Algorithmic and programming (Second part)

L1, S2

February 2018

Vincent Rodin
Université de Bretagne Occidentale
vincent.rodin@univ-brest.fr
http://labsticc.univ-brest.fr/~rodin/FTP/Enseignements/L1/AlgoEtProg

Algorithmic et programming Outline

Licence 1, second half year

- ▷ Writing conventions
- ▷ Procedures and functions
- ▷ Data structures and array
- ▷ Algorithm complexity
- ▷ Sorting algorithms
- ▷ Queue and stack ...

... towards modular programming

Writing conventions of C programs

Explicit the implicit
{ – }
Indentation
Variables: declaration, lifetime, visibility (shadowing)
Expressions and (–)
Comments

Explicit the implicit

All that is explicit can help :

- ▷ to obtain a more robust code
- ▷ to reduce the number of syntax errors
- ▷ to obtain a more readable code (for yourself and the others)

And thus...

"Explicit the implicit"

... even if it is longer to write !

{ – } (1)

In order to reduce the number of syntax errors and errors during the execution, always write { – } even if it is not necessary !

▷ **In conditional statement** `if`

```
#include <stdio.h>

int main(void)
{
 int a = 0;

 if (a == 0)
 {
   printf("It is zero\n");
 }
 else
 {
  printf("It is not zero\n");
 }
 return 0;
}
```

It allows to delimit the blocks of statements (blocks scope)

{ – } (2)

▷ **In loop statement** `while`

```
#include <stdio.h>

int main(void)
{
 int i;

 i=0;
 while (i<=9)
 {
  printf("%d\n",i); /* 0 => 9 */
  i = i + 1;
 }

 printf("%d\n",i); /* 10 */

 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 int i;

 i=9;
 while (i>=0)
 {
  printf("%d\n",i); /* 9 => 0 */
  i = i - 1;
 }

 printf("%d\n",i); /* -1 */

 return 0;
}
```


{ – } (3)

▷ **In loop statement** `do ... while`

```
#include <stdio.h>

int main(void)
{
 int i;

 i=0;
 do
 {
  printf("%d\n",i); /* 0 => 9 */
  i = i + 1;
 } while (i<=9);

 printf("%d\n",i); /* 10 */

 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 int i;

 i=9;
 do
 {
  printf("%d\n",i); /* 9 => 0 */
  i = i - 1;
 } while (i>=0);

 printf("%d\n",i); /* -1 */

 return 0;
}
```


{ – } (4)

▷ **In loop statement** `for`

```
#include <stdio.h>

int main(void)
{
 int i;

 for(i=0; i<=9; i = i + 1)
 {
  printf("%d\n",i); /* 0 => 9*/
 }

 printf("%d\n",i);  /* 10 */

 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 int i;

 for(i=9; i>=0; i = i - 1)
 {
  printf("%d\n",i);  /* 9 => 0 */
 }

 printf("%d\n",i);   /* -1 */

 return 0;
}
```

{ – } (5)

▷ **In multi-conditional statement `switch`... only for integers**

```
#include <stdio.h>

int main(void)
{
 int n;
 printf("Give me one value (either 1 or 2):\n");
 scanf("%d",&n);
 switch(n)
 {
  case 1 : { printf("One\n");
             break;
           }
  case 2 : { printf("Two\n");
             break;
           }
  default: { printf("Neither one, nor two\n");
             break;
           }
 }
 return 0;
}
```


Indentation (1)

The indentation allows to obtain a more readable program...

```
/*  Compile with:
    cc page010_sansIndentation.c -o page010_sansIndentation -lm
*/
#include <stdio.h>
#include <math.h>    /* Because using sqrt */
int main(void)
{
double a, b, c, delta, x1, x2;
printf("Solving of ax2 + bx + c = 0\n");
printf("Give me the values of a, b and c :\n");
scanf("%lg",&a); scanf("%lg",&b); scanf("%lg",&c);
if (a==0.0) { printf("Error: a is equal to 0 !\n"); }
else { delta = b*b - 4*a*c; if (delta<0)
{ printf("No solution\n"); } else { if (delta>0.0)
{ x1 = (-b-sqrt(delta))/(2*a); x2 = (-b+sqrt(delta))/(2*a);
printf("Two solutions: %g et %g\n",x1,x2); } else
{ x1 = x2 = -b/(2*a); printf("One solution: %g\n",x1); } } } return 0; }
```

... It is not readable without a good indentation !

Indentation (2)

```
/*  Compile with:
    cc page011_avecIndentation.c -o page011_avecIndentation -lm
*/
#include <stdio.h>
#include <math.h>    /* Because using sqrt */
int main(void)
{
 double a, b, c, delta, x1, x2;
 printf("Solving of ax2 + bx + c = 0\n");
 printf("Give me the values of a, b and c :\n");
 scanf("%lg",&a); scanf("%lg",&b); scanf("%lg",&c);
 if (a==0.0) { printf("Error: a is equal to 0 !\n"); }
 else {
   delta = b*b - 4*a*c;
   if (delta<0) { printf("No solution\n"); }
   else {
     if (delta>0.0) {
       x1 = (-b-sqrt(delta))/(2*a);
       x2 = (-b+sqrt(delta))/(2*a);
       printf("Two solutions: %g et %g\n",x1,x2);
     }
     else {
       x1 = x2 = -b/(2*a);
       printf("One solution: %g\n",x1);
     }
   }
 }
 return 0;
}
```

More readable ?

Variable: declaration, lifetime, visibility (shadowing)

Declaration of variable: at the begining of a block scope { – }.

Warning :

- ▷ **The lifetime of a variable is linked to the block.**
- ▷ **A variable may shadow (hide) another one.**

```
#include <stdio.h>

int main(void)
{
 int a=10, i;

 i=0;
 while (i<=9)
 {
  int a=100;         /* Creation of variable a when entering in the block ... at each loop */

  printf("%d\n", i + a); /* 100 => 109 */
  i = i + 1;
 }

 printf("%d\n", i + a); /* 20 */

 return 0;
}
```

Expressions and (–) (1)

A couple of (and) allows to change the evaluation order of operators...

```
#include <stdio.h>

int main(void)
{
 printf("2+3*4=%d\n",2+3*4);     /* 14 */
 printf("(2+3)*4=%d\n",(2+3)*4); /* 20 */

 return 0;
}
```

You know that : + has a lower priority than ∗

⇒ According to the desired operation, it is necessary to add (and)

Expressions and (–) (2)

Priority : ◇ An operator more or less "attracts" his operands.
◇ In C language, there is 46 operators ... and 17 priority levels !

Remarks:

△ When **two** (or more) **operators** are involved in an **expression**
⇒ the priority between operators is applied...

△ When **two operators** have **the same priority**
⇒ the associativity between operators is applied...

Expressions and (–) (3)

Operator priorities :

◇ Higher to lower priorities
◇ Between two lines, the operators have the same priority

Operators	Symbol	Example	Associativity
Logical NOT (unary)	!	!(a==b)	←
multiplication	*	i * j	→
division	/	i / j	→
modulo (remainder)	%	i % j	→
addition	+	i + j	→
substraction	-	i - j	→
comparisons (relations)	< <= > >=	i < j	→
Equal to, Not equal to	== !=	i == j i != j	→
Logical AND	&&	a && b	→
Logical OR	||	a || b	→

Writing usefull comments

```
/* Usefull comments */
```

Essential for the life cycle of a program...

to improve it, to correct it

...by you or someone else

▷ **Example of useless comment :**

```
i = i + 1;   /* I increase the value of i by 1 */
```

▷ **Examples of usefull comments :**

```
 /* Description of the function, of the algorithm, ... */
int summationIntegers(int n)
{
   ...    /* Description of the main characteristics of the algorithm */
   ...    /* and of the treated special cases, ...                    */
}
```

Procedures and functions

Specification and implementation
Designer versus user
Structuration of algorithms: Divide and rule
Reutilisability of algorithms: Parameterize in order to reuse
Procedures ; Functions
C: declaration - definition
Specification
Implementation
Algorithm : Specification + Implementation
Parameter passing : by value, by address
Procedures and functions: how to choose ?
Example of function types
Procedures and functions: writing algorithms "in english"
Recursive functions

Specification and implementation

▷ **Specification of an algorithm** **What?**

The specification describes the function and the use of an algorithm

What the algorithm do.

The algorithm is seen here as a black box, the inner working is not known.

▷ **Implementation of an algorithm** **How?**

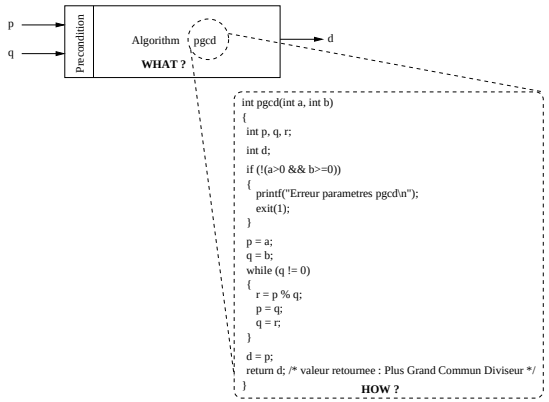
The implementation describes the inner working of the algorithm

How the algorithm do.

The implementation describes the sequence of instructions needed to solve the problem which is adressed.

Specification and implementation

The specification describes the function and the use of the algorithm



The implementation describes the inner working of the algorithm

Designer versus user

▷ **Designer**

The designer of an algorithm defines :
- the **interface** and
- the **implementation**

of the algorithm.

▷ **User**

The user of an algorithm don't need to know the implementation;
only the **interface** of the algorithm is usefull for him.

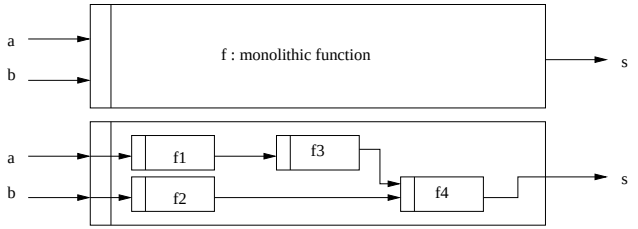
According to the specification of the algorithm,
the user **call** (use) the algorithm :
- as a **function** or
- as a **procedure**. (i.e. procedure : function without return value)

Structuration of algorithms (1)

"Divide and rule"

▷ **Structuration ("Diviser pour régner")**

The procedures and functions allow to decompose a complex program into a set of simpler sub-programs, which can also be decomposed into smaller parts, and so on.



Structuration of algorithms (2)

▷ **Problem**

How to structure an algorithm to make it understandable?

```
#include <stdio.h>                                  |   /* following..*//* Binary calculation of b     */
                                                    |   number=b;       /* same calculation as for a ! */
       /* Calculation of the binary operation      |   ...
             a AND b by calculating binary          |   /* Binary calculation of a AND b */
               representations of a and b  */       |
int main(void)                                      |   for(i=0;i<=nb_Bits-1;i=i+1)
{                                                   |   {
 int a,b,m;                                         |    if (a_Bits[i]+b_Bits[i]==2) { and_Bits[i]=1; }
 int i,nb_Bits=8;                                   |    else { and_Bits[i]=0; }
 int a_Bits[8], b_Bits[8], and_Bits[8];             |   }
 int number,remaining;                              |
                                                    |   /* Decimal calculation of a AND b */
 printf("Give me the values of a and b\n");         |
 scanf("%d",&a); scanf("%d",&b);                    |   number=0; m=1;
                                                    |   for(i=nb_Bits-1;i>=0;i=i-1)
 number=a;      /* Binary calculation of a */       |   {
 for(i=nb_Bits-1;i>=0;i=i-1)                        |    number=number+(m*and_Bits[i]);
 {                                                  |    m=m*2;
  remaining=number%2;                               |   }
     number=number/2;                               |   printf("a AND b: %d\n",number);
  a_Bits[i]=remaining;                              |   return 0;
 }                                                  |  }
```

A bit complicated?

Structuration of algorithms (3)

▷ **Response**

Use functions and procedures!

```
#include <stdio.h>     /* Calculation of the binary operation a AND b by */
                       /* calculating binary representations of a and b  */

  /* It misses here the code of the functions: convBinary, andBinary and convDecimal */

int main(void)
{
 int a, b, number;
 int a_Bits[8], b_Bits[8], and_Bits[8];

 printf("Give me the values of a and b\n");
 scanf("%d",&a); scanf("%d",&b);

 convBinary(a,a_Bits);              /* Function call: binary calculation of a        */
 convBinary(b,b_Bits);              /* Function call: binary calculation of b        */
 andBinary(a_Bits,b_Bits,and_Bits); /* Function call: binary calculation of a AND b  */
 number = convDecimal(and_Bits);    /* Function call: decimal calculation of a AND b */

 printf("a Et b: %d\n",number);
 return 0;
}
```

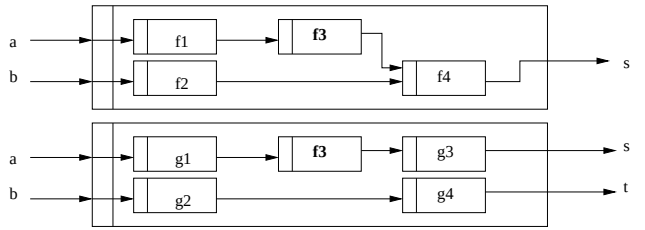
It is much simpler, easier...

Reutilisability of algorithms (1)

"Parameterize in order to reuse"

▷ **Reutilisability**

The functions and procedures are reusable : if we have a function which is able to perform a particular algorithm, we can re-use it in our programs. We do not have to rewrite the algorithm each time that we use it by calling the function.



Reutilisability of algorithms (2)

▷ **Problem**

how to reuse an already existing algorithm without rewrite it?

```
#include <stdio.h>          |     /* following... */
                            |
int main(void)              |     n = 5;
{                           |     r = 1;
 int n,r,i;                 |     for(i=1;i<=n;i=i+1)
                            |     {
 n = 3;                     |      r = r*i;
 r = 1;                     |     }
 for(i=1;i<=n;i=i+1)        |
 {                          |     printf("%d!=%d\n",n,r);
  r = r*i;                  |
 }                          |     return 0;
                            |    }
 printf("%d!=%d\n",n,r);    |
                            |
```


Reutilisability of algorithms (3)

▷ **Response**

Encapsulate the code into functions or procedures.

```
#include <stdio.h>          |
                            |    int main(void)   /* following... */
int factorial(int n)        |    {
{                           |     int n, r;
 int r, i;                  |
                            |     n = 3;
 r=1;                       |     r = factorial(n);
 for(i=1;i<=n;i=i+1)        |     printf("%d!=%d\n",n,r);
 {                          |
  r = r*i;                  |     n = 5;
 }                          |     r = factorial(n);
                            |     printf("%d!=%d\n",n,r);
 return r;                  |
}                           |     return 0;
                            |    }
```


Procedure

▷ **Definition:** A procedure is a named and parametrized block of instructions which **does not return** a value.

▷ **Example**

Type : `void printInt(int val); /* waits an int and returns nothing (void) */`

Code :
```
void printInt(int val)
{
 printf("%d",val);
}
```

Call :
```
#include <stdio.h>                          /* Because using printf...  */
void printInt(int val)
{
 printf("%d",val);
}
int main(void)
{
 int i = 17;
 printInt(i);
 return 0;
}
```


Functions

▷ **Definition:** A function is a named and parametrized block of instructions which **returns** a value.

▷ **Example**

Type : `double sin(double angle); /* waits an angle (double) and returns its sine value (double) */`

Code : Included in the mathematical library $\Longrightarrow$ `-lm` (used by the link editor)

Call :
```
#include <stdio.h>                          /* Because using printf...  */
#include <math.h>          /* Because using double sin(double angle); */
int main(void)
{
 double angle = 3.14, s;
 s = sin(angle);              /* The value can be assigned, printed, used...*/
 printf("The sine value is: %g\n",s);
 return 0;
}
```

C : declaration – definition

In C, usually, we only talk about function ...
... procedure ↔ function with no return value: `void`

Elements of C vocabulary:

▷ **The declaration of a function : the Type !**
⟹ characteristics of the function for its use ≡ header
⟹ Example : `void printInt(int val);`
Usage : `printInt(i);` with `i` an `int`

▷ **The definition of a function : the Code !**
⟹ Example :
```
void printInt(int val)
{
 printf("%d",val);
}
```


Specification of an algorithm

▷ **Name :** a sufficiently explicit identifier

▷ **Description :** A sentence which describes what the algorithm performs

▷ **Parameters :** The list of the input/output parameters of the algorithm

▷ **Preconditions :** A list of boolean expressions that specify the conditions for the application of the algorithm

▷ **Call :** Some examples of the use of the algorithm with the expected results

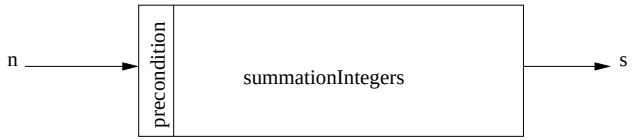
Example of the specification of an algorithm (1)

▷ **Name :** summationIntegers

▷ **Description :** computes the sum s of the n first positive integers :

$$s = 1 + 2 + 3 + ... + n = \sum_{i=1}^{i=n} i$$

▷ **Input/Output parameters :**



▷ **Precondition :** $n \in N^*$

Example of the specification of an algorithm (2)

▷ **Call :** `s = summationIntegers(n)`

Examples :
`s = summationIntegers(1)` ⟶ 1
`s = summationIntegers(5)` ⟶ 15
`s = summationIntegers(57)` ⟶ 1653
...
`s = summationIntegers(n)` ⟶ $\frac{n.(n+1)}{2}$

Specification of an algorithm : the steps

The 5 steps of the specification of an algorithm

◇ To give an explicit name to the algorithm ;

◇ To describe in a sentence what the algorithm performs;

◇ To define the input/output parameters of the algorithm ;

◇ To specify the preconditions on the input parameters ;

◇ To give some examples of the use and the expected results.

Implementation of an algorithm

▷ **Writing of the header and the parameters**

▷ **Indication of a return value (only for function)**

▷ **Writing of the preconditions**

▷ **Definition auxiliary variables (if necessary)**

▷ **Writing of the algorithm**

▷ **Tests of the algorithm**

Without forgetting : comments and indentations !...
...and no more than 80 characters per lines.

Implementation and readability

Auxiliary variables

Examples : ◇ `double delta = b*b - 4*a*c;`
◇ `int s = 0;`

▷ **Readability of an algorithm**

⟹ Define and use auxiliary variables...
...to increase the readability of an algorithm !

▷ **Storage of intermediate calculations**

⟹ Define and use auxiliary variables...
...to avoid making the same calculation several times !

Example of implementation : header of the summation of the first integers

```
int summationIntegers(int n)
{









 /* ... */

 /* Function : a return value is needed */
}
```

Example of implementation : return value of the FUNCTION summationIntegers

```
int summationIntegers(int n)
{
 int s;









 /* ... */

 return s; /* Return value of the function */
}
```


Example d'implementation : precondition of the function summationIntegers

```
int summationIntegers(int n)
{
 int s;

 if (n<0)
 {
  printf("summationIntegers: Parameter error\n");
  exit(1); /* End of the program */
 }

 /* ... */

 return s; /* Return value of the function */
}
```


Example of implementation : Auxiliary variables (if needed) of summationIntegers:

```
int summationIntegers(int n)
{
 int s;
   /* Here, if needed, declaration of other variables... */
 if (n<0)
 {
  printf("summationIntegers: Parameter error\n");
  exit(1); /* End of the program */
 }

 /* ... */

 return s; /* Return value of the function */
}
```


Example of implementation : algorithm of the summationIntegers

- **Formula version**

```
s = n*(n+1)/ 2;
```

- **Iterative version**

```
s = 0;                  /* On the same model as s, the integer i must  */
for(i=0;i<=n;i=i+1)     /* be defined at the beginning of the function */
{
  s = s + i;
}
```

- **Recursive version**

```
if (n==0) { s = 0; }
else { s = n + summationIntegers (n-1); }
```

Several implementations can correspond to the same specification !

Example of implementation : algorithm of the summationIntegers

```
int summationIntegers(int n)
{
 int s;

 if (n<0)
 {
  printf("summationIntegers: Parameter error\n");
  exit(1); /* End of the program */
 }

 s = n*(n+1)/2;

 return s; /* Return value of the function */
}
```


Implementation and validation (test sets) : algorithm of the summationIntegers

▷ **The preconditions must be tested!**

```
#include <stdio.h>   /* Because using printf */
#include <stdlib.h>  /* Because using exit   */

/* The code of the function summationIntegers */
/* ... */

int main(void)
{
 int r;

 r = summationIntegers(-1);  /* => summationIntegers: Parameter error */

 printf("%d\n",r);
 return 0;
}
```


Implementation and validation (test sets) : algorithm of the summationIntegers

▷ **The algorithm must be tested!**

```
#include <stdio.h>   /* Because using printf */
#include <stdlib.h>  /* Because using exit   */

/* The code of the function summationIntegers */
/* ... */

int main(void)
{
 int i,r;
 for(i=0;i<=10;i=i+1)
 {
   r = summationIntegers(i); /* => 0 1 3 6 10 15 21 28 36 45 55 */
   printf("%d ",r);
 }
 printf("\n");
 return 0;
}
```


Implementation and validation (test sets) : algorithm of the summation of the first integers

▷ **Validity :** an algorithm must be valid,
in conformity with its specification...

Implementation of an algorithm : the steps

Implementation of an algorithm: the 6 main steps

◇ To give the header of the function or the procedure ;

◇ To indicate the return value (in the case of a function) ;

◇ To write the preconditions ;

◇ To define auxiliary variables (if needed) ;

◇ To write the algorithm ;

◇ To test the algorithm.

Algorithm : Specification + Implementation (1)

▷ **Specification**

◇ To give an explicit name to the algorithm ;

◇ To describe in a sentence what the algorithm performs;

◇ To define the input/output parameters of the algorithm ;

◇ To specify the preconditions on the input parameters ;

◇ To give some examples of the use and the expected results.

▷ **Implementation** ... coupled with the specification !

◇ To give the header of the function or the procedure ;

◇ To indicate the return valeur (in the case of a function) ;

◇ To write the preconditions ;

◇ To define auxiliary variables (if needed) ;

◇ To write the algorithm ;

◇ To test the algorithm.

Algorithm : Specification + Implementation (2)

▷ **Properties of an algorithm**

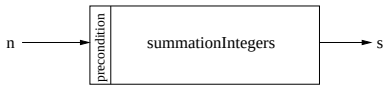
◇ validity : it must conform to the test sets

◇ robustness : it must verify the preconditions

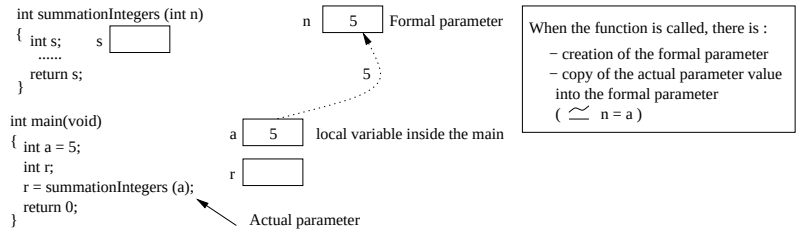
◇ reusable : it must be parameterized correctly

Parameter passing : by value (1)

▷ When specifying an algorithm, it is possible to declare parameters which are used in **input** only.

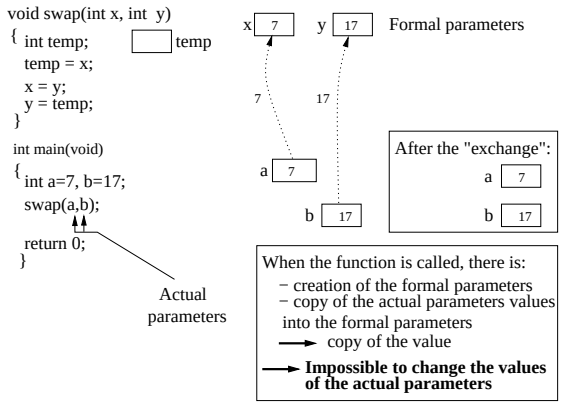


▷ When calling the function, the parameters are passed **by value**.



Parameter passing : by value (2)

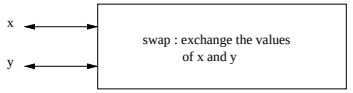
Second example : function for swapping values of 2 variables **(ERROR)**



IT IS IMPOSSIBLE TO MODIFY A VARIABLE WHEN USING A PARAMETER PASSING BY VALUE

Parameter passing : by address (1)

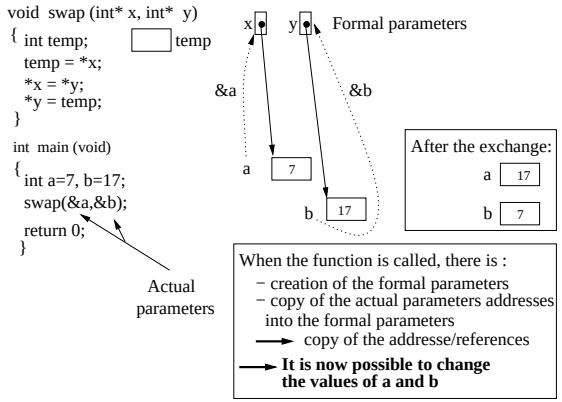
▷ When specifying an algorithm, it is possible to declare parameters which are used both in **input** and **output**.



▷ When calling the function, the parameters are passed **by address**.

▷ Remark : in some other languages, there is also the parameter passing **by reference** (very similar to the parameter passing by address).

Parameter passing : by address (2)



IT IS POSSIBLE TO MODIFY A VARIABLE WHEN USING A PARAMETER PASSING BY ADDRESS

Parameter passing : by address (3)

Summary

In C, in order to transmit (or to pass) parameter by address, it's necessary (1):

▷ To put, when calling the function:
an `&` (ampersand) followed by the name of the actual parameter.
$\Longrightarrow$ It allows to precise that the actual parameter can, eventually, be changed.
Example : `swap(&a,&b);`

▷ `&a` means "*the place where we can find variable* `a`"

▷ `&b` means "*the place where we can find variable* `b`"

Parameter passing : by address (4)

Summary

In C, in order to transmit (or to pass) parameter by address, it's necessary (2):

- ▷ To put in the header of the function:
  a **type** followed by a `*` and the name of the formal parameter.
  Example : `void swap(int* x, int* y)`
- ▷ To put in the body of the function:
  a `*` followed by the name of the formal parameter.
  $\Longrightarrow$ It allows to get the value of the actual parameter or to modify its value.
  Example : `*x = *y;`
- ▷ `x` means "*the place where we can find the TRUE variable (i.e. actual parameter)*"
- ▷ `*x` means "*Go to the place where we can find the TRUE variable*"
- ▷ the same for `y` and `*y`

Parameter passing : by address (5)

Summary

```
#include <stdio.h>

void swap(int* x,int* y)
{
 int temp;
 temp = *x;
 *x   = *y;
 *y   = temp;
}

int main(void)
{
 int a=7,b=17;
 printf("a=%d b=%d\n",a,b); /* a=7 b=17 */
 swap(&a,&b);
 printf("a=%d b=%d\n",a,b); /* a=17 b=7 */

 return 0;
}
```


Procedures and functions : how to choose?

How to choose between procedures and functions ?

Recall: procedure $\leftrightarrow$ function with no return value... `void`

- ▷ **It depends on the number of values which are expected by the calling function :**
  - ◇ 0 : procedure ! ex: `void printInt(int val);`
  - ◇ 1 : function ! ex: `int summationIntegers(int n);`
  - ◇ >1 : mostly procedure... ex: `void swap(int* x,int* y);`

  Except if an error code must be returned in order to know if the function terminates correctly or not...
  ... ex: `int divide(int a, int b, int* aDivb);`

C : examples of function types (1)

```
#include <stdio.h>
#include <stdlib.h>
void printInt(int val)  /* Function with a parameter and without a return value ... void */
{
 printf("%d",val);
}
int summationIntegers(int n)     /* Function with a parameter and with a return value    */
{
 int s;
 if (n<0) { printf("summationIntegers: Parameter error\n"); exit(1); /* Stops the program */
 }
 s = n*(n+1)/2;
 return s;
}
int main(void)
{
 int result;
 result=summationIntegers(5);
 printInt(result);
 return 0;
}
```

C : examples of function types (2)

```
#include <stdio.h>
/* Function divide: - input parameters: 2 integers a and b
   ==============  - output parameter: 1 integer known by its address aDivb
                   - computes the division of a by b... if it is possible !
   Return code : 1 (true), if it is possible or 0 (false), if it is not possible
*/
int divide(int a, int b, int* aDivb)                  /* Function with a return code */
{
 int ok = 0;
 if (b!=0) { ok=1; *aDivb = a/b; }
 return ok;
}
int main(void)
{
 int x=17, y=0, result;
 if (divide(x,y,&result)==1) {
        printf("Result of the division: %d\n",result);
 }
 else { printf("Warning, division by 0 !\n");
 }
 return 0;
}
```


Procedures and functions : writing algorithms in "english" (1)

- ▷ **Procedure/Function** : `procedure` or `function`
- ▷ **Name of the algorithm** : `name` of the function or the procedure
- ▷ **(**
- ▷ **List of the parameters separated by ,**
  - ◇ **In parameters :** `In type nom`
  - ◇ **In/Out parameters :** `InOut type nom`
- ▷ **)**
- ▷ **if function write** : `type` (the type of the return value)
- ▷ **begin**
- ▷ **...**
- ▷ **end**

Procedures and functions : writing algorithms in "english" (2)

Examples:

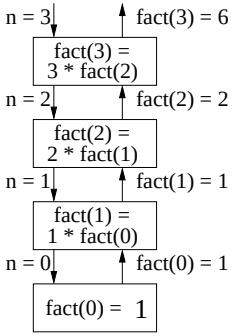
```
▷ function summationIntegers (In int n) : int
  begin
   let int s                          (* auxiliary variable *)
     if (n < 0) then Parameter error      (* precondition *)
     s <- n*(n+1) / 2
     return s;
  end
▷ procedure swap (InOut int x, IntOut int y)
  begin
   let int temp                       (* auxiliary variable *)
     temp <- x
     x    <- y
     y    <- temp
  end
```


Recursive functions (1)

Recursive function : function which, during its execution, calls itself (directly or not).

Example : the factorial, n! = 1∗2∗...∗n Let us recall : 0! = 1

```
int fact(int n)
{
 int result;

 if (n<0) { printf("fact: parameter error\n");
            exit(1); /* End of the program */
 }

 if (n==0)
 {
  result = 1;
 }
 else
 {
  result = n * fact(n-1);
 }
 return result;
}
```



Recursive functions (2)

Advantage : "easy" programming of some algorithms (quick sort,...)
Drawback : not efficient for all the problems.

Some methods exist to obtain a non recursive algorithm from a recursive one.

```
int fact(int n)
{
 int index, result;

 if (n<0) { printf("fact: parameter error\n");
            exit(1); /* End of the program */
 }

 result = 1;
 index  = 1;
 while (index <=n)
 {
  result = index * result;
  index  = index + 1;
 }

 return result;
}
```

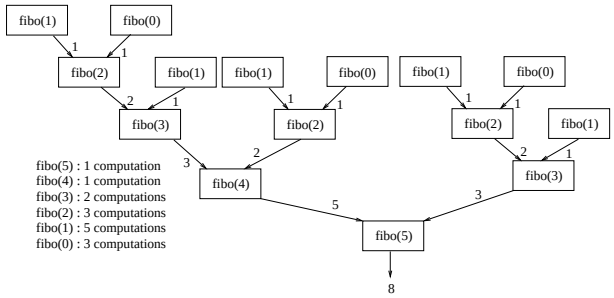

Recursive functions (3)

Another example : Fibonacci sequence

$$\begin{cases} \text{Fibonacci}(0) = 1 \text{ and Fibonacci}(1) = 1 \\ \text{Fibonacci}(n) = \text{Fibonacci}(n\text{-}2) + \text{Fibonacci}(n\text{-}1) \end{cases}$$

```
int fibo(int n)
{
 int result;

 if (n<0) { printf("fibo: parameter error\n");
            exit(1); /* End of the program */
 }

 if (n==0 || n==1)
 {
   result = 1;
 }
 else
 {
  result = fibo(n-1) + fibo(n-2);
 }

 return result;
}
```


Recursive functions (4)



Recursive functions (5)

Non recursive version :

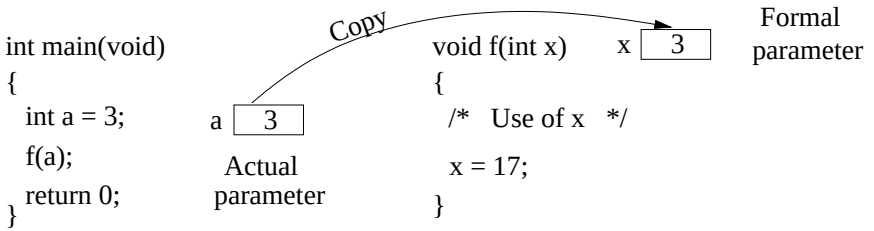
```
int fibo(int n)
{
 int result;

 if (n<0) { printf("fibo: parameter error\n");
            exit(1); /* End of the program */
 }

 if (n==0 || n==1) { result = 1; }
 else
 {
  int index,un,un_1,un_2;
  un_1   = 1;
  un_2   = 1;
  index  = 2;
  while (index <= n)
  {
   un   = un_1 + un_2;
   un_2 = un_1;
   un_1 = un;
   index = index + 1;
  }
  result = un;
 }

 return result;
}
```

Elementary data structures

A feedback on in/out parameters when using C functions

During a function call,
the parameters are passed **by value**.

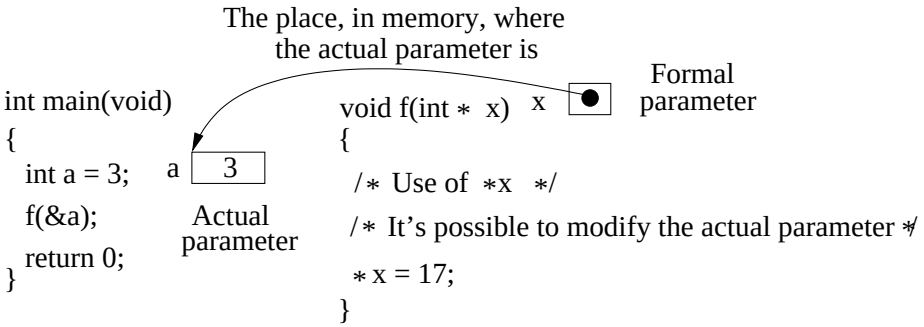
$\Longrightarrow$ The function receives only a **temporary copy** of each parameter.



$\Longrightarrow$ The function can't modify its parameters ! (In)

A feedback on in/out parameters when using C functions

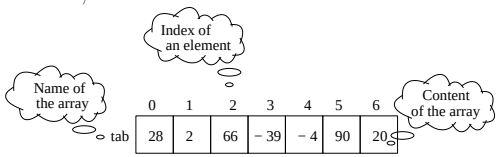
$\Longrightarrow$ SOLUTION: to transmit the place where the variable is and thus
to allow the modification of this variable in the function
(InOut)



Arrays

Definition :

An array *tab* is a **serie of n elements**, which can be accessed by their **rank** i in the array.



Interest :

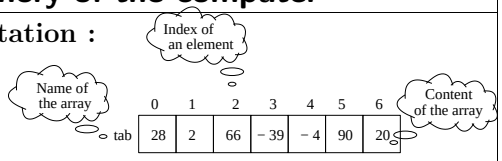
- ▷ A collection of values **of the same data type**.
- ▷ **Global treatments** on these data.

Example where arrays are usefull :

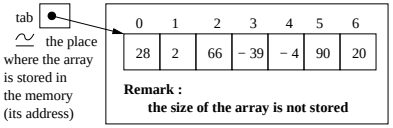
Firsly, the user have to enter a set of int values. Then, the program will print this set after the division of all these numbers by the maximal value of the set.
$\Longrightarrow$ The numbers must be stored in order to treat them globaly.

Arrays in the memory of the computer

Simplified and usual representation :



Memory representation (using C language)




```
int tab[7]={28,2,66,-39,-4,90,20};
int size=7;
```

And then :

- ▷ The size of the array : the number of elements
  - ◇ This number is defined when the array is created, and can't be changed !
  - ◇ The size: not stored in the array ⇒ it must be known by the programmer
- ▷ Valid indexes : integers from 0 to "size of the array - 1"
- ▷ Algorithmic language : the $(i+1)^{th}$ element of array tab is noted $tab[i]$

Common operations on arrays

Main operations generally made on arrays :

- ▷ **Creation** of an array of a size which must be known
- ▷ **Writing** of a value in an element of the array
- ▷ **Reading** of the content of an element of the array
- ▷ **Research** if a value can be found or not in the array
- ▷ **Modification** of the order in which the values are stored in the array (for example: array sorting algorithm)
- ▷ **Operations** between arrays (equality test, assignment, ...)
- ▷ **Operations** on all the elements of an array (for example : + 1 on all elements, ...)
- ▷ **...**

And now, using C language !

Arrays : creation, size, storage, access

- ▷ **Creation of an array of size n of objects of type T :**
  - ◇ T `tab[`n`];` ...the intial value of all the elements is undefined
  - ◇ T `tab[`n`] = {`e_0 `,` e_1`, ... ,` e_{n-1}`};` ...the elements must be known
  - ◇ **Remarks :**
    - ○ the size n must be known when the array is created
    - ○ use of $\{e_0 , e_1, \ldots , e_{n-1}\}$ **only during the creation** not after
- ▷ **How to get the size of an array ? : Not possible !**
  - ◇ Recall : the size must be known by the programmer
- ▷ **Writing a value in an element of the array :**
  - ◇ Modification of the (i+1)th element of the array
  - ◇ `tab[i] = e` $0 \leq i \leq$ size of the array- **1**
- ▷ **Reading of the value of an element of the array :**
  - ◇ Getting the value of the (i+1)th element of the array
  - ◇ `tab[i]` $0 \leq i \leq$ size of the array- **1**

Arrays : creation, size, storage, access

Example 1 : creations and access (1)

```
int main(void)
{
 int tab1[3] = {10, 20, 30};   /* 10, 20, 30 */
 int tab2[3];                  /*  ?,  ?,  ? */
 int i, size = 3;

 for(i=0;i<=size-1;i=i+1)       /* from 0 to size-1 */
 {
  tab2[i]=tab1[i]/10;

  /* trace output */
 }

 return 0;
}
```

Arrays : creation, size, storage, access

Example 1 : creations and access (2)

Trace output :

```
tab1          size     tab2        i

{10 20 30}    3        {1 ? ?}     0
{10 20 30}    3        {1 2 ?}     1
{10 20 30}    3        {1 2 3}     2
```


Arrays : creation, size, storage, access

Example 2 : printing the content of an array (1)

```
#include <stdio.h>

void printTab(int tab[], int size)             /* For 1 dimensional array,       */
{                                              /*  in the header of a function,  */
 int i;                                        /*     it is not necessary        */
 printf("The elements of the array are :\n");  /* to specify the size between [] */
 for(i=0;i<=size-1;i=i+1)
 {
  printf("%d ",tab[i]);
  /* trace output */
 }
}

int main(void)
{
 int t[5] = {90, 67, 2, 50, 23};
 int size = 5;

 printTab(t,size);  printf("\n");
 return 0;
}
```


Arrays : creation, size, storage, access

Example 2 : printing the content of an array (2)

Trace output :

```
tab               size      i   screen

{90 67 2 50 23}   5         0   90
{90 67 2 50 23}   5         1   90 67
{90 67 2 50 23}   5         2   90 67 2
{90 67 2 50 23}   5         3   90 67 2 50
{90 67 2 50 23}   5         4   90 67 2 50 23
```


Arrays : creation, size, storage, access

Example 3 : research of the smallest element of an array (1)

```
#include <stdio.h>
#include <stdlib.h>    /* exit */

int getMinTab(int tab[], int size)
{
 int i, min;
 if (size==0) { printf("getMinTab: tab is empty\n"); exit(1); }      /* precondition */
 min = tab[0];
 for(i=1;i<=size-1;i=i+1)
 {
  if (tab[i] < min) { min = tab[i]; }
  /* trace output */
 }
 return min;
}

int main(void)
{
 int t[5] = {90, 67, 2, 50, 23};
 int size = 5, m;

 m = getMinTab(t,size);  printf("Minimal value is:%d\n", m);
 return 0;
}
```

Arrays : creation, size, storage, access

Example 3 : research of the smallest element of an array (2)

Trace output :

```
                      tab            min

Before the loop : {90 67 2 50 23}     90

During the loop :

tab                size    min    i

{90 67 2 50 23}    5        67    1
{90 67 2 50 23}    5         2    2
{90 67 2 50 23}    5         2    3
{90 67 2 50 23}    5         2    4
```


Arrays : creation, size, storage, access

Example 4 : looking for a particular value in an array (1)

```
#include <stdio.h>

/* input parameters : - tab an array of int
                     - size the size of the array
                     - value, the value to look for
   output parameters: - indexe, the place where an int can be found
                        in order to store the indexe of the value in tab

   Return 1 (true) if the value is present in the array, otherwise 0 (false)

   Remark : the array tab dont need to be sorted before...                  */

int lookingForValue(int tab[], int size, int value, int* index)
{
 int i, found;

 found  = 0;  /* false */
 i      = 0;
 while (found!=1 && i<=size-1)     /* Same as : while (!found && i<=size-1) */
 {
  if (tab[i] == value)  { found  = 1; /* true */
                          *index = i;
                        }
  /* trace output */
  i = i + 1;
 }
 return found;
}
```


Arrays : creation, size, storage, access

Example 4 : looking for a particular value in an array (2)

```
int main(void)
{
 int t[5] = {90, 67, 2, 50, 23};
 int size = 5, val, ind;

 printf("Give me the value to look for? "); scanf("%d",&val);

                                          /* Same as:          */
 if (lookingForValue(t,size,val,&ind)==1)  /* if (lookingForValue(t,size,val,&ind)) */
 {
  printf("%d is located at the index %d\n",val,ind);
 }
 else
 {
  printf("%d is not present in the array\n",val);
 }

 return 0;
}
```


Arrays : creation, size, storage, access

Example 4 : looking for a particular value in an array (3)

Trace output if the user enter the value 50 :

```
tab               size   i   value  found  *index

{90 67 2 50 23}   5      0   50      0       ?
{90 67 2 50 23}   5      1   50      0       ?
{90 67 2 50 23}   5      2   50      0       ?
{90 67 2 50 23}   5      3   50      1       3

Remark: when i is equal to 4, the loop is not performed
```

Passing arrays as parameters

Remark : in the previous examples (2 to 4) the arrays were used as input parameters of functions.

Passing arrays as input/output parameters of functions :

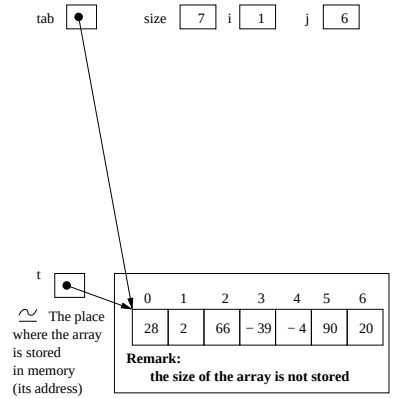
▷ The case of a function which exchanges 2 values in an array ...a first step to sorting !

```
#include <stdio.h>
#include <stdlib.h>  /* exit */

void exchange(int tab[], int size, int i, int j)
{
 int temp;

 if (!(i>=0 && i<=size-1)) { printf("Invalid parameter i\n"); exit(1); }
 if (!(j>=0 && j<=size-1)) { printf("Invalid parameter j\n"); exit(1); }

 temp   = tab[i];
 tab[i] = tab[j];
 tab[j] = temp;
}
```

Why does it works ????

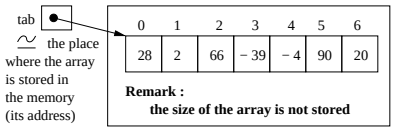
Passing arrays as parameters

```
void exchange(int tab[], int size, int i, int j)
{
 int temp;
                        /* The preconditions ... */
 temp   = tab[i];
 tab[i] = tab[j];
 tab[j] = temp;
}

int main(void)
{
 int t[7] = {28, 2, 66, -39, -4, 90, 20};
 int size = 7;

 printTab(t,size); /* 28 2 66 -39 -4 90 20 */

 exchange(t,size,1,6);

 printTab(t,size); /* 28 20 66 -39 -4 90 2 */

 return 0;
}
```



It is the place where the array is stored in memory which is transmitted !

Passing arrays as parameters



Thus :

▷ During a function call with an array as parameter, the values stored in the array can be changed.

▷ The array is then an input parameter and an output parameter !

Operations between arrays : assignment, equality test

▷ **Assign an array to an other one :** **No direct assignment!**

```
void copyTab(int tabDes[], int tabSrc[], int size)
{
 int i;
 for(i=0; i<=size-1; i=i+1)
 {
  tabDes[i]=tabSrc[i];      /* tabDes and tabSrc must have the same size */
 }
}
```

▷ **Equality test between arrays :** **No direct test !**

```
/* 1 (true) : si tab1 = tab2, 0 (false): si tab1!=tab2 */
int testTab(int tab1[], int tab2[], int size)   /* tab1 and tab2 must */
{                                               /* have the same size */
 int i=0, test = 1;  /* true */

 while (i<=size-1 && test==1)  /* Same as : while (i<=size-1 && test) */
 {
  if (tab1[i]!=tab2[i]) { test = 0; /* false */ }
  i = i + 1;
 }
 return test;
}
```

Operations between arrays : assignment, equality test

And then...

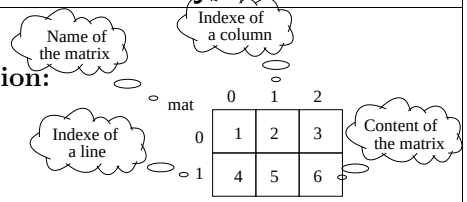
```
int main(void)
{
 int t1[3] = {30, 40, 50};
 int t2[3];

 t2 = t1              /* ____ FORBIDDEN ____ ( No direct assignment between arrays )    */

 t2 = {30, 40, 50};   /* ____ FORBIDDEN ____ ( {.. , .., ..} only during the creation ) */

 if (t1==t2)          /* ____ FORBIDDEN ____ ( No direct test between arrays )          */
 {
  ...
 }
 else
 {
  ...
 }
 return 0;
}
```


Operations between arrays : assignment, equality test

```
#include <stdio.h>
void printTab(int tab[], int size)
{ /* The code of printTab ... */ }
void copyTab(int tabDes[], int tabSrc[], int size)
{ /* The code of copyTab  ... */ }
int testTab(int tab1[], int tab2[], int size)
{ /* The code of testTab  ... */ }
int main(void)
{
 int t1[7] = {28, 2, 66, -39, -4, 90, 20};
 int t2[7];
 int size = 7;

 printTab(t1,size);   /* 28 2 66 -39 -4 90 20 */

 copyTab(t2,t1,size);

 printTab(t2,size);   /* 28 2 66 -39 -4 90 20 */
                                                    /* same as : if (testTab(t1,t2,size)) */
 if (testTab(t1,t2,size)==1) { printf("t1 Equal t2\n"); }
 else                        { printf("t1 Not equal t2\n"); }

 return 0;
}
```

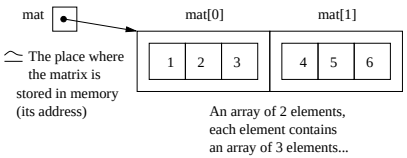

Matrixes: 2 dimensional arrays

Simplified and usual representation:



Memory representation (using C language)

```
int mat[2][3]={ {1, 2, 3}, {4, 5, 6} };
int nbLines=2;
int nbColumns=3;
```



$\Longrightarrow$ **A matrix : an array of arrays**

Matrixes: creation and access

▷ **Creation of a matrix of size nXm of objects of type T :**

◇ T `mat[`n`][`m`];` ...the initial value of all the elements is undefined

◇ But we can also write all the arrays (the lines) constituting the matrix :

T `mat[`n`][`m`] = {` $\{e_{00}, e_{01}, \ldots, e_{0m-1}\}$,
$\{e_{10}, e_{11}, \ldots, e_{1m-1}\}$,
$\ldots$,
$\{e_{n-10}, e_{n-11}, \ldots, e_{n-1m-1}\}$ `};`

◇ **Remarks :**

- the size nXm must be known when the matrix is created
- use of $\{ \{\ldots\}, \ldots, \{\ldots\} \}$ **only during the creation** not after

▷ **Recall, `mat` is an array of arrays :**

◇ **Access to the lines of mat :** `mat[l]` `mat[l]` is an array...

▷ **Then, `mat[l]` is an array :**

◇ **Access to the columns (and consequently elements):** `mat[l][c]`

Matrixes: creation and access

Example 1 : creation and access to matrixes (1)

```
int main(void)
{
 int mat1[2][3] = {{10, 20, 30} , {40, 50, 60}};   /* array of arrays */
 int mat2[2][3]; /*  ?   ?   ?      ?   ?   ?   */
 int nbLines = 2, nbCols = 3;
 int l,c;

 for(l=0;l<=nbLines-1;l=l+1)
 {
  for(c=0;c<=nbCols-1;c=c+1)
  {
   mat2[l][c] = mat1[l][c]/10;
   /* trace output */
  }
 }

 return 0;
}
```


Matrixes: creation and access

Example 1 : creation and access to matrixes (2)

Trace output :

```
l   c   mat1                      mat2

0   0   {{10 20 30} {40 50 60}}    {{1 ? ?} {? ? ?}}
0   1   {{10 20 30} {40 50 60}}    {{1 2 ?} {? ? ?}}
0   2   {{10 20 30} {40 50 60}}    {{1 2 3} {? ? ?}}
1   0   {{10 20 30} {40 50 60}}    {{1 2 3} {4 ? ?}}
1   1   {{10 20 30} {40 50 60}}    {{1 2 3} {4 5 ?}}
1   2   {{10 20 30} {40 50 60}}    {{1 2 3} {4 5 6}}
```


Matrixes: creation and access

Example 2 : printing the content of a matrix (1)

```
#include <stdio.h>

void printMat(int mat[2][3], int nbLines, int nbCols)  /* Multi-dimensional array:    */
{                                                      /* the size must be writen     */
 int l,c;                                              /* between [] => int mat[2][3] */

 for(l=0;l<=nbLines-1;l=l+1)
 {
  for(c=0;c<=nbCols-1;c=c+1)
  {
   printf("%d ",mat[l][c]);
   /* trace output */
  }
  printf("\n");
 }
}
```


Matrixes: creation and access

Example 2 : printing the content of a matrix (1)

```
int main(void)
{
 int m[2][3] = { {1, 2, 3} , {4, 5, 6} };
 int nbLines = 2, nbCols = 3;

 printMat(m,nbLines,nbCols);

 return 0;
}
```

Recall : it is the place where the matrix `m` is stored in memory which is transmitted to the function.
$\Longrightarrow$ The elements of the matrix can, possibly - if needed-, be changed in `printMat`.

Trace output :

```
l   c   mat                 Screen

0   0   {{1 2 3} {4 5 6}}   1
0   1   {{1 2 3} {4 5 6}}   1 2
0   2   {{1 2 3} {4 5 6}}   1 2 3
1   0   {{1 2 3} {4 5 6}}   1 2 3 \n4
1   1   {{1 2 3} {4 5 6}}   1 2 3 \n4 5
1   2   {{1 2 3} {4 5 6}}   1 2 3 \n4 5 6
```

Matrixes: creation and access

Example 3 : research of the smallest element of a matrix (1)

```
#include <stdio.h>
#include <stdlib.h>  /* exit */

int getMinMat(int mat[2][3], int nbLines, int nbCols)
{
 int l,c, min;

 if (nbLines==0||nbCols==0) { printf("getMinMat: mat is empty !\n"); exit(1); }

 min=mat[0][0];
 for(l=0;l<=nbLines-1;l=l+1)
 {
  for(c=0;c<=nbCols-1;c=c+1)
  {
   if (mat[l][c] < min) { min = mat[l][c]; }
   /* trace output */
  }
 }

 return min;
}
```


Matrixes: creation and access

Example 3 : research of the smallest element of a matrix (2)

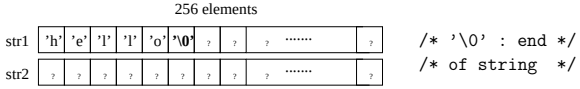
```
int main(void)
{
 int m[2][3] = { {90, 23, 5} , {4, 50, 67} };
 int nbLines = 2, nbCols = 3;

 printf("Minimal value is: %d\n",getMinMat(m,nbLines,nbCols));

 return 0;
}
```

Trace output :

```
l   c   mat                    min

0   0   {{90 23 5} {4 50 67}}   90
0   1   {{90 23 5} {4 50 67}}   23
0   2   {{90 23 5} {4 50 67}}   5
1   0   {{90 23 5} {4 50 67}}   4
1   1   {{90 23 5} {4 50 67}}   4
1   2   {{90 23 5} {4 50 67}}   4
```


Special type of arrays : strings of characters

```
#include <stdio.h>
#include <string.h>    /* Because using strcpy, strcmp, ... */

int main(void)
{
 char str1[256]="hello";
 char str2[256];

 strcpy(str2,str1);                    /* str2=str1 FORBIDDEN */

 str1[0]='H';

 printf("str1=%s str2=%s\n",str1,str2);        /* str1=Hello str2=hello */

 if (strcmp(str1,str2)==0)             /* if (str1==str2) ... FORBIDDEN */
 {
  printf("str1 and str2 are equal\n");
 }
 else
 {
  printf("str1 and str2 are different\n");
 }
 return 0;
}
```




```
/* '\0' : end */
/* of string  */
```


Special type of arrays : strings of characters

And thus, since a string of characters is an array of `char` :

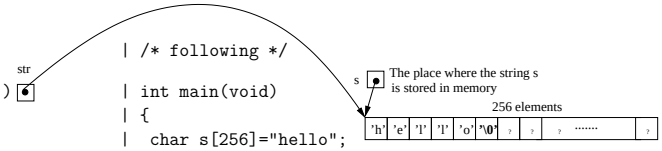
- ▷ Access to the (i+1)th character of a string `str` :
  ... same as for the arrays : `str[i]`
- ▷ To copy one string `str1` in an other one `str2`
  ... same as for the arrays : `str2=str1;` is **FORBIDDEN**
  $\Longrightarrow$ `strcpy(str2,str1)`
- ▷ Equality test between 2 strings of characters
  ... same as for the arrays : `if (str1==str2) ...` is **FORBIDDEN**
  $\Longrightarrow$ `strcmp(str1,str2)`
  `strcmp(str1,str2)` returns :
  - ◇ 0 if `str1` and `str2` are equal
  - ◇ < 0 if `str1` is less than `str2`
  - ◇ > 0 if `str1` is greater than `str2`

Special type of arrays : strings of characters

String functions perform string operations on null terminated strings ('\0')

- ▷ In C, there exist some functions allowing to manipulate strings of characters. Examples: `strcpy`, `strcmp`, ...
- ▷ There exist also a function which computes the length of a string: `strlen`

```
#include <stdio.h>                | /* following */
                                  |
int strlen(char str[])            | int main(void)
{                                 | {
 int i=0;                         |  char s[256]="hello";
                                  |
 while (str[i]!='\0')             |  printf("strlen(%s)=%d\n",s,strlen(s));
 {                                |
  i=i+1;                          |  return 0;
 }                                | }
 return i;                        |                    /* Screen : strlen(hello)=5 */
}                                 |
```



The structures (or records)

Definitions :

- ▷ **Structure or record**
  A record is a data structure which is composed of elements of various types;
- ▷ **Member**
  The elements of a structure are called members.

Examples :

- ▷ Person : – name
  – first name
  – age

The structures (or records)

Interests :

From already defined types,

the structures...
allow to define...
new data types.

$\Longrightarrow$ Structuring the data

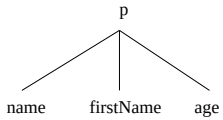
$\Longrightarrow$ Structuring the programs

$\Longrightarrow$ Programming and updating are easier, the programs are more readable....

Access to members

Access to members : .

Example : let `p` a record which represents a person



The access to the members of `p` are: `p.name`, `p.firstName`, `p.age`

Let us note that, using this example:
- ▷ normaly, the age would not be stored...
- ▷ the birth date would rather be stored $\Rightarrow$ the age would be computed !

The structures in C

Example 1 : the type `struct person` (1)

```
#include <stdio.h>    /* Because using printf... */
#include <string.h>   /* Because using strcpy... */

struct person         /* Definition of a new type : struct person */
{
 char name[256];
 char firstName[256];
 int  age;
};
```


The structures in C

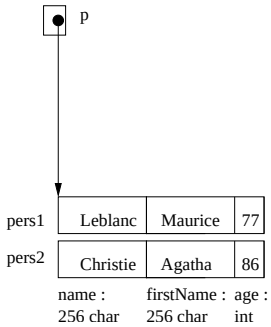
Example 1 : the type `struct person` (2)

```
struct person createPerson(char name[], char firstName[], int age)
{
 struct person p;

 strcpy(p.name,name);                      /* 1) p.name      <= name      */
 strcpy(p.firstName,firstName);            /* 2) p.firstName <= firstName */
 p.age = age;                              /* 3) p.age       <= age       */

 return p;
}

void printPerson(struct person p)
{
 printf("name:       %s\n",p.name);        /* 1) */
 printf("first name: %s\n",p.firstName);   /* 2) */
 printf("age:        %d ans\n",p.age);     /* 3) */
}
```


The structures in C

Example 1 : the type `struct person` (3)

```
void birthday(struct person* p)
{
 (*p).age = (*p).age + 1;

 printf("Happy birthday %s !\n",(*p).firstName);
}

int main(void)
{
 struct person pers1 = {"Leblanc", "Maurice", 77};
 struct person pers2;

 pers2 = createPerson("Christie","Agatha",86);

 printf("Person 1:\n"); printPerson(pers1);
 printf("Person 2:\n"); printPerson(pers2);

 birthday(&pers1);        /* &pers1 : the place where we can find pers1 */

 return 0;
}
```



The structures in C: equality test, assignment (1)

▷ **Equality test between structures**

→ No direct test ! ←

⟹ We have to test members one by one...

⟹ and thus, with the type `struct person` :

```
/* Return 1 (true) : if p1=p2, 0 (false): if p1!=p2 */
int testPerson(struct person p1, struct person p2)
{
 int test = 1;  /* true */
 if (strcmp(p1.name,p2.name)!=0) { test = 0; /* false */}
 else
 if (strcmp(p1.firstName,p2.firstName)!=0) { test = 0; /* false */}
 else
 if (p1.age!=p2.age) { test = 0; /* false */}
 return test;
}
```

The structures in C: equality test, assignment (2)

▷ **Assignement a structure to an other one:**

→ Direct assignment is possible ! ←

$\Longrightarrow$ And thus, with the type `struct person` : `pers2 = pers1;` is possible.

$\Longrightarrow$ **assignment of each member to each member**.

```
int main(void)
{
 struct person pers1 = {"Leblanc", "Maurice", 77};
 struct person pers2;

 pers2 = {"Leblanc", "Maurice", 77}; /* FORBIDDEN !!!! */
                                     /* i.e : {.. , .., ..} only during the creation */

 pers2 = pers1;                      /* Direct assignment ok */

 if (testPerson(pers1,pers2)==1)     /* Recall : if (pers1==pers2) is FORBIDDEN !!!! */
 {
  printf("It is the same person !\n");
 }
 return 0;
}
```


The structures in C

Example 2 : the type `struct person` + the type `struct ePerson (1)`

```
#include <stdio.h>    /* Because using printf... */
#include <string.h>   /* Because using strcpy... */

struct person
{ ...
};
struct person createPerson(char name[], char firstName[], int age)
{ ...
}
void printPerson(struct person p)
{ ...
}
void birthday(struct person* p)
{ ...
}

struct ePerson      /* Definition of a new type : struct ePerson */
{
 struct person p;
 char   email[256];
};
```


The structures in C

Example 2 : the type `struct person` + the type `struct ePerson (2)`

```
struct ePerson createEPerson(char name[], char firstName[], int age, char email[])
{
 struct ePerson ep;

 strcpy(ep.p.name,name);                    /* 1) ep.p.name      <= name      */
 strcpy(ep.p.firstName,firstName);          /* 2) ep.p.firstName <= firstName */
 ep.p.age = age;                            /* 3) ep.p.age       <= age       */
 strcpy(ep.email,email);                    /* 4) ep.mail        <= mail      */

 /* Remark: 1) + 2) + 3) ==> ep.p = createPerson(name,firstName,age); */

 return ep;
}

void printEPerson(struct ePerson ep)
{
 printf("name:       %s\n",ep.p.name);      /* 1) */
 printf("first name: %s\n",ep.p.firstName); /* 2) */
 printf("age:        %d ans\n",ep.p.age);   /* 3) */
 printf("email:      %s\n",ep.email);       /* 4) */

 /* Remark: 1) + 2) + 3) ==> printPerson(ep.p); */
}
```


The structures in C

Example 2 : the type `struct person` + the type `struct ePerson (3)`

```
int main(void)
{
 struct ePerson ePers1={{"Leblanc", "Maurice", 77},"leblanc@univ-brest.fr"};
 struct ePerson ePers2;

 ePers2=createEPerson("Christie","Agatha",86,"christie@univ-brest.fr");

 printf("ePerson 1:\n"); printEPerson(ePers1);
 printf("ePerson 2:\n"); printEPerson(ePers2);

 birthday(&ePers1.p);  /* Same as : birthday(&(ePers1.p)); */

 return 0;
}
```

Algorithm complexity

Definition and goal
Examples of cost computation
Complexity measurement and algorithms comparison
Accurate algorithms vs Heuristic approaches

Definition and goal

Definition

Complexity is a measurement of the "difficulty" of an algorithm computation according to the size `n` of the data.

▷ In this course, we will use the **time complexity**
(= number of elementary operations needed to perform the algorithm).

▷ The **memory complexity** also exists
(= size of the memory which is needed to perform the algorithm).

Goal

For a given problem, we want to write the "best" algorithm :

▷ whatever the programming language or the computer used,

▷ with a large amount of data (example : arrays of big size)

Remark : we do not consider here programming "ruses".

Examples of cost computation

Sequential search of a value `v` in an array `tab` of `n` elements (1) :

```
int lookingFor(int tab[], int n, int v)  /* n : the size of array tab */
{                                        /* v : the value to look for */
 int i, found;

 found = 0;  /* false */
 i     = 0;
 while (!found && i<=n-1)     /* Same as : while (found!=1 && i<=n-1) */
 {
  if (tab[i] == v) { found  = 1; /* true */ }

  i = i + 1;
 }

 return found;
}
```

This algorithm return : – `1` if the value `v` is present in the array `tab`
– `0` (false) otherwise.

Examples of cost computation

Sequential search of a value `v` in an array `tab` of `n` elements (2) :

▷ What is the cost of algorithm?

▷ If `v` is present in `tab` then all the ranks in `tab` are possible for `v`

▷ The complexity regarding the number of comparisons between `v` and an element of `tab` :

◇ Best case : `1` ; worst case : `n`

◇ Average : $\frac{n+1}{2}$

▷ The order of the complexity :

◇ If we consider a computer which is twice quicker then `/2` is not important $\Rightarrow$ `n+1`

◇ If `n` is large then `+1` is not important $\Rightarrow$ `n`

$\Longrightarrow O(n)$

Examples of cost computation

Dichotomic search of a value `v` in a sorted array `tab` of `n` elements :

Principle (see TD-TP) :

- ▷ The array must be sorted before...
- ▷ We compare the value `v` to look for and the middle element `m` of the array :
  - ◇ If $\mathtt{v} = \mathtt{m}$ then the value is found !
  - ◇ If $\mathtt{v} > \mathtt{m}$ then the value `v` must be searched in the right sub-array
  - ◇ If $\mathtt{v} < \mathtt{m}$ then the value `v` must be searched in the left sub-array
- ▷ The algorithm will be stopped if the value `v` is found or if the research space becomes empty.

Main idea : at each step, the research space is divided by 2
$\Longrightarrow O(log_2(\mathtt{n}))$

Examples of cost computation

Product `m` of 2 matrixes `m1` et `m2` of size `n`X`n` :

```
for(l=0; l<=n-1; l=l+1)
{
 for(c=0; c<=n-1; c=c+1)
 {
  m[l][c] = 0;
  for(i=0; i<=n-1; i=i+1)
  {
   m[l][c] = m[l][c] + m1[l][i]*m2[i][c];
  }
 }
}
```

- ▷ What is the cost of algorithm?
- ▷ Complexity when considering only multiplications :
  - ◇ internal loop : `n` multiplications
  - ◇ middle loop : $\mathtt{n}^2$ multiplications
  - ◇ external loop : $\mathtt{n}^3$ multiplications $\Longrightarrow O(\mathtt{n}^3)$

Complexity measurement and algorithms comparison

Complexity consideration allows to compare algorithms

Example :
- ▷ Let A_1 an algorithm with the complexity $O(\mathtt{n}^2)$
- ▷ Let A_2 an algorithm with the complexity $O(2\mathtt{n})$

$\Longrightarrow$ A_2 is better than A_1 when $\mathtt{n} > 2$

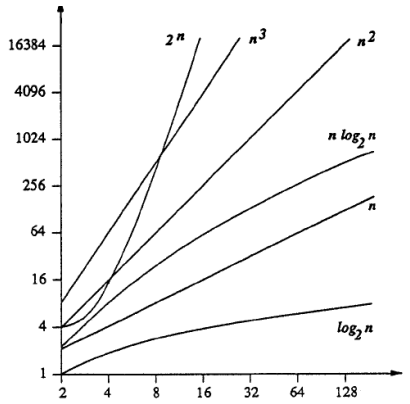
In order to compare two algorithms, we will evaluate the order of the complexity of these algorithms :

$$1 < log_2\,\mathtt{n} < \mathtt{n} < \mathtt{n}.log_2\,\mathtt{n} < \mathtt{n}^2 < \mathtt{n}^3 < ... < 2^{\mathtt{n}} < ...$$

(complexity ≡ asymptotic value: when `n` becomes ∞)

Complexity measurement and algorithms comparison

In order to compare :



Complexity measurement and algorithms comparison

How the computation time varies depending on the size of the data?

... while executing 10^6 operations per second

n \ complexity	1	$log_2(n)$	n	$n.log_2(n)$	n^2	n^3	2^n
10^2	$\approx 1\ \mu s$	6,64 μs	0,1 ms	0,66 ms	10 ms	1 s	4.10^{16} a
10^3	$\approx 1\ \mu s$	9,97 μs	1 ms	9,97 ms	1 s	16,66 m	∞
10^4	$\approx 1\ \mu s$	13,29 μs	10 ms	0,13 s	1,66 m	11,55 j	∞
10^5	$\approx 1\ \mu s$	16,61 μs	0,1 s	1,66 s	2,78 h	31,7 a	∞
10^6	$\approx 1\ \mu s$	19,93 μs	1 s	19,93 s	11,57 j	3.10^7 a	∞

Complexity measurement and algorithms comparison

What is the amount of data that can be treated in a given time?

... while executing again 10^6 operations per second

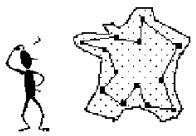
time \ complexity	1	$log_2(n)$	n	$n.log_2(n)$	n^2	n^3	2^n
1 s	∞	∞	10^6	$6{,}3.10^4$	10^3	10^2	19
1 m	∞	∞	6.10^7	$2{,}8.10^6$	7.10^3	4.10^2	25
1 h	∞	∞	38.10^8	$1{,}3.10^8$	6.10^4	15.10^2	31
1 j	∞	∞	$8{,}6.10^{10}$	$2{,}7.10^9$	$2{,}9.10^5$	44.10^2	36

$\infty : > 10^{100}$

Accurate algorithms vs Heuristic approaches

Accurate algorithms

Classical algorithms solve problems accurately...
but there are sometimes too slow !



Example : Travelling Salesman Problem (TSP).

- ▷ To visit each city exactly once
- ▷ To return to the origin city
- ▷ While minimizing the route...Trying to find the shortest one.

$\Longrightarrow O(n!)$

And then, if it takes $1\mu s$ to compute the length of a route :

Nb towns	Nb possibilities	Computation time
5	12	$12\mu s$
10	181440	0,18 ms
15	43 billions	12 hours
20	60.10^{15}	1928 years
25	310.10^{21}	9,8 billions of years

Heuristic approaches

Heuristics provide a quick but approximate resolution.

Examples : genetic algorithms, artificial immune systems,...

Sorting algorithms

Definition
A very useful function : `exchange`
Bubble sort
Merge sort
Quick sort

Definitions

Specification of a sorting algorithm

- ▷ **Input** : 1 array
- ▷ **Output** : 1 array containing exactly the same values as the input array, but these values are sorted. It means that :

  $\forall i, j \quad i < j \Longrightarrow t[i] \leq t[j]$ (increasing order)

  $\forall i, j \quad i < j \Longrightarrow t[i] \geq t[j]$ (decreasing order)
- ▷ Remarks : – the same array can be used as an input and output parameter
  – in this course, we only consider the increasing order sorting

Several sorting algorithms :

- ▷ Bubble sort
- ▷ Selection sort
- ▷ Insertion sort
- ▷ Merge sort
- ▷ Quick sort
- ▷ ...

A very useful function : `exchange`

Function which exchanges 2 values in an array

Many sorting algorithms use a function to swap (or exchange) the values stored in two elements of an array.

```
void exchange(int tab[], int size, int i, int j)
{
 int temp;

 if (!(i>=0 && i<=size-1)) { printf("Invalid parameter i\n"); exit(1); }
 if (!(j>=0 && j<=size-1)) { printf("Invalid parameter j\n"); exit(1); }

 temp   = tab[i];
 tab[i] = tab[j];
 tab[j] = temp;
}
```


Bubble sort : definition

Principle

The greatest elements are "pushed" to the end of the array.
For the `n` first elements (and then for the `n-1`, ...), we perform :

For each index, starting from 0 :

- ▷ We compare successively an element `e` and its following element in the array.
- ▷ If `e` is greater than its following element, we exchange the values, otherwise nothing to do.

Example:

	input array :	3 4 2 3
First pass :	▷ 3 compared to 4 ($\nleftrightarrow$)	3 4 2 3
	▷ 4 compared to 2 ($\leftrightarrow$)	3 2 4 3
	▷ 4 compared to 3 ($\leftrightarrow$)	3 2 3 4
Second pass :	▷ 3 compared to 2 ($\leftrightarrow$)	2 3 3 4
	▷ 3 compared to 3 ($\nleftrightarrow$)	2 3 3 4
Third pass :	▷ 2 compared to 3 ($\nleftrightarrow$)	2 3 3 4
Fourth pass :		

Bubble sort : code

```
void bubbleSort(int tab[], int size)
{
 int i,j;
 int n = size;

 for(i=0; i<=n-1 ; i=i+1)
 {
  for(j=0; j<=n-1-i-1 ; j=j+1)   /* n-1 : as usual                                  */
  {                              /* -i : because the last i elements are already sorted */
   if (tab[j]>tab[j+1])          /* -1 : because j+1 (in order to avoid an overflow )   */
   {
    exchange(tab,size,j,j+1);
   }
   /* trace output */
  }
 }
}
```

Bubble sort : function call

```
#include <stdio.h>
#include <stdlib.h>  /* exit */
void printTab(int tab[], int size)
{
 /* the code of printTab */
}
void exchange(int tab[], int size, int i, int j)
{
 /* the code of exchange */
}
void bubbleSort(int tab[], int size)
{
 /* the code of bubbleSort */
}
int main(void)
{
 int tab[5] = {90,67,2,50,23};
 int size   = 5;
 printTab(tab,size);      /* {90,67,2,50,23} */
 bubbleSort(tab,size);
 printTab(tab,size);      /* {2,23,50,67,90} */
 return 0;
}
```


Bubble sort : trace output

Trace output : with the array {90, 67, 2, 50, 23}

i	j	tab	size
?	?	{90 67 2 50 23}	5
0	0	{67 90 2 50 23}	5
0	1	{67 2 90 50 23}	5
0	2	{67 2 50 90 23}	5
0	3	{67 2 50 23 90}	5
1	0	{ 2 67 50 23 90}	5
1	1	{ 2 50 67 23 90}	5
1	2	{ 2 50 23 67 90}	5
2	0	{ 2 50 23 67 90}	5
2	1	{ 2 23 50 67 90}	5
3	0	{ 2 23 50 67 90}	5
?	?	{ 2 23 50 67 90}	5

Bubble sort : complexity

Number of elementary operations needed for the bubble sort :

- ▷ For the internal loop (`j`) : comparison : 1 ;
  exchange (if performed): 3 assignments ;
  `j` management : 3 (addition, assignment, test)

  $\Longrightarrow$ 7 operations
- ▷ For the external loop (`i`) : `i` management : 3 (addition, assignment, test) ;
  the internal loop (`j`)

  $\Longrightarrow 3 * \mathtt{n} + 7 * ((\mathtt{n}-1) + (\mathtt{n}-2) + ... + 1)$

  $\Longrightarrow 3 * \mathtt{n} + 7 * \frac{(\mathtt{n}-1)*\mathtt{n}}{2} = \frac{7}{2}\,\mathtt{n}^2 - \frac{\mathtt{n}}{2}$ operations

And thus :

Bubble sort: complexity is $O(\mathtt{n}^2)$ ... It is a bad sorting algorithm!
because some sorts are in $0(\mathtt{n}.log_2(\mathtt{n}))$

Bubble sort : improvement

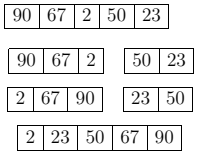
```
void bubbleSortRuse(int tab[], int size)
{
 int inversion = 1; /* true at the beginning */
 int i,j;
 int n = size;

 i=0;
 while (i<=n-1 && inversion) /* inversion allows to stop if the array is already sorted */
 {
  inversion = 0;    /* false */

  for(j=0; j<=n-1-i-1 ; j=j+1)   /* n-1 : as usual                                    */
  {                              /* -i : because the last i elements are already sorted */
   if (tab[j]>tab[j+1])          /* -1 : because j+1 (in order to avoid an overflow )   */
   {
    exchange(tab,size,j,j+1);
    inversion = 1;  /* true */
   }
  }
  i = i + 1;
 }
}
```

The complexity is always $O(\mathtt{n}^2)$

Merge sort : definition

Principle : divide and rule ("diviser pour régner")

Method :



- ▷ The array is divided in two sub-arrays
- ▷ Each sub-array is sorted
- ▷ The two sorted sub-array are merged

Drawback of merge sort: copy of temporary sub-arrays...

Merge sort : code (0)

A function to check if an array is sorted or not

```
/* Return 1 (true)  if the array is sorted      */
/* Return 0 (false) if the array is not sorted  */

int isSorted(int tab[], int size)
{
 int sorted = 1; /* true by default */
 int i;

 i = 1;
 while (sorted && i<=size-1)  /* Same as : while (sorted==1 && i<=size-1) */
 {
  if (tab[i-1]>tab[i])
  {
   sorted = 0; /* false */
  }
  i = i + 1;
 }

 return sorted;
}
```


Merge sort : code (0')

Copy of a sub-array of `tab` ⟶ copy in `subTab` for indexes between `lowerBound` and `upperBound`...

```
void copySubArray(int tab[], int subTab[], int lowerBound, int upperBound)
{
 int i,j;
 j=0;
 for(i=lowerBound; i<=upperBound; i=i+1)
 {
  subTab[j]=tab[i];
  j=j+1;
 }
}
```


Merge sort : code (1)

Merge of two sorted sub-arrays (1)

```
void merge(int tab[], int l, int m, int r)
{
  int i;                        /* index on tab  */

  int nbElemsTab1 = m - l + 1;
  int tab1[nbElemsTab1];
  int i1;                       /* index on tab1 */

  int nbElemsTab2 = r - m;
  int tab2[nbElemsTab2];
  int i2;                       /* index on tab2 */

  copySubArray(tab,tab1,l,m);   /* Copy, in array tab1, of elements */
                                /* of tab from l to m indexes       */

  copySubArray(tab,tab2,m+1,r); /* Copy, in array tab2, of elements */
                                /* of tab from m+1 to d indexes     */

  if (!isSorted(tab1,nbElemsTab1)) { printf("tab1 is not sorted\n"); exit(1); }
  if (!isSorted(tab2,nbElemsTab2)) { printf("tab2 is not sorted\n"); exit(1); }
```

Merge sort : code (2)

Merge of two sorted sub-arrays (2)

```
  i1=0; i2=0;
  for(i=l; i<=r; i=i+1)
  {
   if (i1<=nbElemsTab1-1 && i2<=nbElemsTab2-1)
   {
    if (tab1[i1]<tab2[i2]) { tab[i]=tab1[i1]; i1=i1+1; }
    else                   { tab[i]=tab2[i2]; i2=i2+1; }
   }
   else
   {   /* if one of the arrays is empty before the other */
    if (i1<=nbElemsTab1-1)
    {
     tab[i]=tab1[i1]; i1=i1+1; /* tab2 is empty */
    }
    else
    {
     tab[i]=tab2[i2]; i2=i2+1; /* tab1 is empty */
    }
   }
  }
}
```


Merge sort : code (3)

Dividing the array, sorting of the two sub-arrays and merging

```
void mergeSortRec(int tab[], int size, int start, int stop)
{
 int n = size;
 if (start<0 || start >= size ||
     stop <0 || stop >= size)   { printf("Error on start or stop\n");
                                  exit(1);
 }
 if (start < stop)
 {
  int m = (start+stop)/2;
  mergeSortRec(tab,size,start,m);
  mergeSortRec(tab,size,m+1,stop);
  merge(tab,start,m,stop);
  /* trace output */
 }
}
void mergeSort(int tab[], int size)
{
 mergeSortRec(tab,size,0,size-1);
}
```


Merge sort : function call

```
#include <stdio.h>
#include <stdlib.h>  /* exit */

void printTab(int tab[], int size)
{ /* the code of printTab */ }
int isSorted(int tab[], int size)
{ /* the code of isSorted */ }
void copySubArray(int tab[], int subTab[], int lowerBound, int upperBound)
{ /* the code of copySubArray */ }
void merge(int tab[], int l, int m, int r)
{ /* the code of merge */ }
void mergeSortRec(int tab[], int size, int start, int stop)
{ /* the code of mergeSortRec */ }
void mergeSort(int tab[], int size)
{
 mergeSortRec(tab,size,0,size-1);
}

int main(void)
{
 int tab[5]= {90,67,2,50,23};
 int size = 5;
 printTab(tab,size);      /* {90,67,2,50,23} */
 mergeSort(tab,size);
 printTab(tab,size);      /* {2,23,50,67,90} */
 return 0;
}
```


Merge sort : trace output

Trace output : with the array {90, 67, 2, 50, 23}

```
m   tab
-------------------------
?   {90  67   2  50  23}
-------------------------
0   {67  90   2  50  23}
1   { 2  67  90  50  23}
3   { 2  67  90  23  50}
2   { 2  23  50  67  90}
-------------------------
?   { 2  23  50  67  90}
-------------------------
```

Merge sort : complexity (1)

In order to simplify, we only consider the comparisons between the elements of the array.

Evaluation of the complexity $C(\mathtt{n})$:

- ▷ When executing the function `merge` on the full array of size `n`, `(n - 1)` comparisons are performed.
- ▷ If `n` is large, we can simplify and consider that we have `n` comparisons.
- ▷ Considering that we divide the array in two sub-arrays on which we do the same: $C(\mathtt{n}) = \mathtt{n} + 2{*}C(\mathtt{n}/2)$

$$\Longrightarrow C(\mathtt{n}) = \mathtt{n} + \mathtt{n} * \mathtt{log}_2(\mathtt{n})$$ ... see the following proof

$$\Longrightarrow C(\mathtt{n}) = O(\mathtt{n} * \mathtt{log}_2(\mathtt{n}))$$

Drawback of merge sort: copy of temporary sub-arrays...

Merge sort : complexity (2)

Proof :

In order to simplify, we consider that we have an array of size $\mathtt{n} = 2^p$.

$C(\mathtt{n}) = \mathtt{n} + 2{*}C(\mathtt{n}/2)$

$\Rightarrow C(2^p) = 2^p + 2{*}C(2^{p-1})$

$\Rightarrow C(2^p) = 2^p + 2^p{*}p$ ....To demonstrate by induction (see next slide). ....A démontrer par récurrence.

$\Rightarrow C(\mathtt{n}) = \mathtt{n} + \mathtt{n}{*}log_2(\mathtt{n})$ ... YES !

Recall : $\mathtt{n} = 2^p \Longleftrightarrow p = log_2(\mathtt{n})$

Merge sort : complexity (3)

Let us prove by induction : $C(2^p) = 2^p + 2{*}C(2^{p-1}) \Rightarrow C(2^p) = 2^p + 2^p{*}p$

- ▷ **Initialization :**

  $C(0) = 0$ (no data...)

  $C(2^0) = C(1) = 1 + 2{*}C(0) = 1 = 2^0 + 2^0{*}0$

  $C(2^1) = C(2) = 2 + 2{*}C(1) = 4 = 2^1 + 2^1{*}1$

  $C(2^2) = C(4) = 4 + 2{*}C(2) = 12 = 2^2 + 2^2{*}2$

  ...

- ▷ **Induction :**

$$\begin{aligned} C(2^{p+1}) &= 2^{p+1} + 2{*}C(2^p) \\ &= 2^{p+1} + 2{*}(2^p + 2^p{*}p) \\ &= 2^{p+1} + 2^{p+1} + 2^{p+1}{*}p \\ &= 2^{p+1} + 2^{p+1}{*}(1 + p) \\ &= 2^{p+1} + 2^{p+1}{*}(p + 1) \end{aligned}$$ ... YES !

Quick sort : definition (1)

A sort :

- ▷ without temporary arrays and
- ▷ a complexity in $O(\mathtt{n} * \mathtt{log}_2(\mathtt{n}))$

A "divide and rule" approach

- ▷ Dividing the array in two sub-arrays
- ▷ Sorting each sub-arrays

Quick sort : definition (2)

Principle :

- ▷ Pick an element called **pivot** (usually the first one, ex: **T[0]**)
- ▷ Reorder the elements of the array in order to have:
  - ◇ On the right, all the elements >**pivot**
  - ◇ On the left, all the elements ≤**pivot**
- ▷ This principle is applied recursively to the two obtained sub-arrays

Remark : no need to merge the two sorted sub-arrays...

Queue and stack

Abstract data type
Queue (FIFO: First In First Out) ... *File* in french
Stack (LIFO: Last In First Out) ... *Pile* in french
One more step towards abstraction : modular programming

Abstract data type

Main goal : "formal" description of a data type.

Abstract type :

- ▷ **Type :** t
- ▷ **Used types :** all the types used by type t
- ▷ **Operations :** operations which are possibles on an object of type t
- ▷ **Preconditions :** preconditions on these operations
- ▷ **Axioms :** ... which, on an object of type t, must be always true !

Remark : this "formal" description does not depend on a specific programming language.

Abstract data type : example

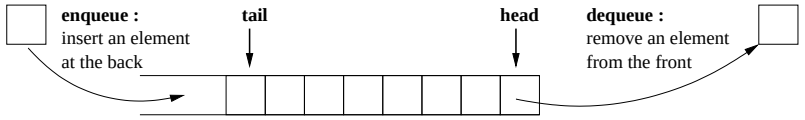
- ▷ **Type :** boolean
- ▷ **Used types :** –
- ▷ **Operations :**
  - ◇ true : → boolean
  - ◇ false : → boolean
  - ◇ not (¬) : boolean → boolean
  - ◇ and (∧) : boolean X boolean → boolean
  - ◇ or (∨) : boolean X boolean → boolean
- ▷ **Preconditions :** –
- ▷ **Axioms :**
  - ◇ ¬(true) = false
  - ◇ ¬(false) = true
  - ◇ a ∧ true = a
  - ◇ a ∧ false = false — Recall : ∧ is commutative
  - ◇ a ∨ true = true
  - ◇ a ∨ false = a — Recall : ∨ is commutative

Queue: introduction and interest

Introduction :

▷ The queues are used in computer science to manage objects that are waiting a later process **in the order of their arrival**.

▷ In a queue the elements are always: **- added at the end** and **- extracted from the front**.



A queue (First In Fisrt Out: FIFO)

Interest : queues are widely used data structures because they allow to manage the access to computer resources (printer, processor, ...).

In this course, we only consider the case of queues containing integers. These integers can be, for example, a ticket number...

Queue: abstract type (1)

▷ **Type :** Queue containing objects of type element

▷ **Used types :** boolean, element

▷ **Operations :** In english...

- ◇ créer : → queue ... create
- ◇ estVide : queue → boolean ... isEmpty
- ◇ estPleine : queue → boolean ... isFull
- ◇ premier : queue → element ... first
- ◇ enfiler : queue X element → queue ... enqueue
- ◇ défiler : queue → queue ... dequeue

▷ **Preconditions :**

- ◇ premier(q) **iff** estVide(q) = false
- ◇ enfiler(q,e) **iff** estPleine(q) = false
- ◇ défiler(q) **iff** estVide(q) = false

With q (a queue) and e (an element).

Queue: abstract type (2)

▷ **Axioms :**

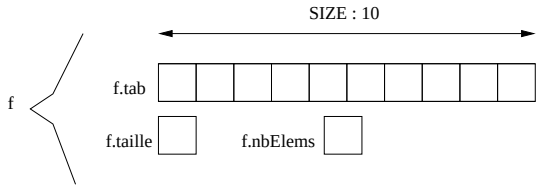
- ◇ estVide(créer()) = true
- ◇ estVide(enfiler(q,e)) = false
- ◇ estVide(q) = true ⇒ premier(enfiler(q,e)) = e
- ◇ estVide(q) = false ⇒ premier(enfiler(q,e)) = premier(q)
- ◇ estVide(q) = true ⇒ estVide(défiler(enfiler(q,e))) = true
- ◇ estVide(q) = false ⇒ défiler(enfiler(q,e)) = enfiler(défiler(q) ,e)

With q (a queue) and e (an element).

Queue : naive C implementation (1)

```
#define SIZE 10

struct stfile { int tab[SIZE];
                int taille;    /* the size of the array tab               */
                int nbElems;   /* the number of values stored in the array */
              };

typedef struct stfile file;    /* file  <=> queue in english... */
```

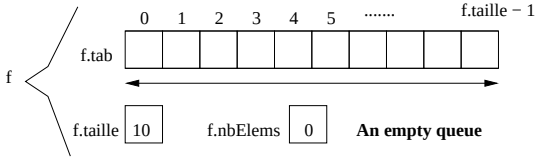
if `f` is a `file` :



Queue : naive C implementation (2)

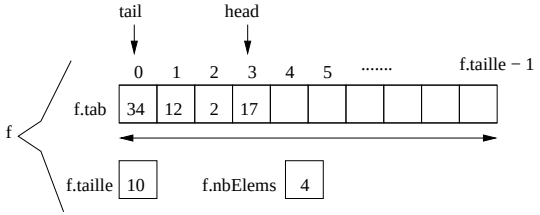
creerFile :

```
file creerFile(void)
{
 file f;
 f.taille  = SIZE;
 f.nbElems = 0;
 return f;
}
```



afficherFile :

```
void afficherFile(file f)
{
 int i;
 printf("[HEAD] ");
 for(i=f.nbElems-1 ; i>=0; i=i-1)
 {
  printf("%d ",f.tab[i]);
 }
 printf("[TAIL]");
}
```



Queue : naive C implementation (3)

estVideFile :

```
int estVideFile(file f)
{
 int returnValue = 0;
 if (f.nbElems == 0) returnValue=1;
 return returnValue;
}
```

estPleineFile

```
int estPleineFile(file f)
{
 int returnValue = 0;
 if (f.nbElems == f.taille) returnValue=1;
 return returnValue;
}
```

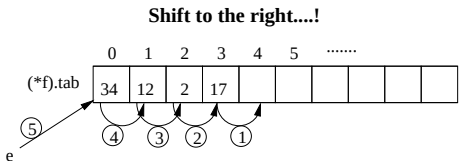
premierFile

```
int premierFile(file f)
{
 if (estVideFile(f)) { printf("The queue is empty!!!...\n"); exit(1); }
 return f.tab[f.nbElems-1];
}
```


Queue : naive C implementation (4)

enfilerFile :

```
void enfilerFile(file* f, int e)
{
 int i;

 if (estPleineFile(*f)) { printf("The queue is full!!!...\n"); exit(1); }

 for(i=(*f).nbElems-1 ; i>=0; i=i-1)
 {
  (*f).tab[i+1] = (*f).tab[i];
 }

 (*f).tab[0]  = e;
 (*f).nbElems = (*f).nbElems + 1;
}
```



Queue : naive C implementation (5)

defilerFile :

```
void defilerFile(file* f)
{
 if (estVideFile(*f)) { printf("The queue is empty!!!...\n"); exit(1); }

 (*f).nbElems = (*f).nbElems - 1;
}
```

Queue : naive C implementation (6)

A menu allowing to test the functions (1) :

```
int menu (void)
{
 int rep, minRep = 0 , maxRep = 4;

 do
 {
   printf("\n");
   printf("0. Stop the program\n");
   printf("1. Print the queue\n");
   printf("2. Print the head of the queue\n");
   printf("3. Put an element in the queue\n");
   printf("4. Remove an element of the queue\n");

   printf("Your choice:");  scanf("%d",&rep);

   if (rep < minRep || rep > maxRep) { printf("Entry error\n"); }

  } while (rep < minRep || rep > maxRep);

 return rep;
}
```


Queue : naive C implementation (7)

A menu allowing to test the functions (2) :

```
int main(void)
{
 file f = creerFile();
 int rep;

 rep = menu();

 while (rep!=0)
 {                                /* We can also program with some if..else */
  switch (rep)
  {
    case 1: { afficherFile(f);
              printf("\n");
              break;
            }
    case 2: { if (estVideFile(f))
              {
               printf("Warning, the queue is empty!\n");
              }
              else { int head = premierFile(f);
                     printf("The head of the queue is : %d\n",head);
                   }
              break;
            }
                                                          /* main ... to follow */
```


Queue : naive C implementation (8)

A menu allowing to test the functions (3) :

```
    case 3: { if (estPleineFile(f))                          /* main ... the following part */
              {
               printf("Warning, the queue is full!\n");
              }
              else { int elem;
                     printf("Element to put in the queue?\n");
                     scanf("%d",&elem);
                     enfilerFile(&f,elem);
                   }
              break;
            }
   default: { if (estVideFile(f))
              {
               printf("Warning, the queue is empty!\n");
              }
              else { defilerFile(&f); }
              break;
            }
  }

  rep = menu();
 }

 printf("Stop the program\n");
 return 0;
}
```


Queue : C implementation, naive (9) ⇒ circular (1)

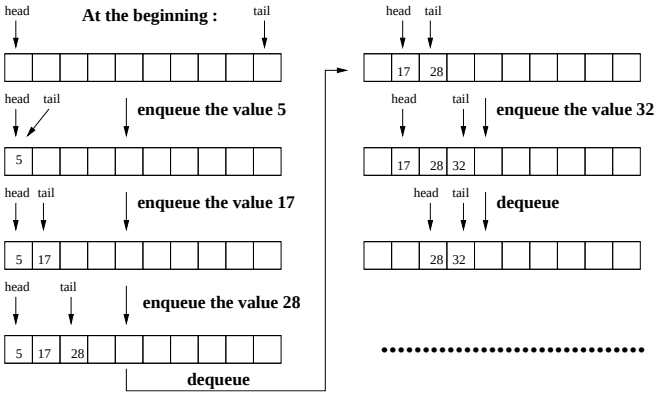
Attention : this implementation is a quite naive !

- ▷ **Advantage :** the implementation is very easy
- ▷ **Drawbach :** the shift during `enfilerFile`

Solution : a circular implementation of a queue

```
#define SIZE 10

struct stfile { int tab[SIZE];
                int taille;    /* taille <=> size in english, the size of the array tab */
                int nbElems;   /* usefull by not necessary : number of values stored    */
                int tete;      /* tete  <=> head in english, index of the first element */
                int queue;     /* queue <=> tail in english, index of the last  element */
              };

typedef struct stfile file;
```

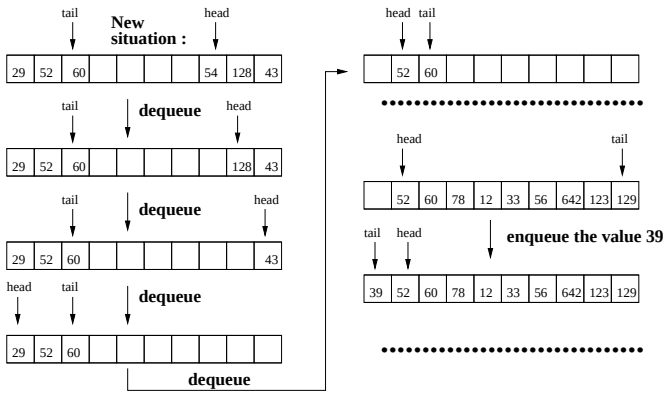
Queue : circular C implementation (2)



`enqueueFile` $\Longrightarrow$ Evolution of the tail : `tail = (tail + 1) % size`
and, the element is assigned in position `tail`

`dequeue` $\Longrightarrow$ Evolution of the head : `head = (head + 1) % size`

Queue : circular C implementation (3)



`enqueueFile` $\Longrightarrow$ Evolution of the tail : `tail = (tail + 1) % size`
and, the element is assigned in position `tail`

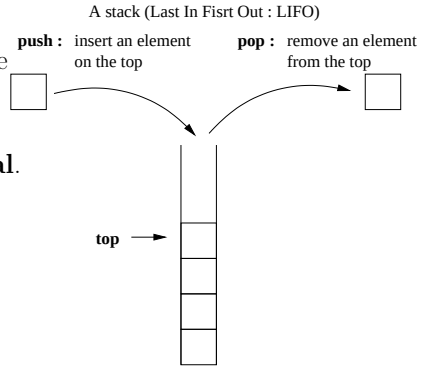
`dequeue` $\Longrightarrow$ Evolution of the head : `head = (head + 1) % size`

Stack: introduction and interest

Introduction :

▷ The stack are used in computer science to manage objects that are waiting a later process in
the reverse order of their arrival.

▷ In a stack the elements are always:
added on the top and
extracted from the top.



Interest : stacks are widely used data structures because they allow, for example, to implement function calls.

Here, we only consider the case of stacks containing integers...

Stack : abstract type (1)

▷ **Type :** Stack containing objects of type element
▷ **Used types :** boolean, element
▷ **Operations :** In english...

◇ créer : $\longrightarrow$ stack	... create
◇ estVide : stack $\longrightarrow$ boolean	... isEmpty
◇ estPleine : stack $\longrightarrow$ boolean	... isFull
◇ sommet : stack $\longrightarrow$ element	... top
◇ empiler : stack X element $\longrightarrow$ stack	... push
◇ dépiler : stack $\longrightarrow$ stack	... pop

▷ **Preconditions :**
◇ sommet(s) **iff** estVide(s) = false
◇ empiler(s,e) **iff** estPleine(s) = false
◇ dépiler(s) **iff** estVide(s) = false

With s (a stack) and e (an element).

Stack : abstract type (2)

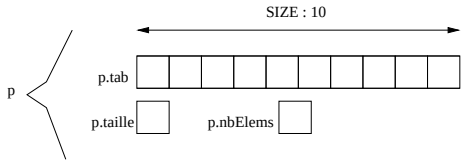
▷ **Axioms :**
- ◇ estVide(créer()) = true
- ◇ estVide(empiler(s,e)) = false
- ◇ sommet(empiler(s,e)) = e
- ◇ dépiler(empiler(s,e)) = s

With s (a stack) and e (an element).

Stack : C implementation (1)

```
#define SIZE 10

struct stpile { int tab[SIZE];
                int taille;   /* the size of the array tab   */
                int nbElems;  /* the number of values stored */
              };

typedef struct stpile pile;   /* pile <=> stack in english...*/
```

If `p` is a `pile` :



Stack : C implementation (2)

It remains to implement the following functions:

▷ `pile creerPile(void);`
▷ `int  estVidePile(pile p);`
▷ `int  estPleinePile(pile p);`
▷ `int  sommetPile(pile p);`
▷ `void empilerPile(pile* p, int e);`
▷ `void depilerPile(pile* p);`

... and also `void afficherPile(pile p);`

One more step towards abstraction: modular programming

A programmer uses a set of functions on a given type.

Question :

Is it possible to hide to the programmer how this data type and the associated functions are implemented ?

Answer :

Yes, by using the **modular programming** and the **modules** !

Interests :
- ◇ Encapsulation of data and treatments (To encapsulate : to hide, to confine,...).
- ◇ Reutilisability of existing modules (without rewriting !)

One more step towards abstraction: modular programming using C language (1)

Example using C language :

▷ We show only to the programmers the interface file ("fichier d'interface").
Thus, in the case of the stacks,
the programmers can read/use a file called : `modulePile.h`
`modulePile.h` $\simeq$ the type + the headers of the functions

▷ We write the implementation file ("fichier d'implémentation")...
but separately from the interface file.
Programmers don't need to know the content of this implementation file.
Thus, in the case of the stacks,
the codes of the functions can be written in a file called : `modulePile.c`
`modulePile.c` $\simeq$ the codes of the functions

One more step towards abstraction: modular programming using C language (2)

Thus, in the interface file ("fichier d'interface") `modulePile.h` :

```
#ifndef _PILE_H_                                    /* modulePile.h */
#define _PILE_H_
#define SIZE 10

struct stpile { int tab[SIZE];
                int taille;     /* the size of the array tab   */
                int nbElems;    /* the number of values stored */
              };

typedef struct stpile pile;

pile creerPile(void);
void afficherPile(pile p);
int  estVidePile(pile p);
int  estPleinePile(pile p);
int  sommetPile(pile p);
void empilerPile(pile* p, int e);
void depilerPile(pile* p);
#endif
```


One more step towards abstraction: modular programming using C language (3)

Thus, in the implementation file ("fichier d'implémentation") `modulePile.c` :

```
#include <stdio.h>                                  /* modulePile.c */
#include <stdlib.h>   /* Because using exit */

#include "modulePile.h"

/*************************************************************************/

pile creerPile(void)
{
 pile p;

 p.taille  = SIZE;
 p.nbElems = 0;

 return p;
}

/*************************************************************************/

/* ... And all the other functions ... */
```


One more step towards abstraction: modular programming using C language (4)

Compilation of the module : `cc -c modulePile.c`

$\Longrightarrow$ `modulePile.o`

... This binary object code can't be executed !

One more step towards abstraction: modular programming using C language (5)

Example using this module :

```
#include <stdio.h>                  /* useModulePile.c */

#include "modulePile.h"

int main(void)
{
 pile p = creerPile();
 int  n;

 printf("Value to put in the stack? ");
 scanf("%d",&n);

 if (!estPleinePile(p)) { empilerPile(&p,n); }

 afficherPile(p);

 /* .... */

 return 0;
}
```

In order to obtain an executable :

```
cc modulePile.o useModulePile.c -o useModulePile
```