ALGORITHMIQUE ET STRUCTURES DE DONNEES
PROGRAMMATION EN LANGAGE C

P L A N

1. Les notions de base du langage C
2. Les structures (enregistrements)
3. Les fonctions
4. Les pointeurs
5. L'allocation dynamique de la mémoire
   - 5.1. L'allocation dynamique des tableaux
   - 5.2. Les listes chainées
6. Les Piles et les Files
7. La récursivité
8. Les arbres

Bibliographie

1. **Le langage C** Norme ANSI, 2ème édition *de Brian W. Kernighan et Dennis M. Ritchie (2004) Editions DUNOD*
2. **Maîtrise des algorithmes en C** *de Kyle Loudon (*2005) Editions O'Reilly
3. **Algorithmes en langage C** , Robert Sedgewick, Edition : 1991, InterEditions
4. **Langage C,** Bernard Leroy, Edition : 1994, Sybex
5. **Introduction à l'algorithmique**, 2ème édition, Thomas Cormen *et al*, Dunod, 2002.
6. **Méthodologie de la programmation en C**, 4ème édition, Achille Braquelaire, Dunod, 2005.
7. **Langage C**, Claude Delannoy, Eyrolles, 2002.

Un peu d'histoire

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre The C Programming language [6]. Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990.

Les dangers de C

C est un langage près de la machine, donc dangereux et bien que C soit un langage de programmation structuré, il ne nous force pas à adopter un certain style de programmation (comme p.ex. Pascal). Dans un certain sens, tout est permis même la commande « goto », si redoutée par les puristes ne manque pas en C. Le programmeur a donc beaucoup de libertés, mais aussi des responsabilités: il doit veiller lui-même à adopter un style de programmation propre, solide et compréhensible.

Qualité du programme

Les caractéristiques qu'un bon programmeur doit rechercher dans l'établissement de ses programmes sont :

- Clarté : le programme doit être lisible dans sa globalité par n'importe quel autre programmeur. Sa logique doit être facilement appréhendée. L'un des principaux objectifs du C était précisément de permettre le développement de programmes ordonnés, clairs et lisibles.
- Simplicité : toujours donner la préférence aux solutions simples qui renforcent la précision et la clarté du programme. Compromis entre niveau de précision et clarté du programme.
- Efficacité : vitesse d'exécution et rationalisation de l'utilisation de la mémoire.
- Modularité : de nombreux programmes se prêtent bien à un découpage en sous-tâches clairement identifiées. Programmer ces sous-tâches dans des modules distincts. En C, ces modules sont matérialisés par des fonctions. La modularité rend un programme plus précis et plus clair ; sa maintenance est facilitée (le fait de modifier ou de faire évoluer un programme).
- Extensibilité : permet une meilleure réutilisation des fonctions.

1. Les notions de base du langage C

Les programmes en C sont composés essentiellement de fonctions et de variables.

1.1 La fonction *main*

La fonction **main** est la fonction principale des programmes en C. Elle se trouve obligatoirement dans tous les programmes. L'exécution d'un programme entraîne automatiquement l'appel de la fonction **main**.

```
main()
{
   <déclarations>
   <instructions>
}
```

1.2 Structure générale d'un programme C

```
inclure des bibliothèques (#include)
définir des constantes (#define)
définir des types
définir des fonctions
main()
{   <déclaration des variables locales du programme principal>
   <instructions>
}
```

Un programme C est un ensemble de fonctions. Tout programme comporte au moins une fonction principale désignée par main.

Exemple :

```
#include<stdio.h>
main()
{  /*  ce programme affiche le message « Bonjour » */
    printf("Bonjour") ;
}
```

- main : indique qu'il s'agit d'un programme principal
- les ( ) indiquent que la fonction main n'a pas de paramètres
- /* */ permettent d'encadrer un commentaire
- printf est une fonction qui est définie dans un fichier particulier appelé bibliothèque (ou librairie) d'E/S du langage C
- #include<stdio.h> pour que printf soit reconnue l'ors de la compilation

a) Les identificateurs

Les identificateurs sont les noms propres du programme.

Les noms des fonctions et des variables en C sont composés d'une suite de lettres et de chiffres. Le premier caractère doit être une lettre. Le symbole '_' est aussi considéré comme une lettre.

* L'ensemble des symboles utilisables est donc: {0,1,2,...,9,A,B,...,Z,_,a,b,...,z}
* Le premier caractère doit être une lettre (ou le symbole '_')
* **C distingue les majuscules et les minuscules, ainsi: *NOM* est différent de *nom***
* La longueur des identificateurs n'est pas limitée, mais C distingue seulement les 31 premiers caractères.
* **Les mots clés du C doivent être écrits en minuscules**

b) Les commentaires

Un commentaire commence toujours par les deux symboles '**/***' et se termine par les symboles '***/**'. Il est interdit d'utiliser des commentaires imbriqués.

Exemples : **/* Ceci est un commentaire correct */**
/* Ceci est /* évidemment */ défendu */

c) Utilisation des bibliothèques de fonctions

Pour pouvoir les utiliser, il faut inclure des fichiers en-tête (*header files* - extension **.H**) dans nos programmes.

L'instruction ***#include*** insère les fichiers en-tête indiqués comme arguments dans le texte du programme au moment de la compilation.

Exemple :

Nous avons écrit un programme qui fait appel à des fonctions mathématiques prédéfinies. Nous devons donc inclure le fichier en-tête correspondant dans le code source de notre programme à l'aide de l'instruction: **#include <math.h>**

1.3 Variables, types de base, déclarations

a) Les entiers

définition	*description*	*domaine min*	*domaine max*	*nombre d'octets*
short	entier court	-32768	32767	2
int	entier standard	-2147483648	2147483647	4
long	entier long	-2147483648	2147483647	4
unsigned short	entier court	0	65535	2
unsigned int	entier standard	0	65535	2
unsigned long	entier long	0	4294967295	4

sizeof(short)<=sizeof(int)<=sizeof(long)

Dans certaines machines on a short →2 int →2 long→4

dans d'autres on a short →2 int →4 long→4

b) Les réels

En C, nous avons le choix entre deux types de réels: **float**, **double**

définition	*précision*	*mantisse*	*domaine min*	*domaine max*	*nombre d'octets*
float	simple	6	$3.4 * 10^{-38}$	$3.4 * 10^{38}$	4
double	double	15	$1.7 * 10^{-308}$	$1.7 * 10^{308}$	8

La ***mantisse*** : Les chiffres significatifs du réel sans la virgule

Exemple : 123.4 ou 1234e -1 mantisse→1234

c) Les caractères

Le type **char** *(provenant de l'anglais character)* permet de stocker la valeur ASCII d'un caractère, c'est-à-dire un nombre entier !

Par défaut les nombres sont signés, cela signifie qu'ils comportent un signe. Pour stocker l'information concernant le signe (en binaire), les ordinateurs utilisent le « complément à deux ». Une donnée de type char est donc signée, cela signifie bien sûr pas que la lettre possède un signe mais tout simplement que dans la mémoire la valeur codant le caractère peut être négative.

définition	description	domaine min	domaine max	nombre d'octets
char	Caractère	-128	127	1
unsigned char	Caractère	0	255	1

d) La définition des constantes :

Une constante est un objet auquel on attribue une valeur à la déclaration et que l'on ne peut pas changer tout au long du programme. On défini des constantes en utilisant la directive **define** ou le mot clé **const**.

Syntaxe : #define <idf> valeur ou bien const <idf> = valeur

Exemple : #define max 100 ou bien const max=100

Remarque :

Une constante de type caractère est placée entre deux apostrophes : 'd' ou 'D' ... comme suit #define C 'F' et chaine de caractères ainsi #define end "Fin"

e) La déclaration des variables :

<Type> <NomVar1>, <NomVar2>,..., <NomVarN>;

Exemples :

int compteur,X,Y; float hauteur,largeur; double M; char C;

f) Initialisation des variables

En C, il est possible d'initialiser les variables lors de leur déclaration:

int MAX = 1023; float X = 1.05;

Remarque : En C il n'existe pas de type spécial pour les variables **booléennes**. Si l'utilisation d'une variable booléenne est indispensable, on utilisera une variable du type **int**. Les opérations logiques en C retournent toujours des résultats du type **int** : **0** pour faux et **1** pour vrai

g) Convertions de type automatique

En général on converti des types plus « petits » en des types plus « larges » de cette façon on ne perd pas en précision.

Lors de l'affectation, la donnée à droite du signe d'égalité est convertie dans le type à gauche du signe d'égalité. Dans ce cas il peut y avoir une perte de précision si le type de la destination est plus faible que celui de la source.

Exemple : int i=8 ; float x=12.5 ; double y ;

y = i * x ; - i → est converti en float
- i*x → est converti en double
- affecter le résultat à y

1.4 Les opérateurs standard

Affectation : **< Nom Variable > = < Expression > ;**

Opérateurs arithmétiques

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

Opérateurs logiques

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

Opérateurs de comparaison

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

Remarque : L'opérateur **/** effectue une division entière quand les deux opérandes sont des entiers, en effet 5/2 donne comme résultat 2. Si l'on désire obtenir un résultat réel, on va forcer le type en faisant se que l'on appelle un ***« cast »*** comme suit : (float)5/2 donnera comme résultat 2.5. Et bien sur si l'un des deux opérandes est un réel (ou les deux) le résultat sera un réel (5.0/2 → 2.5).

Opérateurs d'affectation

+=	ajouter à
-=	diminuer de
***=**	multiplier par
/=	diviser par
%=	modulo
X = i++	passe d'abord la valeur de i à X et ***incrémente après***
X = i--	passe d'abord la valeur de i à X et ***décrémente après***
X = ++i	***incrémente d'abord*** et passe la valeur incrémentée à X
X = --i	***décrémente d'abord*** et passe la valeur décrémentée à X

Pour la plupart des expressions de la forme: **expr1 = (expr1) op (expr2)**
Il existe une formulation équivalente: **expr1 op= expr2**
L'affectation **i = i + 2** peut s'écrire **i + = 2**

Les priorités des opérateurs

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

Exemple récapitulatif :

```
main()
{ int i, j=2 ;  float x=2.5 ;
  i=j+x ;
  x=x+i ;    /*  ici x=6.5 (et non 7) car dans i=j+x  j+x a été convertie en int */
   …
}
main()
{ float x=3/2 ;  /* ici x=1  division entière */
  x=3/2.    /* ici x=1.5   resultat réel */
  …
int a=3, b, c ;
b=++a ;  /*  a=4 , b=4 car on a a=a+1 ; b=a  */
c=b++; /*  c=4, b=5  car on a c=b; b=b+1;  */
…
int a=3, b=4;  float c;
c=a/b;   /* c=0  division entière  */
c=(float)a/b ;  /* c=0.75  */
…
float a=3, b=4, c ;    c=a/b;  /*  c=0.75  */
c=(int)a/b;  /*  c=0  */   …   }
```

1.5 Tableaux et chaînes de caractères

1.5.1 Les tableaux à une dimension

a. Déclaration:

<TypeSimple> <NomTableau> [<Dimension>] ;

Exemples ; int A[25] ; long B[25] ; float F[100] ; double D[100] ; char ch[30] ;

b. Initialisation :

Exemples : int A[5] = {10, 20, 30, 40, 50};
float B[4] = {-1.05, 3.33, 87e-5, -12.3E4};
int C[10] = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};

Remarque :

- Il faut évidemment veiller à ce que le nombre de valeurs dans la liste corresponde à la dimension du tableau. Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées par zéro.
- Si la dimension n'est pas indiquée explicitement lors de l'initialisation, alors l'ordinateur ***réserve automatiquement*** le nombre d'octets nécessaires.
  Exemple :
  int A[ ] = {10, 20, 30, 40, 50}; ==> Réservation de 5*taille d'un entier
  (Si un entier occupe 2 octets donc pour ce tableau seront réservés10 octets)

c. Accès aux composantes

En déclarant un tableau par: **int A[5];** Nous avons défini un tableau **A** avec **5** composantes, auxquelles on peut accéder par: **A[0], A[1], ... , A[4]**

1.5.2 Les tableaux à deux dimensions

a. Déclaration :

<TypeSimple> <NomTabl> [<DimLigne>] [<DimCol>] ;

Exemples : int A[10][10] ; long B[10][10] ; float F[10][10] ; double D[10][10] ;
char ch[15][30] ;

b. Mémorisation :

Les composantes d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire.

c. Initialisation :

Exemples

int A[3][10] ={ { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90},
{10, 11, 12, 13, 14, 15, 16, 17, 18,19},
{1, 12, 23, 34, 45, 56, 67, 78, 89, 90} };

d. Accès aux composantes :

<NomTableau> [<Ligne>] [<Colonne>] ;

En déclarant un tableau par: **int M[5][3];** Nous avons défini un tableau (matrice) **M** avec **5** lignes et chaque ligne contient **3** composantes, auxquelles on peut accéder par: **M[0][0], M[0][1], ... , M[4][0],…,M[4][2]**

1.5.3 Les Chaînes de caractères :

Il n'existe pas de type spécial *chaîne* ou *string* en C. Une chaîne de caractères est traitée comme un *tableau à une dimension de caractères* (vecteur de caractères).

a. Déclaration :

char <NomVariable> [<Longueur>] ;

Exemples : char NOM [20]; char PRENOM [20]; char PHRASE [300];

Remarque : Lors de la déclaration, nous devons indiquer l'espace à réserver en mémoire pour le stockage de la chaîne. La représentation interne d'une chaîne de caractères est terminée par le symbole **'\0'** (NUL). Ainsi, pour un texte de **n** caractères, nous devons prévoir **n+1** octets.

***b.* Les chaînes de caractères constantes**

Les chaînes de caractères constantes sont indiquées entre guillemets. La chaîne de caractères vide est alors: **""**

"x" est un **tableau de caractères** qui contient deux caractères:
la lettre 'x' et le caractère NUL: '**\0**' est codé dans deux octets

'x' est un **caractère** et est codé dans un octet

***c.* Initialisation de chaînes de caractères :**

char CHAINE [ ] = "Salam";

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** pour la marque de fin de chaines (ici: 6 octets). Nous pouvons aussi indiquer explicitement le nombre d'octets à réserver, si celui-ci est supérieur ou égal à la longueur de la chaîne d'initialisation.

1.5.4 Tableaux de chaînes de caractères

Un tableau de chaînes de caractères correspond à un tableau à deux dimensions du type **char**, où ***chaque ligne contient une chaîne de caractères***.

a. ***Déclaration*** : La déclaration **char JOUR[7][9];** Réserve l'espace en mémoire pour 7 mots contenant 9 caractères (dont 8 caractères significatifs).

b. **Initialisation : char JOUR[7][9] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"};**

c. ***Accès aux chaînes*** : Il est possible d'accéder aux différentes ***chaînes de caractères*** d'un tableau, en indiquant simplement la ligne correspondante : JOUR[3] → "mercredi"

Exemple récapitulatif : Initialisation de chaîne de caractère :
Lesquelles des chaînes suivantes sont initialisées correctement ?
Corrigez les déclarations fausses.

a) char a[ ] = "un\ndeux\ntrois\n" ;
b) char b[12] = "un deux trois" ;
c) char c[] = 'abcdefg' ;
d) char d[10] = 'x' ;
e) char e[5] = "cinq" ;
f) char f[] = "Cette " "phrase" "est coupée";
g) char g[2] = { 'a', '\0' };
h) char h[4] = { 'a', 'b', 'c' } ;

Solution

a) char a[] = "un\ndeux\ntrois\n";
Déclaration correcte
b) char b[12] = "un deux trois";
Déclaration incorrecte, la chaîne d'initialisation dépasse le bloc de mémoire réservé.
Correction: char b[14] = "un deux trois"; ou mieux: char b[] = "un deux trois";
c) char c[] = 'abcdefg';
Déclaration incorrecte: Les symboles ' et ' encadrent des caractères;
Pour initialiser avec une chaîne de caractères, il faut utiliser les guillemets (ou indiquer une liste de caractères).
Correction: char c[] = "abcdefg";

d) char d[10] = 'x';
Déclaration incorrecte: Il faut utiliser une liste de caractères ou une chaîne pour l'initialisation. Correction: char d[10] = {'x', '\0'} ou mieux: char d[10] = "x";

e) char e[5] = "cinq"; Déclaration correcte ;

f) char f[] = "Cette ", "phrase", "est coupée";
Déclaration incorrecte ; On ne peut affecter plusieurs chaînes séparées ainsi.

g) char g[2] = {'a', '\0'}; Déclaration correcte ;

h) char h[4] = {'a', 'b', 'c'};
Déclaration incorrecte: Dans une liste de caractères, il faut aussi indiquer le symbole de fin de chaîne. Correction: char h[4] = {'a', 'b', 'c', '\0'};

1.6 Actions élémentaires et structures de contrôle

1.6.1 Lire et écrire des données

La bibliothèque standard <*stdio*> contient un ensemble de fonctions qui assurent la communication de la machine avec le monde extérieur telles que : printf (écrire) et scanf (lire). Donc tout programme qui utilise ces fonctions doit être précédé de la directive : ***#include<stdio.h>***

a) printf( ) : ***printf ("<format>",<Expr1>,<Expr2>, ...) ;***

La partie "***<format>"*** est une chaîne de caractères qui peut contenir du texte, des séquences d'échappement, des spécificateurs de format.

Les ***spécificateurs de format*** indiquent la manière dont les valeurs des expressions sont affichées et commencent toujours par % et se terminent par un ou deux caractères. La partie "<format>" contient ***exactement un*** spécificateur de format pour chaque expression.

- Spécificateurs de format pour printf

%d ou %i	int	entier relatif
%u	int	entier naturel (unsigned)
%c	int	caractère
%l	long	entier long
%f	double	rationnel en notation décimale
%e	double	rationnel en notation scientifique
%s	char*	chaîne de caractères

Exemples :

(1) La suite d'instructions: int A = 12; int B = 5;
printf("%i fois %iest %i\n", A, B, A*B);
Va afficher sur l'écran: 12 fois 5 est 60

(2) char JOUR[7][9]= {"dimanche","lundi", "mardi", "mercredi", "jeudi",
"vendredi", "samedi"};
int i = 4; printf("Aujourd'hui, c'est %s !\n", JOUR[i]);
Affichera la phrase : Aujourd'hui, c'est jeudi !

- Les séquences d'échappement

\a	sonnerie	****	trait oblique
\b	curseur arrière	**\?**	point d'interrogation
\t	tabulation	**\'**	apostrophe
\n	nouvelle ligne	**\"**	guillemets
\r	retour au début de ligne	**\f**	saut de page (imprimante)
\0	NUL	**\v**	tabulateur vertical

b) scanf() : ***scanf("<format>",<AdrVar1>,<AdrVar2>, ...) ;***

La chaîne de format détermine comment les données lues doivent être interprétées.

Les données lues correctement sont mémorisées successivement aux ***adresses* <AdrVar1>,...**

L'adresse d'une variable est indiquée par le nom de la variable précédé du signe **&**.

***Exemples* :**

(1) int Jour, Mois, Annee;

scanf("%i %i %i", &Jour, &Mois, &Annee); →Lit trois entiers relatifs et les valeurs sont attribuées respectivement aux trois variables **Jour**, **Mois** et **Annee**.

(2)
```
#inlude<stdio.h>
main()
{  int a, b, res ;  printf(" Donner 2 valeurs entières\n ") ;
   scanf("%d %d",&a,&b);    res=a*b ;
   printf("%d  fois  %d = %d",a,b,res);    }
```

1.6.2 Les fonctions puts et gets

Comme nous l'avons déjà vu, la bibliothèque <*stdio*> nous offre des fonctions qui effectuent l'entrée et la sortie des données. A côté des fonctions printf et scanf que nous connaissons déjà, nous y trouvons les deux fonctions puts et gets, spécialement conçues pour l'écriture et la lecture.

- **puts** :

puts est idéale pour écrire une chaîne constante ou le contenu d'une variable.

Syntaxe: **puts(<Chaîne>)**

puts écrit la chaîne de caractères désignée par <Chaîne> sur *l'écran* et provoque un retour à la ligne.

puts(TXT); est équivalent à **printf("%s\n",TXT);**

Exemples

```
char TEXTE[ ] = "Voici une première ligne.";
puts(TEXTE);
puts("Voici une deuxième ligne.");
```

- **gets**

gets est idéal pour lire une ou plusieurs lignes de texte (p.ex. des phrases) terminées par un retour à la ligne.

Syntaxe: **gets(<Chaîne>)**

gets lit une ***ligne*** de caractères et la copie à l'adresse indiquée par <Chaîne>. Le retour à la ligne final est remplacé par le symbole de fin de chaîne **'\0'**.

Exemple :

```
int MAXI = 1000;
char LIGNE[MAXI];
gets(LIGNE);
```

Important : scanf avec le spécificateur **%s** permet de lire ***un seul mot***.

Exemples :

```
char LIEU[25];
int JOUR, MOIS, ANNEE;
printf("Entrez lieu et date de naissance : \n");
scanf("%s %d %d %d", LIEU, &JOUR, &MOIS, &ANNEE);
```

Remarque :

La fonction **scanf** a besoin des ***adresses de ses arguments:*** Les noms des variables numériques (**int**, **char**, **long**, **float**, ...) doivent être marqués par le symbole **'&',** Comme le nom d'une chaîne de caractères est le représentant de l'adresse du premier caractère de la chaîne*, **il ne doit pas être précédé de l'opérateur adresse '&' !***

1.6.3 Les structures de contrôle :

1.6.3.1 L'instruction conditionnelle IF

a) if - else

```
if ( <expression> )
        <bloc d'instructions 1> ;
   else
        <bloc d'instructions 2> ;
```

Exemple

```
if (a > b)
  max = a;
else
  max = b;
```

b) if sans else

```
if ( <expression> )
                <bloc d'instructions> ;
```

Exemple :

```
i) if (N>0)
     if (A>B) MAX=A;
     else   MAX=B;
```

(ou bien)

```
ii) if (N>0)
        if (A>B)  MAX=A;
    else   MAX=B;
```

Pour N=0, A=1 et B=2,

* dans la première interprétation (i), MAX reste inchangé,

* dans la deuxième interprétation (ii), MAX obtiendrait la valeur de B.

En C le « else » est toujours lié au dernier if. Pour éviter des confusions, il est recommandé d'utiliser des accolades **{ }**.

Exemple

Pour forcer la deuxième interprétation de l'expression ci-dessus, nous pouvons écrire:

```
if (N>0)
   {  if (A>B)     MAX=A;   }
 else   MAX=B;
```

c) if - else if - ... - else

```
if ( <expr1> )     <bloc1>;
   else if (<expr2>)   <bloc2>;
        else if (<expr3>)    <bloc3>;
             else if (<exprN>)    <blocN>;
                  else <blocN+1> ;
```

Les opérateurs conditionnels : **expr1> ? <expr2> : <expr3> ;**

* Si <expr1> fournit une valeur différente de zéro, alors la valeur de <expr2> est le résultat

* Si <expr1> fournit la valeur zéro, alors la valeur de <expr3> est le résultat

Exemple : if (A>B) MAX=A; else MAX=B;

Peut être remplacée par: MAX = (A > B) ? A : B;

1.6.3.2 L'instruction switch

```
switch (expression)
    { case constante-1 : <instruction-1>;break;
      case constante-2 : <instruction-2>;break;
        …
      case constante-n : <instruction-n>;break;
      default: <instruction n+1>;
    }
```

Remarque : **break** permet de sortir du switch

Exemple : switch (operateur)

```
{ case '+'   :  s=a+b ; break ;
  case '-'   :  s=a-b;  break;
  case '*'   :  s=a*b;  break;
  case '/'   :  if (b!=0) s=a/b;
                else printf("division par zero");
                break;
  default    : printf("erreur");
}
```

1.6.3.3 Les instructions iterative : while, do-while, for

- while:

```
while ( <expression> )
    {  <bloc d'instructions> ;  }
```

Exemple

```
/* Afficher les nombres de 0 à 9 */
int i = 0;
while (i<10)
    {   printf("%d \n", i);   i++;  }
```

- do – while

La structure **do - while** est semblable à la structure **while**, avec la différence suivante :

- **while** évalue la condition ***avant*** d'exécuter le bloc d'instructions,
- **do - while** évalue la condition ***après*** avoir exécuté le bloc d'instructions. Ainsi le bloc d'instructions est exécuté au moins une fois.

```
do
    {  <bloc d'instructions>;  }
while ( <expression> );
```

Exemple: lire un ensemble de caractères jusqu'à la rencontre d'un point et compter le nombre de 'e' et 'E'.

```
 #include<stdio.h>
main()
{ char car; int  cpt=0;
do
  {  scanf("%c", &car);
     if (car=='e' || car=='E') cpt++ ;
  }
while (car !='.');
printf("Le nombre de 'e' et 'E' = %d ",cpt);
}
```

- for :

```
for ( <expr1> ; <expr2> ; <expr3> )
      {   <bloc d'instructions>;  }
```

<expr1> est évaluée une fois avant le passage de la boucle. Elle est utilisée pour initialiser les données de la boucle.

<expr2> est évaluée avant chaque passage de la boucle. Elle est utilisée pour décider si la boucle est répétée ou non.

<expr3> est évaluée à la fin de chaque passage de la boucle. Elle est utilisée pour réinitialiser les données de la boucle.

Exemples **:**

(1) int i; / Calcul et affichage du carré des 20 premiers nombres */*
for (i=1 ; i<=20 ; i++)
*printf("Le carré de %d est %d \n", i, i*i);*

(2) /* Affichage et Lecture des éléments d'un tableau à une dimension */

```
main()
{ int A[5];
  int i;  /* Compteur */
  for (i=0; i<5; i++)
      scanf("%d", &A[i]);
  for (i=0; i<5; i++)
     printf("%d ", A[i]); ou bien  printf("%d\t", A[i]); /* tabulateur */
}
```

(3) /* Affichage des éléments d'un tableau à deux dimensions (Matrice) ***/***

```
main()
{ long  A[10][20];  int i,j;
 /* Pour chaque ligne ... */
 for (i=0; i<10; i++)
   { for (j=0; j<20; j++)     /* considérer chaque composante*/
            printf("%d", A[i][j]);
     printf("\n");  /* Retour à la ligne */
   }
}
```

1.6.4 Quelques bibliothèques de fonctions :

a. Les fonctions de <math.h>

exp(X)	fonction exponentielle
log(X)	logarithme naturel
log_{10}(X)	logarithme à base 10
pow(X,Y)	X exposant Y
sqrt(X)	racine carrée de X

fabs(X)	valeur absolue de X
floor(X)	arrondir en moins
ceil(X)	arrondir en plus
fmod(X,Y)	reste rationnel de X/Y (même signe que X)

sin(X) cos(X) tan(X)	sinus, cosinus, tangente de X
asin(X) acos(X) atan(X)	arcsin(X), arccos(X), arctan(X)
sinh(X) cosh(X) tanh(X)	sinus, cosinus, tangente hyperboliques de X

b. Les fonctions de <string.h>

Dans le tableau suivant, <n> représente un nombre du type **int**. Les symboles <s> et <t> peuvent être remplacés par :

- une chaîne de caractères constante
- le nom d'une variable déclarée comme tableau de **char**
- un pointeur sur **char**

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> = 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>
Strchr(<s>,<c>)	la recherche d'un caractère c dans une chaine s et retourne son adresse dans s
Strstr(<s1>,<s2>)	la recherche d'une sous-chaine s2 dans la chaine S1 et retourne l'adresse de s2 dans s1

c. Les fonctions de <stdlib.h>

La bibliothèque <*stdlib*> contient des déclarations de fonctions pour la conversion de nombres en chaînes de caractères et vice-versa.

Conversion de chaînes de caractères en nombres

atoi(<s>) retourne la valeur numérique représentée par <s> comme **int**

atol(<s>) retourne la valeur numérique représentée par <s> comme **long**

atof(<s>) retourne la valeur numérique représentée par <s> comme **double** (!)

Règles générales pour la conversion:

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Conversion de nombres en chaînes de caractères

itoa (<n_int>, <s>, <b>)

ltoa (<n_long>, <s>, <b>)

ultoa (<n_uns_long>, <s>, <b>)

d. Les fonctions de <ctype>

Les fonctions de ***classification*** suivantes fournissent un résultat du type **int** différent de zéro, si la condition respective est remplie, sinon zéro.

La fonction:	retourne une valeur différente de zéro,
isupper(<c>)	si <c> est une majuscule **('A'...'Z')**
islower(<c>)	si <c> est une minuscule **('a'...'z')**
isdigit(<c>)	si <c> est un chiffre décimal **('0'...'9')**
isalpha(<c>)	si **islower(<c>)** ou **isupper(<c>)**
isalnum(<c>)	si **isalpha(<c>)** ou **isdigit(<c>)**
isxdigit(<c>)	si <c> est un chiffre hexadécimal (**'0'...'9'** ou **'A'...'F'** ou **'a'...'f'**)
isspace(<c>)	si <c> est un signe d'espacement (**' '**, **'\t'**, **'\n'**, **'\r'**, **'\f'**)

Les fonctions de ***conversion*** suivantes fournissent une valeur du type **int** qui peut être représentée comme caractère; la valeur originale de <c> reste inchangée:

tolower(<c>) retourne <c> converti en minuscule si <c> est une majuscule
toupper(<c>) retourne <c> converti en majuscule si <c> est une minuscule

Chapitre 2 : Définition de types et de structures

2.1 Définition de types

La définition de types doit se faire à l'extérieure de toutes les fonctions.

Pour alléger l'écriture des programmes on peut affecter un nouvel identificateur à un type composé à l'aide de l'instruction **typedef :**

```
typedef   <type>   <définition> ;
```

Exemple:

```
# define max 100
typedef  int tab[max];
typedef  char chaine[20];
…
main()
{  tab T1, T2;
   chaine  s1, s2 ;
…
}
```

L'instruction ***typedef*** est utilisée tout particulièrement avec les structures présentées dans la section suivante.

2.2 Les structures

Une structure (ou enregistrement) permet de regrouper plusieurs variables de types différents (appelées champs) et de leur donner un nom.

a. Déclaration de structures

Syntaxe :

```
typedef struct   {  type-1 champ-1 ;
                    type-2 champ-2 ;
                          …
                    type-n champ-n
                 }  <nom de la structure>;
```

***Exemple* :**

```
typedef struct
{  char  Nom[20], Prenom[20];
   int  Age;
   float Taille;
} personne ;
```

***Remarques** :*

- Une telle déclaration définit un modèle d'objet. Elle n'engendre pas de réservation mémoire.
- Dans une structure, tous les noms de champs doivent être distincts. Par contre rien n'empêche d'avoir 2 structures avec des noms de champs en commun, l'ambiguïté sera levée par la présence du nom de la structure concernée.

b. Accès à un champ

Syntaxe : **<ident_objet_struct> . <ident_champ>**

L'opérateur d'accès est le symbole **"."** (Point) placé entre l'identificateur de la structure et l'identificateur du champ désigné.

Exemple :

```
main( )
{  personne P ;
 ... P.Age = 45;...
}
```

c. Utilisation des structures

exemple :

```
typedef struct
{       char nom[20], prenom[20];
        int age;
        float note;
} fiche;
```

On déclare des variables par exemple :

```
fiche f1,f2;
strcpy(f1.nom,"Badi");
strcpy(f1.prenom,"Ali");
f1.age = 20;   f1.note = 11.5;
```

Remarque :

- L'affectation globale est possible avec les structures, on peut écrire: **f2 = f1;**
- Par contre on ne peut pas comparer deux structures (il faut comparer champ par champ)

d. Structurer les données

Exemple :

```
typedef struct   { int Jour,Mois,Annee; } Date;
typedef struct  {  char Nom[20], Adresse[30];
                    Date Naissance;
                  }; personne ;
personne P ;
```

On peut alors écrire : if (P.Naissance.Jour = = 20) ...

e. Tableaux de structures

Exemple :

```
typedef struct   { int Jour,Mois,Annee; } Date;

typedef struct  {  char Nom[20], Adresse[30];
                    Date Naissance;
                  }; personne ;
personne T[20] ;
```

Chapitre 3 : Les fonctions

3.1 Définition et déclaration de fonctions

a) Définition d'une fonction en C

```
<TypeRés> <NomFonct> (<TypePar1><NomPar1>, <TypePar2> <NomPar2>, ... )
  {
   <déclarations locales>
   <instructions>
  }
```

Attention ! *Si nous choisissons un nom de fonction qui existe déjà dans une bibliothèque, notre fonction* ***cache*** *la fonction prédéfinie.*

b) Important

- Une fonction peut fournir comme résultat:
  - un type arithmétique,
  - une structure (définie par **struct**),
  - un pointeur, (sera défini au chapitre suivant)
  - **void** (la fonction correspond alors à une "***procédure***"). Si une fonction ne fournit pas de résultat, il faut indiquer **void** comme type du résultat. En C, il n'existe pas de structure spéciale pour la définition de *procédures* comme en Pascal et en langage algorithmique.
- Si une fonction n'a pas de paramètres, on peut déclarer la liste des paramètres comme **(void)** ou simplement comme **()**.
- Une fonction **ne peut pas** fournir comme résultat des tableaux, des chaînes de caractères ou des fonctions. (***Attention:*** Il est cependant possible de renvoyer un pointeur sur le premier élément d'un tableau ou d'une chaîne de caractères.)
- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).

3.2 Renvoyer un résultat

Par définition, toutes les fonctions fournissent un résultat d'un type que nous devons déclarer. Une fonction peut renvoyer une valeur d'un type simple ou l'adresse d'une variable ou d'un tableau en utilisant la commande **return.**

L'instruction **return <expression>;** a les effets suivants:

- évaluation de l'<expression>
- conversion automatique du résultat de l'expression dans le type de la fonction
- renvoi du résultat
- terminaison de la fonction

Exemple :

```
int   somme ( )                    /* ou bien  */  int somme( )
 {  int a=1276, b=498, s ;                          { int a=1276, b=498;
    s=a+b ;                                              return( a+b);  }
    return(s) ;  }
```

Remarque :

Si nous quittons une fonction (d'un type différent de **void**) sans renvoyer de résultat à l'aide de **return**, la valeur transmise à la fonction appelante est indéfinie. Le résultat d'une telle action est imprévisible.

3.3 L'appel d'une fonction :

- La fonction ne retourne pas de résultat par son nom (procédure)

Syntaxe : ***nom_fonction (var1, var2,…) ;***

(var1, var2, …) : représente la liste des paramètres ***effectifs*** (paramètres d'appels) qui doivent correspondre en type, en nombre et dans l'ordre à la liste des paramètres ***formels***.

Exemple :

```
#include<stdio.h>
somme()
{ int a=1276, b=498,s ;
   s=a+b;  printf("La somme=%d",s) ;
}
main()
{   …  somme();  …  }
```

- La fonction retourne un résultat dans son nom, elle peut être affectée à une variable de même type :

Syntaxe: ***x = nom_fonction (var1, var2, …);***

Exemple :

```
int   somme ()
 {  int a=1276, b=498, s ;
    s=a+b;
    return(s);
 }
main()
 {
    printf("somme= %d", somme());
 }
```

```
int   somme ( )  /* ou bien  */
 {  int a=1276, b=498 ;
     return(a+b) ;
 }

 main()
 { int res ;
   res=somme() ;
   printf("somme= %d", res);
 }
```

3.4 Variables locales

Les variables déclarées dans un bloc d'instructions sont *uniquement visibles à l'intérieur de ce bloc*. On dit que ce sont des ***variables locales*** à ce bloc.

Exemple

La déclaration de la variable i se trouve à l'intérieur d'un bloc d'instructions conditionnel. Elle n'est pas visible à l'extérieur de ce bloc, ni même dans la fonction qui l'entoure.

```
…
if   (N>0)  {   int i;   for (i=0; i<N; i++)     ...  }
```

3.5 Variables globales

Les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions *sont disponibles à toutes les fonctions du programme*. Ce sont alors des ***variables globales***. En général, les variables globales sont déclarées immédiatement derrière les instructions **#include** au début du programme.

Exemple

La variable S est déclarée globalement pour pouvoir être utilisée dans les procédures A et B.

```
#include <stdio.h>
int S;
void A(...)
{  ... if (S>0)   S--;
      else     ...
}
void B(...)
{ ... S++;  ...  }
```

3.6 Paramètres d'une fonction

Les paramètres ou arguments sont les 'boîtes aux lettres' d'une fonction. Elles acceptent les données de l'extérieur et déterminent les actions et le résultat de la fonction. Techniquement, nous pouvons résumer le rôle des paramètres en C de la façon suivante:

Les paramètres d'une fonction sont simplement des variables locales qui sont initialisées par les valeurs obtenues lors de l'appel.

Lors d'un appel, le nombre et l'ordre des paramètres doivent nécessairement correspondre aux indications de la déclaration de la fonction. Les paramètres sont **automatiquement convertis** dans les types de la déclaration avant d'être passés à la fonction.

Exemple
Le prototype (voir §3.7) de la fonction **pow** (bibliothèque <*math*>) est déclaré comme suit: **double pow (double, double);**
Au cours des instructions, **int A, B; ... A = pow (B, 2);**
Nous assistons à trois conversions automatiques: avant d'être transmis à la fonction, la valeur de B est convertie en **double**; la valeur 2 est convertie en 2.0 . Comme **pow** est du type **double**, le résultat de la fonction doit être converti en **int** avant d'être affecté à A.

Evidemment, il existe aussi des fonctions qui fournissent leurs résultats ou exécutent une action sans avoir besoin de données. La liste des paramètres contient alors la déclaration **void** ou elle reste vide (exp.: **double PI(void)** ou **double PI( )**).

3.6.1 Passage des paramètres par valeur

En C, le passage des paramètres se fait toujours par la valeur, c.-à-d. les fonctions n'obtiennent que les ***valeurs*** de leurs paramètres et n'ont pas d'accès aux variables elles-mêmes.
Les paramètres d'une fonction sont à considérer comme des ***variables locales*** qui sont initialisées automatiquement par les valeurs indiquées lors d'un appel.
A l'intérieur de la fonction, nous pouvons donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes.

Exemple

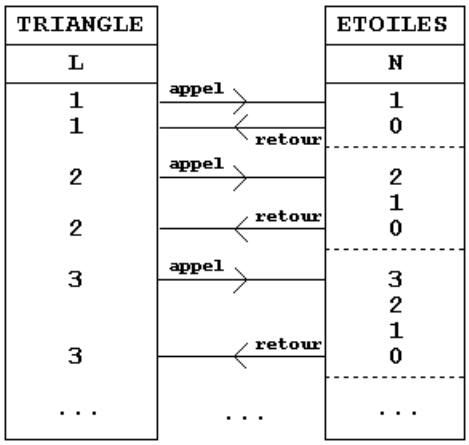
La fonction ETOILES dessine une ligne de N étoiles. Le paramètre N est modifié à l'intérieur de la fonction:

```
 void ETOILES(int N)
{ while (N>0) {   printf("*");   N--;  }
  printf("\n");
}
```

La fonction TRIANGLE, appelle la fonction ETOILES en utilisant la variable L comme paramètre:

```
void TRIANGLE(void)
{   int L; for (L=1; L<10; L++)    ETOILES(L);  }
```

Au moment de l'appel, la *valeur* de L est copiée dans N. La variable N peut donc être décrémentée à l'intérieur de ETOILES, sans influencer la valeur originale de L. Schématiquement, le passage des paramètres peut être représenté comme suit:



3.6.2 Passage de l'adresse d'une variable

Pour changer la valeur d'une variable de la fonction appelante, nous allons procéder comme suit:

- la fonction appelante doit ***fournir l'adresse de la variable*** *(paramètre effectif)* en utilisant le symbole **&** et,
- la fonction appelée doit ***déclarer le paramètre formel*** comme *Pointeur*[1] en utilisant le symbole ***** qui signifie le contenu de l'adresse.

[1] Défini dans le chapitre suivant

On peut alors atteindre la variable à l'aide de l'adresse.

Exemple :

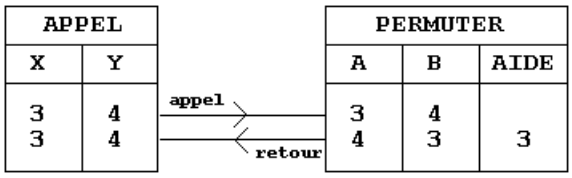
Nous voulons écrire une fonction PERMUTER qui échange le contenu de deux variables du type **int**. En première approche, nous écrivons la fonction suivante:

```
void PERMUTER (int A, int B)
{ int AIDE;   AIDE = A;    A = B;    B = AIDE;  }
```

Nous appelons la fonction pour deux variables X et Y par: **PERMUTER(X, Y);**

Résultat: X et Y restent inchangés !

Explication: Lors de l'appel, les *valeurs* de X et Y sont copiées dans les paramètres A et B. PERMUTER échange bien le contenu des variables *locales* A et B, mais les valeurs de X et Y restent les mêmes.



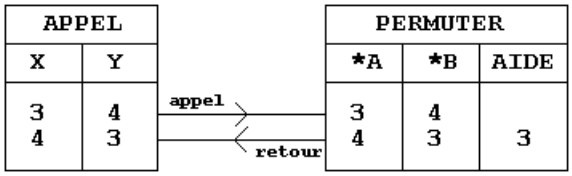
Pour pouvoir modifier le contenu de X et de Y, la fonction PERMUTER a besoin des adresses de X et Y. Nous réécrivons alors la fonction comme suit:

```
void PERMUTER (int *A, int *B)
{    int AIDE; AIDE = *A; *A = *B; *B = AIDE;    }
```

Nous appelons la fonction par: ***PERMUTER*(&X, &Y);**

Résultat: Le contenu des variables X et Y est échangé !

Explication: Lors de l'appel, les *adresses* de X et de Y sont copiées dans A et B. PERMUTER échange ensuite le contenu des adresses indiquées par A et B. C'est-à-dire échange *A et *B (contenu de A et contenu de B)



3.7 Déclaration globale des fonctions

Il faut déclarer chaque fonction avant de pouvoir l'utiliser. Si dans le texte du programme la fonction **est définie** avant son premier appel, elle n'a pas besoin d'être déclarée.
En déclarant toutes les fonctions globalement au début du texte du programme, nous ne sommes pas forcés de nous occuper de la dépendance entre les fonctions. Cette solution est la plus simple et la plus sûre pour des programmes complexes contenant une grande quantité de dépendances. Il est quand même recommandé de définir les fonctions selon l'ordre de leur hiérarchie.

Prototype d'une fonction
Pour déclarer globalement une fonction il faut donner le ***prototype*** de la fonction qui indique uniquement le type des données transmises et reçues par la fonction.

<TypeRés> <NomFonct> (<TypePar1>, <TypePar2>, ...); ou bien
<TypeRés> <NomFonct> (<TypePar1><NomPar1>,<TypePar2> <NomPar2>, ...);

Exemple :

```
#include<stdio.h>
void  affichesom (int  , int   );  /* prototype */
void affichediff (int  ,  int  ) ; /* prototype */
void main()
{ int x,y ;
  scanf("%d %d",&x,&y) ;
  …
}
void affichesom(int x, int y)
{    printf("La somme = %d",x+y) ;  }
void affichediff(int x, int y)
{    printf("La difference=%d",x-y) ;  }
```

3.8 Passage de structures comme argument de fonctions

- On peut transférer une structure entière comme argument d'une fonction.
- Une fonction peut retourner une structure.

Exemple :

```
/* Une fonction qui permute deux dates */
typedef  struct  { int Jour, Mois, Annee; } Date;

void permuter(Date *D1, Date *D2)
{ Date   D=*D1 ;
        *D1=*D2 ;
        *D2=D ;
}
/*   Une fonction qui retourne la date du lendemain  */
Date dateLendemain(Date D)
{  Date DLend;
   …
   return DLend;
}
```

Calculer la somme de deux nombres entiers.

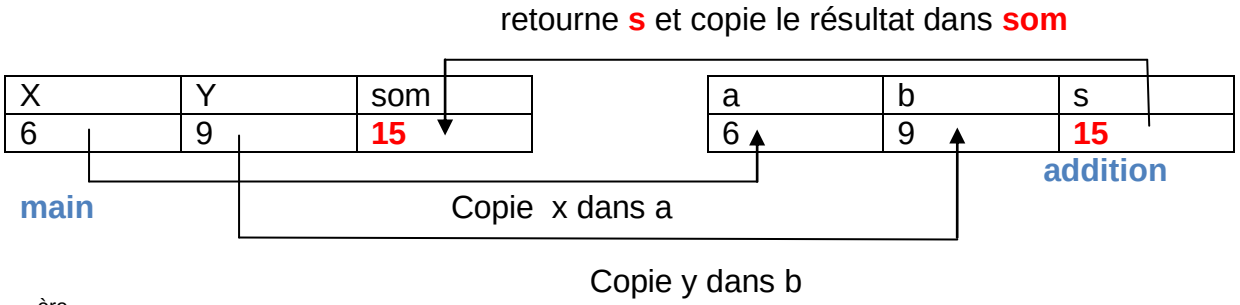
```
int addition(int a,  int b)
{   int s;
     s=a+b;  /* ici   S c'est le résultat de la somme de a+b  */
     return(s);  /* la fonction retourne ce résultat  */
}

main()
{  int x, y; int som ;
   Printf("donnez deux valeurs entières\n");
   Scanf("%d%d",&x,&y) ;
   som=addition(x,y) ;  /*  x et y sont transmis par valeur  car pour calculer la
                                  somme on a besoin des deux valeurs */
   /*  et som on lui affecte le résultat de la fonction addition puis on l'affiche */
   printf("la somme de %d + %d = %d",a , b, som) ;
 /* ici on affiche le résultat de la somme si la fonction ne retourne pas ce résultat on
n'affichera rien ?  il faut que la fonction envoie le résultat à main pour pouvoir
l'afficher   */
}
```

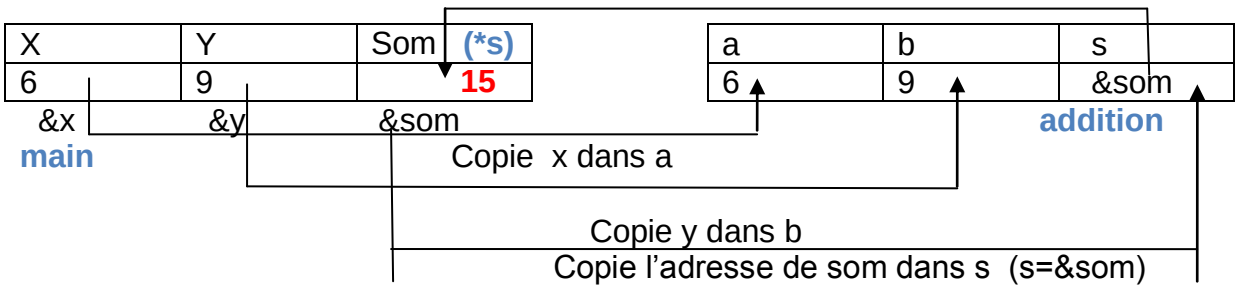
2ème solution :

```
void  addition(int a,  int b, int *s)  /* s est un pointeur sur som de main */
{      *s=a+b;  /* ici   *s c'est le résultat de la somme de a+b  */
     /* pas de return car la fonction est void mais le résultat c'est *s  donnée dans les
paramètres : les deux solutions sont correctes   */
}

main()
{  int x, y; int som ;
   Printf("donnez deux valeurs entières\n");
   Scanf("%d%d",&x,&y) ;
   addition(x,y, &som) ;  /*  x et y sont transmis par valeur  car pour calculer la
                                  somme on a besoin des deux valeurs */
/*  pour pouvoir obtenir le résultat de addition dans main,  som  est transmis par
adresse  c'est-à-dire on copie dans s (de la fonction addition) l'adresse de som
(s=&som)  et donc la somme sera affectée au contenu de s (*s=a+b) le contenu de s
c'est som.  * /
   printf("la somme de %d + %d = %d",a , b, som) ;
/* ici on affiche le résultat de la  somme, il faut que la fonction envois le résultat à
main pour pouvoir l'afficher   grâce à &som*/
}
```

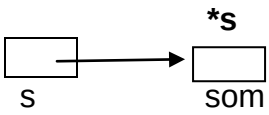


1ère solution la commande return copie la valeur de s dans som et ainsi on envoi le résultat à main.



2ème solution le calcul est effectué directement dans som grâce à son adresse qui est dans s.

s pointe sur **som**



Encore des exemples :

Transmission par valeur :

```
void  test1(int   x)
{   x=x+1 ;   /* ici je modifie  x  et x=11 */}
main()
{ int   y =10 ;
  test1(y) ; /* ici copie de la valeur de  y dans x de la fonction test1  donc x=y */
  printf(" y= %d", y); /*  ici  affiche y=10 car  nous avons fait une transmission par
                           valeur   nous avons modifié x mais pas y  */
```

Transmission par adresse :

```
void  test2(int  * x)
{   *x=*x+1 ;   /* ici je modifie  *x (le contenu de x c'est-à-dire y)  et   *x=11     */}
main()
{ int   y =10 ;
  test1(&y) ; /* ici copie de l'adresse de y dans x de la fonction test2  donc x=&y */
  printf(" y= %d", y); /*  ici  affiche y=11 car  nous avons fait une transmission par
                           adresse  nous avons modifié y  */
```

Chapitre 4. LES POINTEURS

4.1 Définition*:*

*Un **pointeur** est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.*

En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

Si un pointeur P contient l'adresse d'une variable A, on dit que ***'P pointe sur A'***.

4.2 Déclaration d'un pointeur

<Type> *<NomPointeur>

4.3 Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin

- d'un opérateur *'adresse de'*: **&** pour obtenir l'adresse d'une variable.
- d'un opérateur *'contenu de'*: ***** pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

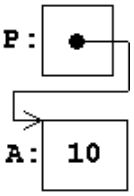
L'opérateur 'adresse de' : &

& <NomVariable> fournit l'adresse de la variable <NomVariable>

Représentation schématique

Soit **P** un pointeur non initialisé, et **A** une variable (du même type) contenant la valeur 10 :

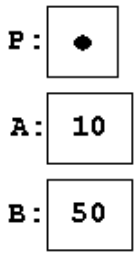
Alors l'instruction **P = &A;** affecte l'adresse de la variable A à la variable P. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:



L'opérateur 'contenu de' : *

***<NomPointeur>** désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>

Exemple : Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé:

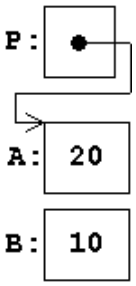


Après les instructions :

P = &A; P pointe sur A

B = *P; le contenu de A (Référencé par *P) est affecté à B

***P = 20;** le contenu de A (Référencé par *P) est mis à 20.



4.4 Les opérations élémentaires sur pointeurs

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

Priorité de * et &

- Les opérateurs ***** et **&** ont la même priorité que les autres opérateurs unaires (la négation **!**, l'incrémentation **++**, la décrémentation **--**). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

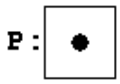
Exemple Après l'instruction **P = &X;** les expressions suivantes, sont équivalentes:

Y = *P+1	**Y = X+1**
***P = *P+10**	**X = X+10**
***P += 2**	**X += 2**
Y=++*P	**Y=++X** (incrémente puis affecte)
Y=(*P)++	**Y=X++** (affecte puis incrémente)

Dans le dernier cas, les parenthèses sont nécessaires. Comme les opérateurs unaires * et ++ sont évalués *de droite à gauche*, sans les parenthèses le *pointeur* P serait incrémenté, *non pas l'objet* sur lequel P pointe. On peut uniquement affecter des adresses à un pointeur.

Le pointeur NULL

La valeur NULL est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.



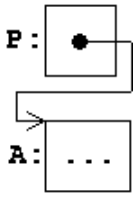

```
int *P;
P = NULL;
```

Remarque : Les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soient P1 et P2 deux pointeurs sur **int**, alors l'affectation **P1 = P2;** copie le contenu de P2 vers P1 alors P1 pointe sur le même objet que P2.

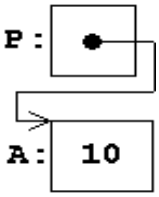
Résumé :

Après les instructions:

int A; int *P; P = &A;



***P=10 ;**



A désigne le contenu de A

&A désigne l'adresse de A

En outre

P désigne l'adresse de A

***P** désigne le contenu de A

&P désigne l'adresse du pointeur P

***A** est illégal (puisque A n'est pas un pointeur)

4.5 Pointeurs et fonctions

- Passage par adresse

Exemple : *scanf ("%d %d", &x,&y) ;*

*void permuter(int *a, int *b) Appel permuter(&x, &y) ;*

4.6 Pointeurs et tableaux

Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de son premier élément. En d'autre termes: **&tableau[0]** et **tableau** sont une seule et même adresse.

Il faut retenir donc que :

- ***le nom d'un tableau est un pointeur constant sur le premier élément du tableau.***

  Soit A le nom d'un tableau alors :

A+0 désigne &A[0]
A+1 désigne &A[1]
…
A+i désigne &A[i]

- ***Si P est un pointeur sur un tableau A, l'instruction P=A*** est équivalente à **&A[0]** alors :

P désigne &A[0]
P+1 désigne &A[1]
P+2 désigne &A[2]
…
P+i désigne &A[i]

et **P* désigne le contenu de l'élément pointé par *P*.

*P	désigne A[0]
*(P+1)	désigne A[1]
*(P+2)	désigne A[2]
…	
*(P+i)	désigne A[i]

- ***Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante***

  Plus généralement,

  P+i pointe sur la **i^{ème}** composante derrière **P** et

  P- i pointe sur la **i^{ème}** composante devant **P**.

Exemple
En déclarant un tableau **A** de type **int** et un pointeur **P** sur **int : int A[10]; int *P;**

Ainsi, après les instructions :
P = A; le pointeur **P** pointe sur **A[0]** est équivalent à P=&A[0] **et** P=A+0
***P = 3;** le contenu de **P** reçoit **3** est équivalent à A[0]=3

Autre Exemple
Soit **A** un tableau contenant des éléments du type **float** et ***P*** un pointeur sur **float**:
float A[20], X;
float *P;

Après les instructions,
P = A;
X = *(P+9); X contient la valeur du 10ème élément de **A**, (c.-à-d. celle de A[9]).

Important :
Il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau:
- ***Un pointeur est une variable***, donc des opérations comme **P = A** ou **P++** sont permises.
- ***Le nom d'un tableau est une constante***, donc des opérations comme **A = P** ou **A++** sont impossibles, au même titre que 3++.

Exemple
Les deux programmes suivants copient les éléments positifs d'un tableau T dans un deuxième tableau POS.

Formalisme tableau
```
#include<stdio.h>
#include<conio.h>
main()
{ int T[10] ,n, Pos[10], i, j ;
  scanf("%d",&n);
for(i=0;i<n;i++) scanf("%d", &T[i]);

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

Formalisme pointeur
```
#include<stdio.h>
#include<conio.h>
main()
{ int T[10] ,n, Pos[10], i, j ;
  scanf("%d",&n);
for(i=0;i<n;i++) scanf("%d", T+i);

for (j=0,i=0 ; i<n ; i++)
    if (*(T+i)>0)
      { *(Pos+j) = *(T+i);  j++;   }
for(i=0; i<j; i++) printf("%d", *(Pos+i));
}
```

4.7 Arithmétique des pointeurs

- Affectation par un pointeur sur le même type

Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction **P1=P2;** fait pointer P1 sur le même objet que P2.

- Addition et soustraction d'un nombre entier

Si P pointe sur l'élément A[i] d'un tableau, alors

P+n pointe sur A[i+n]

P-n pointe sur A[i-n]

Exemple : p=A ; ou bien p=&A[0]
p=p+9 → p pointe sur A[9]
p=p-1 → p pointe sur A[8]

- Incrémentation et décrémentation d'un pointeur

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

P++; P pointe sur A[i+1]

P+=n; P pointe sur A[i+n]

P--; P pointe sur A[i-1]

P-=n; P pointe sur A[i-n]

Exemple : Initialiser un tableau avec des 1

```
int  t[10], i ;
for(i=0 ;i<10 ;i++)
        t[i]=1 ; // oubien  *(t+i)=1
```

```
int t[10], *p;
for(p=t ;p<t+10 ;p++)
    *p=1;
```

Domaine des opérations

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies ***à l'intérieur d'un tableau***. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

- Soustraction de deux pointeurs

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau*:

P1-P2 fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction **P1-P2** est

- négatif, si P1 précède P2
- zéro, si P1 = P2
- positif, si P2 précède P1
- indéfini, si P1 et P2 ne pointent pas dans le même tableau

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

- Comparaison de deux pointeurs

On peut comparer deux pointeurs par **<**, **>**, **<=**, **>=**, **==**, **!=**.

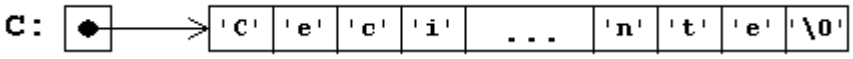
La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

4.8 Pointeurs et chaînes de caractères

a) Affectation

On peut attribuer *l'adresse d'une chaîne de caractères constante* à un pointeur sur **char**:

Exemple : **char *C; *//*** C est un pointeur sur **1** ou plusieurs caractères (chaine)
C = "Ceci est une chaîne de caractères constante";



Nous pouvons lire cette chaîne constante (p.ex: pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible.

b) Initialisation

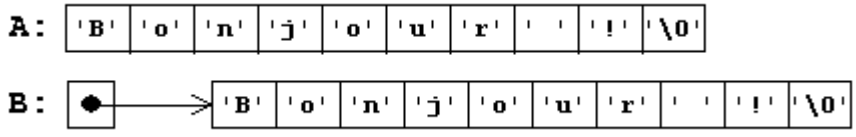
Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante: **char *B = "Bonjour !";**

Attention ! Il existe une différence importante entre les deux déclarations:

```
char A[ ] = "Bonjour !";   /* un tableau  */
char *B  = "Bonjour !";   /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom **A** va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. ***La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.***

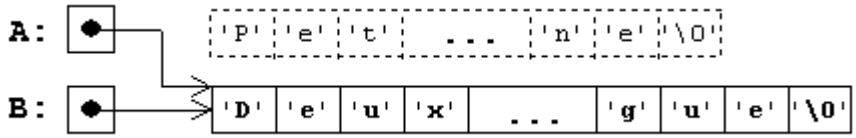


c) Modification

Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur (allocation dynamique sera vue au chapitre 5):

Exemple : **char *A = "Petite chaîne";**
char *B = "Deuxième chaîne un peu plus longue";
A = B;

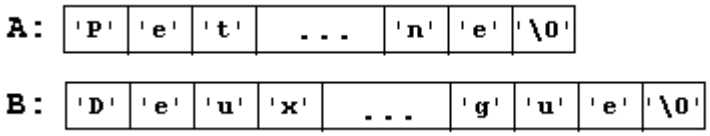
Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:



Important : Les affectations discutées ***ci-dessus*** ne peuvent pas être effectuées avec des tableaux de caractères:

Exemple :

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B;    /* IMPOSSIBLE -> ERREUR !!! */  (A est une adresse constante)
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */ (C aussi est une constante)
```



Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C. Ces opérations sont impossibles et illégales parce que ***l'adresse représentée par le nom d'un tableau reste toujours constante***.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre ou déléguer cette charge à une fonction de ***<string> (strcpy(s1,s2)*** **).**

4.9 Pointeurs et tableaux à deux dimensions

Exemple : Le tableau M à deux dimensions est défini comme suit:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe sur le ***tableau*** M[0] qui a la valeur: **{0,1,2,3,4,5,6,7,8,9}. (M représente &M[0])**
L'expression (M+1) est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a la valeur: **{10,11,12,13,14,15,16,17,18,19}. (M+1 représente &M[1])**

Explication
Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice M est le ***vecteur*** **{0,1,2,3,4,5,6,7,8,9}**, le deuxième élément est **{10,11,12,13,14,15,16,17,18,19}** et ainsi de suite.
L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que: **M+I** désigne l'adresse du tableau **M [I]**

&M[0][0] équivalent à M[0]+0 équivalent à *(M+0)+0
&M[0][1] équivalent à M[0]+1 équivalent à *(M+0)+1
&M[1][0] équivalent à M[1]+0 équivalent à *(M+1)+0
…

&M[i][j] équivalent à M[i]+j équivalent à *(M+i)+j

M[0][0] équivalent à *(M[0]+0) équivalent à *(*(M+0)+0)
M[0][1] équivalent à *(M[0]+1) équivalent à *(*(M+0)+1)
M[1][0] équivalent à *(M[1]+0) équivalent à *(*(M+1)+0)
…

M[i][j] équivalent à *(M[i]+j) équivalent à *(*(M+i)+j)

Important : M n'est pas de type **int *,** mais c'est un pointeur sur des blocs (lignes) donc si on désire pointer un pointeur P sur une matrice il faut faire une **conversion forcée** comme suit :
Int *p ; ~~p=M~~ ; ← faux
P=(int*) M ; convertir M qui est un pointeur sur un tableau en un pointeur sur un int

4.10 Tableaux de pointeurs

Déclaration : **<Type> *<NomTableau>[<N>]**

***Exemple* :** **double *A[10];**
Déclare un tableau de 10 pointeurs sur des réels du type **double** dont les adresses et les valeurs ne sont pas encore définies.

Remarque
Le plus souvent, les tableaux de pointeurs sont utilisés pour mémoriser de façon économique des *chaînes de caractères de différentes longueurs*.

4.11 Pointeurs et structures

Exemple :
typedef struct { int Jour, Mois, Annee; } Date;
Date *Ptr_Date; /* Ptr_Date pointe sur des objets de type Date */
Il est alors possible d'utiliser ce pointeur de la façon suivante :
(*Ptr_Date).jour=12 ou bien Ptr_Date->Jour = 12;

4.12 Passage d'un tableau comme argument d'une fonction

Exemple :
Retourner l'élément le plus petit parmi les composantes d'un tableau de ***n*** entiers données
On peut écrire les prototypes suivants (tous sont équivalents)

- /* la fonction retourne le résultat à travers son nom et le corps de la fonction peut être identique pour les trois prototypes suivants* /

```
int minimum1 (int t[50], int n) ;
int minimum1 (int t[ ], int n) ;
int minimum1 (int *t, int n) ;
```

- /* la fonction retourne le résultat à travers l'argument min */

```
void minimum2 (int *t, int n, int *min) ;
```

int minimum1 (int t[ ], int n) ;

```
{ int min=t[0] ; int i ;
for(i=1 ;i<n ;i++)
    if (t[i]<min) min=t[i] ;
return min;
}
```

// Appel min=minimum1(t, n) ;

Ou bien

```
int minimum1 (int *t, int n)
{ int min=*t, i ;
for(i=1 ;i<n ;i++)
      if (*(t+i)<min) min=*(t+i) ;
return min;
}
```

// Appel min=minimum1(t, n) ;

Ou bien

```
void minimum2 (int t[ ], int n, int *min) ;
{ int *min=t[0] ; int i ;
for(i=1 ;i<n ;i++)
      if (t[i]<*min) *min=t[i] ;
}
```

// Appel minimum2(t, n, &min) ;

Exercice1 : char *mois[12]={"Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Aout", "Septembre", "Octobre", "Novembre", "Décembre"} ;

a) Afficher chaque mois.
b) Afficher le 1er caractère de chaque mois.

……
……
……
……
……
……

Exercice 2:

Ecrire une fonction qui rempli une matrice carrée T[20][20] avec des 1, puis une fonction qui rempli la diagonale avec des zéros en utilisant uniquement un pointeur.

……
……
……
……
……
……
……
……
……

Exercice 3: Soit la déclaration suivante :

char *Jour[ ] = {"dimanche","lundi","mardi","mercredi","jeudi", "vendredi","samedi"};

Ecrire une fonction qui étant donné un entier **A,** qui peut prendre les valeurs de 1 à 7, affiche le jour de la semaine correspondant.

……
……

Exercice 4:

a- double (*a)[12] ; …….
b- double *a[12] ; …….
c- char *a[12] ; …….
d- char *d[4]={"Nord", "Sud", "Est", "Ouest"} ; ……..
i- que représente d ? ……
ii- que désigne d+2 ? ……
iii- quelle est la valeur de *d ? ……
iv- quelle est la valeur de *(d+2) ? …….
v- quelle est la différence entre d[3] et *(d+3) ? ……..
vi- que vaut *(*(d+2)+1) ? ……..

Exercice1 : char *mois[12]={"Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Aout", "Septembre", "Octobre", "Novembre", "Decembre"} ;

c) Afficher chaque mois.
d) Afficher le 1er caractère de chaque mois.

Solution :

a) int i ;
for(i=0; i<12; i++) printf("%s\n", mois[i]); /* Affiche le mois */
b) for(i=0; i<12; i++) printf("%c\n", *mois[i]) ; /* Affiche le 1er caractère du mois */
printf("%c\n", *mois[i]+1) ; /* affiche le 2ème caractère du mois */

Exercice 2:

Ecrire une fonction qui rempli une matrice carrée T[20][20] avec des 1, puis une fonction qui rempli la diagonale avec des zéros.

Solution :

```
void remplir(int T[20][20], int n)
{ int i,j ;
  for(i=0; i<n; i++)                    /*  Appel  remplir(T, n)  */
      for(j=0; j<n; j++)    T[i][j]=1;
}
void diag(int *p, int   n)
{  int i;
   for(i=0; i<n; i++)  { *p=0; p=p+(n+1);  }    /*  Appel  diag(T,n)   */
}
```

Exercice 3: Soit la déclaration suivante :

char *Jour[ ] = {"dimanche","lundi","mardi","mercredi","jeudi", "vendredi","samedi"};

Ecrire une fonction qui étant donné un entier **A,** qui peut prendre les valeurs de 1 à 7, affiche le jour de la semaine correspondant.

Solution : … printf(''%s'',Jour[A-1]) ;

Exercice 4:

a- double (*a)[12] ; → a → *a : 0 11

b- double *a[12] ; → tableau de 12 pointeurs sur des réels
c- char *a[12] ; → tableau de 12 pointeurs sur des caractères ou des chaines
d- char *d[4]={"Nord", "Sud", "Est", "Ouest"} ; tableau de 4 chaines
vii- que représente d ? → &d[0]
viii- que désigne d+2 ? → &d[2]
ix- quelle est la valeur de *d ? → d[0]=&d[0][0]=&'N' → "Nord"
x- quelle est la valeur de *(d+2) ? → d[2]=&d[2][0]=&'E' → "Est"
xi- quelle est la différence entre d[3] et *(d+3) ? aucune = &'O' → "Ouest"
xii- que vaut *(*(d+2) +1) ? → *(d[2]+1) = 's' du mot "Est"

Exercices sur les chaines :

1. La fonction copietab() copie les éléments d'une chaine de caractères T[] dans une autre chaine S[].

1ère Solution :

```
Void copietab(char S[ ], char t[ ])
{
  int i =0 ;
  while (T[i] !='\0')
  {   S[i] = T[i]; i++;  }
}
```

2ème Solution: Un programmeur expérimenté préfère la solution suivante :

```
Void copietab(char *s, char *T)
{
     while (*S++ = *T++) ;
}
```

2. En utilisant les fonctions de String.h, écrire une fonction qui supprime toutes les occurrences d'un caractère donné C dans une chaine S donnée.

Exemple : S= "element" C='e' → S=lmnt

Solution:

```
Void supprimcar(char *S, char C)
{ char *p=S
  while (p=strchr(p,C)!=NULL)  // ou bien while (p=strchr(S,C))
       strcpy(p,p+1);
}
```

3. Compter le nombre d'occurrences d'un caractère C dans une chaine S

Solution :

```
int compter(char *S, char C)
{ int cp=0 ; char *p=S ;
  while (p=strchr(S,C)) {nb++; p++; }
return nb;
}
```

Chapitre 5 : Allocation dynamique de mémoire

5.1 Introduction

Si nous générons des données pendant l'exécution d'un programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de ***l'allocation dynamique*** de la mémoire.

Déclaration statique de données

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des données. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation. Nous parlons alors de la ***déclaration statique*** des variables.

Exemple : int T[10] ; char Mot[30] ; …

Allocation dynamique

Problème

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

Exemple1 : int *T ; char * Mot ; …

Exemple2 : Nous voulons lire 10 phrases (de différentes tailles) au clavier et les mémoriser en utilisant un tableau de pointeurs sur **char**. Nous déclarons ce tableau de pointeurs par: **char *TEXTE[10];**

Pour les 10 pointeurs, nous avons besoin de *10*p octets*. Ce nombre est connu dès le départ et les octets sont réservés automatiquement. Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...
La réservation de la mémoire pour les 10 phrases peut donc seulement se faire *pendant l'exécution du programme*. Nous parlons dans ce cas de l'***allocation dynamique*** de la mémoire.

5.2 La fonction malloc

La fonction **malloc** de la bibliothèque ***<stdlib>*** nous aide à localiser et à réserver de la mémoire au cours d'un programme.

La fonction **malloc(<N>)** fournit l'adresse d'un bloc en mémoire de <N> octets libres ou la valeur zéro s'il n'y a pas assez de mémoire.

Exemple :
Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de 4000 caractères. Nous disposons d'un pointeur T sur **char** (**char *T**).
Alors l'instruction: **T = malloc(4000);** fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

5.3 L'opérateur sizeof

Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type. ***sizeof()*** revoie donc le nombre d'octets utilisés pour stocker un objet.
L'opérateur **sizeof** nous aide alors à préserver la portabilité du programme.

L'opérateur unaire sizeof
sizeof <var> fournit la grandeur de la variable <var>
sizeof <cons> fournit la grandeur de la constante <const>
sizeof (<type>) fournit la grandeur pour un objet du type <type>

Exemple
Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X est lue au clavier:

```
int X;
int *PNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &X);
PNum = (int *)malloc(X*sizeof(int));
```

Remarque: Si l'espace mémoire doit contenir un autre type que char il faut forcer le type de la fonction malloc, comme présenté dans l'exemple précédent.

5.4 La commande exit

S'il n'y a pas assez de mémoire pour effectuer une action avec succès, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande **exit** (de ***<stdlib>***) et de renvoyer une valeur différente de zéro comme code d'erreur du programme.

Exemple

Le programme à la page suivante lit 10 phrases au clavier, recherche des blocs de mémoire libres assez grands pour la mémorisation et passe les adresses aux composantes du tableau **TEXTE[]**. S'il n'y a pas assez de mémoire pour une chaîne, le programme affiche un message d'erreur et interrompt le programme avec le ***code d'erreur -1.***

Nous devons utiliser une variable d'aide **INTRO** comme zone intermédiaire (non dynamique). Pour cette raison, la longueur maximale d'une phrase est fixée à 500 caractères.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{    char INTRO[500];
     char *TEXTE[10];
     int i;

   for (i=0; i<10; i++)
     {  gets(INTRO);

        TEXTE[i] = malloc(strlen(INTRO)+1);  /* Réservation de la mémoire */

        if (TEXTE[i])   /* S'il y a assez de mémoire  */
          strcpy(TEXTE[i], INTRO); /* copier la phrase à l'adresse fournie par malloc  */
        else
         {   /* sinon quitter le programme après un message d'erreur. */
          printf("ERREUR: Pas assez de mémoire \n");
          exit(-1);
         }
    }
}
```

5.5 La fonction free

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque <*stdlib*>.

free(<Pointeur>)

Libère le bloc de mémoire désigné par le **<Pointeur>**; n'a pas d'effet si le pointeur a la valeur zéro.

Exemple : Soit le programme suivant :

```
#include<stdio.h>
#include<stdlib.h>
main()
{  int i=3;   int *p;
   printf("valeur de p avant allocation=%d\n", p) ;
   p=(int *) malloc (sizeof(int)) ;
   printf("valeur de p après allocation=%d\n", p) ;
   *p=i ;
   printf("valeur de *p=%d\n", *p) ;
}
```

objet	***adresse***	***valeur***	
i	1245060	3	
p	1245064	0	Avant allocation
i	1245060	3	
p	1245064	8004260	
*p	8004260	?(int)	Après allocation
i	1245060	3	
p	1245064	8004260	
*p	8004260	3	

```
main()
{  int i=3 ;  int *p ;   p=&i ;  }
```

objet	***adresse***	***valeur***	
i	1245060	3	
p	1245064	1245060	i et *p sont identiques (même adresse)
*p	1245060	3	

5.6 Les Listes chaînées

5.6.1/ Définitions

a) Structures récursives : Les structures récursives font référence à elles mêmes. On cite : les listes chainées et les arbres.

b) Listes chainées : C'est une structure dynamique qui s'agrandit au fur et à mesure de la lecture des données. Une liste est constituée de ***cellules chainées***.
Une cellule est composée de deux champs :

- **i)** ***élément***, contenant un élément de la liste
- **ii)** ***suivant***, contenant un pointeur sur une autre cellule.
- **iii)** les cellules ne sont pas rangées séquentiellement en mémoire. D'une exécution à l'autre leur localisation peut changer.
- **iv)** une position (adresse) est un pointeur sur une cellule.
- **v)** rajouter ou supprimer un élément ne nécessite pas de décalage.
- **vi)** une liste est un pointeur sur la cellule qui contient le premier élément de la liste que l'on appelle ***tête de liste***.
- **vii)** la liste vide est le pointeur ***NULL***.

5.6.2/ Listes simplement chaînées

a) <u>Déclaration</u>

- **typedef struct** <ident1> { <type_elements> <nom var_elem>;
  struct <ident1> ***** <nom var_suivant>; // pointeur
  } <ident2> ;

<u>exemple :</u> typedef **struct ListEtud** {char nom[20], prenom[20] ;
float moyenne ;
struct ListEtud * svt ;} noeudE ;
struct ListEtud * teteEt ;
type des cellules

/* ou bien * /

- typedef struct <ident1> * <ident3> ; // nouveau type ident3
  typedef struct <ident1> { <type_elements> <nom var_elem> ;
  <ident3> <nom var_suivant> ; // pointeur
  } <ident2> ;

<u>exemple :</u> typedef struct ListEtud * **ListEt** ; ⟵ type des cellules
typedef struct ListEtud {char nom[20], prenom[20] ;
float moyenne ;
ListEt svt ;} noeudE ;
ListEt teteEt ;

Remarque : Pour la suite du cours on utilisera la déclaration ci-dessous :

```
typedef  <type>   typelem ;   // <type> peut être un int, float, char …
typedef struct  id1 * Liste ;
typedef struct id1 { typelem  element ;
                     Liste suivant; } noeud;
Liste L;
```

b) Accès à une valeur d'une liste

Liste L ;
typelem a=**(*L).**element ;

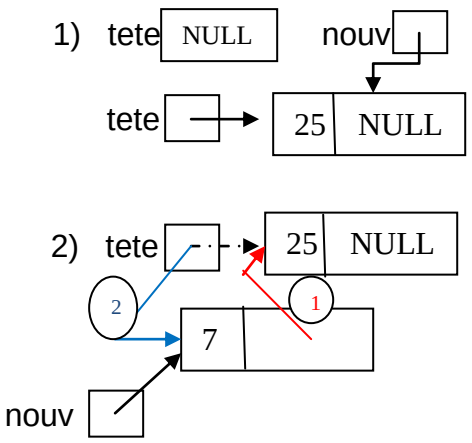
mais généralement on utilise la notation pointée →
typelem a=L→element ;

c) Création d'un nœud

```
Liste  creer_noeud( )
{  Liste  L= (Liste)malloc(sizeof(nœud)) ;
   if ( ! L) {   printf("erreur d'allocation\n") ;
                 exit(-1) ;
               }
   return(L) ;
}
```

d) Ajout d'un élément en tête de liste

```
void ajout_tete(Liste *tete, typelem E)  // tete est transmise par adresse
 { Liste nouv=creer_noeud ( ) ;
  Nouv→element=E ;
  nouv→suivant=*tete ; (1)
  *tete=nouv ; (2)
}
```



e) Ajout d'un élément après une adresse donnée

```
void ajout(Liste  *prd, typelem E)
 { Liste nouv=creer_noeud ( ) ;
  nouv→element=E ;
  nouv→suivant=*prd->suivant ;
  *prd→suivant=nouv ;
  *prd=nouv ; // Le nouvel élément devient le précédent pour un autre éventuel ajout
 }
```

f) Suppression d'un élément en tête de liste

```
void supprim_tete(Liste *tete)
{    Liste temp=*tete ;
     * tete=*tete→suivant ;
     free(temp) ;
 }
```

g) Suppression de l'élément après une adresse donnée

```
void supprim(Liste prd, Liste p)
{   prd→suivant=p→suivant ;
    free(p) ;
}
```

h) Création d'une liste

- **Création FIFO :**

```
typedef   int  typelem ;  /* Créer une liste de n valeurs entières */
Liste creer_listeFifo ( )
{  Liste tete=NULL, prd ; int n, i;  typelem E ;
   scanf("%d",&n) ;
   scanf("%d",&E) ;
   ajout_tete(&tete, E) ;
 prd=tete;
for(i=2 ; i<=n; i++)
   {  scanf("%d", &E);
      ajout(&prd, E);
   }
return(tete);
}
```

- **Création LIFO :**

/* créer une liste de n valeurs entières */

```
Liste créer_listeLifo ( )
{  Liste tete=NULL ; int n, i ; typelem E;
   scanf("%d",&n) ;
   for(i=1 ; i<=n; i++)
      {  scanf("%d",&E);
         ajout_tete(&tete, E);
      }
   return(tete);
}
```

i) Parcours d'une liste

Exemple : Afficher les éléments d'une liste d'entiers, dont le point d'entrée est **tete**.

```
void  Affiche_liste(Liste tete)
{       while(tete !=NULL)
            {  printf("%d\t",tete→element);  tete=tete→suivant;    }
}
```

Ou bien :

```
void  Affiche_liste(Liste tete)
{ for ( ; tete!=NULL; tete=tete→suivant)
        printf("%d\t", tete→element);
}
```

```
void  Affiche_liste(Liste tete)
{ Liste p ;
   for (p=tete; p!=NULL; p=p→suivant)
         printf("%d\t", p→element);
}
```

j) Recherche d'une valeur donnée

- Recherche d'une valeur donnée, dans une liste **L** d'entiers ***triés*** par ordre croissant et retourne son adresse, si elle existe, sinon elle retourne l'adresse où elle doit être insérée ainsi que l'adresse de l'élément précédent

```
Liste rechercheT(Liste L, typelem val, Liste *prd)
{ while(L !=NULL && L→element<Val)
      {  *prd=L;
         L=L→suivant;    }
  if (L!=NULL && L→element==val) return L;
  else return NULL;
}
```

- Recherche d'une valeur donnée, dans une liste **L** d'entiers ***quelconques*** et retourne son adresse, si elle existe sinon retourne NULL

```
Liste recherche(Liste L, typelem val)
{
   while(L !=NULL && L→element!=Val)
      {
         L=L→suivant;
       }
  return L;
}
```

- Recherche d'une valeur donnée, dans une liste **L** d'entiers ***quelconques*** et retourne son adresse, si elle existe, ainsi que l'adresse de l'élément précédent car cette fonction sera utilisée pour supprimer une valeur.

```
Liste rechercheS(Liste L, typelem val, Liste *prd)
{
   while(L !=NULL && L→element!=Val)
      {  *prd=L;
         L=L→suivant;
       }
  return L;
}
```

k) Mise à jour d'une liste

- **Modification :**

Exemple : Remplacer dans une liste **L** d'entiers, une valeur donnée **X** par une valeur donnée **Y**.

```
void  Modifier(Liste L, int X, int Y))
{  while (L !=NULL && L→element!=X)  L=L→suivant;
   if (L!=NULL) L→element=Y;
   else  printf("%d n'existe pas",X);
 }
```

Ou bien :

```
void  Modifier(Liste L, int X, int Y))
{  Liste prd,
    p=recherche(L, X, &prd);
    if (p!=NULL)  p→element=Y ;
    else  printf("%d n'existe pas",X);
}
```

- **Insertion :**

Exemple : Insérer une valeur **Val** donnée dans une liste d'entiers triés dans l'ordre croissant, de point d'entrée **tete**.

```
void  Inserer(Liste *tete, int Val)
{
   Liste p=*tete, prd=NULL;
   p=rechercheT(*tete, val, &prd);
   if (p= =*tete)     ajout_tete(tete, val) ;
   else  ajout_Apres(&prd, val) ;
 }
```

- **Suppression :**

Exemple : Supprimer une valeur **Val** donnée dans une liste d'entiers quelconques, de point d'entrée **tete**.

```
void  Supprimer(Liste *tete, int Val))
{
   Liste p, prd;
    if (p=rechercheS(*tete, val, &prd) )
          if (p= =*tete)  supprim_tete(tete) ;
          else  supprim(prd, p) ;
   else printf("Suppression impossible, la valeur n'existe pas\n") ;
}
```

5.6.3/ Listes bidirectionnelles

a) Déclaration

```
typedef  struct ne *ListeB ;
typedef  struct ne  { <type_elements> element ;
                      ListeB suivant, precedent ;
                    } nœudB ;
 ListeB L ;
```

b) Création d'un nœud

```
ListeB créer_noeudB( )
{  ListeB L= (ListeB)malloc(sizeof(nœudB)) ;
   if ( ! L) { printf("erreur d'allocation\n") ; exit(-1) ;}
   return(L) ;
}
```

c) Ajouter un élément en tete de liste

```
void ajout_teteB (ListeB *tete, typelem E)
{ ListeB nouv=créer_noeudB ( ) ;
  nouv→element=E;
  nouv→precedent=NULL;
  nouv->suivant=*tete;
  if (*tete !=NULL) *tete→precedent=nouv ;
  *tete=nouv;
 }
```

d) Ajouter un élément après une adresse donnée

```
void ajoutB (ListeB *prd, typelem E)
{ ListeB  nouv=creer_noeudB ( ) ;
  ListeB  p=*prd→suivant ;
  nouv→element=E ;
  *prd→suivant=nouv;
  nouv→precedent=*prd ;
  nouv->suivant=p ;
  if (p !=NULL) p→prd=nouv ;
  *prd=nouv ;
}
```

e) Création d'une liste

- **Création FIFO :**

```
typedef int  typelem ;

ListeB créer_listeBFifo( )
{    /* créer une liste de n valeurs entières */
   ListeB tete=NULL, prd; int n, i ;  typelem E;
   scanf("%d",&n) ;
   scanf("%d",&E) ;
   ajout_teteB( &tete, E) ;
  prd=tete;
   for(i=2 ; i<=n; i++)
     {  scanf("%d",&E);
        ajoutB(&prd, E);
     }
  return(tete);
}
```

- **Création LIFO :**

```
ListeB créer_listeBLifo ( )
{   /* créer une liste de n valeurs entières */
   ListeB tete; int n, i ; typelem E;
   scanf("%d",&n) ;
   tete=NULL ;
   for(i=1 ; i<=n; i++)
      {  scanf("%d",&E);
         ajout_teteB( &tete, E) ;
       }
   return(tete);
}
```

f) Mise à jour d'une liste

- **Insertion :**

Exemple : Insérer une valeur **Val** donnée dans une liste bidirectionnelle **L** d'entiers triée dans l'ordre croissant.

```
ListeB rechercheBT(ListeB tete, typelem val)
{   ListeB p=tete ;
    while (p !=NULL && p→element<val) p=p→suivant ;
    return p;
}
```

```
void  Inserer(ListeB *L, int val))
{
   ListeB p, prd=NULL, temp ;
   p=rechercheBT(*L, val);
   if (p= =*L)  ajout_teteB(L, val);   /* insertion en tête*/
   else    ajoutB(p→precedent, val) ;
}
```

- **Suppression :**

Exemple : Supprimer une valeur **Val** donnée dans une liste bidirectionnelle **L** d'entiers quelconques.

```
ListeB rechercheB(ListeB tete, typelem val)
{  ListeB p ;
   while(p !=NULL && p→element !=val) p=p→suivant;
  return p;
}

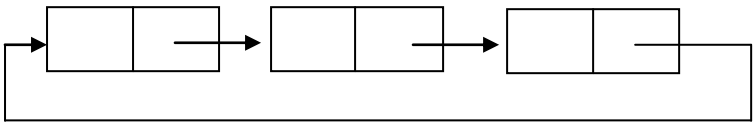
void supprim_teteB (ListeB *tete)
{  ListeB p=*tete;
    *tete=*tete→suivant;
    if (*tete!=NULL) *tete→precedent=NULL;
    free(p);
}

void supprimB (ListeB p)
{  ListeB prd=p→precedent, R=p→suivant;
   prd→suivant=R;
  if (R !=NULL) R→precedent=prd ;
  free(p);
}

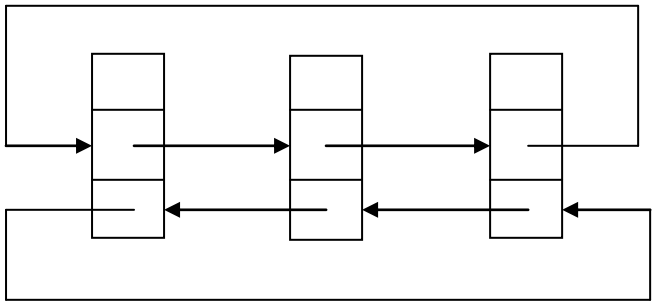
void  Supprimer(ListeB *L, typelem Val))
{  ListeB p=*L, prd;
   p=rechercheB(*L, val) ;
   if (p ! =NULL)
     {
       if (p= =*L)  supprim_TeteB(L) ;   /* suppression en tête*/
       else  supprimB(p) ;
      }
  else printf("Suppression impossible, la valeur n'existe pas\n") ;
}
```

5.5.3/ Listes circulaires

a) Listes circulaires simplement chaînées



b) Listes circulaires bidirectionnelle



Exemple : Vérifier si un mot donné représenté dans une liste chainée bidirectionnelle circulaire de caractères est un palindrome.

Solution :

```
ListeB dernier(ListeB tete)        // si la liste n'est pas circulaire
{  while (tete→suivant != NULL) tete=tete→suivant ;
   return tete ;
}
```

Ou bien :

```
 ListeB dernier(ListeB tete)      // si la liste n'est pas circulaire
{  for (  ;tete→suivant != NULL ; tete=tete→suivant) { } ;
   return tete ;
}
```

```
int Palindrome(ListeB tete)
{  listeB queue=dernier(tete) ;   /* cette instruction est exécutée si la liste n'est pas
                                     Circulaire */
queue=tete->precedent ; // si la liste est circulaire
  while (tete !=queue && tete→suivant !=queue && tete→element==queue→element)
       {    tete=tete→suivant ; queue=queue→precedent ;   }
  if (tete→element==queue→element) return 1 ;
  else return 0;   }
```

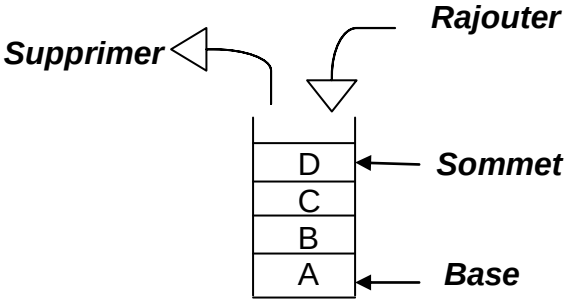
Chapitre 6 : Les PILES et les FILES

6.1 Les PILES :

6.1.1 Définition :

Une pile est une structure de donnée contenant une lise d'éléments telle qu'un élément ne peut lui être retiré ou ajouté que par une seule extrémité appelé ***sommet*** de la pile. Une pile est utilisée pour stocker ***temporairement*** des données.

Exemple : pile d'assiettes, pile de dossiers …



Un élément ne peut être rajouté ou supprimé qu'au sommet de la pile. Le dernier élément ajouté sera le premier retiré. C'est donc une structure de type ***LIFO*** *(« Last in, first out » = « dernier arrivé premier sorti »)*.
Les éléments sont retirés dans l'ordre inverse de celui de leur introduction.

6.1.2 Les opérations sur les piles :

Les opérations sur les piles sont les suivantes :

- ***Ajout*** d'un élément : l'action consistant à ajouter un nouvel élément au sommet de la pile s'appelle ***empiler*****,** puis mettre à jour ce sommet.
- ***Suppression*** d'un élément : l'action consistant à retirer un élément, celui qui est au sommet, s'appelle ***désempiler***, à condition que la ***pile ne soit pas vide***.
- ***Consultation*** : consulter le sommet de la pile en recopiant dans une variable l'élément du sommet sans affecter ni le contenu ni la position du sommet.

La manipulation des opérations sur les piles peut dépendre du type de la représentation de la pile : contiguë ou chaînée.

6.1.3 Représentation contiguë et chainée :

6.1.3.1 Implémentation d'une pile par un tableau

On peut représenter une pile par un tableau (contiguë), la déclaration de la pile doit alors tenir compte de la taille maximale. C'est-à-dire si le tableau est plein, il n'est pas possible de rajouter d'éléments.

Il est donc nécessaire avant de rajouter un élément dans la pile de tester si la ***pile n'est pas pleine***.

A	B	C	D	…..					
0	1	2	3						taille max
Base			Sommet						

6.1.3.2 Implémentation d'une pile par une liste chainée

Dans une pile représentée par une liste la suppression et l'insertion se font en tête de la liste.

6.1.3.3 Opérations de manipulation des deux représentations

Représentation contigüe

1) Déclaration

```
#define max 100
typedef int typelem;
typedef struct { typelem T[max];
                 int sommet;
               } pile;

pile p;
```

2) Initialisation

```
pile  initpile ( )
{ pile p ;
  p.sommet=-1;
  return(p) ;
}
```

3) Test si la pile est vide

```
 int  pilevide(pile p)
{ if (p.sommet==-1) return(1) ;
 else return(0);
}
```

4) Test si la pile est pleine

```
int pilepleine (pile p)
{ if (p.sommet==max-1) return(1) ;
  else return(0) ;
}
```

5) Consultation du sommet de la pile

```
typelem  sommetpile(pile p)
{ typelem  x ;
   x= p.T[p.sommet];
   return(x) ;
}
```

6) Ajout d'un element

```
void empiler(pile *p, typelem x)
{  (*p).sommet++;
   (*p).T[(*p).sommet] = x;
}
```

Représentation chaînée

1) Déclaration

```
typedef int typelem ;
typedef struct no *pile;
 typedef struct no
        { typelem valeur ;
          pile svt ;
        } nœud ;
 pile p;
```

2) Initialisation

```
 pile  initpile ( )
{ pile p ;
  p=NULL;
  return(p) ;
}
```

3) Test si la pile est vide

```
 int  pilevide(pile p)
{ if (p==NULL) return(1);
  else return(0);
}
```

N'existe pas

5) Consultation du sommet de la pile

```
typelem  sommetpile(pile p)
{ typelem  x ;
   x=p->valeur ;
   return(x) ;
 }
```

6) Ajout d'un element

```
void empiler(pile *p, typelem x)
{ pile temp = créer_noeud() ;
  temp ->valeur = x ;
  temp->svt=*p ; *p=temp ;
}
```

7) Suppression d'un élément

```
void désempiler(pile *p, typelem *x)
{
   *x = (*p).T[(*p).sommet];
   (*p).sommet - -;
}
```

```
void désempiler(pile *p, typelem *x)
{ pile temp ;
   *x = (*p) ->valeur  ;
   temp=*p ;
   *p=(*p)->svt; free(temp) ;
 }
```

Remarque :
Si le mode de représentation n'est pas spécifié on utilisera les fonctions précédemment définies dans leur généralité comme suit :

p=initpile() ;
empiler(&p,x) ;
désempiler(&p,&x) ;
x=sommetpile(p) ;
if (!pilevide(p)) …

p étant déclaré : *pile p ;* *Les fonctions sont alors considérées comme prédéfinies.*

6.1.4 Transformation des expressions :

6.1.4.1 Présentation du problème

Une utilisation courante des piles est l'élaboration par le compilateur d'une forme intermédiaire de l'expression. Après l'analyse syntaxique et lexicale, l'expression est traduite en une forme intermédiaire plus facilement évaluable.

Soit l'expression : A + B, son évaluation ne peut être faite immédiatement lors de la rencontre d'un opérateur car le 2ème opérande n'est pas encore connu par la machine. Par contre si l'expression pouvait être écrite sous la forme AB+ alors elle serait directement évaluable car les deux opérandes sont connus avant l'opérateur.
La notation **< *Opérande*> < *Opérateur*> < *Opérande*>** est dite INFIXE.
La transformation en autre représentation plus facilement évaluable est dite POSTFIXE ou PLONAISE SUFFIXE a qui a la forme :

< *Opérande Gauche* > < *Opérande Droit* > < *Opérateur*>

6.1.4.2 Transformation des expressions Infixées en Postfixées

Exemple :

(A+B)*3 ⟶ AB+3*
A+B*3 ⟶ AB3*+
(A≤B) and (not C) ⟶ AB≤C not and
(A+B)-(C*D)/E ⟶ AB+CD*E/-

Pour transformer une expression Q Infixée en expression Postfixée il faut lui appliquer un certain nombre de règles.
Q étant donnée, nous utilisons deux piles :
- une pile P (des opérateurs) pour les traitements intermédiaires,
- et une pile R qui contiendra le résultat final en fin de traitement.

Algorithme de transformation :

1- Ecrire l'expression Q avec ***')'*** à sa fin
2- Initialiser la pile P avec ***'('***
3- **Début** boucle : *lire* un élément de Q (progresser de gauche à droite)
    4- Si ***opérande*** le mettre dans la pile R
    5- Si ***'('*** la mettre dans la pile p
    6- Si ***opérateur Opt***
        - désempiler P et mettre dans R tous les opérateurs de priorité **>** à ***Opt*** jusqu'à '('
        - empiler ***Opt*** dans P
    7- Si ***')'*** - désempiler p et mettre dans R tous les opérateurs jusqu'à ***'('***
        - supprimer ***'('*** de P
    8- Recommencer 3, 4, 5, 6 et 7 jusqu'à la fin de Q
9- Fin boucle
10- Fin.

Exemple d'application :

Q	P	R
A	(	A
+	(	A
B	(+	AB
-	(+	AB
C	(+ -	ABC
*	(+ -	ABC
D	(+ - *	ABCD
)	(+ - *	ABCD * - +

6.1.4.3 Evaluation des expressions Postfixées

Algorithme :

1- Ecrire l'expression R avec ***')'*** à sa fin
2- **Début** boucle : *lire* un élément de R (progresser de gauche à droite)
    3- Si ***opérande*** le mettre dans la pile p
    4- Si ***opérateur Opt***
        - désempiler les 2 premiers opérandes ***Op1*** et ***Op2*** de P
        - évaluer ***Op2 Opt Op1*** (faire attention à l'ordre)
        - empiler le résultat de l'évaluation intermédiaire dans P
    5- Recommencer 2, 3, et 4 jusqu'à la fin de R
6- Fin boucle
7- Fin.

Exemple :

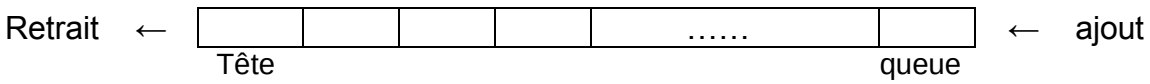
R	P
A	A
B	AB
C	ABC
D	ABCD
*	AB(C*D)
-	A+(B - (C*D))
+	(A+(B – (C*D)))
)	

Remarque : Les parenthèses, contenues dans la pile P ne servent qu'à montrer dans quel ordre sont effectuées les opérations.

6.2 Les FILES :

6.2.1 Définition

La notion de file est très courante dans notre vie pratique. Sa principale caractéristique est le « 1er arrivé 1er servi » ou ***FiFo « first in first out »***.



Une file permet deux accès : un accès appelé tête désigne le 1er élément à supprimer et un accès appelé queue désigne le dernier élément ajouté.
Comme pour les piles une file peut être représenté de 2 manières : ***contiguë*** ou ***chaînée.***

6.2.2 Opérations de manipulation des deux représentations

Représentation contigüe

Représentation chaînée

1) Déclaration

```
#define max 100
typedef int typelem;
typedef struct { typelem T[max];
                 int queue, tete;
               } file;

file F;
```

```
typedef int typelem ;
typedef struct no *file;
typedef struct no
    { typelem valeur ;
      file svt ;
    } nœud ;
file F;
```

2) Initialisation

```
file  initfile ( )
{ file F ;
  F.queue=-1;  F.tete=-1;
  return(F) ;
}
```

```
file  initfile ( )
 { file F ;
   F=NULL;
   return(F) ;
 }
```

3) Test si la file est vide

```
 int  filevide(file F)
{ if (F.tete==-1) return(1) ;
 else return(0);
}
```

```
 int  filevide(file F)
{ if (F==NULL) return(1);
   else return(0);
 }
```

4) Test si la file est pleine

```
int filepleine (file F)
{ if (F.queue==max) return(1) ;
  else return(0) ;
}
```

N'existe pas

5) Consultation de la tête de file

```
typelem  sommetfile(file F)
{ typelem  x ;
   x= F.T[F.tete];
   return(x) ;
}
```

```
typelem  sommetfile(file F)
  { typelem  x ;
    x=F->valeur ;
    return(x) ;
  }
```

6) Consultation de la queue de file

```
typelem  queuefile(file F)
{
   typelem  x ;
   x= F.T[F.queue];
   return(x) ;
}
```

```
typelem  queuefile(file F)
{ typelem  x ;
  file  queue=dernierelem(F);
   x=queue->valeur ;
   return(x) ;
}
```

7) Ajout d'un element

```
void emfiler(file *F, typelem x)
{  (*F).queue++;
   (*F).T[(*F).queue] = x;
   If (filevide(*F)) (*F).tete++;
}
```

```
void emfiler(file *F, typelem x)
{ file queue, temp=créer_noeud() ;
  temp ->valeur = x ;
  temp->svt=NULL ;
  if (filevide(*F)) *F=temp;
  else {queue=dernierelem(*F);
         queue->svt=temp;
         queue=temp;  }
  }
```

8) Suppression d'un élément

```
void désemfiler(file *F, typelem *x)
{  int i ;
   *x = (*F).T[(*F).tete];
   for(i=(*F).tete;i<(*F).queue;i++)
       (*F).T[i]=(*F).T[i+1];
   (*F).queue--;
}
```

```
void désemfiler(file *F, typelem *x)
{ file temp ;
   *x = (*F) ->valeur  ;
    temp=*F;
   *F=(*F)->svt;
   free(temp) ;
```

Chapitre 7 : La Récursivité

7.1 Définitions :

7.1.1 Objet récursif :

Un objet est dit récursif s'il est utilisé directement ou indirectement dans sa définition.

Exemple :
En définissant une expression arithmétique <expr> ou un identificateur <idf> ou une constante <cste> nous donnons une définition récursive comme suit :
Si θ est un opérateur on aura : <expr> → <expr> θ <expr> / <idf> / <cste>.

7.1.2 Programmation récursive :

La programmation récursive est une technique de programmation qui remplace les instructions de boucle (while, for, etc.) par des appels de fonctions.

7.1.3 Action paramétrée récursive :

Une action paramétrée P est dite récursive si son exécution provoque ou entraîne un ou plusieurs appels à P. Ces appels sont dits récursifs.

7.1.4 Algorithme récursif :

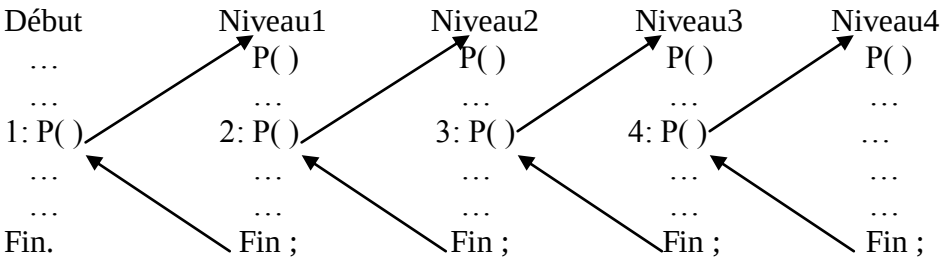
Un algorithme récursif est un algorithme qui contient une ou plusieurs actions paramétrées récursives.

7.1.5 Auto-imbrication :

L'auto-imbrication est le fait qu'une action paramétrée P peut s'appeler elle même avant que sa première exécution ne soit terminée, la seconde exécution peut de nouveau faire appel à P et ainsi de suite.

Exemple :
Soit l'action paramétrée suivante :

```
Action P( )
 Début
   P( );
 Fin ;
```



Chaque appel à P se fait à un niveau donné. L'exécution de P au niveau i se termine avant l'exécution de P au niveau i-1.
Remarque : Le nombre de niveaux doit être fini quelque soit les valeurs des paramètres d'appel, autrement l'algorithme va boucler indéfiniment.

7.2 Principes de construction d'algorithmes récursifs :

Exemple :

Le calcul de la valeur factorielle d'un nombre donné n (n>=0) peut se faire de deux manières différentes :

1ère méthode :

n ! = 1*2*3*…*n-1*n sachant que 0 !=1

On obtient alors l'algorithme itératif (classique) donné par la fonction suivante :

```
int fact (int n)
{ int i,p=1 ;
   for(i=1; i<=n; i++) p=p*i;
 return p;
}
```

2ème méthode :

0!=1 ; 1!=1 ; 2!=1*2 ; 3!=1*2*3 ; 4!=1*2*3*4 ; 5!=1*2*3*4*5 …
=2 =6 =24 =120 …

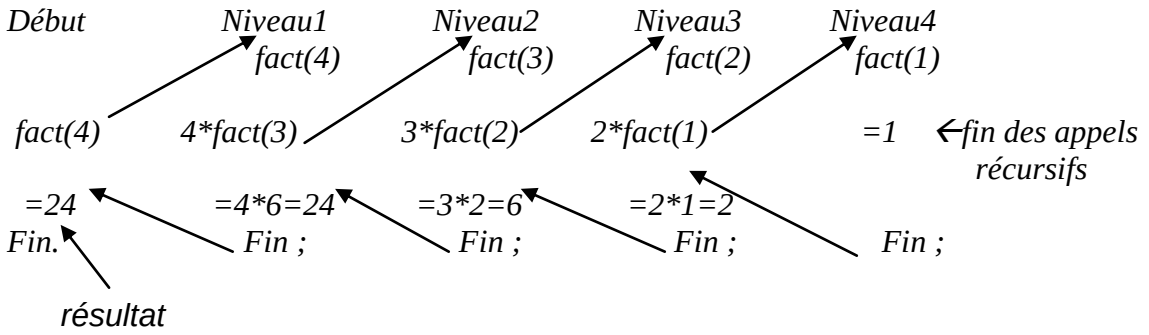
Nous remarquons que:
0!=1 ; 1!= 0!*1 ; 2!=1!*2 ; 3!= 2!*3 ; 4!=3!*4 ; 5!=4!*5

De façon générale pour un n donné n>0 on a : n! = (n-1) !*n
On constate donc la définition de n! est récursive, puisqu'elle se réfère à elle même quand elle applique (n-1)!

Le calcul de la valeur factorielle d'un nombre est défini par :
a) Si n=0 ou n=1 alors n !=1
b) Si n>0 alors n != (n-1) !*n

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

Déroulement de fact (4)



Remarques :

a) Quand n=0, la valeur de n ! est donnée directement. 0 est appelé ***valeur de base***.

b) Pour un n≠0 donné, la valeur de n ! est définie en fonction d'une valeur plus petite que n et plus voisine de la valeur de base 0.

c) La variable n, est testée à chaque fois pour savoir s'il faut exécuter
Si n=0 alors n !=1
ou Si n>0 alors n != (n-1) !*n
n est appelé ***variable de commande***

Les principes de construction d'actions paramétrées récursives sont donc :

- Le nombre d'appels récursifs (niveaux) doit être fini. Il faut donc que les paramètres contiennent une ou plusieurs variables de commande qui sont testées à chaque niveau pour savoir si on doit continuer ou non les appels récursifs.
- Déterminer le ou les cas particuliers qui sont exécutées directement sans appels récursifs. Dans ces cas, les variables de commandes sont égales aux valeurs de base (n=0).
- Décomposer le problème initial en sous problèmes de même nature, telle que des décompositions successives aboutissent toujours à l'un des cas particuliers.
- Le principe de la récursivité est que les appels récursifs doivent être uniquement sur des données plus petites n != n* (n-1) !
Plus petit que n !

7.3 Schémas généraux d'actions récursives :

1er schéma :

```
Action P( )
  Début
    Si < condition> alors  P( )
                    sinon Q ;
  Fin ;
```

2ème schéma :

```
Action P( )
  Début
     Tant que  < condition> faire  P( )
     Q ;
  Fin ;
```

Où Q permet la résolution directe du problème (ne contient pas d'appel récursif).

7.4 Récursivité directe et récursivité indirecte :

Une action qui fait appel à elle-même explicitement dans sa définition est dite récursive directement ou récursivité simple.

Si une action A fait référence (ou appel) à une action B qui elle fait appel directement ou indirectement à A, on parle alors de récursivité indirecte ou récursivité croisée.

```
Action A( )
 Début
   Si < condition> alors  B( )
                  sinon QA ;
 Fin ;
```

```
Action B( )
  Début
   Si < condition> alors  A( )
                  sinon QB ;
  Fin ;
```

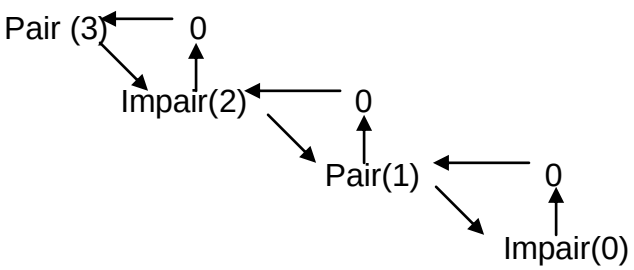
Exemple :

- Récursivité directe : <expr> → <expr> θ <expr>
- Récursivité indirecte :
  <expr> → <terme> θ <terme>
  <terme> → <expr>

- Pour construire une définition récursive de la parité, remarquons tout d'abord que 0 est un entier pair, et qu'il n'est pas impair. Ensuite remarquons que un entier n est pair (resp. impair) ssi l'entier n-1 est impair (resp. pair).

```
int  Pair (int n)
{  if (n= =0) return 1 ;
   else return Impair(n-1);
}
```

```
int Impair (int n)
{ if (n= =0) return 0 ;
  else return Pair(n-1);
}
```



Résumé :

```
Action récursive (paramètres)
  <Déclaration des variables locales> ;
Début
  Si (Test d'arrêt) alors < instructions du point d'arrêt >
  Sinon
      instructions ;
      récursive(paramètres changés) /* appels récursifs */
      instructions ;
   Fsi ;
Fin ;
```

7.5 Différents types de récursivité :

a) **Récursivité simple :** une fonction simplement récursive, c'est une fonction qui s'appelle elle-même une seule fois, comme c'était le cas pour la fonction factorielle.

b) **Récursivité multiples :** une fonction peut exécuter plusieurs appels récursifs – typiquement deux parfois plus.

Exemple :
```
void afficheMotRec (chaine TabMot[ ], int n, int  i)
{ if (i<n)  { afficheMotRec(TabMot, n, i+1) ;
              printf("%s", TabMot[i]);
              afficheMotRec(TabMot, n, i+1);
            }
}
```

c) **Récursivité à droite :**

Si l'exécution d'un appel récursif n'est jamais suivie par l'exécution d'une autre instruction, cet appel est dit récursif à droite ou encore appelée ***récursivité terminale***. L'exécution d'un tel appel termine l'exécution de l'action et ***ne nécessite pas une pile***.
Une fonction récursive ***non terminale nécessite une pile***.

Exemple :
```
void afficheMotRec1 (chaine TabMot[ ], int n, int  i)
{ if (i<n)  { printf("%s", TabMot[i]);
              afficheMotRec(TabMot, n, i+1);
            }
} /* afficheMotRec1 est une fonction récursive terminale */

void afficheMotRec2 (chaine TabMot[ ], int n, int  i)
{ if (i<n)  { afficheMotRec(TabMot, n, i+1) ;
              printf("%s", TabMot[i]);
            }
} /* afficheMotRec2 est fonction récursive non terminale */
```

7.6 Fonctionnement de la récursivité

Un programme ne peut s'exécuter que s'il est chargé en mémoire centrale, chaque instruction du programme se trouve à une adresse donnée de la mémoire.
Lorsqu'un programme fait appel à une fonction, le système sauvegarde l'adresse de retour (adresse de l'instruction qui suit l'appel), ainsi que les valeurs des variables locales.
Quand une fonction ***f*** appelle une fonction ***g,*** on doit sauvegarder l'adresse de retour de ***f*** (paramètres et variables locales) avant l'appel de ***g***, ce contexte doit être récupéré après le retour de ***g***.

S'il y a plusieurs appels imbriqués, le système gère ***une pile*** pour sauvegarder (empiler) les différents contextes des différents appels récursifs.

Les paramètres de l'appel récursif changent. A chaque appel les variables locales sont stockées dans une pile. Ensuite les paramètres ainsi que les variables locales sont désempilées au fur et à mesure qu'on remonte les niveaux.
Lors de l'i^{ème} appel sont empilés :

- Les valeurs des paramètres au niveau i
- Les valeurs des variables locales du niveau i
- L'adresse de retour au niveau i

A la fin des appels récursifs retour du niveau i+1 au niveau i :

- Retour au programme principal si la pile est vide
- Dépiler l'adresse de retour
- Dépiler le contexte du niveau i (les valeurs des variables du niveau i
- Exécuter l'instruction suivant le dernier appel

7.6 Elimination de la récursivité :

La récursivité ***simplifie la structure d'un programme*** mais la plupart du temps, le gain en simplicité vaut une baisse relative des performances d'exécution.
La récursivité est souvent couteuse en temps et en espace mémoire car elle nécessite l'emploi de techniques spéciales de compilation, à savoir ***le concept de pile***.

Ces techniques sont généralement plus ***coûteuses en temps d'exécution*** que celles fondées sur l'itération. Aussi certains langages de programmation n'admettent pas la récursivité (exemple : Fortran). Ainsi il arrive que l'on souhaite éliminer la récursivité.

A cet effet il est intéressant de noter que l'on peut montrer que, si le langage de programmation utilisé le permet, il est toujours possible de transformer une action itérative en une action récursive ; cependant, la réciproque n'est pas vraie.

Les problèmes qu'il faut résoudre en utilisant la récursivité sont les problèmes ***typiquement récursifs*** et non itératifs, c'est-à-dire, soit des problèmes qui ne peuvent pas être résolus de façon itérative, soit des problèmes pour lesquels une ***formulation*** récursive est particulièrement ***simple et naturelle.***

D'une manière générale, on évitera donc d'utiliser la récursivité lorsqu'on peut la remplacer par une définition itérative, à moins de bénéficier d'un gain considérable en simplicité.

Exemple :

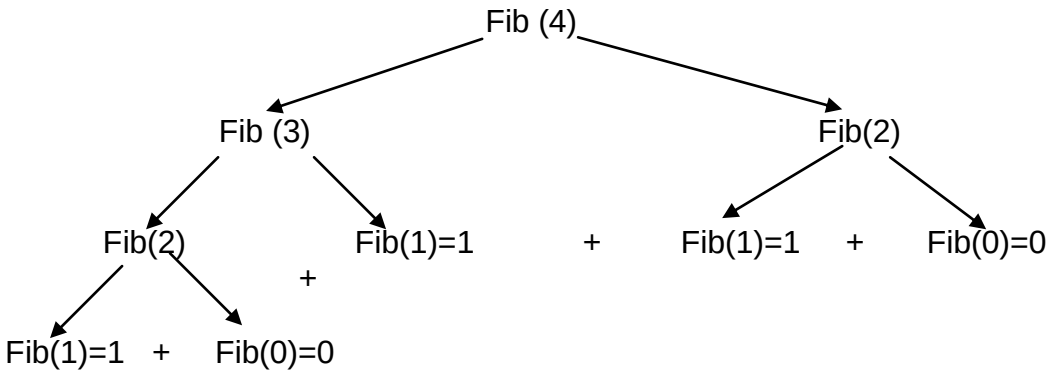
Les nombres de Fibonnaci : $F_0=0$, $F_1=1$

$$F_n=F_{n-1} + F_{n-2} \quad n>=2.$$

La fonction récursive permettant d'obtenir ces nombres est :

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

L'exécution de cette fonction récursive pour n=4 nous donne l'arbre suivant :



Fib(4)=3 (9 appels pour arriver au résultat)

Les valeurs successive de cette suite : 0, 1, 1, 2, 3, 5, 8, 12, 21, 34, 55,…

Voici maintenant la fonction itérative équivalente à la fonction récursive Fib.

```
int  Fib( int n)
{  int x, y, z, i ;
   x=1 ;  y=1 ; z=1 ; /* x=Fib(0)  et y= Fib(1) */
   for( i=2 ; i<=n ; i++)
     {  z=x+y ;
        x=y;
        y=z;
     }
   return z ;
}
```

Pour n=4 : x=1
y=1
i=2 : z=2 x=1 y=2
i=3 : z=3 x=2 y=3
i=4 : z=5 x=3 y=5
Résultat z=5

Le problème se situe au nombre d'appels à la fonction, nous constatons que pour la solution récursive le nombre d'appels est un nombre ***exponentiel*** (c'est une mauvaise solution très coûteuse) alors que la solution itérative ne coute que ***n appels***.

Cette version itérative peut à son tour se convertir en une nouvelle version récursive :

```
int  Fib(int x, int y, int n)
{ if (n= = 0 ||  n= =1)  return y;   else  return Fib(y, x+y, n-1);
}
```

n=4 : x=1, y=1
Fib(1,1,4) → Fib(1,2,3) → Fib(2,3,2) → Fib(3,5,1) = 5 donc Fib(4)=5

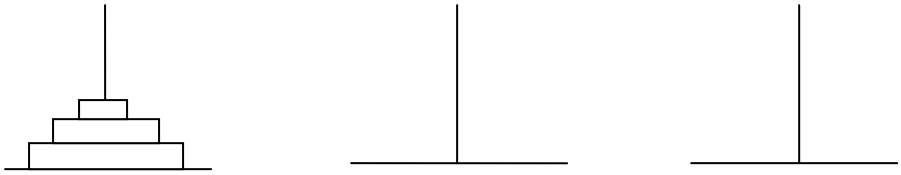
On a que 4 appels, le temps d'exécution est devenu ***linéaire***.

Comme on peut le constater, l'élimination de la récursivité est parfois très simple, elle revient à écrire une boucle, à condition d'avoir bien fait attention à l'exécution. Mais parfois elle est extrêmement difficile à mettre en œuvre.

Exemple : les tours de Hanoï

Le problème des tours de Hanoï consiste à déplacer N disques de diamètres différents d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.



Tour A | Tour B (intermédiaire) | Tour C (résultat)

7.5.1 Elimination de la récursivité terminale

Un algorithme est dit récursif terminal (ou récursif à droite) s'il ne contient aucun traitement après un appel récursif.
- Dans ce cas le contexte de la fonction n'est pas empilé.
- L'appel récursif sera remplacé par une boucle while.

Cas1 :

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x));}
}
```

```
f(x) /* itrérative*/
{ while(condition(x)) { A ; x=g(x) ;}
}
```

Cas2 :

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x));}
  else  B ;
}
```

```
f(x) /* itrérative*/
{ while(condition(x)) { A ; x=g(x) ;}
  B ;
}
```

7.5.2 Elimination de la récursivité non terminale

a) cas d'un seul appel récursif:

Ici pour pouvoir dérécursiver, il va falloir sauvegarder le contexte de l'appel récursif.

Cas1 :

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x));}   → nécessite une pile
  B ;
}
```

```
f(x) /* itrérative*/
{ pile p=initpile();
  while(condition(x)) { A ; empiler(&p,x); x=g(x) ;}
  while(!pilevide(p)) {desempiler(&p,&x) ;B ;}
}
```

Cas2 :

```
f(x) /* récursive*/
{ if (condition(x))
      {A1 ; f(g(x)); A2 ;}
  else B ;
}
```

```
f(x) /* itrérative*/
{ pile p=initpile();
  while(condition(x))
       { A1 ; empiler(&p,x); x=g(x) ;}
  B;
  while(!pilevide(p)) {desempiler(&p,&x) ;A2 ;}
}
```

b) cas de deux appels récursifs:

Le 2ème appel est récursif à droite (terminal)

```
f(x) /* récursive*/
{ if (condition(x))  {A ; f(g(x)); f(h(x)) ;}
 }
```

Si on élimine le 2ème appel :

```
while(condition(x)) {A ; f(g(x)) ; x=h(x) ;}
```

Le schéma itératif équivalent à f est :

```
f(x) /* itérative*/
{ pile p=initpile();
  while(condition(x))
   { while(condition(x) { A(x) ; empiler(&p,x); x=g(x) ;}
     desempiler(&p,&x) ;x=h(x) ;
   }
}
```

ALGORITHME Q(U)
si *C*(*U*) alors *D*(*U*);Q(a(*U*));*F*(*U*)
sinon *T*(*U*)

ALGORITHME Q'(U)
empiler(nouvel_appel, U)
tant que pile non vide faire
dépiler(état, V)
si état = nouvel_appel alors $U \leftarrow V$
si C(U) alors D(U)
empiler(fin, U)
empiler(nouvel_appel, a(U))
sinon T(U)
si état = fin alors $U \leftarrow V$
F(U)

Exercices :

1) Déroulez les fonctions calcul1 et calcul2, que constatez-vous ?

```
float calcul1 (int n)
{ if (n==0) return (2) ;
  else return(1/2(calcul1(n-1)+2)) ;
}
```

C'est une fonction qui se termine.

```
float calcul2 (int n)
{ if (n==0) return (2) ;
  else return(1/2(calcul2(n-2)+2)) ;
}
```

calcul2 ne se termine pas si ***n est impair***
calcul2 se termine si ***n est pair***

$\forall$ n le résultat de calcul1 est égal à 2
$\forall$ n pair le résultat de calcul2 est égal à 2

2) Ecrire une fonction itérative puis récursive qui calcul la somme des n premiers nombres.

- ```
  int sommeIter (int n)
  {  int i, s=0 ;
     for(i=1 ;i<=n ; i++)
      s=s+i ;
    return s ;
  }
  ```

- ```
  int sommeIter (int n)
  {  int i, s=0 ;
     for(i=n ;i>0 ; i--)
      s=s+i ;
    return s ;
  }
  ```

- ```
  int sommeRec(int n)
  {  if (n==0) return 0 ;
     else  return(n+somme(n-1)) ;
  }
  ```

Remarque: Le concept de récursivité est spécialement mis en valeur dans les définitions mathématiques. Les mathématiques utilisent plutôt le mot récurrence.

3) Le plus grand commun diviseur (pgcd):
Le pgcd de deux entiers A et B est le plus grand entier qui divise à la fois A et B.

```
int pgcdRec(int A, int B)
{
  if (B==0) return A ;
  else  return( pgcd(B, A%B);
}
```

```
int pgcdIter(int A, int B)
{ int reste;
  while (B !=0)
    { reste=A%B ;
      A=B ;  B=reste;
    }
 return(A);
}
```
```

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define max 50
typedef char chaine[max];
```

int palind(chaine t,int i,int j)

```
{ printf("palindrom i= %d  j= %d \n",i,j);
  if(i>=j) return(1);
  else if (t[i]==t[j]) return(palind(t,i+1,j-1));
       else return(0);
}
main()
{ chaine ch; int res, l;
  printf("donnez un mot\n");
  scanf("%s",ch); l=strlen(ch)-1;
  res=palind(ch,0,l);
  if (res==0) printf("%s n'est pas un mot palindrome\n",ch);
  else printf("%s est un mot palindrome\n",ch);
  getch();
}
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define max 50
typedef char chaine[max];
```

void Inverser(chaine t,int i,int j)

```
{char x;
  if (i<j) { x=t[i];t[i]=t[j],t[j]=x;Inverser(t,i+1,j-1);}
}

main()
{ chaine ch; int l;
  printf("donnez un mot\n");
  scanf("%s",ch); l=strlen(ch)-1;
  Inverser(ch,0,l);
  printf("\n apres inversion : %s",ch);
  getch();
}
```

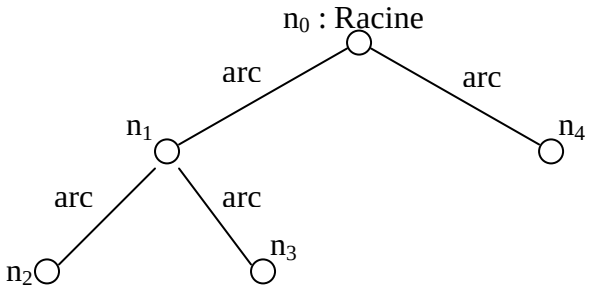
Chapitre 8 : Les Arbres

8.1 Définitions :

a) Arbre :

L'arbre est une structure de donnée récursive constituée :

- d'un ensemble de points appelés ***nœuds,***
- d'un nœud particulier appelé ***racine***,
- d'un ensemble de couples (n_1, n_2) reliant le nœud n_1 au nœud n_2 appelés ***arcs*** (ou arêtes). Le nœud n_1 est appelé ***père*** de n_2. Le nœud n_2 est appelé ***fils*** de n_1.



b) Feuilles :

Les nœuds qui n'ont aucun fils sont appelés feuilles ou nœuds terminaux (les nœuds n_2 , n_3 et n_4 sont des feuilles).

c) Chemin :

On appelle chemin la suite de nœuds n_0 n_1 … n_k telle que (n_{i-1} , n_i) est un arc pour tout i ϵ { 0,…,k}. L'entier k est appelé longueur du chemin n_0 n_1 … n_k .
k c'est aussi le nombre d'arcs.

Le nombre d'arcs d'un arbre = nombre de nœuds - 1.

d) Sous-arbre :

Les autres nœuds (sauf la racine n_0) sont constitués de nœuds fils, qui sont eux même des arbres. Ces arbres sont appelés sous-arbres de la racine.
Exemple : Les nœuds n_1 n_2 et n_3 constituent un sous-arbre.

e) Hauteur :

La hauteur d'un nœud est la longueur du plus long chemin allant de ce nœud jusqu'à une feuille.
La hauteur d'un arbre est la hauteur de la racine (nombre de nœuds).

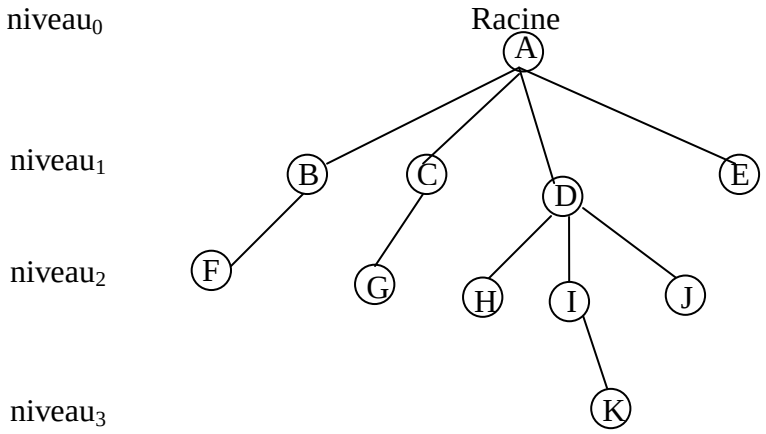
f) Niveau ou profondeur :

La profondeur d'un nœud est la longueur du chemin allant de la racine jusqu'à ce nœud. Tous les nœuds d'un arbre de même profondeur sont au même niveau.
Exemple : Les nœuds n_1 et n_4 ont la même profondeur et sont donc au même niveau.

g) Ascendance et Descendance :

Soit un nœud ***a*** et un nœud ***b*** s'il existe un chemin du nœud ***a*** au nœud ***b*** on dit que ***a*** est un ascendant de ***b*** ou que ***b*** est un descendant de ***a***.

Exemple récapitulatif:

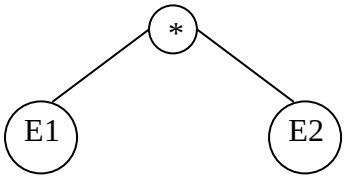


La racine c'est : A
Les nœuds fils de A sont : B C D E
Le nombre de sous-arbres = 4
Le père de F c'est B
B est un ascendant de F
F est un descendant de B
Les feuilles de l'arbre sont : F G H K J E
La hauteur de l'arbre = 4 (nombre de nœuds)
La longueur du chemin A-F = 2 (nombre d'arcs)
La profondeur de l'arbre = 3
La profondeur du nœud G = 2

Un arbre peut aussi être représenté sous forme parenthésée :
(A (B(F), C(G), D(H, I(K), J), E))

h) Arbre étiqueté:

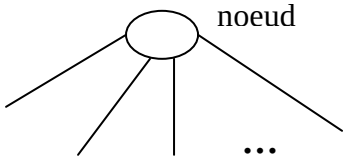
Un arbre étiqueté est un arbre dont chaque nœud possède une information ou étiquette. Cette étiquette peut être de nature très variée : entier, réel, caractère, chaine… ou une structure complexe.
Exemple : on peut représenter une expression par un arbre



E1* E2

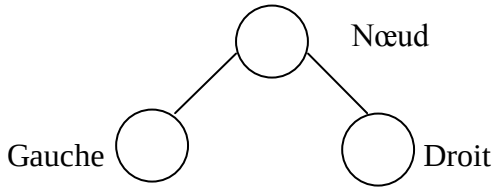
i) Arbre n-aire :

Un arbre n-aire est un arbre dont les nœuds ont au plus n successeurs.



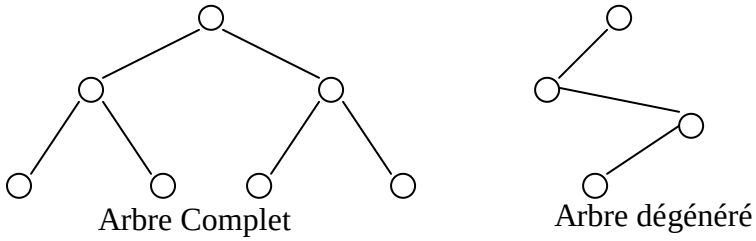
ii) Arbre binaire :

Un arbre binaire est dit binaire si tout nœud de l'arbre a 0, 1, ou 2 successeurs. Ces successeurs sont alors appelés respectivement successeur gauche et successeur droit.



Lorsque tous les nœuds d'un arbre binaire ont deux ou zéro successeurs ont dit que l'arbre est ***homogène*** ou ***complet***

Un arbre binaire est dit ***dégénéré*** si tous ses nœuds n'ont qu'un seul descendant.



Un arbre complet de hauteur ***h*** a un nombre de nœud = 2^h -1 et le nombre de feuilles est $2^{(h-1)}$.

Exemple : *h*=3 nombre de nœuds = 2^3-1=7 nombre de feuilles = $2^{(3-1)}$ = 4

iii) Arbre binaire équilibré: C'est un arbre binaire tel que les hauteurs des deux sous arbres SAG, SAD (sous arbre gauche, sous arbre droit) de tout nœud de l'arbre diffèrent de 1 au plus. Ou encore le nombre de nœuds de SAG et le nombre de nœuds du SAD diffèrent au maximum de 1.

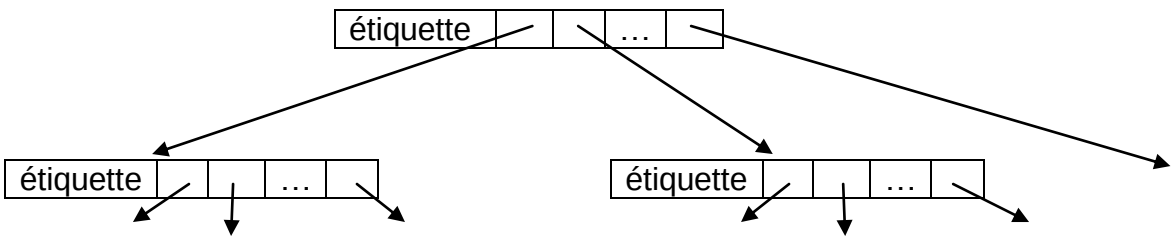
8.2 Représentation des arbres

8.2.1 Arbre n-aire :

a) Une manière de représenter un arbre est d'associer à chaque nœud un enregistrement contenant un ou plusieurs champs pour coder l'étiquette et d'un tableau de pointeurs vers les nœuds fils. La taille du tableau est donnée par le nombre maximum de fils des nœuds de l'arbre.

étiquette	P1	P2	…	P3

Déclaration : typedef struct no *arbre ;
typedef struct no { arbre tab[max_fils] ; /* tableau de pointeurs sur des arbres*/
typelem étiquette ; } nœud ;

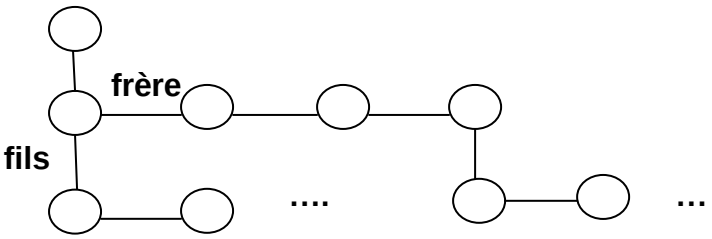


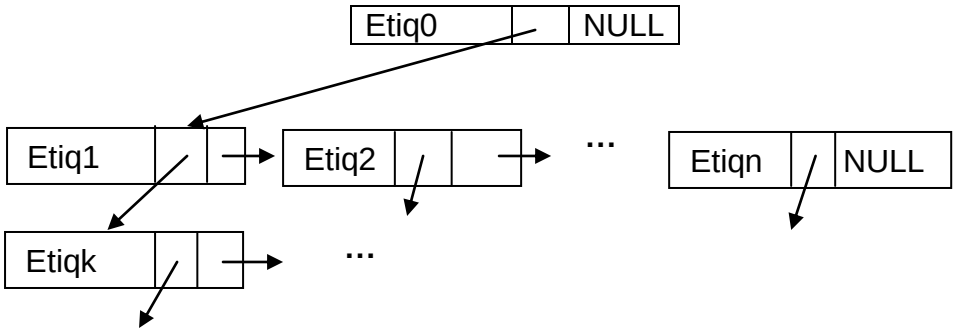
Inconvénients :

- l'arbre contient un petit nombre de nœuds ayant beaucoup de fils. (tableaux de grandes tailles)
- L'arbre contient beaucoup de nœuds ayant peu de fils. (plusieurs tableaux) Ceci conduit à consommer beaucoup d'espace mémoire.

b) Avec deux pointeurs fils et frère.
Afin de contourner l'inconvénient du tableau, on utilise un pointeur vers fils ainée et chaque fils possède un lien vers son frère le plus proche.

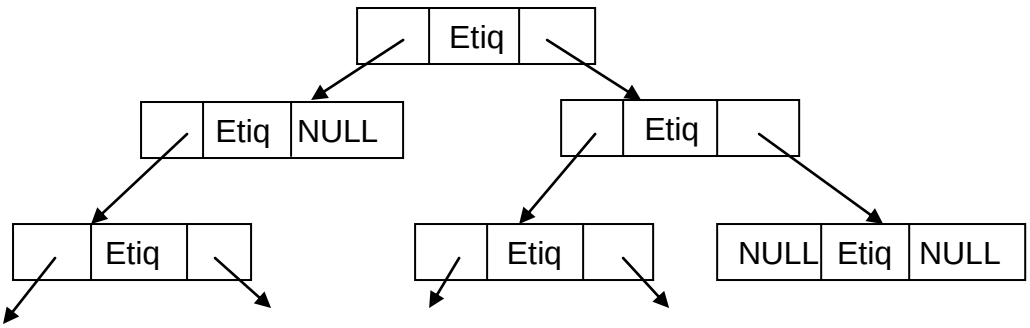
Déclaration : typedef struct no *arbre ;
typedef struct no { arbre fils, frère;
typelem étiquette ; } nœud ;





8.2.2 Arbre binaire :

Déclaration : typedef struct no *arbre ;
typedef struct no { arbre gauche, droit ;
typelem étiquette ; } nœud ;



8.3 Parcours d'un arbre

8.3.1 Parcours d'un arbre n-aire

a) En préordre : à partir d'un nœud quelconque on effectue :

- quelque chose sur ce nœud
- l'ensemble des opérations sur le fils ainé
- « « « « « suivant
- …
- l'ensemble des opérations sur le dernier fils.

b) En postordre : à partir d'un nœud quelconque on effectue :

- l'ensemble des opérations sur le fils ainé
- « « « « « suivant
- …
- l'ensemble des opérations sur le dernier fils.
- quelque chose sur ce nœud

8.3.2 Parcours d'un arbre binaire

a) ***En préordre*** (préfixe) : Racine - Fils gauche - Fils droit (RAC - SAG - SAD[1] ou bien RAC - SAD - SAG).

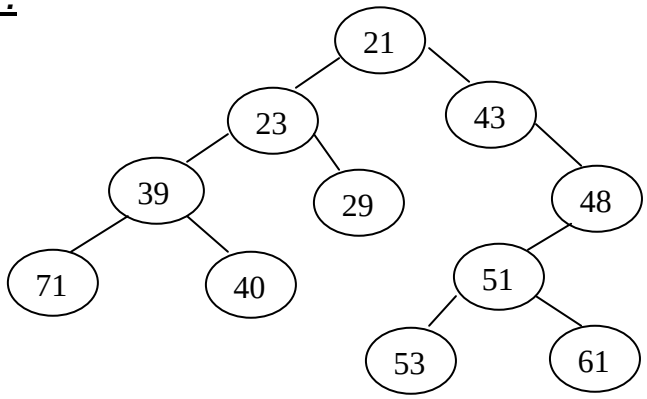
b) ***En postordre*** (Postfixe) : SAG – SAD - RAC ou bien SAD - SAG - RAC.

c) ***En ordre*** (infixe): SAG - RAC - SAD ou bien SAD - RAC - SAG (*arbre binaire uniquement)*

8.4 Arbre binaire ordonné

a) Verticalement : Un arbre binaire est ordonné verticalement, si la clé de tout nœud non feuille est inférieure (respectivement supérieure) à celle de ses fils (et donc par récurrence à celle de tous ses descendants).

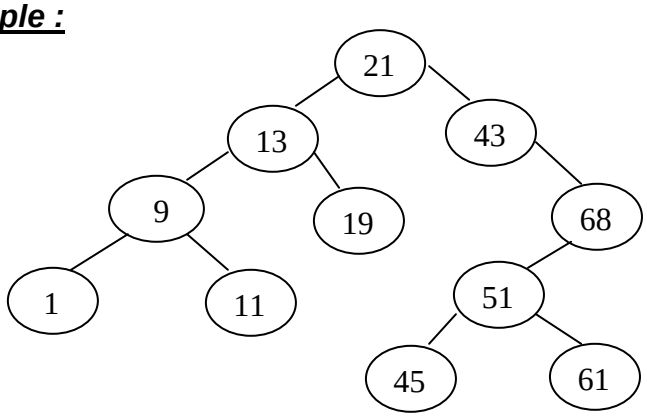
Exemple :



[1] RAC : Racine SAG : Sous Arbre Gauche SAD : Sous Arbre Droit

a) Horizontalement : Un arbre binaire est ordonné horizontalement (de gauche à droite) si la clé de tout nœud non feuille est supérieure ou égale à toutes celles de son sous arbre gauche et inférieure ou égale à toutes celles de son sous arbre droit. Ce type d'arbre est appelé aussi ***arbre binaire de recherche***.

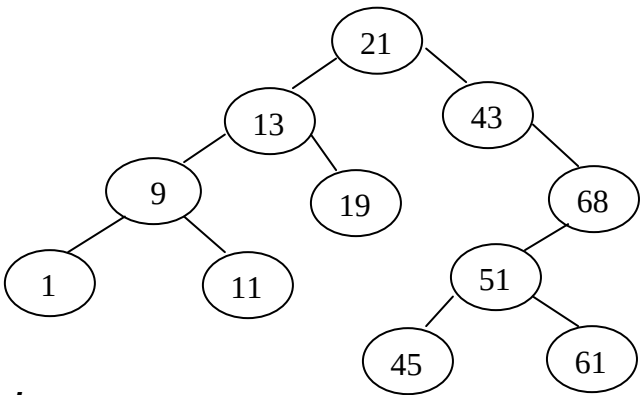
Exemple :



Inconvénient : Un arbre binaire est ordonné horizontalement on a affaire à une relation d'ordre total (on vérifie facilement que l'arbre est totalement ordonné dans un parcours en ordre, la notion d'ordre vertical est seulement une notion d'ordre partiel (il n'y a pas d'ordre à priori entre deux nœuds frères, (ou plus généralement entre deux nœuds d'un même niveau) c'est pourquoi la recherche dans un arbre binaire ordonné verticalement n'est pas dichotomique et peut conduire à un parcours exhaustif de l'arbre.

8.5 Arbre Binaire de Recherche (ABR)

8.5.1 Exemple de parcours d'un ABR:



a) En Préordre :

21 – 13 – 9 – 1 – 11 – 19 – 43 – 68 – 51 – 45 – 61

b) En Postordre :

1 – 11 – 9 – 19 – 13 – 45 – 61 – 51 – 68 – 43 – 21

c) En Ordre :

1 – 9 – 11 – 13 – 19 – 21 – 43 – 45 – 51 – 61 - 68

8.5.1.1 Fonctions récursives de parcours d'un ABR

a) Préordre (Préfixé)

```
void  parcours (arbre A)
{ if (A !=NULL)
    {  afficher(A→etiquette) ;
       parcours(A→gauche) ;
       parcours(A→ droit);
    }
}
```

b) En ordre (Infixé)

```
void  parcours (arbre A)
{ if (A !=NULL)
    {  parcours(A→gauche) ;
       afficher(A→etiquette) ;
       parcours(A→ droit);
    }
}
```

c) En postordre (postfixé)

```
void  parcours (arbre A)
{ if (A !=NULL)
    {  parcours(A→gauche) ;
       parcours(A→ droit);
       afficher(A→etiquette) ;
    }
}
```

8.5.1.2 Fonctions itératives de parcours d'un arbre de recherche

a) Préordre (Préfixé)

```
void  parcours (arbre A)
{ pile s=initpile() ;
  empiler(&s,NULL) ;
  while (A !=NULL)
    {  afficher(A→etiquette) ;
       if (A→droit !=NULL) empiler(&s,A→droit);
       if (A→gauche!=NULL) A=A→gauche;
       else desempiler(&s,&A) ;
    }
}
```

b) En ordre (Infixé)

```
void  parcours (arbre A)
{ pile s=initpile() ;
  while (A !=NULL) { empiler(&s,A); A=A→gauche; }
  while (!pilevide(s))
    {  desempiler(&s, &A) ;
       afficher(A→etiquette) ;
       if (A→droit !=NULL)
         { A=A→droit ;
           while( A !=NULL) { empiler(&s,A) ; A=A→gauche);  }
         }
    }
}
```

c) En postordre (postfixé)

```
void  parcours (arbre A)
{ pile pg=initpile(), pd=initpile() ;
  while (A !=NULL)
   {  empiler(&pg,A);
      if (A->droit !=NULL)
           { empiler(&pg,NULL); empiler(&pd, A->droit) ; }
      A=A->gauche;
   }
  while (!pilevide(pg))
   {  desempiler(&pg, &A) ;
      if (A!=NULL) printf("%d  ",A->etiquette) ;
      else { if(!pilevide(pd))
                { desempiler(&pd,&A);
                     if ( A->gauche ==NULL && A->droit ==NULL)
                                printf("%d  ", A->etiquette);
                     else  while( A !=NULL)
                                { empiler(&pg,A) ;
                                  if (A->droit !=NULL)
                                       { empiler(&pg,NULL);empiler(&pd,A->droit) ;}
                                  A=A->gauche;
                                }
                }
             }
   }
}
```

8.5.2 Recherche dans un ABR

a) Fonction recherche itérative :

```
int recherche(arbre A, typelem val)
{  int trouv=0 ;
   while(trouv==0 && A !=NULL)
    if (val==A→etiquette) trouv=1;
    else if (val <A→etiquette) A= A→gauche ;
           else A=A→droite ;
  return (trouv) ;
}
```

b) Fonction recherche récursive

```
int recherche(arbre A, typelem val)
{  If (A ==NULL) return 0 ;
   else if (val==A→etiquette) return 1;
        else if (val <A→etiquette) return recherche(A→gauche,val);
             else return recherche(A→droite,val) ;
}
```

8.5.3 Insertion d'un élément dans un ABR

Remarques : - Les étiquettes dans un arbre binaire ordonné horizontalement sont toujours uniques (elles ne sont pas dupliquées), donc l'élément à insérer est toujours une feuille.
- On utilise une autre fonction de recherche qui retourne (en paramètre) l'adresse de l'élément précédent.

```
int Recherche(arbre A, arbre *prd, typelem Val)
{  if (A==NULL) return(0);
   else { if (A->etiquette==Val) return(1);
          else { if (A->etiquette>Val) return(Recherche(A->gauche, &A ,Val));
                 else return (Recherche(A->droit, &A ,Val));
               }
        }
}
```

```
void Insert (arbre *racine,typelem Val)
{ arbre prd=NULL, A;
    If (Recherche(*racine,&prd,Val)==1) printf("Existe deja\n");
    else { A=(arbre)malloc(sizeof(noeud));
           A->etiquette=Val;   A->gauche=NULL;  A->droit=NULL;
           if (prd!=NULL)
                 if (Val<prd->etiquette) prd->gauche=A;
                 else prd->droit=A;
           else *racine=A; /* Quand l'arbre est vide  */
         }
}
```

8.5.4 Suppression d'un élément dans un ABR

Trois types de suppressions se présentent à nous :

a) Si l'élément est une feuille, alors on le supprime simplement. On a deux cas :

a.1) supprimer une feuille qui se trouve à gauche d'un nœud.
Exemple : prd->gauche=NULL ; free(A) ;

a.2) supprimer une feuille qui se trouve à droite d'un nœud.
Exemple : prd->droit=NULL ; free(A) ;

b) Si l'élément n' a qu'un seul descendant, alors on le remplace par ce descendant. On a quatre cas :

b.1) Exemple : prd->droit=A->droit ; free(A) ;

b.2) Exemple : prd->droit=A->gauche ; free(A) ;

b.3) Exemple : prd->gauche=A->droit ;free(A) ;

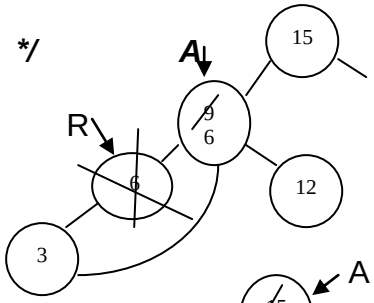
b.4) Exemple : prd->gauche=A->gauche ; free(A) ;

c) Si l'élément a deux descendants, on le remplace au choix soit par :
- L'élément le plus à droite du SAG (la valeur max)
- L'élément le plus à gauche du SAD (la valeur min)

On a deux cas :

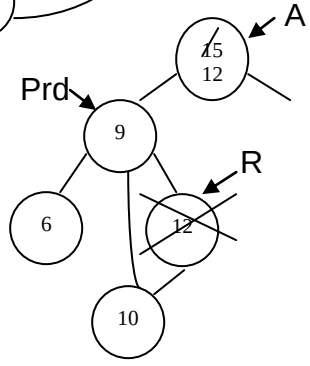
c.1) Exemple : ***/* R l'élément le plus à gauche */***

A->etiquette = R->etiquette ;
A->gauche=R->gauche ;
free(R) ;



c.2) Exemple :

A->etiquette = R->etiquette ;
prd->droit=R->gauche ;
free(R) ;



```
void suppFeuille(arbre prd, arbre A)
{   if (Val<prd->etiquette) prd->gauche=NULL;
    else prd->droit=NULL;
    free(A);
 }

void supp1Fils(arbre prd, arbre A)
{ if (A->gauche==NULL)
          if (Val<prd->etiquette) prd->gauche=A->droit;
          else prd->droit=A->droit;

  else  if (Val<prd->etiquette)  prd->gauche=A->gauche;
         else prd->droit=A->gauche;

  free(*A) ; *A=NULL;
 }

void remplace(arbre R, arbre A, arbre prd)                    /* 2 descendants */
{ if (R->droit!=NULL)  { prd=R;  remplace(R->droit, A, prd); }
  else { A->etiquette=R->etiquette;
           if (prd!=NULL) prd->droit=R->gauche;
           else A->gauche=R->gauche;
           free(R);
         }
}

void Supprim(arbre *racine, typelem Val)
{ arbre A, prd=NULL;

  if (Recherche(*racine, &prd, Val)==0)
                printf(" L'élément à supprimer n'existe pas\n");

  else {  if (val<prd->etiquette)    A=prd->gauche;   else A=prd->droit;

             /* Racine */
           if (prd==NULL && A->gauche==NULL && A->droit==NULL)
                { free(A);  printf("Arbre Vide\n");  *racine=NULL;  }

           else   /*  Feuille  */
                  if (A->gauche==NULL && A->droit==NULL)  suppFeuille(prd, A) ;

                       /* 1 seul descendant */
                  else if (A->gauche==NULL)  || (A->droit==NULL) supp1Fils(prd, A) ;

                            /* 2 descendants */
                       else remplace(A->gauche, A, NULL);
                                                  ↖
         }                                          Sous arbre gauche
}
```

8.5.5 Construction d'un arbre binaire de recherche

```
void constarbre(tab t, int i, int n, arbre *racine)
{
    if (i<n) { Insert(racine, t[i]);
               constarbre(t, i+1, n, racine);
             }
}
```

/ Autre fonction supprime */*

```
void Supprim(arbre *racine, typelem Val)          L'adresse de VAL l'élément
{ arbre A, prd=NULL;                                  à supprimer
  if (Recherche(*racine, &prd, Val, &A)==0)
              printf(" L'élément à supprimer n'existe pas\n");
  else { if (prd==NULL && A->gauche==NULL && A->droit==NULL) /* Racine */
              { free(A);  printf("Arbre Vide\n");  *racine=NULL;}
         else {   /*  Feuille  */
                  if (A->gauche==NULL && A->droit==NULL)
                    { if (Val<prd->etiquette) prd->gauche=NULL;
                      else prd->droit=NULL;
                      free(A);
                    }
                  else if (A->gauche==NULL)  /* 1 seul descendant */
                         { if (Val<prd->etiquette) prd->gauche=A->droit;
                           else prd->droit=A->droit;
                           free(A);
                         }
                       else if (A->droit==NULL)  /* 1 seul descendant  */
                              { if (Val<prd->etiquette)prd->gauche=A->gauche;
                                else prd->droit=A->gauche;
                                free(A);
                              }
                            else remplace(A->gauche, A, NULL);
              }                                          Sous arbre gauche
       }
}
```