Université Abdelmalek ESSAADI
Faculté des Sciences
- Tétouan -

Filières : **SMI** – **S4**

Structures de données

Résumé du cours

Adnan SOURI

Créé le : 5 Ramadan 1431
Révisé le : 13 Ramadan 1442

Plan du cours

1. Structures de données

En Programmation, il est habituel d'avoir besoin d'un mécanisme permettant de grouper un certain nombre de variables de types différents au sein d'une même entité. La réponse à ce besoin est le concept de structure ou d'enregistrement : une structure ou enregistrement est un ensemble d'éléments de types différents repérés par un nom. On peut la modéliser comme une variable englobant plusieurs autres variables de la manière suivante :

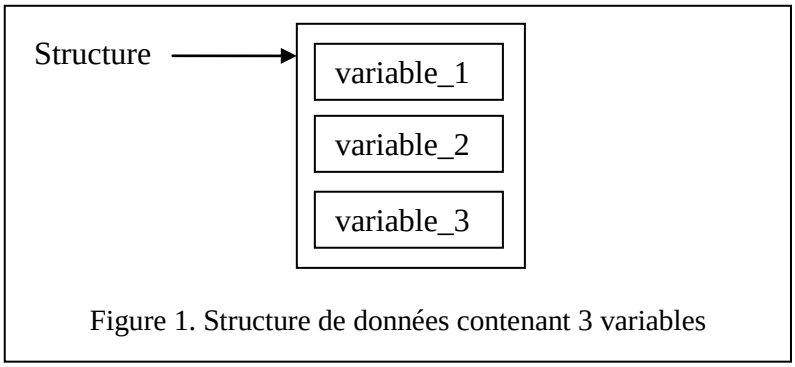


Figure 1. Structure de données contenant 3 variables

Les structures organisent les données composites, en particulier dans des programmes gourmands, en permettant de traiter un groupe de variables, de types différents, liées entre elles, comme un tout et non comme des entités séparées.

Le principal apport des structures est la possibilité de définir l'affectation de ces structures ; on peut copier les structures, les affecter, les passer en arguments à des fonctions et les faire retourner par des fonctions. On peut désormais les initialiser.

1.1 Définition

Une structure est une variable composée qui rassemble une ou plusieurs variables, qui peuvent être de types différents, que l'on regroupe sous un seul nom pour les manipuler facilement. Les structures s'appellent aussi enregistrements. Chaque élément de la structure, appelé membre ou champ, est désigné par un identificateur.

On distingue la déclaration d'un modèle de structure de celle d'une variable de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est « modele » suit la syntaxe suivante :

```
struct modele{
            type_1 champ_1;
            type_2 champ_2;
            ...
            type_n champ_n;
};
```

N.B : Cette déclaration n'entraîne aucune réservation mémoire.

On vient juste de créer un modèle de variables. Pour déclarer une variable de type structure correspondant au modèle précédent, on utilise la syntaxe :

```
struct modele  variable;
```

Cette instruction permet de réserver un emplacement nommé « `variable` » de type « **`struct`** `modele` » destiné à contenir tous les champs de la structure.

De manière semblable on peut déclarer plusieurs variables (trois dans l'exemple suivant) de type « **struct** modele » :

```
struct modele  var1, var2, var3;
```

N.B : On doit remarquer qu'il faut écrire « **struct** modele » et non pas « modele ».

Ceci étant dit, on déduit que le mot clé **struct**, en lui-même, n'est pas un type proprement dit, mais plutôt il permet de créer de nouveaux types adaptés selon la nature du problème à résoudre.

Exemple. Prenons l'exemple d'une structure qui représente un étudiant. Les membres de cette structure seront le nom, le prénom et la Note. La définition de cette structure serait :

```
struct etudiant{
            char nom[10];
            char prenom[10];
            int  Note ;
};
```

La déclaration suivante permet de déclarer deux variables `e1` et `e2` de type « **struct** etudiant » dont chacune est constituée des trois champs nom, prénom et note.

```
struct etudiant        e1, e2;
```

Remarques.

1- On aurait pu déclarer des variables de type structure lors de la définition du modèle de la structure. Cette déclaration sera plus compacte et aura la forme suivante :

```
struct etudiant {
            char nom[10];
            char prenom[10];
            int  Note ;
}e1, e2, e3;
```

2- De même, on peut utiliser le mot clé **typedef** lors de la définition de la structure des données :

```
typedef struct {
            char nom[10];
            char prenom[10];
            int  Note ;
}etudiant;
```

Ainsi, la déclaration ultérieure des variables de type « etudiant » peut se faire de la manière suivante :

```
etudiant X;
etudiant e,f,g ;
```

Cette fois-ci, `X, e, f` et `g` sont des variables de type `etudiant` et dont chacune contient les trois champs nom, prénom et note.

3- Dans la suite du cours, on utilisera la définition citée au paragraphe (*1.1 Définition*) pour définir une structure de données quelconque, car elle permet de bien séparer la définition du type structure de la déclaration des variables et leurs utilisations.

1.2 Manipulation globale des structures

Contrairement aux tableaux, une structure, comme une expression d'un type primitif, bénéficie de la sémantique des valeurs. On peut affecter une expression d'un type structure à une variable de type structure (pourvu que ces deux types soient identiques).

Exemple. Soit la définition de la structure suivante :

```
struct etudiant{
            char nom[10];
            char prenom[10];
            int  Note ;
};
```

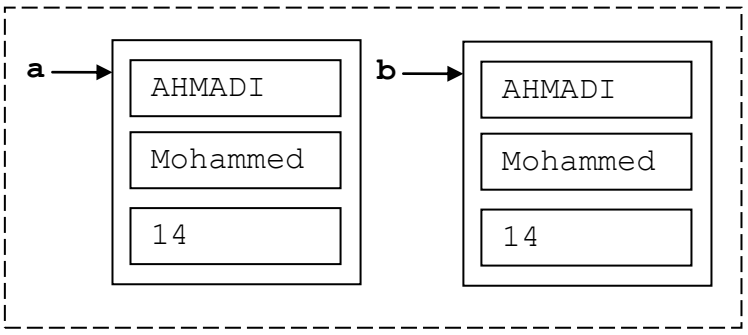
Une variable de type structure peut être initialisée au moment de sa déclaration, du moins dans le cas d'une variable globale. La syntaxe est la même que pour un tableau :

```
struct etudiant        a = {"AHMADI","Mohammed",14};
```

Une instruction d'affectation, par exemple, est permise, et elle implique que les champs de la variable **b** ont <u>les mêmes valeurs</u> que les champs de la variable **a.**

```
struct etudiant b;

b = a ;
```



<u>N.B</u> : Aucune comparaison n'est possible sur les structures, même avec les opérateurs == et !=.

1.3 Manipulation des structures par champ

La manipulation d'une structure nécessite aussi de manipuler ses membres (champs). Le besoin de l'accès aux champs d'une structure un par un est motivé par le fait que, à un instant donné, on voudrait effectuer des opérations, non pas sur la structure elle-même, mais uniquement sur quelques champs de cette dernière.

Pour désigner un champ d'une structure, on utilise l'opérateur « . » (point) qui est l'opérateur de sélection de champ. On note « **a.nom** », pour désigner le champ « nom » de la variable « a ». a étant la structure étudiant définie au paragraphe précédent.

Par exemple, si a et b sont deux variables de type struct étudiant, on accédera au champ nom de a par a.nom et on accédera au champ note de b par b.note. Les champs ainsi désignés se comportent comme n'importe quelle variable.

On peut effectuer sur le champ i de la structure toutes les opérations valides sur des données de type i. Par exemple le champ prénom de la structure a supporte toutes les opérations permises sur les chaînes de caractères. Pour le champ note, on peut effectuer n'importe quelle opération arithmétique ou logique.

Exemple. Le programme suivant permet de définir une structure de type complexe. On déclare deux complexes `z1` et `z2`, on calcule et on affiche leur somme. Ceci étant dit, on doit calculer la somme des parties réelles et imaginaires (manipulation des champs).

```
#include <stdio.h>
main(){
   struct nbComplexe{
           float reel;
           float imaginaire;
   };
struct nbComplexe z1={3,5},z2={2,4}; // z1 vaut 3+5i et z2
vaut 2+4i
struct nbComplexe z3;
z3.reel = z1.reel + z2.reel;
z3.imaginaire = z1.imaginaire + z2.imaginaire;

printf("la somme est %d+%di", z3.reel, z3.imaginaire);
}
```

1.4 Structures et fonctions

Comme on l'a déjà cité dans le paragraphe (*1.2.1 Manipulation globale*), une structure bénéficie de la sémantique des valeurs, ce qui lui donne la possibilité d'être passée comme argument des fonctions. Le résultat d'une fonction peut être une structure.

Exemple. Le programme suivant permet de définir une structure de type complexe. La fonction « somme » permet de calculer la somme de deux complexes passés par valeur.

```
#include <stdio.h>

struct nbComplexe{
           float reel;
           float imaginaire;
           };
main(){
   struct nbComplexe somme(struct nbComplexe,struct
nbComplexe);
   struct nbComplexe z1={3,5},z2={2,4},z;
   z = somme(z1,z2);
   printf("la somme est %d+%di", z.reel, z.imaginaire);
}

//========================== définition de la fonction

struct nbComplexe somme(struct nbComplexe z1, struct
nbComplexe z2){
   struct nbComplexe z3;
   z3.reel = z1.reel + z2.reel;
   z3.imaginaire = z1.imaginaire + z2.imaginaire;
   return(z3);
}
```

N.B : A l'intérieur d'une fonction, la portée est limitée à la fonction. La variable `z3` est une variable locale et sa valeur est retournée comme résultat est copiée dans la variable `z`

1.5 Tableaux de structures

A partir d'une structure définie, on peut déclarer un tableau de structures. Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple. Supposons que l'on ait déjà déclaré la structure **struct** etudiant, si on veut déclarer un tableau de 100 structures de ce type, on écrira :

```
struct etudiant        T[100];
```

Pour accéder aux champs de la structure, on précise l'indice de cette structure. Par exemple, pour accéder à la note de l'étudiant d'indice i, on écrit : **T[i].note = 13 ;**

1.6 Structures et pointeurs

c Un pointeur est une variable dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

```
type   *nom_du_pointeur;
```

Où **type** est le type de la variable pointée. Cette instruction déclare un identificateur, nom_du_pointeur, associé à une variable dont la valeur est l'adresse d'une autre variable de type **type**. L'identificateur nom_du_pointeur est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle variable, sa valeur est modifiable.

De même pour les structures, on déclare un pointeur p vers une structure de type **struct** modele :

```
struct modele      *p ;
```

On pourra alors affecter à p des adresses de **struct** modele. On prendra l'exemple de la structure étudiant dans le code suivant :

```
main( ){
...
struct etudiant e;  // e est une variable de type struct
etudiant
struct etudiant *p; // p est un pointeur vers une structure
etudiant

p = &e;
...
}
```

p est un pointeur vers la structure e. *p désigne la structure e elle-même. Pour accéder aux champs de cette structure, on peut utiliser la variable (pointeur) p. **(*p).nom** et **(*p).note** sont équivalents successivement à e.nom et e.note.

L'accès aux champs peut se faire aussi avec l'opérateur -> (le signe moins suivi de l'opérateur supérieur). (*p).champ est équivalent à p->champ.

1.7 Imbrication de structures

Une structure peut avoir parmi ses champs d'autres structures. On parle de l'imbrication des structures. Soit la structure « date » définie comme suit :

```
struct date{
            int jour;
            int mois;
            int annee;
};
```

Etant donnée la structure « étudiant » déjà définie, on veut préciser la date de naissance pour un étudiant donné. La structure « étudiant » va contenir parmi ces champs un champ de type « date » pour désigner la date de naissance de l'étudiant.

```
struct etudiant{
            char        nom[10];
            char        prenom[10];
            int         Note ;
            struct date D ;
};
```

La date, ici, est une structure à l'intérieur d'une autre structure. Pour manipuler le jour de naissance, par exemple, on accède au champ `jour` de la manière suivante : **`e.D.jour`**, avec `e` une variable de type **`struct`** `etudiant` et `D` le champ `date` de cette structure.

2. Allocation dynamique de la mémoire

2.1 Notion d'adresse et classes d'allocation

2.1.1 Notion d'adresse mémoire

Lors de la déclaration d'une variable, une case mémoire (zone, espace, emplacement) de taille correspondante est allouée et on pourra faire référence à cette case mémoire grâce au nom de la variable. Toutes les variables manipulées par un programme sont naturellement stockées en mémoire, et cette mémoire doit lui être réservée (allouée) afin que deux variables ne soient pas inscrites dans la même case.

Commençons par une vue simplifiée de l'organisation de la mémoire. Elle peut être considérée comme un arrangement de cases consécutives numérotées ou adressées, que l'on peut manipuler individuellement ou par groupes de cases contigües.

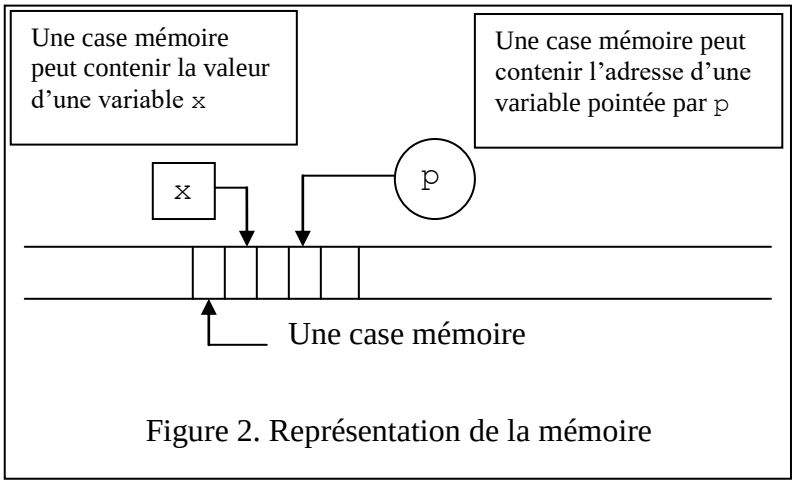


Figure 2. Représentation de la mémoire

Les cases mémoire sont destinées à contenir des valeurs de variables. Chacune de ces cases à un numéro l'identifiant, soit une adresse mémoire. Les adresses mémoire à leur tour peuvent être stockées dans des cases mémoire et manipulées par les pointeurs comme n'importe quelle autre variable.

2.1.2 Classes d'allocation

1- Automatique. Dans un programme, on déclare des variables mais on ne réserve pas de la mémoire, c'est le compilateur qui s'en occupe. Les données sont créées au fur et à mesure de l'exécution du programme et l'allocation se fait lors de l'exécution de chaque instruction de déclaration d'une variable. Dans une fonction, l'allocation s'effectue à l'entrée.

2- Statique. Allouée une seule fois au moment de l'édition de liens d'un programme. Les variables occupent un emplacement défini lors de la compilation et durant toute l'exécution du programme. C'est le cas des variables globales et des variables déclarées avec le mot clé "`static`".

3- Dynamique. Dans certains cas, relativement fréquents, on ne peut tout simplement pas savoir à l'avance (avant la compilation) quel est le nombre de variables qui seront manipulées par le programme ni quelle sera leur taille. Il convient alors de choisir d'autres classes d'allocation de mémoire qui dépendra de chaque exécution d'un programme donnée. On parlera de l'allocation dynamique de la mémoire ; en fonction des besoins de l'utilisateur et selon le déroulement de l'exécution, on alloue de l'espace mémoire dynamiquement. Les données sont créées ou détruites à l'initiative du programmeur. Elles sont créées dans le tas de la mémoire.

2.2 Allocation dynamique de la mémoire

En gestion dynamique, c'est le programmeur qui commande et retourne la mémoire au système d'exploitation. La gestion dynamique de la mémoire est une méthode de programmation qui consiste à allouer et libérer explicitement la mémoire dans un programme.

Puisqu'une variable allouée dynamiquement n'a pas d'identificateur étant donné qu'elle n'était a priori pas connue à la compilation, et n'a donc pas pu être déclarée, on utilisera alors un pointeur pour référencer cet espace mémoire alloué dynamiquement à cette variable.

N.B : Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection `*`, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée **`NULL`** définie dans `stdio.h`. En général, cette constante vaut 0. Le test `p == NULL` permet de savoir si le pointeur `p` pointe vers une variable. On peut initialiser un pointeur `p` par une affectation sur `p`. Par exemple, on peut affecter à `p` l'adresse d'une autre variable. Il est également possible d'affecter directement une valeur à `*p`. Mais pour cela, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate.

2.3 Fonctions d'allocation de la mémoire

N.B : Ce sont des fonctions de la bibliothèque « `stdlib.h` ».

1- malloc. La fonction qui réalise l'opération d'allocation dynamique de la mémoire en C, son prototype est : `void* malloc(size_t size);`

Cette fonction, lors de l'exécution, admet un paramètre qui est la taille en octets de l'élément désiré, alloue une zone mémoire, s'il y en a de disponible, et renvoie un pointeur au début de cette zone. S'il n'y a plus de mémoire disponible, elle renvoie `NULL`.

Exemple 1. Pour allouer de l'espace mémoire à un nombre de caractères (5 dans l'exemple ci-après), on doit déclarer un pointeur de type `char` et ensuite on alloue dynamiquement de l'espace mémoire en précisant la taille de cet espace à la fonction `malloc`.

```
...
char *p;
p = (char *)malloc(5);
...
```

La partie du code précédente permet d'allouer dynamiquement 5 cases mémoire et de les faire référencer par le pointeur `p`. Cet espace est destiner à contenir des valeurs de type `char`.

Remarque. On remarque clairement l'utilisation de l'opérateur « cast » pour convertir le type de retour de la fonction `malloc`, qui est en principe `void*`, vers le type du pointeur `p`, qui est `char*` dans l'exemple précédent. Cette conversion sera faite tout le temps.

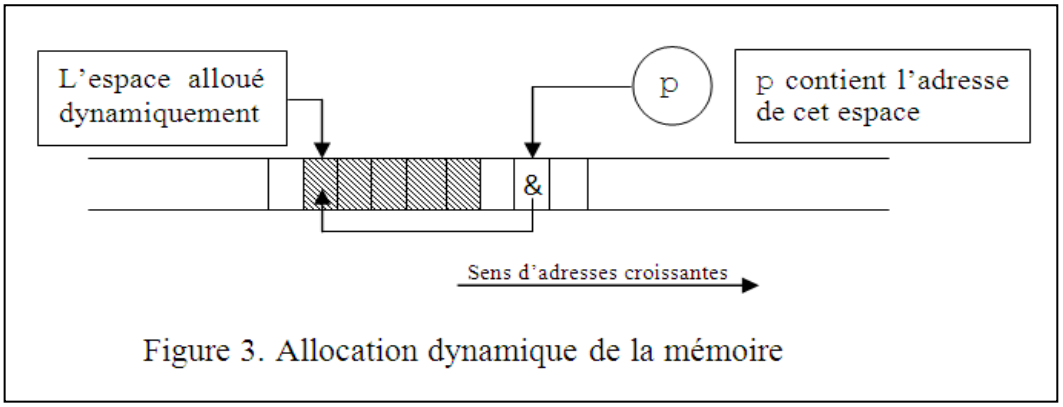


Figure 3. Allocation dynamique de la mémoire

Exemple 2. L'allocation dynamique s'avère très utile lorsqu'elle appliquée aux structures, vu que les structures sont aussi des variables. Cependant, étant donnée une structure quelconque, on ne peut pas envisager sa taille qu'elle occupera en mémoire ! Il est évident alors d'utiliser l'opérateur `sizeof()`.

```
...
struct nbComplexe *p;
p = (struct nbComplexe *)malloc(sizeof(struct nbComplexe *));
...
```

Cette partie du code permettra d'allouer dynamiquement un espace mémoire adéquat pour le stockage des différents champs de la structure « nbComplexe » et la taille de cet espace mémoire est parfaitement identique à la taille de la structure elle-même puisqu'on utilise l'opérateur **`sizeof`**.

Dans un autre exemple, on pourrait même écrire :

```
...
int *p;
p = (int *)malloc(sizeof(int));
...
```

Ce qui permettra d'allouer un espace mémoire destiné à contenir une valeur de type **`int`**.

Exemple 3. La fonction `malloc` permet également d'allouer un espace pour plusieurs objets contigus en mémoire. On peut écrire par exemple :

```
...
int *p;
p = (int*)malloc(21 * sizeof(int));
...
```

On a ainsi alloué, à l'adresse donnée par la valeur de `p`, des cases en mémoire, qui permettent de stocker 21 variables de type `int`.

2- calloc. Cette fonction permet l'allocation dynamique de la mémoire pour plusieurs blocs de tailles identiques, son prototype est : `void* calloc(type nb_elements, size_t size);`
Elle admet deux paramètres :

1- Le premier est le nombre d'éléments désirés ;
2- Le second est la taille en octets de chaque élément.

Son but est d'allouer un espace suffisant pour contenir les éléments demandés et de rendre un pointeur vers cet espace. Le bloc alloué est initialisé à 0.

Utilisation typique :

```
...
struct etudiant *p;
int nb_elements;

... /* initialisation de nb_elements */

p=(struct etudiant *)calloc(nb_elements , sizeof(struct
etudiant));
...
```

On allouerait alors « nb_elements » d'espaces mémoires. Chaque espace mémoire à la taille d'une structure « étudiant ». On peut alors utiliser les éléments p[0], p[1], ... p[nb_elements-1] pour manipules les variables.

Exemple 4. Pour allouer de l'espace mémoire à un nombre de structures « étudiant » (3 dans l'exemple ci-après), on doit déclarer un pointeur de type `struct etudiant` et ensuite on alloue dynamiquement de l'espace mémoire en précisant le nombre (3) et la taille de cet espace (`sizeof(struct etudiant)`) à la fonction `calloc`. `p` pointe vers l'espace mémoire alloué.

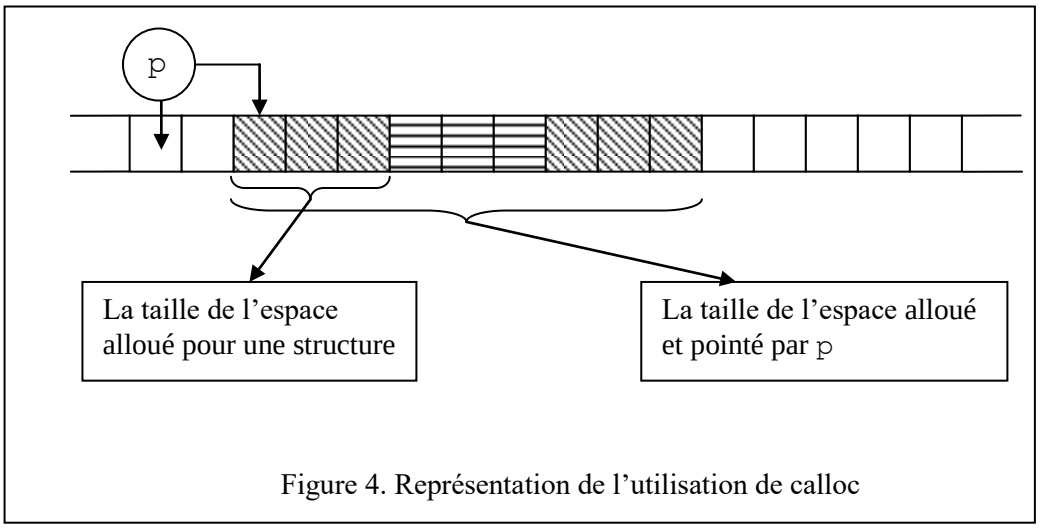


Figure 4. Représentation de l'utilisation de calloc

3- realloc. Cette fonction permet de réallouer la mémoire, déjà allouée, encore une fois tout en précisant une nouvelle taille. Elle est utilisée forcément après l'utilisation de `malloc` ou `calloc`. Son principe d'utilisation est le suivant : une fois qu'on a alloué de l'espace mémoire et on travaillé avec, dans une étape ultérieure, il se trouve qu'il est souhaitable de changer la taille de l'espace déjà alloué, on utilise alors `realloc`. Son prototype est : `void* realloc(void* Bloc, size_t Taille);`

1- `Bloc` : Pointeur sur le bloc mémoire créé par `malloc`, `calloc` ou `realloc`
2- `Taille` : Nouvelle taille de `Bloc`

`realloc` alors agrandit ou diminue la taille de la mémoire pointée par `Bloc` à la taille `Taille`. S'il n'est pas possible d'effectuer le changement, dans le cas d'agrandissement de la taille de la mémoire pointée par `Bloc`, `realloc` réserve un autre espace mémoire et y copie les données du premier bloc. Un pointeur générique sur le nouvel espace mémoire est retourné.

Si `Bloc` égale 0, `realloc` effectue le même travail que `malloc`. Si `Taille` est égale à 0, la mémoire est libéré et le pointeur `NULL` (0) est retourné.

2.4 Fonction de libération de la mémoire

La libération de la mémoire en C s'effectue par la fonction free de la bibliothèque «`stdlib.h` ». Son prototype est : `void` **`free`**`(type* Bloc) ;`

Où `Bloc` désigne le pointeur qui pointe vers un espace mémoire qui a été déjà alloué par l'une des fonctions de l'allocation dynamique.

Si on prend l'exemple 4 du paragraphe (*2.3 Fonctions d'allocation de la mémoire*), free(p) permet de libérer l'espace mémoire qui a été alloué pour stocker les variables pour les 3 structures étudiants.

2.5 Utilisation d'un tableau dynamique

L'équivalence entre les tableaux et les pointeurs permet de réaliser des tableaux dynamiques, c'est-à-dire des tableaux dont la taille n'est pas connue au moment de la compilation mais uniquement lors de l'exécution, lorsque le tableau commence à exister. Pour cela il suffit de :

1- Remplacer la déclaration du tableau par celle d'un pointeur ;
2- Allouer l'espace à l'exécution, avant toute utilisation du tableau, par un appel d'une fonction d'allocation dynamique ;
3- Dans le cas d'un tableau local (dans une fonction), libérer l'espace à la fin de l'utilisation.

Pour tout le reste, l'utilisation de ces tableaux dynamiques est identique à celle des tableaux normaux (ce qui, pour le programmeur, est une très bonne nouvelle !)

Exemple. La fonction ci-après calcule dans un tableau la $N^{ème}$ ligne du triangle de Pascal. On notera qu'une seule ligne diffère entre la version à tableau statique (vu en première année de SUP) et celle avec tableau dynamique.

```
void Pascal(int N) {

   int i, j, p;

   int *tab;

   tab = malloc(N*sizeof(int));

   for (i = 0 ; i<N ; i++) {
       tab[0] = tab[i] = 1;
       for (p = i - 1 ; p > 0 ; p--)
           tab[p] += tab[p - 1];
           // exploitation de tab ; par exemple, affichage
       for (j = 0; j<=i; j++)
           printf("%5d", tab[j]);
       printf("\n");
   }
   free(tab);  //à ne pas oublier tab est une variable
locale
}
```

3. Les listes chaînées

La structure la plus utilisée pour manipuler plusieurs données est le tableau, qui est implémenté de façon native dans le langage C.

Cependant dans certains cas, les tableaux, qui représentent une organisation contigüe en mémoire, ne constituent pas la meilleure solution pour stocker et manipuler les données. Cette représentation, dite contigüe, impose que la taille maximale du tableau soit connue a priori (on a besoin du nombre d'éléments du tableau lors de l'allocation dynamique).

En plus, pour insérer un élément dans un tableau, il faut d'abord déplacer tous les éléments qui sont en amont de l'endroit où l'on souhaite effectuer l'insertion, ce qui peut prendre un temps non négligeable proportionnel à la taille du tableau et à l'emplacement du futur élément. Il en est de même pour la suppression d'un élément ; bien sûr ceci ne s'applique pas en cas d'ajout ou de suppression en fin de tableau.

Pour faire face aux problèmes de la taille ainsi que la manipulation des données, on peut organiser les données sous une autre forme qui permet, d'une part, de construire une suite d'éléments de même type, sans prévoir de taille à l'avance. D'autre part, on permettra une gestion plus dynamique de ces données. La liste chaînée sera le formalisme le plus idéal pour cette conception.

Les listes chaînées sont des structures de données à accès indirect. Elles sont dynamiques d'une part. En effet, on alloue de la mémoire pour chaque maillon de la chaîne, donc si la taille de la liste varie beaucoup on prend moins de ressources inutilement. D'autre part, on peut facilement manipuler des éléments de la liste.

Avec une liste chaînée, le temps d'insertion et de suppression, par exemple, d'un élément est constant quelques soient l'emplacement de celui-ci et la taille de la liste. Elles sont aussi très pratiques pour réarranger les données, cet avantage est la conséquence directe de la facilité de manipulation des éléments.

Plus concrètement, une liste simplement chaînée est en fait constituée de maillons. Le maillon, élément de base de la chaîne, est une structure appelée *cellule* ou bien *élément* qui contient :

1- La valeur d'un élément de la liste ;
2- Un pointeur sur l'élément suivant.

La figure suivante représente de façon schématique un élément d'une liste :

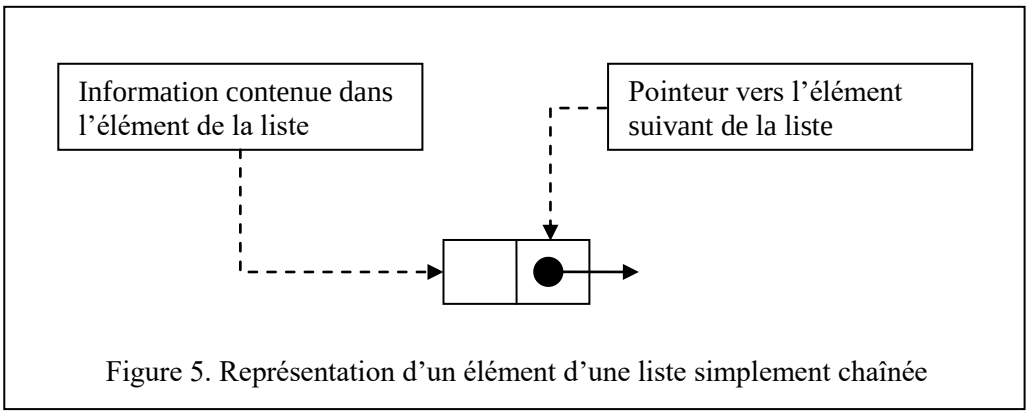


Figure 5. Représentation d'un élément d'une liste simplement chaînée

La liste est alors définie comme un pointeur sur le premier élément de la chaîne. Chaque élément pointe vers l'élément suivant. Le dernier élément pointe sur la liste vide `NULL`. Voici une représentation possible :

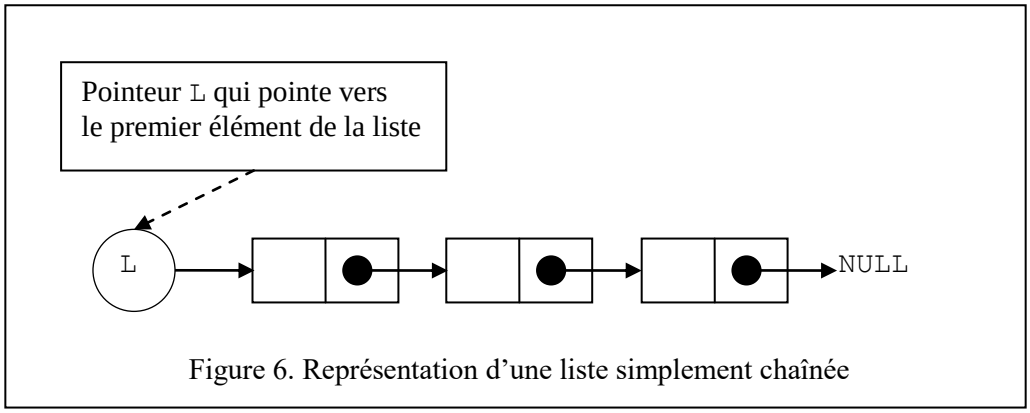


Figure 6. Représentation d'une liste simplement chaînée

Bien sûr, ce n'est qu'une représentation, il se peut que les structures qui composent la liste ne soient pas placées dans l'ordre en mémoire et encore moins de façon contiguë. Chaque élément pourra comporter plusieurs champs qui peuvent être examinés dès lors que l'on possède un pointeur sur cet élément. Ces ensembles supportent potentiellement toute une série d'opérations : Initialisation de la liste, ajout d'un élément, suppression d'un élément, recherche d'un élément, accès a l'élément suivant, accès aux champs d'un élément, calcul de la taille de la liste, teste si la liste est vide, affichage de la liste..etc.

3.1 Déclaration d'une liste chaînée

Pour déclarer une liste, on déclare en effet un élément, le modèle de structure, de la liste. Cet élément doit contenir deux champs ; un champ d'informations qui stocke les informations de l'élément et un champ de type pointeur vers une structure du même type.

```
struct modele{

        type    info ;              // le champ information

        struct modele *suivant;
                        // pointeur vers l'élément suivant

};
```

N.B : « **type** » peut être un type simple (int, char, ..) ou bien un type composé (structure).

Le principal problème des listes simplement chaînées est l'absence de pointeur sur l'élément précédent, il est donc possible de parcourir la chaîne uniquement du début vers la fin. Pour faciliter l'implémentation, il convient de faire quelques simplifications :

1- Sauvegarde du premier élément de la liste, pointé par `L` ;
2- Parcourir la liste avec un pointeur temporaire ;

N.B : Dans la suite de ce cours, on va considérer l'exemple d'une liste d'entiers afin d'illustrer certains exemples de manipulation. Il en va de même pour n'importe quels autres types de listes. La déclaration est la suivante :

```
struct Liste{
            int valeur;
            struct Liste *suivant;
};
```

3.2 Création d'une liste chaînée

La création d'une liste, en programmation, est la déclaration d'une variable de type « liste » proprement dit. Puisqu'il s'agit d'une création, la liste alors est forcément vide ; elle ne contient aucun élément, se qui se traduit par la déclaration d'un pointeur qui pointe vers `NULL`. Ce pointeur a la possibilité de pointer prochainement vers le début d'une liste chaînée de même type.

```
struct Liste *L ;
L = NULL ;
```

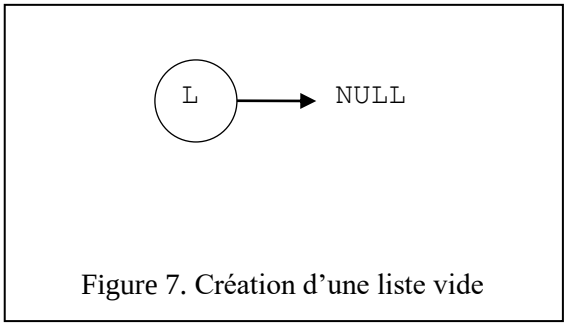


Figure 7. Création d'une liste vide

3.3 Insertion d'un élément dans une liste

Les implémentations d'insertion décrites dans ce paragraphe ne sont que des exemples. On peut avoir des fonctions avec plusieurs variations (arguments, valeur de retour) dans le cas de déclarations globales par exemple. Les cas traités considèrent aussi des listes avec des éléments existants ; c'est-à-dire on ne teste pas les cas ou la liste peut être vide par exemple. Ceci peut faire l'objet d'autres implémentations.

3.3.1 Insertion au début d'une liste

L'insertion ou bien l'ajout d'un élément au début de la liste nécessite tout d'abord la création de l'élément à ajouter, qui va être noté « tmp ». Cet élément nécessite une allocation dynamique de la mémoire. Puis on peut l'insérer dans la liste grâce aux instructions de branchement.

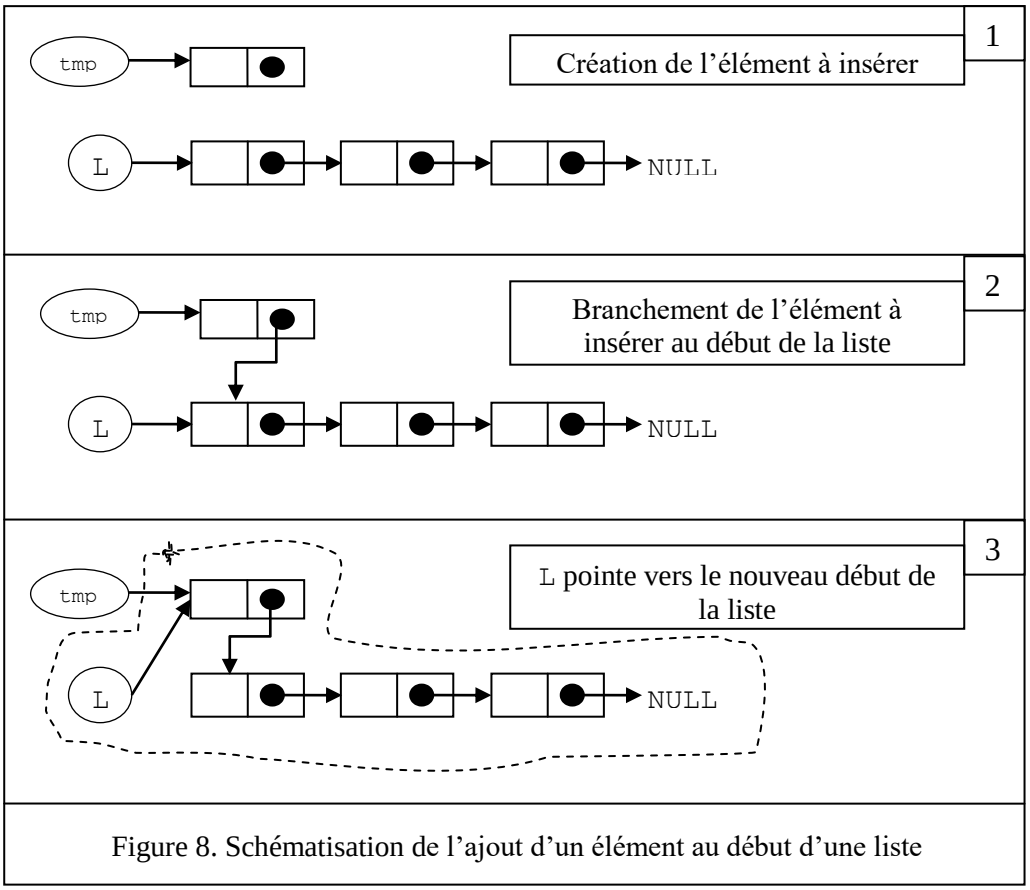


Figure 8. Schématisation de l'ajout d'un élément au début d'une liste

```
struct  Liste* AjoutDebut(struct  Liste* L){
   int v;
   struct  Liste* tmp;
   tmp =(struct Liste*)malloc(sizeof(struct Liste));

   printf("\n Entrer la valeur à ajouter :");
   scanf("%d",&v);

   tmp->valeur=v;
   tmp->suivant=L;
   L=tmp;
   return (L);
}
```

3.3.2 Insertion au milieu d'une liste

L'ajout d'un élément au milieu de la liste nécessite le parcours de cette liste jusqu'à l'endroit approprié. Premièrement, on crée l'élément à ajouter, noté « tmp ». Cet élément nécessite une allocation dynamique de la mémoire. Puis on parcourt la liste avec un nouveau pointeur, noté « courant » en comparant les éléments de la liste avec celui ou on veut insérer. Une fois l'emplacement d'insertion trouvé, on peut insérer l'élément « tmp » dans la liste grâce aux instructions de branchement.

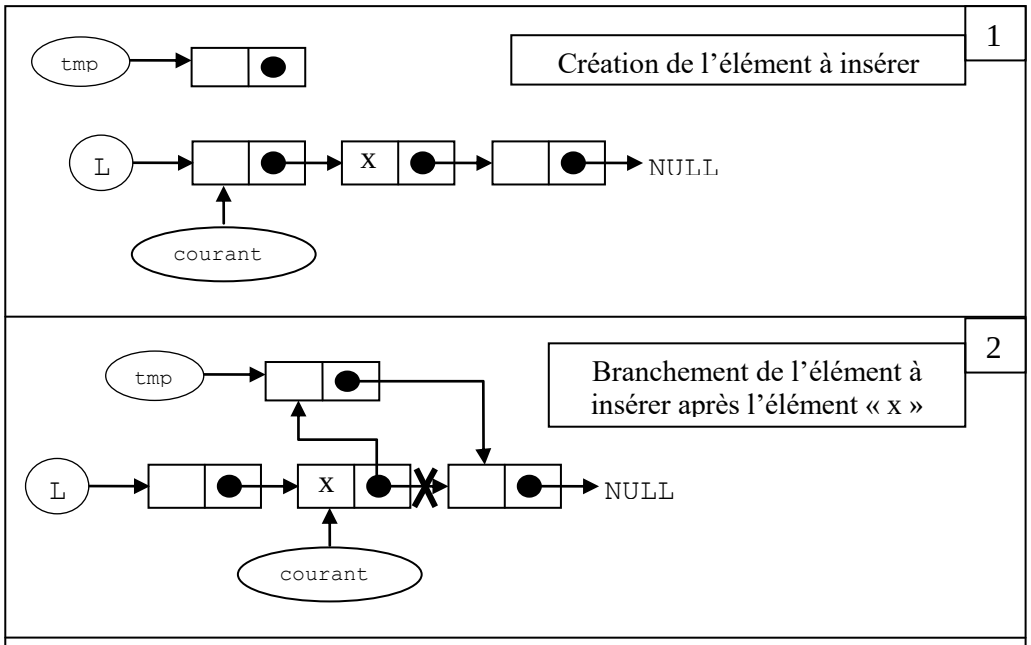


Figure 9. Schématisation de l'ajout d'un élément au milieu d'une liste

```
struct  Liste* AjoutMilieu(struct  Liste* L,int x){
   int v;
   struct  Liste *tmp , *courant;
   courant = L;
   tmp =(struct Liste*)malloc(sizeof(struct Liste));

   printf("\n Entrer la valeur à ajouter :");
   scanf("%d",&v);

   tmp->valeur=v;
   while(courant->valeur != x && courant != NULL){
       courant = courant->suivant ;
   }
   tmp->suivant = courant->suivant;
   courant->suivant=tmp ;
   return (L);
}
```

3.3.3 Insertion à la fin d'une liste

L'ajout d'un élément à la fin d'une liste est plus simple que l'ajout au milieu. Il s'agit de parcourir la liste avec un pointeur, noté « courant » tout en comparant son élément suivant avec NULL, une fois on atteint cette condition, on peut insérer l'élément « tmp » qui est toujours crée de la même façon.

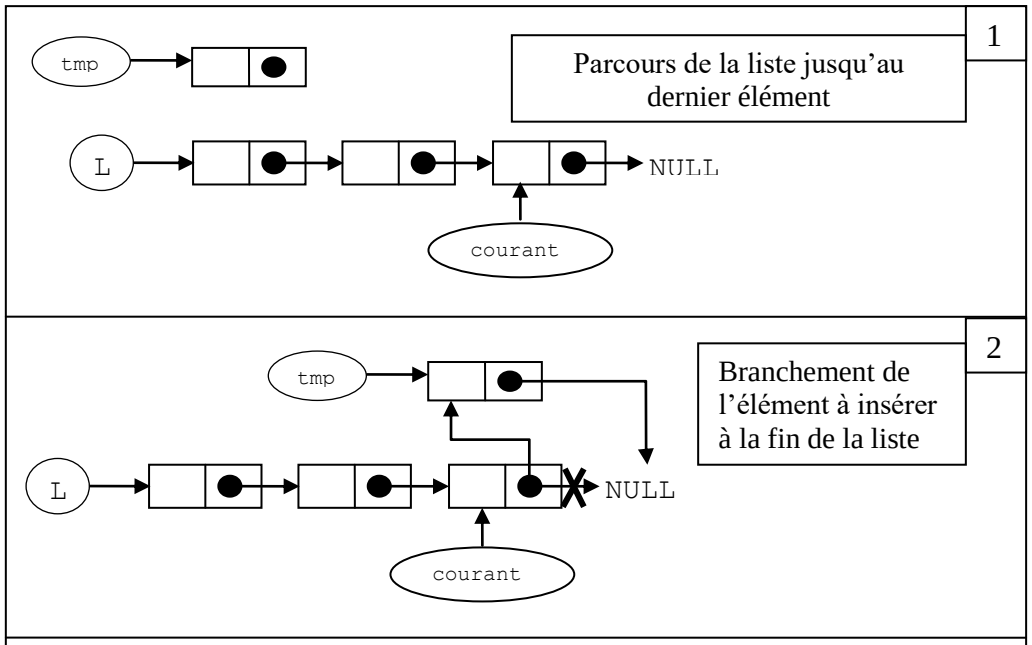


Figure 10. Schématisation de l'ajout d'un élément à la fin d'une liste

```
struct  Liste* AjoutFin(struct  Liste* L){
   int v;
   struct  Liste   *tmp,*courant;
   courant = L;
   tmp =(struct Liste*)malloc(sizeof(struct Liste));

   printf("\n Entrer la valeur à ajouter :");
   scanf("%d",&v);

   tmp->valeur=v;
   while(courant->suivant != NULL){
       courant = courant->suivant ;
   }
   tmp->suivant=NULL;
   courant->suivant=tmp ;
   return (L);
}
```

3.4 Suppression d'un élément d'une liste

La suppression d'un élément ne demande pas d'allocation de mémoire ! Il suffit tout simplement que l'élément précédent de l'élément à supprimer pointe vers l'élément suivant de ce dernier.

3.4.1 Suppression au début d'une liste

Supprimer le premier élément de la liste consiste à déplacer le pointeur du début de la liste vers le second élément.

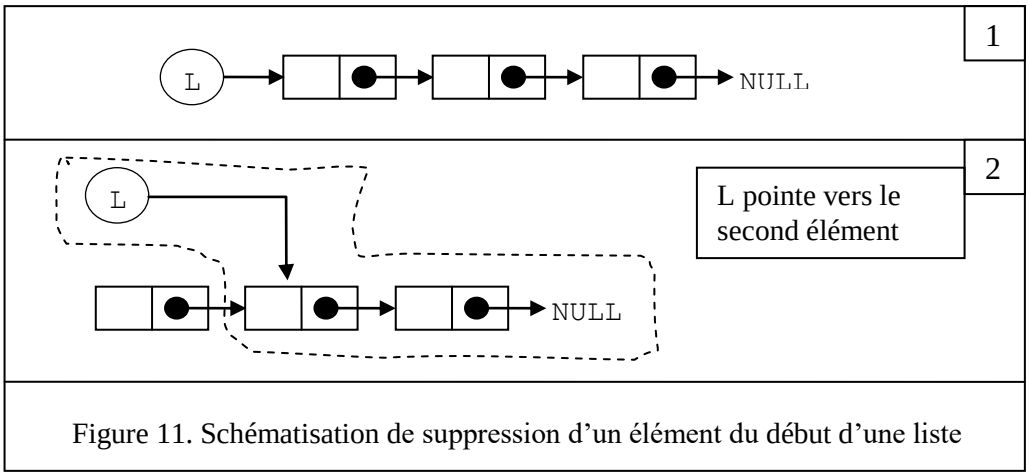


Figure 11. Schématisation de suppression d'un élément du début d'une liste

```
struct  Liste* SuppDebut(struct  Liste* L){

   L = L->suivant ;
   return (L);
}
```

3.4.2 Suppression au milieu d'une liste

Supprimer un élément au milieu de la liste consiste à préciser tout d'abord cet élément. Ensuite il faut que son précédent pointe vers son suivant. Pou ce faire il faut pointer vers l'élément qui est juste avant celui à supprimer. Le lier alors avec le suivant de l'élément à supprimer.

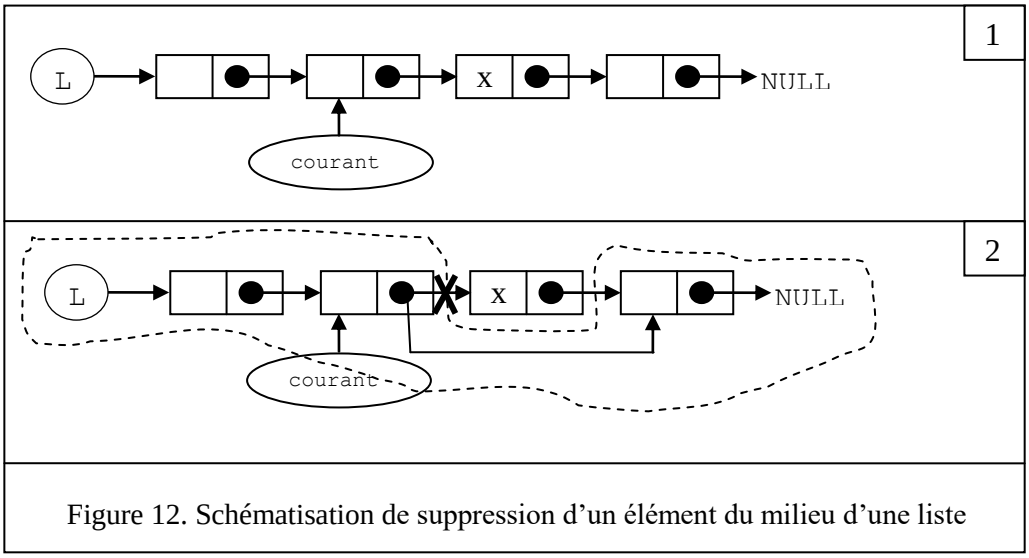


Figure 12. Schématisation de suppression d'un élément du milieu d'une liste

```
struct  Liste* SuppMilieu(struct  Liste* L, int x){

   struct  Liste *courant;
   courant = L;

   while(courant->suivant->valeur != x && courant !=
NULL){
        courant = courant->suivant ;
```

```
    }
    if(courant != NULL){
        courant->suivant = x->suivant;
        free(x) ;
    }
    return (L);
}
```

3.4.3 Suppression à la fin d'une liste

Pour supprimer le dernier élément de la liste, il faut deux pointeur ; un pour pointer sur cet élément, et le deuxième pour pointer juste avant. Cet élément va pointer vers NULL.

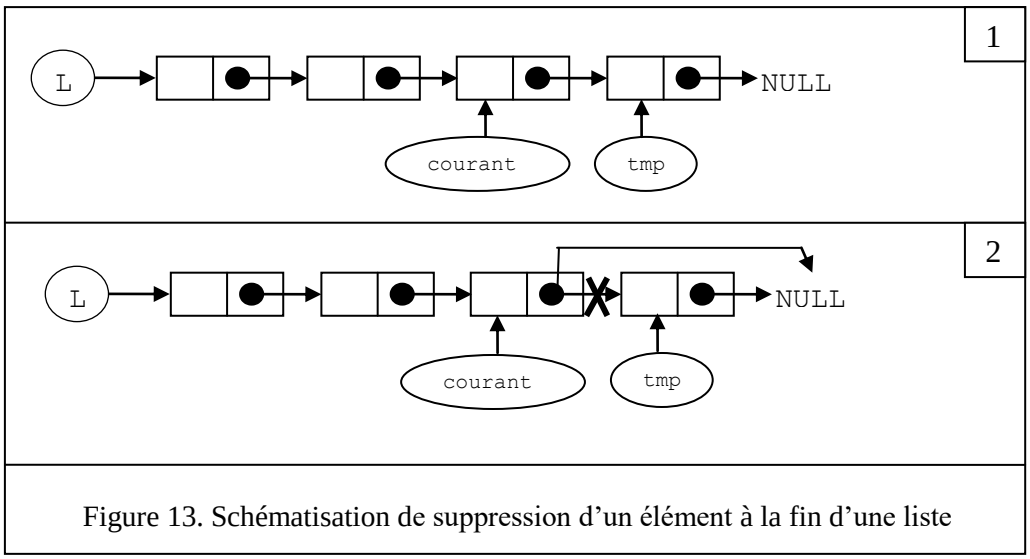


Figure 13. Schématisation de suppression d'un élément à la fin d'une liste

```
struct  Liste* SuppFin(struct  Liste* L, int x){

    struct  Liste *tmp,*courant;
    courant = L;
    tmp = L->suivant ;

    while( tmp->suivant ){

        tmp = tmp->suivant ;
        courant = courant->suivant ;
    }
    courant->suivant = NULL;
    free(tmp) ;
    }
    return (L);
}
```

3.5 Recherche d'un élément dans une liste

Dans l'implémentation suivante on va chercher si un élément « i » existe dans la liste ou non. S'il existe on retourne un pointeur sur cet élément, sinon on retourne `NULL`.

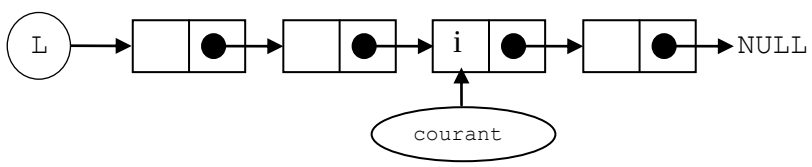


Figure 14. Schématisation de la recherche d'un élément dans une liste

```
struct  Liste* Recherche(struct  Liste* L, int i){

   struct  Liste *courant;
   courant = L;

   while(courant->valeur != i && courant != NULL){
       courant = courant->suivant ;
   }

   if(courant != NULL)
       return (courant);
   else
       return (NULL);
}
```

3.6 Etude des cas particuliers : Pile et File

3.6.1 La pile

Une pile (en anglais *stack*) est une structure de données fondée sur le principe *Dernier entré, premier sorti*, en anglais LIFO (*Last In, First Out*), ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés. Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée. Ceci étant dit, les différentes opérations sur les piles ne sont autorisées qu'au début de la pile.

3.6.1 La file

Une file est une structure de données basée sur le principe du *Premier entré, premier sorti*, en anglais FIFO (*First In, First Out*), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés. Le fonctionnement ressemble à une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file.

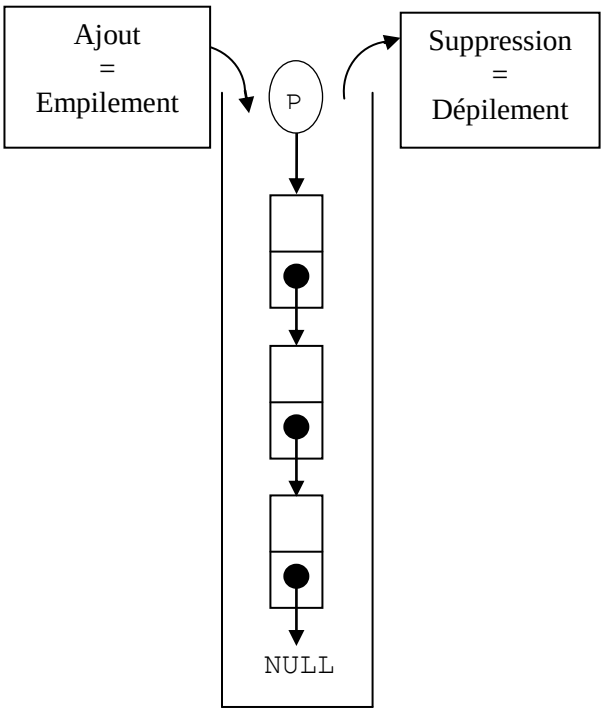


Figure 15. Représentation d'une pile

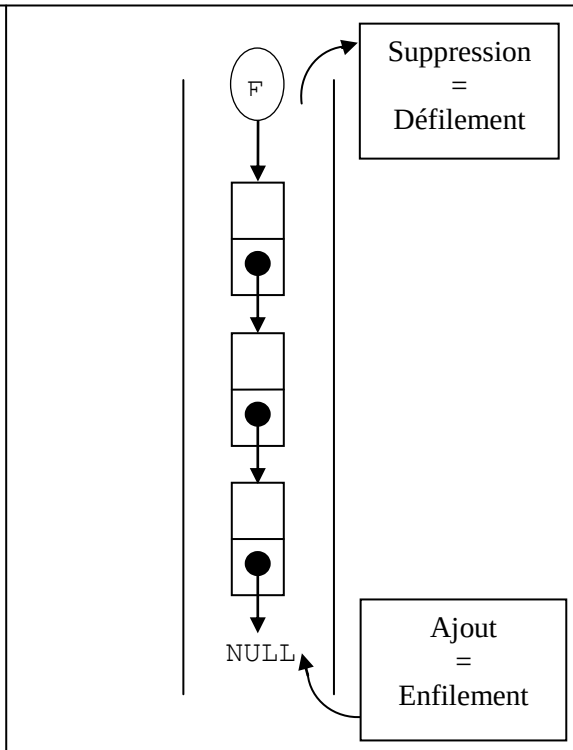


Figure 16. Représentation d'une file