



ALGORITHMIQUE ET PROGRAMMATION EN LANGAGE C

Mohamed Ali Mahjoub

medalimahjoub@gmail.com

mohamedali.mahjoub@eniso.u-sousse.tn

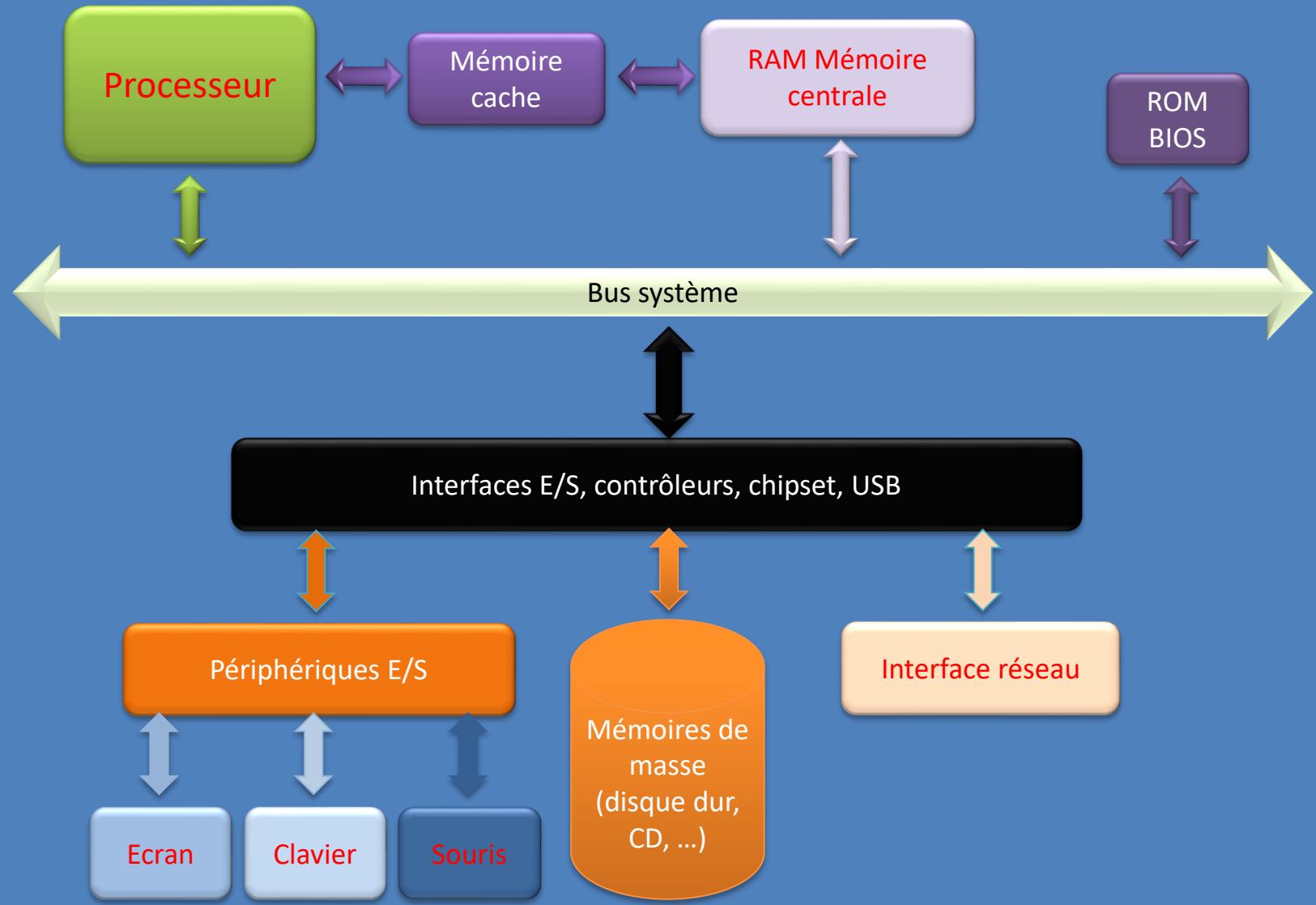
Premières Années EI

Eniso 2020-2021

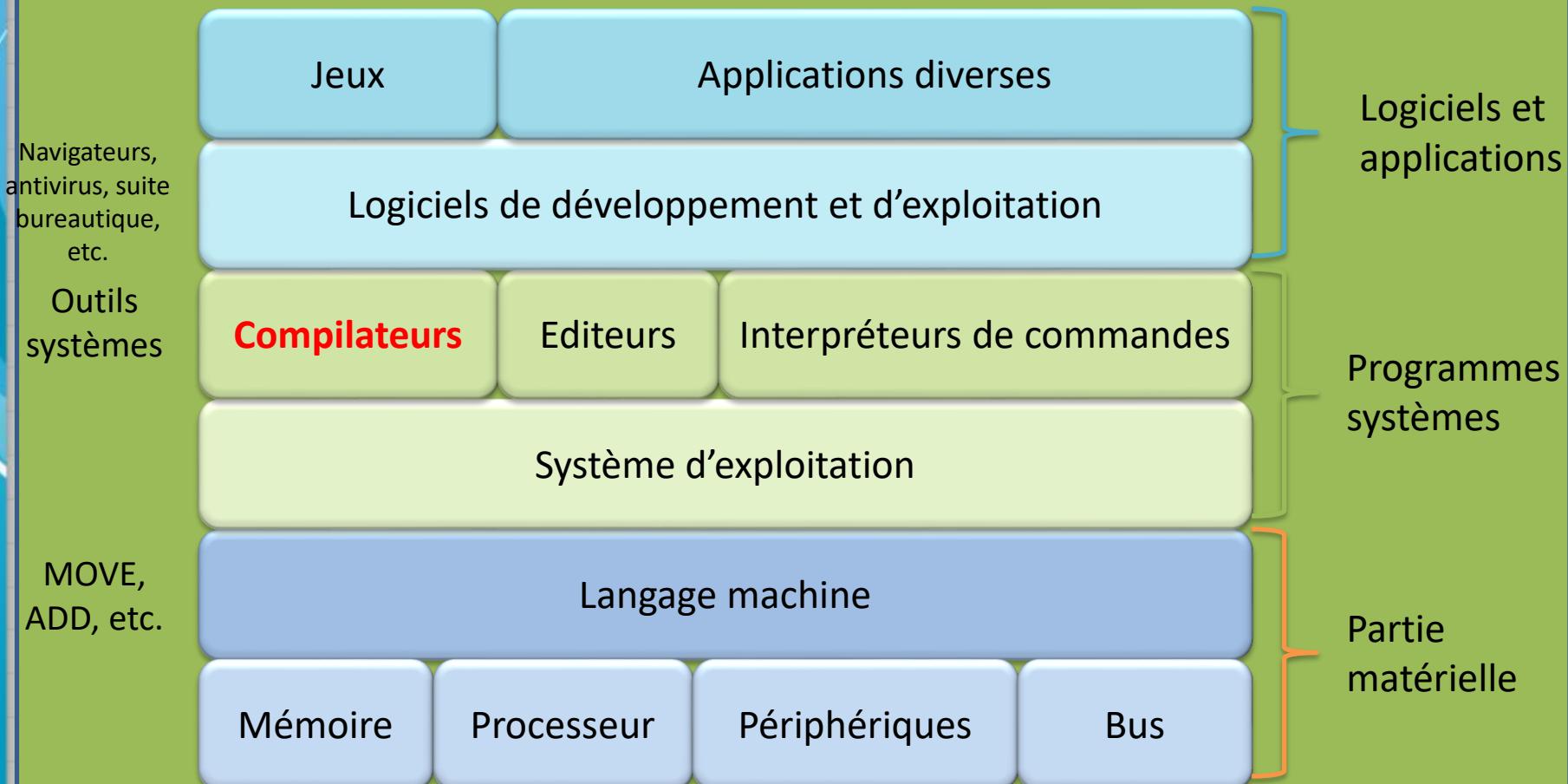
Historique

- 1947 : premier ordinateur ENIAC
 - Années 1950 premiers ordinateurs commercialisés : les transistors
 - Années 1960 mini ordinateurs : les circuits intégrés
 - Années 1970 micro ordinateurs : VLSI (very large scale integration)
 - 1980 : Réseaux locaux
 - 1990 : Internet web 1.0
 - 2000 : Internet web 2.0 :web collaboratif
 - 2010 : web 3.0 Internet of Things
 - Villes intelligentes, maisons intelligentes
-
- Programmation procédurale (**C**, cobol, pascal...)
 - Programmation orientée objet (**C++**, java, python, ...)

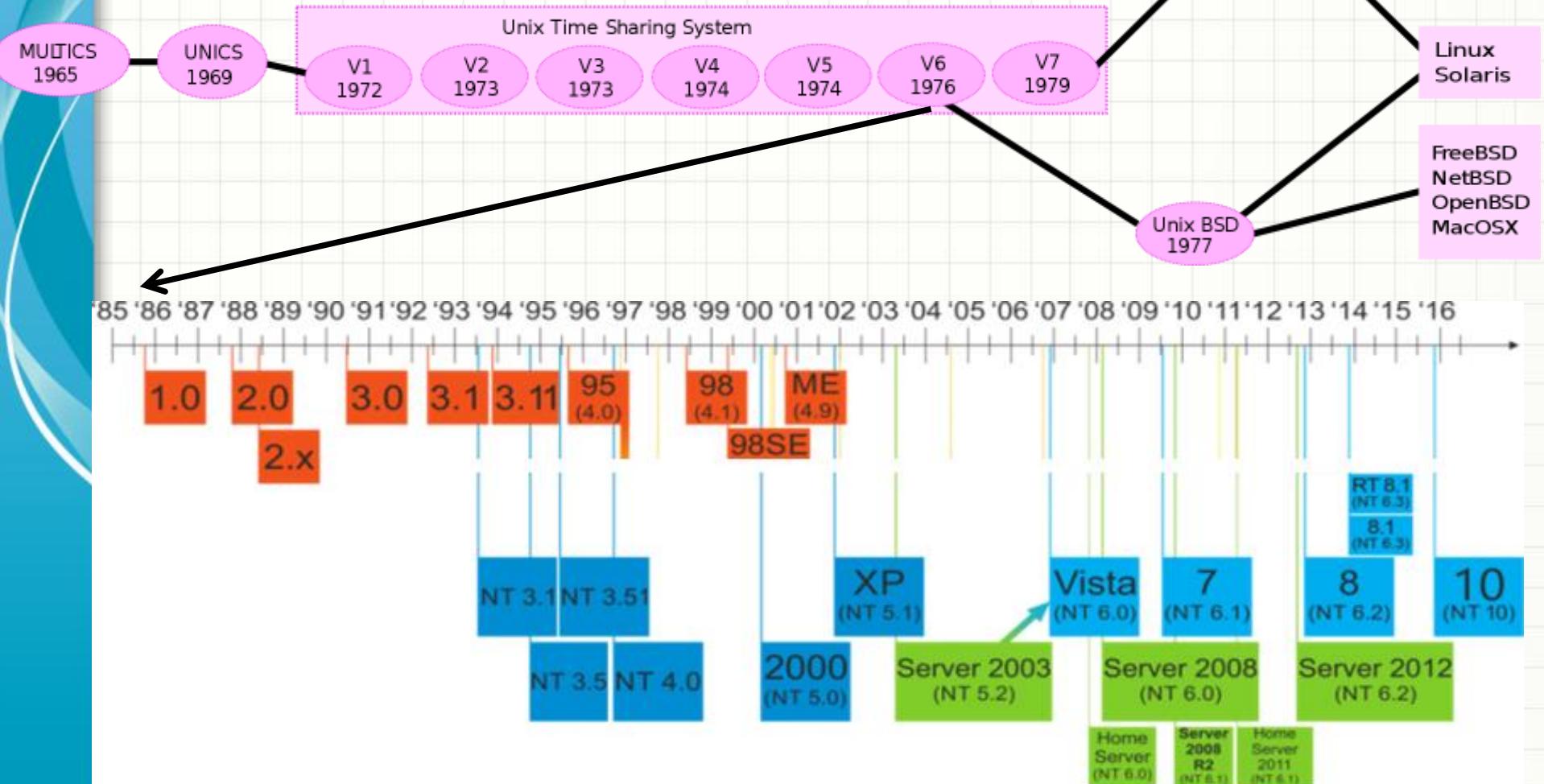
Structure d'un ordinateur



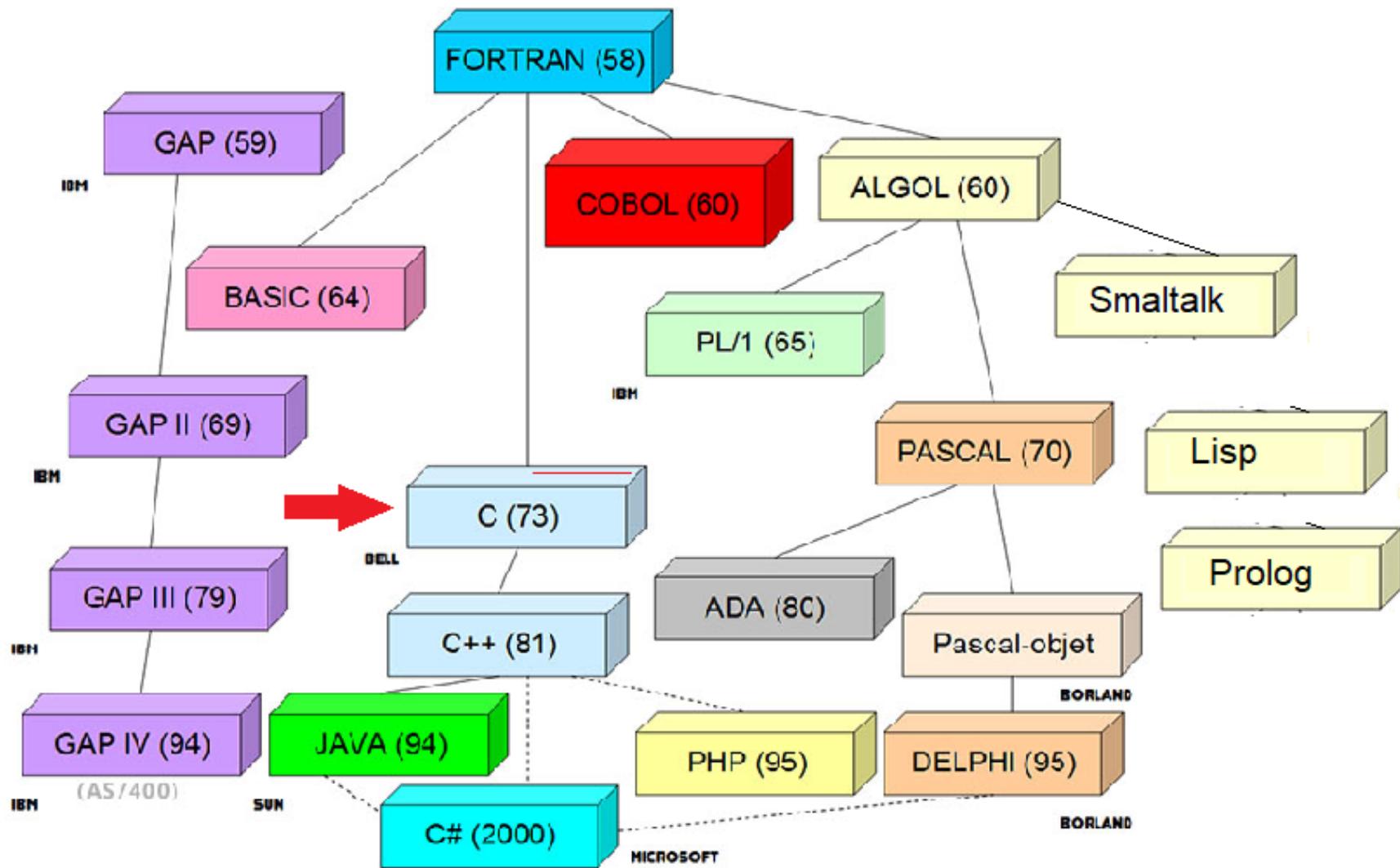
Architecture en couches



Historique (Operating Systems)



Historique (langages)



Présentation du langage C

Historique

- Au milieu des années 1980 la popularité du langage était établie.
- De nombreux compilateurs ont été écrits, mais comportant quelques incompatibilités portant atteinte à l'objectif de portabilité (**devcpp** gratuit sur le web).
- Il s'est ensuivi un travail de normalisation effectué par le comité de normalisation X3J11 de l'ANSI qui a abouti en 1988 avec la parution par la suite du manuel :
 - < The C programming Language – 2ème édition >.

Présentation du langage C

Intérêts

- Langage polyvalent permettant le développement de systèmes d'exploitation, de programmes applicatifs scientifiques et de gestion.
- Langage structuré.
- Langage évolué qui permet néanmoins d'effectuer des opérations de bas niveau (< assembleur d'Unix >).
- Portabilité (en respectant la norme !) due à l'emploi de bibliothèques dans lesquelles sont reléguées les fonctionnalités liées à la machine.
- Grande efficacité et puissance.
- Compatible avec le système Unix
- Langage permissif !!

Présentation du langage C

Qualités

- Clarté
- Simplicité
- Modularité
- Extensibilité
- Langage Haut Bas niveau
- Code optimisé

Généralités

- Jeu de caractères utilisés
- Mots réservés
- Structure d'un programme C
- Compilation et Edition des liens

Généralités : Jeu de caractères

- 26 lettres de l'alphabet (minuscules, majuscules)
- chiffres 0 à 9
- caractères spéciaux :
 - ! * + \ " < # (= | { >
 - %) ~;] / ^ - [: , ?
 - & _ } ‘ . (espace) (_ underscore)
- séquences d'échappement :
 - passage à la ligne (\n),
 - tabulation (\t),
 - backspace (\b).

Généralités : mots réservés

- auto extern sizeof break float
- case for char goto switch
- Static const if typedef continue
- int default long unsigned do
- register void double return volatile
- else short while enum signed
- union struct

Structure d'un programme C

- une ou plusieurs fonctions dont l'une doit s'appeler **main** stockées dans un ou plusieurs fichiers.
- Fonction :
 - entête (type et nom de la fonction suivis d'une liste d'arguments entre parenthèses),
 - instruction composée constituant le corps de la fonction.
- Instruction composée : délimitée par les caractères { et }
- Instruction simple : se termine par ;
- Commentaire : encadré par les délimiteurs
 /* et */
- Instruction préprocesseur : commence par #

Exemple d'un programme C

```
#include <stdio.h>
/*directive de compilation */
#define PI 3.14159
/* calcul de la surface d'un cercle */
main()
{
    float rayon, surface;
    float calcul(float rayon);
    printf("Rayon = ? ");
    scanf("%f", &rayon);
    surface = calcul(rayon);
    printf("Surface = %f\n",
    surface);
}

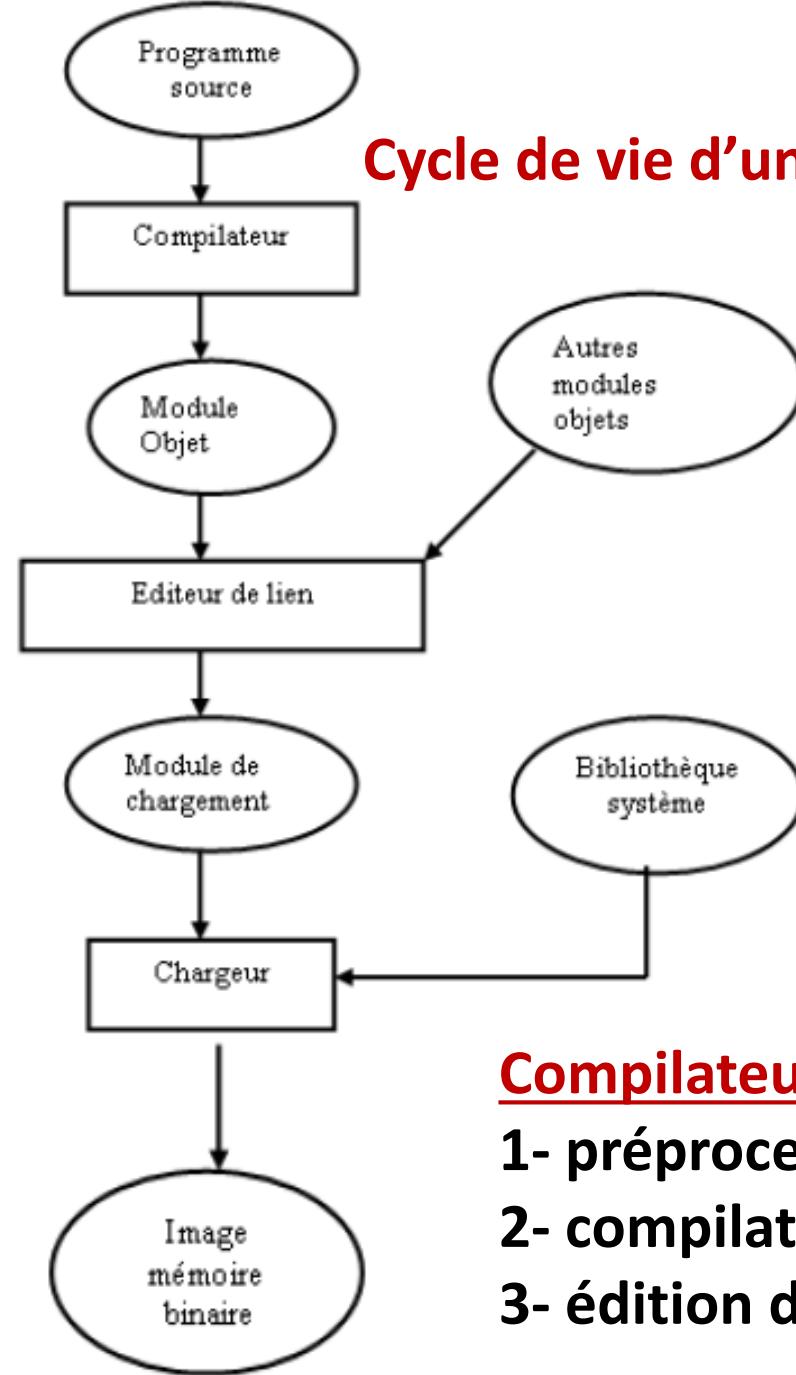
/* definition de fonction */
float calcul(float r)
{
    /* definition de la variable locale */
    float a;
    a = PI * r * r;
    return(a);
}
```

Compilation et édition des liens

Moment de compilation

Moment de chargement

Moment de l'exécution



Cycle de vie d'un programme

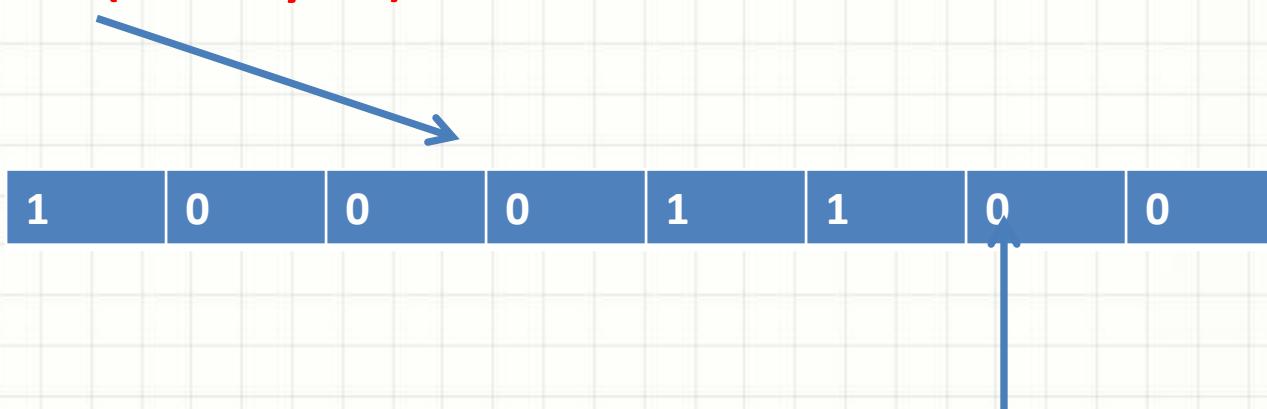
Compilateur C :
1- préprocesseur
2- compilation
3- édition des liens

Les types simples et structurés

- Le langage contient des types de base qui sont les entiers, les réels simple et double précision et les caractères que l'on identifie à l'aide des mots-clés **int**, **float**, **double** et **char** respectivement.
- De plus il existe un type ensemble vide : le type **void**.
- Les mots-clés **short** et **long** permettent d'influer sur la taille mémoire des entiers et des réels.
- Type tableau T[N]

Unités de mesure

- Octet (ou Byte) = 8 bits



- Bits {0,1}

- Chaque caractère est représenté dans la mémoire par 1 octet
- D'où : avec 1 octet on peut représenter (coder) 256 caractères
- Codage ASCII

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Structures simples : remarques

- La taille d'un **entier** par défaut est soit 2 soit 4 octets (dépend de la machine). 4 octets est la taille la plus courante.
- La taille d'un **entier court** est en général 2 octets et celle d'un **entier long** 4 octets.
- La taille d'un entier court est inférieure ou égale à la taille d'un entier par défaut qui est elle-même inférieure ou égale à celle d'un entier long.
- Les types **short int** et **long int** peuvent être abrégés en short et long.
- **float** est le type des réels en **simple précision**
- **double** est le type des réel double précision
- Le type **char** occupe **un octet**. Un caractère est considéré comme un entier qu'on pourra donc utiliser dans une expression arithmétique.

Structures simples : entrées/sorties

- **scanf()**, **printf()** : lecture et écriture formatée de données

- Exemples :

1- **int** x,y;

printf("donner deux entiers"); **scanf**("%d%d",&x,&y);

2- **int** JOUR, MOIS, ANNEE; **float** x;

scanf("%d %d %d%f", &JOUR, &MOIS, &ANNEE,&x);

3- **int** A = 4; **int** B = -7;

printf("%d * %d = %d\n", A, B, A*B);

4- **char** x='a';

scanf("%c",&x); **printf**("%c",x);

Lecture/écriture : Exemple

```
#include <stdio.h>
// lire et afficher trois variables
main() {
    float A; int B; char C;
    printf("donner un réel ");
    scanf("%4.2f",&A);
    /*%4.2f : A est sur 4 chiffres dont deux après la virgule*/
    printf("donner un entier ");
    scanf("%d",&B);
    printf("donner un caractère ");
    scanf("%c",&C);
    printf("A = %f B = %d C = %c", A, B, C);
}
```

Exercice

Ecrire un programme en langage C qui permet de saisir deux entiers naturels A et B et qui réalise leur permutation,

- 1- En utilisant une variable intermédiaire
- 2- sans utiliser une variable intermédiaire

Exemple :

A = 50 B = 100

Affichage

A=100 B=50

Permutation de deux variables

1

```
#include <stdio.h>
main() {
    float A, B,C;
    printf("donner un réel A");
    scanf("%f",&A);
    printf("donner un réel B");
    scanf("%f",&B);
    C = B;
    B = A;
    A = C;
    printf("%f et %f ", A, B);
}
```

2

```
#include <stdio.h>
main() {
    float A, B;
    printf("donner un réel A");
    scanf("%f",&A);
    printf("donner un réel B");
    scanf("%f",&B);
    B = A + B;
    A = B - A;
    B = B - A;
    printf("%f et %f ", A, B);
}
```

Les structures de contrôle :

Les conditions

- if (expression)
 partie-alors
 [else
 partie-sinon]
- La partie-alors et la partie-sinon peuvent être indifféremment une instruction élémentaire ou composée.
- La partie-alors sera exécutée si la valeur de l'expression
- entre parenthèses est non nulle. Sinon, si le test comporte une partie-sinon c'est celle-ci qui sera exécutée.

Les conditions : exemples

- La partie-alors et la partie-sinon peuvent être indifféremment une instruction élémentaire ou composée.
- La partie-alors sera exécutée si la valeur de l'expression entre parenthèses est non nulle. Sinon, si le test comporte une partie-sinon c'est celle-ci qui sera exécutée.

Les conditions : exemples

- Si plusieurs tests sont imbriqués, chaque partie **else** est reliée au **if** le plus proche qui n'est pas déjà associé à une partie **else**.

```
if( x >=16 )
    printf( "très bien" );
else if( x >=14 )
    printf( "Bien" );
else if( x >=12 )
    printf( "Assez B." );
else
    printf( "Passable" );
```



```
if( x >=16 ) {
    printf( "très bien" );
} else if( x >=14 ) {
    printf( "Bien" );
} else if( x >=12 ) {
    printf( "Assez B." );
} else
    printf( "Passable" );
```

Les conditions ; exemple

- Ecrire un programme en langage C qui lit deux réels quelconques A et B et qui permet de résoudre l'équation :
- $AX + B = 0$

Les conditions ; exemple

```
#include <stdio.h>
main() {
    float A, B, X;
    printf("donner un réel A"); scanf("%f",&A);
    printf("donner un réel B"); scanf("%f",&B);
    If (A != 0) { X = -B/A;
                  printf("solution = %f",X);}
    else if (B != 0)
        printf("Pas de solution !");
    else
        printf("tout réel est une solution");
}
```

Les conditions ; exemple

- Ecrire un programme en langage C qui lit trois réels quelconques A, B et C et qui permet de résoudre l'équation :
- $AX^2 + BX + C = 0$
- #include <math.h>
- On se donne la fonction racine carré : **sqrt**

Les conditions ; équation de second degré Ax²+Bx+C=0

```
#include <stdio.h>
#include <maths.h>
main() {
    float A, B, C, X1, X2, delta;
    printf("donner un réel A\n"); scanf("%f",&A);
    printf("donner un réel B\n"); scanf("%f",&B);
    printf("donner un réel C\n"); scanf("%f",&C);
    if (A != 0) { delta = B*B-4*A*C;
        if (delta <0) printf("pas de solution dans R");
        else if (delta == 0)
            { x1 = -B/(2*A); printf("racine double %f", x1) }
        else { x1 = (-B-sqrt(delta))/(2*A));
            x2 = (-B+sqrt(delta))/(2*A));
            printf("deux racines distinctes %f et %f", x1, x2);
        }
    }
    else if (B!=0) { x1 = -B/C; printf("solution = %f", x1)}
    else if (C == 0) printf("tout réel est une solution ");
    else printf("pas de solution dans R");
}
```

Exercice

Ecrire un programme C qui lit l'heure de départ d'un train (heure et minutes HD, MD) et la durée de trajet (heure et minutes HT, MT) et calcule et affiche l'heure d'arrivée ? (heure et minutes HA, MA),

NB : On suppose que l'heure de départ et la durée du trajet sont correctes :

HD < 24

MD < 60

HT < 24

MT < 60

Exemple : HD=8 MD = 40

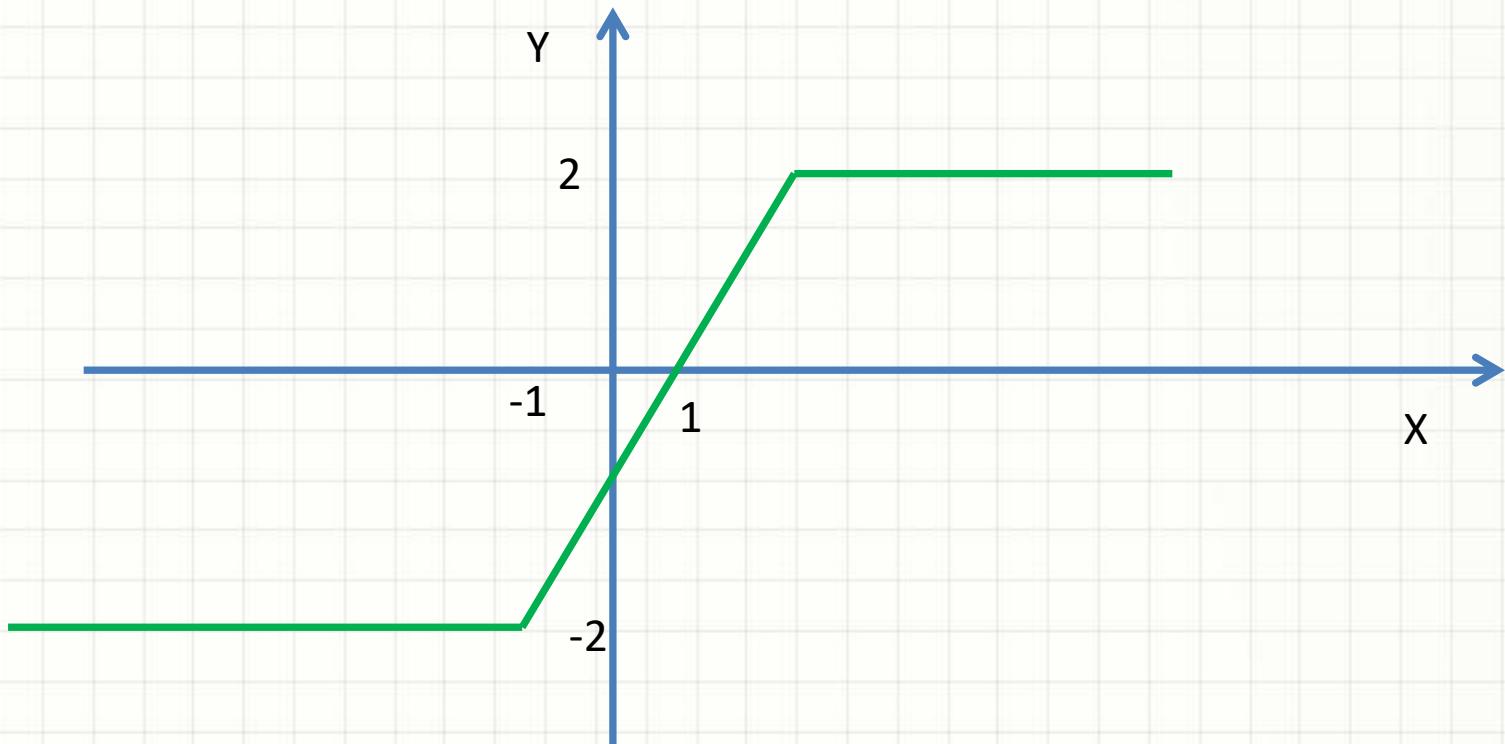
HT = 2 MT = 30 → HA = 11 MA = 10

```
#include <stdio.h>
main()
{
    int HD, MD, HT, MT;
    int HA, MA;

    printf("donner l'heure de départ");
    scanf("%d",&HD); scanf("%d",&MD);
    printf("donner la durée");
    scanf("%d",&HT); scanf("%d",&MT);
    HA = HD + HT;
    MA = MD + MT;
    if (MA >=60) {MA = MA - 60; HA = HA + 1;}
    if (HA >= 24) HA = HA - 24;
    printf("heure d'arrivée = %d %d", HA, MA);
}
```

Exercice

Etant donnée une fonction f définie par le graphique ci-dessous, calculer Y pour un X donné ?



```
#include <stdio.h>
main()
{
    float X, Y;

    printf("donner X");
    scanf("%f", &X);

    if (X <= -1)
        Y = -2;
    else
        if (X < 3) Y= X - 1;
        else Y = 2;

    printf(" Y = %f", Y);
}
```

expression

Opérateur

Opérande

Opération

The diagram highlights the expression `Y= X - 1;` with a green oval. The operator `-` is highlighted with a yellow oval, and the operand `1` is highlighted with a blue oval. Arrows point from the labels to their respective parts of the expression.

Les conditions ; exemple

- Ecrire un programme en langage C qui lit trois réels quelconques A, B et C et qui permet d'afficher la plus grande valeur parmi les trois variables ?

Version 1

```
#include <stdio.h>

main() {
    float A, B, C, max;
    printf("donner un réel A"); scanf("%f",&A);
    printf("donner un réel B"); scanf("%f",&B);
    printf("donner un réel C"); scanf("%f",&C);

    max = A;
    if (max < B) max = B;
    if (max < C) max = C;

    printf("max de %f %f et %f = %f", A, B, C , max);
}
```

Version 2

```
#include <stdio.h>
main() {
    float A, B, C, max;
    printf("donner un réel A"); scanf("%f",&A);
    printf("donner un réel B"); scanf("%f",&B);
    printf("donner un réel C"); scanf("%f",&C);

    if (A > B) max = A;
    else max = B;

    if (max < C) max = C;

    printf("max de %f %f et %f = %f", A, B, C , max);
}
```

Version 3

```
#include <stdio.h>
main() {
    float A, B, C, max;
    printf("donner un réel A"); scanf("%f",&A);
    printf("donner un réel B"); scanf("%f",&B);
    printf("donner un réel C"); scanf("%f",&C);
    if (A > B)
        if (A > C) max = A;
        else max = C;
    else
        if (B>C) max = B;
        else max = C;
    printf("max de %f %f et %f = %f", A, B, C , max);
}
```

Exercice

**Permuter le contenu de trois variables A, B et C de façon que
A soit supérieur à B et B soit supérieur à C ?**

Exemple 1:

A = 5 B = 10 C = -3

Résultat : A=10 B = 5 C = -3

Exemple 2:

A = 5 B = -3 C = 20

Résultat : A=20 B = 5 C = -3

Permutation de trois variables

```
#include <stdio.h>
main() {
    float A, B, C, max, x;
    printf("donner un réel A"); scanf("%f",&A);
    printf("donner un réel B"); scanf("%f",&B);
    printf("donner un réel C"); scanf("%f",&C);
    max = A;
    if (B > max) max = B;
    if (C > max) max = C;
    if (max == B) {B = A; A=max;}
    if (max == C) {C = A; A=max;}
    if (C>B) {x = C; C=B; B=x;}
    printf("variables permutées %f %f et %f ", A, B, C );
}
```

Les boucles : while

- Deux structures

1) while (expression)
 corps-de-boucle

2) do
 corps-de-boucle
 while (expression);

//boucle répéter

- La partie corps-de-boucle peut être soit une instruction élémentaire soit une instruction composée.
- Dans la boucle while le test de continuation s'effectue avant d'entamer le corps-de-boucle qui, de ce fait, peut ne jamais s'exécuter.
- Par contre dans la boucle do-while ce test est effectué après le corps-de-boucle, lequel sera alors exécuté au moins une fois.

Boucles : exemple

- Ecrivez un programme qui lit N nombres entiers au clavier et qui affiche leur somme, leur produit et leur moyenne. Choisissez un type approprié pour les valeurs à afficher. Le nombre N est à entrer au clavier. Résolvez ce problème,
 - en utilisant **while**,
 - en utilisant **do - while**,
 - en utilisant **for**.
 - Laquelle des trois variantes est la plus naturelle pour ce problème?

Boucles : exemple while ...

```
#include <stdio.h>

main()
{
    int N,i ; /* N : nombre de données , i : compteur*/
    int NOMB; /* nombre courant */
    int SOM, PROD ; /* la somme et le produit des nombres */
    printf("Nombre de données : ?"); scanf("%d", &N);
    SOM=0; PROD=1; I=1;
    while(I<=N) {
        printf(" le nombre numéro %d: ", I);
        scanf("%d", &NOMB); SOM = SOM+NOMB;
        PROD = NOMB*PROD; I=I+1;
    }
    printf("La somme des %d nombres = %d \n", N, SOM); printf("Le produit des
    %d nombres = %d\n", N, PROD);
    printf("La moyenne des %d nombres = %f\n", N, (float) SOM/N);
}
```

Boucles : exemple do ... while

```
#include <stdio.h>

main()
{
    int N,i ; /* N : nombre de données , i : compteur*/
    int NOMB; /* nombre courant */
    int SOM, PROD ; /* la somme et le produit des nombres */
    printf("Nombre de données : "); scanf("%d", &N);
    SOM=0; PROD=1; I=1;
    do { printf("%d. nombre : ", I); scanf("%d", &NOMB);
          SOM = SOM+NOMB;
          PROD = PROD*NOMB; I=I+1; }
    while(I<N);
    printf("La somme des %d nombres = %d \n", N, SOM);
    printf("Le produit des %d nombres = %f\n", N, PROD); printf("La
    moyenne des %d nombres = %f\n", N, (float) SOM/N);
}
```

Les boucles : Pour... faire

- **for ([expr1]; [expr2]; [expr3])**
corps-de-boucle
- L'expression expr1 est évaluée une seule fois, au début de l'exécution de la boucle.
- L'expression expr2 est évaluée et testée avant chaque passage dans la boucle.
- L'expression expr3 est évaluée après chaque passage.
- Ces 3 expressions jouent respectivement le rôle :
 - d'expression d'initialisation,
 - de test d'arrêt,
 - d'incrémentation.

Boucles : exemple for ()

```
#include <stdio.h>

main()
{
    int N,i ; /* N : nombre de données , i : compteur*/
    int NOMB; /* nombre courant */
    int SOM, PROD ; /* la somme et le produit des nombres */
    printf("Nombre de données : "); scanf("%d", &N);
    I=1 ;
    for (SOM=0, PROD=1; I<=N ; I=I+1)
    {
        printf("%d. nombre : ", I); scanf("%d", &NOMB);
        SOM = SOM+NOMB; PROD = PROD*NOMB;
    }
    printf("La somme des %d nombres = %d \n", N, SOM);
    printf("Le produit des %d nombres = %f\n", N, PROD); printf("La
moyenne des %d nombres = %f\n", N, (float) SOM/N);
}
```

L'aiguillage

- L'instruction switch définit un aiguillage qui permet d'effectuer un branchement à une étiquette de cas en fonction de la valeur d'une expression.
- Syntaxe

switch (expression)

{

case etiq1 : [liste d'instructions]

case etiq2 : [liste d'instructions]

...

case etiqn : [liste d'instructions]

[default: [liste d'instructions]]

}

- Les étiquettes de cas (etiq1, etiq2, ..., etiqn) doivent être des expressions constantes.

L'aiguillage : exemple

```
#include <stdio.h>    //jour de la semaine  
main()  
{  int N; /* Numéro jour de la semaine */  
    printf("Numéro du jour de la semaine : "); scanf("%d", &N);  
    switch (N)  
    {    case 1 : printf("Lundi "); break;  
        case 2 : printf("Mardi "); break;  
        case 3 : printf("Mercredi "); break;  
        case 4 : printf("Jeudi"); break;  
        case 5 : printf("Vendredi"); break;  
        case 6 : printf("Samedi"); break;  
        case 7 : printf("Dimanche"); break;  
    } }
```

Instructions d'échappement

- Ces instructions permettent de rompre le déroulement séquentiel d'une suite d'instructions.
- Instruction **continue** ;
- Son rôle est de forcer le passage à l'itération suivante de la boucle la plus proche.
- Instruction **break** ;
- Permet de quitter la boucle ou l'aiguillage le plus proche.
- **exit(expression);**
- Un programme peut être interrompu au moyen de la fonction exit. L'argument de cette fonction doit être un entier indiquant le code de terminaison du processus.

Instructions d'échappement

```
#include <stdio.h>

main()
{
    int i;
    for (l=1 ; l<=10 ; l=l+1)
        { if ((l>=4) && (l<=8)) continue;
          printf(" nombre courant : %d ", l);
        }
}
```

Résultats:

nombre courant:1

nombre courant:2

nombre courant:3

nombre courant:9

nombre courant:10

Exemple

Etant donnés deux entiers naturels A et B, écrire un programme en langage C permettant de calculer le produit A*B en utilisant uniquement des additions.

$$5 \times 7 = 5+5+5+5+5+5+5$$

$$5 \times 7 = 7+7+7+7+7$$

Boucles : exemple for ()

```
#include <stdio.h>

main()
{ int I ; /*I : compteur*/
  int A,B; /* nombre s à multiplier */
  int SOM ; /* la somme des nombres */
  Printf("donner deux entiers A et B");
  Scanf("%d",&A); Scanf("%d",&B);
  for (SOM=0,I=1 ; I<=B ; I=I+1)
  {
    SOM = SOM + A; /* SOM += A;*/
  }
  printf("Le produit de %d et %d = %d\n", A,B, SOM);
}
```

Boucles : exemple for ()

```
#include <stdio.h>

main()
{ int I ; /*I: compteur*/
  int A,B; /* nombre s à multiplier */
  int SOM ; /* la somme des nombres */

  Printf("donner deux entiers A et B");
  Scanf("%d",&A); Scanf("%d",&B);
  SOM=0; I=1;
  While (I<=B)
    { SOM = SOM + A; I=I+1;}
  printf("Le produit de %d et %d = %d\n", A,B, SOM);
}
```

Application

Etant donnés deux entiers naturels A et B (supposé non Nul), écrire un programme en langage C permettant de calculer le quotient et le reste de la division entière de A par B en utilisant uniquement des additions et des soustractions.

$$A=20 \quad B=3 \quad q=0$$

$$20-3 = 17 \quad q=1 \quad 17-3 = 14 \quad q=2 \quad 14-3=11 \quad q=3$$

$$11-3 = 8 \quad q=4 \quad 8-3 = 5 \quad q=5 \quad 5-3 = 2 \quad q=6 \quad 2-3<0 \quad \text{!!! Arrêt}$$

Division entière

```
#include <stdio.h>
main()
{
    int A, B, Q, R, AUX; /* diviser A par B */
                           /* Q : quotient R : reste */
    printf(" donner la valeur de A :\n"); scanf("%d", &A);
    printf(" donner la valeur de B :\n"); scanf("%d", &B);
    Q=0; AUX=A;
    while (AUX>=B) {
        AUX = AUX - B;
        Q = Q + 1;
    }
    R = AUX;
    printf("quotient = %d et reste = %d\n",Q,R); }
```

Tables de vérité

&&

ET	VRAI	FAUX
OU	VRAI	FAUX
NON	VRAI	FAUX

||

ET	VRAI	FAUX
OU	VRAI	FAUX
NON	VRAI	FAUX

!

ET	VRAI	FAUX
OU	FAUX	VRAI
NON	VRAI	FAUX

Algorithme Nombre premier

Ecrire un programme en langage C qui permet de saisir un entier naturel N et qui permet de vérifier si N est un nombre premier ou non. Un nombre est dit premier s'il a que deux diviseurs 1 et lui-même.

Exemples :

5, 11, 13, 23,

ET logique : **&&**

OU logique **||**

!= différent

/ division

% reste de la division entière

```
#include <stdio.h>

main()
{
    int N, i;
    printf("donner un entier naturel");
    scanf("%d", &N);
    i = 2;
    while ( (i<=N/2) && (N%i != 0))
        i = i+1;
```

```
if (i < N/2) print f("%d est NON premier",N);
else printf ("%d est Premier ",N);
```

```
}
```

```
#include <stdio.h>

main()
{
    int N, i, existe;
    printf("donner un entier naturel");
    scanf("%d", &N);
    i = 1; existe = 0;
    while ( (i<=N/2) && (existe == 0))
    {
        i = i+1;
        if (N%i==0) existe = 1;
    }
    if (existe ==0) printf("%d est premier",N);
    else printf("%d est non premier",N);
}
```

Afficher tous les nombres premiers dans un intervalle A..B ?

```
#include <stdio.h>
main()
{
    int i, j, existe, A, B;
    do { printf("donner deux entiers A et B");
          scanf("%d", &A); scanf("%d", &B); }
    while ((A >B) || (A<0) || (B<0));
    for (j=A; j<=B; j=j+1) {
        i = 1; existe = 0;
        while ( (i<=j/2) && (existe == 0)) {
            i = i+1;
            if (j%i==0) existe = 1; }
        if (existe ==0) printf("%d : ", j);
    } }
```

Nombre parfait

Ecrire un programme en langage C qui permet de saisir un entier naturel N et qui permet de vérifier si N est un nombre parfait ou non. Un nombre est dit parfait s'il est égale à la somme de ses diviseurs sauf lui-même.

Exemple :

6 est un nombre parfait car $6 = 1 + 2 + 3$

28 est un nombre parfait car :

$$28 = 1 + 2 + 4 + 7 + 14$$

Nombre parfait

```
#include <stdio.h>

main()
{
    int N, i, s;
    printf(" donner la valeur de N :\n"); scanf("%d", &N);
    s=0;i=1;
    for (; i<=N/2; i=i+1)
        { if (N%i == 0) s= s + i; }

    if (s==N)
        printf("%d parfait \n",N);
    else printf("%d non parfait \n",N);

}
```

Afficher tous les nombres parfaits dans un intervalle A..B ?

```
#include <stdio.h>

main()
{
    int i, j, A, B,s;
    do { printf("donner deux entiers A et B");
          scanf("%d", &A); scanf("%d", &B); }
    while ((A >B) || (A<0) || (B<0));
    for (j=A; j<=B; j=j+1) {
        s=0;
        for (i=1; i<=j/2; i=i+1)
            { if (j%i == 0) s= s + i; }
        if (j==s) printf("%d : ", j);
    }
}
```

Afficher tous les nombres parfaits <1000

```
#include <stdio.h>
main()
{
    int j, i, s;
    for (j=1;j<1000; j=j+1)
    {
        for (s=0,i=1; i<=j/2; i=i+1)
            if (j%i == 0) s= s + i;
        if (s==j)
            printf("%d parfait \n",j);
    }
}
```

Raccourcis

Instructions	Raccourcis
$x = x + 1;$	$x++;$ OU $++x;$
$x = x + y;$	$x += y;$
$x = x * y;$	$x *= y;$
$i = i + 1;$	$i++;$ OU $++i;$
$for (....)$ $\{ x = i++; \} \quad \{ x = ++i; \}$	$for (...)$ $\{ x = i; \quad i = i + 1; \} \quad \{ i = i + 1; \quad x = i; \}$
$A = j --;$	$A = j;$ $j = j - 1;$
$A = --j;$	$J = j - 1;$ $A = j;$
$for (....)$ $\{ x = ++i; \}$ $\{ Y = j ++; \}$	$for (...)$ $\{ i = i + 1; \quad x = i; \}$ $\{ Y = j; \quad j = j + 1; \}$

- **Exercice 1**

Ecrire un programme C qui calcule N! ?

- **Exercice 2**

Etant donnée une somme d'argent S en Dinars, on veut représenter cette somme avec un nombre minimum de pièces de :

20 D (P20), 10 D (P10), 5 D (P5), 1 D (P1).

Ecrire l'algorithme qui lit S et qui affiche en sortie les valeurs de P20, P10, P5 et P1 ?

- **Exercice 3**

Considérons la somme suivante :

$$\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{(-1)^{n+1}}{n}$$

Ecrire un algorithme calculant la somme pour un n donné, où n est un entier supposé positif à saisir ?

```
#include <stdio.h>
main()
{
    int i, N, P=1;

    do
        printf("donner un entier positif");
        scanf("%d",&N);
    while (N<0);

    for (i=1, P=1; i<=N; i++)
        P = P*i;

    printf("factorielle de %d = %d\n", N, P);
}
```

```
#include <stdio.h>

main()
{
    int S, P20=0, P10=0, P5=0, P1=0, R;
    printf("donner la somme S\n");
    scanf("%d", &S);
    P20 = S/20;
    if (S%20 != 0) {
        R = S%20;
        P10 = R/10;
        if ( R%10 != 0 ) {
            R = R%10;
            P5 = R/5;
            if (R%5 != 0) P1 = R%5; }
    }
    printf("les quantités sont = %d %d %d %d\n", P20, P10, P5, P1);
}
```

```
#include <stdio.h>

main()
{ int N, i;
  float S ;
  printf("donner la somme N\n");
  scanf("%d", &N);
  for (i=1, S=0;i<=N; i++) {
    if (i % 2 == 0)
      S = S - (float) 1/i;
    else S = S + (float) 1/i;
  }
  printf("la somme S= %f \n", S);
}
```

- **Exercice 4**

Ecrire un programme C qui saisit deux entiers naturel N et M supposés strictement positifs et qui permet de calculer la somme suivante :

$$S = \sum_{i=0}^N \prod_{j=1}^M (i + j)$$

Exemple : N=2 M=3

	j=1	j=2	j=3			
i=0	1	*	2	*	3	=6
i=1	2	*	3	*	4	=24
i=2	3	*	4	*	5	=60

$$S=90$$

```
#include <stdio.h>

main()
{
    int N, M, i,j, som, prod;
    printf ("donner un entier N");scanf("%d",&N);
    printf ("donner un entier M");scanf("%d",&M);
    som=0;
    for (i=0; i<=N; i++)
    {
        prod=1;
        for (j=1; j<=M; j++)
            prod = prod * (i+j);    // prod *= (i+j)
        som = som + prod;        // som += prod
    }
    printf("somme = %d",som);
}
```

- **Exercice 5**

Ecrire un programme C qui saisit un entier N qui doit être strictement positif, et qui permet de calculer la variable M représentant le nombre d'occurrences de chiffre 2 compris dans N ?

Exemple :

$$N = 1142256322472225 \rightarrow M = 7$$

$$N = 12252 \rightarrow M = 3$$

$$N = 22352522 \rightarrow M = 5$$

```
#include <stdio.h>
main()
{
    long int N, M=0, K;
    do { printf ("donner une valeur de N");
          scanf ("%d", &N);}
    while (N<=0);
    K = N;
    do {
        if (K % 10 == 2) M++;
        k = k / 10;
    } while (K>0);
    printf (" nombre de répétitions de %ld = %d\n",N, M);
}
```

- **Exercice 6**

Ecrire un programme C qui saisit deux entiers naturels A et b et qui permet de calculer leur PGCD (plus grand commun diviseur) en se basant sur des soustractions successives ?

Exemple : A = 40 B=22

$$A = A - B = 40 - 22 = 18$$

$$B = B - A = 22 - 18 = 4$$

$$A = A - B = 18 - 4 = 14$$

$$A = A - B = 14 - 4 = 10$$

$$A = A - B = 10 - 4 = 6$$

$$A = A - B = 6 - 4 = 2$$

$$B = B - A = 4 - 2 = 2$$

$$A = A - B = \textcolor{red}{2-2} = 0$$

- **Exercice 6** Ecrire un programme C qui saisit deux entiers naturels A et B et qui permet de calculer leur PGCD (plus grand commun diviseur) en se basant sur des soustractions successives ?

```
#include <stdio.h>
int main()
{ int A, B, A1, B1;
  do { printf("donner deux entiers positifs");
        scanf("%d%d ",&A, &B); }
  while ((A<0) || (B<0) );
  A1=A; B1=B;
  While (A1 != B1)
    if (A1 > B1) A1 = A1 - B1;
    else B1 = B1 - A1;
  Printf("PGCD = %d",A1);
}
```

Opérateurs arithmétiques

- Il existe 5 opérateurs arithmétiques : + - * / %
- Lorsque les types des deux opérandes sont différents, il y a **conversion implicite** dans le type le plus fort suivant certaines règles.
 - + si l'un des opérandes est de type long double, convertir l'autre en long double,
 - + sinon, si l'un des opérandes est de type double, convertir l'autre en double,
 - + sinon, si l'un des opérandes est de type float, convertir l'autre en float,
 - + sinon, si l'un des opérandes est de type unsigned long int, convertir l'autre en unsigned long int,
 - + sinon, si l'un des operandes est de type long int, convertir l'autre en long int,
 - + sinon, si l'un des operandes est de type unsigned int, convertir l'autre en unsigned int,
 - + sinon, les deux operandes sont de type int.

Opérateurs arithmétiques

- **float X;**
- **X = 5/7;**
- **X = 5./7;**
- **X = (float) 5/7;**
-
- **float Y;**
- **Y = (float) 5/7;**

Opérateurs logiques

- Les opérateurs logiques comprennent :
 - + 4 opérateurs relationnels : < <= > >= !
 - + 2 opérateurs de comparaison : == !=
 - + 2 opérateurs de conjonction : et logique (&&), ou logique (||).
- Le résultat de l'expression :
 - + **!expr1** est vrai si expr1 est fausse et faux si expr1 est vraie ;
 - + **expr1 && expr2** est vrai si les deux expressions expr1 et expr2 sont vraies et faux sinon.
 - + **expr1 | expr2** est vrai si l'une au moins des expressions expr1, expr2 est vraie et faux sinon.

Opérateurs suite

- Opérateurs d'adressage & *
& appliqué à une variable renvoi son adresse, * appliqué à un pointeur renvoi l'objet pointé
- Opérateur de forçage de type
(type) expression
Exemple : (float) 5/3
- Opérateurs d'incrémentation et de décrémentation ++ (i++ ou ++i) -- (i- ou -*i)
 - *int x, i=10;*
 - *x=i++; (x=i puis i=i+1)* *x=10*
 - *x=++i; (i=i+1 puis x=i)* *x=11*
- Opérateur de taille **sizeof**
 - renvoi la taille en octets de son opérande
 - **Sizeof(float)** renvoi la taille d'un réel, etc,

Ordre d'évaluation des opérateurs

Classe d'opérateur	Opérateur(s)	Associativité
Parenthésage	()	de gauche à droite
Appel de fonction	()	de gauche à droite
Suffixes ou	[] -> . ++ --	
Un-aires préfixes	& * + - ! ++ - sizeof sizeof()	de droite à gauche
Changement de type	(type)	de droite à gauche
Multiplicatifs	* / %	de gauche à droite
Additifs	+ -	de gauche à droite
Egalités	== !=	de gauche à droite
Comparaisons et logique	< <= > >=	de gauche à droite
ou logique	&&	de gauche à droite
Affectations	= += -= *= /= %= &= = ^= <<= >>=	de droite à gauche

Exercice :

Ecrire un programme C qui permet de lire une suite de caractères terminée par le caractère point : ‘.’ et de compter et afficher le nombre de mots ‘LE’ ?

Exemple :

Suite de caractères : FGH JH LEJJJ FDT LE NNM PO L E LE X.

Le programme affiche : 3

```
... do {  
    printf("donner un caractère");  
    scanf("%c",&c1);  
    ...  
} while (c1!='.'); ...
```

```
... do {  
    printf("donner un caractère");  
    scanf("%c",&c1);  
    printf("donner un caractère");  
    scanf("%c",&c2);  
    if (c1=='L') && (c2=='E')  
        i = i+1;  
} while ((c1!= '.') || (c2!= '.'))
```

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

```
Int main()
```

```
{
```

```
int i=0; char c1,c2=' ';
```

```
do {
```

```
    printf("donner un caractère :"); c1=getch();
```

```
    if ((c2=='L') && (c1=='E')) i = i + 1;
```

```
    c2 = c1; }
```

```
while (c1 != ':')
```

```
printf("nombre de mot LE = %d",i);
```

```
return 0;
```

```
}
```

```
#include <stdio.h>

main()
{ char car; int i=0;
  do
    printf("donner un caractère \n");
    scanf("%c", &car);
    if (car == 'L') {
      printf("donner un caractère \n");
      scanf("%c", &car);
      if (car=='E') i++;
    }
  while (car != '.')
  printf("nombre de mots LE = %d\n",i);
}
```

Types structurés : Les Tableaux

- Les tableaux sont certainement les variables structurées les plus populaires. Ils sont disponibles dans tous les langages de programmation et servent à résoudre une multitude de problèmes.
- Dans une première approche, le traitement des tableaux en C ne diffère pas de celui des autres langages de programmation.
- Nous allons cependant voir plus loin que le langage C permet un accès encore plus direct et plus rapide aux données d'un tableau.
- Les chaînes de caractères sont déclarées en C comme tableaux de caractères et permettent l'utilisation d'un certain nombre de notations et de fonctions spéciales.

Les Tableaux : exemple 1

- Saisir un tableau à une dimension, affichage du contenu ainsi que la somme de ses éléments
- #include <stdio.h>

```
main() {  
    int T[50]; /* tableau donné */  
    int N, I; /* dimension et compteur */  
    printf("Dimension du tableau (max.50) : ");  
    scanf("%d", &N ); for (I=0; I<=N-1; I=I+1) {  
        printf("Elément %d : ", I); scanf("%d", &T[I]); }  
    /* affichage du tableau */  
    for (I=0; I<=N-1; I=I+1) printf("%d ", T[I]); printf("\n");  
    for (SOM=0, I=0; I<=N-1; I=I+1) SOM = SOM+T[I];  
    printf("Somme de éléments : %d\n", SOM); }
```

Les Tableaux : exemple 2

- On peut initialiser un tableau au moment de la déclaration
- #include <stdio.h>

```
#define N 6
```

```
int main()
```

```
Int k;
```

```
{ int T[]={10,5,9,20,30,7}; int T1[k];  
    int l, min=T[0], max=T[0]; /* compteur, min, max */  
    for (l=1; l<N; l=l+1)  
    { if (T[l]<min) min = T[l];  
        if (T[l]>max) max = T[l];    }  
    printf("minimum = %d\n",min);  
    printf("maximum = %d\n",max);  
    //return 0; }
```

Les Tableaux à deux dimensions

- En C, un tableau à deux dimensions A est à interpréter comme un tableau (uni-dimensionnel) de dimension L dont chaque composante est un tableau (uni-dimensionnel) de dimension C.

NOTE :

45	34	...	50	48
39	24	...	49	45
...
40	40	...	54	44

10 lignes

20 colonnes

- On dit qu'un tableau à deux dimensions est **carré**, si L est égal à C.
- En faisant le rapprochement avec les mathématiques, on peut dire que "*A* est un vecteur de *L* vecteurs de dimension *C*", ou mieux: "*A* est une **matrice** de dimensions *L et C*".

Les Tableaux à deux dimensions

- Considérons un tableau NOTES à une dimension pour mémoriser les notes de 20 élèves d'une classe dans un devoir:
- **int NOTE[20] = {45, 34, ... , 50, 48};**
- Pour mémoriser les notes des élèves dans les 10 devoirs d'un trimestre, nous pouvons rassembler plusieurs de ces tableaux uni-dimensionnels dans un tableau NOTES à deux dimensions :
**int NOTE[10][20] = {{45, 34, ... , 50, 48},
{39, 24, ... , 49, 45},
... {40, 40, ... , 54, 44}};**

NOTE :	45	34	...	50	48	10 lignes
	39	24	...	49	45	
...	
40	40	...	54	44		

20 colonnes

Les Tableaux 2D : exemple

```
#include <stdio.h>

main()
{
    int A[3][4]={{3,5,9,8},{6,-9,14,6}, {5,7,-2,4}};
    int I,J;
    /* Pour chaque ligne ... */
    for (I=0; I<3; I=I+1)
    { /* ... considérer chaque composante */
        for (J=0; J<4; J=J+1)
            printf("%d", A[I][J]);
        /* Retour à la ligne */
        printf("\n"); }

    return 0; }
```

Les Tableaux 2D : exemple 2

```
/* saisie d'un tableau */  
#include <stdio.h>  
  
Int main()  
{  
    int A[5][10];  
    int I,J;  
    /* Pour chaque ligne ... */  
    for (I=0; I<5; I++)  
        /* ... considérer chaque composante */  
        for (J=0; J<10; J++)  
            scanf("%d", &A[I][J]);  
  
    return 0;  
}
```

Chaînes de caractères

- En C, **pas de type prédéfini chaîne de caractères**. En pratique on utilise des tableaux de caractères.
- Convention : le dernier caractère utile est suivi du caractère \0 (de code ascii 0)
- Exemples :

`char t[10];` (9 caractères max, puisque une case réservée pour \0;

`strcpy(t,"abcdefghi")`

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	------

chaque lettre est accessible par l'indice

`char t[12];`

`strcpy(t,"abcdefghi");`

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	0	?	?
-----	-----	-----	-----	-----	-----	-----	-----	-----	---	---	---

- Initialisation

`char t[12]= "abcdefghi";` ou `char t[]="abcdefghi";`

- Constante chaîne de caractères

`#define ceciestuntexte "Bonjour"`

Chaînes de caractères

- En général, les tableaux sont initialisés par l'indication de la liste des éléments du tableau entre accolades:

```
char MACHAINE[ ] = {'H','e','l','l','o','\0'};
```
- Pour le cas spécial des tableaux de caractères, nous pouvons utiliser une initialisation plus confortable en indiquant simplement une chaîne de caractères constante:

```
char MACHAINE[ ] = "Hello";
```

"H" → H\0 'H' → H
- Lors de l'initialisation par [], l'ordinateur réserve automatiquement le nombre d'octets nécessaires pour la chaîne, c.-à-d.: le nombre de caractères + 1 (ici: 6 octets).
- Nous pouvons aussi indiquer explicitement le nombre d'octets à réservé, si celui-ci est supérieur ou égal à la longueur de la chaîne d'initialisation.

Chaînes de caractères

Exemples d'initialisation

- `char MACHAINE[] = "Hello";`
- `char MACHAINE[6] = "Hello";`
- `char MACHAINE[] = {'H','e','l','l','o','\0'};`
- `char MACHAINE[8] = "Hello";`
- par contre:
- `char MACHAINE[5] = "Hello";` donnera une erreur à l'exécution
- `char MACHAINE[4] = "Hello";` donnera une erreur à la compilation.

Chaînes de caractères

Fonctions de bibliothèque **<string.h>**:

- **strlen(<s>)** fournit la longueur de la chaîne sans compter le '\0' final
- **strcpy(<s>, <t>)** copie <t> vers <s>
- **strcat(<s>, <t>)** ajoute <t> à la fin de <s>
- **strcmp(<s>, <t>)** compare <s> et <t> lexicographiquement et fournit un résultat:
 - négatif si <s> précède <t>
 - zéro si <s> est égal à <t>
 - positif si <s> suit <t>
- **strncpy(<s>, <t>, <n>)** copie au plus <n> caractères de <t> vers <s>
- **strncat(<s>, <t>, <n>)** ajoute au plus <n> caractères de <t> à la fin de <s>

Application:

Calculer les N premiers termes de la suite suivante ainsi que leur somme :

$$U_1 = 2$$

$$U_2 = -3$$

$$U_{n+2} = U_{n+1} + U_n$$

Corrigé

```
#include <stdio.h>
#define max 100
main() {
    int T[max];
    int N, i;
    do
        printf("donner une valeur de N"); scanf("%d",&N);
    while ((N<1) || (N>100))
        T[0]=2; T[1]=-3;
        for (i=2;i<=N-1;i++)
            T[i] = T[i-1] + T[i-2];
        for (i=0;i<=N-1;i++) printf ("%d ",T[i]);
    }
```

Application NON voir Fonctions

1. Déclarer un tableau X de Nmax(100) valeurs
2. Saisir un entier N tel que devant être $N \leq N_{\text{max}}$
3. Saisir X de N valeurs
4. Calculer la moyenne des valeurs
5. Calculer l'écart type

$$E = \sqrt{\frac{1}{N} \sum_{i=1}^N (\bar{x} - x_i)^2}$$

- La fonction racine carré : sqrt (math.h)

Corrigé

```
#include <stdio.h>
#include <math.h>
const Nmax =100
#define Nmax 100

main() {
    int X[Nmax], i,N;
    float s, m;
    do {
        printf("donner une valeur de N");
        scanf("%d",&N);
    while (N>100) || (N<0))

    for (i=0; i<N; i++) {
        printf("donner la %d ème valeur", i);
        scanf("%d", &X[i]);
    }
```

```
for (s=0,i=0; i<N; i++)
    s += X[i];
m = s/N;

for (s=0,i=0; i<N; i++)
    s += (m-X[i])*(m-X[i]);

E = sqrt(s/N);

printf('moyenne = %f',m);
printf('ecart type = %f',E);
}
```

$$44 \text{ (base 10)} = 101100 \text{ (base 2)}$$

44	2
0	22

22	2
0	11

11	2
1	5

5	2
1	2

2	2
0	1

1	2
1	0

Ecrire un programme C qui permet de convertir un entier N (supposé < 1000) en son équivalent binaire ?

```
#include <stdio.h>
#define Nmax 10      // const Nmax = 10
main()
{ int N, N1, T[Nmax], i=0,j;
printf("donner une valeur de N"); scanf("%d",&N);
N1=N;
while (N1!=0) {
    T[i] = N1%2;
    N1 = N1/2; i++; }

for (j=i-1; j>=0; j--)
printf ("%d ",T[j]);
}
```

Calculer la transposée du Tableau suivant ?

14	5	10	2	3
5	2	5	45	8
6	-6	6	6	9
4	-5	10	44	-10
20	11	100	12	-12

```
int T[5][5] = {{14,5,10,2,3}, {5,2,5,45,8}, {6,-6,6,6,9}, {4,-5,10,44,-10}, {20,11,100,12,-12}};
```

14	5	6	4	20
5	2	-6	-5	11
10	5	6	10	100
2	45	6	44	12
3	8	9	-10	-12

```
#include <stdio.h>
main() {
int N=5, i, j, x;
int T[5][5] = {{14,5,10,2,3}, {5,2,5,45,8},
                {6,-6,6,6,9}, {4,-5,10,44,-10},
                {20,11,100,12,-12}};

for (i=0;i<N; i++)
    for (j=0;j<i;j++)
        { x = T[j][i]; T[j][i]= T[i][j];
          T[i][j] = x; }

for (i=0;i<N; i++) {
    for (j=0;j<N;j++)
        printf("%d ", T[i][j]);
    printf("\n"); }
}
```

Exercices : insertion – suppression de valeur dans un tableau

Insertion ($x=50$ pos=4)

4	10	20	-9	14	7	33	8	78
---	----	----	----	----	---	----	---	----

4	10	20	-9	50	14	7	33	8
---	----	----	----	----	----	---	----	---

Suppression ($x=-9$)

4	10	20	-9	14	7	33	8	78
---	----	----	----	----	---	----	---	----

4	10	20	14	7	33	8	78	78
---	----	----	----	---	----	---	----	----

Corrigé : insertion

```
#include <stdio.h>
#define N 10
int main()
{ int T[]={10,5,9,20,30,7,4, 11, 8,20};
  int x, i, pos ;
  printf("donner x\n") ; scanf("%d",&x) ;
  printf("donner sa position \n") ; scanf("%d",&pos) ;
  if ((pos <= N-1) && (pos >=0))
  {
    If (pos == N-1) T[pos] = x;
    Else { for (i=N-2 ; i>=pos ; i=i-1)
            T[i+1] = T[i];
           T[pos] = x; }
  }
  else printf("erreur de position \n");
for (i=0; i<=N-1; i=i+1) printf("%d ", T[i]); printf("\n");
return 0; }
```

Corrigé : suppression

```
#include <stdio.h>
#define N 10
int main()
{ int T[]={10,5,9,20,30,7,4, 11, 8,20};
  int x, i, k ;
  printf("donner x\n") ; scanf("%d",&x) ;
  i=0 ;
  while ((T[i] !=x) && (i<=N-1)) i=i+1;    //chercher x
  if (i==N) printf("erreur de position \n");
  else {
    if (i==N-1) T[i]=0;
    else { for (k=i;k<=N-2; k=k+1) T[K] = T[k+1];
            T[N-1] = 0;}
  }
  for (i=0; i<=N-1; i=i+1) printf("%d ", T[i]); printf("\n");
  return 0;
}
```

Recherche de 1^{ère} position de x dans un tableau

```
#include <stdio.h>
#define N 10
int main()
{ int T[]={10,5,9,20,30,7,4, 11, 8,20};
  int x, i;

  printf("donner x\n") ; scanf("%d",&x) ;
  i=0 ;
  while ((T[i] !=x) && (i<=N-1)) i=i+1;
  if (i==N) printf(" NON trouvé \n");
  else
    /* première position */
    printf("trouvé à la position %d ", i);
  return 0;
}
```

Recherche Dichotomique de x dans un tableau Ordonné

Pré-condition : un tableau T ordonné, dans un ordre croissant par exemple.

Principe de recherche : les paramètres sont les mêmes que pour la recherche séquentielle, mais la méthode de recherche est complètement différente. On s'inspire de la manière de consulter un dictionnaire.

1-On calcule M l'indice milieu de T

2-On compare X à T[M], trois cas peuvent se présenter :

X=T[M] on a trouvé, on s'arrête

$X < T[M]$, il faut recommencer le même processus mais en ne considérant que le sous vecteur de gauche de M.

$X > T[M]$, c'est le sous vecteur de droite qu'il faut considérer
On s'arrête lorsque $X = T[M]$, ou qu'il n'y a plus de sous vecteur à définir.

Recherche Dichotomique de x dans un tableau Ordonné

Exemple: T =

5	6	8	10	12	14	16	22
---	---	---	----	----	----	----	----

1) X=16

T[1..8] milieu=4

T[4]=10

X>T[4] chercher à droite

T[5..8] milieu=6

T[6]=14

X>T[6] chercher à droite

T[7..8] milieu=7

T[7]=16

X =T[7] **trouvé**

2) X=13

T[1..8] milieu=4

T[4]=10

X>T[4] chercher à droite

T[5..8] milieu=6

T[6]=14

X<T[6] chercher à gauche

T[5..6] milieu=5

T[5]=12

X <>T[5] **échec**

Recherche Dichotomique

```
#include <stdio.h>
#define N 10
#define true 1
#define false 0
int main()
{ int T[]={10,15,19,20,30,31,34, 111, 800,2000};
  int x, M, G, D;
  printf("donner x\n"); scanf("%d",&x) ;
  G=0; D=N-1; trouve = false;
  While ((G<=D) && (trouve==false)) {
    M=(G+D)/2;
    if (T[M]==x) trouve = true;
    else if (T[M] > x) D=M-1;
    else G=M+1;}
  If (trouve== true) printf("trouvé à la position %d",M);
  else printf("Non trouvé ");}
```

Exercice

- Écrire un programme C qui initialise un tableaux de N valeurs entières et qui permet de vérifier si un tableau T est symétrique ou non

Exemple de tableau symétrique

12	3	5	2	5	3	12
----	---	---	---	---	---	----

Exemple de tableau non symétrique

12	3	11	2	5	4	12
----	---	----	---	---	---	----

Corrigé : Tableau symétrique ?

```
#include <stdio.h>
#define N 10
main()
{ int T[]={10,5,9,20,30,7,4, 11, 8,20};
  int i=-1,j=N;

  do {
    i++;
    j--;
  }
  while ((T[i] == T[j]) && (i<N/2))

  if (i == N/2) printf("tableau symétrique");
  else printf("tableau NON symétrique");

}
```

Corrigé : Tableau symétrique ?

```
#include <stdio.h>
#define N 10
main()
{ int T[]={10,5,9,20,30,7,4, 11, 8,20};
  int i=-1;

  do {
    i++;
  }
  while ((T[i] == T[N-i-1]) && (i<N/2))

  if (i == N/2) printf("tableau symétrique");
  else printf("tableau NON symétrique");

}
```

Etant donnés deux tableaux T1 et T2 tous deux ordonnés par ordre croissant, écrire le code C permettant de fusionner T1 et T2 en un autre tableau T3 qui soit lui aussi ordonné?

```
#include <stdio.h>
#define N1 6
#define N2 4
#define N3 10
main()
{ int T1[N1]={3,4,9,10,20, 30};
int T2[N2]={4,7, 10, 1000};
int T3[N3];
..... }
```

T3 = {3,4,4,7,9,10,10, 20,30,1000}

Corrigé : Fusion

```
#include <stdio.h>
#define N1 6
#define N2 4
#define N3 10
int main()
{ int i=0,j=0,k=0; int T1[N1]={3,5,9,10,20, 30};
  int T2[N2] = {-4,7, 10, 1000} ; T3[N3] ;
```

```
while ((i<=N1-1) && (j<=N2-1)) {
    if (T1[i]<T2[j]) {T3[k]=T1[i]; i=i+1;}
    else {T3[k]=T2[j]; j=j+1;}
    K=k+1; }
```

```
While (i<=N1-1) {T3[k]=T1[i]; i=i+1; k=k+1;} //compléter par T1
While (j<=N2-1) {T3[k]=T2[j]; j=j+1; k=k+1;} // compléter par T2
```

```
for (i=0; i<=N3-1; i=i+1) printf("%d ", T3[i]); printf("\n");
return 0; }
```

Les algorithmes de Tri : tri par sélection

- Principe :
- -amener le $\min(T[1] \dots T[N])$ en $T[1]$
- -...
- -amener le $\min (T[I] \dots T[N])$ en $T[I]$
- Solution :
- Pour I de 1 jusqu'à $N-1$
- Faire
 - -rechercher les minimum parmi $T[I] \dots T[N]$
 - -agir pour que ce minimum soit en $T[I]$
- Fin pour

Les algorithmes de Tri

Tri par sélection

Exemple

T=

Itération 1 : T=

7	5	8	4	6
---	---	---	---	---

Itération 2 T=

4	5	8	7	6
---	---	---	---	---

Itération 3 T=

4	5	8	7	6
---	---	---	---	---

Itération 4 T=

4	5	6	7	8
---	---	---	---	---

4	5	6	7	8
---	---	---	---	---

```
#include <stdio.h>
#define N 11
main()
{ int T[N]={3,5,9,8, 6,-9,14, 5,7,-2,4};
  int I,J, x, min , P;
  for (I=0; I<= N-2; I=I+1) {
    min = T[I]; P=I;
    for (j=I+1; J<=N-1; J=J+1)
      if (T[J]<min) { min = T[J]; P=J; }
    if (P != I) {x=T[I]; T[I] = T[P]; T[P] = x;}
  }
  For (I=0; I<N; I=I+1) printf("%d : ", T[I]);
  return 0; }
```

Les algorithmes de Tri : tri par propagation

- Principe :
- Par opposition au tri précédent où l'on compare un élément particulier aux autres éléments, il s'agit ici de comparer deux éléments consécutifs (couples) et des les échanger s'ils ne sont pas dans le bon ordre.
- Solution :
- 1ère étape : comparer successivement et permuter si nécessaire
 - T1 et T2
 - T2 et T3
 - ...
 - T_{n-1} et T_n
- répéter tant qu'il y a des permutations

Les algorithmes de Tri : tri par propagation

Exemple

T=

7	5	8	4	6
---	---	---	---	---

Itération 1 : T=

5	7	4	6	8
---	---	---	---	---

Itération 2 : T=

5	4	6	7	8
---	---	---	---	---

Itération 3 : T=

4	5	6	7	8
---	---	---	---	---

Itération 4 : T=

4	5	6	7	8
---	---	---	---	---

```
#include <stdio.h>
#define N 11
#define true 1
#define false 0
Int main()
{ int T[N]={3,5,9,8, 6,-9,14, 5,7,-2,4};
    int I=1,J, x, Permut=true;
    While (permut == true) {
        permut=false;
        for (j=0; J<N-i; J=J+1)
            if (T[J]>T[J+1]) { x = T[J]; T[J]=T[J+1];
                T[J+1]=x; Permut=true; }
        i=i+1;
    }
    For (I=0; I<N; I=I+1) printf("%d : ", T[I]);
    return 0;
}
```

Les algorithmes de Tri : tri par Insertion

- Principe :
- -algorithme d'insertion d'une valeur dans un vecteur ordonné
- -un joueur aux cartes qui les classe par ordre, il insère chaque nouvelle carte ramassée à sa place parmi celles qu'il possède déjà
- -en supposant T_1, T_2, \dots, T_{i-1} ordonnés, on examine T_i et on l'insère à sa place.

Les algorithmes de Tri : tri par Insertion

Principe d'Algorithme

- Pour $i \leftarrow 2$ jusqu'à N faire
- Sauvegarder $T[i]$
- Chercher la position P de $T[i]$
- Décaler les cases précédant $T[i]$ à partir de P à droite d'une case
- Insérer $T[i]$ dans P
- Fin pour

Les algorithmes de Tri : tri par Insertion

Exemple :

T=

7	5	8	4	6
---	---	---	---	---

Itération 1) T=

5	7	8	4	6
---	---	---	---	---

Itération 2) T=

5	7	8	4	6
---	---	---	---	---

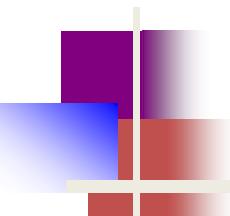
Itération 3) T=

4	5	7	8	6
---	---	---	---	---

Itération 4) T=

4	5	6	7	8
---	---	---	---	---

```
#include <stdio.h>
#define N 11
main()
{ int T[N]={3,5,9,8, 6,-9,14, 5,7,-2,4};
  int I,J, k,x;
  for (I=1; I<= N-1; I=I+1) {
    x = T[I];
    /* chercher la bonne position */
    J=0;  while (T[I]>T[J]) J=J+1;
    /* décalage */
    if (J != I) { for (k=I-1; K>=J; K=K-1;) T[K+1] = T[K];
      T[J] = x; /* insertion */
    }
  } For (I=0; I<N; I=I+1) printf("%d : ", T[I]); return 0; }
```



Les sous programmes Fonctions & Procédures

- Notion de sous-programmes
- Variables locales et variables globales
- Structure d'un programme
- Les fonctions
- Les procédures
- Applications

*Dans ce chapitre nous allons parler de programme et de sous-programme. Il faut comprendre ces mots comme programme **algorithmique** indépendant de toute implantation C*

Par exemple...

- Résoudre le problème suivant :
- Ecrire un programme qui affiche en ordre croissant les notes d'une promotion suivies de la note la plus faible, de la note la plus élevée et de la moyenne
- Revient à résoudre les problèmes suivants :
 - Remplir un tableau de naturels avec des notes saisies par l'utilisateur
 - Afficher un tableau de naturels
 - Trier un tableau de naturel en ordre croissant
 - Trouver le plus petit naturel d'un tableau
 - Trouver le plus grand naturel d'un tableau
 - Calculer la moyenne d'un tableau de naturels
- Chacun de ces sous problèmes devient un nouveau problème à résoudre
- Si on considère que l'on sait résoudre ces sous problèmes, alors on sait quasiment résoudre le problème initial

Sous programme

- Ecrire un programme qui résout un problème revient toujours à écrire des sous-programmes qui résolvent des sous parties du problème initial
- En algorithmique il existe deux types de sous-programmes :
 - Les fonctions
 - Les procédures
- Un sous-programme est obligatoirement caractérisé par un nom (un identifiant) unique
- Lorsqu'un sous programme a été explicité (on a donné l'algorithme), son nom devient une nouvelle instruction, qui peut être utilisé dans d'autres (sous-)programmes
- Le (sous-)programme qui utilise un sous-programme est appelé
- **(sous-)programme appelant**

Types de variables

- La **portée** d'une variable est l'ensemble des sous-programmes où cette variable est connue (les instructions de ces sous-programmes peuvent utiliser cette variable)
- Une variable définie au niveau du programme principal (celui qui résout le problème initial, le problème de plus haut niveau) est appelée **variable globale**
- Sa portée est totale : **tout** sous-programme du programme principal peut utiliser cette variable
- Une variable définie au sein d'un sous programme est appelée **variable locale**
- La portée d'une variable locale est uniquement le sous-programme qui la déclare
- Lorsque le nom d'une variable locale est identique à une variable globale, la variable globale est localement masquée
- Dans ce sous-programme la variable globale devient inaccessible

Structure d'un programme algorithmique

- Un programme doit suivre la structure suivante :

Programme *nom du programme*

Définition des constantes

Définition des types

Déclaration des variables globales

Définition des sous-programmes

début

instructions du programme principal

fin

Les paramètres

- Un paramètre d'un sous-programme est une variable locale particulière qui est associée à une variable ou constante (numérique ou définie par le programmeur) du sous-programme appelant :
 - Puisque un paramètre est une variable locale, un paramètre admet un type
 - Lorsque le (sous-)programme appelant appelle le sous-programme il doit indiquer la variable (ou la constante), de même type, qui est associée au paramètre
- Par exemple, si le sous-programme *sqrt* permet de calculer la racine carrée d'un réel:
 - Ce sous-programme admet un seul paramètre de type réel positif
 - Le (sous-)programme qui utilise *sqrt* doit donner le réel positif dont il veut calculer la racine carrée

Le passage de paramètres

- Il existe trois types d'association (que l'on nomme **passage de paramètre**) entre le paramètre et la variable du (sous-)programme appelant :
 - Le **passage de paramètre en entrée**
 - Le **passage de paramètre en sortie**
 - Le **passage de paramètre en entrée/sortie**

Le passage de paramètres en entrée

- Les instructions du sous-programme ne peuvent pas modifier l'entité (variable ou constante) du (sous-)programme appelant
- C'est le seul passage de paramètre qui admet l'utilisation d'une constante
- Par exemple :
 - le sous-programme **sqrt** permettant de calculer la racine carrée d'un nombre admet un paramètre en entrée
 - le sous-programme **écrire** qui permet d'afficher des informations admet un paramètre entré

Le passage de paramètres en sortie

- Les instructions du sous-programme affecte obligatoirement une valeur à ce paramètre (valeur qui est donc aussi affectée à la variable associée du (sous-)programme appelant)
- C'est pour cela qu'on ne peut pas utiliser de constante pour ce type de paramètre
- Par exemple :
 - le sous-programme **lire** qui permet de mettre dans des variables des valeurs saisies par l'utilisateur admet n paramètres en sortie

Le passage de paramètres en entrée/sortie

- Passage de paramètre qui combine les deux précédentes
- A utiliser lorsque le sous-programme doit utiliser et/ou modifier la valeur de la variable du (sous-) programme appelant
- Comme pour le passage de paramètre en sortie, on ne peut pas utiliser de constante
- Par exemple :
 - le sous-programme **échanger** qui permet d'échanger les valeurs de deux variables

Les fonctions

- Les fonctions sont des sous-programmes admettant des paramètres et retournant un **seul** résultat (comme les fonctions mathématiques $y=f(x,y,\dots)$)
 - les paramètres sont en nombre fixe ($>=0$)
 - une fonction possède un seul type, qui est le type de la valeur renvoyée
 - le passage de paramètre est **uniquement en entrée** : c'est pour cela qu'il n'est pas précisé lors de l'appel, on peut donc utiliser comme paramètre des variables, des constantes mais aussi des résultats de fonction
 - la valeur de retour est spécifiée par l'instruction **retourner**
- Généralement l'identifiant d'une fonction est un nom

Fonctions : syntaxe

- **fonction** *nom de la fonction (paramètre(s) de la fonction)* : *type de la valeur renvoyée*

Déclaration variables locales

début

*instructions de la fonction avec au moins une fois
l'instruction retourner*

fin

- On utilise une fonction en précisant son nom suivi des paramètres entre parenthèses
- Les parenthèses sont toujours présentes même lorsqu'il n'y a pas de paramètre

Exemple de programme

Programme *exemple1*

var a : Entier, b : entier

fonction val_abs (N : Entier) : entier

var v : entier

début

si N>=0 alors

 v \leftarrow N

Sinon v \leftarrow -N **fin si**

Retourner (v)

Fin val_abs

début

 écrire("Entrez un entier :")

 lire(a)

 B \leftarrow val_abs(a)

 écrire("la valeur absolue de ",a," est ",b)

Fin exemple1

Paramètre **formel** sert à
Définir la fonction

Lors de l'exécution de la fonction
Val_abs, la variable *a* et le paramètre
N sont associés par un passage
de paramètre en entrée : La
valeur de *a* est copiée dans *N*

Paramètre **effectif** sert à
exécuter la fonction

Les procédures

- Les procédures sont des sous-programmes qui ne retournent **aucun** résultat
- Par contre elles admettent des paramètres avec des passages :
 - en entrée, prefixés par **Entrée** (ou **E**)
 - en sortie, prefixés par **Sortie** (ou **S**)
 - en entrée/sortie, prefixés par **Entrée/Sortie** (ou **E/S**)
- Généralement le nom d'une procédure est un verbe

Syntaxe

procédure *nom de la procédure* (**E** paramètre(s) en entrée; **S** paramètre(s) en sortie; **E/S** paramètre(s) en entrée/sortie)

Déclaration *variable(s) locale(s)*

début

Instructions de la procédure

fin

- Et on appelle une procédure comme une fonction, en indiquant son nom suivi des paramètres entre parenthèses

Exemple

Programme *exemple2*

var a : Entier, b : entier

procédure echanger (E/S val1 Entier; E/S val2 Entier)

var temp : Entier

Début

temp \leftarrow val1

val1 \leftarrow val2

val2 \leftarrow temp

fin

Début

écrire("Entrez deux entiers :")

lire(a,b)

echanger(a,b)

écrire("a=",a," et b = ",b)

Fin exemple2

Lors de l'exécution de la procédure *echanger*, la variable *a* et le paramètre *val1* sont associés par un passage de paramètre en entrée/sortie : Toute modification sur *val1* est effectuée sur *a* (de même pour *b* et *val2*)

Application 1

1. Ecrire une fonction **parfait** qui accepte comme paramètre entré un entier k et qui permet de vérifier si k est parfait ou non (résultat logique) ?
2. Ecrire un programme algorithmique nommé NombreParfaits permettant de faire appel à la fonction parfait pour afficher tous les nombres parfaits ≤ 1000 ?

Solution

fonction **parfait(k : entier) : logique**

Var

 S, i : entier

Début

 s \leftarrow 0

 pour i \leftarrow 1 jusqu'à k div 2

 faire

 si k mod i = 0 alors

 s \leftarrow s + i

 fin si

 fin faire

 si s=k alors retourner (vrai)

 sinon retourner (faux)

 fin si

Fin

Algorithme NombreParfaits

Var

 i : entier

Début

 pour i \leftarrow 1 jusqu'à 1000

 faire

 si (parfait (i)=vrai)

 alors

 écrire (i)

 fin si

 fin faire

fin NombreParfaits

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

```
int parfait(int k);
int main() {
    int i,r;
    for (i=1; i<=1000; i++) {
        r= parfait(i);
        if (r == 1) printf ("%d\n", i);}
}
```

```
int parfait(int k)
{
    int i, s=0;
    for (i=1; i<=k/2; i++)
        if (k % i == 0) s = s + i;
    if (s == k) return 1;
    else return 0;
}
```

Solution 2 avec procédure

procédure **parfait**(E k : entier, S p : logique)

Var

 S, i : entier

Début

 s \leftarrow 0

 pour i \leftarrow 1 jusqu'à k div 2

 faire

 si k mod i = 0 alors

 s \leftarrow s + i

 fin si

 fin faire

 si s=k alors p \leftarrow vrai

 sinon p \leftarrow faux

 fin si

Fin

Algorithme NombreParfaits

Var

i : entier

r : logique

Début

 pour *i* \leftarrow 1 jusqu'à 1000

 faire

parfait (*i*, *r*)

 si *r* = vrai alors

 écrire (*i*)

 fin si

 fin faire

fin NombreParfaits

```
#include <stdio.h>

void parfait(int k, int *y);
int main() {
    int i,r;
    for (i=1; i<=1000; i++) {
        parfait(i,&r);
        if (r == 1) printf (" %d\n", i);}
    }
```

```
void parfait(int k, int *y)
{
    int i, s=0;
    for (i=1; i<=k/2; i++)
        if (k % i == 0) s = s + i;
    if (s == k) *y= 1;
    else *y= 0;
}
```

Fonctions récursives

- **Exemple : N !**

« le produit des entiers de 1 à N (pour $N \geq 1$) est égal à 1 si $N=1$, à N multiplié par le produit des entiers de 1 à $N-1$ sinon ».

- ***fonction factorielle (n: entier) : entier***

début

si n = 0 alors

retourner 1

sinon

*retourner n * factorielle(n-1)*

finsi

Fin factorielle

```
Int factorielle (int N)
{
    if (N==0)
        return 1;
    else
        return N*factorielle(N-1);
}
```

- Une fonction récursive est une fonction qui s'utilise elle-même : elle déduit le traitement d'une donnée ou d'un ensemble de données du même traitement appliqué à une donnée plus petite ou à un ensemble de données moins nombreuses jusqu'à obtenir un cas évident

Fonctions récursives

- Exemple : **suites numériques**

$U_0 \leftarrow 1$

$U_1 \leftarrow 2$

$U_{n+2} \leftarrow U_{n+1} + U_n$

```
int suite ( int N)
{ if (N==0) return 1;
else
    if (N == 1) return 2;
    else return (suite(N-1) + suite(N-2));
}
```

Fonctions récursives

- **Exercices**
- 1. Ecrire une fonction récursive qui calcule la somme des éléments d'un tableau.
 - Deux paramètres : un tableau d'entiers tab et un nombre N. Le but de la fonction est de renvoyer la somme des éléments du tableau.
 - cas de base : taille du tableau = 1
 - variation du compteur à chaque appel : +1
- 2. Ecrire une fonction récursive qui calcule la somme des n premiers carrés. Par exemple, si n vaut 3, cette fonction calculera $1^2 + 2^2 + 3^2$.
 - Ce sous programme n'est défini que pour un n supérieur à 0.
 - Un seul paramètre n, qui doit être positif.
 - cas de base : n=1.
 - variation de n à chaque appel : +1

```
int somme(int *tab; int N)
{
    if (N==1) return tab[0];
    else return (somme(tab, N-1) + tab[N-1]);
}
```

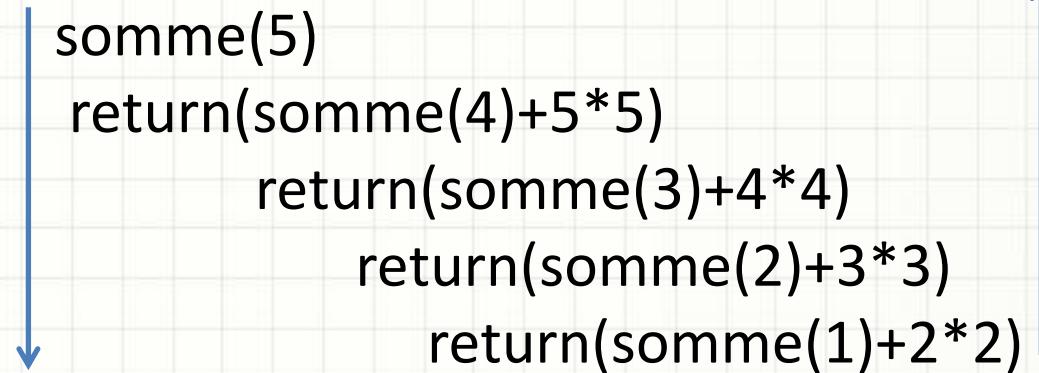
A call graph diagram showing the execution flow of the `somme` function. A vertical blue line with a downward-pointing arrow on its left side represents the call stack. To the right of the stack, five horizontal arrows point upwards, each labeled with a return value from a recursive call. The top-most arrow is labeled `somme(T1,4)`. The second arrow is labeled `return(somme(T1,3)+6)`. The third arrow is labeled `return(somme(T1,2)-1)`. The fourth arrow is labeled `return(somme(T1,1)+5)`. The bottom-most arrow is labeled `return(4)`.

```
somme(T1,4)
return(somme(T1,3)+6)
return(somme(T1,2)-1)
return(somme(T1,1)+5)
return(4)
```

```
main()
{
    int T1[]={4,5,-1,6}, x;
    x = somme( T1,4);
    printf("%d", x);
    ...
}
```

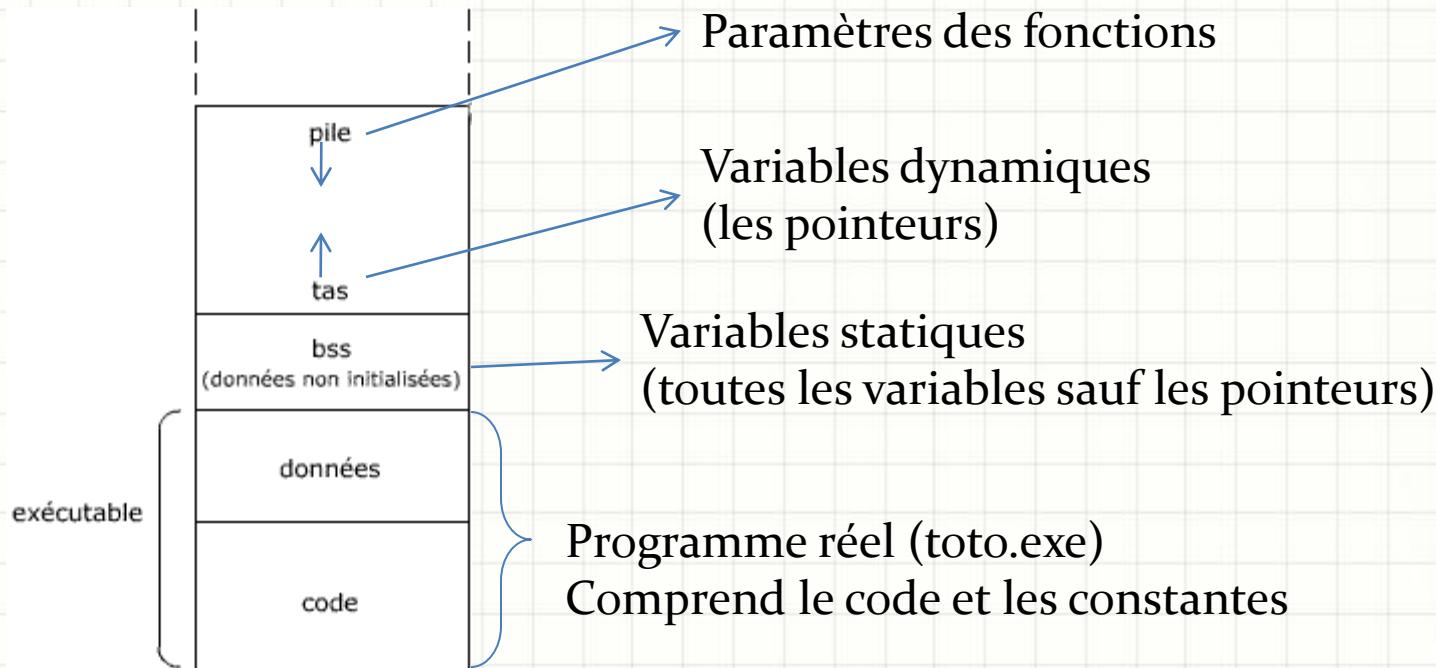
```
int somme(int N)
{
    if (N==1) return 1;
    else return somme(N-1)+N*N;
}
```

```
main()
{
    int N=5;
    somme(N);
    ...
}
```



Le matériel

• Organisation de la mémoire

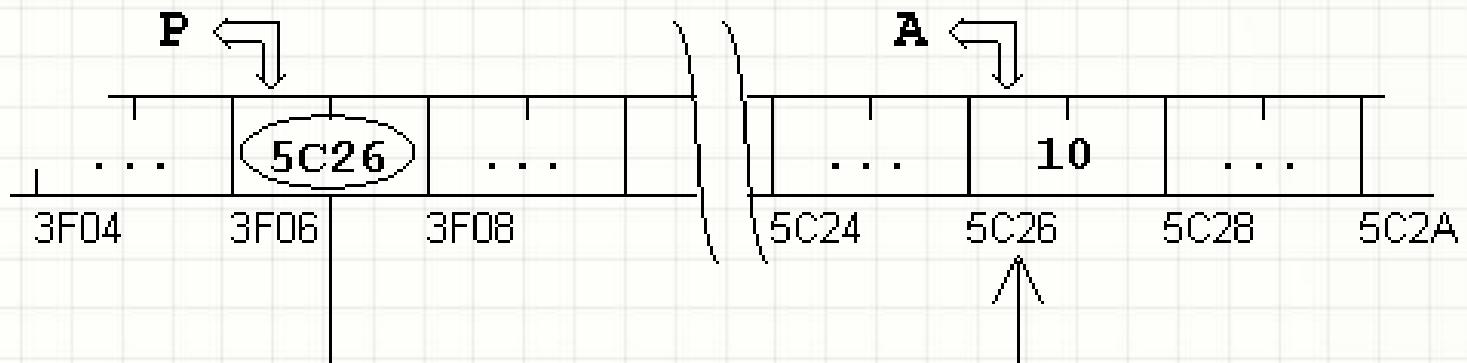


Pointeurs

- Un pointeur, c'est une variable dont la valeur est l'adresse d'une cellule de la mémoire
- Traduction :
 - Imaginons que la mémoire de l'ordinateur, c'est un grand tableau.
 - Un pointeur, c'est alors une variable de type **int** (un nombre) qui permet de se souvenir d'une case particulière.
- On comprend bien que le pointeur ne se soucis pas de savoir ce qu'il y a dans la case, mais bien de l'adresse de la case.

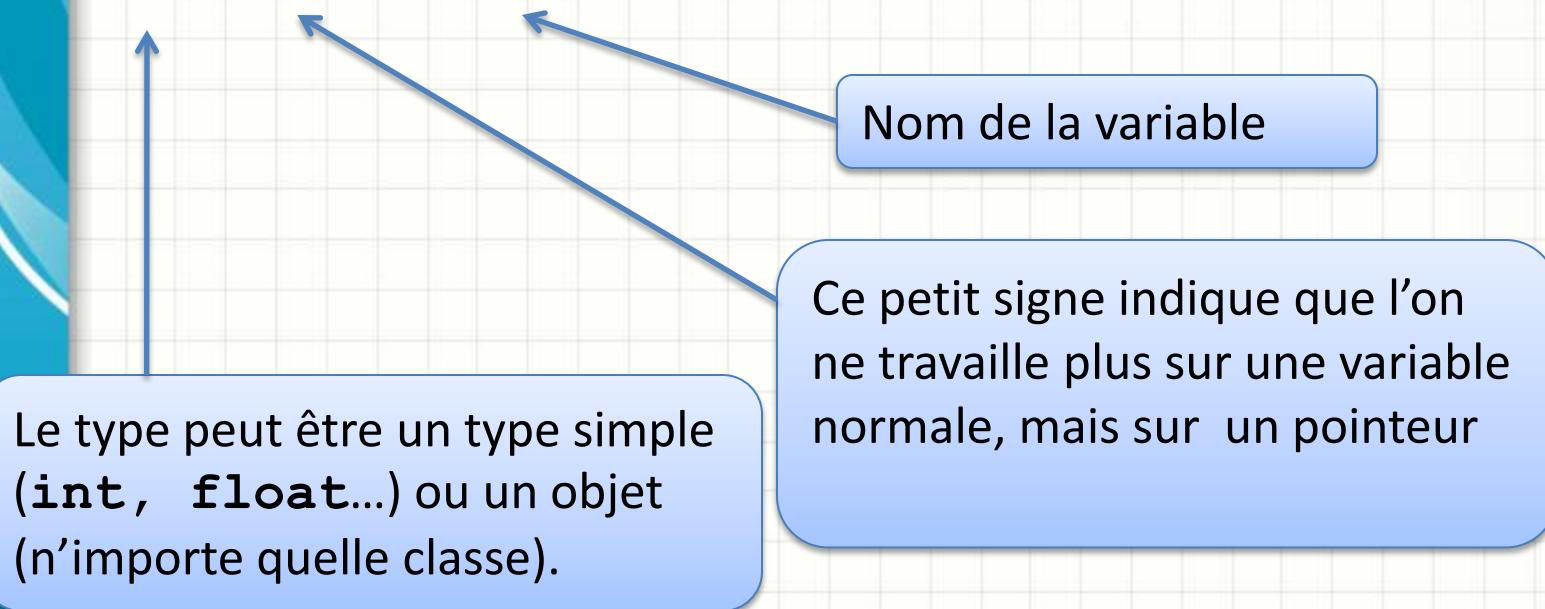
Pointeurs

- Outre l'utilisation d'une variable à travers son nom A, nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée **pointeur**. Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.
- **Adressage indirect:** Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.
- **Exemple**
- Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



Syntaxe

- Déclaration : *
- **type *idPtr;**



Syntaxe

- Opérateur d'adresse : &

```
printf('' %d '' , &idVar) ;
```

Plutôt que d'afficher la valeur de la variable, on affiche l'adresse de la case mémoire

Ce petit signe indique que l'on veux récupérer l'adresse d'une variable

Nom d'une variable de n'importe quel type

Syntaxe

- Opérateur de déréférencement : *

— `printf(" %d ", *p);`

Nom d'un pointeur

Ce petit signe indique que l'on veux récupérer la valeur située à une adresse précise

Plutôt que d'afficher l'adresse du pointeur, on affiche la valeur qui est dans la case mémoire

Syntaxe

- Exemple :

```
char c = 'a' ;
```

On déclare une variable normale
de type caractère

```
char *p;
```

On déclare un pointeur sur une
case de type caractère

```
p=&c;
```

On donne l'adresse de la variable
c au pointeur **p**

```
printf( ' %c' , *p) ;
```

On affiche la valeur de la case
pointée par **p**, elle est égale à ('a')

Exemple :

```
#include <stdio.h>
int main() {
    int A = 10, B=50;
    int *p;
    p= &A;
    B = *p;
    *p = 50;
    printf("A= %d et B = %d",A, B);
    return 0;
}
```

$$A = 50 \text{ et } B = 10$$

Pointeurs et tableaux

- En C, il existe une relation très étroite entre tableaux et pointeurs.
Chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.
- En général, les versions formulées avec des pointeurs sont plus compactes et plus efficientes, surtout à l'intérieur de fonctions.
- En C le nom d'un tableau représente l'adresse de son premier élément. En d'autres termes:

&TAB[0] et **TAB**

sont une seule et même adresse. En simplifiant, nous pouvons retenir que *le nom d'un tableau est un pointeur constant sur le premier élément du tableau*.

- ***Exemple***

En déclarant un tableau A de type **int** et un pointeur P sur **int**,

int A[10]; int *P;

- l'instruction: **P = A;** est équivalente à **P = &A[0];**

Exercice

Soit P un pointeur qui "pointe" sur un tableau A:

int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};

int *P;

P = A; // ou P = &A[0]

Quelles valeurs ou **adresses** fournissent ces expressions:

a- $*P+2$ = 14

b- $*(P+2)$ = 34

c- $P+1$ = adresse du 1^{er} élément + 1

d- $\&A[4]-3$ = $\&A[1]$

e- $A+3$ = $\&A[3]$

f- $\&A[7]-P$ = 7

g- $*(P+(*P-10))$ = 34

Exercice

Ecrire un programme qui lit deux tableaux A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A.

Utiliser le formalisme pointeur à chaque fois que cela est possible.

Solution

```
#include <stdio.h>
main()
{
int I,N,M,A[100], B[50];
printf("Dimension du tableau A) : ");
scanf("%d", &N );
for (I=0; I<N; I++)
{   printf("Elément %d : ", I);
    scanf("%d", A+I);  &A[I]  }
printf("Dimension du tableau B: ");
scanf("%d", &M );
for (I=0; I<M; I++)  {
printf("Elément %d : ", I);
scanf("%d", B+I);  }
```

```
/* Affichage des tableaux */
printf("Tableau donné A :\n");
for (I=0; I<N; I++)  A[I]
printf("%d ", *(A+I)); printf("\n");
printf("Tableau donné B :\n");
for (I=0; I<M; I++)
printf("%d ", *(B+I)); printf("\n");
```

```
/*Copie de B à la fin de A */
for (I=0; I<M; I++)
*(A+N+I) = *(B+I);
A[N+I]=B[I];
N = N+M; /* Edition du résultat */
printf("Tableau résultat A :\n");
for (I=0; I<N; I++)
printf("%d ", *(A+I)); A[I]
printf("\n");
return 0;}
```

Application

Écrire un programme en langage C qui range **les éléments d'un tableau A** du type int dans l'ordre inverse.

Le programme utilisera des pointeurs P1 et P2 et une variable numérique **aux** pour **la permutation** des éléments.

Solution

```
#include<stdio.h>
main()
{ int t[10],n,aux,i,j;
  printf("donnez la dimension du tab");
  scanf("%d",&n);
  for(i=0;i<n;i=i+1)
    {printf("donnez les valeurs t[%d]=",i);
     scanf("%d",&t[i]);}
```

```
i=0; j=n-1;
while (i<j)
{ aux=t[i];
  t[i] = t[j];
  t[j]=aux;
  i++; --j; }
```

```
for(i=0;i<n;i=i+1)
  printf("t[%d]=%d",i,t[i]);
}
```

Solution

```
#include<stdio.h>
main()
{ int t[10],n,aux,*i,*j;
  printf("donnez la dimension du tab");
  scanf("%d",&n);
  for(i=0;i<n;i=i+1)
    {printf("donnez les valeurs t[%d]=",i);
     scanf("%d",&t[i]);}
```

```
i=0; j=n-1;
while (i<j)
{ aux=t[i];
  t[i] = t[j];
  t[j]=aux;
  i++; --j; }
```

```
for(i=0;i<n;i=i+1)
  printf("t[%d]=%d",i,t[i]);
}
```

Solution

```
#include<stdio.h>
main()
{ int t[10],*p1,*p2,n,aux,i;
  printf("donnez la dimension du tab");
  scanf("%d",&n);
  for(i=0;i<n;i=i+1)
  {printf("donnez les valeurs t[%d]=",i);
   scanf("%d",t+i);}


p1=t;



p2=t+n-1;



while (p1<p2)


{ aux=*p1;
  *p1=*p2;
  *p2=aux;
  p1=p1+1;
  p2=p2-1;}


for(i=0;i<n;i=i+1)


  printf("t[%d]=%d",i,(t+i));}
```

Les fonctions en C

- Dans les langages de programmation il existe deux techniques de passage d'arguments :
 - par adresse,
 - par valeur.
- Des langages comme Fortran ou PL/1 ont choisi
- la 1^{ère} résolution, tandis qu'un langage comme Pascal offre les deux possibilités au programmeur.
- Le langage C a choisi la 2^esolution.
- Si un argument doit être passé par adresse, c'est le programmeur qui en prend l'initiative et ceci grâce à l'opérateur d'adressage (&).

Exemple 1:

```
#include <stdio.h>
void main()
{
    int a1, b1, c1;
    int somme(int a, int b);
    a1 = 3;
    b1 = 8;
    c1 = somme (a1, b1);
    printf("Somme de a1 et b1 : %d\n", somme(a1,b1));
}
```

```
int somme(int a, int b)
{ return(a + b);}
```

Exemple 1:

```
#include <stdio.h>
void main()
{
    int a1, b1, c1;
    void somme(int a, int b, int *c)
    a1 = 3;
    b1 = 8;
    somme(a1, b1, &c1);
    printf("Somme de a1 et b1 : %d\n", c1);
}
```

```
void somme(int a, int b, int *c)
{ *c = a + b;}
```

Exemple 2:

- Fonction de permutation de deux variables

```
void echanger_valeur(int *p1, int *p2)
```

```
{
```

```
    int tmp = *p1;  
    *p1 = *p2;  
    *p2 = tmp;
```

```
}
```

```
//exemple d'appel
```

```
int x = 5;
```

```
int y = 7;
```

```
echanger_valeur(&x, &y);  
printf("%d %d",x,y);
```

/*x vaudra 7 et y vaudra 5 après l'exécution*/

Ex: multiplication russe

Si A est pair alors

$$A * B = (A/2) * (B*2)$$

Si A est impair alors

$$A * B = (A-1) * B + B$$

Ecrire une fonction **Produit** (procédure) qui calcul $A*B$

Ecrire une fonction main appelant **Produit** pour calculer le produit de deux entier quelconques X et Y ?

Ex: multiplication russe

Si A est pair alors

$$A * B = (A/2) * (B*2)$$

Si A est impair alors

$$A * B = (A-1) * B + B$$

Exemple

$$A=12 \quad B = 5$$

$$\begin{aligned} A*B &= 6 * 10 \\ &= 3 * 20 \\ &= 2 * 20 + 20 \\ &= 1 * 40 + 20 \\ &= 0 * 40 + 40 + 20 \end{aligned}$$

Ex: multiplication russe

```
int Produit (int A, int B)
{
    int P=0;
    while (A != 0)
    {
        if (A%2 == 0)
            { A=A/2; B = B*2; }
        else{A = A -1; P = P + B;}
    }
    return P;
}
```

```
#include <stdio.h>
main()
{
    int Produit (int A, int B)
    int X, Y, P1;
    printf("donner X et Y");
    scanf("%d", &X);
    scanf("%d", &Y);
    P1 = Produit(X, Y);
    printf("Produit=%d",P1);
}
```

Ex: multiplication russe

```
void Produit (int A, int B, int *P)
{
    *P=0;
    while (A != 0)
    {
        if (A%2 == 0)
            { A = A/2; B = B*2; }
        else
            {A = A -1; *P = *P + B;}
    }
}
```

```
#include <stdio.h>
main()
{
    void Produit (int A, int B, int *P);
    int X, Y, P1;
    printf("donner X et Y");
    scanf("%d", &X);
    scanf("%d", &Y);
    Produit(X, Y, &P1);
    printf("Produit=%d",P1);
}
```

Exercice :

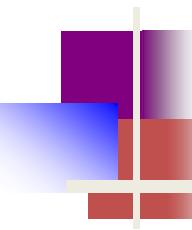
Ecrire une fonction sans type appelée ORDRE qui prend comme paramètres trois entiers A, B et C et qui permet de réarranger leurs contenus de façon que A soit inférieur à B et B soit inférieur à C ?

Tester la fonction dans main() ?

Exemple :

A=10 B=20 C=15 → A=10 B=15 C=20

A=1 B=-1 C=-10 → A=-10 B=-1 C=1



```
min = A;
```

```
If (min > B) min = B;
```

```
If (min > C) min = C;
```

```
If (min == B) {x = A; A=B; B=x;}
```

```
If (min == C) {x = A; A=C; C=x;}
```

```
If (B > C) {x=B; B=C; C=x;}
```

```
void ORDRE (int *A, int *B, int *C)
{
    int min,a;
    min = *A;
    if (min > *B) min = *B;
    if (min > *C) min = *C;
    if (min == *B) {a=*A; *A=*B; *B=a;}
    if (min == *C) {a=*A; *A=*C; *C=a;}
    if (*B>*C) {a=*B; *B=*C; *C=a;}
}

int main()
{
    int x,y,z;
    printf("donner x y z \n");
    scanf("%d",&x);scanf("%d",&y);scanf("%d",&z);
    printf("Avant : x = %d y = %d z = %d\n",x,y,z);
    ORDRE (&x, &y, &z);
    printf("Après : x = %d y = %d z = %d\n",x,y,z);
}
```

```
void ORDRE (int *A, int *B, int *C)
{
    int min,a;
    min = *A;
    if (min > *B) min = *B;
    if (min > *C) min = *C;
    if (min == *B) {a=*A; *A=*B; *B=a;}
    if (min == *C) {a=*A; *A=*C; *C=a;}
    if (*B>*C) {a=*B; *B=*C; *C=a;}
}

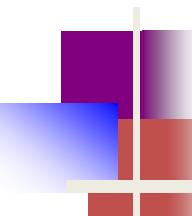
int main()
{
    int *x,*y,*z; int x1, y1, z1;
    printf("donner x y z \n");
    scanf("%d",&x1);scanf("%d",&y1);scanf("%d",&z1);
    *x = x1; *y = y1; *z = z1;
    printf("Avant : x = %d y = %d z = %d\n",*x,*y,*z);
    ORDRE (x, y, z);
    printf("Après : x = %d y = %d z = %d\n",*x,*y,*z);
}
```

Exercice :

Ecrire deux fonctions qui calculent la valeur X^N pour une valeur réelle X (type **float**) et une valeur entière positive N (type **int**) :

- a) EXP1 retourne la valeur X^N comme résultat.
- b) EXP2 affecte la valeur X^N à X.

Ecrire un programme en langage C qui teste les deux fonctions à l'aide de valeurs lues au clavier ?



```
float EXP1 (float X, int N);
int main()
{
    float X; int N;
    printf("donnner un tréel \n");
    scanf("%f",&X);
    printf("donnner un entier positif \n");
    scanf("%d",&N);
    printf("%f puissance %d = %f\n", X, N, EXP1(X, N)); }
```

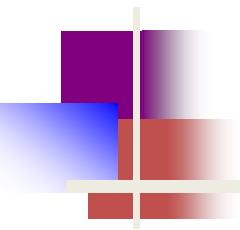
```
float EXP1 (float X, int N)
{
    float P=1; int i;
    for (i=1; i<=N; i++) P = P*X;
    return P;
}
```

```
void EXP2 (float *X, int N)
{
    float P=1; int i;
    for (i=1; i<=N; i++) P = P* (*X);
    *X = P;
}
```

```
int main()
{   float X; int N1;
    printf("donner un réel \n");
    scanf("%f",&X);
    printf("donner un entier positif \n");
    scanf("%d",&N1);
    EXP2(&X, N1);
    printf("Résultat = %f\n", X); }
```

Exercice :

Ecrire une fonction sans type ADDITION permettant de calculer l'addition de deux tableaux de dimension N où N est une constante à déclarer ?



```
void ADDITION (int *T1, int *T2, int *T3, int N);  
void ADDITION (int T1[], int T2[], int T3[], int N);
```

Tester la fonction dans main() ?

Exercice

- Fonction d'Addition de deux vecteurs - Appel dans la fonction main

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

```
#define N1 6
```

```
Void addition(int *V1, int *V2, int *V3, int N)
```

```
{ int i;
```

```
    for (i=0; i<N; i++) V3[i]=V1[i]+V2[i]; // *(V3+i)=*(V1+i) + *(V2+i);
```

```
}
```

```
void main()
```

```
{ int i;
```

```
    int T1[N1];    int T2[N1];    int T3[N1];
```

```
    for (i=0;i<N1; i++) {
```

```
        printf("donner une valeur"); scanf("%d",T1+i);
```

```
        printf("donner une valeur"); scanf("%d",T2+i); }
```

```
    addition (T1, T2, T3, N1);
```

```
    for (i=0; i<N1; i++) printf ("%d : ",T3[i]);
```

```
}
```

Passage d'arguments à la fonction main

- Lorsqu'un exécutable est lancé sous un interpréteur de commandes (shell), un processus est créé et son exécution commence par la fonction main à laquelle des arguments sont transmis après avoir été générés par le shell.
- Ces arguments sont constitués de :
 - ceux fournis au lancement de l'exécutable,
 - leur nombre (y compris l'exécutable),
 - L'environnement du shell.
- Les premier et dernier sont transmis sous forme de vecteurs de pointeurs de caractères.

Passage d'arguments à la fonction main

- Par convention :
 - **argc** désigne le nombre d'arguments transmis au moment du lancement de l'exécutable,
 - **argv** désigne le vecteur contenant les différents arguments,
 - **envp** désigne le vecteur contenant les informations sur l'environnement.
- Les arguments précédents sont transmis à la fonction main dans cet ordre.

Passage d'arguments à la fonction main

```
#include <stdio.h>

main( int argc, char *argv[], char *envp[] )
{
    for( i=0;i<=argc;i++)
        printf( "%s\n", argv [i]);
    printf("\n");
    i=0;
    while (env[i] != NULL)
    {
        printf("%s : ",env[i]); i++;
    }
}
```

Signed - unsigned



- Les deux mots-clés **unsigned** et **signed** peuvent s'appliquer aux types caractère et entier pour indiquer si le **bit de poids fort** doit être considéré ou non comme un bit de signe.
- Les entiers sont signés par défaut, tandis que les caractères peuvent l'être ou pas suivant le compilateur utilisé.
- Une déclaration telle que `unsigned char` permettra de designer une quantité comprise entre 0 et 255, tandis que `signed char` désignera une quantité comprise entre -128 et +127.
- De même `unsigned long` permettra de designer une quantité comprise entre 0 et $2^{32}-1$, et `long` une quantité comprise entre -2^{31} et $2^{31}-1$.

Constantes

- Les constantes symboliques

const int x = 10; ou **#define x 10**

- Les constantes chaînes de caractère : une suite de caractères entre guillemets.
- En mémoire cette suite de caractères se termine par le caractère NULL ('\0').
- Ne pas confondre "A" et 'A' qui n'ont pas du tout la même signification !
- Pour écrire une chaîne de caractères sur plusieurs lignes on peut :
 - soit terminer chaque ligne par \,
 - soit la découper en plusieurs constantes chaîne de caractères, le compilateur effectuera automatiquement la concaténation.

Constantes chaîne de caractères

Exemples :

1- `char *chaine = "\`

-----\

Pour écrire une chaîne sur plusieurs lignes, \

il suffit de terminer chaque ligne par \n\

-----\n";

2- `char *chaine = "écriture d'une "`

`"chaîne de caractères "`

`"sur plusieurs "`

`"lignes\n\n";`

Exercices :

- Ecrire un programme qui lit un verbe régulier en "er" au clavier et qui en affiche la conjugaison au présent de l'indicatif de ce verbe. Contrôlez s'il s'agit bien d'un verbe en "er" avant de conjuguer. Utiliser les fonctions **strcat (s1,s2)** et **strlen(s)**.

strcat (s1,s2) ajouter s2 à la fin de s1

exemple : s1=manger s2=marcher → s1=mangermarcher

Exemple:

- **Verbe : fêter**
je fête
tu fêtes
il fête
nous fêtons
vous fêtez
ils fêtent

Corrigé :

```
#include <stdio.h>
#include <string.h>
main()
{
    char VERB[20];
    char AFFI[30];
    int L;
    printf("Verbe : "); scanf ("%s",VERB);

    /* Contrôler s'il s'agit d'un
       verbe en 'er' */
    L=strlen(VERB);
    if ((VERB[L-2]!='e') || (VERB[L-1]!='r'))
        printf("\n Non de premier groupe.!");
    else {
```

```
        /* Couper la terminaison
           'er'. */
        VERB[L-2]='\0';
        /* Conjuguer ... */
        AFFI[0]='\0';
        strcat(AFFI, "je ");
        strcat(AFFI, VERB);
        strcat(AFFI, "e");
        printf("%s\n",AFFI);
        ...
        AFFI[0]='\0';
        strcat(AFFI, "ils ");
        strcat(AFFI, VERB);
        strcat(AFFI, "ent");
        printf("%s\n",AFFI);
    }
```

Exercice :

1. Ecrire une fonction my_concat qui accepte deux chaînes de caractères CH1 et CH2 et qui copie la première moitié de CH1 et la première moitié de CH2 dans une troisième chaîne CH3 ? *NB : utiliser les fonctions strncat(s,t,n) et strncpy(s,t,n)*
void my_concat(char *ch1, char *ch2, char *ch3) ?

Exemple:

ch1=Bonjour

ch2=Bonsoir

ch3=BonBon

2. Appeler cette fonction dans main() ?

Corrigé

- Ecrire une fonction my_concat qui deux chaînes de caractères CH1 et CH2 et qui copie la première moitié de CH1 et la première moitié de CH2 dans une troisième chaîne CH3 ? appeler cette fonction dans main() ?

```
void my_concat(char *ch1, char *ch2, char *ch3)
{ strncpy(ch3, ch1, strlen(ch1)/2);
  strncat(ch3, ch2, strlen(ch2)/2); }
```

```
int main()
{ char chaine1[20],chaine2[20], chaine3[20];
  printf("donner deux chaines de caractères\n");
  scanf("%s",chaine1); scanf("%s",chaine2);
  my_concat(chaine1, chaine2, chaine3);
  printf("Un demi %s plus un demi %s donne %s\n", chaine1, chaine2, chaine3);
  return 0;
}
```

Exercices :

1. Ecrire une fonction `my_strcpy` qui permet de copier une chaîne `ch2` dans une autre chaîne `ch1` sans utiliser la fonction `strcpy` ? **void my_strcpy(char *ch1, char *ch2)**, appeler cette fonction dans `main()` ?
2. Ecrire la fonction **void coller_chaine(char *src, char *dest)** qui recopie les caractères de la chaîne `src` à **la fin** de la chaîne `dest` (fonction de concaténation) ? tester la fonction dans `main()` ? *Sans utiliser la fonction strcat*
3. On dispose de deux tableaux, `rang[]` un tableau de rangs et `*nom[]` un tableau de noms d'élèves. Ecrire une fonction **Tri** qui réalise le tri du tableau `rang[]` en mettant à jour le tableau `*nom[]` : **void tri(int rang[], int N, char *nom[])**

Exercice 1

- Ecrire une fonction my_strcpy qui permet de copier une chaîne ch2 dans une autre chaîne ch1 sans utiliser la fonction strcpy ? void **my_strcpy(char *ch1, char *ch2)**, appeler cette fonction dans main() ?

```
void my_strcpy(char *ch1, char *ch2)
{
    int i=0;
    while (ch2[i]!='\0')
        {ch1[i]=ch2[i]; i++;}
    ch1[i]= '\0';
}

void my_strcpy(char *ch1, char *ch2);
int main()
{
    char chaine1[20],chaine2[20];
    printf("donner deux chaines de caractères\n");
    scanf("%s",chaine2);
    my_strcpy(chaine1, chaine2);
    printf("%s",chaine1);    return 1; }
```

Exercice 1

- Ecrire une fonction my_strcpy qui permet de copier une chaîne ch2 dans une autre chaîne ch1 sans utiliser la fonction strcpy ?

```
char * my_strcpy(char *ch1, char *ch2)
```

appeler cette fonction dans main() ?

Exercice 1 bis

- Ecrire une fonction my_strcpy qui permet de copier une chaîne ch2 dans une autre chaîne ch1 sans utiliser la fonction strcpy ? **char * my_strcpy(char *ch1, char *ch2)**, appeler cette fonction dans main() ?

char *my_strcpy(char *ch1, char *ch2)

```
{ int i=0;  
while (ch2[i]!='\0')  
    {ch1[i]=ch2[i]; i++;} // {*(ch1+i)=*(ch2+i); i++;}  
ch1[i]= '\0'; return ch1;  
}
```

int main()

```
{ char chaine1[20],chaine2[20];  
printf("donner deux chaînes de caractères\n");  
scanf("%s",chaine1); scanf("%s",chaine2);  
printf("%s\n", my_strcpy(chaine1, chaine2));  
return 1; }
```

Exercice 2

- Ecrire la fonction **coller_chaine(char *src, *dest)** qui recopie les caractères de la chaîne src à **la fin** de la chaîne dest (fonction de concaténation) ?
tester la fonction dans main() ?

```
void coller_chaine(char *src, char *dest)
```

```
{ int i=0,j=0;  
    while (dest[j]!='\0') j++;  
    while (src[i]!='\0') {  
        dest[j+i]=src[i];      i++;}  
    dest[j+i]='\0'; }
```

```
int main()
```

```
{   char chaine1[10],chaine2[20];  
    printf("donner une chaîne\n");  scanf("%s",chaine1);  
    printf("donner une chaîne\n");  scanf("%s",chaine2);  
    coller_chaine(chaine1, chaine2);  
    printf("%s : %s\n",chaine1, chaine2);  
    return 1; }
```

Exercice 3

- On dispose de deux tableaux, rang[] un tableau de rangs et *nom[] un tableau de noms d'élèves. Ecrire une fonction Tri qui réalise le tri du tableau rang[] en mettant à jour le tableau *nom[] : void tri(int rang[], int N, char *nom[])

void tri(int rang[], int N, char *nom[])

{char aux[20];

int i,j,P,min;

for (i=0;i<N-2;i++)

{ min=rang[i]; j=i+1;

for (j=i+1;j<N-1; j++)

{if (rang[j]<min) {min=rang[j] ;P=j;}

if (P!=i) { rang[P]=rang[i]; rang[i]=min;

.....

.....

.....

} }

Exercice 3

- On dispose de deux tableaux, rang[] un tableau de rangs et *nom[] un tableau de noms d'élèves. Ecrire une fonction Tri qui réalise le tri du tableau rang[] en mettant à jour le tableau *nom[] : void tri(int rang[], int N, char *nom[])

```
void tri(int rang[], int N, char *nom[])
```

```
{char aux[20];
int i,j,P,min;
for (i=0;i<N-2;i++)
{ min=rang[i]; j=i+1;
  for (j=i+1;j<N-1; j++)
    {if (rang[j]<min) {min=rang[j] ;P=j;}
     if (P!=i) { rang[P]=rang[i]; rang[i]=min;
      strcpy(aux,nom[i]);
      strcpy(nom[i],nom[P]);
      strcpy(nom[P],aux); }
    }
```

Allocation dynamique

Un tableau est alloué de manière statique : nombre d'éléments constant.

Alloué lors de la compilation (avant exécution)

problème pour déterminer la taille optimale, donnée à l'exécution

- surestimation et perte de place
- de plus, le tableau est un pointeur constant.

Il faudrait un système permettant d'allouer un nombre d'éléments connu seulement à l'exécution : c'est **l'allocation dynamique**.

Allocation dynamique

Faire le lien entre le pointeur non initialisé et une zone de mémoire de la taille que l'on veut.

On peut obtenir cette zone de mémoire par l'emploi de **malloc**, qui est une fonction prévue à cet effet.

Il suffit de donner à **malloc** le nombre d'octets désirés (attention, utilisation probable de sizeof), et **malloc renvoie un pointeur de type void* sur la zone de mémoire allouée.**

Si malloc n'a pas pu trouver une telle zone mémoire, il renvoie NULL.

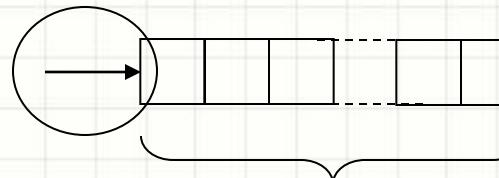
Appel par : malloc(nombre_d_octets_voulus);

Allocation dynamique

Symbolisation de l'effet de malloc:

si on utilise par exemple malloc(n), on a :

malloc() renvoie le pointeur



Zone de n octets

Pour accéder à cette zone, il faut impérativement l'affecter à un pointeur existant. On trouvera donc **toujours** malloc à droite d'un opérateur d'affectation.

Il ne faut pas oublier de transtyper le résultat de malloc() qui est de type void* en le type du pointeur auquel on affecte le résultat.

Allocation dynamique

```
nbElem = une valeur entière saisie au clavier  
pt_int = (int *)malloc(nbElem*sizeof(int));
```

Détail de cette ligne : analyse de l'expression à droite de l'opérateur d'affectation :

(int *) : transtypage : car malloc() donne un pointeur void *, et pt_int est de type int *

malloc : appel à la fonction

nbElem*sizeof(int) : n'oublions pas que malloc reçoit un nombre d'octets à allouer ! Ici, on veut allouer nbElem éléments, qui sont chacun de type int ! Or un int occupe plus d'un octet. Il occupe sizeof(int) octets !

Donc le nombre total d'octets à demander est : **nombre d'éléments * taille de chaque élément en octets.**

Allocation dynamique

Exemples d'utilisation :

Tableau de la taille requise, pas de perte de mémoire !

```
#include <stdio.h>
void main()
{
    int *pt_int;
    int nbElem;

    printf("combien d'elements dans le tableau ?:");
    scanf("%d",&nbElem);
    pt_int = (int *)malloc(nbElem*sizeof(int));
    if (pt_int == NULL)
        printf("L'allocation n'a pas fonctionné\n");
    else{
        /* saisie puis par exemple affichage du contenu du tableau */
        for (i=0;i<nbElem; i++)
            printf("%d ",pt_int[i]);
        free(pt_int);
    }
}
```

Libération de l'espace alloué

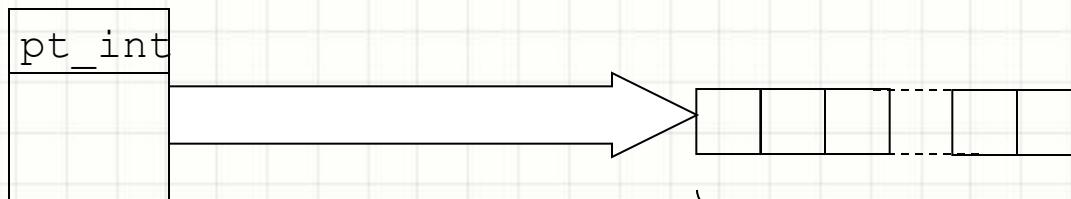
Lorsque la mémoire allouée dynamiquement n'est plus utile (le plus souvent, à la fin d'un programme), il est nécessaire de la libérer : la rendre disponible pour le système d'exploitation.

Fonction `free` qui réalise le contraire de `malloc()`.

`free(pointeur_vers_la_zone_allouée);`

On ne peut libérer que des zones allouées dynamiquement : pas de `free` avec un tableau statique, même si le compilateur l'accepte.

`free(pt_int);`



À chaque `malloc()` doit correspondre un `free()` dans un programme !

Zone allouée par `malloc`

Tableau de pointeurs

Puisqu'un pointeur est une variable comme une autre, on peut aussi les ranger dans un tableau: restriction, il faut que les pointeurs soient tous des pointeurs vers le même type.

Pour la déclaration :

type ***tab[NB_ELEM]** signifie : le contenu de chaque case du tableau est une valeur ayant le type donné, ou encore ; chaque case du tableau est un pointeur.

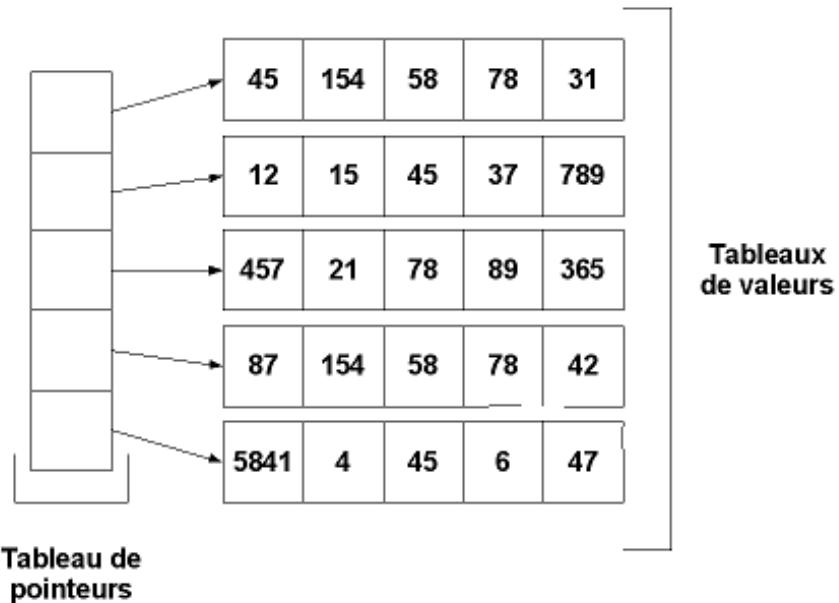
On peut s'en servir, par exemple, pour gérer un petit dictionnaire, c'est l'exemple que nous allons développer dans la suite du cours.

char *tab[100];

Tableau de pointeurs

Exemple :

`int *tab1[100];` tab1 = tableau de 100 pointeurs, chacun pointant sur un entier.



`char *tab2[50];` tab2= tableau de 50 pointeurs, chacun pointant sur un caractère,

Exemple : saisie d'un tableau à deux dimensions

```
#include <stdio.h>
const int nbElem=5;
void main()
{
    int *pt_int[nbElem];
    int i, j, nbligne;
    printf(" Nombre de lignes ?:");
    scanf("%d",&nbligne);
    for (i=0; i<nbElem; i++) {
        pt_int[i]=(int *)malloc(nbligne*sizeof(int));
        if (pt_int [i]==NULL)
            exit;
    } for (i=0;i<nbligne;i++)
        for (j=0;j<nbElem;j++)
            scanf("%d",&pt_int[i][j]);
    for (i=0;i<nbligne;i++) for (j=0;j<nbElem;j++)
        printf("%d ",pt_int[i][j]);
}
```

Tableau de pointeurs

Libération de la mémoire

La libération de mémoire allouée doit se faire en parcourant le tableau, et en libérant pointeur par pointeur dans ce tableau :

.....

```
{ int *pt_int[nbElem];
```

```
int i, j, nbligne;
```

.....

```
for (i=0; i<nbElem; i++) {
```

.....

```
    free (pt_int[i])
```

```
}
```

```
}
```

L'erreur à ne pas faire ! **free(pt_int)**

Pt_int est un type automatique, donc sera libéré automatiquement à la fin de la fonction dans laquelle il est déclaré.

Exercice

Ecrire un programme C qui permet de convertir un entier N quelconque en son équivalent binaire ?

44	2
0	22

22	2
0	11

11	2
1	5

5	2
1	2

2	2
0	1

1	2
1	0

Exemple :

$$44 \text{ (base 10)} = 101100 \text{ (base 2)}$$

```
#include <stdio.h>
void main()
{
    int N, N1, *restes , i, taille=0;
    printf("donner un entier");
    scanf("%d",&N);
    N1=N;
    while (N1!=0)
    {N1 = N1/2; taille++;}
    restes = (int *)malloc(taille*sizeof(int));
    N1=N;  if (restes == NULL)
        printf("erreur de taille de tableau\n");
    else {    for (i=0;i<taille;i++) {
        restes[i] = N1 % 2;
        N1 = N1/2; }
    for (i=taille-1;i>=0;i--) printf("%d : ",restes[i]);
    free (restes);
}
}
```

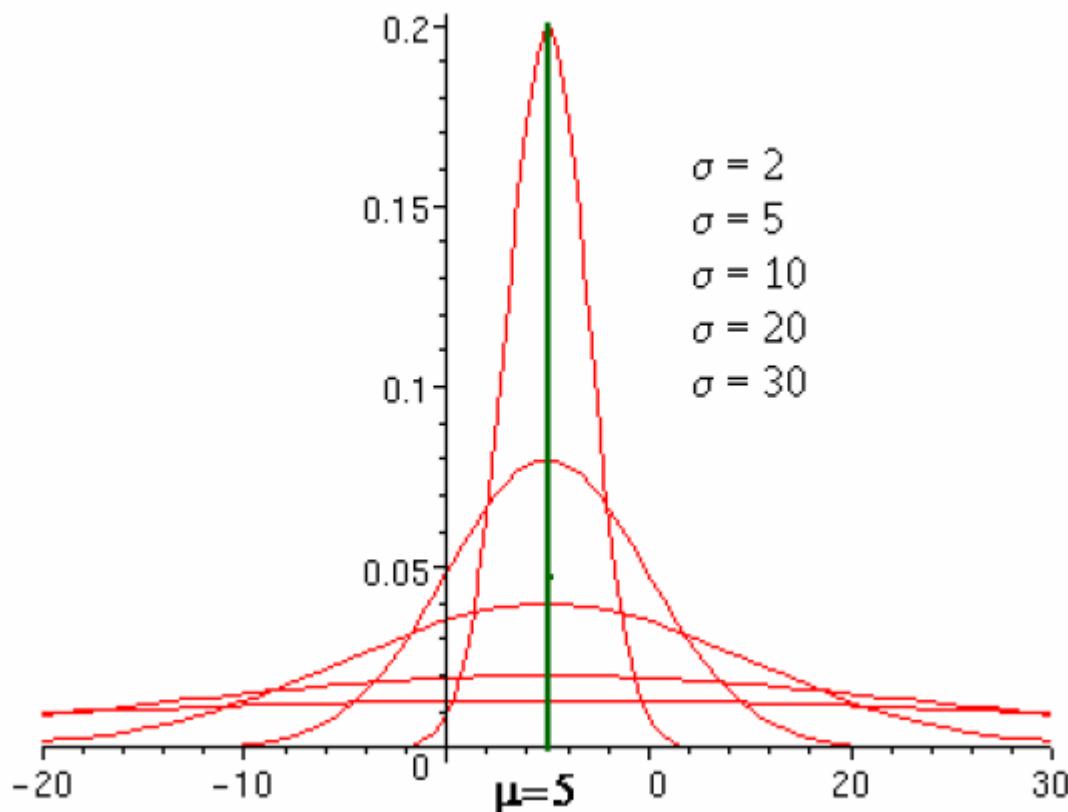
Exercice :

- 1- écrire une fonction sans type nommée saisie qui permet de saisir un tableau de N réels, chaque valeur doit être comprise dans l'intervalle 0..20 ?
- 2- écrire une fonction calcul sans type permettant de calculer pour un tableau de réel de dimension N l'écart type et la moyenne ?

$$\text{écart type} = \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - moy)^2}$$

- 3- écrire la fonction main qui permet de 1) déclarer un pointeur sur réel 2) allouer dynamiquement un tableau Notes de réels de dimension N à saisir 3) appeler la fonction saisie pour remplir le tableau Notes 4) appeler la fonction calcul pour le calcul de l'écart type et la moyenne 5) afficher les résultats ?

$$x \mapsto f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \quad \text{avec } \mu \in \mathbb{R} \text{ et } \sigma \in \mathbb{R}^+$$



Exercice :

1- écrire une fonction sans type saisie qui permet de saisir un tableau de N réels, chaque valeur doit être comprise dans l'intervalle 0..20 ? **void saisie (int N, float *T);**

2- écrire une fonction calcul sans type permettant de calculer pour un tableau de réel de dimension N l'écart type et la moyenne ? **void calcul (int N, float *T, float *m, float *E);**

$$\text{écart type} = \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - moy)^2}$$

3- écrire la fonction main qui permet de 1) déclarer un pointeur sur réel 2) allouer dynamiquement un tableau Notes de réels de dimension N à saisir 3) appeler la fonction saisie pour remplir le tableau Notes 4) appeler la fonction calcul pour le calcul de l'écart type et la moyenne 5) afficher les résultats ?

Exercice :

1- écrire une fonction sans type saisie qui permet de saisir un tableau de N réels, chaque valeur doit être comprise dans l'intervalle 0..20 ?

void saisie (int N, float *T)

{

int i;

for (i=0; i<N; i++)

{

do {

printf(“donner une valeur réelle : ”);

scanf(“%f”,&T[i]); // scanf(“%f”,T+i); }

while ((T[i]<0) || (T[i]>20));

}

}

Exercice :

2- écrire une fonction calcul sans type permettant de calculer pour un tableau de réel de dimension N l'écart type et la

$$\text{écart type} = \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - moy)^2}$$

void calcul (int N, float *T, float *m, float *E)

{

int i; float s=0;

for (i=0; i<N; i++) s = s+ T[i];

***m = s/N;**

s = 0;

for (i=0; i<N; i++)

s = s+ (T[i] - *m)* (T[i] - *m); //pow(T[i] - *m,2)

***E = sqrt(s/N);**

}

Exercice :

void saisie (int N, float *T);

void calcul (int N, float *T, float *m, float *E);

3- écrire la fonction main qui permet de 1) déclarer un pointeur sur réel 2) allouer dynamiquement un tableau Notes de réels de dimension N à saisir 3) appeler la fonction saisie pour remplir le tableau Notes 4) appeler la fonction calcul pour le calcul de l'écart type et la moyenne 5) afficher les résultats ?

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

main() {
    float *notes; int N; float M, E;
    printf("donner le nombre de notes ");
    scanf("%d", &N);
    notes = (float *)malloc(N*sizeof(float));
    if (notes == NULL)
        printf("erreur d'allocation de mémoire");
    else { saisie(N, notes);
            calcul (N, notes, &M, &E);
            printf("moyenne = %f ecart type = %f\n", M, E); } }
```

Exercice **void CODE (int N, int *T, int *C, int *D)**

Soit un tableau **T** contenant **N** entiers, on cherche à compresser son contenu en appliquant le principe suivant : Chaque séquence d'éléments identiques est représentée par un couple (**V**, **L**) où **V** est la valeur qui se répète le long de la séquence et **L** est sa longueur.

Par exemple si nous appliquons cette méthode sur le tableau **T** suivant :

T	12	12	8	8	8	8	4	9	9	9	9	9	4	4
---	----	----	---	---	---	---	---	---	---	---	---	---	---	---

Alors en sortie nous obtiendrons un autre tableau :

C	12	2	8	4	4	1	9	5	4	2				
---	----	---	---	---	---	---	---	---	---	---	--	--	--	--

Ainsi pour chaque indice **i** de **C**, nous obtiendrons :

- Dans **C[i]** la valeur d'une séquence d'entiers identiques dans **T**.
- Dans **C[i + 1]** la longueur de cette séquence.

1) Ecrire une fonction en langage C nommée **CODE** permettant de compresser un tableau **T** contenant **N** entiers en appliquant la méthode décrite en haut. Les paramètres de sortie de cette fonction sont le tableau compressé **C** et sa taille effective **D**.

2) Ecrire une fonction en langage C nommée **DECODE** permettant de décompresser un tableau **C** de taille **D**. Le paramètre de sortie de cette fonction est le tableau original **T**.

3) Ecrire la fonction **main()** incluant la saisie dynamique de **T** ?

Exercice :

1) Ecrire une fonction en langage C nommée **CODE** permettant de compresser un tableau **T** contenant **N** entiers en appliquant la méthode décrite en haut. Les paramètres de sortie de cette fonction sont le tableau compressé **C** et sa taille effective **D**.

```
void CODE (int N, int *T, int *C, int *D)
{ int i=0, j=0, L=1;
  while (i<N-1)
  {
    while ((T[i] == T[i+1]) && (i<N-1))
      { L++; i++;}
    C[j] = T[i];
    C[j+1] = L;
    i++;
    j = j +2;
    L=1;
  }
  *D = j;
}
```

Exercice :

2) Ecrire une fonction en langage C nommée **DECODE** permettant de décompresser un tableau **C** de taille **D**. Le paramètre de sortie de cette fonction est le tableau original **T**.

```
void DECODE (int T[], int C[], int D, int *N) // (int *T, int *C, int D)
{
    int i=0, j, k=0;

    for (i=0; i<D; i+=2)
    {
        for (j=0; j<C[i+1]; j++)
        {
            T[k]=C[i];
            k++;
        }
        *N = k;
    }
}
```

Exercice :

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

```
main() {
```

```
    int *T, *C; int N;
```

```
    printf("donner le nombre de valeurs ");
```

```
    scanf("%d", &N);
```

```
    T = (int *)malloc(N*sizeof(int));
```

```
    C = (int *)malloc(2*N*sizeof(int));
```

```
    if ((T == NULL) || (C==NULL))
```

```
        printf("erreur d'allocation de mémoire");
```

```
    else { for (i=0;i<N; i++)
```

```
    {
```

```
        printf("donner un entier");
```

```
        scanf("%d",&T[i]);}
```

```
    CODE(N, T, C, &D);
```

```
    for (i=0;i<2*N; i++) printf("%d ",C[i]);
```

```
    DECODE(T, C, D, &N);
```

```
    for (i=0;i<N; i++) printf("%d ",T[i]);
```

```
    free(T); free( C );
```

```
}
```

Les Structures

- Une structure est un assemblage de variables qui peuvent avoir différents types. Contrairement aux tableaux qui vous obligent à utiliser le même type dans tout le tableau, vous pouvez créer une structure comportant des variables de types long, char, int et double à la fois.
- Les structures sont généralement définies dans les fichiers .h, au même titre donc que les prototypes et les define. Voici un exemple de structure :

```
struct NomDeVotreStructure
{ int variable1;
  int variable2;
  int autreVariable;
  float nombreDecimal; };
```

- Une définition de structure commence par le mot-clé **struct**, suivi du nom de votre structure

Les Structures

- Exemples
- struct Coordonnées

```
{  
    int x; // Abscisses  
    int y; // Ordonnées  
};
```

*Structure utile pour créer un programme
De gestion de points dans le plan*

- struct Personne

```
{  
    char nom[100];  
    char prenom[100];  
    char adresse[1000];  
    int age;  
    int sexe; // Booléen : 1 = garçon, 0 = fille  
};
```

*Structure utile pour créer un programme
De gestion de carnet d'adresses*

Les Structures – définition de types

- L'utilisation de **typedef** permet de faciliter les définitions de prototypes pour un code plus lisible. Il est ainsi possible de créer un type équivalent à une structure
- ```
typedef struct {
 int x; // Abscisses
 int y; // Ordonnées
} Coordonnées ;
Coordonnées c1, c2, c3;
```
- ```
typedef struct {  
    char nom[100];  
    char prenom[100];  
    char adresse[1000];  
    int age;  
    int sexe;  
} Personne ;
```
- ```
typedef struct {
 int jour;
 int mois;
 int annee;
} date;
```

# Les Structures – définition de types

```
#include <stdio.h>
typedef struct {
 char nom[100];
 char prenom[100];
 char adresse[1000];
 int age;
 int sexe;
} Personne ;
```

```
int main(int argc, char *argv[])
{
 Personne utilisateur, user;
 printf("Quel est votre nom ? ");
 scanf("%s", utilisateur.nom);
 printf("Votre prenom ? ");
 scanf("%s", utilisateur.prenom);
 scanf("%d",&utilisateur.age);
 printf("Vous vous appelez %s %s",
 utilisateur.prenom, utilisateur.nom);
user = utilisateur;
 return 0;
}
```

Affectation possible entre objets de même type

# Les Structures – initialisation

- Pour les structures comme pour les variables, tableaux et pointeurs, il est vivement conseillé de les initialiser dès leur création pour éviter qu'elles ne contiennent « n'importe quoi »!
- Pour rappel :
- **une variable** : on met sa valeur à 0 (cas le plus simple) ;
- **un pointeur** : on met sa valeur à NULL (préféré à 0) ;
- **un tableau** : on met chacune de ses valeurs à 0.
- Pour les structures, l'initialisation va un peu ressembler à celle d'un tableau. En effet, on peut faire à la déclaration de la variable :
- Coordonnees point = {0, 0}; Cela définira, dans l'ordre, point.x = 0 et point.y = 0.
- Personne utilisateur = {"", "", "", 0, 0};

# Les Structures – Exemple

Ecrire un programme C qui définit une structure *point* qui contiendra les deux coordonnées d'un point du plan. Puis lit deux points et affiche la distance entre ces deux derniers.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
struct point{ typedef struct {
 float x; float y; float x; float y;
}; } point;
int main()
{ struct point A,B; point A, B;
 float dist;
 printf("Entrez les coordonnées du point A:\n");
 scanf("%f%f",&A.x,&A.y);
 printf("Entrez les coordonnées du point B:\n");
 scanf("%f%f",&B.x,&B.y);
 dist = sqrt((A.x-B.x)*(A.x-B.x) + (A.y-B.y)*(A.y-B.y));
 printf("La distance entre les points A et B est: %.2f\n",dist);
 return 0;
}
```

# Les Structures – Exercice 1

Ecrire un programme C qui définit une structure *etudiant* où un étudiant est représenté par son nom, son prénom et une note. Lit ensuite une liste de 100 étudiants entrée par l'utilisateur et affiche les noms de tous les étudiants ayant une note supérieure ou égale à **10** sur 20.

# Les Structures – Exercice 1

Ecrire un programme C qui définit une structure *etudiant* où un étudiant est représenté par son nom, son prénom et une note. Lit ensuite une liste de 100 d'étudiants entrée par l'utilisateur et affiche les noms de tous les étudiants ayant une note supérieure ou égale à 10 sur 20.

```
#include<stdio.h>
#include<stdlib.h>
struct etudiant{
 char nom[20]; char prenom[20]; int note;
};

int main()
{ struct etudiant t[100];
 int i;
 for(i=0;i<100;i++) {
 printf("donnez le nom, prenom et la note de l'etudiant %d:\n",i+1);
 scanf("%s%s%d",t[i].nom,t[i].prenom,&t[i].note);
 }
 for(i=0;i<100;i++) {
 if(t[i].note>=10)
 printf("%s %s\n",t[i].nom,t[i].prenom);
 }
 return 0;
}

typedef struct {
 char nom[20]; char prenom[20]; int note;
} etudiant;

etudiant t[100];
```

# Les Structures – Exercice 2

Ecrire un programme C qui lit un ensemble de N personnes avec leurs âges, dans un tableau de structures (utiliser malloc) , et supprime ensuite toutes celles qui sont âgées de vingt ans et plus.

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

```
typedef struct {
 char nom[20];
 int age;
} Personne;
```

# Les Structures – Exercice 2

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

typedef struct {
 char nom[20];
 int age;
} Personne;

int main()
{
 Personne *T;
 int N,i,j;

 printf("Nbre de personnes à lire:\n");
 scanf("%d",&N);
 T= (Personne *)malloc(N*sizeof(Personne));
 if (T == NULL)
 printf("erreur de mémoire !!");
 else
 {
```

```
 for(i=0;i<N;i++)
 {
 printf(" Nom de la personne %d:",i+1);
 scanf("%s",T[i].nom);
 printf("Entrez son age:\t");
 scanf("%d",&T[i].age);
 }

 for(i=0;i<N;i++)
 {
 if(T[i].age>=20)
 {
 for(j=i+1;j<N;j++)
 T[j-1] = T[j];
 N--;
 i--;
 }
 }
 }
}
```

# Pointeurs sur structures

- Intéressant pour deux raisons : 1) savoir comment envoyer un pointeur de structure à une fonction pour que celle-ci puisse modifier le contenu de la variable. 2) **Création de listes chaînées : les piles et les files.**

```
1) int main(int argc, char *argv[])
{ Coordonnees monPoint;
 initialiserCoordonnees(&monPoint);
 return 0;
}
void initialiserCoordonnees(Coordonnees *point)
{ (*point).x = 0; (*point).y = 0; }
Ou bien { point->x = 0; point->y = 0; }
```

# Les Structures – pointeurs

```
int main(int argc, char *argv[])
{
 Personne *user1, user2;
 user1=(Personne *)malloc(sizeof(personne));
 if (user1== NULL) exit(1);
 Else {
 printf("Quel est votre nom ? ");
 scanf("%s", (*user1).nom); // ou user1->nom
 printf("Votre prenom ? ");
 scanf("%s", user1->prenom);
 scnaf("%d",user1->age);
 printf("Vous vous appelez %s %s", user1->prenom, user1->nom);
user2 = *user1;
 printf("Vous vous appelez %s %s", user2.prenom, user2.nom); }
 return 0;
}
```

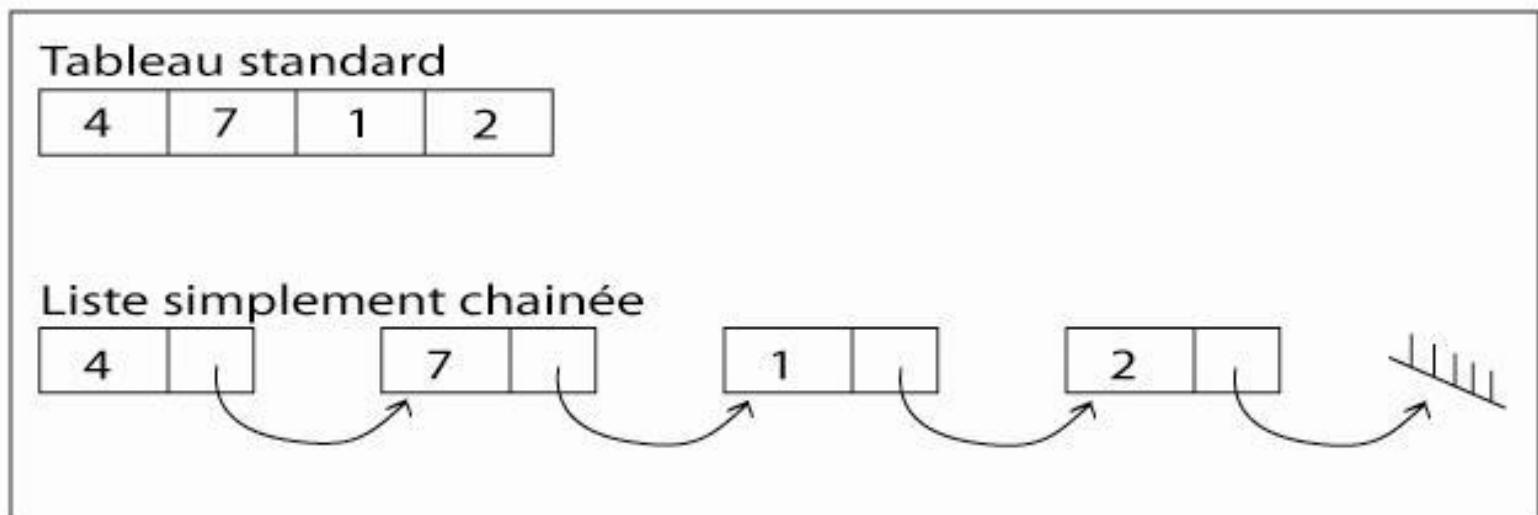
```
typedef struct {
 char nom[100];
 char prenom[100];
 char adresse[1000];
 int age;
 int sexe;
} Personne ;
```

# Les listes chaînées

- Lorsque on crée un tableau, les éléments de celui-ci sont placés de façon contiguë en mémoire. Pour pouvoir le créer, il vous faut connaître sa taille.
- Si on veut par exemple supprimer un élément au milieu du tableau, il faut recopier les éléments temporairement, réallouer de la mémoire pour le tableau, puis le remplir à partir de l'élément supprimé.
- En bref, ce sont beaucoup de manipulations coûteuses en ressources.
- Une liste chaînée est différente dans le sens où les éléments de votre liste sont répartis dans la mémoire et reliés entre eux par des pointeurs.
- On peut ajouter et enlever des éléments d'une liste chaînée à n'importe quel endroit, à n'importe quel instant, sans devoir recréer la liste entière.

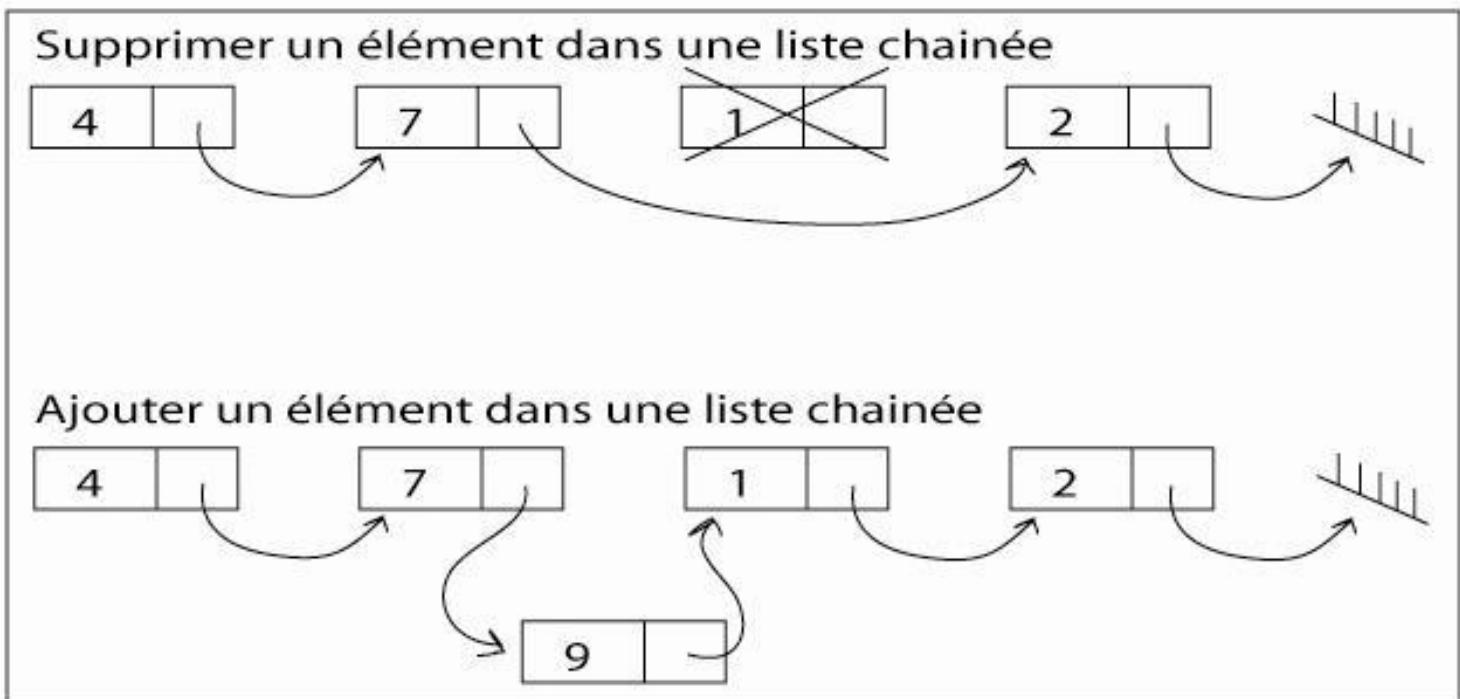
# Les listes chaînées

- Dans une liste chaînée, la taille est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet.  
**Il est en revanche impossible d'accéder directement à l'élément  $i$  de la liste chainée.** Pour déclarer une liste chaînée, il suffit de créer le pointeur qui va pointer sur le premier élément de votre liste chaînée, aucune taille n'est donc à spécifier.
- Il est possible d'ajouter, de supprimer, d'intervertir des éléments d'une liste chaînée sans avoir à recréer la liste en entier, mais en manipulant simplement leurs pointeurs.



# Les listes chaînées

- Chaque élément d'une liste chaînée est composé de deux parties :
  - la valeur que vous voulez stocker,
  - l'adresse de l'élément suivant, s'il existe.  
S'il n'y a plus d'élément suivant, alors l'adresse sera NULL, et désignera le bout de la chaîne.
- Ajout et la suppression d'un élément d'une liste chaînée.



# Les listes chaînées : déclaration

- Quel type sera l'élément de la liste chaînée ?
- On peut créer des listes chaînées de n'importe quel type d'éléments : entiers, caractères, structures, tableaux, voir même d'autres listes chaînées... Il est même possible de combiner plusieurs types dans une même liste.

```
#include <stdlib.h>

typedef struct element
{
 int val;
 struct element *next;
} element;
element* llist;
```

# Les listes chaînées : déclaration

```
#include <stdlib.h>

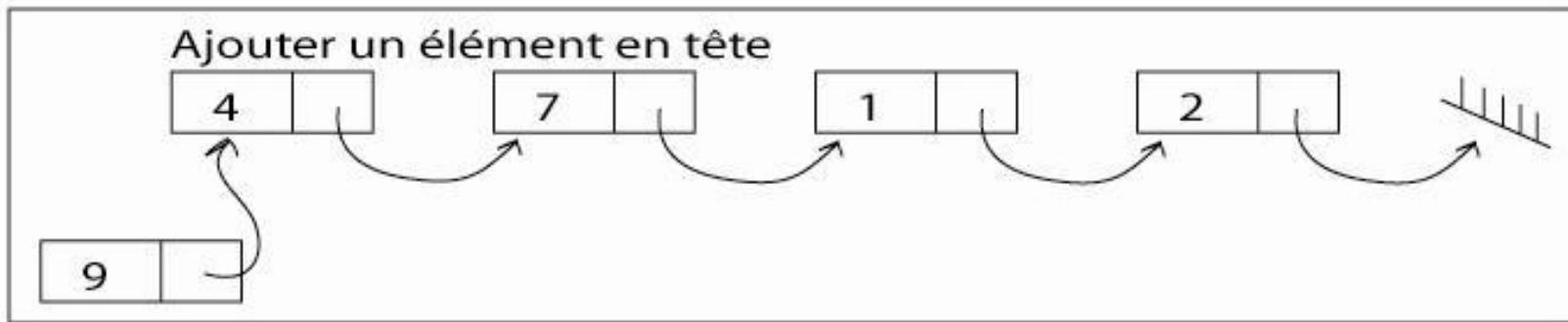
#include <stdlib.h>

typedef struct element
{ int val;
 struct element *next;
} element;
typedef element* llist;

int main(int argc, char **argv)
{ /* Déclarons 3 listes chaînées de façons différentes mais équivalentes */
 llist ma_liste1 = NULL;
 element *ma_liste2 = NULL;
 struct element *ma_liste3 = NULL;
 return 0; }
```

# Les listes chaînées : Ajout en tête

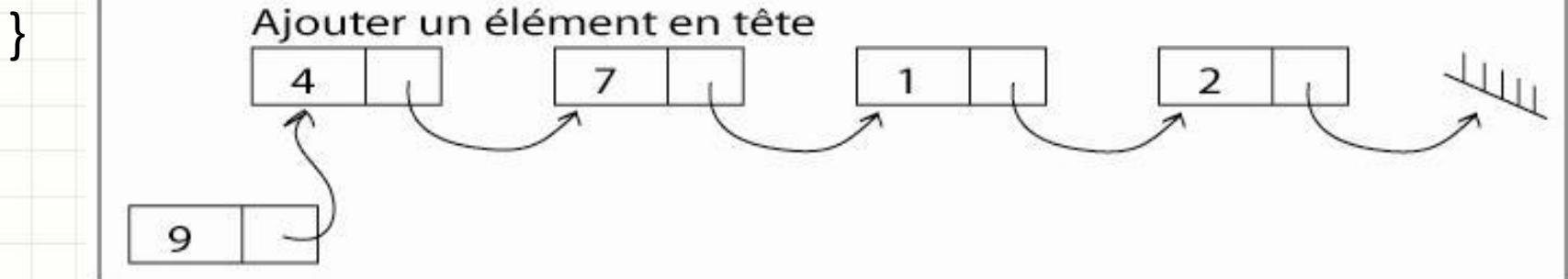
- ❖ Lors d'un ajout en tête, nous allons créer un élément, lui assigner la valeur que l'on veut ajouter, puis pour terminer, raccorder cet élément à la liste passée en paramètre.
- ❖ Lors d'un ajout en tête, on devra donc assigner à next l'adresse du premier élément de la liste passé en paramètre. Visualisons tout ceci sur un schéma :



- ❖ C'est l'ajout le plus simple des deux. Il suffit de créer un nouvel élément puis de le relier au début de la liste originale. Si l'original est , (vide) c'est NULL qui sera assigné au champ next du nouvel élément. La liste contiendra dans ce cas-là un seul élément.

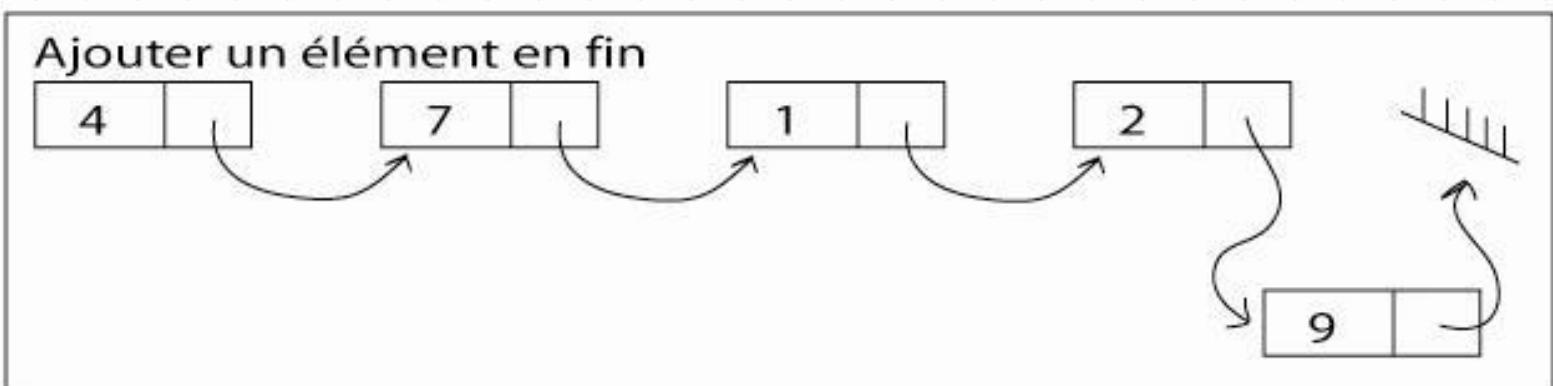
# Les listes chaînées : Ajout en tête

```
Ilist ajouterEnTete(Ilist tete_liste, int valeur)
{
 /* On crée un nouvel élément */
 element * nouvelElement;
 nouvelElement = (element *)malloc(sizeof(element));
 /* On assigne la valeur au nouvel élément */
 nouvelElement->val = valeur; //(*nouvelElement).val = valeur;
 /* On assigne l'adresse de l'élément suivant au nouvel élément */
 nouvelElement->next = tete_liste;
 /* On retourne la nouvelle liste, le pointeur sur le premier élément */
 return nouvelElement;
}
```



# Les listes chaînées : Ajout en fin de liste

Cette fois-ci, c'est un peu plus compliqué. Il nous faut tout d'abord créer un nouvel élément, lui assigner sa valeur, et mettre l'adresse de l'élément suivant à NULL. En effet, comme cet élément va terminer la liste nous devons signaler qu'il n'y a plus d'élément suivant. Ensuite, il faut faire pointer le dernier élément de liste originale sur le nouvel élément que nous venons de créer. Pour ce faire, il faut créer un pointeur temporaire sur **element** qui va se déplacer d'élément en élément, et regarder si cet élément est le dernier de la liste. Un élément sera forcément le dernier de la liste si NULL est assigné à son champ next.



# Les listes chaînées : Ajout en fin de liste

```
Ilist ajouterEnFin(Ilist liste, int valeur)
{ element* nouvelElement = malloc(sizeof(element));
 nouvelElement->val = valeur;
 nouvelElement->next = NULL;
 if(liste == NULL) return nouvelElement;
 else {
 /* Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on
 indique que le dernier élément de la liste est relié au nouvel élément */
 element* temp=liste;
 while(temp->next != NULL) temp = temp->next;
 temp->next = nouvelElement;
 return liste; }
 }
```

# Les listes chaînées : Affichage

```
void afficherListe(llist liste)
{
 element *tmp = liste;
 /* Tant que l'on n'est pas au bout de la liste */
 while(tmp != NULL)
 {
 /* On affiche */
 printf("%d ", tmp->val);
 /* On avance d'une case */
 tmp = tmp->next;
 }
}
```

La seule chose à faire est de se déplacer le long de la liste chaînée grâce au pointeur tmp. Si ce pointeur tmp pointe sur NULL, c'est que l'on a atteint le bout de la chaîne, sinon c'est que nous sommes sur un élément dont il faut afficher la valeur.

# Les listes chaînées : Application

- Utiliser les trois fonctions que nous avons vues jusqu'à présent :
  - ajouterEnTete (tete, val)
  - ajouterEnFin (tete, val)
  - afficherListe (tete)
- Vous devez écrire la fonction main permettant de remplir et afficher la liste chaînée ci-dessous. Vous ne devrez utiliser qu'une seule boucle for.
- 10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10

# Les listes chaînées : Application

```
int main()
{
 llist ma_liste = NULL;
 int i;
 for(i=1;i<=10;i++)
 {
 ma_liste = ajouterEnTete(ma_liste, i);
 ma_liste = ajouterEnFin(ma_liste, i);
 }
 afficherListe(ma_liste);
 // il reste à libérer l'espace mémoire occupé par la liste
 return 0;
}
```

```
typedef struct element
{ int val;
struct element *next;
} element;
typedef element* llist;
```

# Les listes chaînées : Application

- On veut écrire une fonction qui renvoie 1 si la liste est vide, et 0 si elle contient au moins un élément.

```
int estVide(l liste liste)
{ if(liste == NULL)
 return 1;
else return 0;
}
```

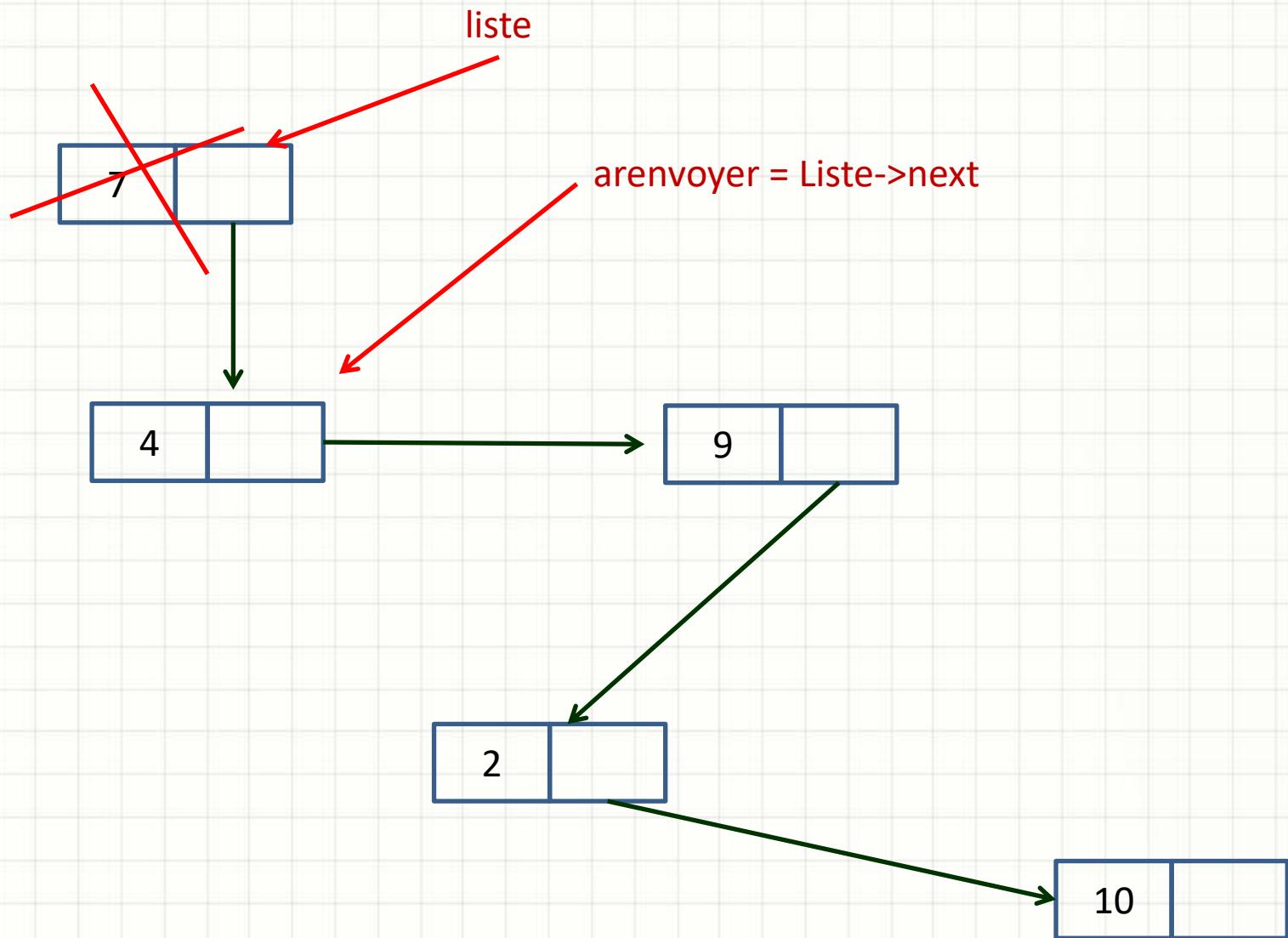
- Utilisation dans main():

```
.....
if(estVide(ma_liste))
 printf("La liste est vide");
else
 afficherListe(ma_liste);
```

Ecriture condensée :

```
int estVide(l list liste)
{
 return (liste == NULL)? 1 : 0;
```

# Effacer le premier élément

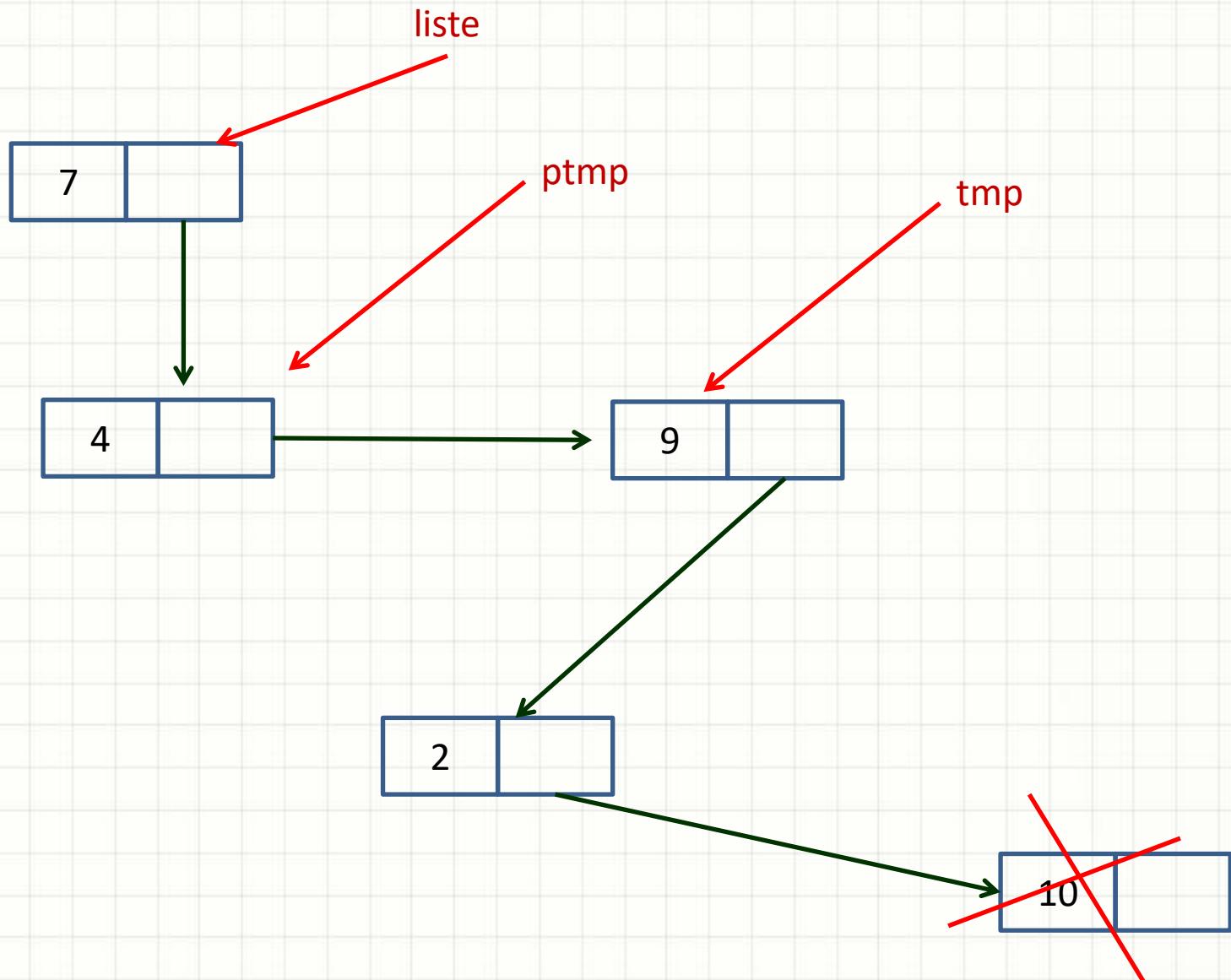


# Les listes chaînées : Suppression en tête

```
Ilist supprimerElementEnTete(Ilist liste)
{
 if(liste != NULL)
 { /* Si la liste est non vide, on se prépare à renvoyer l'adresse
 de l'élément en 2ème position */
 Ilist aRenvoyer = liste->next;
 /* On libère le premier élément */
 free(liste);
 /* On retourne le nouveau début de la liste */
 return aRenvoyer; }
 else return NULL;
}
```

Attention quand même à ne pas libérer le premier élément avant d'avoir stocké l'adresse du second, sans quoi il sera impossible de la récupérer.

# Effacer le dernier élément



# Les listes chaînées : Suppression en fin de liste

```
Ilist supprimerElementEnFin(Ilist liste)
{ if(liste == NULL) return NULL;
 if(liste->next == NULL) {
 free(liste); return NULL; }

 element* tmp = liste;
 element* ptmp = liste;
 while(tmp->next != NULL) {
 /* ptmp stock l'adresse de tmp */
 ptmp = tmp; /* On déplace tmp (mais ptmp garde l'ancienne valeur de tmp */
 tmp = tmp->next; }

 ptmp->next = NULL;
 free(tmp);
 return liste; }
```

# Suppression en fin de liste (avec commentaires)

```
Ilist supprimerElementEnFin(Ilist liste)
{ /* Si la liste est vide, on retourne NULL */
 if(liste == NULL) return NULL;
 /* Si la liste contient un seul élément */
 if(liste->next == NULL) {
 /* On le libère et on retourne NULL (la liste est maintenant vide) */
 free(liste); return NULL; }
 /* Si la liste contient au moins deux éléments */
 element* tmp = liste;
 element* ptmp = liste;
 while(tmp->next != NULL) {
 /* ptmp stock l'adresse de tmp */
 ptmp = tmp; /* On déplace tmp (mais ptmp garde l'ancienne valeur de tmp */
 tmp = tmp->next; }
 /* A la sortie de la boucle, tmp pointe sur le dernier élément, et ptmp sur l'avant-dernier.
 On indique que l'avant-dernier devient la fin de la liste et on supprime le dernier élément */
 ptmp->next = NULL;
 free(tmp);
 return liste; }
```

# Recherche d'un élément dans une liste

But : renvoyer l'adresse du premier élément trouvé ayant une certaine valeur. Si aucun élément n'est trouvé, on renverra NULL

```
llist rechercherElement(llist liste, int valeur)
{ element *tmp=liste;
/* Tant que l'on n'est pas au bout de la liste */
while(tmp != NULL)
{ if(tmp->val == valeur)
/* Si l'élément a la valeur recherchée, on renvoie son adresse */
 return tmp;
 tmp = tmp->next;
} return NULL;
}
```

# Compter le nombre d'occurrences d'une valeur

On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée.

```
int nombreOccurrences(llist liste, int valeur)
```

```
{ int i = 0;
 if(liste == NULL) return 0;
 /* Sinon, tant qu'il y a encore un élément ayant la val = valeur */
 while((liste = rechercherElement(liste, valeur)) != NULL)
 { liste = liste->next;
 i++;
 }
 /* Et on retourne le nombre d'occurrences */
 return i;
}
```

# Compter le nombre d'occurrences d'une valeur

On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée.

```
int nombreOccurrences(Ilist liste, int valeur)
```

```
{ int i = 0;
 if(liste == NULL) return 0;
 Ilist tmp=liste;
 while(tmp!=NULL)
 { if (tmp->val==valeur) i++;
 tmp = tmp->next;
 }
 /* Et on retourne le nombre d'occurrences */
 return i;
}
```

# Recherche du i-ème élément

```
llist element_i(llist liste, int indice)
{
 int i; element *tmp=liste
 /* On se déplace de i cases, tant que c'est possible */
 for(i=0; i<indice && liste != NULL; i++)
 tmp = tmp->next;
 /* Si l'élément est NULL, c'est que la liste contient moins de i éléments */
 if(tmp == NULL)
 return NULL;
 else
 {
 /* Sinon on renvoie l'adresse de l'élément i */
 return tmp; }
}
```

## Exercice :

Fonction pour effacer tous les éléments d'une liste ?

1- version séquentielle

2- version récursive

Ilist effacer (Ilist liste)

Ilist p; // p est un pointeur

p->val <- -> (\*p).val

p->next <- -> (\*p).next

```
typedef struct element
{ int val;
 struct element *next;
} element;
typedef element* Ilist;
```

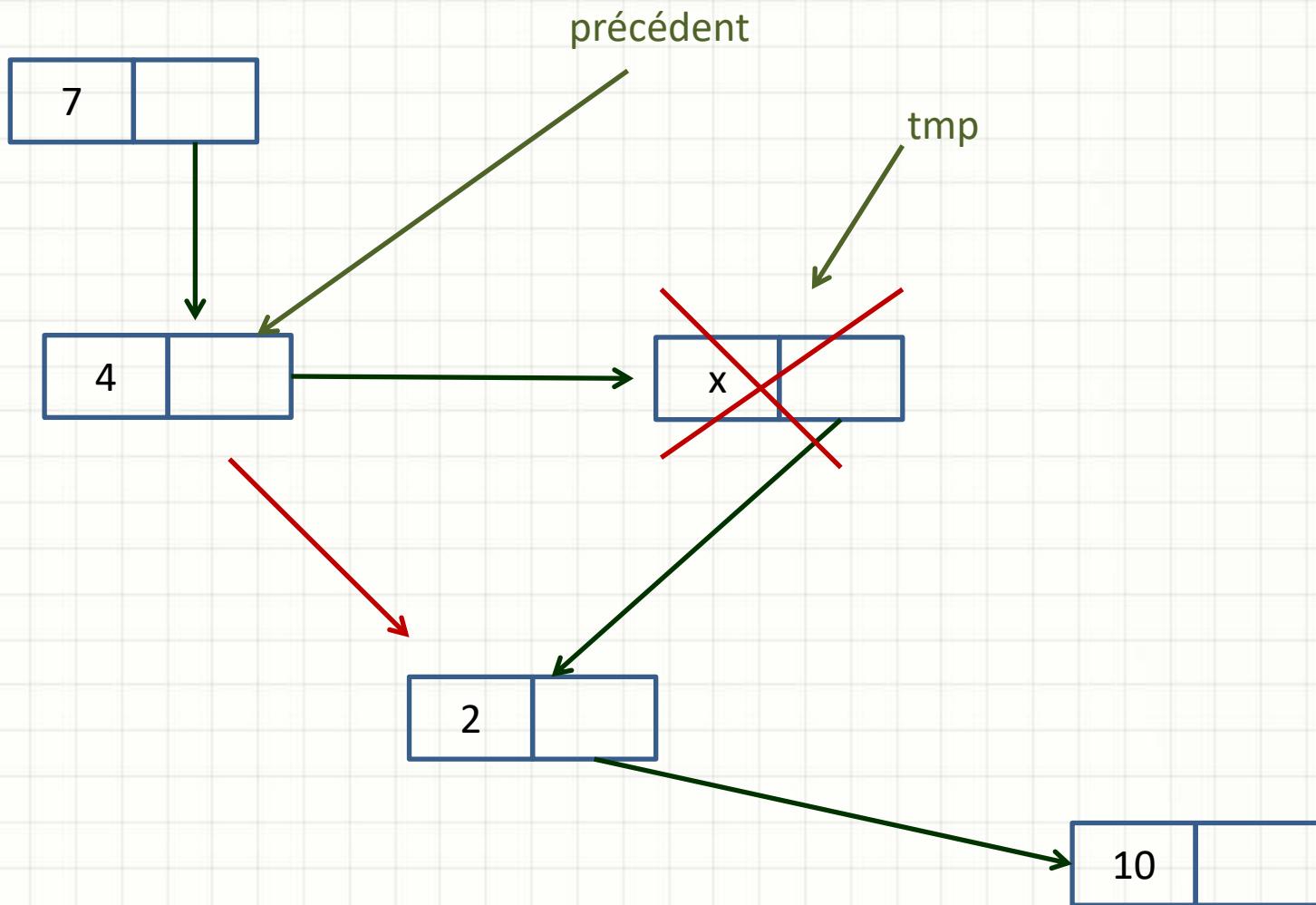
# Effacer tous les éléments d'une liste (ver 1)

```
llist effacerListe(llist liste)
{
 element* tmp = liste;
 element* tmpnext;
 while(tmp != NULL)
 { /* On stocke l'élément suivant pour pouvoir ensuite avancer */
 tmpnext = tmp->next;
 /* On efface l'élément courant */
 free(tmp);
 /* On avance d'une case */
 tmp = tmpnext;
 }
 /* La liste est vide : on retourne NULL */
 return NULL;
}
```

# Effacer tous les éléments d'une liste (ver 2)

```
Ilist effacerListe(Ilist liste)
{ if(liste == NULL)
 { /* Si la liste est vide, il n'y a rien à effacer, on retourne
 une liste vide i.e. NULL */
 return NULL; }
 else
 { /* Sinon, on efface le premier élément */
 element *tmp;
 tmp = liste->next;
 free(liste);
 return effacerListe(tmp);
 }
}
```

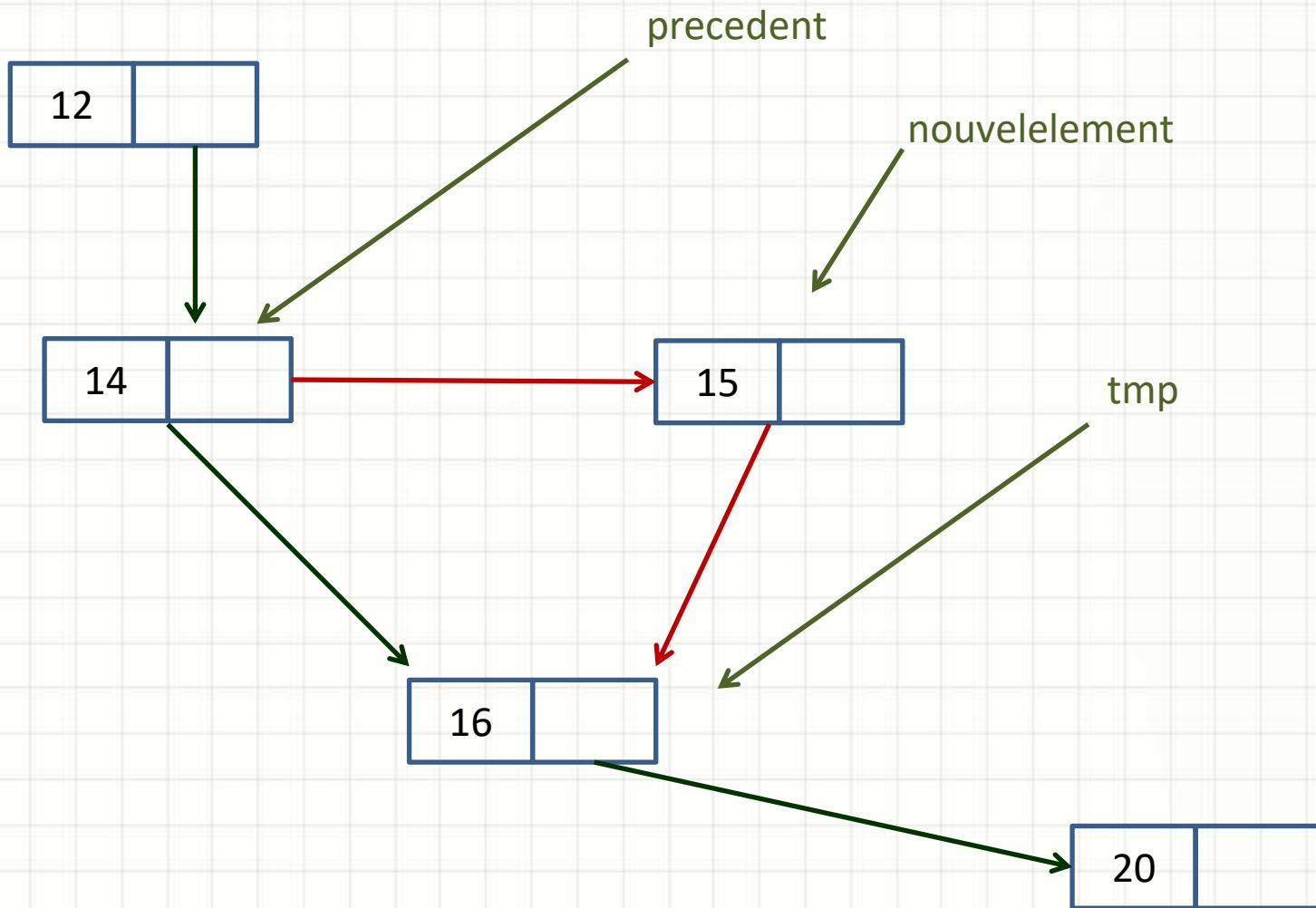
# Effacer le premier élément de valeur = x



# Effacer le premier élément de valeur = x

```
llist effacerElementListe(llist liste, int x)
{ if(liste == NULL) return NULL;
 else
 { /* Sinon, on parcourt et on s'arrête au premier élément trouvé */
 element *tmp=liste, *precedent=liste;
 while ((tmp->next!=NULL) && (tmp->val!=x))
 { precedent = tmp; tmp = tmp->next;}
 if (tmp->val!=x) return NULL;
 else { if (tmp->next==NULL)
 tmp = supprimerEnfinListe(liste,x);
 else if (tmp == liste) tmp = supprimerEnteteListe(liste,x);
 else {precedent->next = tmp->next;
 free(tmp);return (liste);}
 }
 return tmp;
 }
```

# Ajouter élément de valeur = x dans une liste ordonnée



# Ajouter élément de valeur = x dans une liste ordonnée

```
llist AjouterElementListe(llist liste, int x)
{ if(liste == NULL) liste=ajouterEntete(liste, x);
else
{ element* nouvelElement = malloc(sizeof(element));
nouvelElement->val = x;
nouvelElement->next = NULL;
/* Sinon, on parcourt et on s'arrête à la bonne position*/
element *tmp=liste, *precedent=liste;
while ((x>tmp->val) && (tmp->next!=NULL))
 {precedent = tmp; tmp = tmp->next;}
if (tmp->next == NULL) ajouterEnFin(liste,x);
else {precedent->next = nouvelElement;
nouvelElement ->next = tmp; }
}
```

# Lire et écrire dans des fichiers

- Le défaut avec les variables, c'est qu'elles n'existent que dans la mémoire vive. Une fois votre programme arrêté, toutes vos variables sont supprimées de la mémoire et il n'est pas possible de retrouver ensuite leur valeur.
- Comment peut-on, dans ce cas-là, enregistrer les meilleurs scores obtenus à son jeu ? Comment peut-on faire un éditeur de texte si tout le texte écrit disparaît lorsqu'on arrête le programme ?
- Heureusement, on peut lire et écrire dans des fichiers en langage C. Ces fichiers seront écrits sur le disque dur de votre ordinateur.
- L'avantage est donc qu'ils restent là, même si vous arrêtez le programme ou l'ordinateur.
- Pour lire et écrire dans des fichiers, nous allons avoir besoin de réutiliser tout ce que nous avons appris jusqu'ici : pointeurs, structures, chaînes de caractères, etc.

# Ouvrir et fermer un fichier

- Pour lire et écrire dans des fichiers, nous allons nous servir de fonctions situées dans la bibliothèque stdio.h. Cette bibliothèque-là contient aussi les fonctions printf et scanf que nous connaissons bien !
1. On appelle la fonction d'**ouverture de fichier** fopen qui nous renvoie un pointeur sur le fichier.
  2. **On vérifie si l'ouverture a réussi** en testant la valeur du pointeur qu'on a reçu. Si le pointeur vaut NULL, c'est que l'ouverture du fichier n'a pas fonctionné, dans ce cas on ne peut pas continuer (il faut afficher un message d'erreur).
  3. Si l'ouverture a fonctionné (p!=NULL), alors on peut **lire et écrire dans le fichier** à travers différentes fonctions.
  4. Une fois qu'on a **terminé de travailler sur le fichier**, il faut penser à le « fermer » avec la fonction fclose.

# Modes d'ouverture d'un fichier

- "**r**" : **lecture seule**. Vous pourrez lire le contenu du fichier, mais pas y écrire. *Le fichier doit avoir été créé au préalable.*
- "**w**" : **écriture seule**. Vous pourrez écrire dans le fichier, mais pas lire son contenu. *Si le fichier n'existe pas, il sera créé.*
- "**a**" : **mode d'ajout**. Vous écrirez dans le fichier, en partant de la fin du fichier. Vous ajouterez donc du texte à la fin du fichier. *Si le fichier n'existe pas, il sera créé.*
- "**r+**" : **lecture et écriture**. Vous pourrez lire et écrire dans le fichier. *Le fichier doit avoir été créé au préalable.*
- "**w+**" : **lecture et écriture, avec suppression du contenu au préalable**. Le fichier est donc d'abord vidé de son contenu, vous pouvez y écrire, et le lire ensuite. *Si le fichier n'existe pas, il sera créé.*
- "**a+**" : **ajout en lecture / écriture à la fin**. Vous écrivez et lisez du texte à partir de la fin du fichier. *Si le fichier n'existe pas, il sera créé.*

# Modes d'ouverture d'un fichier : mode binaire

- Pour chaque mode qu'on a vu là, si vous ajoutez un "b" après le premier caractère
  - ("rb", "wb", "ab", "rb+", "wb+", "ab+")
- alors le fichier est ouvert en mode binaire.
- C'est un mode un peu particulier que nous ne verrons pas ici. En fait, le mode texte est fait pour stocker... du texte comme le nom l'indique (uniquement des caractères affichables),
- tandis que le mode binaire permet de stocker des informations octet par octet (des nombres, principalement). C'est sensiblement différent.

■

# Premier exemple de code

```
int main(int argc, char *argv[])
{
 FILE* fichier = NULL;
 fichier = fopen("test.txt", "r+");
 if (fichier != NULL)
 {
 // On peut lire et écrire dans le fichier
 }
 else
 {
 // On affiche un message d'erreur si on veut
 printf("Impossible d'ouvrir le fichier test.txt");
 }
 return 0;
}
```

# Exemple 2 de code

```
int main(int argc, char *argv[])
{
 FILE* fichier = NULL;
 fichier = fopen("test.txt", "r+");
 if (fichier != NULL)
 {
 // On lit et on écrit dans le fichier
 // ...
 fclose(fichier); // On ferme le fichier qui a été ouvert
 }
 return 0;
}
```

# Ecriture dans un fichier texte

```
int main(int argc, char *argv[])
{
 FILE* fichier = NULL;
 int age = 0;
 fichier = fopen("test.txt", "w");
 if (fichier != NULL)
 {
 // On demande l'âge
 printf("Quel age avez-vous ? ");
 scanf("%d", &age);
 // On l'écrit dans le fichier
 fprintf(fichier, "Le Monsieur qui utilise le programme, il a %d ans", age);
 fclose(fichier); }
 return 0;
}
```

# Lecture à partir d'un fichier texte

```
int main(int argc, char *argv[])
{
 FILE* fichier = NULL;
 int x,y,z;

 fichier = fopen("test.txt", "r");
 if (fichier != NULL)
 {
 fscanf(fichier, "%d %d %d", &x, &y, &z);
 printf("Les valeurs de x,y et z sont : %d, %d et %d", x,y,z);
 fclose(fichier);
 }
 return 0;
}
```

# Se déplacer dans un fichier

- Chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier. Vous pouvez imaginer que c'est exactement comme le curseur de votre éditeur de texte (tel Bloc-Notes). Il indique où vous êtes dans le fichier, et donc où vous allez écrire.
- En résumé, le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier.
- Il existe trois fonctions à connaître :
  - **ftell** : indique à quelle position vous êtes actuellement dans le fichier ;
  - **fseek** : positionne le curseur à un endroit précis ;
  - **rewind** : remet le curseur au début du fichier (*c'est équivalent à demander à la fonction fseekde positionner le curseur au début*).

## Permutation de trois variables

```
#include <stdio.h>
main() {
 float A, B, C, max;
 printf("donner un réel A"); scanf("%f",&A);
 printf("donner un réel B"); scanf("%f",&B);
 printf("donner un réel C"); scanf("%f",&C);
 max = A;
 if (max < B)
 {max = B; B=A; A=max;}
 if (max < C)
 {max = C; C=A; A=max;}
 if (B < C)
 { max = C; C=B; B=max;}
 printf("variables permutées %f %f et %f = %f", A, B, C);
}
```