – operators overload

- operator>, operator<, operator==, …
  - In class Complexe

```
bool Complexe::operator==(const Complexe &z) const
  { if( (Imag ==  z.Imag) && (Reel==z.Reel) )
        return true;
    else return false;
  }
```

  - In main function

```
Complexe z1(1,2), z2(1,3);
if(z1 == z2) …
```

INSA LYON

International Master in Embedded Systems and Medical Image Engineering

IMESI

Computer Science 1

Useful data structures

M.Sc IMESI

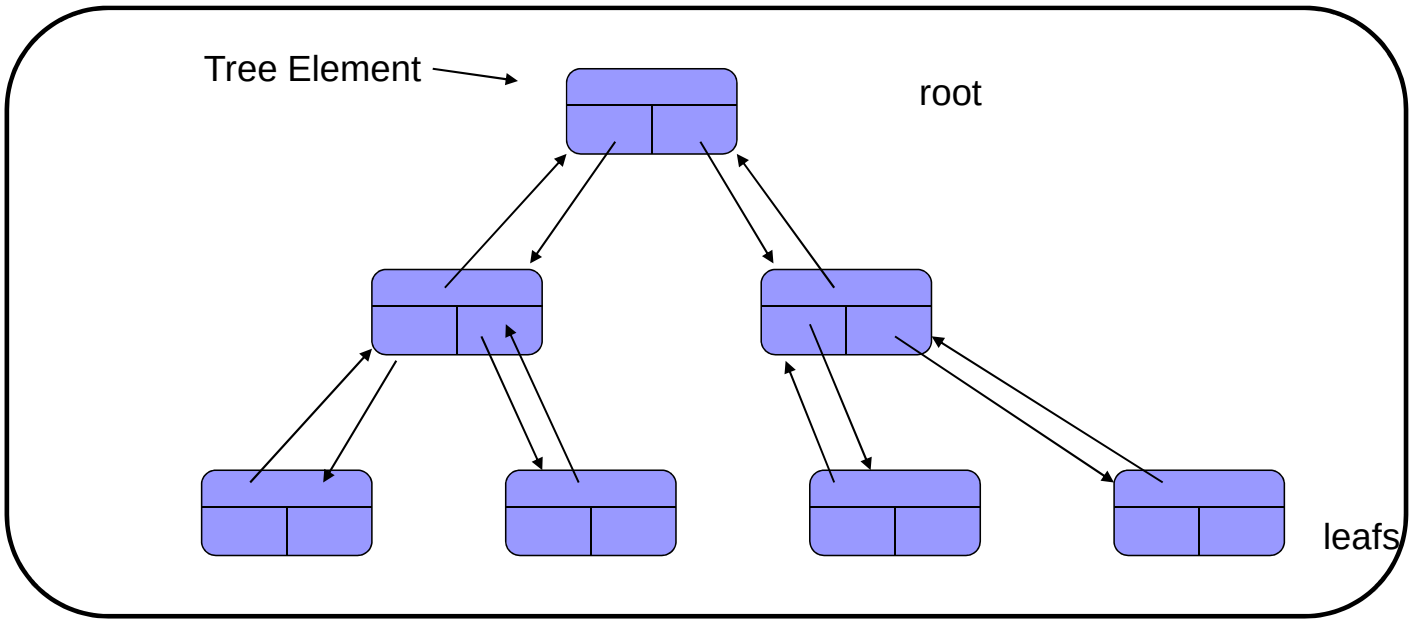
Summary

Useful data structures

- Array, Matrix, Volume
- Lists
  - Doubly linked lists
  - Chained lists
  - Queues, stacks
- Hash table
- Trees
- Graph… CS1-3

Tree, binary tree

Simple binary tree



Standard Containers of C++

- deque
  - A deque (double-ended queue) is a sequence container that supports fast insertions and deletions at the beginning and end of the container. Inserting or deleting at any other position is slow, but indexing to any item is fast. Items are not stored contiguously. The header is <deque>.
- list
  - A list is a sequence container that supports rapid insertion or deletion at any position but does not support random access. Items are not stored contiguously. The header is <list>.
- map , multimap
  - A map (or dictionary) is an associative container that stores pairs of keys and associated values. The keys determine the order of items in the container. map requires unique keys. multimap permits duplicate keys. The header for map and multimap is <map>.
- set, multiset
  - A set is an associative container that stores keys in ascending order. set requires unique keys. multiset permits duplicate keys. The header for set and multiset is <set>.
- vector
  - A vector is a sequence container that is like an array, except that it can grow as needed. Items can be rapidly added or removed only at the end. At other positions, inserting and deleting items is slower. Items are stored contiguously. The header is <vector>.

Standard adapters of C++

- Based on containers
- priority_queue
  - A priority queue is organized so that the largest element is always the first. You can push an item onto the queue, examine the first element, or remove the first element. The header is <queue>.
- queue
  - A queue is a sequence of elements that lets you add elements at one end and remove them at the other end. This organization is commonly known as FIFO (first-in, first-out). The header is <queue>.
- stack
  - A stack is a sequence that lets you add and remove elements only at one end. This organization is commonly known as LIFO (last-in, first-out). The header is <stack>.

Useful pseudo-containers of C++

basic_string, string, wstring

- Represent character strings. The string class templates meet almost all of the requirements of a sequence container, and you can use their iterators with the standard algorithms. Nonetheless, they fall short of meeting all the requirements of a container, such as lacking front and back member functions. The header is <string>.

- valarray
  - Represents an array of numeric values optimized for computational efficiency. A valarray lacks iterators, and as part of the optimization, the compiler is free to make assumptions that prevent valarray from being used with the standard algorithms. The header is <valarray>.

INSA LYON

International Master in Embedded Systems and Medical Image Engineering

IMESI

Computer Science 1

Exercises

M.Sc IMESI

Tools…

- Write a C++ function that prints all elements of a given array.
- Write a C++ function verifying that a given array is sorted. This function returns true only if the array is sorted.
- Write a C++ function that randomly initializes all elements of an array.

Shell Sort

- Divide and conquer approach (without recursive calls), *D.L. Shell 1959*
  1. Shell sort divides a long list into several smaller lists
  2. It sorts each smaller list by using an algorithm such as insertion sort or bubble sort
  3. It combines all the smaller lists into a large list
  4. It divides the new large list into several smaller lists again, but into smaller lists than in step 1
  5. It repeats steps 2 through 4 (if necessary) until a single sorted list remains

Shell Sort…

- The idea of Shell sort is the following:
  - Divide the data sequence in a two-dimensional array
  - Sort the columns of the array (the effect is that the data sequence is partially sorted)
  - The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns.
  - In the last step, the array consists of only one column.

Shell Sort

- Illustrate the shell sort algorithm on the array {5,6,8,2,1}. At the first iteration, the smaller lists have at most 2 elements
- Write the shell sort algorithm
- Try to give the efficiency of your algorithm (worst case analysis)
- Prove the correctness of shell-sort algorithm using the loop invariant:
  - Each sub array is sorted
- Write and test your algorithm in C++


```
void shellsort (int[] a, int n)
{
int i, j, k, h, v;
int[] cols = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,
  1968, 861, 336, 112, 48, 21, 7, 3, 1}
  for (k=0; k<16; k++)
    {
    h=cols[k];
    for (i=h; i<n; i++)
      {
      v=a[i];
      j=i;
      while (j>=h && a[j-h]>v)
        {
        a[j]=a[j-h];
        j=j-h;
        }
      a[j]=v;
      }
    }
}
```

http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/shell/shellen.htm

Searching algorithms

- **Sequential search algorithm**
  - A sequential search can start at the beginning or end of a list. It then proceeds to examine every item in the list until it finds the one item that it's searching for. Then it stops.
- ➔ Write a sequential search algorithm
- ➔ Give it efficiency (worst case analysis)
- The list is sorted… use a **binary search algorithm** that:
  - Cuts the list in half
  - Examines the number in the middle of the list
  - Determines if the searched value lies in the left or in the right half of the list
  - Repeats these steps on the right (correct) side of the list until it finds what's looking for
- ➔ Give the efficiency of binary search algorithm
- ➔ Write this algorithm in C++