

RSVJ/Ministère de la Défense Nationale
22nd Logistic Wing Force Aérienne
Systems Enabling UNIX

Programmation avec C

2nd EDITION

Vincent Jamart

1st Edition: Année Académique 1997-1998

2nd Edition: Année Académique 1999-2000

Table Des Matières

Conception d'un programme	3
Types d'erreurs	3
Eviter les erreurs	3
Vérification	4
Avantages d'un programme pour la maintenance	4
Langages :interpréteur ou compilateur ?	4
Schéma de compilation en C	annexe A
 Programmation en C	 5
Les données	7
Type de données	7
Simple statements (Instructions simples)	8
Structured statements (Instructions structurées)	8
Déclaration/Définition de variables/constantes	8
Incrémentation "++"	8
IF-THEN-ELSE Statement	9
IF imbriqués	9
switch-case statement	9
Boucles	10
Fonctions d'entrée/sortie	11
 Fonctions dans un programme	 13
Passage d'arguments à une fonction	14
Recursive des fonctions	14
Déclarations internes/externes	16
 Les Tableaux(vecteurs)	 17
Les chaînes de caractères (strings)	19
Passages de tableaux comme arguments à une fonction	19
Tableaux multidimensionnels (matrices)	21
 Les Pointeurs	 22
Opérations sur les pointeurs	26
Utilisation de la fonction SIZEOF	27
Passage des arguments de la ligne de commande	27
 Les Structures	 28
Les Unions	33
 Les Fichiers: import/export, texte/binaire	 35
Fichiers texte	35
Fichiers binaire	38
 Une structure auto-référentielle: la Pile	 41
Allocation Dynamique de la mémoire et implémentation a la Pile	42
 Listes Chainées	 45
Listes Doublement Chainées	53
Listes Circulaires	54
Récurrence des Listes Chainées	55
 Arbres Binaires	 57
Récurrence des Arbres Binaires	58
 Annexes	 60

CONCEPTION D'UN PROGRAMME

- 1) **ANALYSE** :(fonctionnelle) :
 - Que veut le client
 - Définition correcte des données
 - Définition correcte des résultats
- 1') **ANALYSE** :(organique) :
 - Adaptée au matériel (langage, O.S.)
- 2) **CONCEPTION** :
 - Analyse TOP-DOWN (découpage du programme)
 - PSEUDO-CODE *indépendant du programme*
- 3) **CODAGE** :
 - Pseudo-code > Langage
- 4) **MISE AU POINT** :
 - Mise en commun des différents modules (>topdown)
- 5) **VERIFICATION** :
 - Création d'un fichier *test*
- 6) **DOCUMENTATION** (externe) :
 - Pour les utilisateurs : mode d'emploi et analyse
 - Types de données IN/OUT
 - Fonctions du programmes
- 6') **DOCUMENTATION** (interne) :
 - Algorithme
 - Remarques dans le fichier langage pour les programmeurs
- 7) **MAINTENANCE ET MISE A JOUR** :
 - Mise à jour des encodages de données
 - Correction des petites erreurs découvertes à l'emploi

Donnée : *Information avec laquelle un ordinateur peut travailler*

Objet : *Ce qui appartient à un programme (variables, constantes, compteurs, modules,.)*

TYPES D'ERREURS

- **LOGIQUES** (<>=)
- **SYNTAXIQUES** (printf, include)
- **LEXICALES** (idem)
- **SEMANTIQUE** (mauvais types de données, non définies, plusieurs fois, absurdes)
- **EXECUTION** (division par 0, dépassement de données, fichier absent)
- **!!!! PONCTUATION !!!!**

EVITER LES ERREURS

- **Comment l'utilisateur va-t-il introduire les données ?**
- **Quelles erreurs peut-il commettre ?**
- **Lisibilité et compréhension des commandes**
- **Plusieurs possibilités d'utilisation du programme : avancé/débutant**
- **Rattrapage en cas d'erreurs (Annuler/marche arrière)**

VERIFICATION

- Initialisation des var/const
- > au lieu de <
- = au lieu de <>
- Confusion : 1 l ou I, 0 et O, 2 et Z
- Test de sortie d'une boucle
- Noms de variables ressemblants
- Formules de calcul trop complexes

AVANTAGES D'UN PROGRAMME POUR LA MAINTENANCE

- Intégrité du programme : pas d'à-peu-près
- Clarté facile, clair d'utilisation
- Simplicité : éviter les formules trop compliquées
- Efficacité : compromis vitesse/espace mémoire
- Modularité : utiliser des procédures (modules)
- Extensibilité : pouvoir ajouter qqch et adapter le programme au fil du temps

LANGAGES

BASIC, QBASIC, Visual Basic, Perl	: INTERPRETEUR
PASCAL, C ; Fortran, COBOL,...	: COMPILATEUR

INTERPRETEUR

Interpréteur : interprète (exécute) les commandes lignes par ligne : peut dépasser la capacité matérielle ou se planter en cours de programme

Compilateur : voir schémas

PROGRAMMATION EN C

ANALYSE

- 1) <Déclaration des variables>
- 2) <Lecture des valeur, du taux, nombre d'années>
- 3) <Calcul du coefficient pour la formule $i=r/100$ >
- 4) <Détermination de la valeur future avec la formule : $f=p*(1+i)$ >
- 5) <Ecriture du résultat>

PSEUDO-CODE

PGM (Calcul_des_interets_composes)

BEGIN

 Declaration des variables

 IN(clavier ; p,r,n)

 <Calcul de i>

 <Calcul de la valeur future de F>

 OUT(ecran ; F)

END

FONCTION Calcul de i

BEGIN

 ($i=r / 100$)

END

FONCTION Calcul de la valeur future de F

BEGIN

 ($F= p*pow(1+i),n$)

END

*topdown du pseudo-code

LANGAGE C (ANSI)

#include <stdio.h>

instruction préprocesseur de la librairie in/out

#include <math.h>

instruction préprocesseur de la librairie mathématique

Si la librairie est entre <>, elle se trouve dans le répertoire des librairies.

Si elle se trouve entre « », elle se situe dans le répertoire courant.

/*calcul des interets composes*/

remarque entre /* */

main ()

corps du programme, précède de VOID si la fonction ne renvoie

aucune valeur (renvoi de valeur :c= main(a,b))

{ équivalent au BEGIN

float p,r,n,i,f ;

définition des variables (float= nombres flottants)

/*lecture des données*/

printf(« Montant du deopt(p) : ») ;

printf(« ») ; sortie vers l'écran de « »

scanf(« %f », &p) ;

scanf(« ») ; entrée des données au clavier

%f définit l'entrée comme un float (p129)

i= r/100	&p définit l'adresse (&)de la variable (p)
F=p*pow((1+i),n) ;	i=r/100 > = est l'assignation (== est la comparaison)
	fonction pow extraite de la librairie math.h
/*affichage des resultats*/	
printf(« \n La valeur (F) est : %2f\n »,F) ;	\n indique un retour à la ligne (p57)
	%2f définit la sortie de f comme un float
	avec une précision de 2 (%d=entier)
	(F est la variable)
}	équivalent à un END

LES DONNES

L'OBJET (data) a trois attributs :

chiffres !case sensitive MAJ-min !!)

Type (= integer,.. espace mémoire et opérations

valables sur ce type)

Valeur (=l'objet lui-même, soit var,const (3,10))

Valeur	Ce qui est FIXE	Ce qui CHANGE
VARIABLE :	nom et type	valeur
CONSTANTE	nom, type, valeur	

➤ *Tout objet doit être déclaré avant utilisation*

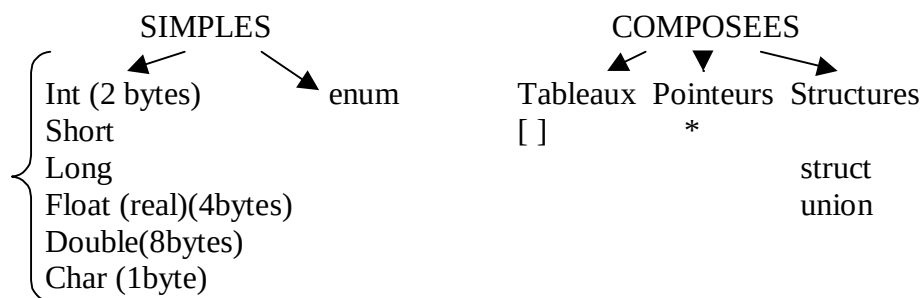
CASE MEMOIRE (2 bytes/16bits)

```
(int a;) > | i | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _
```

^ Bit pattern qui déclare le contenu de la variable a

(int= 2bytes > 16bits-1 bit pattern, donc le contenu est codé sur 15 bits)

TYPES DE DONNES



Structure

X	X	X
$\wedge_{\text{nom}}$	$\wedge_{\text{num tel.}}$	$\wedge_{\text{salaire}}$

Union

X		
---	--	--

 $\wedge^{\text{nom,num tel, salaire}}$

Ordre d'importance des types :

long double >double>unsigned long integer>long integer>unsigned>integer>character

SIMPLE STATEMENTS (INSTRUCTIONS DE CONTROLE SIMPLES)

Nulle :	;	
Expression	= + - * /	Suivi de ;
Break	break	Suivi de ;
Continue	continue	Suivi de ;
Goto (branchement)	goto	Suivi de ; (SPAGHETTI !)

STRUCTURED STATEMENTS (INSTRUCTIONS COMPOSEES STRUCTUREES)

Block Statement	{ xxxxxx ; xxxxxx ; }
LOOP Statement	DO WHILE FOR
Conditional Statement (Control)	IF SWITCH

DECLARATION/DEFINITION DE VARIABLES/CONSTANTES

- Déclarer une variable : `int= cpt ;`
- Définir une variable : `cpt= 0 ;`
- Affectation (assignation) : `var<5` `var=5 ;`
- **DECLARATION-DEFINITION** : `int cpt = 0 ;`
- Définition d'une **constante symbolique** en préprocesseur (donc sans ;) :
`#define const 10`

TYPE CASTING : force le type d'un résultat

`X= (float)i/j` > i est forcé float
`X= (float)(i/j)` > i et j sont forcé float

Exemple : `(float)(2/4)` > `(float)0` > 0.0
`(float)2/4` > `2.0/4` > 0.5

- « %6.3f » 6 emplacements float (---.---) dont 3 décimaux ! le point est compté !
- \n caractère de formatage (tableau p.57)
- **BOOLEAN** : 0=false ; <>0= TRUE (1 ou n'importe quel nombre sauf 0)

INCREMENTATION ++

a+1 peut s'effectuer de 3 façons :

- a+1 (normal)** dans une ligne d'une boucle, par exemple (exécuté après le ;)
- a++ (post incrémentation)** après l'exécution de la ligne (après le ;)

++a (pré incrémentation) avant l'exécution de la ligne (avant le ;)

- `printf(« %f » a++) ;`
- *sort '5.0' car c'est la valeur float (real) de a puis incrémente a de 1*
- `b=a ;`
- *b prend la valeur de a mais a a été incrémenté (a+1) juste après la sortie '5.0'*
- `printf(« %d » b) ;`
- *sort '6' car c'est la valeur entière de b*

IF-THEN-ELSE STATEMENT

```
if (condition)
{
instruction ;
}
else
{
instruction
}
```

équivalent à: condition?alors:sinon;

Test d'égalité	==
Test de différence <>	!=
Test a<b ou a<=b	> ou <=
Test a>b ou a>=b	>= ou >
ET logique (CAND, les deux vrais)	&&
OU logique (un des deux vrai)	
NOT logique (COR)	!

- *Tableau complet p.10*

IF IMBRIQUES(ELSEIF)

```
if (condition1)
{
action1 ;
}
else
    if(condition2) ;
    {
        action2 ;
    }
    else action3 ;
```

SWITCH-CASE STATEMENT

```
switch (variable)
{
case valeur1:
    action1 ;
    action2;
```

```

        break; /*arrêt execution du switch, sinon continue tous les case*/
case valeur2:
    action3;
    break;
default: /*non-obligatoire*/
    defaultaction;
}

```

BOUCLES

Boucle DO...LOOP (TANT QUE)

```

while (condition de sortie) /*donc au moins un test*/
{
    instructions ;
    incrémentation ;
}

```

Boucle LOOP...UNTIL (JUSQUE)

```

do /*s'exécutera au moins une fois avant le test*/
{
    instructions ;
    while (condition de sortie)
}

```

Boucle FOR... NEXT(pas)(DE...A)

```

for (initialisation ; condition de sortie ; incrémentation)
{
    instructions ;
}

```

rem : *équivalent à*

```

{
    initialisation ;
    while (condition de sortie) ;
    {
        instructions ;
        incrémentation ;
    }
}

```

exemple : **for**(i=1,j=1 ; i<5 ; i++,j--) **printf** (« %d %d », i,j) ;
 équivalent à : **for**(i=j=1 ; (i<5)&&(j<=100) ; i++,j--) **printf** (« %d %d », i,j) ;

break; /*force la sortie d'une boucle*/

continue; /*réexécute la boucle mais ignore la fin de celle-ci (après le continue)

FONCTIONS D'ENTREE/SORTIE

bibliothèque: #include <stdio.h> et éventuellement <ctype.h> pour putchar

	LECTURE	ECRITURE	rem
CARACTERE	getc()	putc()	' '
	fgetc()	fputc()	à partir d'un fichier
CHAINE	gets()	puts()	string « »
	fgets()	fput()	à partir d'un fichier
FORMATE	scanf()	printf()	
	fscanf()	fprintf()	à partir d'un fichier
ENREGISTREMENT	fread()	fwrite()	à partir d'un fichier

Enregistrement : int int char
 NUM AGE LOC

Stdin : clavier (get)]
 Stdout : écran (put)] STDIO
 Stderr : erreurs (ecran)]

>ENTREE

CARACTERES

getchar () (variable renvoyée qui contient le resultat)

- 1) **c=getc (stdin) ;** > c contient ce qui a été entré au clavier
- 2) **c=getch () ;** > sait automatiquement la source (stdin)
- 3) **c=getchar ()** > affiche ce qui a été tapé et effectue un \n

➤ *rem : on peut utiliser un getch pour faire une pause d'affichage : dans une boucle, après x lignes affichées (compteur), on introduit un getch qui attend un enter (\n) pour continuer.*

CHAINES

gets (=get string) saisit une chaîne et l'enregistre après le\n et fait \0 automatiquement pour fermer le tableau crée et contenant la chaîne (seule façon d'enter une chaîne en C)

```
{
char ch [50] définir un tableau de 50 cases appelé « ch »
gets(ch) ;
}
```

- 1) **cgets** > idem gets mais sans \n

<SORTIE

CARACTERES

`putch( )` > `putch('A');`

1) `putchar ( )`

2) `putc (stdout)` > unité de sortie à définir

CHAINES

`puts( )` > `puts(« chaîne »);` > envoie un `\n`
`cputs ( )` > `cputs(« chaîne »);` > n'envoie pas un `\n`

FORMATES

>SORTIE :PRINTF (pas de \n automatique)

`printf(« string »);` > `printf (« bonjour »);`

`printf(« %d », i);` > affiche ce que contient la variable `i` au format `d` (p.129)

>ENTREE : SCANF(p.129)

`scanf(« type », &variable);`

`scanf(« %d », &i);` > enregistre l'entrée dans la variable `i` au format `d` (p.129)

! `scanf` d'une phrase avec espaces vers un tableau:

`scanf ("%abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ]", ligne);`

ou terminer l'entrée uniquement avec ENTER, en ignorant les espaces:

`scanf ("%[\n]", ligne);`

caractères provoquant un EOL: `[^abcd...]`

!!! Après plusieurs `scanf`, le tampon(buffer) clavier s'emplit et il faut effectuer un `fflush(stdin)`; pour le vider, c'est pour cela que `scanf` est déconseillé pour plus d'une entrée !!!

FONCTIONS DANS UN PROGRAMME

type données_renvoyées nom_fonction (type paramètres_formels)

On place une déclaration prototype de la fonction avant la fonction main, suivie de ;
(par exemple : *int fonction_exemple (int a, char lettre) ;*)

On commence la fonction par cette même ligne, mais sans le ;

La fonction principale d'un programme en C est le main, déclarée comme ceci :

```
void main ()
{
...
}
```

Les accolades servent à définir le début et la fin de cette fonction, comme BEGIN et END en pseudo-code. La variable VOID indique que la fonction de renvoie rien. On peut utiliser VOID comme paramètres reçus si la fonction ne reçoit rien non plus (void fonction (void) ;). **Une fonction ne peut renvoyer qu'une valeur à l'aide d'une seule variable.** Le renvoi s'effectue avec la commande *return(variable)*. Il est utile d'utiliser des fonctions pour scinder un problème en plusieurs sous-problèmes plus faciles à résoudre.

Exemple d'un programme utilisant une fonction :

```
#include <stdio.h>
int fonction2 (int a) ; /*déclaration préprocesseur (prototypage) du prototype de la fonction
                        'fonction2' qui reçoit un entier 'a' et renvoie un entier */

void main ()           /*fonction main */
{
    int a=0, b;         /*déclaration dans la fonction main des variables a, de valeur
                        0 et b comme entiers. Il sont connus dans cette fonction
                        uniquement.(voir chapitre sur les déclarations internes/externes */
    b= fonction2 (a) ; /* appel de la fonction 'fonction2' qui reçoit la valeur de la
                        variable 'a' et renvoie la valeur qu'elle aura traitée dans
                        une variable nommée 'b' */
    printf (« Le résultat b est de : %d et a= %d », b ,a) ;
}

int fonction2 (int a)  /* fonction fonction2 */
{
    int a ;             /* déclaration de la variable a comme entier dans la fonction */
    a++ ;               /* incrémentation de a de 1 (a=a+1) après le ;
    a*=2 ;              /* multiplication de a par 2 (a=a*2)
    return (a) ;        /* renvoi du résultat des opérations sur a vers le lieu d'appel
                        ici, la fonction main() */
}
```

➤ On peut aussi créer sa propre bibliothèque de fonctions comme un fichier .h que l'on mettra en #include

Calculer le maximum de 3 variables: *int max(var3, max(int var1,var2));*

PASSAGE D'ARGUMENTS A UNE FONCTION

Il existe deux passages d'arguments possibles vers une fonction :

LE PASSAGE PAR VALEUR

(fichier exemple : swap2.c)

Paramètres formels (=copie) : la valeur de la variable ne change pas dans le main, mais seulement dans la fonction. Celui-ci renvoie les valeurs vers le main qui peut les gérer sous un autre nom. Les variables utilisées par la fonction sont ensuite détruites mais restent inchangées dans le main

LE PASSAGE PAR REFERENCE

(fichier exemple : swap.c)

On passe l'adresse de la variable : **pointeurs**. On travaille donc sur les variables réelles via les formats. > VOIR LE CHAPITRE SUR LES POINTEURS

RECURSIVITE DES FONCTIONS

Définition : Une fonction est dite récursive lorsque de manière directe ou indirecte, elle s'appelle d'elle-même.

RECURSIVITE DIRECTE

Lorsque la fonction est appelée dans le corps même de la fonction.

```
...
int fct (char a, char b)
{
    char c,d ;
    int x ;
    ...
    x= fct (c,d) ;
    ...
}
```

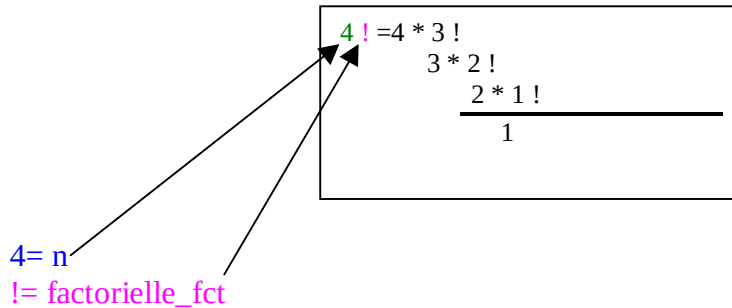
RECURSIVITE INDIRECTE

Lorsque l'appel de la fonction est réalisé dans une fonction appelée dans la première fonction.

```
...
int fct2 (char c, char d)
{
    char e, f ;
    int x ;
    ...
    x=fct_3 (e,f) ;
    ...
}
int fct_3 (char g, h)
{
    char g,h ;
    int y ;
    ...
    y= fct2 (g,h) ;
}
```

Exemple de fonction récursive : la factorielle

```
int factorielle_fct (int n)
{
    if (n == 0){
        return (1) ;}
    else {return (n * factorielle_fct (n-1)) ;}
}
```



n=1 > cas de base mais 0!=1 ...

Cette fonction utilise le système de la PILE pour son calcul (voir CHAPITRE SUR LES PILES LIFO)

Propriétés de la récursivité :

- Les boucles peuvent être remplacées par des fonctions récursives, mais ce n'est pas toujours l'inverse .
- Les fonctions récursives simplifient le traitement d'un problème.
- Plus puissant qu'une boucle mais plus difficile à manipuler
- La pile contient un nombre limité de données, et ne peut être programmée qu'avec des langages ayant une gestion dynamique de la mémoire.
- La profondeur de récursivité (hauteur de la pile) est déterminée lors de l'exécution et non pas lors de la compilation, puisque chaque fois une variable est créée (>propriétés des fonctions)
- Il faut une condition de sortie
- Il y a empilement et dépilement : activation et désactivation de la pile (LIFO)

La récursivité sera grandement exploitée dans les listes chaînées et les arbres.

DECLARATIONS INTERNES/EXTERNES (SCOPE)

\$= déclaration de variables

#include

\$ variables globales

```

main ()
{
  $ variables locales
  {
    {bloc
      $ variable du bloc
    }
  }
}

```

Endroit de déclaration	KEYWORD	CLASSE	DUREE	PORTEE
Dans la fonction	auto	Automatique	Temporaire	Var. locales par défaut
Dans la fonction	register	Registre	Temporaire	Var. locales dans un registre, plus rapide mais limité
Externe	static	Statique	Persistante	Variables locales déclarées ailleurs
Externe	extern	Externe déclarée	Persistante	Variables globales et fonctions déclarées dans un autre fichier
Externe	extern	Externe non-déclarée	Persistante	Variables globales mais dans un seul fichier

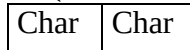
LES TABLEAUX(VECTEURS) []

Définition : Un tableau est un type structuré de données composées dans un espace mémoire continu alloué pour des données.

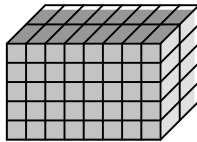
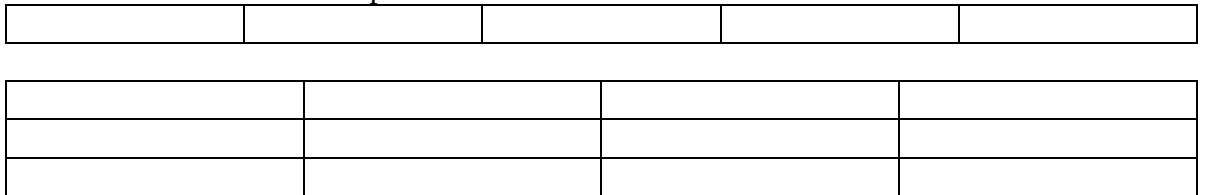
ou

Un tableau une collection de données de même type disposées de manière continue en mémoire.

- Suite séquentielle de cellules (données de même type)



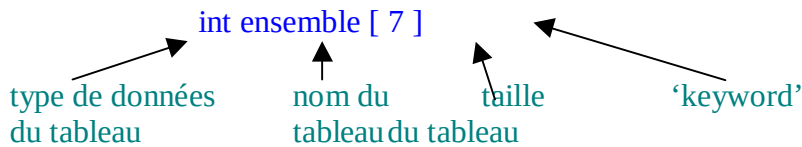
- Taille du tableau connue dès la déclaration (=dimension * type)
- Dimension du tableau : pas de limite



1,2,3 dimensions,...

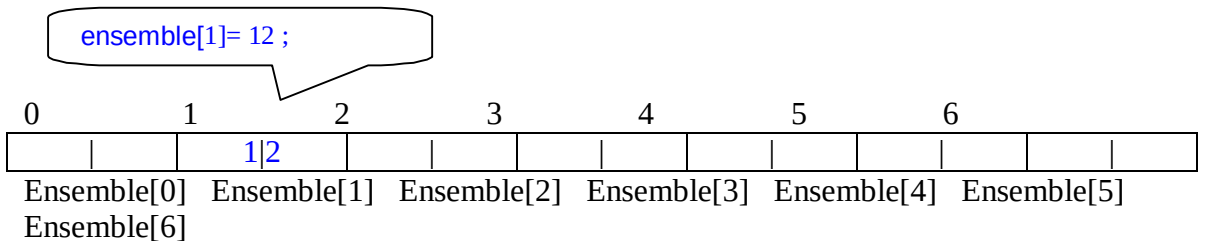
Déclaration:

Un tableau de 7 entiers de nom 'ensemble'



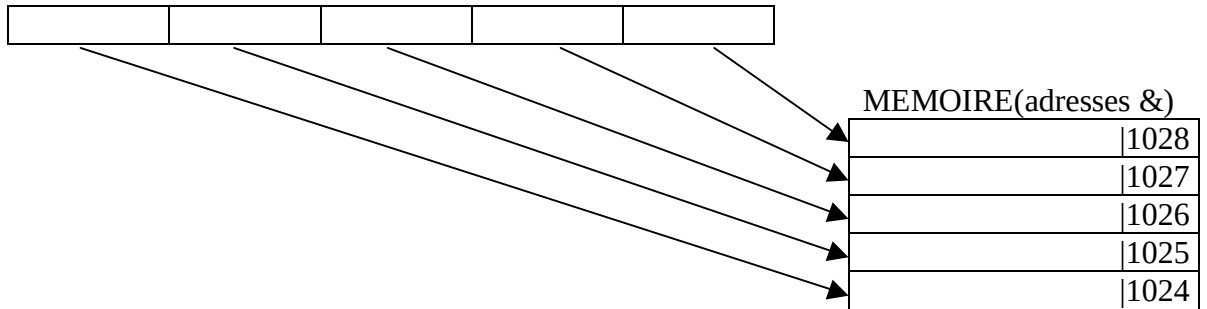
Représentation :

(un integer est codé sur 2 bits)



Adresses continues :

TABLEAU



L'adresse d'un tableau indique sa première case. Le nom d'un tableau est un pointeur vers cette première case.

L'adresse de l'élément [2] = &ensemble[2]

➤ Pointeurs

Déclaration avec définition :

`int list[ ] = {3,22,4,7} ;`

ou

```
int list [4] ;
{
    list= {3,22,4,7}
}
```

1009	???????	????
1008	7	List[3]
1007		
1006	4	List[2]
1005		
1004	22	List[1]
1003		
1002	3	List[0]
1001		
1000	???????	????
Adresse mémoire	DONNEE	ELEMENT

`&list[1] = 1002` /* Adresse de l'élément 1 */

`list[0] = 3` /* Donnée de l'élément 0 */

`&list[0] = 1000`

`list[2] = 4`

`list [4] = ? ? ? ?` ERREUR A LA COMPILATION : ELEMENT INDEFINI !

`&list=` ERREUR A LA COMPILATION : L'adresse est variable à chaque exécution du programme + relocation!

`list= 1000` /* Adresse du tableau */

LES CHAINES DE CARACTERES (strings)

Les strings n'existent pas comme tels en C !

Définition : Une chaîne de caractères est un tableau composé de caractères alphanumériques.

- Rem :** - Toute chaîne est déclarée par un tableau
 - Une chaîne de caractères se termine par '\0'
 ➤ la taille du tableau en dépend : mot de 9 lettres = ch[10]

- comparer 2 chaînes : faire une procédure qui va comparer la valeur ASCII de chaque caractère car. par car.

```
char chaine1[20];
char chaine2[20];
```

```
...
if (chaine1==chaine2)
```

JAMAIS VRAI : les tableaux ne sont pas à la même adresse !

- 'A' = caractère ASCII 65

65

A

« A » = chaîne = tableau

'A'	\0
-----	----

« A »

- chaine1[] = « 123 » ;

'1'	'2'	'3'	'\0'
-----	-----	-----	------

chaine1

PASSAGES DE TABLEAUX COMME ARGUMENT A UNE FONCTION

- Impossible de passer un tableau entier tel quel comme argument
- Pointeur utilisé pour le passage à l'adresse
- Impossible de passer un tableau entier tel quel comme argument
- Pointeur utilisé pour le passage à l'adresse

Exemple :

```
#include <stdio.h>

void main ()
{
    int i [10] ;    /*déclaration d 'un tableau i de 10 entiers */
    ....
    void fonct1 (i) ;    /*appel de la fonction fonct1 recevant i (le pointeur vers le
                        tableau) et ne renvoyant rien */
}
```

Pour recevoir i dans la fonction fonct1 :

➤ **utilisation d'un pointeur**

```
void fonct1 (int *a)
{
    ....
}
```

➤ **utilisation d'un tableau dimensionné**

```
void fonct1 (int a[10])
{
    ....
}
```

➤ **utilisation d'un tableau non-dimensionné**

```
#include <stdio.h>

void main ()
{
    int tab [10] ;
    int a ;
    ....
    a = fonct1 (tab) ;
    ....
}

void fonct1 (int a [ ])
{
    ....
}
```

➤ Aller chercher le 3^e élément du tableau tab et y mettre la valeur 10

`tab [2]=10`

➤ envoyer la valeur de la 3^e case d'un tableau à une fonction

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

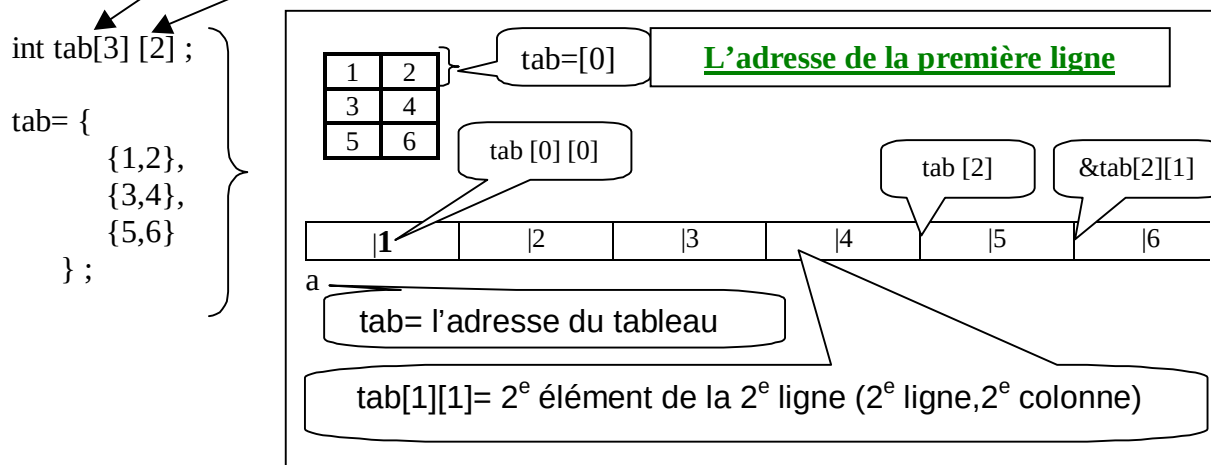
```
int fonction_envoi (int n) ;
```

```
void main ()
{
    int tab[10] ;
    int a ;
    ....
    a= fonction_envoi (tab[2]) ;
    ....
}
```

```
int fonction_envoi (int n)
{
    ....
    n++ ;
    ....
    return (a) ;
}
```

LES TABLEAUX MULTIDIMENSIONNELS (MATRICES) [] []

Ordre : **lignes-colonnes**



!!! La taille maximum d'un tableau avec bibliothèques ANSI est de 64k !!!

LES POINTEURS *

Définition : Un pointeur est un type de variable qui contient l'adresse mémoire d'une autre variable

VARIABLE

Déclaration:

`int a;`

Déclaration-définition:

`Int a=5;`

Représentation :

5

a

POINTEUR

Déclaration:

`int *p;`

Déclaration-définition:

`Int *p = &a;`

Représentation :

&a

*p

nom du pointeur

syntaxe : `int *p = &c`

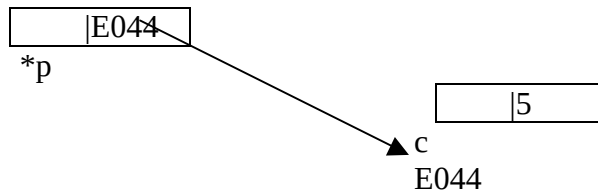
nom de la variable pointée

type de la variable pointée(*) keyword du pointeur opérateur d'adressage de la variable pointée

- Un tableau n'a pas besoin de l'opérateur & pour être pointé.
- Un pointeur contient une adresse mémoire, donc un entier

Représentation :

(un integer est codé sur 2 bits)



Exemple de déclaration d'un pointeur de nom 'pat' vers un tableau 'tab' de caractères de 2lignes/3colonnes.

```
char tab [2][3] ;
char [2][3] *pat = tab ;
```

➤ Aller chercher une valeur dans un pointeur (inversion de variable)

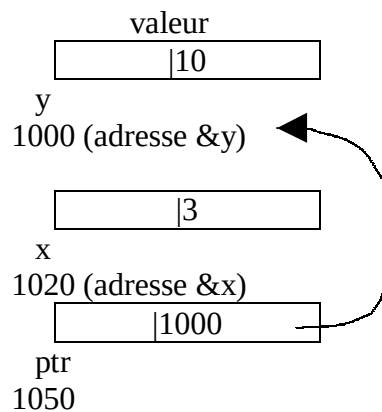
```
int x ;
int y ;
int *ptr= &x ;
```

*y=*ptr ; /*y prend la valeur de la variable vers laquelle ptr pointe (x).Maintenant y prend la valeur de x*/*

```
x=y ;
ptr= &y ;
```

Représentation de l'action d'un pointeur :

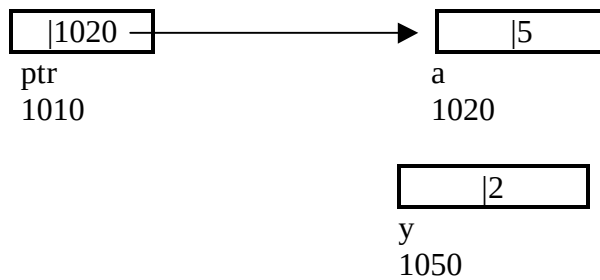
```
ptr= &y ;
*ptr= y ;
```



Code C

```
int a=5 ;
int y=2 ;
int *ptr=&a ;
```

Représentation Schématique



variable **a=5**
 variable **ptr=1020**
 adresse **&a=1020**
 valeur pointeur ***ptr=5**
 ***y= RIEN** car pas un pointeur
 adresse du pointeur **&ptr=1010**

Remarques sur les syntaxes portant à confusion :

`a*=*ptr ;` `a=5*5` =25

`y=a* *ptr ;` `y=5*5` =25

Valeur du pointeur ptr

~~`y= a*ptr`~~ `y=1020` **ERREUR**

➤ **Il existe un pointeur nul : `int *ptr= NULL ;`**

Exemples de pointeurs

`int i,j,*p ;`

`i=5 ;`

5
|_|_|
i
1000

`p=&i ;`

1000
|_|_|
p
1020

`j=*p ;`

5
|_|_|
j
1040

`*p= j+2 ;`

7
|_|_|
i
1000

VALEURS FINALES :

1000
|_|_|
p
1020

5
|_|_|
j
1040

OPERATIONS SUR LES POINTEURS

On peut effectuer les opérations suivantes :

+ **-** **<** **>** **<=** **=>**

On peut faire : **ptr1 - ptr2**

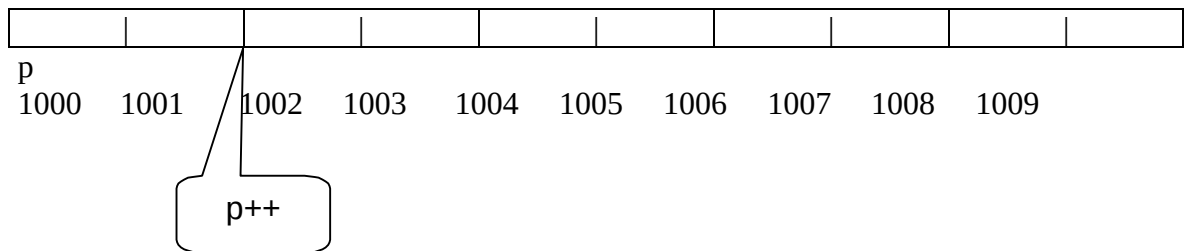
Exemples

```
int *p
```

```
p=p+1            /*p++*/
```

```
/*on passe à la case mémoire suivante en bytes*/
```

```
/*un integer prend 2 bytes*/
```



```
printf (" %d ", p) ;    > ?
```

TOUT DEPEND DU TYPE DE LA VARIABLE VERS LAQUELLE LE POINTEUR POINTE

Si p avait pointé vers un char, la valeur de p aurait été de 1001 car un char ne prend qu'un byte

```
tab[10] ;            }
ptr=tab ;            }    passe à l'élément suivant du tableau (tab[0] > tab[1])
ptr++ ;              }
```

Si on pointe vers un tableau, on pointe en fait vers son 1^{er} élément (element[0]). Si on affiche la valeur d'un pointeur vers le tableau 4|5|6 , ce sera 4.

Si on incrémente le pointeur de 1, il affichera 5 et ainsi de suite...Utile pour afficher une chaîne de caractères !

POST ou PRE –incrémentation ? différences :

```
*(++p) avant le ;    |_|_|
                     p
                     1001
après le ;            |_|_|
                     p
                     1001
```

```
*(p++) avant le ;    |_|_|
                     p
                     1000
après le ;            |_|_|
                     p
                     1001
```

UTILISATION DE LA FONCTION SIZEOF

La fonction sizeof renvoie la taille du type utilisé

$p + \text{nombre} = p + (\text{nombre} * \text{sizeof}(\text{type}))$

Utile pour une incrémentation de pointeur et une allocation dynamique

PASSAGES DES ARGUMENTS DE LA LIGNE DE COMMANDE

Utilisation des paramètres lors de l'exécution du programme dans le système d'exploitation

Exemples : MS-DOS `dir /s /p` ou `copy file1.exe file2.exe`
 UNIX `ls -a` ou `cp file1 file2`

En C, il existe deux paramètres de réception, ou arguments, qui doivent être utilisés à cet effet : argc et argv.

Le programme calc_2.c est en fait le programme calc_1.c remanié afin d'introduire l'opération mathématique lors de la commande, tels des arguments.

A:\> calc_2 4+3

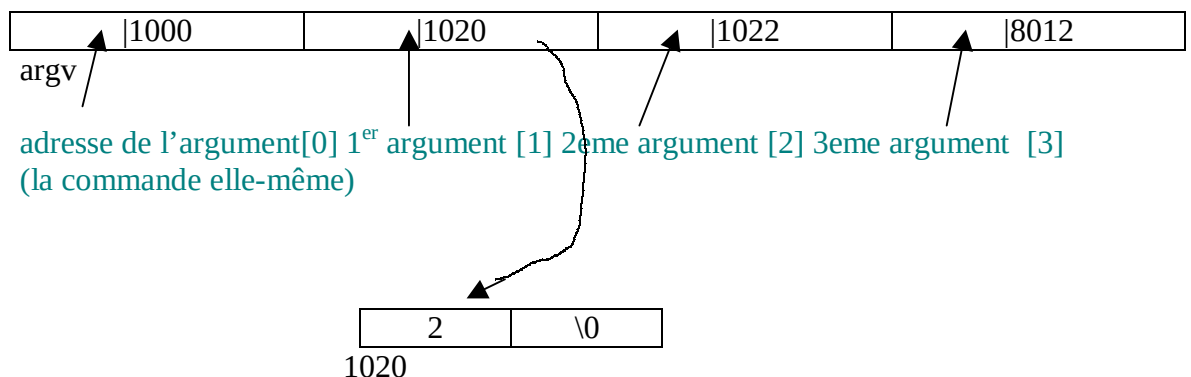
Syntaxe :

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

compteur d'arguments

le numéro entre les crochets indique le Nieme argument : argv[0] étant la commande elle-même, argv[1] le 4 , argv[2] le + et argv[3] le 3 dans l'exemple de la calculatrice ci-dessus.

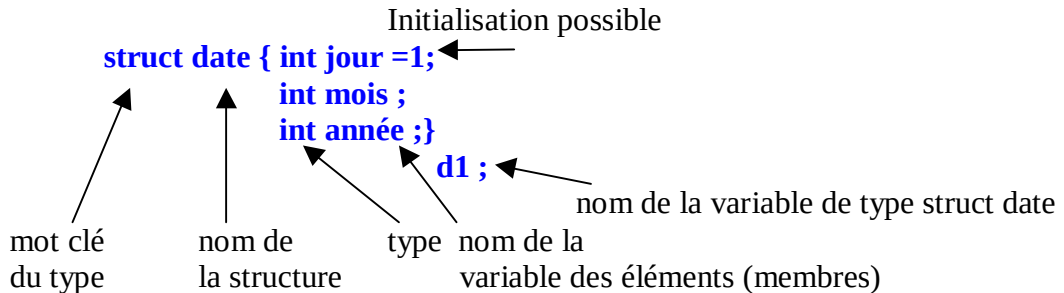
argv est un tableau de pointeurs qui pointent vers des caractères.



le 1^{er} argument : une variable chaîne de caractères

LES STRUCTURES

Au départ, les structures ont les mêmes propriétés qu'un tableau. Mis à part que les données dans une structure peuvent avoir des **types différents**.



Un nouveau type '**date**' a été créé. On va utiliser ce type pour déclarer deux nouvelles variables "my_birthday" et "d2":

```
struct date my_birthday, d2 ;
```

On définit ensuite: `my_birthday={22,6,98}`

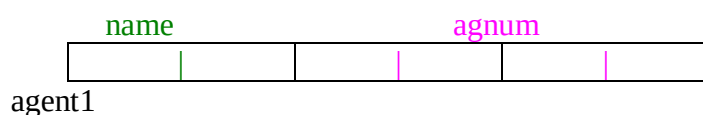
Ainsi, la variable '*my_birthday*' contient les valeurs 22 06 98

Déclaration de *agent1* : variable de type

```
struct_personnel {char *name ;
                  long int agnum ; } agent1 ;
```

On peut aussi utiliser:

```
typedef struct_personnel personnel;
personnel agent1; /*déclaration de la variable "agent1 de type
(structurel) personnel*/
```



```
struct_personnel ptr ;      /*déclaration d'un pointeur de type struct_personnel*/
ptr=&agent1 ;              /*le pointeur pointe vers l'adresse de agent1*/
ptr++                      /*passe à « une struct_personnel (6bytes)plus loin */
printf (« %d », ptr) ;     /*affiche l'adresse pointée*/
```

Pour mettre une valeur dans « name » avec un pointeur, il faut utiliser :

```
struct_personnel *ptr
ptr= &agent1
```

avec pointeur : `ptr -> name = « Roger » ;`

soit

sans pointeur : `ptr=agent1.name= »Roger » ;`

pour afficher : `printf(« %c » ptr->name) ;`

ou encore : déclarer `long int *ptr2=&(agent1.agnum) ;`

puis, comme avec pointeur : `*ptr2= 4000`

```
struct eleve { int age ;
               char *nom ;
               char prenom ;
               long int nrmecanographique ;} ;
```

```
struct eleve_82F [8]= {
    {24, »Brebant », 'F', 940716}
    {19, »Evrard », 'F', 970111}
    {19, »Jamart », 'V', 9702116}
    {21, »Rolin », 'D', 97012234}
    {21 »Martens », 'P', 96123456}
    {22 »Simon », 'C', 96214678}
    {20 »Hamende », 'M', 97876234}
    } ;
```

Mettre un 3eme élément : `82F[2]= {25, »Marechal », 'G', 9501234} ;`

```
Ou
    ptr & 82F [2] ;
    *ptr= {25, »Marechal », 'G', 9501234} ;
    ptr++ ;
```

Changer le premier élément : `82F[0].nom= »Dupuis » ;`

```
Ou
    ptr= &82F[0] ;
    ptr -> nom= « Dubois » ;
```

Exemples supplémentaires

```
struct date {
    int jour;
    int mois;
    int annee;
};

struct compte {
    int no_compte;
    char etat;
    char nom[80];
    float solde;
    struct date date_der_vers;
};
```

```
struct cpt1={501,'B',"depenses",-20000,10,12,1999};
```

```
/* accès a une donnée*/
```

```
soit:
```

```
    cpt.date_der_vers.annee
    cpt.solde+=10000;
```

```
ou:
```

```
    struct compte *pcpt;
    pcpt=&cpt;
    *pcpt->solde;
```

```
/*creation de comptes:*/
```

```
soit:
```

```
    struct compte tab[100] /*100 comptes*/
```

```
ou:
```

```
    struct compte tab[]={
                                501,'B',"depenses",-20000,12,12,1999,
                                502,'O',"recettes",35000,21,01,2000,
                                503,'A',"divers",10000,20,01,2000
                                };

```

Passage d'une structure à une fonction: par valeur

```
void affichedate (struct date madate)
```

```
{
    printf("%d/%d/%d\n", madate.jour,madate.mois,madate.annee);
}
```

```
void affichecompte (struct compte cpt)
```

```
{
    printf("Numero:%d\n",cpt.no_compte);
    printf("Etat:%c\n",cpt.etat);
    printf("Libelé:%s\nSolde:%f\n",cpt.nom,cpt.solde);
    affichedate (cpt.date_der_vers);
};
```

```
int main ()
```

```
{
    affichecompte(tab[1]); /*on va traiter le compte 501*/
    return 0;
}
```

Passage d'une structure à une fonction: par parametre

```
void ajoutsolde (float *psolde)
```

```
{
    (*psolde)++;
}
```

```
int main()
{
    ajoutsolde(&cpt.solde);
    return 0;
}
```

Retour d'une structure depuis une fonction

```
struct compte ajoutsolde (struct compte cpt)
{
    struct compte moncpt;
    moncpt.solde ++;
    return moncpt;    /*retour de la copie comme résultat>modifie la structure */
}

int main()
{
    tab[1]=ajoutsolde(tab[1]);
    return 0;
}
```

Création d'un type nouveau: typedef (simplifie l'accès dans les structures)

```
typedef char chaine[80]; /*creation du type de données chaine: simple abus de langage*/
chaine ch1;    /*declaration de la variable ch1 de type ^ chaine*/

typedef struct compte cptbank; /*permet de remplacer struct compte par cptbank*/

cptbank compte[100]; /*declaration d'un tableau de 100 comptes de type struct compte*/

/*ainsi on peut encore aller aussi loin que l'on veut:*/

typedef cptbank tcpt[100];
tcpt tableau[100];

typedef char string[20];
string nom;
```

Exemples supplémentaires de manipulation de structures/fonctions

`scanf(" %[\n]",var.nom);` /*pour permettre les espaces dans le string*/

```
struct demo *pstruct=NULL;
pstruct=&d;
```

`/*Attention:...*/`

```
void mafunct (int tab[])
{
    printf("%d",sizeof tab);    /*output: 4*/
}
```

```
int main ()
{
    int tab[100];
    printf("%d",sizeof tab);    /*output...: 400 */
    mafunct (int tab[]);
}
```

UNIONS: TYPE COMPLEXE

Définition:

Structure dont tous les membres se partagent la même zone en mémoire.

```
union monunion
{
    double tabd[200000];
    char memo[1600000];
};
```

Comparaison avec la structure:

```
struct t
{
    int i;
    float f;
};

/*en mémoire:      |_i_|_f_| */
```

```
union t
{
    int i;
    float f;
}

/*en memoire      |_i_|
                   f */
```

L'union occupe la place mémoire égale à la taille du plus grand de ses membres. c'est utile dans le cas où les données introduites sont soit int soit float.. Il y a un gain de l'espace d'un int lorsqu'on introduit uniquement un float.

Manipulations d'unions:

```
typedef union
```

```
{
    double tabd[200000];
    char memo[1600000];
} montype;
```

```
typedef struct
```

```
{
    long int i;
    montype tab;
    char classe;
} mastruct;
```

```
mastruct v[10]; v[0].classe='s';    /*on utilisera seulement 16Mb au lieu de 32Mb*/
```

```
strcpy(v[0].tab.memo,"bonjour");
```

```
v[1].classe='d';
```

```
for (i=0;i<200000;i++)
```

```
{
    v[1].tab.tabd[i]=i;
}
```

FICHIERS: IMPORT/EXPORT, TEXTE/BINAIRE

Types de fichiers: -texte : accès séquentiel
 -binaires : accès séquentiel et direct(offset)

Par défaut: stdin: clavier; stdout:ecran; stderr:ecran

Manipulations générales de fichiers:

Déscripteur:

FILE *filename1,*filename2;

Ouverture:

filename1= fopen("texte.txt", "r");

Fermeture:

fclose(filename1);

Test de fin de fichier:

fEOF(filename1);

FICHIERS TEXTE

Manipulations spécifiques aux fichiers texte:

Modes d'ouverture:

r	:	lecture seule, debut fichier
w	:	écriture seule, écrase
a	:	écriture seule, fin de fichier (ajout)
r+	:	r/w, debut fichier
w+	:	r/w, écrase
a+	:	r/w, fin de fichier (ajout)

Se replacer au début:

rewind(filename1);

Lecture:

fgets(chaine,80,filename1); /*taille maximum:80*/

/*on peut donc aussi faire: fgets(chaine,80,stdin); qui équivaut à gets(chaine); mais avec une standardisation des canaux I/O.*/

fscanf(filename1,"%d",&i);

fscanf(filename1,"%s %s %d",s.nom,s.prenom,&(s.age));

/*fscanf retourne 1 s'il a su lire 1 élément, 0 si c'est EOF*/

On peut donc traiter le EOF comme ceci:

```
while (fscanf(filename1,%d,&i)==1)
{
    <operations sur le fichier>
}
if (!feof(filename1))
{
    printf("Erreur sur le support de données");
    exit (1);
}
```

Passage d'un fichier dans une fonction:

```
fonctionname (FILE *filename1)
{
    ...
}

int main()
{
    fonctionname(filename1);
}
```

Ecriture:

fputs (chaine,filename1);

fputc(c,filename1);

fprintf (filename1 "%d",i);

On peut donc traiter le EOF comme ceci:

```
if (fprintf(filename1 "%d",i)==EOF) /*EOF=-1*/
{
    /*erreur ou fin de fichier*/
    fprintf(stderr,"\nErreur d'écriture\n"); /*stderr est l'écran par défaut*/
}
```

REM: Un passage à la ligne en DOS se fait par crlf mais uniquement par cr en UNIX. On a donc des fichiers DOS avec chaque fin de ligne en ^M sous UNIX (>script)

Exemple: copie d'un fichier texte dans un autre fichier texte:

```
#include <stdio.h>
/*Les noms des fichiers sont passés en argument à la ligne de commande*/
/*Le format crlf est modifié si on passe de UNIX à DOS*/

int main (int argc, char *argv[])
{
    int c; /*tampon de copie*/
    FILE *in=stdin, *out=NULL;
    if (argc < 2 || argc > 3)
    {
        fprintf(stderr, "Min. 1 et Max. 2 paramètres à la commande");
        return(1);
    }

    if (argc>=2)
    {
        out=fopen(argv[2],"w"); /*ouverture du fichier cible en écriture*/
        if (out==NULL)
        {
            fprintf(stderr,"Erreur d'ouverture du fichier cible %s\n",argv[2]);
            return (1);
        }
    }

    if (argc==3)
    {
        in=fopen(argv[1],"r"); /*ouverture du fichier source en lecture*/

        if(in == NULL)
        {
            fprintf(stderr, "Fichier source %s vide.\n",argv[1]);
            return 1;
        }
    }

    while ((c=fgetc(in))!=EOF)
    {
        /*lecture tant que pas fin de fichier et stockage dans "c"*/
        fputc (c,out); /*écriture dans le fichier output*/
    }

    if (!feof(in))
    {
        /*on ne sait plus lire dans le fichier input mais ce n'est pas EOF*/
        fprintf(stderr, "Erreur de lecture dans Fichier Input.");
        fclose(in);
        return 1;
    }
    fclose (out);

    return 0;
}
```

FICHIERS BINAIRE

Manipulations spécifiques aux fichiers binaires:

Modes d'ouverture:

rb	:	lecture seule, debut fichier
wb	:	écriture seule, écrase
ab	:	écriture seule, fin de fichier (ajout)
rb+	:	r/w, debut fichier
wb+	:	r/w, écrase
ab+	:	r/w, fin de fichier (ajout)

Accès séquentiel (exemples):

```
fclose (fiche);
int i=8;
int tab[15];
struct eleve { int ident;
               char nom[30];
               char prenom[30];
               int age; }
struct eleve elem1;
```

```
fread(&i,sizeof (int), 1, fiche); /*lit un int du support vers la mémoire*/
fwrite(&i,sizeof (int), 1, fiche); /*écrit un int de la mémoire vers le support */
```

```
fread(&elem1,sizeof (struct eleve), 1, fiche); /*lit une structure du support vers la
mémoire*/
fwrite(&elem1,sizeof (struct eleve), 1, fiche); /*écrit une structure de la mémoire vers le
support */
```

```
fread(tab,sizeof (int), 15, fiche); /*lit un tableau de 15 int du support vers la mémoire*/
fwrite(tab,sizeof (int), 15, fiche); /*écrit un tableau de 15 int de la mémoire vers le
support */
```

Accès direct

On se place sur l'offset de l'élément n à lire, c'est à dire qu'on se place à la fin de l'élément qui le précède juste (donc on lit les n-1 premiers éléments):

```
fseek (filename1, [EFFET DE DEPLACEMENT],[SEEK_TYPE]);
```

SEEK_SET: début du fichier
SEEK_CUR: position actuelle
SEEK_END: fin de fichier

Accès direct (exemples):

`fseek (fiche, 0L, SEEK_SET); /* se déplace au début du fichier */`

`fseek (fiche, 3 * sizeof(SEEK_CUR), SEEK_SET); /*se déplace de 3 éléments de la taille de celui sur lequel on se trouve.*/`

Exemple: donner la taille totale ou partielle d'un fichier binaire

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

```
FILE *fichier = NULL;
```

```
int taillebin(FILE *fichier)
{
```

```
    int choix;
```

```
    printf("Voulez vous savoir la taille totale ou depuis ici? (0/1)");
```

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

```
    if (choix==0)
```

```
    {
```

```
        fseek (fichier,0L,SEEK_END); /*du début à la fin */
```

```
        return ftell(fichier);    /*taille totale */
```

```
    }
```

```
    else
```

```
    {
```

```
        fseek (fichier,0L,SEEK_SET); /*retour au début du fichier*/
```

```
        fseek (fichier,2*sizeof(SEEK_SET),SEEK_CUR);/*se placer 2 éléments de même taille plus loin que le début du fichier */
```

```
        return ftell(fichier); /*taille du fichier à partir de l'emplacement actuel jusque la fin du fichier */
```

```
    }
```

```
}
```

```
/* ATTENTION AU PASSAGE D'UN FICHIER PAR POINTEUR DANS UNE FONCTION
CAR ON DOIT METTRE UN POINTEUR SUR UN POINTEUR DE FICHIER*/
```

```
int ouvrir_file()
```

```
{
```

```
    fichier = fopen("test.bin", "rb");
```

```
    if (fichier == NULL)
```

```
    {
```

```
        fprintf(stderr, "Impossible d'ouvrir le fichier test.bin\n");
```

```
        return 1;
```

```
    }
```

```
    else
```

```
    {
```

```
        return 0;
```

```
    }
```

```
}
```

```
void fermer_fichier()
```

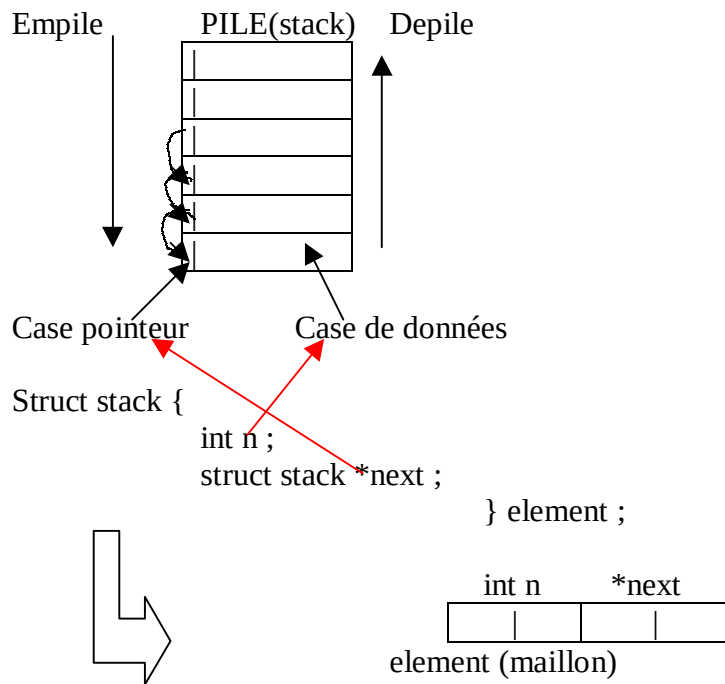
```
{
```

```
    fclose(fichier);
```

```
}
```

```
int main ()
{
    /*ouvrir_file();*/
    if(ouvrir_file()==0)
    {
        printf("Taille:%d",taillebin(fichier));
        fermer_fichier();
        return 0;
    }
    else
    {
        printf("Erreur sur fichier");
    }
}
```

UNE STRUCTURE AUTO-REFERENTIELLE: LA PILE



Ces listes chaînées seront détaillées plus loin

ALLOCATION DYNAMIQUE DE LA MEMOIRE ET IMPLEMENTATION A LA PILE

Malloc : fonction qui retourne un pointeur vers une zone de n bytes contigus et à laquelle on transmet le type de la structure. Cette fonction est contenue dans la bibliothèque « alloc.h » ou « stdout.h ».

```
void malloc (size_t n) ; /* fonction malloc qui renvoie un pointeur vers rien et qui reçoit
                           une variable n de type size_t */
```

```
pos=(struct node*) malloc (sizeof (struct node)) ;
```



 nom type casting appel de taille en bytes de
 d'une malloc de la variable
 variable
 pointeur
 vers node

Exemple simple de malloc:

```
d=malloc(sizeof(double));
*d=5.0; /*écriture rapide de 5 pour un float: indiquer le . */
*d=*d+2.0;
printf("valeur de d:%f",d);
d=NULL;
free(d);
```

Free : Supprime les espaces mémoire de **malloc** devenus inutiles.

```
void free (*pos) ;
```



 adresse de la mémoire à libérer

malloc d'un tableau

```
int *dtab;
int size;
printf("nombre d'éléments du tableau d'entiers?");
scanf("%d",&size);
dtab=(int*)malloc(size * sizeof(int));
dtab[0]=5;
```

modifier la taille (+ ou -) du tableau ci-avant

```
dtab=realloc(dtab,sizeof(int)*30 /*ajout de 30 cases*/
```

Allocation dynamique de tableaux multidimensionnels (matrices)

```

/*[10][6] d'entiers:*/
int **tab;
tab=(int*) malloc (sizeof(int*)*10); /*création de la 1e dimension 'index vertical'*/
for (i=0;i<10;++i)
{
    tab[i]=NULL; /*mise à blanc des pointeurs d'index, plus propre*/
}

for (i=0;i<10;++i);
{
    tab[i]=(int*)malloc(sizeof(int)*5)); /*création de la 2e dimension pointée dans
'l'index vertical' */
}
    
```

Application à la pile

Créer un nouvel élément de la pile

```

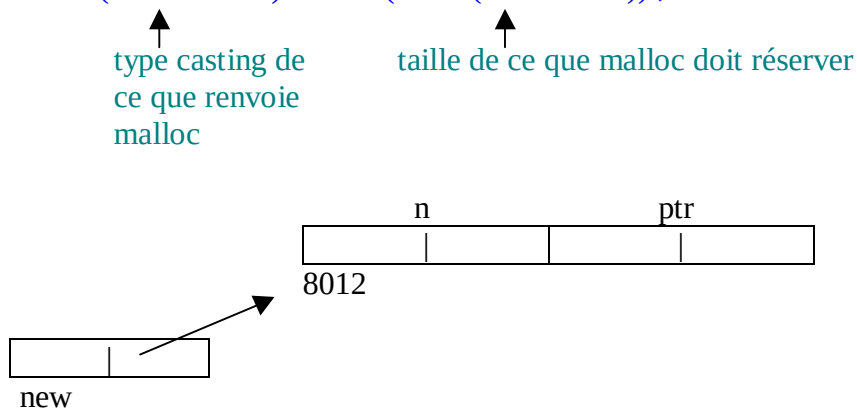
cpt=1 ;
struct stack {int n ;
               struct stack *ptr ;
               } ;
    
```

1) Créer la pile vide

```
struct stack *new=NULL,top=ptr ;
```

2) Assigner une adresse mémoire

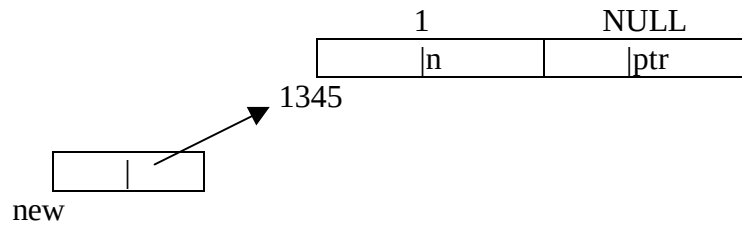
```
new= (struct stack*) malloc (sizeof(struct stack)) ;
```



3) Remplir la valeur n du 1^{er} élément de la pile

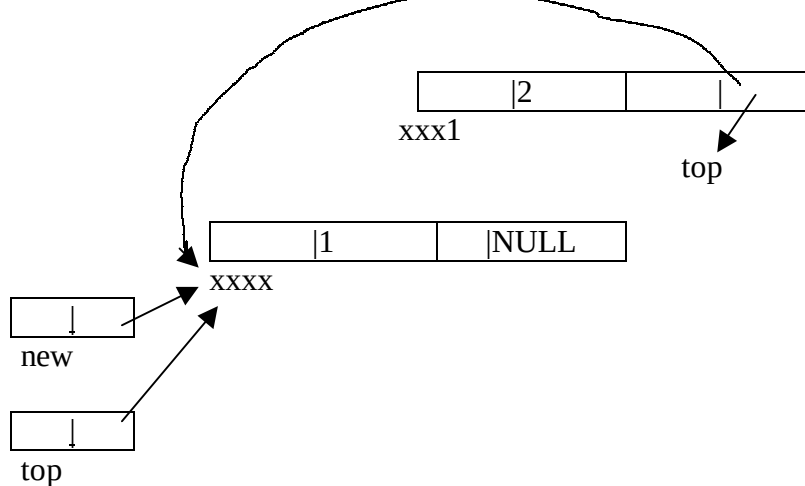
```

new -> n= cpt
new -> ptr= NULL
    
```



4) Remplir la suite d'une pile

`top=new ;` /* envoie le pointeur vers top puis déplace le sommet */



`new->ptr=top ;`
`n= cpt ++ ;` ou `new ->n =cpt ++ ;`

`top=new ;` /*déclare que c'est le dernier élément qui vient d'être crée et qu'il est le top de la pile */

5) Dépiler (a répéter autant de fois qu'il y a d'éléments créés)

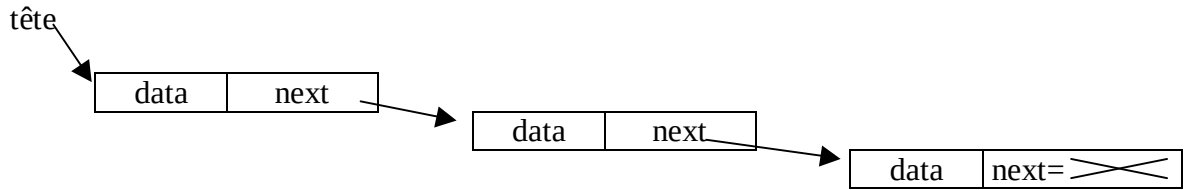
Afficher `printf (« %d », top -> n) ;` /* chaine avec espaces : utiliser puts */
`top= new`

Utiliser **free** pour supprimer l'élément de la pile qui a déjà été imprimé car il est devenu inutile.

`free (new) ;`

Passer à l'élément suivant : `new= top ;`

LISTES CHAINEES (linked lists)



```

struct maillon {
    int data;
    struct maillon *next;
}
typedef struct maillon pmaillon; /*creation du type structuré pmaillon*/
    
```

NOTIONS:

Créer un maillon seul

```

pmaillon p; /*déclaration d'un nouveau maillon nommé p*/

p=(pmaillon) malloc (sizeof(struct pmaillon)); /*création du maillon p*/

p->data=6; /*inscrire la valeur 6 dans le champ data du maillon p*/
    
```

Libérer un maillon seul

```
free(p);
```

Lecture séquentielle à partir de p

```

pmaillon p;

for (p=tête; p!=NULL;p=p->next)
{
    printf("\nValeur: %d", p->data);
}
    
```

Généralisations

/*toujours commencer par initialiser la tête à NULL*/

1) Liste vide

```

if(tete=NULL)
{
    tete=(pmaillon)malloc(sizeof(struct maillon));
    tete->data=v; /*valeur décidée pour ce maillon */
    tete->next=NULL;
}
    
```

2) Liste existe, insérer en début de liste

```
p=(pmaillon)malloc(sizeof(struct maillon));
p->data=v;
p->next=tete; /*accrocher à la tête de liste existante */
tete=p;
```

2) Liste existe, insérer dans ou en fin de liste

```
p=(pmaillon)malloc(sizeof(struct maillon));
p->data=v;
p->next=NULL;
q1->next=p; /*nécessite un pointeur sur l'élément juste inférieur à p. on raccroche */
p->next=q2; /*nécessite un pointeur sur l'élément juste supérieur à p. on pointe */
```

3) Liste existe, supprimer la tête de liste

```
tmp=tete;
tete=tmp->next;
free(tmp);
```

4) Liste existe, supprimer dans ou la fin de liste

```
tmp=p->next;
p->next=tmp->next;
free(tmp);
```

FONCTIONS DE TRAITEMENT:

```
void initialiser_tete (pmaillon tete) /*pointeur sur pointeur*/
{
    *tete=NULL;
}

int main ()
{
    pmaillon tete;
    initialiser(&tete);
}

/*****/

void afficher(pmaillon tete)
{
    pmaillon p=NULL;
    p=tete;
    while (p!=NULL)
    {
        printf("\nValeur: %d",p->data);
    }
}
```

```

        p=p->next;
    }
}

/*****/

void insertion_debut (pmaillon *tete, int valeur)
{
    pmaillon p;
    p=(pmaillon)malloc(sizeof(struct maillon));
    p->data=valeur;
    p->next=*tete;
    *tete=p;
}

/*****/

void insertion_fin (pmaillon *tete, int valeur)
{
    pmaillon p,q;
    p=(pmaillon)malloc(sizeof(struct maillon));
    p->data=valeur;
    p->next=NULL;
    if (*tete=NULL)
    {
        *tete=p;
    }
    else
    {
        q=tete;
        while (q->next !=NULL)
        {
            q=q->next;
        }
        q->next=p;
    }
}

/*****/

void vider (pmaillon *tete)
{
    pmaillon tmp;
    while (tete !=NULL)
    {
        tmp=tete;
        tete=tete->next;
        free(tmp);
    }
}

```

EXEMPLES DE FONCTIONS D'UTILISATION:

pmaillon inverse (pmaillon liste)

```
{
    /*écrit une liste existante dans une nouvelle de droite à gauche → insertion en
    tête*/

    pmaillon p;
    pmaillon nv_liste=NULL;
    p=liste;
    while (p!=NULL)
    {
        nv_liste=inserer_tete(nv_liste,p->data);
        p=p->next;
    }
    return nv_liste;
}
```

pmaillon mergehead (pmaillon tete, int val)

```
{
    /*ajoute le contenu d'une liste à l'avant d'une autre */

    pmaillon tmp;
    tmp=(pmaillon)malloc(sizeof(struct maillon));
    tmp->data=val;
    tmp->next=tete;
    tete=tmp;
    return tete;
}
```

pmaillon mergeend (pmaillon liste1, pmaillon liste2)

```
{
    /*ajoute le contenu d'une liste à la suite d'une autre */

    if(liste1==NULL)
    {
        return liste2;
    }
    else
    {
        p=liste1;
        while (p->next !=NULL)
        {
            p=p->next;
        }
        p->next=liste2;
        return liste1;
    }
}
```

```
int present (pmaillon liste, int elem)
{
    /*teste si un élément est présent dans une liste et retourne 1 ou 0*/

    int sortie=0;
    pmaillon p;
    p=liste;
    while (p !=NULL)&&(sortie==0)
    {
        if (p->data==elem)
        {
            sortie=1;
        }
        p=p->next;
    }
    return sortie;
}
```

/****** TRI: tri d'une liste par ordre décroissant *****/

```
pmaillon decroche(int v, pmaillon liste, pmaillon maillon)
{
    pmaillon p;
    if(maillon==liste)
    {
        p=liste;
        liste=liste->next;
        p->next=NULL;
    }
    else
    {
        p=liste;
        while(p->next!=maillon)
        {
            p=p->next;
        }
        p->next=maillon->next;
        maillon->next=NULL;
    }
    return liste;
}
```

```
pmaillon min (pmaillon liste)
{
    pmaillon p;
    pmaillon ppetit;
    p=liste;
    ppetit=liste;
    if (liste==NULL)
```

```

        {
            return NULL;
        }
        else
        {
            while (p!=NULL)
            {
                if (p->data < ppetit->data)
                {
                    ppetit=p;
                }
                p=p->next;
            }
            return ppetit;
        }
    }

pmaillon insert_maillon (pmaillon liste->pmaillon, pmaillon maillon)
{
    maillon->next=liste;
    liste=maillon;
    return liste;
}

pmaillon tri (pmaillon liste)
{
    /*enfin..*/
    pmaillon p;
    pmaillon nv_liste=NULL;
    while (liste !=NULL)
    {
        p=min(liste);
        liste=decroche(liste,p);
        nv_liste=insert_maillon (nv_liste,p);
    }
    return nv_liste;
}

/*****TRI2: tri par insertion *****/

pmaillon decroche_tete (pmaillon liste)
{
    if (liste!=NULL)
    {
        pmaillon p;
        p=liste;
        liste=p->next;
        p->next=NULL;
    }
    return liste;
}

```

```

    }

    pmaillon insert_tri (pmaillon liste, pmaillon maillon)
    {
        int sortie=0;
        pmaillon p;
        if (liste==NULL)
        {
            liste=maillon;
        }
        else
        {
            p=liste;
            if (p->data < maillon->data)
            {
                maillon->next=liste;
                liste=maillon;
            }
            else
            {
                while ((p->next !=NULL)&&(sortie==0))
                {
                    if ((p->next)->data < maillon->data)
                    {
                        sortie=1;
                    }
                    else
                    {
                        p=p->next;
                    }
                    maillon->next=p->next;
                    p->next=maillon;
                }
            }
        }
        return liste;
    }

    pmaillon tri2 (pmaillon liste)
    {
        pmaillon nv_liste=NULL;
        pmaillon p;
        while (liste !=NULL)
        {
            p=liste;
            liste=decroche_tete(liste);
            nv_liste=insert_tri(nv_liste, p);
        }
        return nv_liste;
    }

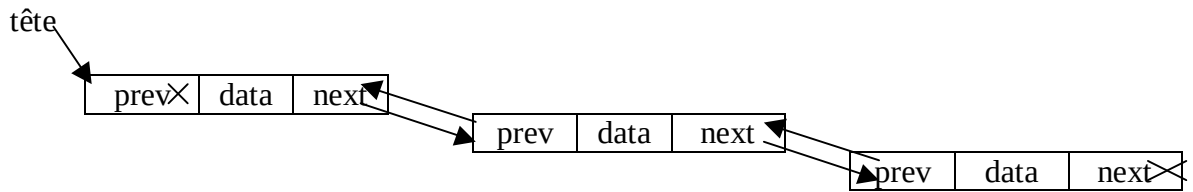
```

STRUCTURES UTILISANT DES LISTES:

PILE LIFO: insert_tete; depile_tete

FILE FIFO: insert_fin; depile_tete; pointeur supplémentaire sur fin de liste pour les ajouts.

LISTES DOUBLEMENT CHAINEES (double linked lists)



```

struct maillon {
    int data;
    struct maillon *next;
    struct maillon *prev;
}
typedef struct maillon pmaillon; /*creation du type structuré pmaillon*/
  
```

/****** Tri d'une liste doublement chaînée (voir le TP sur l'agenda2) *****/

```

pmaillon insere_tete (pmaillon liste, pmaillon maillon)
{
    maillon->next=liste;
    maillon->prev=NULL;
    liste->prev=maillon;
    liste=maillon;

    return liste;
}

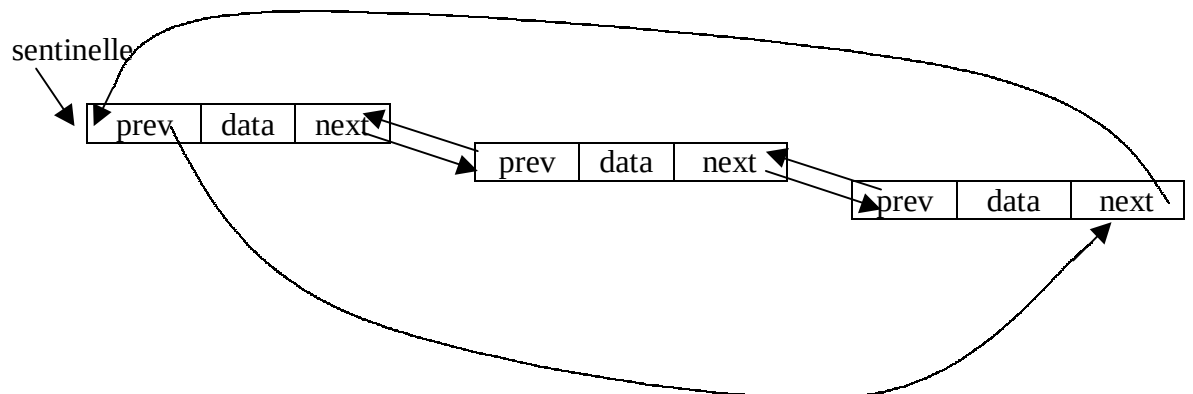
pmaillon insere_trie (pmaillon liste, pmaillon maillon)
{
    pmaillon p;
    int sortie;
    if (liste==NULL)
    {
        liste=maillon;
    }
    else
    {
        p=liste;
        if(p->data < maillon)
        {
            liste insere_tete (liste,maillon)
        }
        else
        {
            while ((p->next!=NULL)&&(sortie==0))
            {
                if ((p->next)->data < maillon->data)
                {
                    sortie=1;
                }
            }
        }
    }
}
  
```

```

    }
    else
    {
        p=p->next;
    }
    maillon->next=p->next;    /*!!! ordre*/
    maillon->prev=p;
    p->next=maillon;
    if (maillon->next !=NULL)
    {
        (maillon->next)->prev=maillon;
    }
}
}
return liste;
}

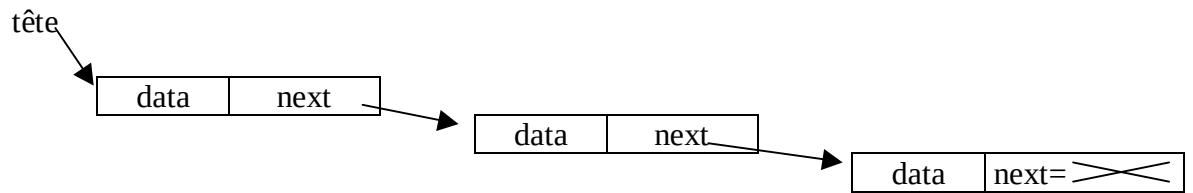
```

LISTES CIRCULAIRES (round robin)

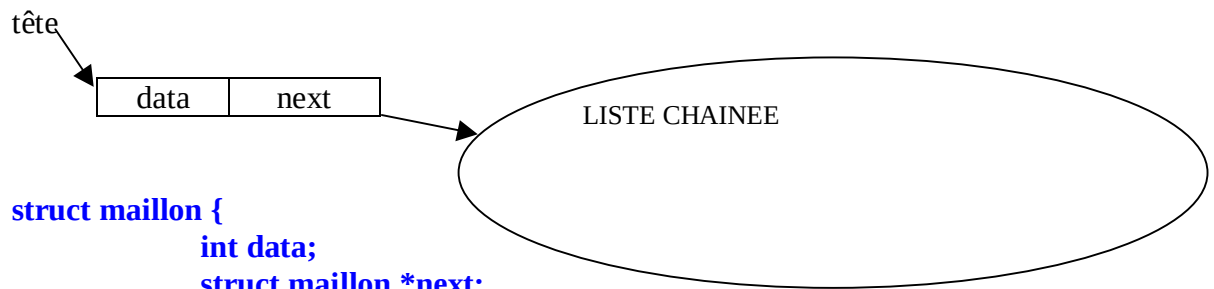


Utilisé pour les schedulers des systèmes d'exploitation.

RECURENCE DES LISTES CHAINEES



Equivaut à:



```
struct maillon {
    int data;
    struct maillon *next;
}
```

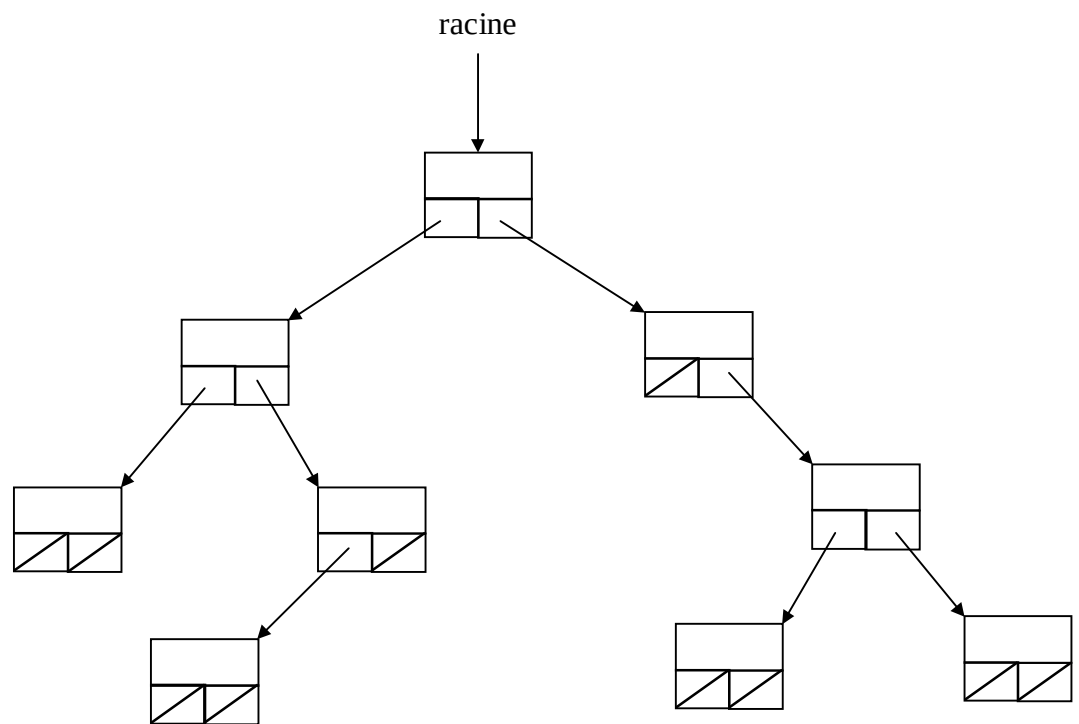
```
void afficher_recurrent (pmaillon liste)
{
    if (liste !=NULL)
    {
        printf("%d\n",liste->data);
        afficher_recurrent(liste->next);
    }
}
```

```
void afficher_inverse_recurrent (pmaillon liste)
{
    if (liste !=NULL)
    {
        afficher_inverse_recurrent(liste->next);
        printf("%d\n",liste->data);
    }
}
```

```
void inserer_recurrent (pmaillon liste, pmaillon maillon)
{
    if (liste !=NULL)
    {
        liste=maillon;
    }
    else
    {
        if(maillon->data < liste->data)
        {
            maillon->next=liste;
        }
    }
}
```

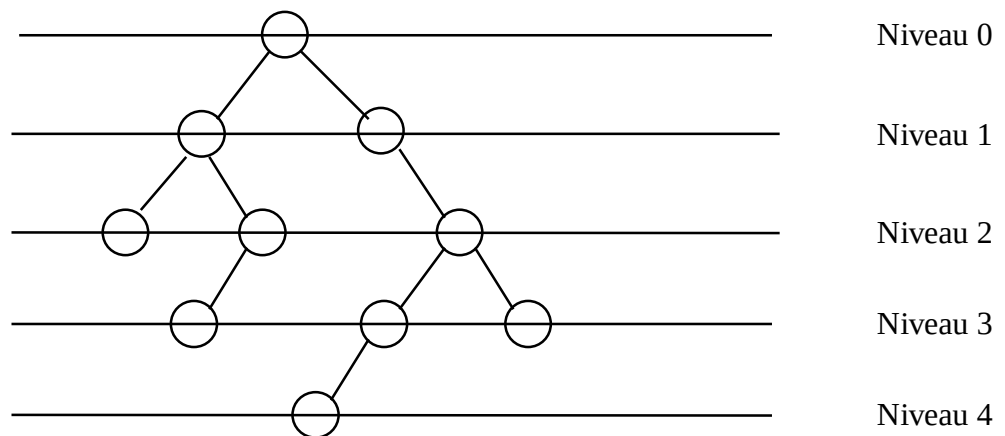
```
        liste=maillon;
    }
    else
    {
        liste->next=inserer_recurent (liste->next, maillon);
    }
}
return liste;
}
```

ARBRES BINAIRES



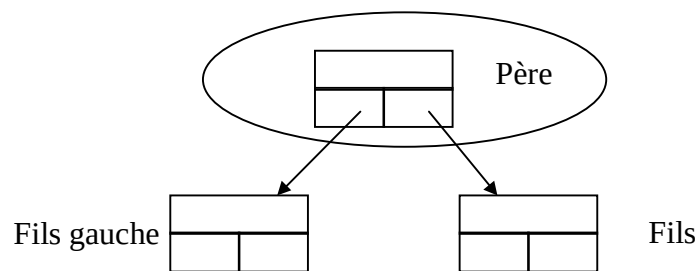
Un maillon d'un arbre est appelé nœud. La tête de l'arbre est appelé racine. Un nœud qui ne possède pas de successeur est appelé feuille ou nœud terminal. Un nœud qui n'est pas la racine et qui n'est pas une feuille peut être appelé nœud interne.

Hauteur de l'Arbre



Le noeud:

Il s'agit d'une structure réursive dont la base est:

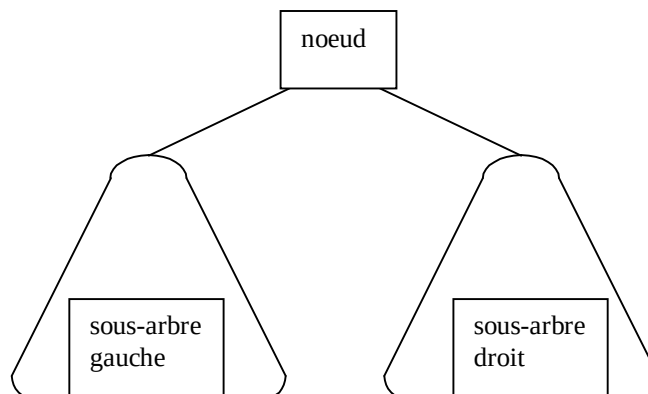


```
struct noeud {
    int data;
    struct noeud *fg;
    struct noeud *fd;
}

typedef struct noeud *pnoeud;
```

RECURENCE DES ARBRES BINAIRES

Un arbre peut être soit vide soit constitué d'un nœud qui possède un sous-arbre à gauche et un sous-arbre à droite.



Affichage: 6solutions:

NGD
NDG
GND
GDN
DNG
DGN (note: est l'inverse (negation) de NGD ...)

```

void afficher_arbre (pnoeud racine)
{
    if(racine!=NULL)
    {
        /*ordre: NGD */
        printf("\nValeur du nœud: %d",racine->data); /*nœud*/
        afficher_arbre (racine->fg); /*gauche*/
        afficher_arbre (racine->fd); /*droit*/
    }
}
    
```

algorithme généralisé du NDG:

```

si (racine <> NULL)
    <travail sur le nœud>
    <traitement récursif sur racine-> fils gauche>
    <traitement récursif sur racine-> fils droit>
sinon
    <traitement de l'arbre vide>
    
```

```

int complet (pnoeud racine)
{
    /*Un arbre est dit complet s'il existe et s'il a 0 ou 2 fils(vides ou complets)*/
    
```

```

int cd,cg;

    if(racine==NULL)
    {
        return -1;
    }
    else
    {
        cg=complet(racine->fd);
        cd=complet(racine->fg);
        if (cg * cd > 0)
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
}
    
```

```

/*Un arbre est dit équilibré s'il a le même nombre de nœuds dans le fils droit que
dans le fils gauche. */
    
```

ANNEXES

Sources:

- The C Programming Language, B.W. Kernighan/D.M. Ritchie –Prentice Hall
- Programmation en C, R. Servais/C. Georis -K TechS/E.R. Tech
- Programmation en C, M. Applaincourt/C. Catonio -Université Mons-Hainaut

Contact:

Vincent Jamart <vincent.jamart@be.linux.org>
<http://www.rtfm.be/vjamart>