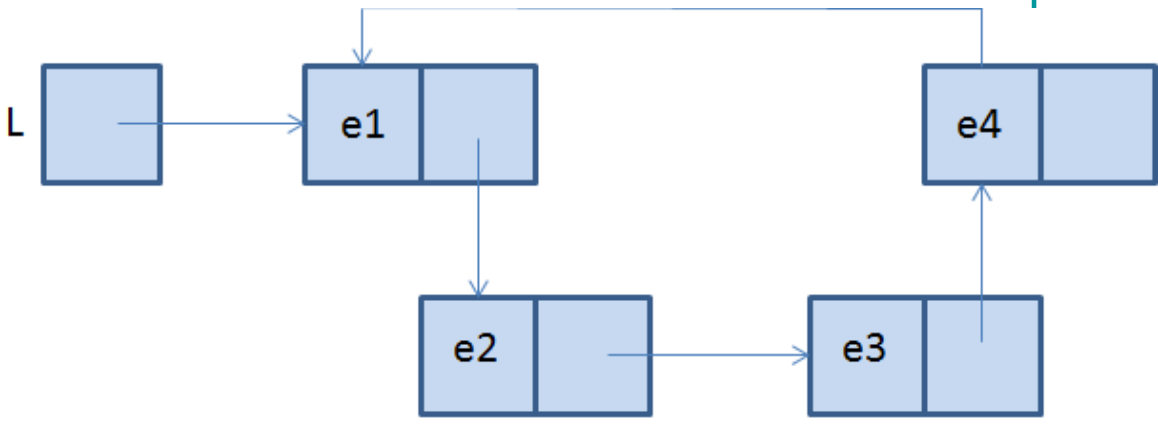
RAPPORT

CRÉATION D'UNE APPLICATION DES LISTES CHAINÉES SUR LES NOMBRES COMPLEXES



Rélisé par;

- Amal Aderdor
- Laila hamza
- Maria Ejjbiri
- Asma Asaad
- Asma Elfahim

Encadré par:

- Mr Youssef Es-aady

Sommaire :

INTRODUCTION

Au cours de notre formation en Licence d'Education en Informatique, nous sommes amenés à maîtriser la programmation en langage C.

Pour ce module **-Structures des données-**, enseigné par **Monsieur YOUSSEF ES-SAADY**, on est censés développer nos prérequis en programmation, on a donc commencé par la manipulation des listes chaînées.

Le problème est de construire une liste chaînée pour manipuler des nombres complexes. Nous allons programmer notre code en utilisant le **langage C.**

La première étape était d'écrire les structures dont on aura besoin:

- Liste
- Maillon
- Complexe

. Ensuite, on a réparti la problématique en fonctions pour faciliter la rédaction du code.

Finalement, après de nombreuses analyses, nous avons abouti à un code, qui sera détaillé et expliqué dans le chapitre suivant.

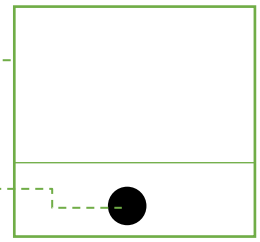
Chapitre 1 : La création des structures :

1. Structure d'un nombre complexe :

Pour les nombres complexes, on va créer une structure qu'on va définir en tant qu'un type Tcomplexe et qui contiendra deux parties, une partie réelle et une partie imaginaire.

```
typedef struct complexe {
    float Reel ;
    float Im ;
}Tcomplexe;
```

2. Structure d'un maillon :



Un maillon est un élément divisé en deux parties :

- Une valeur
- Un pointeur

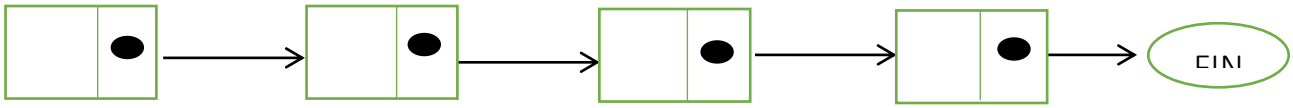
Dans cet exercice, la valeur du maillon est un nombre complexe.

```
struct maillon
{
    Tcomplexe valeur; //valeur
    struct maillon *suivant;
};
typedef struct maillon *Tmaillon;
```

3. Structure d'une liste :

La liste chainée est une structure qui permet de stocker une suite d'éléments de même type. Elle possède les maillons suivants :

- ✓ Tête de la liste
- ✓ Queue de la liste



Sa structure est :

```
struct liste
    {
        int nmbrElements;
        Tmaillon *pTete ;
        Tmaillon *pQueue ;

    };
typedef struct liste TListe;
```

Chapitre 2 : Les fonctions

1. Création d'une liste vide :

Comme on a défini un nouveau type **TListe** qui contient une tête et une queue, il suffit de créer un nouveau pointeur **pL** de ce type (**TListe**), lui allouer un espace dans la mémoire et préciser que le contenu des maillons (tête et queue) est nul. Puis retourner la valeur du pointeur pour obtenir une liste vide.

```
11    TListe *CreerListe()
12    {
13        TListe *pL;
14        pL= malloc(sizeof(TListe));
15        pL->pTete = NULL;
16        pL->pQueue = NULL;
17        return pL;
18    }
```

2. Création d'un nombre complexe :

Cette fonction aura comme paramètres deux variables **(x, y)** de type double. Première chose à faire est de créer un pointeur **Z** de type **Tcomplexe**, dont sa partie **réelle** sera la valeur de **'x'** et la partie **imaginaire** celle de **'y'**.

```
21    Tcomplexe CreeComplexe(double x , double y)
22    {
23        Tcomplexe *Z;
24        Z=malloc(sizeof(Tcomplexe));
25        Z->Reel=x;
26        Z->Im=y;
27        return *Z;
28    }
```

3. Création d'un maillon :

C'est une fonction qui aura comme paramètre un nombre complexe.

Pareil pour la création d'un nouveau maillon dans la liste, il suffit de créer une variable **m** de type **Tmaillon** *(puisque Tmaillon est définit comme un pointeur)* puis affecter à cette dernière une valeur de type **Tcomplexe** pour la partie **valeur**. Ce maillon ne pointera sur aucun autre maillon.

```
31    Tmaillon CreerMaillon(Tcomplexe val )
32    {
33        Tmaillon m;
34        m = (Tmaillon)malloc(sizeof(Tmaillon));
35        m -> valeur=val;
36        m -> suivant =NULL;
37        return m;
38    }
```

4. Vérifier si la liste est vide :

Une liste est dite vide si le contenu de la tête est nul. Pour ce faire, on utilisera la fonction suivante :

```
40  int ListeVide(TListe*L)
41      {
42          return (L->pTete==NULL);
43      }
```

5. Longueur de la liste :

On crée un maillon **pL**, on lui affecte la valeur de la tête (*ptête*) pour parcourir la liste à partir de son premier élément jusqu'au dernier. Soit **longueur** un indice qui a comme valeur initiale **0**. A chaque fois que **pL** prend la valeur de son successeur on y ajoute **1** à l'aide de la boucle **while** qui a comme condition d'arrêt un **suivant égal à NULL**. Puis la fonction retourne la valeur de **longueur.**

```
45  int LongListe (TListe *L)
46      {
47      int longueur =0;
48      Tmaillon pL;
49      pL = L->pTete ;
50      if(ListeVide(L))
51      return longueur;
52      else
53          {
54              while(pL != NULL)
55                  {
56                      longueur++;
57                      pL = pL->suivant;
58                  }
59          }
60      return longueur;
61      }
```

6. Affichage de la liste :

On doit afficher le contenu, tant qu'il existe. Pour ce faire, on a utilisé la boucle while où on affichera la partie réelle et la partie imaginaire du nombre.

```
64  void afficher_liste(TListe *L)
65      {
66          if (ListeVide(L))
67              {
68                  printf("Rien a afficher, la liste est vide\n");
69                  return ;
70              }
71          Tmaillon pt = L->pTete;
72          while(pt != NULL)
73              {
74                  Tcomplexe c = pt->valeur;
75                  printf("%.2f +i %.2f\n",c.Reel,c.Im);
76                  pt = pt->suivant;
77              }
78          printf("\n");
79      }
```

7. Insertion de la tête de la liste :

C'est une fonction qui a comme paramètre une liste et un nombre complexe.

On commence par créer un maillon en utilisant la fonction **CreerMaillon ().** Ensuite, on réalise une liaison entre le maillon crée et la tête de la liste. La fonction retourne **1** si la tête a été insérée, sinon, elle retourne **0**.

```
82  int InsererTete(TListe *L,Tcomplexe val )
83      {
84              Tmaillon tmp;
85              tmp= CreerMaillon(val);
86              if (tmp==NULL) return 0;
87              tmp->suivant= L->pTete;
88              L->pTete=tmp;
89              return 1;
90      }
```

8. Insertion d'un maillon au milieu de la liste :

Dans un premier temps, on alloue l'espace nécessaire au stockage du nouveau maillon et on y place le nouveau nombre val. Puis on demande à l'utilisateur d'entrer l'indice ou ils veulent saisir ce maillon. Il reste alors une étape délicate : l'insertion du nouvel élément dans la liste chaînée.

On prend un paramètre supplémentaire (pred) : l'adresse de celui qui précèdera notre nouvel élément dans la liste. On va parcourir la liste chaînée jusqu' à tomber sur l'élément indiqué (pred) :

➔ Alors on va utiliser la boucle while.

Valeur initiale	N=1 Pred=L->ptete
Instruction à répéter	Tant que (N ! =position) Pred=pred->suivant N++
Condition d'arrêt	N=position

Succ=pred->suivant : On prend un paramètre supplémentaire (succ) : l'adresse de notre nouvel élément précèdera succ dans la liste.

```
91  Tmaillon InsererMaillon(TListe *L, Tcomplexe elementAjouter)
92      {
93      Tmaillon pred;
94      Tmaillon succ;
95      Tmaillon nvM;
96      int position;
97      printf("Ou voulez-vous saisir ce nombre dans la liste?\nREMARQUE: La position suggeree doit etre comprise entre 1 et %d\n",LongListe(L));
98      scanf("%d",&position);
99      nvM = malloc(sizeof(Tmaillon));
00      nvM= CreerMaillon(elementAjouter);
01      if(InsererTete(L,elementAjouter)== NULL)
02          printf("Veuillez saisir un nombre en tete (choix: 1)");
03      else
04          if(position <= LongListe(L) )
05          {
06              int i=1;
07              pred=L->pTete;
08              while (i!=position)
09              {
10                  pred=pred->suivant;
11                  i++;
12              }
13              succ=pred->suivant;
14              succ->suivant=nvM;
15              nvM->suivant=succ;
16          }
17              else printf("Erreur d'insertion!!\n");
18              return nvM;
19      }
```

9. Supprimer la tête de la liste :

Si le contenu de la tête n'est pas nul, je déplace le contenu de la tête vers un nouveau maillon. Ensuite, la nouvelle tête de la liste devient le maillon crée et je supprime en utilisant **free ()**.

```
130   void SupprTete(TListe* L)
131       {
132           if (L->pTete!=NULL)
133               {
134                   Tmaillon P=L->pTete;
135                   L->pTete=P->suivant;
136                   free(P);
137               }
138       }
```

10. Insérer la queue de la liste :

Dans un premier temps, on alloue l'espace nécessaire au stockage du nouveau maillon et on y place le nouveau nombre val.

Il reste alors l'insertion du nouvel élément à la fin de la liste chaînée.

On va parcourir la liste chaînée jusqu' à tomber sur le dernier élément.

Valeur initiale	P= L->ptete
Instruction à répéter	P=p->suivant
Condition d'arrêt	P-> suivant

Alors on va utiliser la boucle while.

D'où l'adresse du dernier élément (p) qui précèdera notre nouvel élément dans la liste : p->suivant=nouveau

```
87    void AjouterFin(TListe* L, Tcomplexe val)
88    {
89        Tmaillon nouveau=malloc(sizeof(Tmaillon));
90        if(nouveau==NULL) return ;
91        nouveau=CreerMaillon(val);
92        Tmaillon p = L->pTete;
93            while(p->suivant!=NULL)
94            p = p->suivant;
95            p->suivant=nouveau;
96            L->nmbrElements+=1;
97    }
```

Conclusion :

On a organisé notre programme comme une application de gestion de nombres complexes. Pour ce faire, le projet a été divisé en trois fichiers, ‘liste.h’ ; ‘liste. c’et main.c. Vous trouverez le code source dans le fichier zip NbCpx.

On voit que ce TP est particulièrement riche ainsi qu’il nous a permis de mieux comprendre la notion des pointeurs et maîtriser la manipulation des listes chainées.

Cependant, malgré les nombreuses fonctions que nous avons créées, nous ne pensons pas avoir épuisé la question, et nous espérons sincèrement susciter d'autres fonctions pour mieux maîtriser les listes chaînées.