MATIÈRE: ALGORITHMIQUE ET LANGAGE C
NIVEAU: 3ÈME GÉNIE INFORMATIQUE

FICHE MODULE

ENSEIGNANTS:
MME. FEYROUZ HAMDAOUI, M. SOFIENE BEN AHMED ET M. MOEZ HAMMAMI

2025/2026 –SU1

PRÉSENTATION

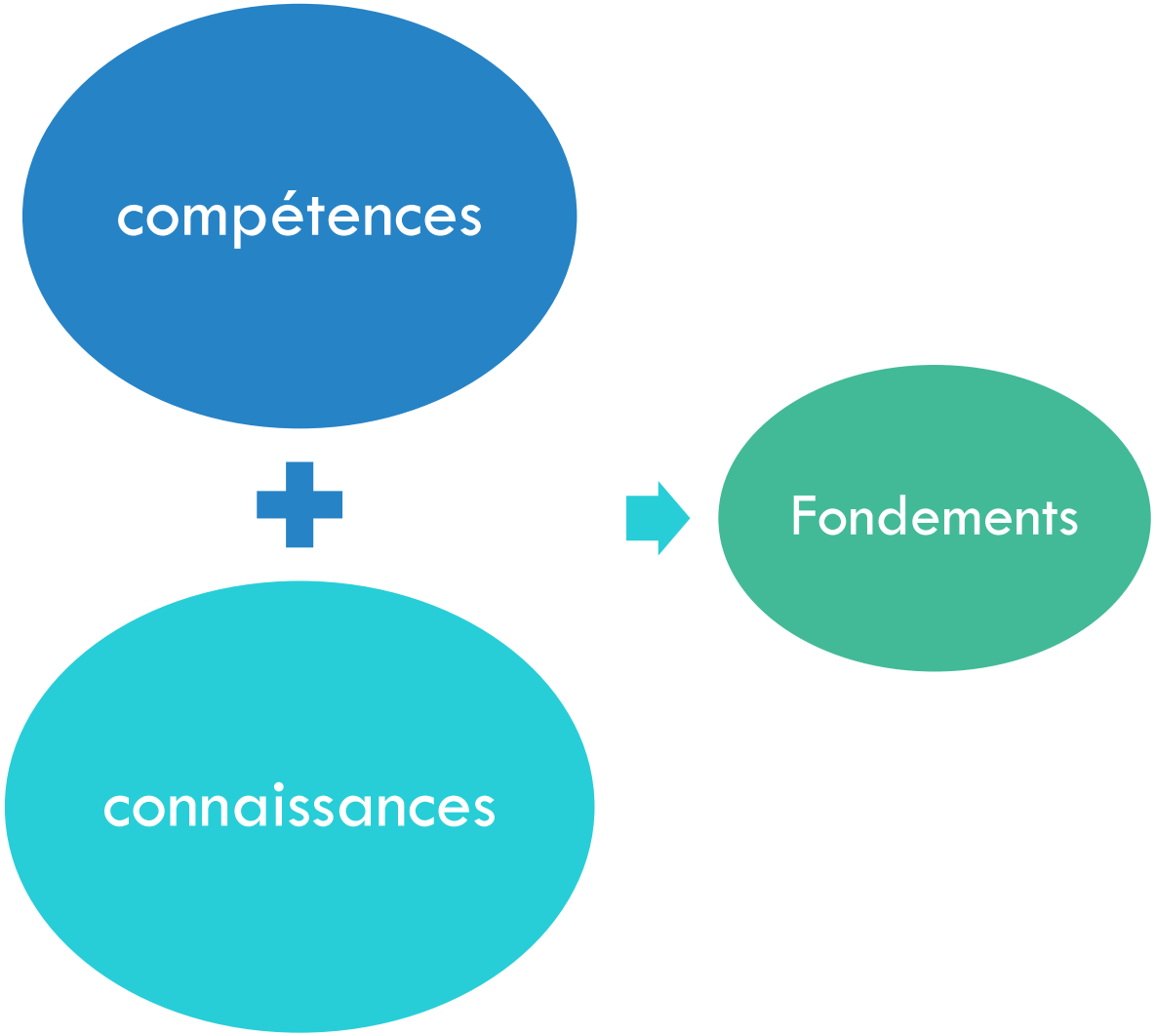
- Volume Horaire/semestre cours /TP intégré : 39H
- Modalité d'évaluation:

  30% Examen TP (Décembre)

  20% Devoir surveillé (Novembre)

  50% Examen semestriel (Janvier)

OBJECTIFS



PLANNING

Partie 1: Notions de base

Introduction + types de données (types intégrés + énuméré + tableaux + enregistrement + exemples) et structures de contrôle (choix + boucles + exemple)

Partie 2: Sous-Progs et Récursivité

- sous-programmes + décomposition + variable locale/globale + paramètre formel/réel + passage de paramètres + exemple(s)
- récursivité
- recherche et tri (insertion, sélection, à bulles, rapide)

Partie 3 : TDA et arbres

- pointeurs + types abstraits de données (liste + pile + file + exemples)
- arbre binaire + exemples

AVANT-GOÛT..

https://visualgo.net/en

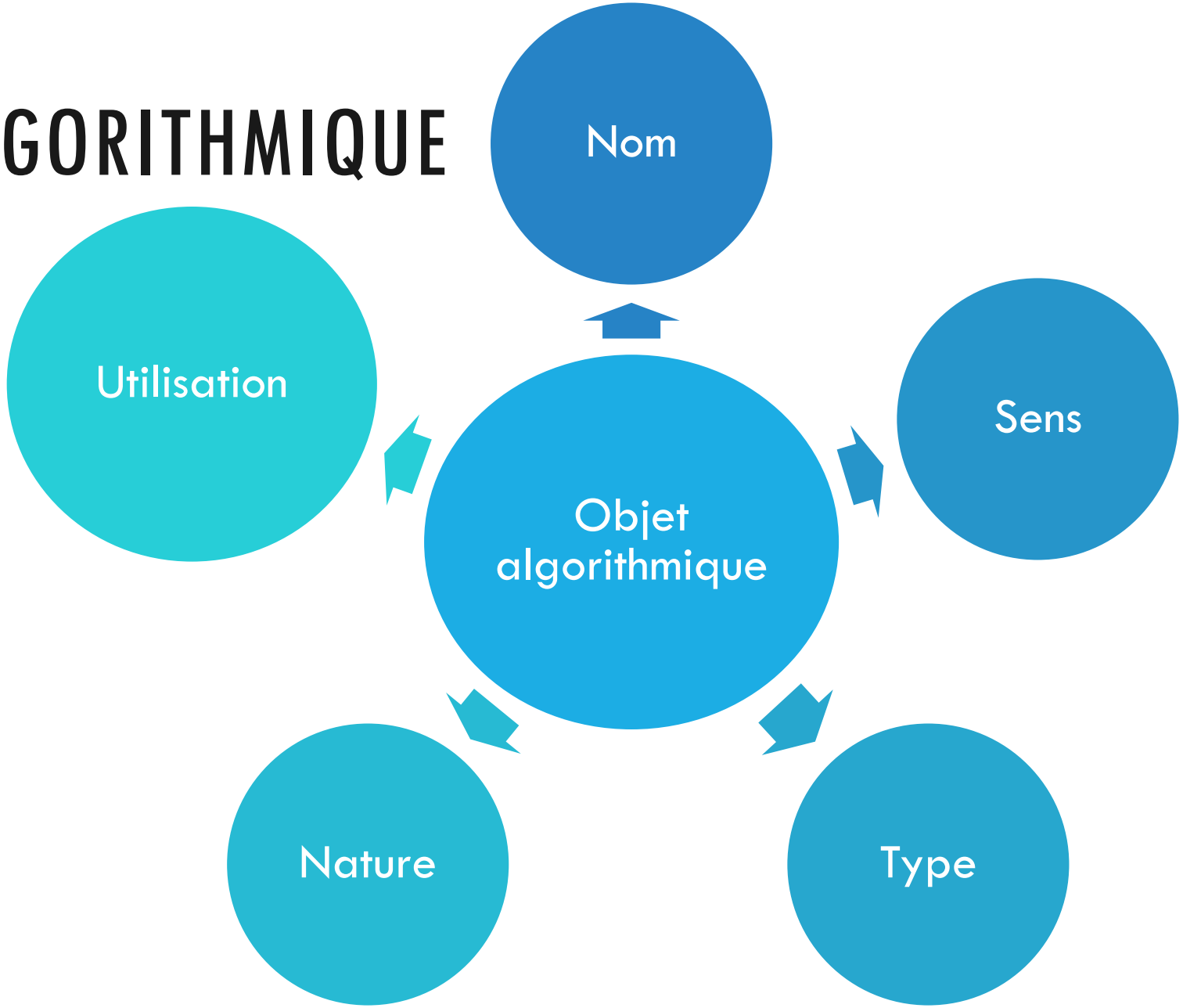
http://algo-visualizer.jasonpark.me/#path=greedy/majority_element/basic

PARTIE 1

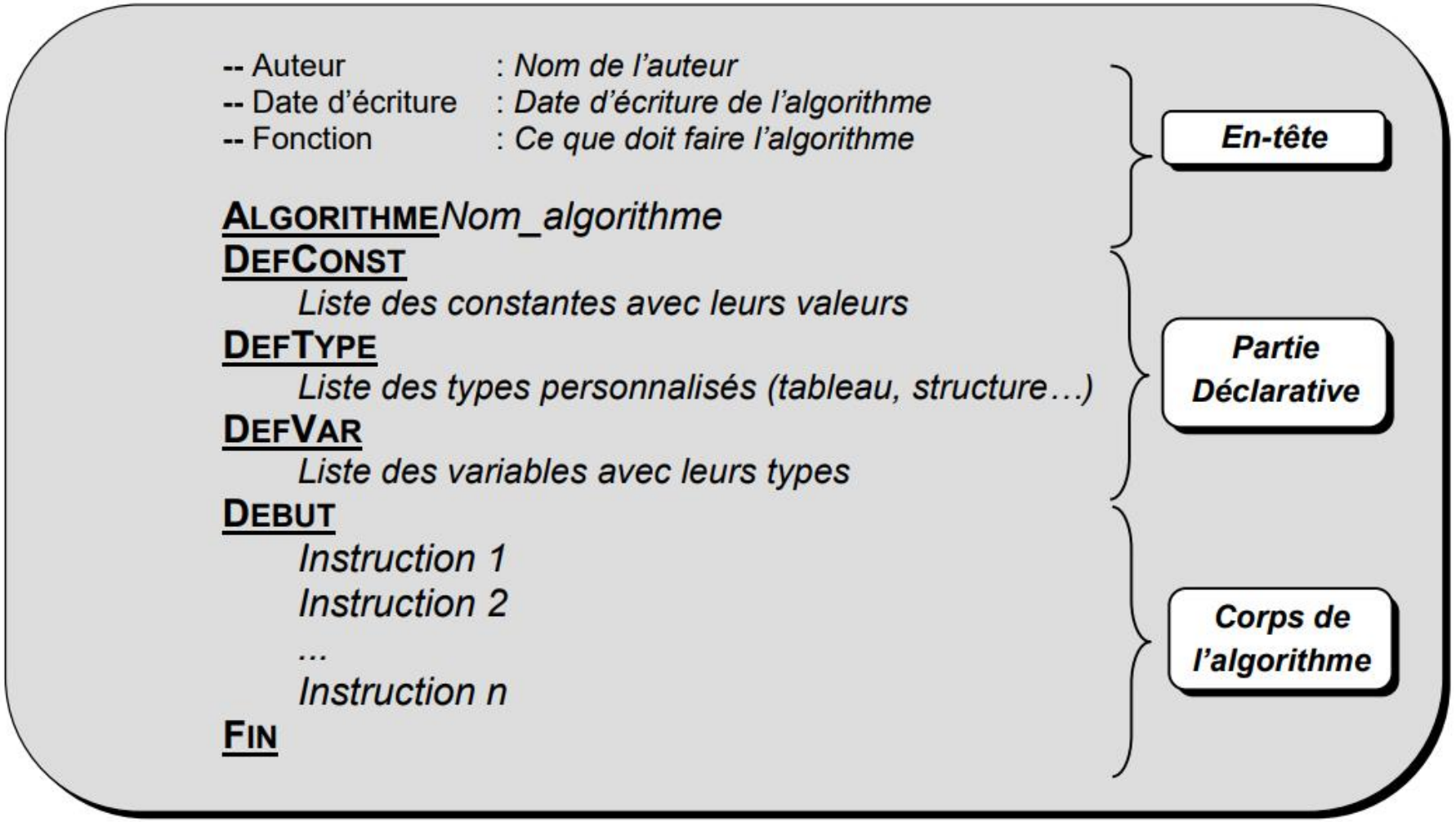
ENSEIGNANTS:
MME. FEYROUZ HAMDAOUI, M. SOFIENE BEN AHMED ET M. MOEZ HAMMAMI

2025/2026 –SU1

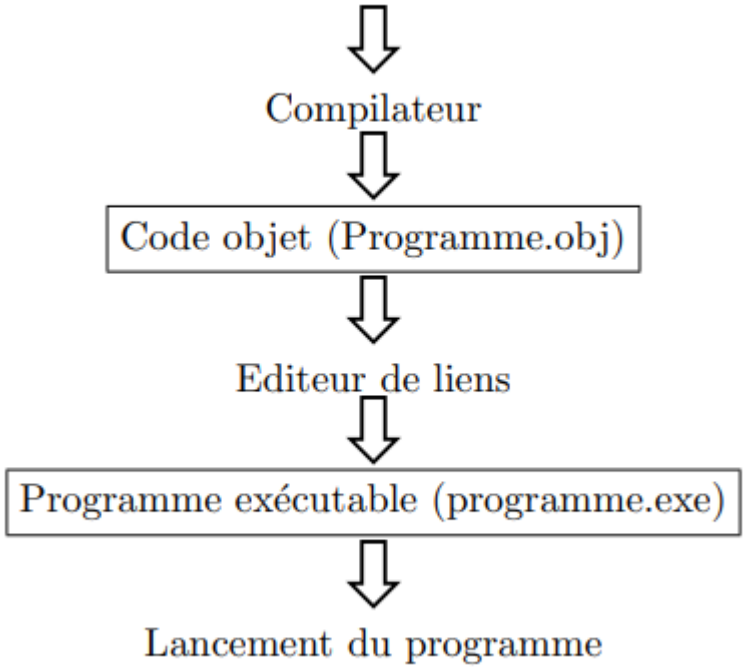
OBJET ALGORITHMIQUE



STRUCTURE D'ALGORITHME



LES PHASES D'UN PROGRAMME EN C



STRUCTURE D'UN PROGRAMME EN C

```
#include<stdio.h>
void main()
{
    <Déclarations>
    <Instructions>
}
```

Remarque :
Chaque programme en C doit renfermer une fonction du nom de "main"(Fonction principale de chaque programme C).
Exemple :

```
void main()
{
     printf("Bonjour");
}
```

LES DIRECTIVES DE PRÉCOMPILATION

Elles commencent toutes par un #.

commande	signification
`#include <stdio.h>`	*permet d'utiliser les fonctions* `printf()` *et* `scanf()`
`#include <math.h>`	*permet d'utiliser les fonctions mathématiques*
`#define PI 3.14159`	*définit la constante* `PI`
`#undef PI`	*à partir de cet endroit, la constante* `PI` *n'est plus définie*

`#ifdef PI`
instructions 1 ...
`#else`
instructions 2 ...
`#endif`
	si la constante `PI` *est définie, on compile les instructions 1, sinon, les instructions 2*

LA FONCTION MAIN()

Elle commence par une accolade ouvrante { et se termine par une accolade fermante }. À l'intérieur, chaque instruction se termine par un point-virgule. Toute variable doit être déclarée.

```
main(){
    int i; /* declaration des variables */    instruction_1;
    instruction_2;
    ...
}
```

Exemple de programme simple :

```
#include <stdio.h>
/* Mon 1er programme en C */
main(){
    printf("Hello world\n");
}
```

LES COMMENTAIRES

Un commentaire est une suite de caractères placés entre des délimiteurs.

Exemple

```
/*Les commentaires documentent les programmes*/
```

Ajoutons un commentaire au programme précédant :

```
#include<stdio.h>
void main()
{
    printf("Bonjour");  /*Affichage d'un message*/
}
```

LES CONSTANTES

Constantes entières `1,2,3,...`

Constantes caractères `'a','A',...`

Constantes chaînes de caractères `"Bonjour"`

Pas de constantes logiques Pour faire des tests, on utilise un entier. 0 est équivalent a faux et tout ce qui est $\neq 0$ est vrai.

LES VARIABLES
NOMS DES VARIABLES

- Le C fait la différence entres les MAJUSCULES et les minuscules.
- Pour éviter les confusions, on écrit les noms des variables en minuscule et on réserve les majuscules pour les constantes symboliques définies par un #define.
- Les noms doivent commencer par une lettre et ne contenir aucun blanc. Le seul caractère spécial admis est le soulignement (_). Il existe un certain nombre de noms réservés (while, if, case, ...), dont on ne doit pas se servir pour nommer les variables. De plus, on ne doit pas utiliser les noms des fonctions pour des variables

DÉCLARATION DES VARIABLES

type	signification	val. min	val. max
`char`	caractère codé sur 1 octet (8 bits)	-2^7	$2^7 - 1$
`short`	entier codé sur 1 octet	-2^7	$2^7 - 1$
`int`	entier codé sur 4 octets	-2^{31}	$2^{31} - 1$
`long`	entier codé sur 8 octets	-2^{63}	$2^{63} - 1$
`float`	réel codé sur 4 octets	$\sim -10^{38}$	$\sim 10^{38}$
`double`	réel codé sur 8 octets	$\sim -10^{308}$	$\sim 10^{308}$

EXEMPLES DE DÉCLARATIONS

On peut faire précéder chaque type par le préfixe unsigned, ce qui force les variables à prendre des valeurs uniquement positives.

déclaration	signification
`int a;`	`a` *est entier*
`int z=4;`	`z` *est entier et vaut 4*
`unsigned int x;`	`x` *est un entier positif (non signé)*
`float zx, zy;`	`zx` *et* `zy` *sont de type réel*
`float zx=15.15;`	`zx` *est de type réel et vaut 15.15*
`double z;`	`z` *est un réel en double précision*
`char zz;`	`zz` *est une variable caractère*
`char zz='a';`	`zz` *vaut* `'a'`

LES TYPES ENTIER

	DEC Alpha	PC Intel (Linux)	
`char`	8 bits	8 bits	caractère
`short`	16 bits	16 bits	entier court
`int`	32 bits	32 bits	entier
`long`	64 bits	32 bits	entier long
`long long`	n.i.	64 bits	entier long (non ANSI)

LES TYPES ENTIER

`signed char`	$[-2^7;2^7[$
`unsigned char`	$[0;2^8[$
`short int`	$[-2^{15};2^{15}[$
`unsigned short int`	$[0;2^{16}[$
`int`	$[-2^{31};2^{31}[$
`unsigned int`	$[0;2^{32}[$
`long int` (DEC alpha)	$[-2^{63};2^{63}[$
`unsigned long int` (DEC alpha)	$[0;2^{64}[$

LES OPÉRATEURS (1/4)

Opérateurs arithmétiques

+	addition
-	soustraction
*	multiplication
/	division (entière et rationnelle!)
%	modulo (reste d'une div. entière)

Opérateurs logiques

&&	et logique (and)
||	ou logique (or)
!	négation logique (not)

Opérateurs de comparaison

==	égal à
!=	différent de
<, <=, >, >=	plus petit que, ...

Opérations logiques

Les résultats des opérations de comparaison et des opérateurs logiques sont du type Int

- la valeur 1 correspond à la valeur booléenne vraie
- la valeur 0 correspond à la valeur booléenne fausse

!Les opérateurs logiques considèrent toute valeur différente de zéro comme vrai et zéro comme faux

LES OPÉRATEURS (2/4)

	32 && 2.3	→	1
	!65.34	→	0
	0||!(32 > 12)	→	0

Opérateurs d'affectation

+=	ajouter à
-=	diminuer de
*=	multiplier par
/=	diviser par
%=	modulo

I++ ou ++I	pour l'incrémentation	(augmentation d'une unité)
I-- ou --I	pour la décrémentation	(diminution d'une unité)

Classes de priorités

Priorité 1 (la plus forte):	()
Priorité 2:	! ++ --
Priorité 3:	* / %
Priorité 4:	+ -
Priorité 5:	< <= > >=
Priorité 6:	== !=
Priorité 7:	&&
Priorité 8:	||
Priorité 9 (la plus faible):	= += -= *= /= %=

LES OPÉRATEURS (1/4)

Le premier opérateur à connaître est l'**affectation "="**. Exemple : {a=b+" ;} Il sert à mettre dans la variable de **gauche** la valeur de ce qui est à droite. Le membre de droite est d'abord évalué, et ensuite, on affecte cette valeur à la variable de gauche. Ainsi l'instruction `i=i+1` a un sens.

Pour les opérations dites naturelles, on utilise les opérateurs `+`, `-`, `*`, `/`, `%`.

`%` est l'opération modulo : `5%2` est le reste de la division de 5 par 2. `5%2` est donc égal à `1`.

Le résultat d'une opération entre types différents se fait dans le type le plus haut. Les types sont classés ainsi :

```
char < int < float < double
```

Par ailleurs, l'opération `'a'+1` a un sens, elle a pour résultat le caractère suivant à `a` dans le code ASCII.

LES OPÉRATEURS (3/4)

Leur utilisation devient délicate quand on les utilise avec d'autres opérateurs. Exemple :

```
int i=1 , j;
j=i++;          /* effectue d'abord j=i et ensuite i=i+1 */
                /* on a alors j=1 et i=2 */
j=++i;          /* effectue d'abord i=i+1 et ensuite j=i */
                /* on a alors j=2 et i=2 */
```

Quand l'opérateur ++ est placé avant une variable, l'incrémentation est effectuée en premier. L'incrémentation est faite en dernier quand ++ est placé aprés la variable. Le comportement est similaire pour – –.

INSTRUCTIONS ELÉMENTAIRES

- Affectation: x ← 5
- lecture: lire (x)
- Ecriture: écrire (x)

LA FONCTION PRINTF()

Elle sert à afficher à l'écran la chaîne de caractère donnée en argument, c'est-à-dire entre parenthèses.

`printf("Bonjour\n");` affichera Bonjour à l'écran.

Certains caractères ont un comportement spécial :

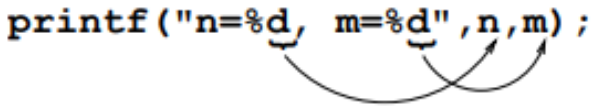
`\n`	retour à la ligne
`\b`	n'imprime pas la lettre précédente
`\r`	n'imprime pas tout ce qui est avant
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\"`	"
`\'`	'
`\?`	?
`\!`	!
`\\`	\

Mais `printf()` permet surtout d'afficher à l'écran la valeur d'une variable :

LA FONCTION PRINTF()

```
main(){
    int n=3, m=4;
    printf("%d",n); /* affiche la valeur de n au format d (decimal) */
    printf("n=%d",n); /* affiche 'n=3' */
    printf("n=%d, m=%d",n,m); /* affiche 'n=3, m=4' */
    printf("n=%5d",n); /* affiche la valeur de n sur 5 caracteres : 'n=     3'
*/
}
```

Le caractère `%` indique le format d'écriture à lécran. Dès qu'un format est rencontré dans la chaîne de caractère entre " ", le programme affiche la valeur de l'argument correspondant.



ATTENTION ! le compilateur n'empêche pas d'écrire un `char` sous le format d'un réel ⇒ affichage de valeurs délirantes. Et si on écrit un `char` avec un format décimal, on affiche la valeur du code ASCII du caractère.

LA FONCTION PRINTF()

TAB. 3.1 – Tableau des formats utilisables

`%d`	`integer`	*entier (décimal)*
`%u`	`unsigned`	*entier non signé (positif)*
`%hd`	`short`	*entier court*
`%d`	`long`	*entier long*
`%f`	`float`	*réel, notation avec le point décimal (ex. 123.15)*
`%e`	`float`	*réel, notation exponentielle (ex. 0.12315E+03)*
`%lf`	`double`	*réel en double précision, notation avec le point décimal*
`%le`	`double`	*réel en double précision, notation exponentielle*
`%c`	`char`	*caractère*
`%s`	`char`	*chaîne de caractères*

Remarque : une chaîne de caractères est un tableau de caractères. Elle se déclare de la façon suivante : `char p[10];`. Mais nous reviendrons sur la notion de tableau plus tard.

LA FONCTION SCANF()

Dans un programme, on peut vouloir qu'une variable n'ait pas la même valeur à chaque exécution. La fonction `scanf()` est faite pour cela. Elle permet de lire la valeur que l'utilisateur rentre au clavier et de la stocker dans la variable donnée en argument.

Elle s'utilise ainsi :

```
main(){
    int a;
    scanf("%d",&a);
}
```

On retrouve les formats de lecture précisés entre " " utilisés pour `printf()`. Pour éviter tout risque d'erreur, on lit et on écrit une même variable avec le même format.

Le **&** est indispensable pour le bon fonctionnement de la fonction. Il indique l'adresse de la variable, mais nous reviendrons sur cette notion d'adresse quand nous aborderons les pointeurs.

ACTIVITÉ 1

- Ecrire un algorithme qui permet de former puis d'afficher un entier r de quatre chiffres à partir de deux entiers m et n.
- On suppose qu'ils sont strictement positifs et formés chacun de deux chiffres et ceci en intercalant le nombre n entre les deux chiffres de m (sans utiliser les chaînes de caractères).
- Exemple :

pour m = 21 et n = 81, l'entier r sera égal à 2811.

STRUCTURES CONDITIONNELLES

```
Si (expression_logique)
    Alors   bloc d'instructions1
    Sinon   bloc d'instructions2
FinSi
```

```
if (expression) then {instruction;}
/* Si expression est vraie alors instruction est executee */



if (expression) {
   instruction 1;
   } else {
   instruction 2;
}
/* Si expression est vraie alors l'instruction 1 est executee */
/* Sinon, c'est l'instruction 2 qui est executee */
```

STRUCTURES CONDITIONNELLES

```
SELON Sélecteur FAIRE
        Liste_valeurs1   : instructions exécutées si Sélecteur est inclus dans Liste_valeurs1
        Liste_valeurs2   : instructions exécutées si Sélecteur est inclus dans Liste_valeurs2
        ...
        Liste_valeursN   : instructions exécutées si Sélecteur est inclus dans Liste_valeursN
        [SINON           : instructions à exécuter si rien ne correspond]
FINSELON
```

STRUCTURES CONDITIONNELLES

Cette instruction s'utilise quand un **entier** ou un caractère prend un nombre fini de valeurs et que chaque valeur implique une instruction différente.

```
switch(i) {
   case 1 : instruction 1; /* si i=1 on exécute l'instruction 1 */
          break; /* et on sort du switch */
   case 2 : instruction 2; /* si i=2 ... */
          break;
   case 10 : instruction 3; /* si i=10 ... */
          break;
   default : instruction 4; /* pour les autres valeurs de i */
          break;
}
```

STRUCTURES CONDITIONNELLES

ATTENTION ! on peut ne pas mettre les `break ;` dans les blocs d'instructions. Mais alors on ne sort pas du `switch`, et on exécute toutes les instructions des `case` suivants, jusqu'à rencontrer un `break ;`.

Si on reprend l'exemple précédent en enlevant tous les `break`, alors

⋆ si i=1 on exécute les instructions 1, 2, 3 et 4

⋆ si i=2 on exécute les instructions 2, 3 et 4

⋆ si i=10 on exécute les instructions 3 et 4

⋆ pour toutes les autres valeurs de i, on n'exécute que l'instruction 4.

ACTIVITÉ 2

- Un nombre n est dit **valide partiel** s'il est multiple de l'un des chiffres qui le composent. Exemple : n =52 est divisible par 2 et non divisible par 5.
- Ecrire un algorithme qui permet de saisir un entier n, on suppose qu'il est formé de deux chiffres, et indique s'il est : valide partiel, **non valide partiel** ou valide total.
- NB : Un nombre n est dit **valide total** s'il est multiple de deux chiffres qui le composent.

STRUCTURES ITÉRATIVES

POUR *Compteur* ***de*** *Début* **à** *Fin* [**PAS** *Incrément*]**FAIRE**
Instruction 1
...
Instruction n
FINPOUR

Elle permet d'exécuter des instructions plusieurs fois sans avoir à écrire toutes les itérations. Dans ce genre de boucle, on doit savoir le nombre d'itérations avant d'6etre dans la boucle. Elle s'utilise ainsi :

```
for (i=0; i<N; i++) {
    instructions ...;
}
```

Dans cette boucle, `i` est le **compteur**. Il ne doit pas être modifié dans les instuctions, sous peine de sortie de boucle et donc d'erreur.

⋆ La $1^{ère}$ instruction entre parenthèses est l'initialisation de la boucle.

⋆ La $2^{ème}$ est la condition de sortie de la boucle : tant qu'elle est vraie, on continue la boucle. ⋆ Et la $3^{ème}$ est l'instruction d'itération : sans elle, le compteur reste à la valeur initiale et on ne sort jamais de la boucle.

STRUCTURES ITÉRATIVES

TANTQUE *(expression_logique)* **FAIRE**
Instruction 1
Instruction 2
...
Instruction n
FINTANTQUE

Contrairement à la boucke `for`, on n'est pas obligés ici de connaître le nombre d'itérations. Il n'y a pas de compteur.

```
while (expression) {
    instructions ...;
}
```

L'expression est évaluée à chaque itération. Tant qu'elle est vraie, les instructions sont exécutées. Si dès le début elle est fausse, les instructions ne sont jamais exécutées.

STRUCTURES ITÉRATIVES

ATTENTION ! Si rien ne vient modifier l'expression dans les instructions, on a alors fait une boucle infinie : `while (1) { instructions }` en est un exmple.
Exemple d'utilisation :

```
#include <stdio.h>
main(){
    float x,R;
    x=1.0;
    R=1.e5;
    while (x < R) {
      x = x+0.1*x;
      printf("x=%f",x);
    }
}
```

STRUCTURES ITÉRATIVES

REPETER
Instruction 1
...
Instruction n
JUSQU'A *(expression_logique)*

À la différence d'une boucle `while`, les instructions sont exécutées au moins une fois : l'expression est évaluée en fin d'itération.

```
do {
   instructions ...;
} while (expression)
```

Les risques des faire une boucle infinie sont les mêmes que pour une boucle `while`.

LES STRUCTURES ITÉRATIVES: LES INSTRUCTIONS BREAK ET CONTINUE

`break` fait sortir de la boucle.

`continue` fait passer la boucle à l'itération suivante.

ACTIVITÉ 3

- Ecrire un algorithme qui saisit un nombre entier nb composé de quatre chiffres puis calcule le **chiffre de chance** correspondant à ce nombre de la façon suivante :

- Faire la somme de tous les chiffres qui composent le nombre saisi.
- Si le nombre calculé est composé de plus d'un chiffre, faire la somme des chiffres qui le composent.
- Répéter ce procédé jusqu'à avoir un nombre composé d'un seul chiffre.

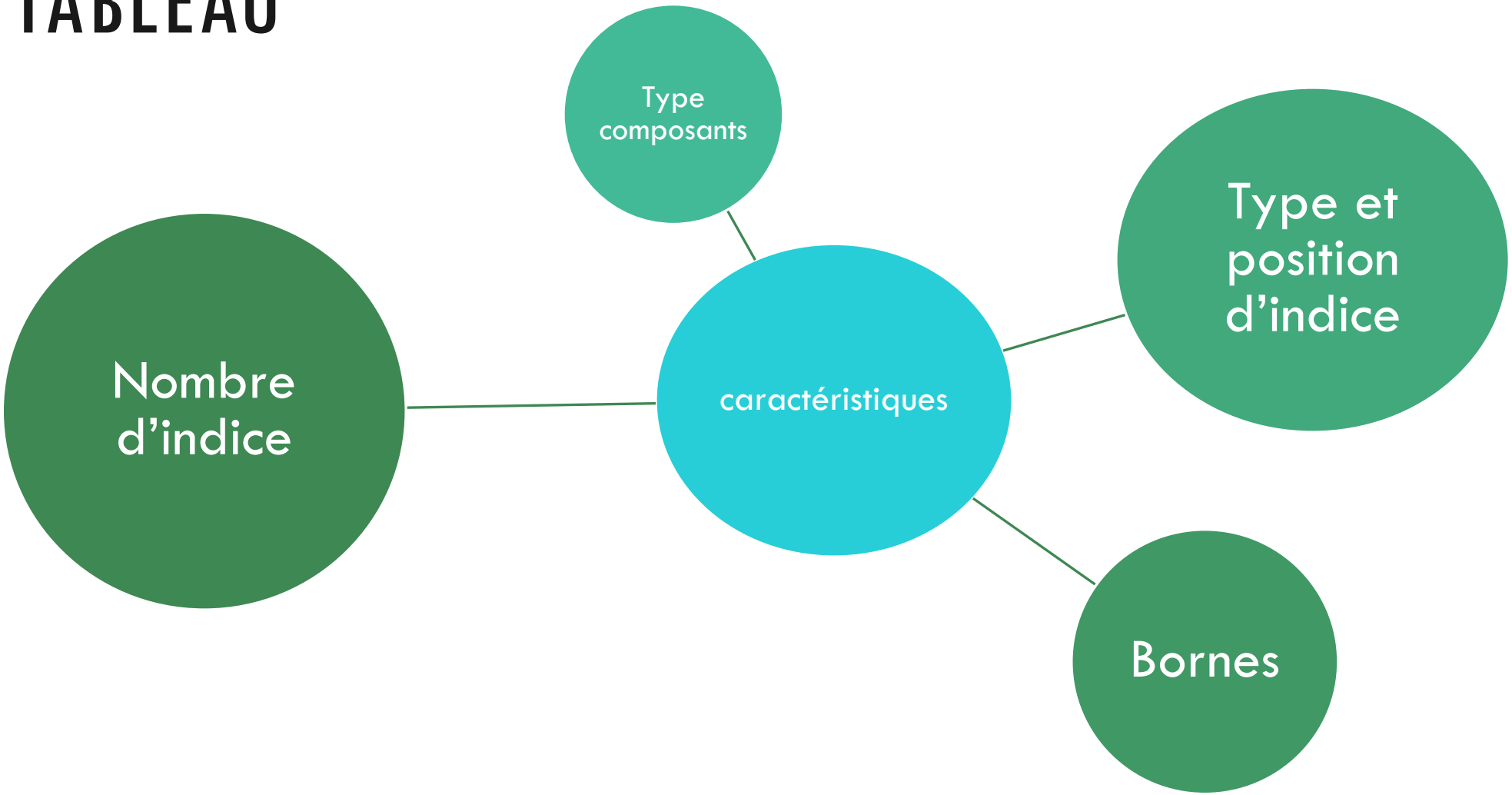
Pour **nb = 9569**, on aura les sommes suivantes :

9 + 5 + 6 + 9 = 29

2 + 9 = 11

1 + 1 = 2 **est le chiffre de chance**

TABLEAU


Type composants
Nombre d'indice
caractéristiques
Type et position d'indice
Bornes

VECTEUR

DefConst

Binf (Entier) = … -- *Borne inférieure*

Bsup (Entier) = … -- *Borne Supérieure*

DefType

Dimension = … -- *Type scalaire (énuméré ou intervalle)*

Nom_Type1 = **Tableau** [*Binf..Bsup*] **de** *Type_Base*

Nom_Type2 = **Tableau** [*Dimension*] **de** *Type_Base*

DefVar

MATRICE

DefType

Dim1 = … -- *type intervalle ou type énuméré*

Dim2 = … -- *type intervalle ou type énuméré*

Nom_Type = **Tableau** [*Dim1*, *Dim2*] **de** *Type_Base*

DefVar

T (Nom_Type) -- Déclaration d'une variable T de type Nom_Type

LES TABLEAUX: DÉCLARATION

Comme une variable, on déclare son type, puis son nom. On rajoute le nombre d'éléments du tableau entre crochets `[ ]` :

`float tab[5];` est un tableau de 5 flottants.

`int tablo[8];` est un tableau de 8 entiers.

ATTENTION !

⋆ Les numéros des éléments d'un tableau de n éléments vont de 0 a $n-1$.

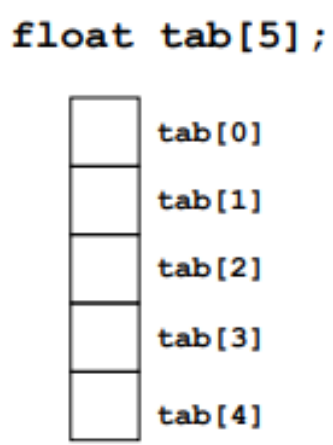
⋆ La taille du tableau doit être une constante (par opposition à variable), donc `int t1[n];` où `n` serait une variable déjà déclarée est une mauvaise déclaration. Par contre si on a défini `#define N 100` en directive, on peut déclarer `int t1[N];` car `N` est alors une constante.

On peut initialiser un tableau lors de sa déclaration :

```
float tab[5] = { 1, 2, 3, 4, 5}; /* init. de tous les éléments de tab */
float toto[10] = {2, 4, 6}; /* equ. à toto[0]=2; toto[1]=4; toto[2]=6; */
                            /* les autres éléments de toto sont mis à 0. */
```

LES TABLEAUX: UTILISATION

Comme shématisé ci-contre, on accède à l'élément *i* d'un tableau en faisant suivre le nom du tableau par le numéro *i* entre crochets. Un élément de tableau s'utilise comme une variable quelconque. On peut donc faire référence à un élément pour une affectation : `x=tab[2]`, `tab[3]=3`, ou dans une expression : `if (tab[i] < 0)`.

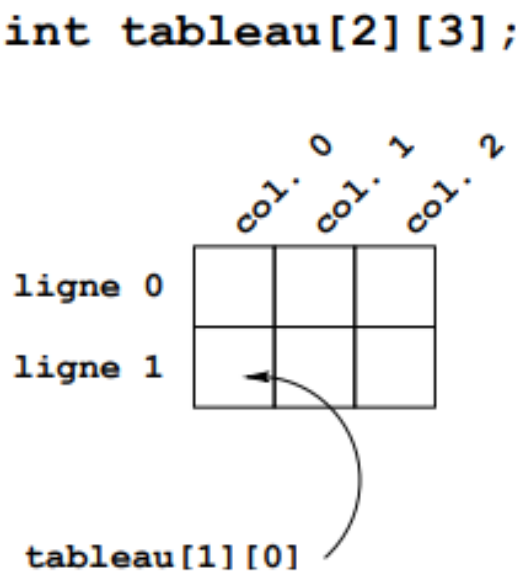


LES TABLEAUX: TABLEAU À 2 DIMENSIONS

Un tableau à 2 dimensions est similaire à une matrice. C'est en fait un tableau de tableau à 1 dimension, il se déclare donc de la façon suivante :

```
int tableau[2][3]; /* tableau de 2 lignes et 3 colonnes */
```

Comme il est shématisé ci-contre, `tableau[i][j]` fait référence à l'élément de la ligne *i* et de la colonne *i* de `tableau`. Tout comme un élément d'un tableau à 1 dimension, `tableau[i][j]` se manipule comme n'importe quelle variable.



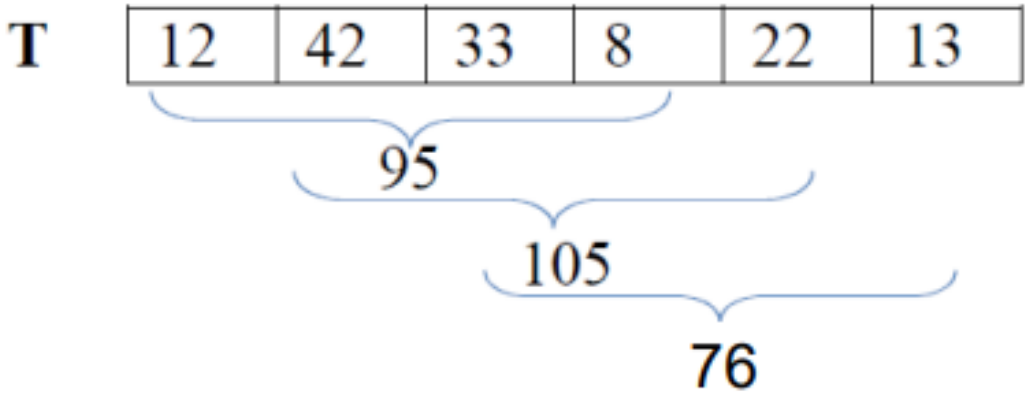
ACTIVITÉ 4 (1/2)

Soit T un tableau de N entiers positifs (5 <=N<=30) et un entier K tel que (1<K<N). On se propose de former un tableau TS par les sommes des K éléments consécutifs du tableau T, de telle façon que TS[i] contiendra la somme des K éléments consécutifs comptés à partir du ième élément du tableau T.
Ecrire un algorithme permettant de remplir un tableau T par N entiers positifs, de saisir K puis de former et d'afficher le tableau TS.

ACTIVITÉ 4 (2/2)

Exemple :

Si N = 6 et K=4 et que le tableau T contient les éléments suivants :



Le programme affichera le tableau TS suivant :

TS	95	105	76

ACTIVITÉ 5

Nous voulons saisir des entiers successifs dans une matrice sous forme spirale.

Exemple :

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

CHAINES DES CARACTÈRES

- Ch: chaine
- Ch: chaine [50]

- La lecture, l'écriture et l'affectation
- Les opérateurs relationnels: (=, ≠, >, ≥, <, ≤)
- Ch[i]

CHAINES DES CARACTÈRES: OPÉRATIONS

- X ←Long (« Tunisie »)
- Ch1 ← Majus (« Tunisie »)
- Ch ← Concat (« Tunisie », « 2020 »)
- S ← Copier (ch, pos, n)
- S ← Copier (Ch, 8, 6)
- P ← position(« Tunisie », « i »)
- Valeur (« 2020 », n, err)
- Valeur (« Tunisie 2020 », n, err)
- Ch ← chaine (2020)

LES TABLEAUX:
CAS D'UN TABLEAU DE CARACTÈRES

Un tableau de caractères est en fait une **chaîne de caractères**. Son initialisation peut se faire de plusieurs façons :

```
char p1[10]='B','o','n','j','o','u','r';
char p2[10]="Bonjour"; /* init. par une chaîne litterale */
char p3[]="Bonjour"; /* p3 aura alors 8 éléments */
```

ATTENTION ! Le compilateur rajoute toujours un caractère *null* à la fin d'une chaîne de caractères. Il faut donc que le tableau ait au moins un élément de plus que la chaîne litterale.

BIBLIOTHÈQUE <STRING.H> EN C

Les principales fonctions sur les chaînes de caractères sont :

- `strlen()` : calcule la longueur d'une chaîne.
- `strcmp()/strncmp()` : compare deux chaînes.
- `strcoll()` : compare deux chaînes selon la *locale*.
- `strchr()` : recherche la première occurrence d'un caractère.
- `strrchr()` : recherche la dernière occurrence d'un caractère.
- `strpbrk()` : recherche la première occ d'un ensemble de car.
- `strstr()` : recherche la première occurrence d'une chaîne.
- `strcpy()/strncpy()` : recopient une chaîne dans une autre.
- `strcat()/strncat()` : concatènent une chaîne à une autre.
- `strspn()/strcspn()` : calculent la longueur d'un préfixe.

ACTIVITÉ 6

On veut écrire un algorithme permettant de lire un mot **mot** et d'afficher les chaînes de caractères dérivées :

- La chaîne formée par le **premier** et le **dernier** caractères de **mot**.
- La chaîne formée par les **deux premiers** (sens normal) et les **deux derniers** (sens inverse) caractères de **mot**.
- etc.

Exemple :

Pour **mot = "Weekend"**, les chaînes dérivées sont :

```
=> Wd
=> Wedn
=> Weedne
=> Weekdnek
=> Weekedneke
=> Weekendnekee
=> WeekenddnekeeW
```

ACTIVITÉ 7 (1/2)

Ce jeu se joue à deux. Le premier joueur choisit un mot que le deuxième joueur est chargé de dévoiler en un nombre fini d'essais. Le jeu se déroule de la façon suivante :

- Le premier joueur masque le mot choisi **mot** de **n** lettres par une chaîne **S** de **n** tirets.
- Le deuxième joueur propose une lettre. Si celle-ci figure dans le mot **mot**, elle sera dévoilée dans la chaîne **S** à la place de chaque **tiret** qui lui correspond.
- Chaque proposition d'une lettre compte un essai. Le nombre d'essais maximum est fixé à **2*n**.

Ecrire un algorithme "Devinette" simulant ce jeu.

ACTIVITÉ 7 (2/2)

Exemple :

```
mot ← "P O U S S E R"
S   ← "- - - - - - -"
```

```
Essai N° 1 :   'A'    S ← "-------"    nb ← 0 Nombre de lettres trouvées
Essai N° 2 :   'E'    S ← "-----E-"    nb ← 1
Essai N° 3 :   'S'    S ← "---SSE-"    nb ← 3
Essai N° 4 :   'e'    S ← "---SSE-"    nb ← 3
Essai N° 5 :   'I'    S ← "---SSE-"    nb ← 3
Essai N° 6 :   'O'    S ← "-O-SSE-"    nb ← 4
Essai N° 7 :   'p'    S ← "PO-SSE-"    nb ← 5
Essai N° 8 :   'u'    S ← "POUSSE-"    nb ← 6
Essai N° 9 :   'R'    S ← "POUSSER"    nb ← 7   (Stop)
```

Le mot est dévoilé, c'est le bon résultat. La longueur du mot étant de **7**, le nombre d'essais autorisé est de **2*7=14**. On a fait seulement **9** essais.

ENREGISTREMENT

Def Type

Structure *Type_Article*

Champ1 : Type_Champ1

Champ2 : Type_Champ2

...

ChampN : Type_ChampN

Fin Structure

Def Var

Art (Type_Article) -- Déclaration d'une variable Art de type Type_Article

ENREGISTREMENT

La définition d'un type structuré se fait avec le mot-clé `struct`.

La syntaxe est la suivante :

```
struct TypeTag {
  Type1 field_1;
  ...
  TypeN field_n;
} var_1, ..., var_m;
```

- Elle déclare qu'il existe un type structuré `struct TypeTag`.
- elle définit ce type comme un agrégat de n champs `<field_1,...,field_n>`, où `field_k` est de type `TypeK`.
- elle déclare m variables `var_1,...,var_m` de ce type.

ENREGISTREMENT

On accède au champ `field` d'une variable `var` par la syntaxe :

```
var.field
```

. . . que ce soit pour écrire la valeur du champ ou la lire :

```
int main (void) {
  struct Point p;
  p.x= 5.0; p.y= 8.0;                    // set fields
  printf ("p is <%f, %f>\n", p.x, p.y); // get fields

  struct FullName name;
  strcpy (name.first, "Regis");
  strcpy (name.last, "Barbanchon");
  printf ("name is <%s %s>\n", name.first, name.last);
  return 0;
}
```

La sortie produite par cet exemple est :

```
p is <5.0, 8.0>
name is <Regis Barbanchon>
```

ACTIVITÉ 8

Écrire un algorithme qui permet de calculer le temps écoulé **diff** (en minutes) entre deux temps **tm1** et **tm2** (en heures et minutes) saisis au clavier. On convertit chaque temps en une valeur entière correspondant au nombre de minutes écoulées depuis minuit. L'algorithme doit effectuer les opérations suivantes :

a) Saisie de deux structures temps **tm1** et **tm2**,

```
Structure Temps
  Hh: Entier  //heures  ∈ [0, 23]
  mm: Entier  //minutes ∈ [0, 59]
FinStructure
```

b) Conversion de deux temps **tm1** et **tm2** en minutes (**tmm1** et **tmm2**),

c) Calcul de la différence entre **tmm1** et **tmm2** en tenant compte du cas particulier de temps qui se situent de part et d'autre de minuit.

- Cas 1 : Si **tm1** = 15:45 et **tm2** = 17:30 Alors temps écoulé : **diff** = 105 minutes
- Cas 2 : Si **tm1** = 23:30 et **tm2** = 00:45 Alors temps écoulé : **diff** = 75 minutes

d) Affichage du résultat : *Nombre de minutes écoulées entre ces deux temps = **diff***