# Les listes chaînées

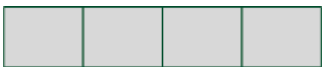
Pour stocker des données en mémoire, nous avons utilisé des variables simples (type `int`, `double`…), des tableaux et des structures personnalisées. Si vous souhaitez stocker une série de données, le plus simple est en général d'utiliser des tableaux.

Toutefois, les tableaux se révèlent parfois assez limités. Par exemple, si vous créez un tableau de 10 cases et que vous vous rendez compte plus tard dans votre programme que vous avez besoin de plus d'espace, il sera impossible d'agrandir ce tableau. De même, il n'est pas possible d'insérer une case au milieu du tableau.

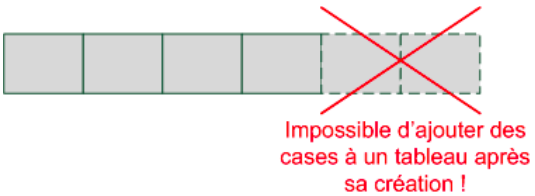
Les listes chaînées représentent une façon d'organiser les données en mémoire de manière beaucoup plus flexible. Comme à la base le langage C ne propose pas ce système de stockage, nous allons devoir le créer nous-mêmes de toutes pièces. C'est un excellent exercice qui vous aidera à être plus à l'aise avec le langage.

## Représentation d'une liste chaînée

Qu'est-ce qu'une liste chaînée ? Je vous propose de partir sur le modèle des tableaux. Un tableau peut être représenté en mémoire comme sur la fig. suivante. Il s'agit ici d'un tableau contenant des `int`.

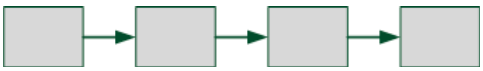


Comme je vous le disais en introduction, le problème des tableaux est qu'ils sont figés. Il n'est pas possible de les agrandir, à moins d'en créer de nouveaux, plus grands (fig. suivante). De même, il n'est pas possible d'y insérer une case au milieu, à moins de décaler tous les autres éléments.

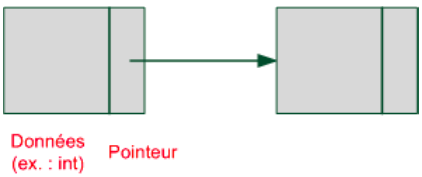



Le langage C ne propose pas d'autre système de stockage de données, mais il est possible de le créer soi-même de toutes pièces. Encore faut-il savoir comment s'y prendre : c'est justement ce que ce chapitre et les suivants vous proposent de découvrir.

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



Chaque élément peut contenir ce que l'on veut : un ou plusieurs `int`, `double`… En plus de cela, chaque élément possède un pointeur vers l'élément suivant (fig. suivante).




Je reconnais que tout cela est encore très théorique et doit vous paraître un peu flou pour le moment. Retenez simplement comment les éléments sont agencés entre eux : ils forment une **chaîne de pointeurs**, d'où le nom de « liste chaînée ».

## Construction d'une liste chaînée

Passons maintenant au concret. Nous allons essayer de créer une structure qui fonctionne sur le principe que nous venons de découvrir.
Je rappelle que tout ce que nous allons faire ici fait appel à des techniques du langage C que vous connaissez déjà. Il n'y a aucun élément nouveau, nous allons nous contenter de créer nos propres structures et fonctions et les transformer en un système logique, capable de se réguler tout seul.

### Un élément de la liste

Pour nos exemples, nous allons créer une liste chaînée de nombres entiers. Chaque élément de la liste aura la forme de la structure suivante :

```
typedef struct Element Element;

struct Element

{

    int nombre;

    Element *suivant;

};
```

Nous avons créé ici un élément d'une liste chaînée, correspondant à la fig. suivante que nous avons vue plus tôt. Que contient cette structure ?

- Une donnée, ici un nombre de type `int` : on pourrait remplacer cela par n'importe quelle autre donnée (un `double`, un tableau…). Cela correspond à ce que vous voulez stocker, c'est à vous de l'adapter en fonction des besoins de votre programme.
- Un pointeur vers un élément du même type appelé `suivant`. C'est ce qui permet de lier les éléments les uns aux autres : chaque élément « sait » où se trouve l'élément suivant en mémoire. Comme je vous le disais plus tôt, les cases ne sont pas côte à côte en mémoire. C'est la grosse différence par rapport aux tableaux. Cela offre davantage de souplesse car on peut plus facilement ajouter de nouvelles cases par la suite au besoin.

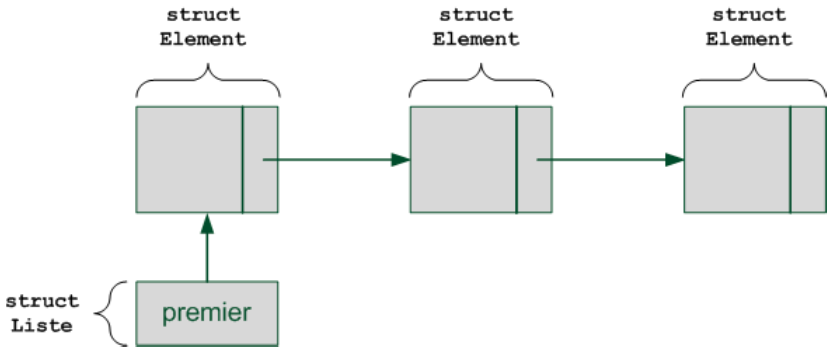
**La structure de contrôle**

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

```
typedef struct Liste Liste;

struct Liste

{

    Element *premier;

};
```

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

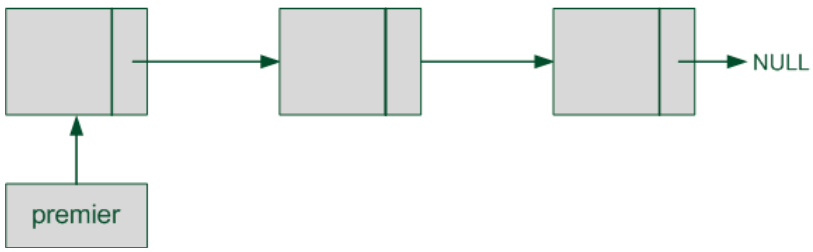
Nous n'aurons besoin de créer qu'un seul exemplaire de la structure `Liste`. Elle permet de contrôler toute la liste (fig. suivante).




**Le dernier élément de la liste**

Notre schéma est presque complet. Il manque une dernière chose : on aimerait retenir le dernier élément de la liste. En effet, il faudra bien arrêter de parcourir la liste à un moment donné. Avec quoi pourrait-on signifier à notre programme « Stop, ceci est le dernier élément » ?

Il serait possible d'ajouter dans la structure `Liste` un pointeur vers le dernier `Element`. Toutefois, il y a encore plus simple : il suffit de faire pointer le dernier élément de la liste vers `NULL`, c'est-à-dire de mettre son pointeur `suivant` à `NULL`. Cela nous permet de réaliser un schéma enfin complet de notre structure de liste chaînée (fig. suivante).




### Les fonctions de gestion de la liste

Nous avons créé deux structures qui permettent de gérer une liste chaînée :

- `Element`, qui correspond à un élément de la liste et que l'on peut dupliquer autant de fois que nécessaire ;
- `Liste`, qui contrôle l'ensemble de la liste. Nous n'en aurons besoin qu'en un seul exemplaire.

C'est bien, mais il manque encore l'essentiel : les fonctions qui vont manipuler la liste chaînée. En effet, on ne va pas modifier « à la main » le contenu des structures à chaque fois qu'on en a besoin ! Il est plus sage et plus propre de passer par des fonctions qui automatisent le travail. Encore faut-il les créer.

À première vue, je dirais qu'on aura besoin de fonctions pour :

- initialiser la liste ;

- ajouter un élément ;
- supprimer un élément ;
- afficher le contenu de la liste ;
- supprimer la liste entière.

On pourrait créer d'autres fonctions (par exemple pour calculer la taille de la liste) mais elles sont moins indispensables. Nous allons ici nous concentrer sur celles que je viens de vous énumérer, ce qui nous fera déjà une bonne base. Je vous inviterai ensuite à réaliser d'autres fonctions pour vous entraîner une fois que vous aurez bien compris le principe.

**Initialiser la liste**

La fonction d'initialisation est la toute première que l'on doit appeler. Elle crée la structure de contrôle et le premier élément de la liste.

Je vous propose la fonction ci-dessous, que nous commenterons juste après, bien entendu :

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

On commence par créer la structure de contrôle`liste`.

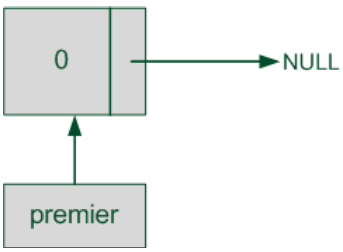
On alloue dynamiquement la structure de contrôle avec un`malloc`. La taille à allouer est calculée automatiquement avec`sizeof(*liste)`. L'ordinateur saura qu'il doit allouer l'espace nécessaire au stockage de la structure`Liste`.

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

Si tout s'est bien passé, on définit les valeurs de notre premier élément :

- la donnée`nombre`est mise à 0 par défaut ;
- le pointeur`suivant`pointe vers`NULL`car le premier élément de notre liste est aussi le dernier pour le moment. Comme on l'a vu plus tôt, le dernier élément doit pointer vers`NULL`pour signaler qu'il est en fin de liste.

Nous avons donc maintenant réussi à créer en mémoire une liste composée d'un seul élément et ayant une forme semblable à la fig. suivante.

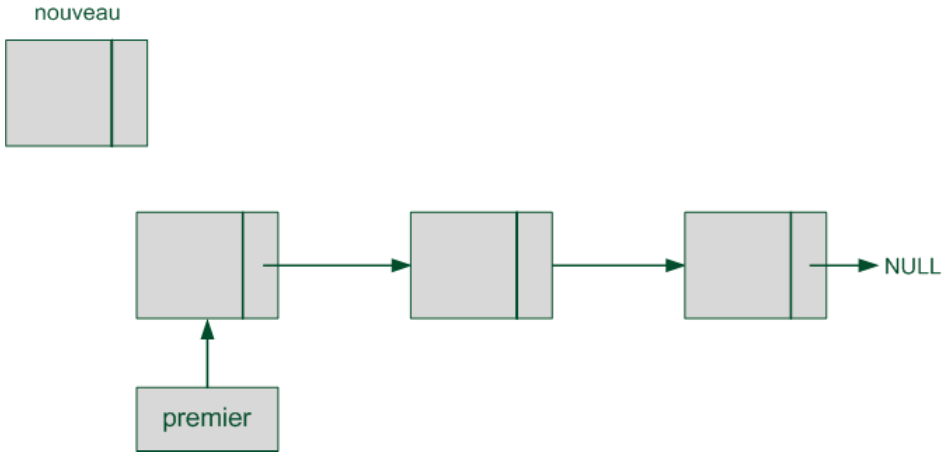



**Ajouter un élément**

Ici, les choses se compliquent un peu. Où va-t-on ajouter un nouvel élément ? Au début de la liste, à la fin, au milieu ?

La réponse est qu'on a le choix. Libre à nous de décider ce que nous faisons. Pour ce chapitre, je propose que l'on voie ensemble l'ajout d'un élément en début de liste. D'une part, c'est simple à comprendre, et d'autre part cela me donnera une occasion à la fin de ce chapitre de vous proposer de réfléchir à la création d'une fonction qui ajoute un élément à un endroit précis de la liste.

Nous devons créer une fonction capable d'insérer un nouvel élément en début de liste. Pour nous mettre en situation, imaginons un cas semblable à la fig. suivante : la liste est composée de trois éléments et on souhaite en ajouter un nouveau au début.




Il va falloir adapter le pointeur`premier`de la liste ainsi que le pointeur`suivant`de notre nouvel élément pour « insérer » correctement celui-ci dans la liste. Je vous propose pour cela ce code source que nous analyserons juste après :

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

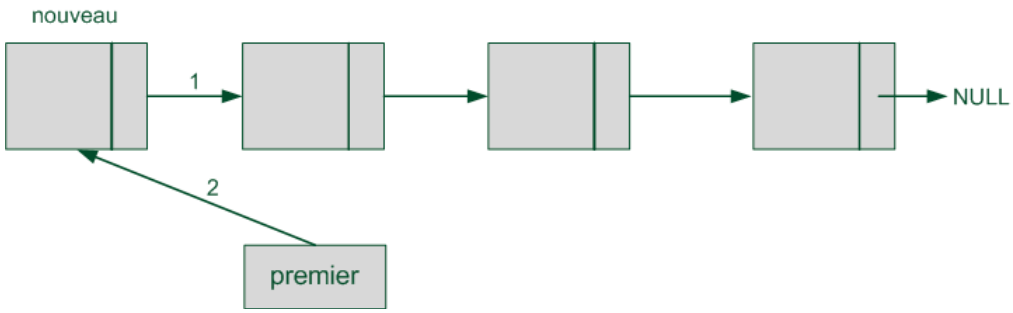
La fonction`insertion()`prend en paramètre l'élément de contrôle`liste`(qui contient l'adresse du premier élément) et le nombre à stocker dans le nouvel élément que l'on va créer.

Dans un premier temps, on alloue l'espace nécessaire au stockage du nouvel élément et on y place le nouveau nombre`nvNombre`. Il reste alors une étape délicate : l'insertion du nouvel élément dans la liste chaînée.

Nous avons ici choisi pour simplifier d'insérer l'élément en début de liste. Pour mettre à jour correctement les pointeurs, nous devons procéder dans cet ordre précis :

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

Cela aura pour effet d'insérer correctement notre nouvel élément dans la liste chaînée (fig. suivante) !




**Supprimer un élément**

De même que pour l'insertion, nous allons ici nous concentrer sur la suppression du premier élément de la liste. Il est techniquement possible de supprimer un élément précis au milieu de la liste, ce sera d'ailleurs un des exercices que je vous proposerai à la fin.

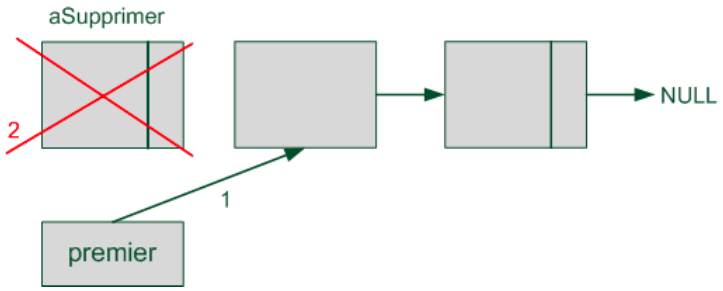
La suppression ne pose pas de difficulté supplémentaire. Il faut cependant bien adapter les pointeurs de la liste dans le bon ordre pour ne « perdre » aucune information.

```
void suppression(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }
    if (liste->premier != NULL)
    {
        Element *aSupprimer = liste->premier;
        liste->premier = liste->premier->suivant;
        free(aSupprimer);
    }
}
```

On commence par vérifier que le pointeur qu'on nous envoie n'est pas`NULL`, sinon on ne peut pas travailler. On vérifie ensuite qu'il y a au moins un élément dans la liste, sinon il n'y a rien à faire.

Ces vérifications effectuées, on peut sauvegarder l'adresse de l'élément à supprimer dans un pointeur`aSupprimer`. On adapte ensuite le pointeur`premier`vers le nouveau premier élément, qui est actuellement en seconde position de la liste chaînée.

Il ne reste plus qu'à supprimer l'élément correspondant à notre pointeur`aSupprimer`avec un`free`(fig. suivante).




Cette fonction est courte mais sauriez-vous la réécrire ? Il faut bien comprendre qu'on doit faire les choses dans un ordre précis :

1. faire pointer`premier`vers le second élément ;
2. supprimer le premier élément avec un`free`.

Si on faisait l'inverse, on perdrait l'adresse du second élément !

**Afficher la liste chaînée**

Pour bien visualiser ce que contient notre liste chaînée, une fonction d'affichage serait idéale ! Il suffit de partir du premier élément et d'afficher chaque élément un à un en « sautant » de bloc en bloc.

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

Cette fonction est simple : on part du premier élément et on affiche le contenu de chaque élément de la liste (un nombre). On se sert du

pointeur`suivant`pour passer à l'élément qui suit à chaque fois.

On peut s'amuser à tester la création de notre liste chaînée et son affichage avec un`main`:

```
int main()
{
    Liste *maListe = initialisation();
    insertion(maListe, 4);
    insertion(maListe, 8);
    insertion(maListe, 15);
    suppression(maListe);
    afficherListe(maListe);
    return 0;
}
```

En plus du premier élément (que l'on a laissé ici à 0), on en ajoute trois nouveaux à cette liste. Puis on en supprime un. Au final, le contenu de la liste chaînée sera donc :

```
8 -> 4 -> 0 -> NULL
```

### Aller plus loin

Nous venons de faire le tour des principales fonctions nécessaires à la gestion d'une liste chaînée : initialisation, ajout d'élément, suppression d'élément, etc. Voici quelques autres fonctions qui manquent et que je vous invite à écrire, ce sera un très bon exercice !

- **Insertion d'un élément en milieu de liste** : actuellement, nous ne pouvons ajouter des éléments qu'au début de la liste, ce qui est généralement suffisant. Si toutefois on veut pouvoir ajouter un élément au milieu, il faut créer une fonction spécifique qui prend un paramètre supplémentaire : l'adresse de celui qui précèdera notre nouvel élément dans la liste. Votre fonction va parcourir la liste chaînée jusqu'à tomber sur l'élément indiqué. Elle y insèrera le petit nouveau juste après.
- **Suppression d'un élément en milieu de liste** : le principe est le même que pour l'insertion en milieu de liste. Cette fois, vous devez ajouter en paramètre l'adresse de l'élément à supprimer.
- **Destruction de la liste** : il suffit de supprimer tous les éléments un à un !
- **Taille de la liste** : cette fonction indique combien il y a d'éléments dans votre liste chaînée. L'idéal, plutôt que d'avoir à calculer cette valeur à chaque fois, serait de maintenir à jour un entier`nbElements`dans la structure`Liste`. Il suffit d'incrémenter ce nombre à chaque fois qu'on ajoute un élément et de le décrémenter quand on en supprime un.

Je vous conseille de regrouper toutes les fonctions de gestion de la liste chaînée dans des fichiers`liste_chainee.c`et`liste_chainee.h`par exemple. Ce sera votre première bibliothèque ! Vous pourrez la réutiliser dans tous les programmes dans lesquels vous avez besoin de listes chaînées.

Vous pouvez télécharger le projet des listes chaînées comprenant les fonctions que nous avons découvertes ensemble. Cela vous fera une bonne base de départ.

**Télécharger le projet**

**En résumé**

- Les listes chaînées constituent un nouveau moyen de stocker des données en mémoire. Elles sont plus flexibles que les tableaux car on peut ajouter et supprimer des « cases » à n'importe quel moment.
- Il n'existe pas en langage C de système de gestion de listes chaînées, il faut l'écrire nous-mêmes ! C'est un excellent moyen de progresser en algorithmique et en programmation en général.
- Dans une liste chaînée, chaque élément est une structure qui contient l'adresse de l'élément suivant.
- Il est conseillé de créer une structure de contrôle (du type`Liste`dans notre cas) qui retient l'adresse du premier élément.
- Il existe une version améliorée — mais plus complexe — des listes chaînées appelée « listes doublement chaînées », dans lesquelles chaque élément possède en plus l'adresse de celui qui le précède.