




# **Contrôlez l'ajout d'éléments avec les piles et les files**



Les piles et les files sont deux variantes des listes chaînées qui permettent de contrôler la manière dont sont ajoutés les nouveaux éléments. Cette fois, on ne va plus insérer de nouveaux éléments au milieu de la liste, mais seulement au début ou à la fin.

Les piles et les files sont très utiles pour des programmes qui doivent traiter des données qui arrivent au fur et à mesure.

### ► **Construisez une structure de pile**

Imaginez une pile de pièces : vous pouvez ajouter des pièces une à une en haut de la pile, mais aussi en enlever depuis le haut de la pile. Il est en revanche impossible d'enlever une pièce depuis le bas de la pile.

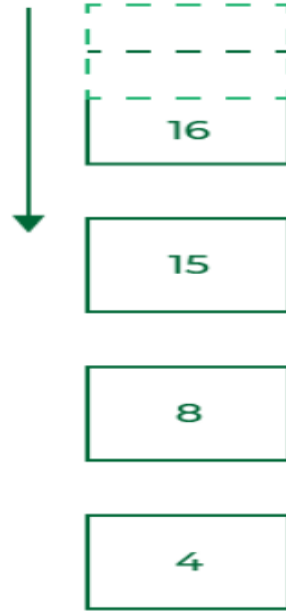
### ► **Comprenez le fonctionnement des piles**

Les piles permettent de stocker des données au fur et à mesure, les unes au-dessus des autres pour pouvoir les récupérer plus tard.

Imaginons par exemple une pile de nombres entiers de type int. Si on ajoute un élément (on parle d'empilage), il sera placé au-dessus, comme dans Tetris :



Empilage

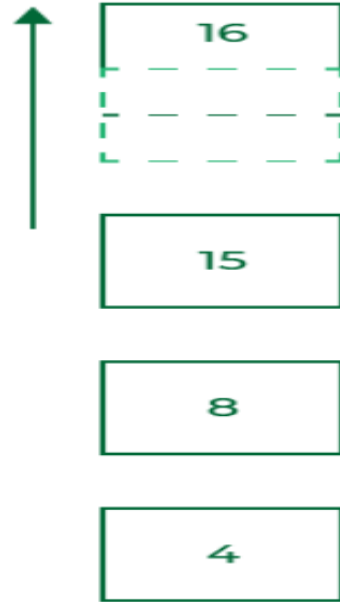


NB: L'opération qui consiste à extraire les nombres de la pile s'appellele **dépilage**. Il s'agit de récupérer les données une à une, en commençant par la dernière qui vient d'être posée tout en haut de la pile.

On enlève les données au fur et à mesure, jusqu'à la dernière tout en bas de la pile.



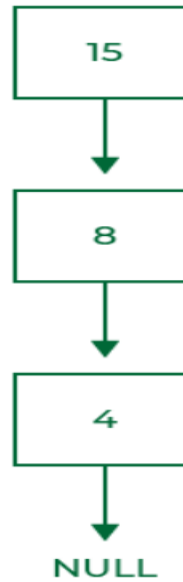
Dépilage



NB: On dit que c'est un algorithme **LIFO**, ce qui signifie "Last In First Out". Traduction : "Le dernier élément qui a été ajouté est le premier à sortir".

Les éléments de la pile sont reliés entre eux à la manière d'une liste chaînée. Ils possèdent un pointeur vers l'élément suivant, et ne sont donc pas forcément placés côte à côte en mémoire.

Le dernier élément (tout en bas de la pile) doit pointer vers NULL pour indiquer qu'on est arrivé au bout :



Il y a des programmes où vous avez besoin de stocker des données temporairement pour les ressortir dans un ordre précis : le dernier élément que vous avez stocké doit être le premier à ressortir.

Votre système d'exploitation utilise ce type d'algorithme pour retenir l'ordre dans lequel les fonctions ont été appelées.



Exemple :

1. Votre programme commence par la fonction **main**(comme toujours).
2. Vous y appelez la fonction **jouer**.
3. Cette fonction **jouer** fait appel à son tour à la fonction charger.
4. Une fois que la fonction **charger** est terminée, on retourne à la fonction **jouer**.
5. Une fois que la fonction **jouer** est terminée, on retourne au **main** .
6. Enfin, une fois le main terminé, il n'y a plus de fonction à appeler, le programme s'achève.

Pour retenir l'ordre dans lequel les fonctions ont été appelées, votre ordinateur crée une pile de ces fonctions au fur et à mesure. Grâce à cette technique, votre ordinateur sait à quelle fonction il doit retourner. Il peut empiler 100 fonctions d'affilée s'il le faut, il retrouvera toujours le main en bas !

## ► Créez un système de pile

Comme pour les listes chaînées, il n'existe pas de système de pile intégré au langage C. Il faut donc le créer nous-mêmes. Chaque élément de la pile aura une structure identique à celle d'une liste chaînée :

```
typedef struct Element Element;  
struct Element  
{  
    int nombre;  
    Element *suivant;  
};
```

La structure de contrôle contiendra l'adresse du premier élément de la pile, celui qui se trouve tout en haut :

```
typedef struct Pile Pile;  
struct Pile  
{  
    Element *premier;  
};
```



Nous aurons besoin des fonctions suivantes :

1. Empilage d'un élément.
2. Dépilage d'un élément.
3. Affichage de la pile.

**NB:** Contrairement aux listes chaînées, on ne parle pas d'ajout ni de suppression. On parle d'empilage et de dépilage, car ces opérations sont limitées à un élément précis. Ainsi, on ne peut ajouter et retirer un élément qu'en haut de la pile.

### ➤ Empilage d'un élément

Notre fonction **empiler** doit prendre en paramètre la structure de contrôle de la pile (de type `Pile`) ainsi que le nouveau nombre à stocker.

NB: nous stockons ici des **int**, mais rien ne vous empêche d'adapter ces exemples avec un autre type de données. On peut stocker n'importe quoi : des **double**, des **char**, des chaînes, des tableaux ou même d'autres structures !



```
void empiler(Pile *pile, int nvNombre)
{
    Element *nouveau = malloc(sizeof(*nouveau));
    if (pile == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = pile->premier;
    pile->premier = nouveau;
}
```

L'ajout se fait en début de pile car il est impossible de le faire au milieu d'une pile : on ajoute toujours par le haut. De ce fait, contrairement aux listes chaînées, on ne doit pas créer de fonction pour insérer un élément au milieu de la pile. Seule la fonction empiler permet d'ajouter un élément.

### ► Dépilage d'un élément

Le rôle de la fonction de dépilage est de supprimer l'élément tout en haut de la pile. Mais elle doit aussi retourner l'élément qu'elle dépile, c'est-à-dire dans notre cas le nombre qui était stocké en haut de la pile.

C'est comme cela que l'on accède aux éléments d'une pile : en les enlevant un à un. On ne parcourt pas la pile pour aller y chercher le second ou le troisième élément. On demande toujours à récupérer le premier.

Notre fonction **depiler** va donc retourner un **int** correspondant au nombre qui se trouvait en tête de pile :

```
int depiler(Pile *pile)
{
    if (pile == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDepile = 0;
    Element *elementDepile = pile->premier;

    if (pile != NULL && pile->premier != NULL)
    {
        nombreDepile = elementDepile->nombre;
        pile->premier = elementDepile->suivant;
        free(elementDepile);
    }

    return nombreDepile;
}
```

On récupère le nombre en tête de pile pour le renvoyer à la fin de la fonction. On modifie l'adresse du premier élément de la pile, puisque celui-ci change. Enfin, bien entendu, on supprime l'ancienne tête de pile grâce à **free**.

## ► Affichez la pile

Bien que cette fonction ne soit pas indispensable (les fonctions **empiler** et **depiler** suffisent à gérer une pile), elle est utile pour tester le fonctionnement de notre pile, et surtout pour visualiser le résultat:

```
void afficherPile(Pile *pile)
{
    if (pile == NULL)
    {
        exit(EXIT_FAILURE);
    }
    Element *actuel = pile->premier;

    while (actuel != NULL)
    {
        printf("%d\n", actuel->nombre);
        actuel = actuel->suivant;
    }

    printf("\n");
}
```

C'est le moment de faire un main pour tester le comportement de notre pile :

```
int main()
{
    Pile *maPile = initialiser();

    empiler(maPile, 4);
    empiler(maPile, 8);
    empiler(maPile, 15);
    empiler(maPile, 16);
    empiler(maPile, 23);
    empiler(maPile, 42);

    printf("\nEtat de la pile :\n");
    afficherPile(maPile);

    printf("Je depile %d\n", depiler(maPile));
    printf("Je depile %d\n", depiler(maPile));

    printf("\nEtat de la pile :\n");
    afficherPile(maPile);

    return 0;
}
```

On affiche l'état de la pile après plusieurs empilages, et une autre fois après quelques dépilages. On affiche aussi le nombre qui est dépilé à chaque fois que l'on dépile.

## ► Construisez une structure de file

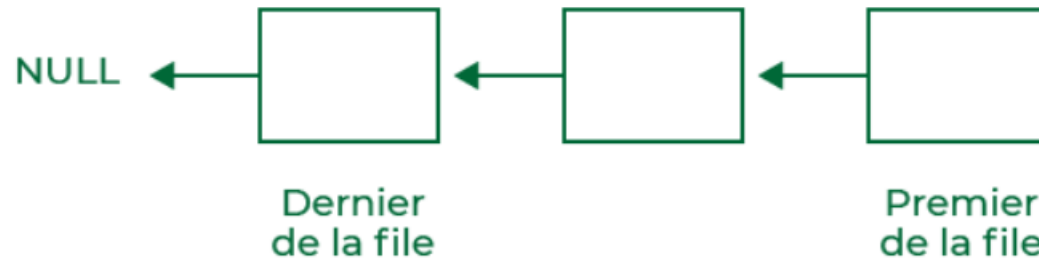
Les files ressemblent assez aux piles, si ce n'est qu'elles fonctionnent dans le sens inverse!

Dans ce système, les éléments s'entassent les uns à la suite des autres. Le premier qu'on fait sortir de la file est le premier à être arrivé. On parle ici d'algorithme **FIFO** ("First In First Out", c'est-à-dire "Le premier qui arrive est le premier à sortir"). C'est ce qu'il se passe par exemple dans la file d'attente pour un guichet de cinéma, le premier arrivé est le premier servi.

En programmation, les files mettent en attente des informations dans l'ordre dans lequel elles sont arrivées.

Dans un logiciel de messagerie instantanée, trois messages reçus à peu de temps d'intervalle forment en fait une file en mémoire : ils sont placés les uns à la suite des autres. Le premier message arrivé s'affiche à l'écran, puis le second, et ainsi de suite.

En C, une file est une liste chaînée où chaque élément pointe vers le suivant, tout comme les piles. Le dernier élément de la file pointe vers NULL :



### ➤ Créez un système de file

Nous allons créer une structure **Element** et une structure de contrôle **File**:

```
typedef struct Element Element;
struct Element
{
    int nombre;
    Element *suivant;
};

typedef struct File File;
struct File
{
    Element *premier;
};
```





Comme pour les piles, chaque élément de la file sera de type **Element**. À l'aide du pointeur **premier**, nous disposerons toujours du premier élément, et nous pourrons remonter jusqu'au dernier.

### ➤ Enfilage d'un élément

La fonction d'enfilage ajoute un élément à la file.

Il y a deux cas à gérer :

1. La file est vide : on crée la file en faisant pointer premier vers le nouvel élément créé.
2. La file n'est pas vide : on parcourt toute la file en partant du premier élément jusqu'à arriver au dernier. On rajoutera notre nouvel élément après le dernier.

Voici comment on peut faire dans la pratique :

```
void enfiler(File *file, int nvNombre)
{
    Element *nouveau = malloc(sizeof(*nouveau));
    if (file == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = NULL;

    if (file->premier != NULL) /* La file n'est pas vide */
    {
        /* On se positionne à la fin de la file */
        Element *elementActuel = file->premier;
        while (elementActuel->suivant != NULL)
        {
            elementActuel = elementActuel->suivant;
        }
        elementActuel->suivant = nouveau;
    }
    else /* La file est vide, notre élément est le premier */
    {
        file->premier = nouveau;
    }
}
```

Vous voyez dans ce code le traitement des deux cas possibles, chacun devant être géré à part. La différence par rapport aux piles, c'est qu'il faut se placer à la fin de la file pour ajouter le nouvel élément : un **while**, et le tour est joué !

## ► Défilage d'un élément

Le défilage ressemble étrangement au dépilage. Étant donné qu'on possède un pointeur vers le premier élément de la file, il nous suffit de l'enlever et de renvoyer sa valeur.

```
int defiler(File *file)
{
    if (file == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDefile = 0;

    /* On vérifie s'il y a quelque chose à défiler */
    if (file->premier != NULL)
    {
        Element *elementDefile = file->premier;

        nombreDefile = elementDefile->nombre;
        file->premier = elementDefile->suivant;
        free(elementDefile);
    }

    return nombreDefile;
}
```