2019

Structures de données en C

Hassan Zekkouri

Zeek Zone – Access center

5/8/2019

Table of Contents

CHAPITRE 1 : RAPPELS 3

1) Variable 3

2) Adressage 3
1) Adressage direct 3
2) Adressage indirect 3

3) Pointeur 4

4) Allocation dynamique de la mémoire <stdlib.h> 4
a) La fonction malloc 5
b) La function calloc 5
c) La fonction realloc 5
d) La fonction free 6

5) Chaînes de caractères <string.h> 6
a) Fonctions spéciales: 6

6) Techniques de programmation 7
a) Recherche 7
b) Tri 8

CHAPITRE 2: GESTION DES FICHIERS 10

1) Généralités 10
a) Ouvrir un fichier : 10

b) Lire ou écrire 10

c) Fermer le fichier, ... 10
a) Les fichiers 'texte', ou fichiers à accès séquentiel : 10
b) Les fichiers binaires ou fichiers à accès direct (aléatoire) : 10

2) Manipulation des fichiers 10
c) Déclaration d'un pointeur sur un fichier 10
d) Ouverture d'un fichier 10
i. Les modes d'ouverture 11
ii. Gestion du résultat de fopen : 11

3) Fermeture de fichier 12

4) Traitement des fichiers textes 12
a) Lecture d'un fichier 12
b) Ecriture dans un fichier 16

5) Traitement des fichiers binaires 17
a) Ecriture d'un fichier binaire 17

b) Lecture d'un fichier binaire 19
c) Accès direct 20

6) Gestion de la fin de fichier 22

CHAPITRE 3 : LES LISTES SIMPLEMENT CHAINEES 24

1) Généralité 24

2) Création et déclaration en C d'une liste 24
a) Déclaration 24
b) Création 25
c) Accès aux champs d'une liste chainée 26
d) A retenir, les points clés des listes chaînées 26

3) Opérations sur les listes chaînées 27
a) Initialisation 27
b) Insertion d'un élément dans la liste 27
c) Retirer un élément dans la liste simplement chaînée 29
d) Affichage et destruction d'une liste 31

CHAPITRE 4 : LES PILES ET LES FILES 33

1) Les Piles 33
a) Généralité 33
b) Construction d'une pile 33
c) Operations sur les piles 34

2) Les Files 36
a) Généralité 36
b) Construction d'une file 36
c) Operations sur les files 37

CHAPITRE 5: LES LISTES DOUBLEMENT CHAINEES 39

1) Création et déclaration d'une liste doublement chaînée 39
a) Déclaration 39
b) Création 40
c) Accès aux champs d'une liste chainée 41
d) A retenir, les points clés des listes doublement chaînées 41

2) Opérations sur les listes doublement chaînées 41
a) Insertion d'un élément dans la liste 41
b) Retirer un élément dans la liste doublement chaînée 44
c) Affichage de la liste doublement chaînée 45
d) Destruction de la liste 46

BIBLIOGRAPHY 46

Chapitre 1 : Rappels

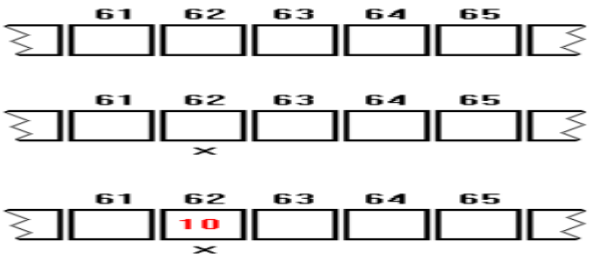
1) Variable

Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée. C'est un nom donné à un emplacement mémoire (RAM) où la valeur sera gardée pour être utilisé pendant l'exécution de programme.

Exemple :

```
1 int x ; // Réserve un emplacement pour un entier en mémoire
2 x = 10 ; // Écrit la valeur 10 dans l'emplacement réservé.
```

Nom variable	adresse	valeur
rien	rien	rien
x	62	rien
x	62	10



2) Adressage

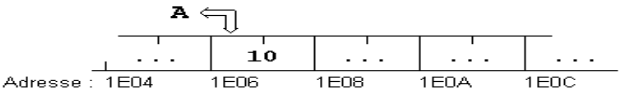
En C, on dispose de deux modes d'adressage :

- Adressage direct : Accès au contenu d'une variable par le nom de la variable.
- Adressage indirect : Accès au contenu d'une variable par le biais de l'adresse de la variable. L'opérateur & permet de récupérer l'adresse d'une variable :
  &x ceci est l'adresse de x.

1) Adressage direct

La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

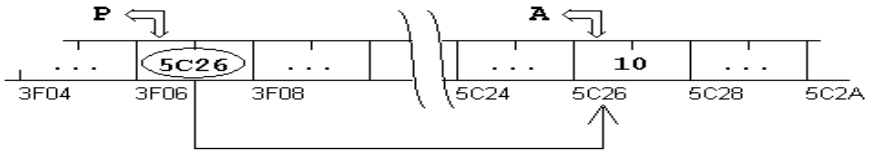
```
short A;
A = 10;
```



2) Adressage indirect

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, *nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée* ***pointeur.*** Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



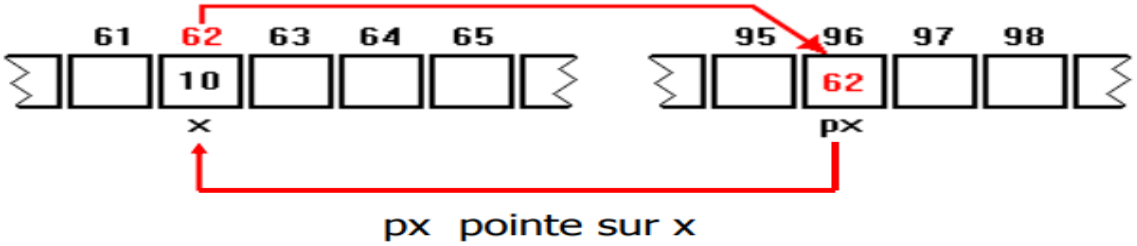
3) Pointeur

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

Chaque pointeur est limité à un type de données.

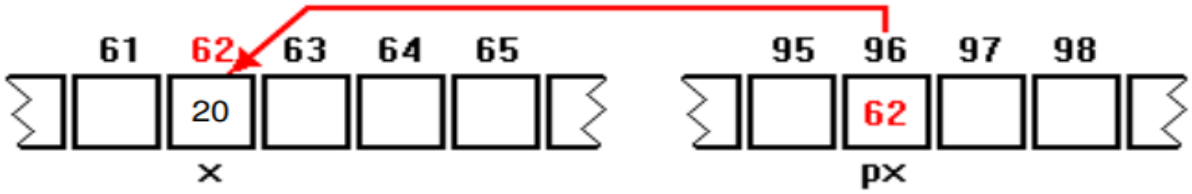
Declaration : <type pointeur> *nom_pointeur;

```
int x = 10 ;
int *px; // Réserve un emplacement pour stocker une adresse mémoire.
px = &x; // Ecrit l'adresse de x dans cet emplacement.
Printf(‘’%d’’, *px); // afficher 10.
```



L'opérateur *permet d'accéder au contenu de la variable pointée :

```
*px = 20; // Maintenant x est égal à 20.
```



Après les instructions:

```
<type> a;
<type> *p;
p = &a;
```

p pointe sur a, p désigne &a (adresse de a) et *p désigne a (valeur ou contenu de a).

4) Allocation dynamique de la mémoire <stdlib.h>

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation.

Comment faire pour réserver de l'espace au moment de l'exécution ?

Solution : Il faut allouer la mémoire dynamiquement.

Allocation dynamique : La réservation de la mémoire pendant l'exécution du programme.

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

Comment faire pour connaitre la grandeur d'un objet ?

L'opérateur unaire sizeof() fournit la grandeur pour un objet de n'importe quel type.

```
int nbr;
int *p;
printf(" Entrez le nombre de valeurs :");
scanf("%d", &nbr);
p = (int*)malloc(nbr*sizeof(int));
```

L'operateur sizeof() : Renvoie la taille en octets de l'argument (soit un type, soit une variable ou une expression) .

```
short A;
char B[5][10];
sizeof(A) ⟹  2
sizeof(B) ⟹  50
sizeof(4.25) ⟹  8
sizeof("abcd“) ⟹  5
sizeof(float) ⟹  4
sizeof(double) ⟹  8
```

a) La fonction malloc

void *malloc (size_t size);

Alloue size octets et renvoie un pointeur sur la mémoire allouée ; La zone de mémoire n'est pas initialisée.

```
long *a = (long *)malloc( sizeof(long) );
int n = 20;
long *tab = (long *) malloc(n * sizeof(long));
```

b) La function calloc

void *calloc (size_t nb, size_t size);

Alloue et remplit de 0 (zéro) la mémoire nécessaire pour nb éléments de size octets et renvoie un pointeur sur la zone allouée.

Attention, ce n'est pas la même signature que malloc !

```
int n = 100;
long *tab = (long*)calloc(n, sizeof(long)); /* n fois 0 */
```

c) La fonction realloc

void *realloc (void *p, size_t size);

Reduit (ou augmente) la taille du bloc de mémoire pointé par p à une taille de size octets ; conserve les size premiers octets a l'adresse p. Le reste de la nouvelle zone n'est pas initialise.

```
int *tab;
tab = (int*) calloc(2, sizeof(int));
tab[0] = 33;
tab[1] = 55;
tab = (int*)realloc(tab, 3*sizeof(int) );
tab[ 2 ] = 77;
```

Attention! il peut toujours se produire des erreurs lors de l'allocation dynamique de mémoire : il faut TOUJOURS vérifier que le pointeur retourné lors de l'allocation n'est pas égal à NULL!

Exemple :

```
int *tab1, *tab2;
                    1 tab1 = (int*) malloc( 5*sizeof(int) );
                     if( tab1 == NULL ) {
                         exit(EXIT_FAILURE);
                     }
OU
                    2 if( ( tab2 = (int*)malloc(50*sizeof(int)) ) == NULL )
                   {
                         exit(EXIT_FAILURE);
                      }
```

d) La fonction free

Lorsqu'un emplacement mémoire n'est plus utilisé, il est important de libérer cet espace. La fonction free réalise cette tâche.

void free(void *p) :

• Libère l'espace mémoire pointé par p qui a été obtenu lors d'un appel à malloc, calloc ou realloc.

• Sinon, ou si il a déjà été libéré avec free(), le comportement est indéterminé.

La fonction free peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par malloc. La fonction free ne change pas le contenu du pointeur; il faut mettre NULL dans le pointeur immédiatement après avoir libéré le bloc de mémoire. Si on ne libère pas explicitement la mémoire à l'aide de free, elle est libérée automatiquement à la fin du programme.

5) Chaînes de caractères <string.h>

On a vu que les chaînes de caractères sont des tableaux, donc des pointeurs avec quelques subtilités. Attention, les chaînes de type char* initialisées avec des " " sont statiques et constantes : on ne peut pas les modifier

```
char s1[] = "toto";
char *s2 = "titi";
s1[0] = 'x'; /* OK */
s2[0] = 'y'; /* NON! */
```

Elles peuvent également être déclarées comme des char * et allouées dynamiquement (malloc(), ...).

```
char *s3 = (char *)malloc( 5*sizeof(char) );
if (s3!=NULL)
   s3[0] = 't'; /* OK */
printf(s3); /* NON! */
printf("%s",s3); /* OK */
```

a) Fonctions spéciales:

```
Size_t strlen (const char *s);
```

Renvoie la longueur de la chaîne de caractères s, sans compter le caractère nul '\o' final.

```
char *strcpy (char *dest, const char *src);
```

Copie la chaîne pointée par src (y compris le '\0' final) dans la chaîne pointée par dest. Les 2 chaînes ne doivent pas se chevaucher et dest doit être assez grande pour la copie. Elle renvoie un pointeur sur la chaîne dest.

```
char *strcat (char *dest, const char *src);
```

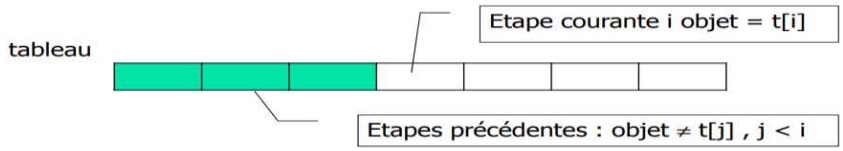
Ajoute la chaîne src à la fin de la chaîne dest en écrasant le '\0' a la fin de dest, puis en ajoutant un nouveau '\0' final. Les chaînes ne doivent pas se chevaucher, et dest doit être assez grande pour le résultat. Elle renvoie un pointeur sur dest.

6) Techniques de programmation

a) Recherche

Principe : comparer les uns après les autres tous les éléments du tableau avec l'objet recherché.
Arrêt :

- si l'objet a été trouvé ;
- ou si tous les éléments ont été passés en revue et l'objet n'a pas été trouvé.



i. *Recherche séquentielle*

```
int rechSequentielle(int t[], int x, int max) {
      int i; int arret= 0;
      i= 0;
      while( (i < max) && ( arret == 0)) {
            if (t[i] == x) arret = 1;
            else i++;
       }
      if (i == max) return -1;
      else return i;
}
```

ii. *Recherche dichotomique (tableau trié au préalable)*

Principe : A chaque étape :

- découpage du tableau en deux sous-tableau à l'aide d'un indice médian (tableau inférieur) et (tableau supérieur) comparaison de la valeur située à l'indice médian et de l'objet recherché,
- Si l'objet recherché est égal à la valeur t[médian] , arrêter la recherche et l'indice est égal à la médian ,
- Sinon si l'objet est supérieur à la valeur t[médian] relancer la recherche avec le tableau supérieur,
- Sinon relancer la recherche avec le tableau inférieur.

```
int rechDichotomique(int t[], int x, int max) {
        int min = 0 , med, pos, arret = 0;
        do{
              med= (max+min)/2;
              if( x == t[med]) {
                        arret = 1; pos = med;
              } else {
                        if( x > t[med])
                           min = med + 1;
                        else
                           max = med - 1;
              }
              if(min >= max) {
                        arret = 1; pos = -1;
              }
        } while(arret != 1);
return pos;
}
```

b) Tri

Tri des données : Réorganiser une suite d'enregistrements, de telle sorte qu'ils soient ordonnés.

i. Tri par sélection

Principe: A chaque étape, chercher le plus petit élément : t[m] et le placer au début en permutant t[o] et t[m].

```
void triSel(int t[], int n) {
      for (int i = 0; i < n; i++){
            int m = i;
             for( int j = i+1; j < n-1; j++)
                  if(t[j] < t[m]) m = j;
            permuter(t, i, m);
      }
}
```

ii. Tri par Bulles

Principe : Placer le plus petit élément en début en comparant les cases contiguës. Réitérer l'opération.

```
void triBulle(int t[ ], int n) {
      for (int i = 0; i < n ; i++)
            for (int j = n-1; j > i ; j--)
                  //Remonter les plus petits
                   if (t[j-1] > t[j])
                        permuter (t, j-1, j);
 }
```

iii. Tri par Bulles optimisé

Principe : Si l'on constate à l'étape i, qu'aucun échange n'a été effectué alors arrêter l'algorithme.

```
void triBulle(int t[], int n) {
      stop = false;
      i = 0;
      do{
            stop = true;
            for (int j = n-1; j > i; j--)
                  if (t[j-1]  >  t[j]) {
                     permuter(t, j-1, j);
                     stop = false;
                  }
                    i++;
      } while( stop == false && i < n );
}
```

iv. Tri par insertion

Principe : On découpe le tableau en deux partie : une parie triée (inférieur) et une partie supérieur non encore triée. A chaque étape, on insère un élément de la partie supérieur dans son emplacement dans la partie triée.

```
void triInsertion(int t[], int n) {
      for (int I = 1; i < n ; i++){
            //Invariant : t[0..i-1] déjà trié
            j = i;
            //recherche la place de t[i] dans t[0 ... i-1]
            while( (j>0) && (t[j-1] > t[i]))
                        j--;
            // On pousse t[j..i-1] vers la droite
            tmp = t[i];
            for (int k = i; k > j; k--)
                  t[k] = t[k-1];
             t[j] = tmp;
      }
}
```

Chapitre 2: Gestion des fichiers

1) Généralités

Définition : Un fichier est un ensemble de données qui peuvent être organisées, stockées sur une mémoire de masse ou sur un support de sauvegarde (disquette, disque dur, CD/DVD, bande DAT). Il peut contenir du texte, de la vidéo, ou des données pour des applications.

Opérations standard à mener pour travailler avec un fichier

a) **Ouvrir un fichier :** lui associer une variable que l'on appelle descripteur de fichier qui permettra l'accès au fichier.
b) **Lire ou écrire** des informations à partir de ce descripteur avec des fonctions spécialement prévues pour les fichiers.
c) **Fermer le fichier,** indiquer que l'on a terminé de travailler avec ce fichier : important dans un environnement multitâches et/ou multi-utilisateur.

Les types de fichiers

a) **Les fichiers 'texte', ou fichiers à accès séquentiel :** les données n'ont aucune structure particulière, on doit lire caractère par caractère ou ligne par ligne. Le caractère séquentiel oblige de parcourir le fichier depuis le début pour accéder à une donnée particulière.

Attention, le fichier ne contient pas forcément que du texte ! Le terme 'texte' est employé parce que le fichier est vu comme une suite de caractères (pas forcément lisibles), les caractères sont des entiers.

b) **Les fichiers binaires ou fichiers à accès direct (aléatoire) :** fichiers organisés par une structure sous-jacente : il est constitué non pas de caractères mais d'enregistrements (au même titre que les struct).

Connaissant la taille d'un enregistrement, on peut se rendre directement à un endroit précis du fichier : d'où le caractère direct.

2) Manipulation des fichiers

Définition : Le descripteur de fichier est un canal de communication entre le programme et un fichier physique situé quelque part sur le disque.

Avant d'ouvrir un fichier, il faut déclarer un descripteur qui sera associé avec ce fichier.

c) Déclaration d'un pointeur sur un fichier

Le type spécifique FILE est utilisé toujours pour déclarer un pointeur sur notre fichier, c'est le descripteur de fichier.

Syntaxe de déclaration :

```
FILE *<descripteur> ;
```

On peut ensuite procéder aux différentes opérations, toujours dans l'ordre: ouvrir, lire ou écrire et puis fermer le fichier.

d) Ouverture d'un fichier

On a vu que l'ouverture du fichier veut dire faire le lien entre le fichier sur disque (qui porte un nom bien défini) et le descripteur. Pour ce faire, on dispose de la fonction fopen.

Prototype :

```
FILE *fopen(char *nom_du_fichier, char *mode_d_ouverture);
```

Le `nom_du_fichier` : c'est le nom classique (relatif ou absolu) de fichier, par exemple :

valeurs.txt ou C:\essais\test\values.dat

Attention au caractère \ dans les chaînes de caractère! Il faut écrire \\ dans les chaînes de caractère !

"C:\\essais\\test\\values.dat"

Le `mode_d_ouverture` : Détermine le type d'accès autorisés pour le fichier.

Les résultats donnés par fopen :

Si l'ouverture n'a pas fonctionné correctement, fopen renvoie la valeur NULL (pointeur indéfini), sinon, le pointeur de fichier est initialisé (la valeur elle-même n'est pas intéressante).

i. Les modes d'ouverture

- " **r**" **(Read) :** accès en lecture seulement, le fichier doit exister.
- " **w**" **(Write) :** ouvre un fichier vide pour y écrire. Si le fichier existe déjà, son contenu est détruit.
- " **a**" **(Append) :** ouvre un fichier en écriture, si le fichier existe déjà, écritures à la suite des données existantes.
- "**r+**" **:** ouverture en lecture/écriture, le fichier doit déjà exister;
- "**w+**" **:** ouverture d'un fichier vide en lecture/écriture, si le fichier existe déjà, son contenu est détruit.
- "**a+**" **:** comme " a".

Remarque : On peut préciser si on travaille avec des fichiers texte ou binaire, en ajoutant 't' ou 'b' au mode d'ouverture :

"**rt**" **: (Read Text)**, "**wb**" **: (Write Binary)** ou encore "**wt+**" **: (Write Text, lecture/écriture)**.

ii. Gestion du résultat de fopen :

Si le fichier n'est pas ouvert correctement, inutile de continuer le programme : on ne pourra rien faire, cela peut être dues : à un mauvais nom de fichier la plupart du temps! Alors, il faut toujours faire un test systématique de la valeur du descripteur de fichier juste après fopen ; si l'ouverture s'est bien passée (et seulement à cette condition) on peut continuer avec la suite du programme.

Exemple de programme où l'on cherche à lire le contenu d'un fichier de type texte.

```
#include<stdio.h> /* pour fopen également*/
void main() {

      FILE *monFic;
      monFic=fopen("toto.txt","rt"); /* rt : Read Text */

      if (monfic==NULL) /* test de l'ouverture */
      {
            printf("fichier non ouvert !\n");
      } else {
            /* suite du programme */
```

```
            /* parmi lequel lecture des valeurs dans le fichier */
      }
}
```

Autre forme 'classique' : le test est fait sur la même ligne que l'ouverture. Réalise exactement la même chose !

```
#include void main() {
      FILE *monFic;
      if ( (monFic=fopen("toto.txt","rt")) == NULL) {
            printf("fichier non ouvert !\n");
      } else {
            /* suite du programme */
            /* parmi lequel lecture des valeurs dans le fichier */
      }
}
```

3) Fermeture de fichier

Quelque soit le type de fichier utilisé, texte ou binaire, il faut, après avoir fini de travailler avec, le fermer : ajoute la fin de fichier.

Utilisation de la fonction **fclose** :

```
int fclose(FILE* <descripteur>);
```

La valeur de retour vaut o si la fermeture s'est bien passée. Inutile de tester cette valeur !

On n'effectue le **fclose** que si le fichier était bien ouvert, c'est à dire si le **fopen** du fichier s'est bien déroulé.

À retenir quand on travaille avec un fichier :

- ouverture
- si l'ouverture s'est bien déroulé alors opérations et traitements
- puis fermeture.

4) Traitement des fichiers textes

Pour lire ou écrire des valeurs dans un fichier texte le langage C nous offre des fonctions spéciales destinées à répondre à ce besoin.

Principe : ressemblance avec saisies (scanf) et affichage (printf) mais avec des sources (de saisie) et destinations (d'affichage) différentes. En fait, une saisie est une lecture depuis le clavier. On peut faire ces lectures à partir d'autres dispositifs d'entrées ou flots d'entrée. Un fichier peut jouer ce rôle. De même, un affichage est une écriture vers l'écran. Écritures possibles vers d'autres dispositifs de sortie ou flots de sortie. Un fichier peut aussi jouer ce rôle.

a) Lecture d'un fichier

Pour lire des données à partir d'un fichier ouvert, on utilise la fonction **fscanf().**

Prototype :

```
int fscanf(FILE *<descripteur>, char *<chaine_format>, <variables>);
```

La valeur de retour est le nombre de lectures réussies avec le format spécifié.

`<descripteur>:` le descripteur du fichier avec lequel on veut travailler !

Les autres paramètres sont comme pour **scanf.**

Remarque : Le descripteur, indique, entre autres, l'endroit du fichier où se déroule la prochaine opération. Après une ouverture, le descripteur indique la première valeur (en fait le premier octet du fichier). A chaque fois qu'une opération aura lieu, le descripteur indiquera l'endroit où s'est terminée cette opération, et donc là où aura lieu la suivante.

Exemples : si on lit 4 octets, le descripteur indiquera 4 octets plus loin que là où il était.

i. Lecture caractère par caractère

On peut lire le fichier caractère par caractère avec le format %c.

Algorithme :

```
Tant que l'on n'est pas à la fin du fichier Faire
    lire un caractère dans le fichier
    afficher à l'écran le caractère lu
FinTQ
```

La fin du fichier est repéré grâce à l'emploi de la fonction feof :

Prototype : `int feof(FILE *<descripteur>);`

Indique si on est à la fin du fichier si la valeur de retour est non nulle, sinon la valeur de retour est o.

La fin du fichier est marquée par un caractère spécial (CTRL-Z) de fin de fichier. Cette fin de fichier est aussi nommée EOF (End Of File).

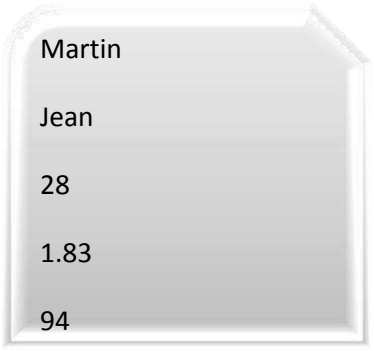
```
#include<stdio.h>
void main(){
      FILE *f; // descripteur
      char c;
      f = fopen("data.txt","rt");

      if (!f) {
                  printf("fichier non ouvert !\n");
       } else {
                  printf("fichier ouvert !\n");
                  while(feof(f) == 0) {
                        fscanf(f,"%c",&c);
                        printf("%c ",c);
                  }
      }
       fclose(f); /* Fermeture de fichier */
}
```

ii. Lecture formatée

Dans le cas où on a une idée sur la manière dont les données sont organisées (format du fichier est connu) on peut formater la fonction **fscanf** avec les formats nécessaires pour effectuer la lecture.

Un exemple de fichier texte : une petite fiche de renseignements. On a collecté quelques informations sur une personne : nom, prénom, âge, taille en m et poids en kg. Le fichier texte est le suivant :



data.txt

En fait, avec les délimiteurs, il est le suivant:

Martin\nJean\n28\n1.83\n94\nCTRL-Z

Choix des formats de lecture :

1 chaîne de caractères : %s

1 chaîne de caractères :%s

1 entier : %d

1 float : %f

1 entier : %d

donc 5 **fscanf** à la suite avec les formats appropriés.

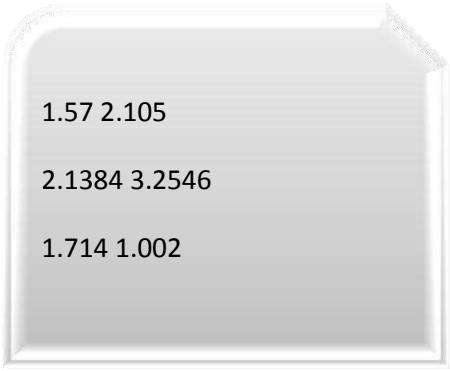
Alors, on peut faire la lecture comme dans le programme suivant :

```
#include <stdio.h>
void main()
{
      FILE *fichier;
      // !! On déclare les variable qui vont contenir
      // !! les données lues
      char nom[50], prenom[50];
      int age, poids;
      float taille;
      fichier = fopen("fiche.txt","rt");

      if (!fichier)
            {
                  printf("fichier non ouvert !\n");
            }else{
                  fscanf(fichier,"%s", nom);
                  fscanf(fichier,"%s", prenom);
                  fscanf(fichier,"%d", &age);
                  fscanf(fichier,"%f", &taille);
```

```
                fscanf(fichier,"%d", &poids);
                printf("donnees lues:\n %s %s \n %d\n %f\n
        %d\n", nom, prenom, age, taille, poids);
            }
            fclose(fichier); /* fermeture du fichier */
        }
```

Un autre exemple : Possibilité d'avoir plusieurs valeurs sur une ligne, attention à bien choisir le format en conséquence. Fichier texte nommé **vals.txt** contenant une suite de coordonnées de points en 2 dimensions.



Chaque ligne comporte deux éléments à lire : plusieurs possibilités

- lire individuellement chaque valeur float par un **fscanf** ;
- lire les deux valeurs de chaque ligne par un seul **fscanf** (lire ligne par ligne).

Solutions :

- on fait 6 lectures d'un float dans la boucle, sans gérer la fin de fichier :

```
#include <stdio.h>
void main()
{
      FILE *f1;
      // !! On déclare les variable qui vont contenir
      // !! les données lues
      float val1, val2;

      f1 = fopen("vals.txt","rt");
      if (!f1)
      {
            printf("fichier non ouvert !\n");
      }else{
            int i = 1;
            for(i = 0; i < 3 ; i++)
            {
                  fscanf(f1 ,"%f%f", &val1, &val2);
                  printf("%f  %f\n", val1, val2);
            }
      }
      fclose(f1); /* fermeture du fichier */
}
```

- on fait 3 lectures d'un float dans la boucle, sans gérer la fin de fichier :

```
#include <stdio.h>
void main()
{
	FILE *f1;
	// !! On déclare les variable qui vont contenir
	// !! les données lues
	float val1;
	f1 = fopen("vals.txt","rt");
	if (!f1)
	{
		printf("fichier non ouvert !\n");
	}else{
		int i = 1, j = 1;
		for(i = 0; i < 6 ; i++)
		{
			fscanf(f1 ,"%f", &val1);
			printf("%f", val1);
			if(j > 0){
				printf(" "); j = -1;
			}else{
				printf("\n"); j = 1;
			}
		}
	}
	fclose(f1); /* fermeture du fichier */
}
```

b) Ecriture dans un fichier

Pour lire des données à partir d'un fichier ouvert, on utilise la fonction **fprintf().**

Prototype :

```
int fprintf(FILE *<descripteur>, char *<chaine_format>, <variables>);
```

La valeur de retour est le nombre de caractères écrits.

`<descripteur>:` le descripteur du fichier avec lequel on veut écrire !

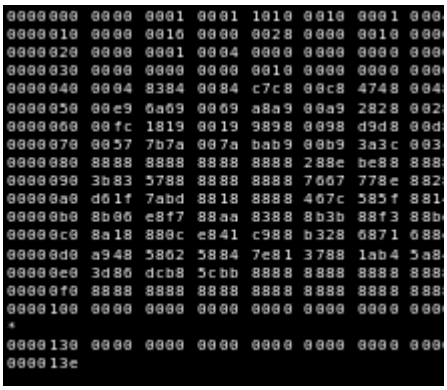
Les autres paramètres sont comme pour **printf**. Un '\n' écrit dans un fichier provoquera un passage à la ligne. Le caractère de fin de fichier est ajouté automatiquement, pas besoin de le mettre.

Exemple : on veut écrire un fichier texte (nouveau) pour stocker les renseignements vus tout à l'heure.

```
#include <stdio.h>
void main()
{
	char nom[50] = "Martin";
	char prenom[50] = "Jean";
	int age = 28;
	float taille = 1.84;
	int poids = 95;
	FILE *ficdata;
```

```
	ficdata = fopen("donnees.txt","wt");
	if (ficdata == NULL)
	{
		printf("fichier non ouvert\n");
	}
	else
	{
		fprintf(ficdata,"%s\n%s\n%d\n%f\n%d\n", nom, prenom, age,
taille, poids);
		fclose(ficdata);
		printf("Ecriture reussite!");
	}
}
```

5) Traitement des fichiers binaires



Si les données sont non formatées, donc pas de repère, pas de lecture/écriture formatée ni caractère par caractère. Ceci est associé au stockage de données complexes comme les structures (struct).

La gestion peut être séquentielle comme pour les fichiers texte, mais il y a la possibilité d'accès direct en déplaçant le pointeur (descripteur) à un endroit précis du fichier sans avoir à tout lire depuis le début.

On doit lire ou écrire paquet de données par paquet de données.

On représentera un fichier binaire comme une suite de données complexes qui ont toutes le même format : comme un tableau qui serait situé sur le disque dont chaque case comprenant une donnée complexe (struct, ...).

a) Ecriture d'un fichier binaire

Pour lire des données à partir d'un fichier ouvert, on utilise la fonction **fwrite().**

Prototype :

```
size_t fwrite(void *<tampon>, size_t size, size_t nb, FILE *<descripteur>);
```

La valeur de retour est le nombre de lectures réussies avec le format spécifié.

Rôle des différents paramètres :

- Type void* : indique un pointeur sur un type qui n'est pas encore défini, permet de la souplesse en utilisant le transtypage.

- Type size_t : indique un entier non signé qui indique une taille (normalement donné par l'emploi de sizeof).
- <tampon>: zone de mémoire où se trouvent les éléments à écrire dans le fichier.
- size : la taille en octets d'un élément complexe à lire, cela permet de décaler correctement le descripteur.
- nb : un entier non signé indiquant le nombre d'éléments complexes à lire. Le nombre total d'octets lus sera donc (size * nb).
- <descripteur>: le descripteur du fichier avec lequel on veut travailler !

La valeur de retour indique le nombre d'éléments effectivement écrits.

Exemple : On a un tableau contenant des struct, à écrire dans un fichier binaire. La structure est définie ainsi :

```
typedef struct complex{
        float re;
        float im;
}cplx;
```

Que va faire le programme?

Déclarer un tableau contenant des valeurs de type cplx;

Initialiser le tableau;

Ouvrir un fichier binaire en écriture (sans append)

Si l'ouverture s'est bien déroulée Alors

Écrire les éléments du tableau dans le fichier

fermer le fichier.

FinSi

Solution :

```
#include <stdio.h>
/* définition du type complex */
typedef struct complex{
      float re;
      float im;
 }cplx;

void main()
{
      cplx tabc[3] = {{1.0, 1.0}, {2.0, 3.0}, {3.0, -1.0}};
      FILE *ficbin;
      if ((ficbin = fopen("comp.bin","wb")) != NULL)
      {
            fwrite(tabc, sizeof(cplx), 1, ficbin);
            fwrite(tabc+1, sizeof(cplx), 2, ficbin);
            printf("Ecriture reussite!");
            fclose(ficbin);
      }
}
```

où remplacer les 2 fwrite par un seul : `fwrite(tabc, sizeof(cplx), 3, ficbin);`

b) Lecture d'un fichier binaire

Pour lire des données à partir d'un fichier ouvert, on utilise la fonction **fread().**

Prototype :

```
size_t fread(void *<tampon>, size_t size, size_t nb, FILE *<descripteur>);
```

La valeur de retour est le nombre de lectures réussies avec le format spécifié.

Rôle des différents paramètres :

- Type void* : indique un pointeur sur un type qui n'est pas encore défini, permet de la souplesse en utilisant le transtypage.
- Type size_t : indique un entier non signé qui indique une taille (normalement donné par l'emploi de sizeof).
- `<tampon>`: zone de mémoire destinée à recevoir les données lues : en fait, lorsque l'on fait une lecture de valeur (scanf, fscanf), le résultat est rangé en mémoire (dans des variables); toujours les données récupérées doivent être rangée quelque part.
- size : la taille en octets d'un élément complexe à lire, cela permet de décaler correctement le descripteur.
- `nb` : un entier non signé indiquant le nombre d'éléments complexes à lire. Le nombre total d'octets lus sera donc (size * nb).
- `<descripteur>:` le descripteur du fichier avec lequel on veut travailler !

La valeur de retour est le nombre d'éléments complexes effectivement lus grâce à **fread**.

Exemple : Lecture d'un fichier binaire pour placer les valeurs lues dans un tableau : il faut par contre connaître par avance le nombre d'éléments stockés dans le fichier ou utiliser une boucle pour lire le fichier. Dans l'exemple qui suit, on va reprendre le fichier créé par l'écriture précédente et le relire. On garde le type créé **cplx**, défini de la même manière. Programme très ressemblant, on utilisera **fread** à la place de **fwrite** !

```
Déclarer un tableau contenant des valeurs de type cplx;

            Declarer le tableau;

            Ouvrir un fichier binaire en lecture (sans append)

            Si l'ouverture s'est bien déroulée Alors

                  Lire les éléments du fichier vers le tableau

                  fermer le fichier.

            FinSi
```

Solution :

```
#include <stdio.h>
/* définition du type complex */
typedef struct complex{
      float re;
      float im;
 }cplx;

void main()
{
```

```
        cplx tabc[3]; /* declaration, non initialisé, on va lire */
        FILE *ficbin;
        if ( (ficbin=fopen("comp.bin","rb")) != NULL)
        {
              fread(tabc,sizeof(cplx),3,ficbin);
              int i;
              for(i = 0; i < 3 ; i++){
                    printf("%f + i(%f)\n", tabc[i].re, tabc[i].im);
              }
              fclose(ficbin);
        }
}
```

c) Accès direct

La fonction **fseek** permet de se déplacer dans le fichier d'un certain nombre d'octets :

Attention à convertir ce nombre ! Par exemple, on doit aller lire le k-ième élément stocké dans un fichier binaire, la valeur des éléments précédents ne nous intéresse pas.

Prototype de la fonction fseek :

```
int fseek( FILE *<descripteur>, long depl, int origin);
```

fseek renvoie la valeur o si elle a bien fonctionné.

Rôle des paramètres :

- `<descripteur>`: le fichier sur lequel on travaille
- depl : valeur du déplacement en nombre d'octets : si l'on connaît la taille en octets d'un élément (on le peut grâce à sizeof), on multiplie par le nombre d'éléments pour obtenir cette valeur.
- Origin : indique à partir de quel endroit du fichier est calculé le déplacement. Il a 3 valeurs possibles seulement :
  - **SEEK_CUR** : par apport à la position actuelle
  - **SEEK_END** : par rapport à la fin du fichier
  - **SEEK_SET** : par rapport au début du fichier

Exemple : avec le fichier contenant les valeurs des nombres complexes, on veut seulement lire la troisième valeur sans se préoccuper des autres :

- on va ouvrir le fichier
- déplacer le descripteur pour aller directement là où se trouve les valeurs du 3ème élément.

On doit se décaler de 2 éléments à partir du début du fichier :

calcul du nombre d'octets dont il faut se déplacer : 2*sizeof(cplx)

```
#include <stdio.h>
/* définition du type complex */
typedef struct complex{
        float re;
        float im;
 }cplx;
```

```
void main()
{
	cplx complexe; /* declaration, non initialisé, on va lire */
	FILE *ficbin;
	if ( (ficbin=fopen("comp.bin","rb")) != NULL)
	{
		fseek(ficbin, 2*sizeof(cplx), SEEK_SET);
		fread(&complexe, sizeof(cplx), 1, ficbin);
			printf("%f + i(%f)\n", complexe.re, complexe.im);
		fclose(ficbin);
	}
}
```

Autre exemple : toujours avec l'exemple sur les nombres complexes : après une lecture par **fread**, on veut revenir un élément en arrière dans le fichier :

on se déplacera à partir de l'endroit où l'on se trouve, le nombre d'octets du déplacement sera donné par : 1*sizeof(t_complex); de plus le déplacement se fait en arrière (valeur négative): faire très attention à la valeur du déplacement, sinon risque d'être perdu dans le fichier.

```
#include <stdio.h>
/* définition du type complex */
typedef struct complex{
      float re;
      float im;
 }cplx;

void main()
{
      cplx complexe, tabc[3]; /* declaration, non initialisé, on va lire */
      FILE *ficbin;
      if ( (ficbin = fopen("comp.bin","rb")) != NULL)
      {
            fread(tabc,sizeof(cplx),3,ficbin);
            int i;
            for(i = 0; i < 3 ; i++){
                  printf("%f + i(%f)\n", tabc[i].re, tabc[i].im);
            }
            // deplacement en arrière
            fseek(ficbin, -1*sizeof(cplx), SEEK_CUR);
            fread(&complexe, sizeof(cplx), 1, ficbin);
            printf("Dernier element: %f + i(%f)\n", complexe.re,
complexe.im);
            fclose(ficbin);
      }
}
```

6) Gestion de la fin de fichier

Attention à l'emploi de la fonction **feof** : on a précisé qu'elle indique si oui ou non on est à la fin du fichier. En fait, elle indique si la dernière opération menée sur le fichier a rencontré la fin de fichier ou non.

```
while(!feof(f1)){
      fscanf(f1,…, …);
      //Traitement (Affichage, …etc);
}
```

Toujours on est obligé de faire la lecture suivante pour rencontrer la fin de fichier. Par contre,

on doit tester la valeur de **feof** avant de ranger la valeur lue.

Solution : Rajouter un **feof** supplémentaire, ou utiliser une variable **fin** booléenne pour marquer que l'on a rencontré la fin du fichier.

```
while(!feof(f1)){
                  fscanf(f1, …, …);
                  if(!feof(f1))
                        //Traitement (Affichage, …etc);
            }
```

OU

```
int fin = 0;
while(!fin){
                  fscanf(f1, …, …);
                  if(!feof(f1))
                        //Traitement (Affichage, …etc);
                  else
                        fin = 1 ; // On a fini
            }
```

Exemple :
On considère un fichier contenant : 0123456789E
On lit caractère par caractère :

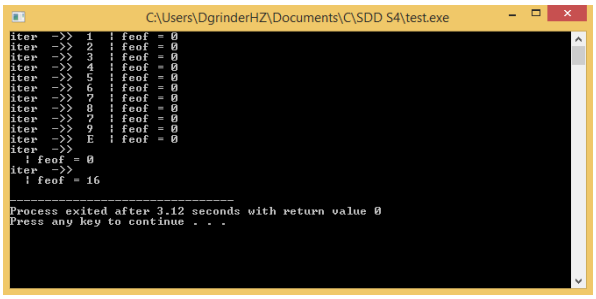
- Premier teste :

```
char c;
while(!feof(f1)){
      fscanf(f1, "%c", &c);
      printf("iter  ->>  %c  | feof = %d\n", c, feof(f1));
}
```

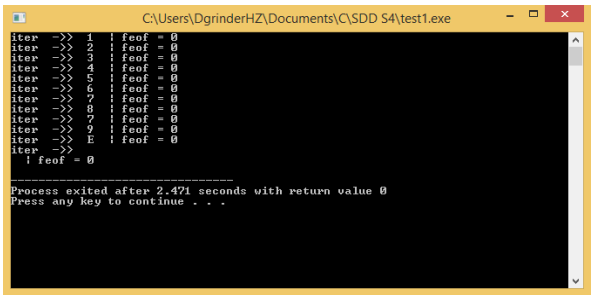
- Deuxième teste :

```
char c;  int fin = 0;
while(!fin){
            fscanf(f1, "%c", &c);
            if(!feof(f1))
                  printf("iter  ->>  %c  | feof = %d\n", c, feof(f1));
            else
                  fin = 1 ; // On a fini
            }
```

Affichage :



Premier Teste



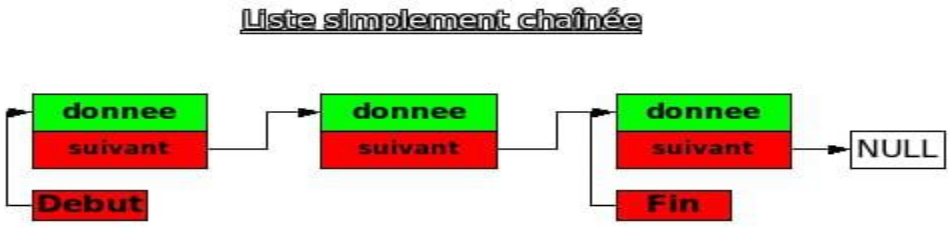
Deusieme teste

Chapitre 3 : Les listes simplement chaînées

1) Généralité

Définition : La liste chaînée est une structure de donnée composée de plusieurs éléments qui sont liés deux à deux entre eux (chaîne).

Une liste chaînée possède une tête (début), une queue (fin) et un ensemble d'éléments. Chaque élément (maillon) contient l'information mémorisé et un pointeur sur l'élément suivant. L'élément suivant de la queue de la liste n'existe pas, la valeur NULL en C.



Une liste est accessible par l'adresse de sa première composante (début/tête). On supposera dans la suite que les valeurs à mémoriser sont d'un type structures, composé de plusieurs champs, nommé **information**. Les listes chaînées peuvent être utilisées quand plusieurs opérations d'insertion et/ou suppression d'éléments sont nécessaires.

La notion de liste chaînée n'est pas liée à une manière concrète de matérialiser la relation successeur. Les éléments sont référencés par leurs adresses dans la mémoire. Chaque élément est alloué dynamiquement, au moment où il commence à être utile. Contrairement aux tableaux, il se peut que les éléments qui composent la liste ne soient pas placés dans l'ordre en mémoire et encore moins de façon contiguë.

Autre caractéristiques :

- La taille est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet.
- Pour déclarer une liste chaînée il suffit de créer le pointeur qui va pointer sur le premier élément de votre liste chaînée, aucune taille n'est à spécifier donc.
- Il est possible d'ajouter, de supprimer, d'intervertir des éléments d'une liste chaînées en manipulant simplement leurs pointeurs.
- Il permet de manipuler des ensembles dynamiques.
- Les listes sont adaptées au parcours séquentiel d'un ensemble discret.
- Les listes sont à la base des Files et des Piles
- Pour les listes les éléments peuvent être éparpillés dans la mémoire. La liaison entre les éléments se fait grâce à un pointeur. En réalité, dans la mémoire la représentation est aléatoire en fonction de l'espace alloué.
- Le pointeur suivant du dernier élément doit pointer vers NULL (la fin de la liste).
- Pour accéder à un élément, la liste est parcourue en commençant avec la tête, le pointeur suivant permettant le déplacement vers le prochain élément. Le déplacement se fait dans une seule direction (séquentielle asymétrique), du premier vers le dernier élément.

2) Création et déclaration en C d'une liste

a) Déclaration

Avant de déclarer une liste, il faut créer les structure necesssaire pour sa construction :

Information, Maillon (éléments) et Liste

L'élément de la liste contiendra un champ **donnee** de type i**nformation** qui dépend du problème posé et un pointeur **suivant**. Le pointeur suivant doit être du même type que l'élément, sinon il ne pourra pas pointer vers l'élément. Le pointeur **suivant** permettra l'accès vers le prochain élément.

Donc, la première chose à faire c'est de définir le type cette information :

```
struct Info
{
      type_1 champs_1 ;
      type_2 champs_2 ;
      …
      type_n champs_n ;
};
typedef struct Info Information;
```

Après, on passe à la définition de l'élément «maillon » :

```
struct maillon
{
      Information donnee;
      struct maillon *suivant;
};
typedef struct maillon Maillon;
```

Enfin, on peut définir notre liste simplement chainée :

```
struct liste
{
      Maillon *debut;
      Maillon *fin;
      int taille;
};
typedef struct liste Liste ;
```

b) Création

Dans cette partie on crée les fonctions d'initialisation d'un élément (maillon) et d'une liste :

Les champs champs_1, champs_2, … et champs_n dépend de **Information**.

```
Maillon *CreerElement(){
      Maillon *m;
      m = (Maillon*)malloc(sizeof(Maillon)) ;
      if (m == NULL) {
            printf("Allocation non reussie\n");
            exit(-1);
      }
      else {
            m->donnee.champs_1=0;
            …
            m->donnee.champs_n = 0;
            m->suivant = NULL;
      }
```

```
return m;
}
```

Pour la liste c'est toujours cette fonction qu'on utitlise :

```
Liste* CreerListe(){
        Liste *L = (Liste*)malloc(sizeof(Liste)) ;
        if (L == NULL) {
                printf("Allocation non reussie\n");
                exit(-1);
        }
        else {
                L->debut = NULL;
                L->fin = NULL;
                L->taille = 0;
        }
return L;
}
```

Une fois terminé toutes ses opérations on peut procéder aux déclarations suivantes :

```
Liste *L ; // déclaration d'un objet de type Liste*
L = CreerListe(); // Création
…
// Déclaration et création d'un objet de type Maillon*
Maillon *element = CreerElement() ;
```

c) Accès aux champs d'une liste chainée

Pour pouvoir accéder aux différents champs d'une liste ou d'un élément d'une liste, il faut respecter les règles lies à la manipulation des pointeurs :

Puisque on va toujours declarer un pointeur sur une liste, on utilisera l'opérateur -> **(flèche).**

Exemple :

```
Liste *L;
L->debut;
L->fin;
L->taille;
L->debut->donnee;
L->debut->suivant;
(L->debut->donnee).champs1;
(L->debut->donnee).champs2;
… etc
```

d) A retenir, les points clés des listes chaînées

- ✓ Chaque élément de la liste est formé de n+1 champs :
  - n champs constituant l'**information** portée par le maillon (l'élément), dépendante du problème particulier considéré ;
  - un champ supplémentaire qui est la matérialisation de la relation successeur (le pointeur **suivant**).
- ✓ Le pointeur **début** contiendra l'adresse du premier élément de la liste ;

- ✓ Le pointeur **fin** contiendra l'adresse du dernier élément de la liste ;
- ✓ La variable entière **taille** contient le nombre d'éléments de la liste.
- ✓ Quel que soit la position dans la liste, les pointeurs **début** et **fin** pointent toujours respectivement vers le premier et le dernier élément.
- ✓ Le champ **taille** contiendra le nombre d'éléments de la liste quel que soit l'opération effectuée sur la liste.

3) Opérations sur les listes chaînées

On peut effectuer un certain nombre d'opérations sur les listes pour faciliter leur manipulation. Elles concernent, l'insertion, la suppression, la recherche, l'affichage d'une liste chaînée, ...etc. Mais tout d'abord, il faut commencer par l'initialisation.

a) Initialisation

Prototype de la fonction : void initialise_liste (Liste *L); Cette opération doit être faite avant toute autre opération sur la liste. Elle initialise le pointeur debut et le pointeur fin avec la valeur NULL, et la taille avec la valeur zero. Definition de la fonction

Remarque : La fonction Liste *CreerListe() crée, alloue et initialise les champs de la liste.

b) Insertion d'un élément dans la liste

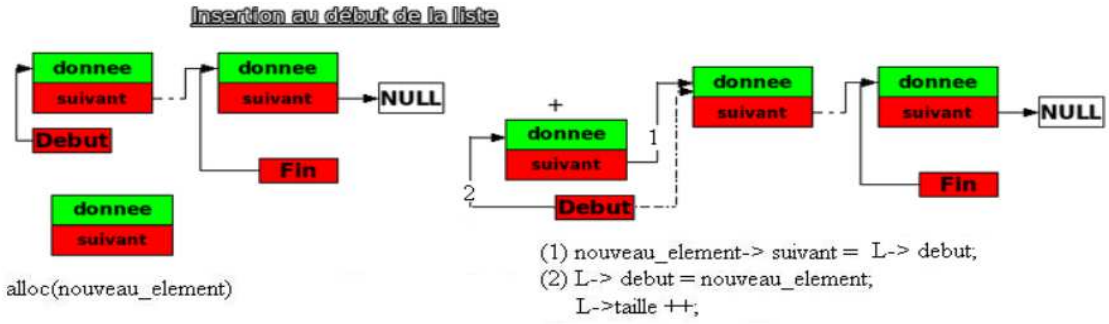
Pour ajouter (insérer) un élément dans la liste il y a plusieurs situations :

1. Insertion au début de la liste ;
2. Insertion à la fin de la liste ;
3. Insertion ailleurs dans un position dans la liste.

Voici l'algorithme général d'insertion et de sauvegarde des éléments :

1. déclaration d'élément à insérer ;
2. allocation de la mémoire pour le nouvel élément ;
3. remplir le contenu du champ **donnee ;**
4. mettre à jour les pointeurs vers le premier et le dernier élément si nécessaire ;
5. mettre à jour la taille de la liste.

i. Insertion au début de la liste



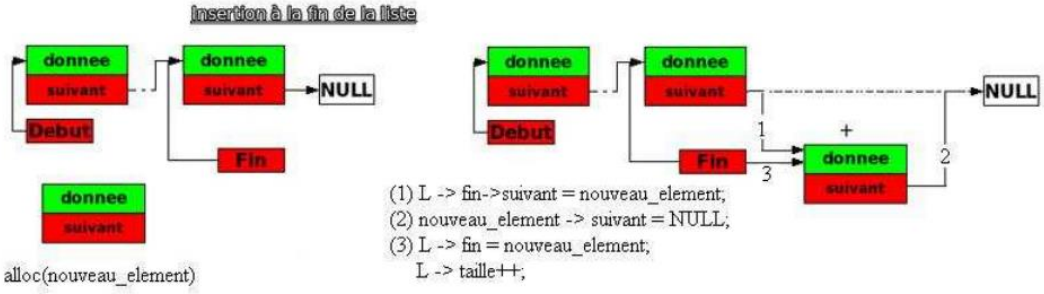
Prototype de la fonction (liste.h): `void insDebutListe (Liste *L, Information val);`

Définition de la fonction (liste.c):

```
void insDebutListe (Liste *L, Information val)
```

```
{
	Maillon *m;
	m = CreerElement();
	m->donnee = val;
	m->suivant = L->debut; // 1
	L->debut = m; // 2
	// si la liste était vide; le debut et la fin coïncident
	if (L->taille == 0)
		L->fin = m;
	L->taille++;
}
```

ii. *Insertion à la fin de la liste*



Prototype de la fonction (liste.h): `void insFinListe (Liste *L, Information val);`

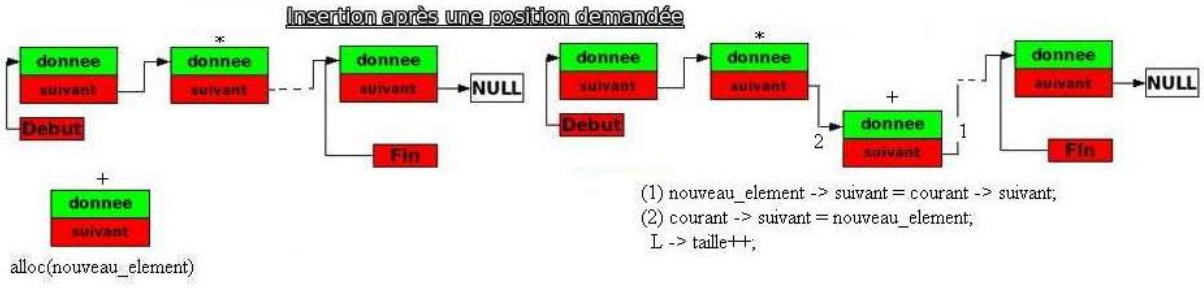
Définition de la fonction (liste.c):

```
void insFinListe (Liste *L, Information val)
{
	Maillon *m;
	m = CreerElement();
	m->donnee = val;
	m->suivant = NULL; // 2
	// Si la fin contient déjà un élément
	// Ajouter m comme son suivant
	if(L->fin != NULL)
		L->fin->suivant = m;// 1
	L->fin = m; // 3

	// si la liste était vide; le début et la fin coïncident
	if (L->taille == 0)
		L->debut = m;
	L->taille++;
}
```

iii. *Insertion ailleurs dans la liste*

L'insertion s'effectuera après une certaine position passée en argument à la fonction. La position indiquée ne doit pas être le dernier élément car si c'est le cas il faut utiliser la fonction d'insertion à la fin de la liste.



Prototype de la fonction (liste.h):

```
int insApresPositionListe (Liste *L, Information val, int pos);
```

La fonction renvoie -1 en cas d'échec sinon elle renvoie 0.

Définition de la fonction (liste.c):

```
int insApresPositionliste (Liste *L, Information val, int pos){
    // La position doit être dans [1, (L->taille)-1]
    if (pos < 1 || pos >= L->taille)
          return -1;
    Maillon *courant, *m;
    int i;
    m = CreerElement();
    m->donnee = val;

    // On se déplace vers la position pos
    courant = L->debut;
    for (i = 1; i < pos; i++) {
          courant = courant->suivant;
          }

    // Mise à jours (chainage)
    m->suivant = courant->suivant; // 1
    courant->suivant = m; // 2
    L->taille++;
return 0;
}
```

c) Retirer un élément dans la liste simplement chaînée

Pour retirer un élément dans la liste il y a plusieurs situations :

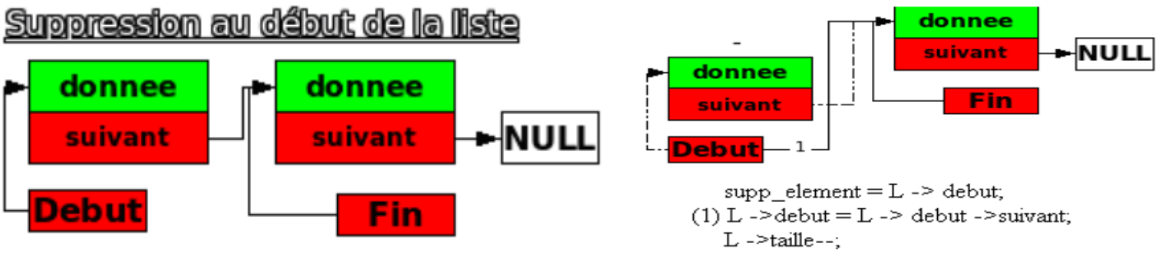
1. Retirer au début de la liste
2. Retirer à la fin de la liste
3. Retirer ailleurs dans la liste

Voici l'algorithme général pour retirer un élément de la liste :

1. Utilisation d'un pointeur temporaire pour sauvegarder l'adresse d'éléments à retirer;
2. L'élément à retirer se trouve après l'élément courant;
3. Faire pointer le pointeur suivant de l'élément courant vers l'adresse du pointeur suivant de l'élément à retirer;

4. Libérer la mémoire occupée par l'élément à retirer;
5. Mettre à jour la taille de la liste.

i. *Retirer au début de la liste*



Prototype de la fonction (liste.h): `Information suppDebutListe (Liste *L) ;`

Définition de la fonction (liste.c):

```
Information suppDebutListe (Liste *L)
{
	Information res;
	Maillon *supp;
	if(L->taille == 0)
		printf(" impossible de supprimer, liste vide !\n");
	else{
		supp = L->debut;
		L->debut = L->debut->suivant;
		L->taille--;
		if (L->taille == 0)
			L->fin = NULL;
		res = supp->donnee;
		free(supp);
	}
	return res;
}
```

ii. *Retirer à la fin de la liste*

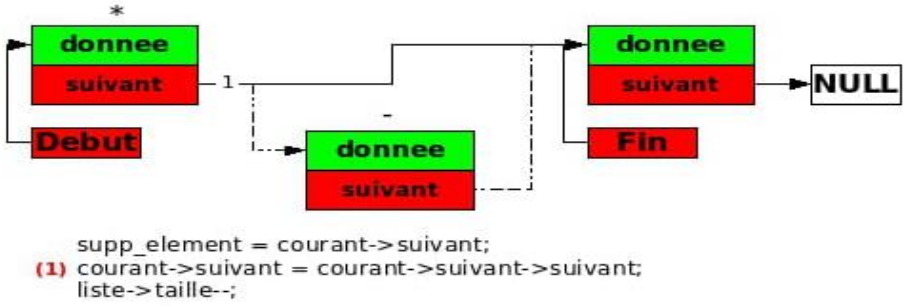
Prototype de la fonction (liste.h): `Information suppFinListe (Liste *L);`

Définition de la fonction (liste.c):

```
Information suppFinListe (Liste *L)
{
	Information res;
	Maillon *pred, *courant;
	if (L->taille == 0)
		printf(" impossible de supprimer dans liste vide\n");
	else {
		courant = L->debut;
		pred = NULL;
		// On se deplace vers le derenier element
		while ( courant->suivant != NULL)
		{
			pred = courant ;
			courant = courant->suivant;
		}
		res = courant->donnee;
		free(courant);
		L->fin = pred;
```

```
            if (pred != NULL) // taille > 1
                  pred->suivant = NULL;
            else // la liste est de taille == 1
                  L->debut = NULL;
            L->taille--;
      }
      return res;
}
```

iii. *Retirer ailleurs dans la liste (après une position)*



Prototype de la fonction (liste.h):

```
int suppApresPositionliste (Liste *liste, int pos);
```

Définition de la fonction (liste.c):

```
int suppApresPositionliste (Liste *liste, int pos){
{
      int i;
      Information res;
      Maillon *courant, *supp;
      if (L->taille <= 1 || pos < 1 || pos >= L->taille)
        printf("Impossible de supprimer à cette position\n");
      else{
            courant = L->debut;
            for (i = 1; i < pos; i++){
                  courant = courant->suivant;
            }
            supp = courant->suivant;
            courant->suivant = courant->suivant->suivant;
            if(courant->suivant == NULL)
                  L->fin = courant;
            res = supp->donnee;
            free (supp);
            L->taille--;
      }
      return res;
}
```

d) Affichage et destruction d'une liste

i. *Affichage de la liste*

Pour afficher la liste entière il faut se positionner au début de la liste. Ensuite en utilisant le pointeur suivant de chaque élément, la liste est parcourue du premier vers le dernier élément. La condition d'arrêt est donnée par le pointeur suivant du dernier élément qui vaut NULL ou la taille de la liste.

Prototype de la fonction (liste.h) : void afficherListe (Liste * L);

Définition de la fonction (liste.c) :

```
void afficherListe (Liste *L)
{
      Maillon *courant;
      courant = L->debut;
      while (courant != NULL)
      {
         printf ("format des champs…\n", (courant->donnee).champ1,…);
         courant = courant->suivant;
      }
}
```

ii. Destruction de la liste

Pour détruire la liste entière et libérer l'espace occupé par les maillons, on utilise par exemple la suppression au début de la liste tant que la taille est plus grande que zéro.

Prototype de la fonction (liste.h) : void detruireListe (Liste * L);

Définition de la fonction (liste.c) :

```
void detruireListe (Liste *L)
{
      while (L->taille > 0)
            suppDebutListe(L);
      free(L);
}
```

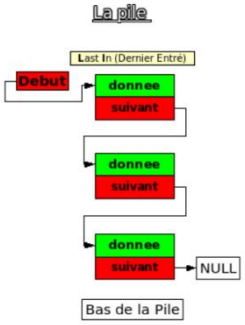
Chapitre 4 : Les Piles et les Files

1) Les Piles

a) Généralité

Une pile est une structure de données dynamique caractérisée par un comportement particulier en ce qui concerne l'insertion et l'extraction des éléments (des éléments y sont introduits, puis extraits) ayant la propriété que, lors d'une extraction, l'élément extrait est celui qui a y été introduit le plus récemment. On dit « dernier entré, premier sorti », **LIFO** « Last In First Out »). Ces caractéristiques sont :

- **Insertion en tête : empiler => Push** ;
- **Suppression en tête : dépiler => Pop ;**
- On repère la pile par son pointeur de début.
- Une pile est un ensemble d'éléments de même type, seul l'élément au sommet est visible.



b) Construction d'une pile

Comme les piles sont basées sur les listes simplement chaînées, alors pour définir un élément de la pile, on procède comme pour une liste simplement chaînée :

Donc, la première chose à faire c'est de définir le type cette information :

```
struct Info
{
      type_1 champs_1 ;
      type_2 champs_2 ;
      …
      type_n champs_n ;
};
typedef struct Info Information;
```

Après, on passe à la définition de l'élément «maillon » :

```
struct maillon
{
      Information donnee;
      struct maillon *suivant;
};
typedef struct maillon Maillon;
```

Enfin, on peut définir notre pile :

```
struct pile
```

```
{
      Maillon *debut;
      int taille;
};
typedef struct pile Pile;
```

c) Operations sur les piles

i. Initialisation

Cette opération doit être faite avant toute autre opération sur la pile. Elle initialise le pointeur ***debut*** avec le pointeur **NULL**, et la ***taille*** avec la valeur **zero**.

Prototype de la fonction : `Pile* CreerPile();`

Définition de la fonction :

```
Pile * CreerPile() {
      Pile *P;
      P = (Pile*)malloc(sizeof(Pile)) ;
      if( P == NULL ) {
            printf("Allocation non reussie\n");
            exit(-1);
      } else {
             P->debut = NULL;
             P->taille = 0;
      }
      return P;
}
```

ii. Insertion d'un élément dans la pile (push)

Voici l'algorithme d'insertion et de sauvegarde des éléments :

- Déclaration de l''élément à insérer ;
- Allocation de la mémoire pour le nouvel élément ;
- Remplir le contenu du champ de **donnee** ;
- Mettre à jour le pointeur **debut** vers le 1er élément (le sommet de la pile) ;
- Mettre à jour la **taille** de la pile.

Prototype de la fonction : `void empiler (Pile *P, Information val);`

Définition de la fonction :

```
void empiler(Pile *P, Information val) {
      Maillon *m;
      M = CreerElement();
      m->donnee = val;
      m->suivant = P->debut;
      P->debut = m;
      P->taille++;
}
```

iii. Retirer un élément de la pile (pop)

Pour dépiler l'élément de la pile, il faut tout simplement retirer l'élément vers lequel pointe le pointeur **debut** (sommet de la pile).

Prototype de la fonction : `Information depiler(Pile *P);`

Les étapes de l'algorithme sont :

- Le pointeur **supp_elem** contiendra l'adresse du 1er élément à supprimer;
- Le pointeur **debut** pointera vers le 2ème élément (après la suppression du 1er élément, le 2ème sera en haut de la pile);
- La taille de la pile sera décrémentée d'un élément.

Définition de la fonction :

```
Information depiler(Pile *P) {
   Maillon *supp_element;
   Information res;
   if (P->taille == 0)
         printf("impossible de dépiler une pile vide");
   else{
         supp_element = P->debut;
         P->debut = P->debut->suivant;
         res = supp_element->donnee;
         free (supp_element);
         P->taille--;
   }
   return res;
}
```

iv. Affichage de la pile

Pour afficher la pile entière, il faut se positionner au début de la pile. Ensuite, en utilisant le pointeur suivant de chaque élément, la pile est parcourue du premier vers le dernier élément. La condition d'arrêt est donnée par la taille de la pile.

Prototype de la fonction : `void affichPile (Pile *P);`

Définition de la fonction :

```
void affichePile(Pile *P) {
      Maillon *courant;
      int i;
      if(P->taille == 0)
            printf(« Pile vide! \n« );
      else{
            courant = P->debut;
            for(i = 0; i < P->taille; i++){
printf("%format des champs \n",(courant ->donnee).champ1, …);
            courant = courant->suivant;
            }
      }
}
```

v. Récupération de la donnée en haut de la pile

Pour récupérer la donnée au sommet la pile sans la supprimer, on peut utiliser une macro définie par comme suit:

```
#define top(P) P->debut->donnee
```

vi. *Destruction de la pile*

Pour détruire la pile entière, on dépile l'élément au sommet tant que la taille est non nulle.

Prototpe de la fonction : `void detruirePile (Pile *P);`

Définition de la fonction :

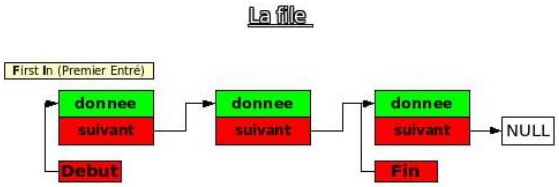
```
void detruirePile (Pile *P){
      while (P->taille > 0)
            depiler (P);
      free(P);
}
```

2) Les Files

a) Généralité

Une file est en générale une structure de liste appelée (FIFO) (First In First Out) «premier entré, premier sorti », (premier arrivé, premier servi). Ces caractéristiques sont :

- **Insertion en queue :** ajouter => enfiler ;
- **Suppression en tête :** supprimer => défiler
- On opère la file par son pointeur de début et son pointeur de fin.
- Dans une file : Les insertions (enfilements) se font à une extrémité appelée queue de la file et les suppressions (défilements) se font à l'autre extrémité appelée tête



b) Construction d'une file

Comme les piles, les files sont aussi basées sur les listes simplement chaînées, alors pour définir un élément de la file, on procède comme pour une liste simplement chaînée :

Donc, la première chose à faire c'est de définir le type cette information :

```
struct Info
{
      type_1 champs_1 ;
      type_2 champs_2 ;
      …
      type_n champs_n ;
};
typedef struct Info Information;
```

Après, on passe à la définition de l'élément «maillon » :

```
struct maillon
{
      Information donnee;
      struct maillon *suivant;
};
typedef struct maillon Maillon;
```

Enfin, on peut définir notre file :

```
struct file
{
	Maillon *debut;
	Maillon *fin;
	int taille;
};
typedef struct file File;
```

c) Operations sur les files

i. Initialisation

Cette opération doit être faite avant toute autre opération sur la file. Elle initialise les pointeurs ***debut et fin*** avec le pointeur **NULL**, et la ***taille*** avec la valeur **zero**.

Définition de la fonction :

```
File* CreerFile() {
	File *F;
	F = (File*)malloc(sizeof(File)) ;
	if( F == NULL ) {
		printf("Allocation non reussie\n");
		exit(-1);
	} else {
		 F->debut = NULL;
		 F->fin = NULL;
		 F->taille = 0;
	}
	return F;
}
```

ii. Insertion d'un élément dans la file (enfiler)

C'est la même algorithme ajouterFin pour les listes !

Définition de la fonction :

```
void enfiler(File *F, Information val) {
	Maillon *m;
	M = CreerElement();
	m->donnee = val;

	if(F -> taille == 0) { //file vide
		F->fin = m;
		F->debut = m;
	}else { // file non vide
		F->fin->suivant = m;
		F->fin = m;
	}
	F->taille++;

}
```

iii. Retirer un élément de la file (défiler)

Pour retirer un élément de la file, il faut tout simplement supprimer l'élément vers lequel pointe le pointeur **debut** (retirer au début comme pour une liste).

Définition de la fonction :

```
Information defiler(File *F) {
      Maillon *supp;
      Information res;
      if (F->taille == 0)
            printf("impossible de defiler une file vide");
      else{
            supp = F->debut;
            F->debut = F->debut->suivant;
            res = supp->donnee;
            free(supp);
            F->taille--;
            if(F->taille==0)
                  F->fin =NULL;
      }
      return res;
}
```

iv. Affichage de la file

Pour afficher la file entière, il faut se positionner au début de la file. Ensuite, en utilisant le pointeur suivant de chaque élément, la file est parcourue du premier vers le dernier élément. La condition d'arrêt est donnée par la taille de la file.

Définition de la fonction :

```
void afficheFile (File *F) {
      Maillon *courant;
      int i;
      if(F->taille == 0)
            printf("File vide! \n");
      else{
            courant = F->debut;
            for(i = 0; i < F->taille; i++){
printf("%format des champs \n",(courant->donnee).champ1, …);
            courant = courant->suivant;
            }
      }
}
```

v. Destruction de la file

Pour détruire la file entière, on défile l'élément au début tant que la taille est non nulle.

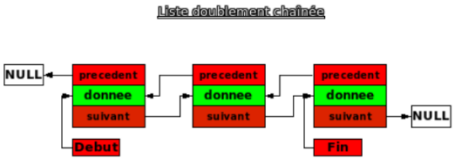
Définition de la fonction :

```
 void detruireFile (File *F){
      while (F->taille > 0)
            defiler (F);
      free(F);
```

```
}
```

Chapitre 5: Les listes doublement chaînées

Principe : Une liste à double chaînage est une liste dont chaque élément pointe à la fois sur son successeur et son prédécesseur. De cette manière, la notion de tête n'a pas d'importance car la liste peut être repérée par l'intermédiaire de n'importe lequel de ses éléments en faisant une recherche avant ou une recherche arrière (symétrique).



1) Création et déclaration d'une liste doublement chaînée

a) Déclaration

Avant de déclarer une liste, il faut créer les structure necesssaire pour sa construction :

Information, MaillonDc (éléments) et ListeDc

On procède de la même manière que pour les listes simplement chaînées, la seule différence est au niveau de la structure **Maillon** où on ajoute un pointeur **precedent.**

Donc, la première chose à faire c'est de définir le type cette information :

```
struct Info
{
      type_1 champs_1 ;
      type_2 champs_2 ;
      …
      type_n champs_n ;
};
typedef struct Info Information;
```

Après, on passe à la définition de l'élément «maillon » :

```
struct maillonDc
{
      Information donnee;
      struct maillon *suivant;
      struct maillon *precedent;
};
typedef struct maillon MaillonDc;
```

Enfin, on peut définir notre liste doublement chainée :

```
struct listeDc
{
```

```
        Maillon *debut;
        Maillon *fin;
        int taille;
};
typedef struct liste ListeDc ;
```

b) Création

Dans cette partie on crée les fonctions d'initialisation d'un élément (maillonDc) et d'une liste :

Les champs champs_1, champs_2, … et champs_n dépend de **Information**.

```
Maillon *CreerElementDc(){
        MaillonDc *m;
        m = (MaillonDc*)malloc(sizeof(MaillonDc)) ;
        if (m == NULL) {
                printf("Allocation non reussie\n");
                exit(-1);
        }
        else {
                m->donnee.champs_1=0;
                …
                m->donnee.champs_n = 0;
                m->suivant = NULL;
                m->precedent = NULL;
        }
return m;
}
```

Pour la liste c'est toujours cette fonction qu'on utitlise :

```
ListeDc* CreerListeDc(){
        ListeDc *L = (ListeDc*)malloc(sizeof(ListeDc)) ;
        if (L == NULL) {
                printf("Allocation non reussie\n");
                exit(-1);
        }
        else {
                L->debut = NULL;
                L->fin = NULL;
                L->taille = 0;
        }
return L;
}
```

Une fois terminé toutes ses opérations on peut procéder aux déclarations suivantes :

```
ListeDc *Ldc ; // déclaration d'un objet de type ListeDc*
Ldc = CreerListeDc(); // Création
…
// Déclaration et création d'un objet de type MaillonDc*
MaillonDc *element = CreerElementDc() ;
```

c) Accès aux champs d'une liste chainée

Pour pouvoir accéder aux différents champs d'une liste ou d'un élément d'une liste, il faut respecter les règles lies à la manipulation des pointeurs :

Puisque on va toujours déclarer un pointeur sur une liste, on utilisera l'opérateur -> **(flèche).**

Exemple :

```
ListeDc *Ldc;
Ldc->debut;
Ldc->fin;
Ldc->taille;
Ldc->debut->donnee;
Ldc->debut->suivant;
Ldc->debut->precedent;
(Ldc->debut->donnee).champs1;
(Ldc->debut->donnee).champs2;
… etc
```

d) A retenir, les points clés des listes doublement chaînées

- ✓ Chaque élément de la liste est formé de n+2 champs :
  - n champs constituant l'**information** portée par le maillon (l'élément), dépendante du problème particulier considéré ;
  - un champ supplémentaire qui est la matérialisation de la relation successeur (le pointeur **suivant**).
  - un autre champ supplémentaire qui est la matérialisation de la relation prédécesseur (le pointeur **precedent**).
- ✓ Le pointeur **début** contiendra l'adresse du premier élément de la liste ;
- ✓ Le pointeur **fin** contiendra l'adresse du dernier élément de la liste ;
- ✓ La variable entière **taille** contient le nombre d'éléments de la liste.
- ✓ Quel que soit la position dans la liste, les pointeurs **début** et **fin** pointent toujours respectivement vers le premier et le dernier élément.
- ✓ Le champ **taille** contiendra le nombre d'éléments de la liste quel que soit l'opération effectuée sur la liste.

2) Opérations sur les listes doublement chaînées

a) Insertion d'un élément dans la liste

Pour ajouter (insérer) un élément dans la liste il y a plusieurs situations :

1. Insertion au début de la liste ;
2. Insertion à la fin de la liste ;
3. Insertion avant un élément ;
4. Insertion après un élément.

i. Insertion au début de la liste

Prototype de la fonction (liste.h): `void insertDebutListeDc (ListeDc *L, Information val);`

Définition de la fonction (liste.c):

```
void insertDebutListeDc (ListeDc *L, Information val)
{
	MaillonDc *m;
	m = CreerElementDc();
	m->donnee = val;
	m->suivant = L->debut; // 1
	if(L->debut != NULL)
		L->debut->precedent = m ;  // 2
	L->debut = m; // 3
	// Si la liste était vide; le debut et la fin coïncident
	if (L->taille == 0)
		L->fin = m;
	L->taille++;
}
```

ii. Insertion à la fin de la liste

Prototype de la fonction (liste.h): `void insertFinListeDc (ListeDc *L, Information val);`

Définition de la fonction (liste.c):

```
void insertFinListeDc (ListeDc *L, Information val)
{
	MaillonDc *m;
	m = CreerElementDc();
	m->donnee = val;
	m->precedent = L->fin;
	// Si la fin contient déjà un élément
	// ajouter m comme son suivant
	if(L-> fin !=NULL)
		L->fin->suivant = m;
	L->fin = m;
	// si la liste était vide; le début et la fin coïncident
	if (L->taille == 0)
		L->debut = m;
	L->taille++;
}
```

iii. Insertion avant une position dans la liste

L'insertion s'effectuera avant une certaine position passée en argument à la fonction.

- ✓ On va parcourir la liste avec un pointeur courant jusqu'à atteindre la position.
- ✓ le pointeur **suivant** du **nouvel élément** pointe vers l'élément **courant**.
- ✓ le pointeur **precedent** du **nouvel élément** pointe vers l'adresse sur la quelle pointe le pointeur **precedent** d'élément **courant**.
- ✓ Le pointeur **suivant** de l'élément qui précède l'élément **courant** pointera vers le **nouvel élément**.
- ✓ Le pointeur **precedent** d'élément **courant** pointe vers le **nouvel élément**.
- ✓ Le pointeur **fin** ne change pas.

- ✓ Le pointeur **debut** change si et seulement si la position choisie est la position 1. (cette situation est aussi gérer par la fonction **insertDebutListeDc**)

Prototype de la fonction (liste.h):

```
int insertAvantPosListeDc (ListeDc *L, Information val, int pos);
```

La fonction renvoie -1 en cas d'échec sinon elle renvoie 0.

Définition de la fonction (liste.c):

```
void insertAvantPosListeDc (ListeDc *L, Information val, int pos) {
    // La position doit être dans [1, (L->taille)]
    if (pos < 1 || pos > L->taille)
          return -1;
    MaillonDc *m, *courant;
    M = CreerElementDc();
    m->donnee = val;
    courant = L->debut;
    int i; -
    for (i = 1; i < pos; i++)
          courant = courant->suivant;
    m->precedent = courant->precedent;
    m->suivant = courant ;
    if (courant->precedent == NULL)
          L->debut = m;
    else {
          courant->precedent->suivant = m;
    }
    courant->precedent = m;
    L->taille++;
    return 0;
}
```

iv. Insertion après une position dans la liste

L'insertion s'effectuera après une certaine position passée en argument à la fonction.

- ✓ On va parcourir la liste avec un pointeur **courant** jusqu'à atteindre la position.
- ✓ Le pointeur **precedent** du **nouvel** élément pointe vers l'élément **courant**.
- ✓ Le pointeur **suivant** du **nouvel** élément pointe vers l'adresse sur la quelle pointe le pointeur **suivant** d'élément **courant**.
- ✓ Le pointeur **precedent** de l'élément qui suit l'élément **courant** pointera vers le **nouvel** élément.
- ✓ Le pointeur **suivant** d'élément **courant** pointe vers le **nouvel** élément.
- ✓ Les pointeurs **debut** ne change pas;
- ✓ Le pointeur **fin** change si et seulement si la position choisie est la position du dernier élément (taille de la liste doublement chaînée). (cette situation est aussi gérer par la fonction **insertFinListeDc**)
- ✓

Prototype de la fonction (liste.h):

```
int insertApresPosListeDc (ListeDc *L, Information val, int pos);
```

La fonction renvoie -1 en cas d'échec sinon elle renvoie 0.

Définition de la fonction (liste.c):

```
  int insApresPositionliste (ListeDc *L, Information val, int pos){
      // La position doit être dans [1, (L->taille)]
      if (pos < 1 || pos > L->taille)
            return -1;
      MaillonDc *courant, *m;
      int i;
      m = CreerElementDc();
      m->donnee = val;
      // On se déplace vers la position pos
      courant = L->debut;
      for (i = 1; i < pos; i++) {
            courant = courant->suivant;
            }
      // Mise à jours
      m->precedent = courant; // 1
      m->suivant = courant->suivant; // 2
      if (courant->suivant == NULL) // 3
            L->fin = m;
      else {
            courant->suivant->precedent = m;
      }
      courant->suivant = m; // 4
      L->taille++;
return 0;
}
```

b) Retirer un élément dans la liste doublement chaînée

Voici l'algorithme de suppression d'un élément de la liste doublement chaînée :

- ✓ Utilisation d'un pointeur temporaire pour sauvegarder l'adresse de l'élément à supprimer. L'élément à supprimer peut se trouver dans n'importe quelle position dans la liste.
- ✓ Retourner le champ donnee de l'élément à supprimer.
- ✓ Libérer la mémoire occupée par l'élément supprimé;
- ✓ Mettre à jour la taille de la liste.

Pour supprimer un élément dans la liste il y a plusieurs situations :

1. Retirer au début de la liste doublement chainée;
2. Retirer à la fin de la liste doublement chaînée
3. Retirer à une position quelconque.

Nous allons créer une seule fonction. Si nous voulons supprimer l'élément au début de la liste nous choisirons la position **1**. Si nous voulons supprimer l'élément à la fin de la liste nous choisirons la position **L ->taille**. Si nous désirons supprimer un élément quelconque alors on choisit sa position dans la liste. (Vous pouvez essayer d'implémenter 3 les fonctions séparément)

Définition de la fonction:

```
Information suppPosListeDc(ListeDc *L, int pos){
      int i;
      Information res;
      MaillonDc *supp_element,*courant;
      if (L->taille == 0){
            printf (“impossible de supprimer Liste vide! \n ”);
            exit(-1) ;
            }

      if( pos == 1) { /* suppression au début /
            supp_element = L->debut;
            L->debut = L->debut->suivant;
            if (L->debut == NULL)
                  L->fin = NULL;
            else
                  L->debut->precedent == NULL;
      } else if (pos == L->taille) { /* suppression à la fin */
            supp_element = L->fin;
            L->fin = L->fin->precedent;
            if (L->fin == NULL)
                  L->debut = NULL;
            else
                  L->fin->suivant = NULL;
      } else{ /* suppression ailleurs */
            courant = L->debut;
            for(i = 1 ;i < pos; i++)
                  courant = courant->suivant;
            supp_element = courant;
            courant->precedent->suivant = courant->suivant;
            courant->suivant->precedent = courant->precedent;
      }
      res = supp_element->donnee;
      free(supp_element);
      L->taille--;
      return res;
}
```

c) Affichage de la liste doublement chaînée

Pour afficher la liste entière il existe deux possibilités, affichage direct ou affichage inverse. En utilisant le pointeur suivant ou précèdent de chaque élément, la liste est parcourue du premier vers le dernier élément (affichage direct) ou du dernier vers le premier élément (affichage inverse).

```
      void afficheDirectListeDc (ListeDc *L) {
            MaillonDc *courant;
            courant = L->debut;
            while(courant != NULL) {
                  printf ("format1… ", (courant->donnee).champ1,…);
                  courant = courant->suivant;
             }
```

```
}

void afficheInverseListeDc (ListeDc *L) {
      MaillonDc *courant;
      courant = L->fin;
      while(courant != NULL) {
            printf ("format1… ", (courant->donnee).champ1,…);
            courant = courant->precedent;
       }
}
```

d) Destruction de la liste

Pour détruire la liste entière, on utilise par exemple la suppression à la position 1 de la liste tant que la taille est plus grande que zéro.

```
void detruireListeDc (ListeDc * L) {
      while (L -> taille > 0)
            suppPosListeDc (L, 1);
      free(L);
}
```

</fin>

Bibliography

Prof. Brahim Aksasse, Diaporama_Cours_SDC_18_19.pdf.

Lien de cours

Commentcamache.net

Zeek Zone