Chapitre 5 : Allocation dynamique de mémoire

5.1 Introduction

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

Déclaration statique de données

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

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

Allocation dynamique

Problème

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

Exemple1 : int *T ; char * Mot ; …

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

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

5.2 La fonction malloc

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

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

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

5.3 L'opérateur sizeof

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

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

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

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

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

5.4 La commande exit

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

Exemple

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

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

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

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

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

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

5.5 La fonction free

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

free(<Pointeur>)

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

Exemple : Soit le programme suivant :

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

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

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

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

5.6 Les Listes chaînées

5.6.1/ Définitions

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

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

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

5.6.2/ Listes simplement chaînées

a) <u>Déclaration</u>

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

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

/* ou bien * /

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

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

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

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

b) Accès à une valeur d'une liste

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

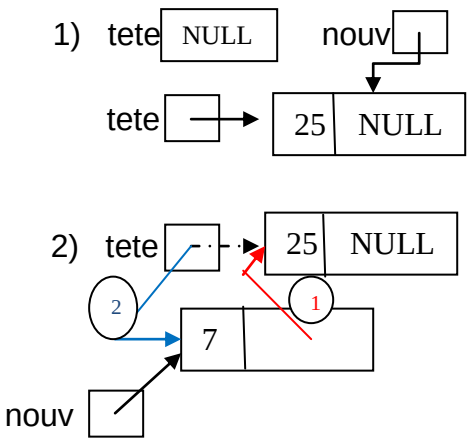
mais généralement on utilise la notation pointée →
typelem a=L→element ;

c) Création d'un nœud

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

d) Ajout d'un élément en tête de liste

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



e) Ajout d'un élément après une adresse donnée

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

f) Suppression d'un élément en tête de liste

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

g) Suppression de l'élément après une adresse donnée

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

h) Création d'une liste

- **Création FIFO :**

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

- **Création LIFO :**

/* créer une liste de n valeurs entières */

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

i) Parcours d'une liste

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

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

Ou bien :

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

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

j) Recherche d'une valeur donnée

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

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

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

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

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

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

k) Mise à jour d'une liste

- **Modification :**

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

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

Ou bien :

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

- **Insertion :**

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

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

- **Suppression :**

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

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

5.6.3/ Listes bidirectionnelles

a) Déclaration

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

b) Création d'un nœud

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

c) Ajouter un élément en tete de liste

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

d) Ajouter un élément après une adresse donnée

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

e) Création d'une liste

- **Création FIFO :**

```
typedef int  typelem ;

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

- **Création LIFO :**

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

f) Mise à jour d'une liste

- **Insertion :**

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

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

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

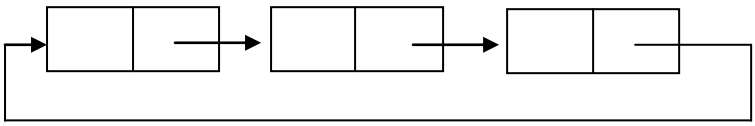
- **Suppression :**

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

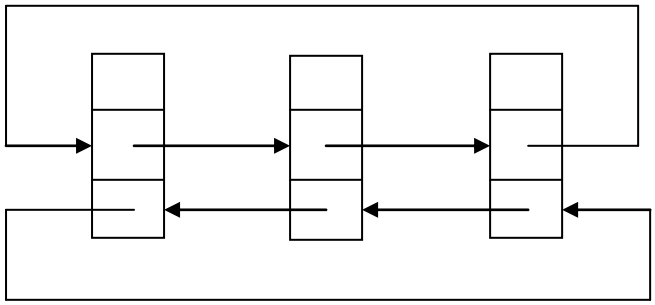
```
ListeB rechercheB(ListeB tete, typelem val)
{  ListeB p ;
   while(p !=NULL && p→element !=val) p=p→suivant;
  return p;
}

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

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

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

5.5.3/ Listes circulaires

a) Listes circulaires simplement chaînées



b) Listes circulaires bidirectionnelle



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

Solution :

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

Ou bien :

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

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