الكلية متعددة التخصصات – ورزازات
ⵜⴰⵖⵡⴰⵍⵜ ⵜⴰⵅⵜⵉⵙⵙⵉⵜ – ⵡⵓⵔⵣⴰⵣⴰⵜ
FACULTÉ POLYDISCIPLINAIRE DE OUARZAZATE

UNIVERSITÉ IBN ZOHR - AGADIR

# Programmation: II

# - Le Langage C –

Filière: SMI: S4

Prof: KRIT Salah-ddine

# Rappels et compléments de C:
# Type et variable pointeurs

❖ **Appel de pointeur:**

La fonction **malloc()** (pour l'anglais Memory allocation, c-`a-d allocation de mémoire) permet l'appel de pointeur. La déclaration de cette fonction se trouve dans le fichier

en-tête **stdlib.h** (pour STandarD LIBrary).

Syntaxe.- La syntaxe de cette fonction est :

**ptr = malloc(entier);**

ou **entier** est une expression entière (positive) et ptr un pointeur.

❖ **Libératioin des pointeurs:**

La libération d'un pointeur se fait en langage C grâce `a la fonction **free().**

La déclaration de cette fonction se trouve également dans le fichier en-tête **stdlib.h**.

**Syntaxe:** La syntaxe de cette fonction est :

**free(ptr) ;**

# Rappels et compléments de C:
# TYPE STRUCTURES

Définition:

➢Une structure rassemble des variables, qui peuvent être de types différents, sous un seul nom ce qui permet de les manipuler facilement. Elle permet de **simplifier l'écriture d'un programme en regroupant des données liées entre elles**.

Un exemple On peut représenter un nombre complexe a l'aide d'une structure.

➢On définit une structure a l'aide du mot-clé: **struct** suivi d'un identificateur (nom de la **structure**) et de la liste des variables qu'elle doit contenir (type et identificateur de chaque variable).

Cette liste est délimitée par des accolades. Chaque variable contenue dans la structure est appelée un champ ou un membre.

**Exemple de définir une structure:**

```
/* Structure personne, composée de deux champs : cNom et iAge */
struct personne
{
char* cNom;
int iAge;
};
/* Déclaration d'un élément individu de type struct personne */
struct personne individu;
```

# ACCÈS À UN ÉLÉMENT D'UNE STRUCTURE

❖ Pour accéder aux champs d'une structure on utilise un point. Et pour accéder à ces mêmes champs à l'aide d'un pointeur sur une structure on utilise une **combinaison du point ou de l'étoile** (voir exemple) ou **une flèche**.

**Exemple d'implimentation :**

```
/* Structure personne, composée de deux champs
:
Nom et Age */
struct personne
{
char* Nom;
int Age;
};
int main()
{
/* Déclaration d'un élément individu de type struct
personne */
/* et d'un pointeur pindividu sur la structure */
struct personne individu;
struct personne *pindividu=NULL;
…
```

```
/* Accès au nom de individu (variable
de type struct personne)*/
printf ("%s", individu.Nom);
/* Accès à l'âge de individu */
printf("%d",individu.Age);
/* Deux méthode d'accès au nom
de pindividu*/
/* (pointeur sur une structure) */
/* 1 : les parenthèses sont importantes
 car . a priorité sur * */
printf("%s",(*pindividu).Nom);
/* 2 : Il est préférable d'utiliser
 cette méthode */
printf("%s", pindividu->Nom);
}
```

# ALLOCATION MÉMOIRE DES POINTEURS SUR LES STRUCTURES

- Il faut toujours faire attention à bien allouer le pointeur sur la structure, ainsi que les champs de la structure, s'il s'agit de pointeurs.

```
/* Allocation mémoire du pointeur pindividu */
/* sur la structure personne */
pindividu = (struct personne*)malloc(sizeof(struct
personne));
/* Allocation mémoire des champs de la structure */
(*pindividu).cNom = (char*)malloc(100);
```

# Rappels et compléments de C:
# Les Structures

- Un exemple d'utilisation : **tableau de pointeurs**

❖ **Problématique:**

On veut disposer en mémoire centrale d'un répertoire téléphonique. Chaque composant de ce répertoire comportera un **nom et un numéro de téléphone**.

❖ **Analyse du problème:**

Les structures de données à lesquelles on peut penser pour ce programme sont un **type structuré** avec les champs **nom et téléphone**, et **un tableau répertoire** dont les éléments sont de ce type **structuré**.

Il faut décider d'une dimension maximum pour ce tableau, par exemple 100.

❖ **Inconvénients:**

Cette façon de faire réserve beaucoup de place mémoire, ce qui risque de laisser peu de place pour faire autre chose. Ceci peut, par exemple, ralentir considérablement une impression dans le cas d'une utilisation de l'ordinateur en mode multitâche.

❖ **Utilisation d'un tableau de pointeurs:**

Une autre façon permettant d'économiser la mémoire non réellement nécessaire, est d'utiliser un tableau dont le type d'un élément est, non pas une structure, mais un type pointeur sur une telle structure. On ne fera pointer de façon effective que pour les emplacements nécessaires.

# DEFINITION DE TYPE

- En C, on peut déclarer les types en utilisant le mot clé **typedef.**
- **Exemples de type:**

```
/* Type d'un pointeur sur un entier */
typedef int *pointentier;
/* Déclaration d'une variable de type pointeur sur un entier */
pointentier a = NULL;              /* a est un pointeur sur un entier */
/* Déclaration d'un type tableau de 10 entiers */
typedef int tableauentiers[10];
/* Déclaration d'un tableau de 10 entiers */
tableauentiers Tableau;
/* Déclaration d'un type pointeur sur un tableau de 10 entiers */
typedef int (*tableauIpointeurs)[10];
/* Déclaration d'un pointeur sur un tableau de entiers */
tableauIpointeurs TabPointeurs;
```

# DEFINITION DE TYPE

- **Voici un exemple d'implimentation de programme manipulant les pointeurs sur tableau d'entiers et les tableaux de pointeurs sur des entiers :**

```
#include <stdio.h>
/* définition du type pointeur sur un tableau de
10 entiers */
typedef int (*tptab)[10];
int main()
{
/* déclaration d'un tableau de 10 pointeurs sur
entier */
int *tpointeurs[10];
/* déclaration d'un pointeur sur un tableau de
10 entiers */
tptab ptab;
int i =0;
/* déclaration et initialisation d'un tableau de
10 entiers */

int iTab[10]={1,5,8,49,5,6,7,3,-1,5};
```

```
/* tableau_de_pointeurs est un tableau statique
donc inutile d'allouer de la mémoire pour le
tableau  mais il faut allouer de la mémoire pour
chaque pointeur du tableau */
for(i=0; i<10; i++)
tpointeurs[i]=(int*)malloc(sizeof(int));
/* Affectation d'une valeur à chaque élément du
tableau */
for(i=0; i<10; i++)
*(tpointeurs[i])=i;
printf("Affichage des éléments du tableau de
pointeurs\n");
/* Affichage des éléments du tableau */
for(i=0; i<10; i++)
printf("le %d Element du tableau a pour valeur :
%d \n",i ,*(tpointeurs[i]));
```

# Rappels et compléments de C: Les Structures

**Un autre exemple d'implimentation en C:** Ceci nous conduit au programme suivant, pour l'initialisation de ce répertoire (en s' arrêtant sur un nom commençant par '#') puis pour l'affichage des données ainsi initialisées :

```
/* Programme_malloc.c */
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
struct donnee
{
char nom[30];
char tel[20];
};
struct donnee *repertoire[100];
int i, j;
/* initialisation */
i = -1;
Do{
printf("\nNom : ");
scanf("%s", individu.nom);
if (individu.nom[0] != '#')
{
printf("Telephone : ");
scanf("%s", individu.tel);
i++;
 repertoire[i] = (struct donnee *) malloc(50);
*(repertoire[i]) = individu;
};
}
while ((individu.nom[0] != '#') && (i < 100));
/* affichage */
for (j = 0; j <= i; j++)
printf("%s : %s\n", repertoire[j]->nom,
repertoire[j]->tel);
}
```

# Rappels et compléments de C: Les Structures auto-référentes

- **Définition:**

Une structure auto-référente est une structure dans laquelle un ou plusieurs champs ont pour type un pointeur sur cette structure.

- **Un exemple d'implimentation:**

Le programme suivant crée une liste chainée dont les éléments sont des structures comprenant un nom, un numéro de téléphone et un pointeur permettant d'accéder à l'élément suivant.

Le répertoire lui-même est un pointeur, de nom **debut**, permettant d'accéder au premier élément.

Pour simplifier, la liste est créée en partant du dernier élément, et non pas du premier.

On affiche, dans une seconde étape, les valeurs de la liste. Celle-ci sera affichée dans l'ordre inverse de l'ordre d'entrée.

# Rappels et compléments de C: Les Structures auto-référentes

```
•/* liste_1.c */
 #include <stdio.h>
 # include <stdlib.h>
 #include <string.h>
 void main(void)
 {
 struct donnee
 {
 char nom[30];
 char tel[20];
 struct donnee *suivant;
 };
 struct donnee item, *debut, *ptr;
 char name[30], telephone[20];
 /* Initialisation de la liste */
 debut = NULL;
 do
 {
 printf("\n Nom : ");
 scanf("%s",name);
 If (name[0] != '#')
 {
 printf("Telephone : ");
 scanf("%s",telephone);
 ptr = (struct donnee *) malloc(sizeof(item));
 strcpy(item.nom, name);
 strcpy(item.tel,telephone);
 item.suivant = debut;
 *ptr = item;
 debut = ptr;
 }}
 while (name[0] != '#');
 /* STRUCTURES AUTO-REFERENTES */
 /* Affichage de la liste */
 printf("\nLa liste dans l\'ordre inverse est :\n");
 while (debut != NULL)
 {
 printf("%s : %s\n",debut->nom, debut->tel);
 debut = debut->suivant;
 }
 /* Liberation des pointeurs */
 while (debut != NULL)
```

# Les fonctions

## ❖ Définition d'une fonction

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme:

***type nom-fonction ( type-1 arg-1,..., type-n arg-n)***

***{***

**[** ***déclarations de variables locales ]***

***liste d'instructions***

***}***

Le corps de la fonction débute éventuellement par **des déclarations de variables**, qui sont **locales** à cette fonction. Il se termine par **l'*instruction de retour à*** *la fonction appelante,* **return**, dont la syntaxe est:

**return(*expression);***

**Voici quelques exemples de définitions de fonctions :**

```
int produit (int a, int b)
{
return(a*b);
}
int puissance (int a, int n)
{
if (n == 0)
return(1);
return(a * puissance(a, n-1));
}
```

```
void imprime_tab (int *tab, int nb_Elements)
{
int i;
for (i = 0; i < nb_Elements; i++)
printf("%d \t",tab[i]);
printf("\n");
return;
}
```

# Appel d'une fonction

- L'appel d'une fonction se fait par l'expression
  ***nom-fonction(para-1,para-2,...,para-n)***

**Déclaration d'une fonction:**

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale main. Si une fonction est définie après son premier appel, elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son *prototype, qui donne le type de la fonction et celui de ses paramètres, sous la forme :* ***type nom-fonction(type-1,...,type-n);***

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction main. Par exemple, on écrira:

```
int puissance (int, int );
int puissance (int a, int n)
{
if (n == 0)
return(1);
return(a * puissance(a, n-1));
}
```

```
main()
{
int a = 2, b = 5;
printf("%d\n", puissance(a,b));
}
```

# La programmation modulaire: Appel d'une fonction

- Nous allons annoncer nos fonctions à l'ordinateur en écrivant ce qu'on appelle des **prototypes**. Ne soyez pas intimidés par ce nom ***high-tech***, ça cache en fait quelque chose de très simple.
- **Exemple:**

```
#include <stdio.h>
#include <stdlib.h>
// La ligne suivante est le prototype de la fonction aireRectangle
double aireRectangle(double largeur, double hauteur);
int main (int argc, char *argv[]) {
printf("Rectangle de largeur 5 et hauteur 10. Aire = %f\n", aireRectangle(5, 10));
printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %f\n", aireRectangle(2.5, 3.5));
printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %f\n", aireRectangle(4.2, 9.7));
return 0; }  // Notre fonction aireRectangle peut maintenant être mise n'importe où dans le code source :
double aireRectangle(double largeur, double hauteur) {
return largeur * hauteur; }
```

**N.B: Un prototype, c'est en fait une indication pour l'ordinateur. Cela lui indique qu'il existe une fonction appelée aireRectangle qui prend tels paramètres en entrée et renvoie une sortie du type que vous indiquez. Cela permet à l'ordinateur de s'organiser.**
**Grâce à cette ligne, vous pouvez maintenant placer vos fonctions dans n'importe quel ordre sans vous prendre la tête !**

# Les headers: Plusieurs fichiers par projet

**Les .h, appelés fichiers headers. Ces fichiers contiennent les prototypes des fonctions.**
**Les .c : les fichiers source. Ces fichiers contiennent les fonctions elles-mêmes.**

- **Les headers**

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

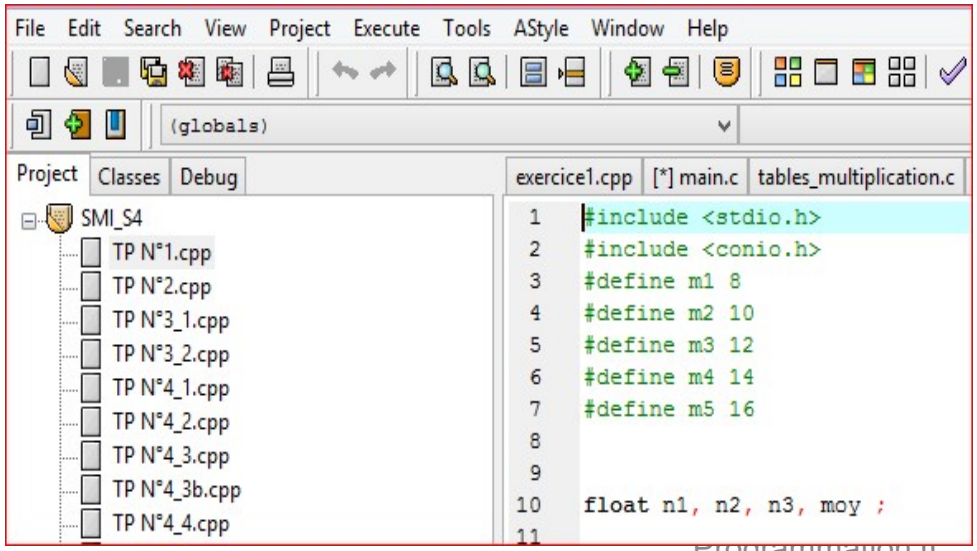
Jusqu'ici, nous n'avions qu'un seul fichier source dans notre projet. Ce fichier source, je vous avais demandé de l'appeler main.c.

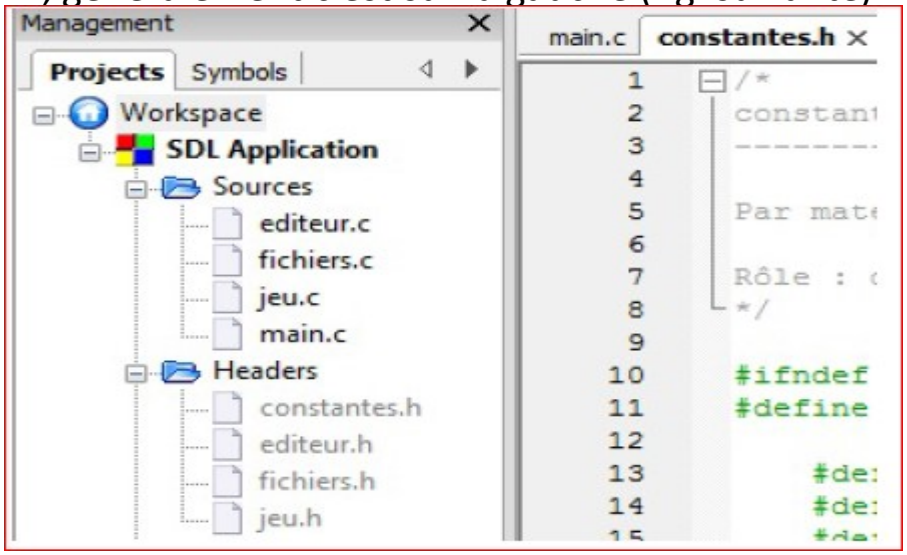
- **Plusieurs fichiers par projet**

Dans la pratique, vos programmes ne seront pas tous écrits dans ce même fichier main.c. Bien sûr, il est *possible* de le faire, mais ce n'est jamais très pratique de se balader dans un fichier de 10 000 lignes (enfin, personnellement, je trouve !). C'est pour cela qu'en général **on crée plusieurs fichiers par projet**.

- **Qu'est-ce qu'un projet, déjà ?**

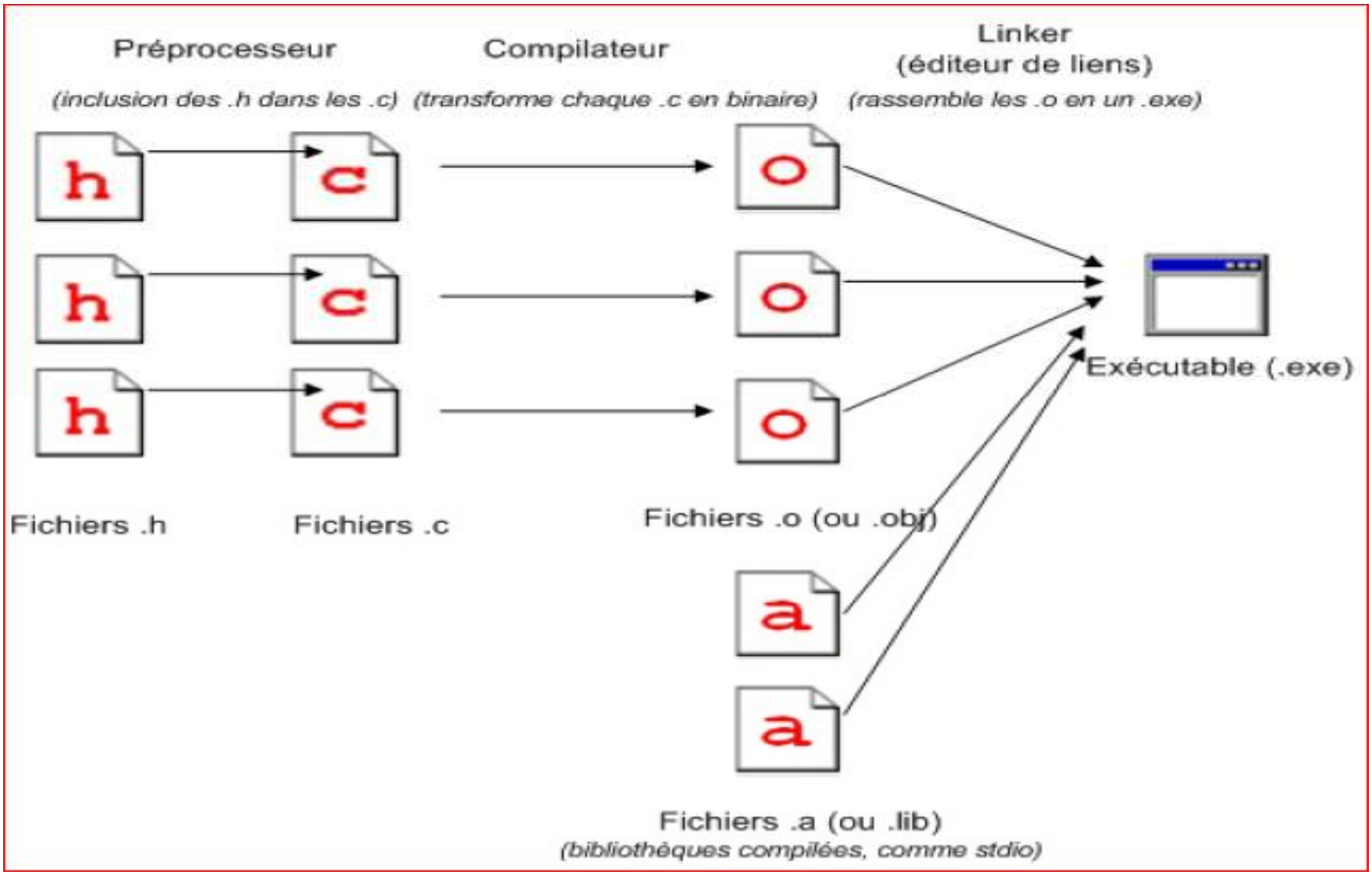
Un projet, c'est l'ensemble des fichiers source de votre programme. Pour le moment, nos projets n'étaient composés que d'un fichier source. Regardez dans votre IDE, généralement c'est sur la gauche (fig. suivante).



# Les headers: Plusieurs fichiers par projet

**Les .h, appelés fichiers headers. Ces fichiers contiennent les prototypes des fonctions.**
**Les .c : les fichiers source. Ces fichiers contiennent les fonctions elles-mêmes.**



# Variable globale

***Variable globale accessible dans tous les fichiers:***

Il est possible de déclarer des variables qui seront accessibles dans toutes les fonctions de tous **les fichiers du projet**. (il faut éviter de le faire)

- Pour déclarer une variable « globale » accessible partout, vous devez faire la déclaration de la variable en dehors des fonctions.
- Vous ferez généralement la déclaration tout en haut du fichier, après les #include.

**Code : C**

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

int resultat = 0; // Déclaration de variable globale

void triple(int nombre); // Prototype de fonction

int main(int argc, char *argv[])
{
    triple(15); // On appelle la fonction triple, qui modifie la
variable globale resultat
    printf("Le triple de 15 est %d\n", resultat); // On a accès à
resultat

    return 0;
}

void triple(int nombre)
{
    resultat = 3 * nombre;
}
```

# Variable globale

***Variable globale accessible uniquement dans un fichier:***

- Il est possible rendre une ***Variable globale*** accessible uniquement dans **le fichier** dans lequel elle se trouve. Ça reste une variable globale quand même, mais disons qu'elle n'est globale qu'aux fonctions de ce fichier, et non à toutes les fonctions du programme.
- Pour créer une variable globale accessible uniquement dans un fichier,
- rajoutez simplement le mot-clé **static** devant :
- **Variable statique à une fonction:**

Attention : c'est un peu plus délicat, ici. Si vous rajoutez le mot-clé **static** devant la déclaration d'une variable à l'intérieur d'une fonction, ça n'a pas le même sens que pour les variables globales.
En fait, la variable **static** n'est plus supprimée à la fin de la fonction. La prochaine fois qu'on appellera la fonction, la variable aura conservé sa valeur.

**Code : C**

```c
static int resultat = 0;
```

**Code : C**

```c
int incremente();

int main(int argc, char *argv[])
{
    printf("%d\n", incremente());
    printf("%d\n", incremente());
    printf("%d\n", incremente());
    printf("%d\n", incremente());

    return 0;
}

int incremente()
{
    static int nombre = 0;

    nombre++;
    return nombre;
}
```

**Code : Console**

```
1
2
3
4
```

# Fichiers .c, .h, .o, .exe, Variable globale

- **Résumé**

- Une variable déclarée dans une fonction est supprimée à la fin de la fonction, elle n'est accessible que dans cette fonction.
- Une variable déclarée dans une fonction avec le mot-clé **static** devant n'est pas supprimée à la fin de la fonction, elle conserve sa valeur au fur et à mesure de l'exécution du programme.
- Une variable déclarée en dehors des fonctions est une variable globale, accessible depuis toutes les fonctions de tous les fichiers source du projet.
- Une variable globale avec le mot-clé **static** devant est globale uniquement dans le fichier dans lequel elle se trouve, elle n'est pas accessible depuis les fonctions des autres fichiers.
- Un programme contient de nombreux fichiers .c. En règle générale, chaque fichier .c a un petit frère du même nom ayant
- l'extension .h (qui signifie **header**). Le .c contient les fonctions tandis que le .h contient les **prototypes**, c'est-à-dire la signature de ces fonctions.
- Le contenu des fichiers .h est inclus en haut des .c par un programme appelé **préprocesseur**.
- Les .c sont transformés en fichiers .o binaires par le **compilateur**.
- Les .o sont assemblés en un exécutable (.exe) par le **linker**, aussi appelé **éditeur de liens**.
- Une variable déclarée dans une fonction n'est pas accessible dans une autre fonction. On parle de **portée des variables**.

# Structures de Données

- Une **structure de données** est une manière particulière de stocker et d'organiser des données dans un ordinateur de façon à pouvoir être utilisées efficacement.
- Une **structure de données** est à la fois une organisation particulière des informations stockées dans une liste d'opérations permettant d'accéder aux données ayant cette structure.
- Différents types de structures de données existent pour répondre à des problèmes très précis exemple :

  Ensembles, listes, arbres, graphes, dictionnaires

- ❖ Ingrédient essentiel pour l'efficacité des algorithmes et des programmes.
- ❖ Permettent d'organiser la gestion des données
- ❖ Une structure de données ne regroupe pas nécessairement des objets du même type.

# Structures de Données
## Type de Données Abstraits

- Un **type abstrait ou une structure de données abstraite TDA est une** spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'elles peuvent effectuer. On qualifie d'abstrait ce type de données car il correspond à un cahier des charges qu'une structure de données doit ensuite implémenter.
- Par exemple : le type abstrait de pile sera défini par 2

opérations :

- Push qui insère un élément dans la structure
- Pop qui extrait les éléments de la structure, avec la contrainte que pop retourne l'élément qui a été empilé le plus récemment et qui n'a pas encore été dépilé
- Les types abstraits sont des entités purement théoriques utilisés principalement pour simplifier la description des algorithmes

# Fichiers et entrées/sorties:

## La gestion des fichiers

- Le C offre la possibilité de lire et d'écrire des données dans un fichier.

- Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (***buffer***), ce qui permet de réduire le nombre d'accès aux périphériques (disque...).

- Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations :
  - l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier,
  - la position de la tête de lecture,
  - le mode d'accès au fichier (lecture ou écriture) ...

Ces informations sont rassemblées dans une structure dont le type, **FILE ***, est défini dans stdio.h. Un objet de type FILE * est appelé *flot de données* (en anglais, stream).

- Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande **fopen**. Cette fonction prend comme argument le nom du fichier, négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction **fclose**.

# Ouverture et fermeture d'un fichier:

## La fonction: fopen

- Cette fonction, de type FILE* ouvre un fichier et lui associe un flot de données. Sa syntaxe est :

**fopen("*nom-de-fichier*", "*mode*")**

- La valeur retournée par fopen est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur NULL. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction fopen est égale à NULL afin de détecter les erreurs (lecture d'un fichier inexistant...).

- Le premier argument de **fopen** est le nom du fichier concerné, fourni sous forme d'une chaîne de caractères.

- Le second argument, ***mode***, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré.

- les *fichiers binaires*, pour lesquels les caractères de contrôle se sont pas interprétés.
- *Les différents modes d'accès sont les suivants :*

# Ouverture et fermeture d'un fichier:

## La fonction: fopen

| | |
|---|---|
| "r" | ouverture d'un fichier texte en lecture |
| "w" | ouverture d'un fichier texte en écriture |
| "a" | ouverture d'un fichier texte en écriture à la fin |
| "rb" | ouverture d'un fichier binaire en lecture |
| "wb" | ouverture d'un fichier binaire en écriture |
| "ab" | ouverture d'un fichier binaire en écriture à la fin |
| "r+" | ouverture d'un fichier texte en lecture/écriture |
| "w+" | ouverture d'un fichier texte en lecture/écriture |
| "a+" | ouverture d'un fichier texte en lecture/écriture à la fin |
| "r+b" | ouverture d'un fichier binaire en lecture/écriture |
| "w+b" | ouverture d'un fichier binaire en lecture/écriture |
| "a+b" | ouverture d'un fichier binaire en lecture/écriture à la fin |

- Ces modes d'accès ont pour particularités :
- Si le mode contient la lettre r, le fichier doit exister.
- Si le mode contient la lettre w, le fichier peut ne pas exister. Dans ce cas, il sera créé.
- Si le fichier existe déjà, son ancien contenu sera perdu.
- Si le mode contient la lettre a, le fichier peut ne pas exister. Dans ce cas, il sera créé.
- Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

# Ouverture et fermeture d'un fichier:

## La fonction: fclose

- **La fonction fclose** permet de fermer le flot qui a été associé à un fichier par la fonction fopen.
- Sa syntaxe est :

**fclose (*flot*)**

- où *flot* est le flot de type FILE* retourné par la fonction fopen correspondant.

La fonction fclose retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

# Fichiers et entrées/sorties:
# Ouverture et Fermuture d'un fichier

❖En C, l'assignation et l'ouverture d'un fichier sont réalisées en un seule opération :

```
FILE *f;
f = fopen(nomfichier, "r") ; // Ouverture en lecture
f = fopen(nomfichier, "w") ; // Ouverture en écriture
==========================================
FILE *f;
f = fclose(nomfichier) ; // Fermeture d'un fichier
```

# Fichiers et entrées/sorties:
# Ouverture et Fermuture d'un fichier

❖**Exemple:**

Dans le programme ci-dessous, nom_**fichier** est une chaîne de caractères qui contient le nom du fichier à ouvrir, et **données** est une variable de **type FILE *.**

```
// ouverture du fichier
FILE *f ;
char *nom_fichier="donnees.txt";
f = fopen(nom_fichier, "r" );              // ouvrir en lecture
if (f == NULL)
{
printf( "Impossible d'ouvrir le fichier %s en lecture\n", nom_fichier) ;
exit(1);
}
```

```
// fermeture du fichier
if(fclose(fic) == EOF){
    printf("Probleme de fermeture du fichier %s", nom_fic);
    exit(1);
}
printf("..... Fermeture du fichier %s\n", nom_fic);
```

# Entrées/Sorties formatées:
# Lecture et Ecriture formatées en C

## ❑La fonction d'écriture fprintf

La fonction fprintf, analogue à printf, permet d'écrire des données dans un fichier. Sa syntaxe est:

**fprintf(*flot*,"*chaîne de contrôle*",*expression-1*, ..., *expression-n*)**

où *flot* est le flot de données retourné par la fonction fopen.

Les spécifications de format utilisées pour la fonction fprintf sont les mêmes que pour printf.

## ❑La fonction de saisie fscanf

La fonction fscanf, analogue à scanf, permet de lire des données dans un fichier. Sa syntaxe est semblable à celle de scanf :

**fscanf(*flot*,"*chaîne de contrôle*",*argument-1*,...,*argument-n*)**

où *flot* est le flot de données retourné par fopen. Les spécifications de format sont ici les mêmes que celles de la fonction scanf.

# Entrées/Sorties formatées:
# Lecture et Ecriture formatées en C

```c
//Le programme suivant va simuler l'écriture d'un fichier de log
#include <stdio.h>
#include <string.h>
  int main() {
    int a = 0;
    int b = 0;

    FILE *f = fopen("nom_fichier.txt", "a");

    fprintf (f, "Le programme a été ouvert\n");

    printf ("Veuillez entrer a :");
    scanf ("%d", &a);

    fprintf (f, "L'utilisateur a entré la valeur %d pour a\n", a);

    printf ("Veuillez entrer b :");
    scanf ("%d", &b);

    fprintf (f, "L'utilisateur a entré la valeur %d pour b\n", b);
    fprintf (f, "La somme de %d et %d est donc %d\n", a, b, a+b);
    fprintf (f, "Le programme se termine\n");

    fclose(f);

    return 0;

}
```

# Entrées/Sorties formatées:
# Lecture et Ecriture formatées en C

❖Ici, nous commençons par déclarer deux variables entières *a* et *b*.
Nous déclarons également un pointeur de fichier qui récupèrera l'adresse renvoyée par la fonction ***fopen***. Grace à cette dernière, nous ouvrons le fichier *log.txt* en ajout, donc tout ce que nous écrirons dans ce fichier s'écrira à la fin de celui-ci.
Ce fichier sera créé si il n'existait pas à l'ouverture.
❖Nous utilisons la fonction ***fprintf*** pour écrire une phrase dans le fichier dont l'adresse est contenue dans le pointeur ***f***. Comme vous le voyez, nous utilisons la fonction ***fprintf*** de la même manière que nous utilisons ***printf*** sauf que nous passons en premier paramètre l'adresse du flux dans lequel doit être écrit la donnée.

❖Nous demandons ensuite d'entrer la valeur de ***a*** que nous récupérons grace au ***scanf***. vous voyez qu'ensuite, nous écrivons une phrase de log comprenant la valeur de la variable ***a***, cela vous démontre que nous pouvons utiliser les variables tout comme dans le ***printf***. Nous faisons de même pour la variable *b* et nous écrivons même dans le fichier la valeur de la somme de ces deux variables avant d'écrire que le programme se termine. Si vous exécutez votre programme, vous verrez un fichier dans votre répertoire de travail qui contiendra les données écrites

# Entrées/Sorties formatées:
# Lecture et Ecriture formatées en C

Il existe 3 fichiers prédéfinis et déjà ouverts:
- **stdin** (standard input) : unité d'entrée (par défaut, le clavier) ;
- **stdout** (standard output) : unité de sortie (par défaut, l'écran) ;
- **stderr** (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

Il est fortement conseillé d'afficher systématiquement les messages d'erreur sur stderr afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

**scanf("%d",&x);** est identique à **fscanf(stdin,"%d",&x);**
**printf("%d",x);** est identique à **fprintf(stdout,"%d",x);**

- pour lire un entier dans un fichier Ascii f ouvert par fopen, on utilise :

**fscanf(f,"%d", &x);**

- Pour écrire un entier (sous forme de texte) dans un fichier f ouvert par fopen, on écrira :

**fprintf(f,"%d",x);**

# Entrées/Sorties formatées:
## Impression et lecture de caractères

Similaires aux fonctions getchar et putchar, les fonctions **fgetc** et **fputc** permettent respectivement de lire et d'écrire un caractère dans un fichier.

- La fonction **fgetc**, de type int, retourne le caractère lu dans le fichier. Elle retourne la constante EOF lorsqu'elle détecte la fin du fichier. Son prototype est:

**int fgetc(FILE* *flot*);**

où ***flot*** est le flot de type FILE* retourné par la fonction **fopen**. Comme pour la fonction **getchar**, il est conseillé de déclarer de type **int** la variable destinée à recevoir la valeur de retour de **fgetc** pour pouvoir détecter correctement la fin de fichier.

- La fonction **fputc** écrit *caractere* dans le flot de données :

**int fputc(int *caractere*, FILE **flot*)**

Elle retourne l'entier correspondant au caractère lu (ou la constante EOF en cas d'erreur).

- Il existe également deux versions optimisées des fonctions **fgetc et fputc** qui sont implémentées par des macros. Il s'agit respectivement de **getc** et **putc**. Leur syntaxe est similaire à celle de fgetc et fputc :

**C = fgetc(stdin);** est identique à **C = getchar();**

**fputc('a', stdout);** est identique à **putchar('a');**

# Entrées/Sorties formatées:

## Impression et lecture de caractères

**Exemple:1: fgets**

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main(int argc, char *argv[])
{
char nom[10];
printf("Quel est votre nom ? ");
fgets(nom, 10, stdin);
printf("Ah ! Vous vous appelez donc: %s !
\n\n", nom);
getch();  }
```

**Exemple:2:fgetc**

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

int main (void)
{
   int x = fgetc(stdin);
   printf ("x = %d ('%c')\n", x, x);
   getch();}
```

**Exemple:3: fputc**

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main(int argc, char *argv[])
{
FILE* fichier = NULL;
fichier = fopen("test.txt", "w");
if (fichier != NULL)
{
fputc('A', fichier);
// Écriture du caractère A
fclose(fichier);
}
getch();
}
```

- **Exemple:**

**Exemple d'implémentation: u**n programme qui lit le contenu du fichier texte entrée, et le recopie caractère par caractère dans le fichier sortie :

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

int main()
{
        //declaration des flux
    FILE *f_in, *f_out;
    int value;
    f_in = fopen("entree.txt", "r");  //ouverture en lecture seule
    f_out = fopen("sortie.txt", "w");  //ouverture en ecriture
    if(!(f_in && f_out))  //si l'ouverture des fichiers echoue, arret du programme
    {
        exit(0);
    }
    fprintf(f_in, "\nProbleme d'ecriture\n");
    while(!feof(f_in))  //tant que je suis pas a la fin du fichier d'entree
    {
        value = fgetc(f_in); //recupere caractere par carectere du fichier d'entree, et rangement dans value
        if (value == EOF) //Si je suis a la fin du fichier (d'entree) j'arrete le programme
        {
        exit(0);
        }
        value = fputc(value, f_out); //je copie les caractere dans le fichier de sortie

        if(value == EOF) //Si je suis a la fin du fichier de sortie donc c'est une erreur,
                         // alors j'arrete le programme en affichant un message d'erreur
        {
        fprintf(stderr, "\nProbleme d'ecriture\n");
        exit(-1);
         }
      }
//J'ai terminer de copier donc je ferme tous les flux
    fclose(f_in);
    fclose(f_out);
    return 0;

}
```

# Entrées/Sorties formatées:
## Impression et lecture de caractères

### Relecture d'un caractére

Il est possible de replacer un caract`ere dans un flot au moyen de la fonction ungetc :

**int ungetc(int *caractere, FILE *flot);**

Cette fonction place le caractère *caractere (converti en unsigned char) dans le flot* ***flot****. En* particulier, si *caractère est égal au dernier caractére lu dans le flot, elle annule le déplacement* provoqué par la lecture précédente. Toutefois, **ungetc** peut être utilisée avec n'importe quel caractére (sauf **EOF**). Par exemple, l'exécution du programme suivant

```
#include <stdio.h>
#include <stdlib.h>
#define  ENTREE  "entree.txt"
int main(void)
{
FILE *f_in;
int c;
if ((f_in = fopen(ENTREE,"r")) == NULL)
{
fprintf(stderr, "\nErreur: Impossible de lire le
fichier %s\n",ENTREE);
return(EXIT_FAILURE);
}
```

```
while ((c = fgetc(f_in)) != EOF)
{
if (c == '0')
ungetc('.',f_in);
putchar(c);
}
fclose(f_in);
return(EXIT_SUCCESS);
}
```

**//sur le fichier entree.txt dont le contenu est 097023 affiche `a l'´ecran 0.970.23**

# *Entrées/Sorties formatées:*
# *Les entrées-sorties binaires: fread, fwrite*

Les fonctions ***fread/fwrite*** attendent, outre le pointeur du fichier, le nombre d'éléments à lire/écrire, la taille en octet d'un élément et l'adresse du « *buffer*» (zone mémoire) où ranger/trouver les données à lire/écrire. Ce *buffer* est en général un tableau, mais il peut aussi être une variable simple ou structurée (voir **Erreur ! Source du renvoi introuvable.**), surtout en accès non séquentiel avec positionnement préalable par *fseek*.
Très important : *fread* et *fwrite* renvoient le **nombre d'éléments effectivement lus ou écrits**. Si la valeur renvoyée par *fread* diffère du nombre d'éléments à lire, c'est que l**a fin du fichier a été rencontrée** lors de la lecture. On se sert donc de la valeur renvoyée par *fread* pour savoir si la fin du fichier est atteinte.

Les prototypes de ***fread*** et ***fwrite*** sont les suivants :

```
int fread( void* adr_buffer, int taille_element, int nb_elements, FILE* fic);
int fwrite( void* adr_buffer, int taille_element, int nb_elements, FILE* fic);
```

# Entrées/Sorties formatées:
# Les entrées-sorties binaires: fwrite, fread

exemple, le programme suivant écrit un tableau d'entiers (contenant les 50 premiers entiers) avec **fwrite** dans le fichier sortie, puis lit ce fichier avec **fread** et imprime les éléments du tableau.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define NB 50
#define F_SORTIE "sortie.txt"
int main(void)
{
FILE *f_in, *f_out;
int *tab1, *tab2;
int i;

tab1 = (int*)malloc(NB * sizeof(int));
tab2 = (int*)malloc(NB * sizeof(int));
for (i = 0 ; i < NB; i++)
tab1[i] = i;
/* ecriture du tableau dans F_SORTIE */
if ((f_out = fopen(F_SORTIE, "wb")) == NULL)
{
fprintf(stderr, "\nImpossible d'ecrire dans le fichier
%s\n",F_SORTIE);
return(EXIT_FAILURE);
}
fwrite(tab1, NB * sizeof(int), 1, f_out);
fclose(f_out);
}
```

```
/* lecture dans F_SORTIE */
if ((f_in = fopen(F_SORTIE, "r")) == NULL)
{
fprintf(stderr, "\nImpossible de lire dans le fichier
%s\n",F_SORTIE);
return(EXIT_FAILURE);

fread(tab2, NB * sizeof(int), 1, f_in);
fclose(f_in);
for (i = 0 ; i < NB; i++)
printf("%d\t",tab2[i]);
printf("\n");
getch();
}
```

Les éléments du tableau sont bien affichés à l'écran. Par contre, on constate que le contenu du fichier sortie n'est pas encodé.

# Entrées/Sorties formatées

## Se déplacer dans un fichier

- Chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier.
- Vous pouvez imaginer que c'est exactement comme le curseur de votre éditeur de texte (tel Bloc-Notes). Il indique où vous êtes dans le fichier, et donc où vous allez écrire.
- En résumé, le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier.

Il existe trois fonctions à connaître :

❖ **ftell** : indique à quelle position vous êtes actuellement dans le fichier ;

❖ **Fseek** : positionne le curseur à un endroit précis ;

❖ **rewind** : remet le curseur au début du fichier (c'est équivalent à demander à la fonction **fseek** de positionner le curseur au début).

# Entrées/Sorties formatées

## Se déplacer dans un fichier

- **ftell : position dans le fichier**

Cette fonction est très simple à utiliser. Elle renvoie la position actuelle du curseur sous la forme d'un long : /*Le nombre renvoyé indique donc la position du curseur dans le fichier.*/

```
long ftell(FILE* pointeurSurFichier);
```

- **fseek : se positionner dans le fichier**

Le prototype de **fseek** est le suivant :

```
int fseek(FILE* pointeurSurFichier, long deplacement, int origine);
```

La fonction **fseek** permet de déplacer le curseur d'un certain nombre de caractères (indiqué par déplacement) à partir de la position indiquée par origine.
Le nombre déplacement peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
Quant au nombre origine, vous pouvez mettre comme valeur l'une des trois constantes (généralement des define) listées ci-dessous :

# Entrées/Sorties formatées

## Se déplacer dans un fichier

- **SEEK_SET** : indique le début du fichier ;
- **SEEK_CUR** : indique la position actuelle du curseur ;
- **SEEK_END** : indique la fin du fichier.
- Voici quelques exemples pour bien comprendre comment on jongle avec deplacement et origine.

/* Le code suivant place le curseur deux caractères après le début :*/

```
fseek(fichier, 2, SEEK_SET);
```

/*Le code suivant place le curseur quatre caractères avant la position courante :*/

```
fseek(fichier, -4, SEEK_CUR);
```

/*Remarquez que deplacement est négatif car on se déplace en arrière.
Le code suivant place le curseur à la fin du fichier :*/

```
fseek(fichier, 0, SEEK_END);
```

# Entrées/Sorties formatées

## Se déplacer dans un fichier

### rewind : retour au début

Cette fonction est équivalente à utiliser fseek pour nous renvoyer à la position 0 dans le fichier. Voici le prototype :

**void rewind(FILE* pointeurSurFichier);**

❖**rename : renommer un fichier**

Voici le prototype de cette fonction **:**

**int rename(const char* ancienNom, const char* nouveauNom);**

❖**remove : supprimer un fichier**

Cette fonction supprime un fichier sans demander son reste :

int remove(const char* fichierASupprimer);

La fonction renvoie 0 si elle a réussi à renommer, sinon elle renvoie une valeur différente de 0.

voici un exemple :

```
int main(int argc, char *argv[])
{
rename("test.txt", "test_renomme.txt");
remove("test.txt");
return 0;
}
```

# Entrées/Sorties formatées: Positionnement dans un fichier

**Exemple:** le programme suivant écrit un tableau d'entiers (contenant les 10 premiers entiers) avec fwrite dans le fichier sortie, puis lit ce fichier avec fread et imprime les éléments du tableau.

**Par exemple**

```
#include <stdio.h>
#include <stdlib.h>
#define NB 10
#define F_SORTIE "sortie.txt"
int main(void)
{
FILE *f_in, *f_out;
int *tab;
int i;
tab = (int*)malloc(NB * sizeof(int));
            /* taille de int =4 octet */
for (i = 0 ; i < NB; i++)
tab[i] = i;
```

```
/* ecriture du tableau dans F_SORTIE */
if ((f_out = fopen(F_SORTIE, "w")) == NULL)

/*ou  f_out = fopen(F_SORTIE, "w"); if (f_out ==
NULL) */
{
fprintf(stderr, "\nImpossible d'ecrire dans le
fichier %s\n",F_SORTIE);
return(EXIT_FAILURE);
}
fwrite(tab, NB * sizeof(int), 1, f_out);
  fclose(f_out);

  /* lecture dans F_SORTIE */
  f_in = fopen(F_SORTIE, "r");
/* on se positionne a la fin du fichier */
  fseek(f_in, 0, SEEK_END);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d", i);
```

# Entrées/Sorties formatées

## Implémentation: Positionnement dans un fichier

```
/* retour au debut du fichier */
  rewind(f_in);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d", i);
/* deplacement de 5 int en avant */
  fseek(f_in, 5*sizeof(int),
SEEK_CUR);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d", i);
```

```
/* deplacement de 2 int en
 arriere */
fseek(f_in, -2 * sizeof(int),
 SEEK_END);
printf("\n position %ld",
 ftell(f_in));
fread(&i, sizeof(i), 1, f_in);
printf("\t i = %d", i);
fclose(f_in);
return(EXIT_SUCCESS);
getch();
```

L'exécution de ce programme affiche à l'écran :

```
position: 40    i = 10
position: 0     i = 0
position: 24    i = 6
position: 32    i = 8
```

On constate en particulier que l'emploi de la fonction fread provoque un déplacement correspondant à la taille de l'objet lu à partir de la position courante

# Les structures et les pointeurs

```
struct Point
{
        int x;
        int y;
};
```

```
struct Rectangle
{
        struct Point p1;
        struct Point p2;
};
```

```
struct Point p;

p.x = 3;
p.y = 5;
```

```
struct Rectangle r;

r.p1.x = 2;
...
```

```
struct Point *q;

q->x = 3;
q->y = 5;
```

définit la variable q de type pointeur sur la structure Point.

# Allocation dynamique de la mémoire
## malloc, calloc

**Alloue de la mémoire:**

```
void TableauDynamique(void)
{
int *s, *t;
s = (int *)malloc(50 * sizeof(int));
t = (int *)calloc(50, sizeof(int));
}
```

De plus, **calloc** (mais pas **malloc** !) initialise la zone allouée avec des zéros.

**free(s)** ou **free(t)** : libère la mémoire pointée par p, allouée par malloc ou calloc.

```
void Exemple (void)
{
struct Point *p;
p = (Point *)malloc(sizeof(Point));
}
```

définit la variable p de type pointeur sur Point, donc réserve la mémoire pour stocker une adresse, mais ne réserve pas la mémoire pour stocker une structure.

# LES LISTES CHAINEES

**Définition:**

- *Pour stocker une collection d'informations et s'affranchir du problème posé lors de la réservation statique de place mémoire (effectuée lors de la déclaration du tableau par exemple), on utilise* **une structure de donnée appelée liste chaînée**.
- **Une liste chaînée** *est constituée d'un ensemble de cellules chaînées entre elles qui contiennent les informations à mémoriser. L'emplacement mémoire nécessaire à chaque Elementest alloué dynamiquement.*
- ***Une liste chaînée*** *est une liste à accès séquentiel, où chaque élément contient une valeur et une référence à l'élément suivant*

L'allocation (ou la libération) se fait élément par élément.

- Les opérations  a implémenter sur une liste peuvent être:

✓ Créer une liste

✓ Supprimer une liste

✓ Rechercher un élément particulier

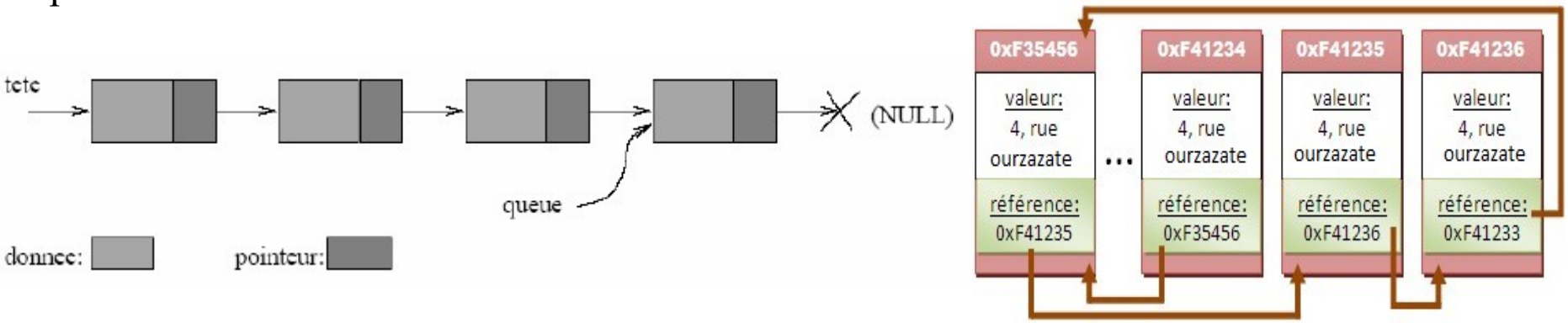
✓ Insérer un élément (en début, en  fin ou au milieu)

✓ Supprimer un élément particulier

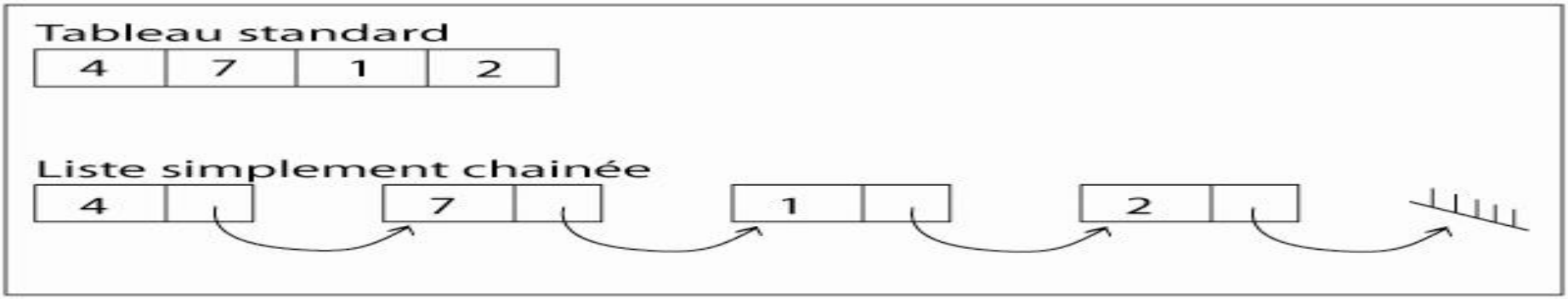
✓ Permuter deux éléments

# LES LISTES CHAINEES

- Une liste chaînée est un moyen d'organiser une série de données en mémoire. Cela consiste à assembler des structures en les liant entre elles à l'aide des pointeurs. On pourrait les représenter comme ceci :




- Une liste chaînée est différente dans le sens où les éléments de votre liste sont répartis dans la mémoire et reliés entre eux par des pointeurs. Vous pouvez ajouter et enlever des éléments d'une liste chaînée à n'importe quel endroit, à n'importe quel instant, sans devoir recréer la liste entière. Nous allons essayer de voir ceci plus en détail sur ces schémas :



# Déclaration d'une Listes Chainées

On peut créer des listes chaînées de n'importe quel type d'éléments : entiers, caractères, structures, tableaux, voir même d'autres listes chaînées. Il est même possible de combiner plusieurs types dans une même liste.
voici la déclaration d'une liste simplement chaînée d'entiers :

```
//Structure d'un élément de liste simplement chaînée:
 typedef struct Element
{
  Int    valeur; //valeur de l'élément
  Element  * suivant; // Pointeur vers éléments
  }Element;
```

Ou

```
typedef struct Element Element;

struct Element
{
int nombre;
Element *suivant;
};
```

On crée le type **élément** qui est une structure contenant un entier (**nombre**) et un pointeur sur élément (**suivant**), qui contiendra l'adresse de l'élément suivant. C'est ce qui permet de lier les éléments les uns aux autres :

chaque élément sait où se trouve l'élément suivant en mémoire . les cases ne sont pas côte à côte en mémoire. C'est la grosse différence par rapport aux tableaux. Cela offre l'avantage de souplesse car on peut plus facilement ajouter de nouvelles cases par la suite au besoin.

# La structure de contrôle

la structure qu'on vient de créer (que l'on dupliquera autant de fois qu'il y a d'éléments), nous allons avoir besoin d'une autre structure pour contrôler l'ensemble de la liste chaînée. Elle aura la forme suivante :

```
struct Liste
{
struct Element * premier ;
} liste ;
```

**Ou**

```
typedef struct Liste liste;
struct Liste
{
Element *premier;
};
```

Cette structure Liste contient un pointeur vers le premier élément de la liste. En effet, il faut conserver l'adresse du premier élément pour savoir où commence la liste. Si on connaît le premier élément, on peut retrouver tous les autres en sautant d'élément en élément à l'aide des pointeurs suivant.

Nous n'aurons besoin de créer qu'un seul exemplaire de la structure Liste. Elle permet de contrôler toute la liste.

# Implantation d'une Listes Chainées

## Le dernier élément de la liste

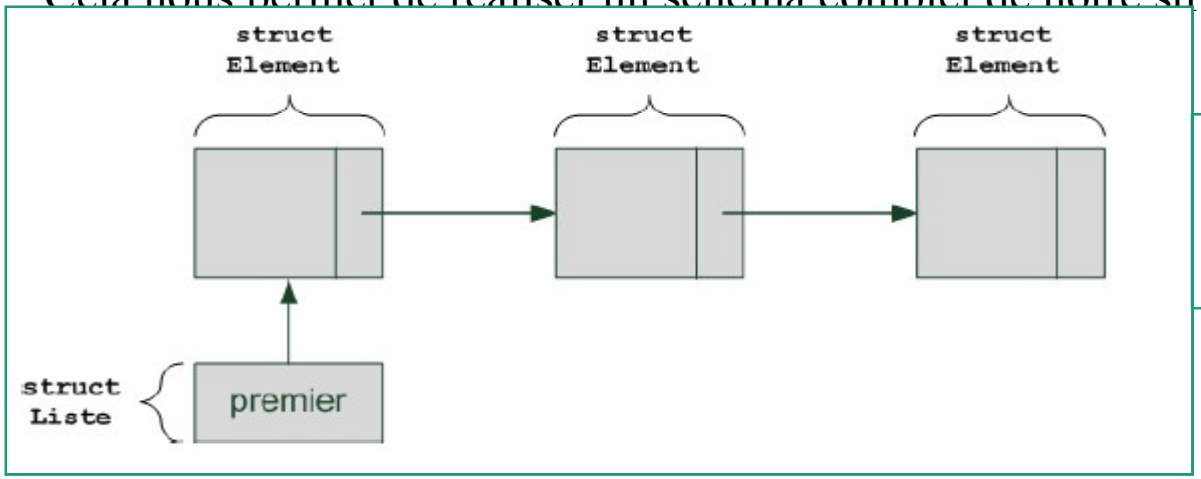
- Pour retenir le dernier élément de la liste:

➢ il faudra bien arrêter de parcourir la liste à un moment donné. Avec quoi pourrait-on signaler à notre programme Stop, ceci est le dernier élément ?

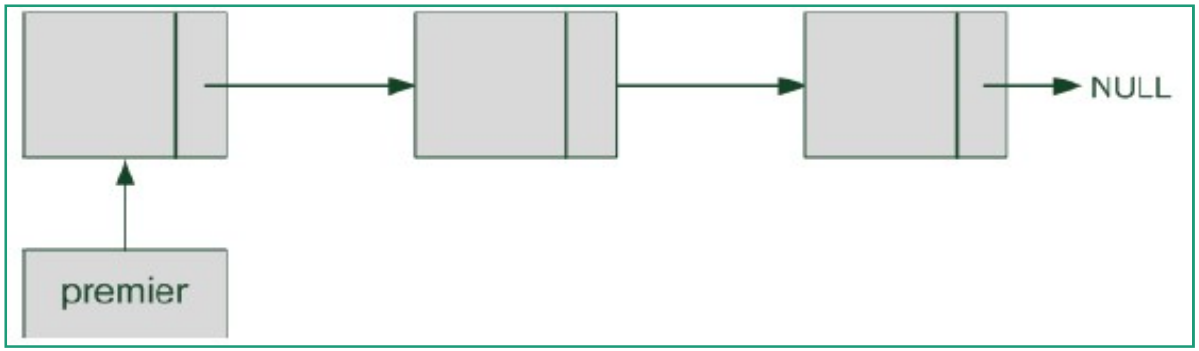
➢ Il serait possible d'ajouter dans la structure Liste un pointeur vers le dernier Elément.

➢ Toutefois, il suffit de faire pointer le dernier élément de la liste vers NULL, c'est-à-dire de mettre son pointeur suivant à NULL.

- Cela nous permet de réaliser un schéma complet de notre structure de liste chaînée.




La structure Liste nous donne des informations sur l'ensemble de la liste chaînée.

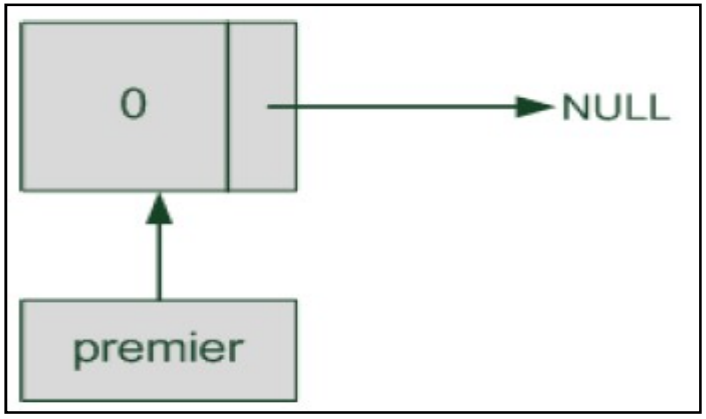



Le dernier élément de la liste pointe vers NULL pour indiquer la fin de liste.

# Implantation d'une Listes Chainées

## Initialiser la liste

```
Liste *initialisation()
{
Liste *liste = malloc(sizeof(*liste));          //1
Element *Element = malloc(sizeof(*Element));
if (liste == NULL || Element == NULL)     //2
{
exit(EXIT_FAILURE);
}
Element->nombre = 0;                              //3
Element->suivant = NULL;                          //4
liste->premier = Element;
return liste;
}
```




On commence par créer la structure de contrôle liste .

1. On l'alloue dynamiquement avec un malloc. La taille à allouer est calculée automatiquement avec sizeof(*liste).

L'ordinateur saura qu'il doit allouer l'espace nécessaire au stockage de la **structure Liste** .

2. On alloue ensuite de la même manière la mémoire nécessaire au stockage du **premier élément**. On vérifie si les allocations dynamiques ont fonctionné. En cas d'erreur, on arrête immédiatement le programme en faisant appel à **exit().**

3. on définit les **valeurs** de notre **premier élément** :

la donnée **nombre** est mise à 0 par défaut ;

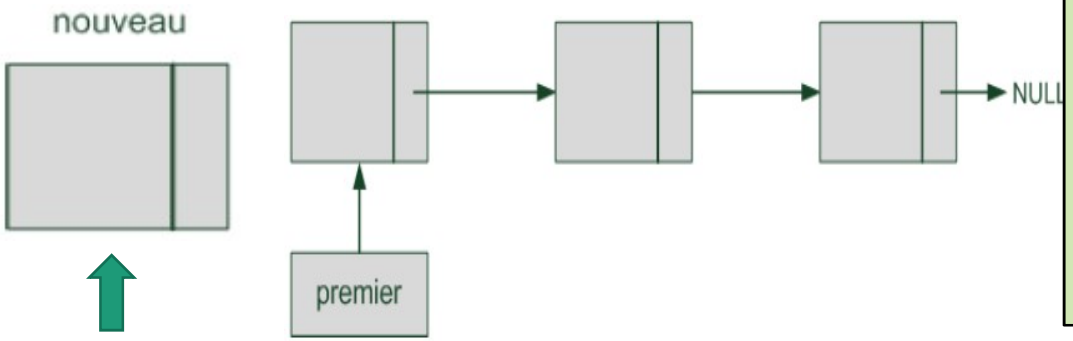
4. le pointeur **suivant** pointe vers NULL car le **premier élément** de notre **liste** est aussi le dernier pour le moment.

Comme on l'a vu plus tôt, le **dernier élément** doit pointer vers **NULL** pour signaler qu'il est en fin de liste.

# Implantation d'une Listes Chainées

## Ajouter un élément

```
void insertion(Liste *liste, int nvNombre)
{
/* Création du nouvel élément */
Element *nouveau = malloc(sizeof(*nouveau)); //1
if (liste == NULL || nouveau == NULL)
{
exit(EXIT_FAILURE);
}
nouveau->nombre = nvNombre;
/* Insertion de l'élément au début de la liste */
nouveau->suivant = liste->premier;   //2
liste->premier = nouveau;                  //3
}
```




Insertion d'un élément en début de liste

- **La fonction insertion()** prend en paramètre l'élément de contrôle **liste** (qui contient l'adresse du premier élément) et le nombre à stocker dans le nouvel élément que l'on va créer.

1. Dans un premier temps, on alloue l'espace nécessaire au stockage du nouvel élément et on y place le nouveau nombre **nvNombre**.

- l'insertion du nouvel élément dans la liste chaînée:

Pour mettre à jour correctement les pointeurs, nous devons procéder dans cet ordre précis :

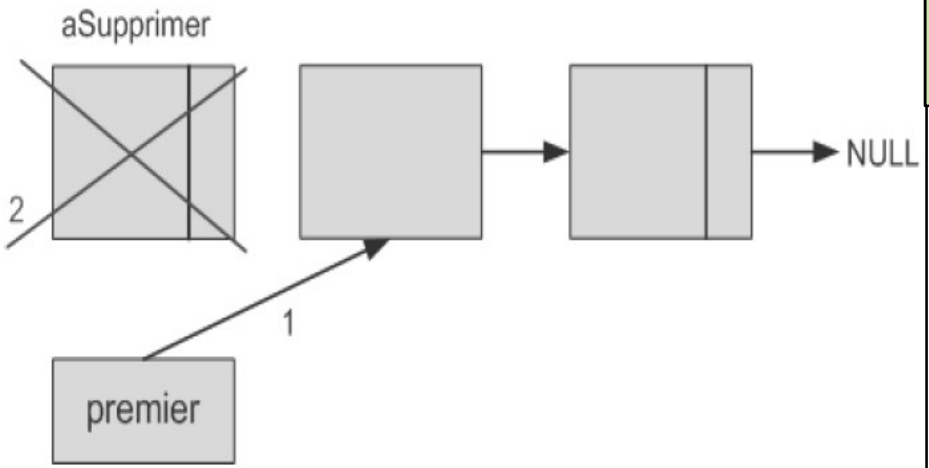
2. faire pointer notre nouvel élément vers son futur successeur, qui est l'actuel premier élément de la liste .

3. faire pointer le pointeur **premier** vers notre nouvel élément.

# Implantation d'une Listes Chainées

## Supprimer un élément

```
void suppression(Liste *liste)
{
if (liste == NULL)                  //1.
{
exit(EXIT_FAILURE);
}
if (liste->premier != NULL)     //2.
{
Element *aSupprimer = liste->premier;   //3.
liste->premier = liste->premier->suivant; //4.
free(aSupprimer);                   //5.
}}
```




1. On commence par vérifier que le pointeur qu'on nous envoie n'est pas **NULL**, sinon on ne peut pas travailler.
2. On vérifie ensuite qu'il y a au moins un élément dans la liste, sinon il n'y a rien à faire.
3. On sauvegarder l'adresse de l'élément à supprimer dans un pointeur **aSupprimer**.
4. On adapte le pointeur **premier** vers le nouveau premier élément, qui est actuellement en seconde position de la liste chaînée.
5. Il ne reste plus qu'à supprimer l'élément correspondant à notre pointeur **aSupprimer** avec un **free(aSuprimer)**

# Implantation d'une Listes Chainées

## Afficher la liste chaînée

```
void afficherListe(Liste *liste)
{
if (liste == NULL)
{
exit(EXIT_FAILURE);
}
Element *actuel = liste->premier;
while (actuel != NULL)
{
printf("%d -> ", actuel->nombre);
actuel = actuel->suivant;
}
printf("\n");
}
```

Il suffit de partir du premier élément et d'afficher chaque élément un à un en sautant de bloc en bloc.

Cette fonction est simple : on part du premier élément et on affiche le contenu de chaque élément de la liste (un nombre). On se sert du pointeur suivant pour passer à l'élément qui suit à chaque fois.

**Exemple:** On peut tester la création de notre liste chaînée et son affichage avec un main :
En plus du premier élément (que l'on a laissé ici à 0), on en ajoute **trois nouveaux** à cette liste. Puis on en **supprime** un. Au final, le contenu de la liste chaînée sera donc :
2 -> 1 -> 0 -> NULL

```
int main()
{
Liste *maListe = initialisation();
insertion(maListe, 1);
insertion(maListe, 2);
insertion(maListe, 3);
suppression(maListe);
afficherListe(maListe);
getch();
}
```

# Implantation d'une Listes Chainées

L'objectif de cette partie des listes chainées c'est d'apprendre comment:

- **Ajouter un élément en tête d'une liste chaîné**
- **Ajouter un élément en fin de liste**
- **Supprimer un élément en tête de liste**
- **Supprimer un élément en fin de liste**
- **Rechercher un élément dans une liste**
- **Compter le nombre d'occurrences d'une valeur**
- **Recherche du i-ème élément**
- **Récupérer la valeur d'un élément**
- **Compter le nombre d'éléments d'une liste chaîné**
- **Effacer tous les éléments ayant une certaine valeur**

# Implantation d'une Listes Chainées

## Insertion d'une nouvelle Element en début de liste

```
void inserer_premier (Liste* L, int valeur)
{
struct Element* nouveau ;
/* On crée un nouvel élément */
nouveau = (struct Element*) malloc (sizeof(struct cellule)) ;
nouveau -> nombre= valeur;                    /* On assigne la valeur au nouvel élément */
nouveau -> suivant = L-> premier ;   /* On assigne l'adresse de l'élément suivant au nouvel élément */
L-> premier = nouveau ;              /* On retourne la nouvelle liste, le pointeur sur le premier élément */
}
```

# Implantation d'une Listes Chainées

## Insertion d'une nouvelle Element en fin de liste

```
void inserer_fin (Liste* L, int valeur)
{
struct Element* nouveau , * parcours;                    /* On crée un nouvel
élément */
nouveau = (struct Element*) malloc (sizeof(struct cellule)) ;
nouveau -> val = valeur;                                 /* On assigne la valeur au nouvel
élément */
nouveau -> suivant = NULL;                               /* On ajoute en fin, donc aucun élément ne va
suivre */
if (L-> premier == NULL)
{                                                        /* Si la liste est videé il suffit de renvoyer
l'élément créé */
L-> premier = nouveau ;
}
Else                    /* Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on
indique que le                  dernier élément de la liste est relié au nouvel élément */
{
parcours = L-> premier ;
while ( parcours -> suivant !=NULL)
{
parcours = parcours -> suivant ;
}
parcours ->suivant = nouveau ;
```

# Implantation d'une Listes Chainées

## Suppression d'une Elementen début de liste

```
float enlever_premier (Liste* L)
{
struct Element* inter;
Int valeur ;
if (L-> premier != NULL)       /* Si la liste est non vide, on se prépare à renvoyer l'adresse
                  de l'élément en 2ème position */
{
inter = L-> premier ;
L-> premier = inter -> suivant;
valeur = inter -> val ;
free(inter) ;                                  /* On libère le premier élément */
return (valeur) ;                                /* On retourne le nouveau début de la liste */
}
else
{
Return  NULL;
}
}
```

# Implantation d'une Listes Chainées

## Suppression d'une nouvelle Element en fin de liste

```
Liste supprimerElementEnFin( Liste liste) {        /* Si la liste est vide, on
retourne NULL */

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

# Implantation d'une Listes Chainées

## Recherche d'une valeur dans la liste

```
int recherche (Liste L, int valeur)
{
struct Element* parcours;
if (L.premier == NULL)
{
return(0) ;
}
else
{
parcours = L.premier ;      /* Tant que l'on n'est pas au bout de la liste */
while (( parcours -> suivant !=NULL) && (parcours -> val !=valeur))
{
parcours = parcours -> suivant ;
}                           /* Si l'élément a la valeur recherchée, on renvoie son adresse */
if (parcours -> val == valeur) return (1);
else return (0);
}
}
```

# Implantation d'une Listes Chainées

## Recherche du i-eme élémenet

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

# Implantation d'une Listes Chainées

## Suppression d'une valeur dans la liste

```
int supprime(Liste* L, int valeur)
{
struct Element* pred, * parcours;
if (L-> premier != NULL)
{
parcours = L-> premier ;
pred = L-> premier ;
while ((parcours -> suivant !=NULL)&&(parcours -> val !=valeur))
{
pred = parcours ;
parcours = parcours -> suivant ;
}
if (parcours -> val == valeur)
{
if (L-> premier == parcours) L-> premier = parcours ->suivant;
else pred->suivant = parcours-> suivant ;
free (parcours) ;
}}}
```

# Implantation d'une Listes Chainées

## Compter le nombre d'éléments d'une liste chaîné

```
int nombreElements(Liste liste)
{
                                        /* Si la liste est vide, il y a 0 élément */
 if(liste == NULL)
return 0;          /* Sinon, il y a un élément (celui que l'on est en train de
traiter) plus                le nombre d'éléments contenus dans le reste de la
liste */
return nombreElements(liste->suivant)+1;
  }
```