

# La fonction calloc

La fonction

```
void *calloc(size_t nmemb, size_t size);
```

alloue et renvoie un pointeur sur une zone de la mémoire de `nmemb * size` octets, tous **initialisés** avec des 0.

# La fonction calloc

La fonction

```
void *calloc(size_t nmemb, size_t size);
```

alloue et renvoie un pointeur sur une zone de la mémoire de `nmemb * size` octets, tous **initialisés** avec des 0.

P.ex.,

```
long *ptr;  
ptr = (long *) calloc(13, sizeof(long));
```

alloue une zone de la mémoire pouvant accueillir 13 valeurs de type `long`, initialisées à 0.

- 5 Allocation dynamique
  - Pointeurs
  - Passage par adresse
  - Allocation dynamique
  - Tableaux dynamiques

# Tableaux dynamiques

Un **tableau dynamique** de **N** valeurs de type **T** est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

# Tableaux dynamiques

Un **tableau dynamique** de **N** valeurs de type **T** est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

P.ex.,

```
int *tab;  
tab = (int *) malloc(sizeof(int) * 66);
```

déclare un tableau dynamique de valeurs de type `int` de taille `66`.

# Tableaux dynamiques

Un **tableau dynamique** de **N** valeurs de type **T** est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

P.ex.,

```
int *tab;  
tab = (int *) malloc(sizeof(int) * 66);
```

déclare un tableau dynamique de valeurs de type `int` de taille `66`.

On lit et écrit dans un tableau dynamique de la même manière que dans un tableau statique. En effet, `tab` pointe vers la première case du tableau, et pour tout  $0 \leq i < 66$ , `(tab + i)` pointe vers la case d'indice `i`.

# Tableaux dynamiques

Un **tableau dynamique** de **N** valeurs de type **T** est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

P.ex.,

```
int *tab;  
tab = (int *) malloc(sizeof(int) * 66);
```

déclare un tableau dynamique de valeurs de type `int` de taille `66`.

On lit et écrit dans un tableau dynamique de la même manière que dans un tableau statique. En effet, `tab` pointe vers la première case du tableau, et pour tout  $0 \leq i < 66$ , `(tab + i)` pointe vers la case d'indice `i`.

La fonction `free` vue précédemment permet de désallouer un tableau. On utilise donc

```
free(tab);
```

pour libérer la zone mémoire occupée par `tab` (c.-à-d. les `66` entiers situés à partir de l'adresse `tab`).

# Création d'un tableau dynamique à deux dimensions

Un **tableau à deux dimensions** est un tableau dont chaque case est elle-même un tableau.

# Création d'un tableau dynamique à deux dimensions

Un **tableau à deux dimensions** est un tableau dont chaque case est elle-même un tableau. Un tableau dynamique à deux dimensions est donc un **pointeur sur un pointeur**.

# Création d'un tableau dynamique à deux dimensions

Un **tableau à deux dimensions** est un tableau dont chaque case est elle-même un tableau. Un tableau dynamique à deux dimensions est donc un **pointeur sur un pointeur**. Ceci se généralise dans le cas des tableaux à plus de deux dimensions.

# Création d'un tableau dynamique à deux dimensions

Un **tableau à deux dimensions** est un tableau dont chaque case est elle-même un tableau. Un tableau dynamique à deux dimensions est donc un **pointeur sur un pointeur**. Ceci se généralise dans le cas des tableaux à plus de deux dimensions.

Les instructions

```
int i, j, **tab;
tab = (int **) malloc(sizeof(int *) * 4);
if (tab == NULL) exit(EXIT_FAILURE);
for (i = 0; i < 4; ++i) {
    tab[i] = (int *) malloc(sizeof(int) * 3);
    if (tab[i] == NULL) exit(EXIT_FAILURE);
    for (j = 0; j < 3; ++j)
        tab[i][j] = 0;
}
```

permettent de créer un tableau dynamique à deux dimensions de taille  $4 \times 3$  de valeurs de type `int` initialisées à 0.

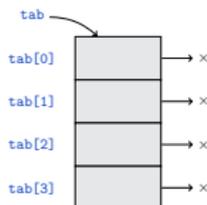
# Création d'un tableau dynamique à deux dimensions

`tab`  $\longrightarrow$   $\times$

Initialement, `tab` est pointeur vers une zone indéterminée de la mémoire.

# Création d'un tableau dynamique à deux dimensions

`tab` → ×

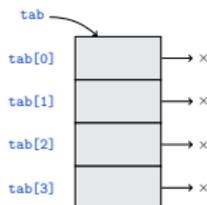


Initialement, `tab` est pointeur vers une zone indéterminée de la mémoire.

Juste après la 1<sup>re</sup> allocation dynamique, `tab` est dans cet état. L'espace de mémoire de 4 pointeurs sur des entiers à été réservé.

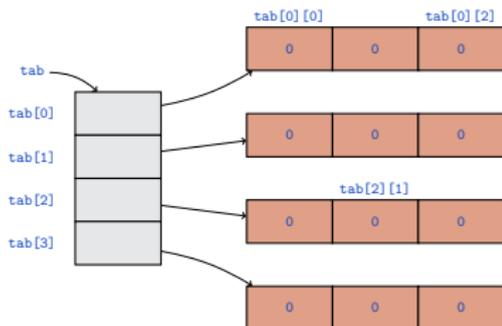
# Création d'un tableau dynamique à deux dimensions

`tab` → ×



Initialement, `tab` est pointeur vers une zone indéterminée de la mémoire.

Juste après la 1<sup>re</sup> allocation dynamique, `tab` est dans cet état. L'espace de mémoire de 4 pointeurs sur des entiers à été réservé.



Après les allocations dynamiques des tableaux à une dimension de taille 3 et des initialisations des cases d'entiers, `tab` est dans cet état.

# Destruction d'un tableau dynamique à deux dimensions

Pour libérer l'espace occupé par un tableau à deux dimensions, on utilise plusieurs fois la fonction `free`.

# Destruction d'un tableau dynamique à deux dimensions

Pour libérer l'espace occupé par un tableau à deux dimensions, on utilise plusieurs fois la fonction `free`.

Les instructions

```
for (i = 0; i < 4; ++i) {
    free(tab[i]);
    tab[i] = NULL;
}
free(tab);
tab = NULL;
```

permettent de libérer l'espace mémoire occupé par un tableau `tab` à deux dimensions de taille  $4 \times N$  de valeurs de type `T` où `N` est un entier strictement positif quelconque et `T` est un type.

# Destruction d'un tableau dynamique à deux dimensions

Pour libérer l'espace occupé par un tableau à deux dimensions, on utilise plusieurs fois la fonction `free`.

Les instructions

```
for (i = 0; i < 4; ++i) {
    free(tab[i]);
    tab[i] = NULL;
}
free(tab);
tab = NULL;
```

permettent de libérer l'espace mémoire occupé par un tableau `tab` à deux dimensions de taille  $4 \times N$  de valeurs de type `T` où `N` est un entier strictement positif quelconque et `T` est un type.

**Remarque** : la connaissance de la seconde dimension du tableau (`N`) n'est pas utile et n'intervient pas dans la suite d'instructions ci-dessus.

# La fonction realloc

Il est possible de **modifier la taille** d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

# La fonction realloc

Il est possible de **modifier la taille** d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

Celle-ci renvoie un pointeur sur une zone de la mémoire de taille `size` octets. Le contenu de cette zone mémoire est le même que celui de la zone pointée par `ptr`.

# La fonction realloc

Il est possible de **modifier la taille** d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

Celle-ci renvoie un pointeur sur une zone de la mémoire de taille `size` octets. Le contenu de cette zone mémoire est le même que celui de la zone pointée par `ptr`.

**Attention** : l'adresse de la zone de la mémoire réallouée peut être différente de son adresse d'origine.

# La fonction realloc

Il est possible de **modifier la taille** d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

Celle-ci renvoie un pointeur sur une zone de la mémoire de taille `size` octets. Le contenu de cette zone mémoire est le même que celui de la zone pointée par `ptr`.

**Attention** : l'adresse de la zone de la mémoire réallouée peut être différente de son adresse d'origine. En effet,

```
int *tab;
tab = (int *) malloc(sizeof(int) * 150);
printf("%p\n", tab);
tab = (int *) realloc(tab, sizeof(int) * 250000);
printf("%p\n", tab);
```

affiche

0x1205010

0x7fb123f9d010.

## Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

## Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;  
int t_max, t_reelle;
```

## Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;  
int t_max, t_reelle;  
t_reelle = 0;
```

# Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;  
int t_max, t_reelle;  
t_reelle = 0;  
t_max = 2;
```

## Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
```

## Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
```

# Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {

}
```

## Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {

    }

}
```

## Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
    }

}
```

```
}
```

## Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
}
```

# Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
}
```

# Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
}
```

## Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
```

# Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
chaine[t_reelle] = '\0';
```

# Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
chaine[t_reelle] = '\0';
chaine = (char *) realloc(chaine, t_reelle + 1); /* Diminution. */
```

# Exemple d'utilisation de realloc

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de '!'.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
chaine[t_reelle] = '\0';
chaine = (char *) realloc(chaine, t_reelle + 1); /* Diminution. */
printf("%s\n", chaine);
```

- 6 Entrées et sorties
  - Sortie
  - Entrée
  - Fichiers
  - Fichiers binaires

- 6 Entrées et sorties
  - Sortie
  - Entrée
  - Fichiers
  - Fichiers binaires

# Écriture formatée

La fonction

```
int printf(char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** sur la sortie standard `stdout`.

# Écriture formatée

La fonction

```
int printf(char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** sur la sortie standard `stdout`.

**Rappel** : cette fonction renvoie le nombre de caractères affichés. Si une erreur a lieu, elle renvoie un entier négatif.

# Écriture formatée

La fonction

```
int printf(char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** sur la sortie standard `stdout`.

**Rappel** : cette fonction renvoie le nombre de caractères affichés. Si une erreur a lieu, elle renvoie un entier négatif.

```
int num;  
num = printf("%s+%d\n",  
            "test", 200);  
printf("%d\n", num);
```

Ces instructions affichent

```
test+200  
9
```

# Indicateurs de conversion

On utilise les **indicateurs de conversion** suivants pour interpréter chaque valeur à écrire (ou à lire, voir la partie suivante) de manière adéquate :

Indicateur de conversion	Affichage
d, i	Entier en base dix
u	Entier non signé en base dix
x, X	Entier en hexadécimal
c	Caractère
e, E	Flottant en notation scientifique
f, g	Flottant
s	Chaîne de caractères
p	Pointeur

# Caractères spéciaux

Certains caractères ne s'affichent pas mais produisent un effet sur la sortie. Ce sont des **caractères spéciaux**.

Caractère spécial	Rôle
<code>\n</code>	Passage à la ligne suivante
<code>\b</code>	Retour en arrière d'un caractère
<code>\f</code>	Passage à la ligne suivante avec alinéa
<code>\r</code>	Retour chariot
<code>\t</code>	Tabulation horizontale
<code>\v</code>	Tabulation verticale

# Caractères d'attribut

Il est possible de faire suivre le % d'un indicateur de conversion de caractères d'attribut pour réaliser un formatage avancé.

Caractère d'attribut	Rôle
0N	Affichage du nombre sur N chiffres (ajout de 0)
N	Affichage du nombre sur N chiffres (ajout d'espaces)
+	Force l'affichage du signe d'un nombre
-	Justifie à gauche un nombre (à droite par défaut)

# Caractères d'attribut

Il est possible de faire suivre le % d'un indicateur de conversion de **caractères d'attribut** pour réaliser un formatage avancé.

Caractère d'attribut	Rôle
<b>0N</b>	Affichage du nombre sur <b>N</b> chiffres (ajout de <b>0</b> )
<b>N</b>	Affichage du nombre sur <b>N</b> chiffres (ajout d'espaces)
<b>+</b>	Force l'affichage du signe d'un nombre
<b>-</b>	Justifie à gauche un nombre (à droite par défaut)

P.ex.,

```
printf("%+5d\n", 23);
```

affiche **+23**

# Caractères d'attribut

Il est possible de faire suivre le % d'un indicateur de conversion de **caractères d'attribut** pour réaliser un formatage avancé.

Caractère d'attribut	Rôle
<code>ON</code>	Affichage du nombre sur <code>N</code> chiffres (ajout de <code>0</code> )
<code>N</code>	Affichage du nombre sur <code>N</code> chiffres (ajout d'espaces)
<code>+</code>	Force l'affichage du signe d'un nombre
<code>-</code>	Justifie à gauche un nombre (à droite par défaut)

P.ex.,

```
printf("%+5d\n", 23);
```

affiche `+23` et

```
printf("%+-5d %d\n", 23, 5);
```

affiche `+23  5`.

# Écriture caractère par caractère

La fonction

```
int putchar(int c);
```

permet d'afficher un caractère sur la sortie standard.

# Écriture caractère par caractère

La fonction

```
int putchar(int c);
```

permet d'afficher un caractère sur la sortie standard.

Cette fonction renvoie le caractère écrit (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) si une erreur a lieu.

# Écriture caractère par caractère

La fonction

```
int putchar(int c);
```

permet d'afficher un caractère sur la sortie standard.

Cette fonction renvoie le caractère écrit (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) si une erreur a lieu.

```
int ret;  
ret = putchar('a');  
if (ret == EOF)  
    exit(EXIT_FAILURE);
```

Ces instructions affichent `a`. Un test est réalisé pour détecter une erreur éventuelle.

## 6 Entrées et sorties

- Sortie
- **Entrée**
- Fichiers
- Fichiers binaires

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

```
int num, ret;  
char chaine[128];  
chaine[0] = '\0';  
ret = scanf("%d W %s",  
           &num, chaine);  
printf("%d %d %s\n",  
       ret, num, chaine);
```

standard un entier, un caractère 'W',  
puis une chaîne de caractères.

Ces instructions lisent sur l'entrée

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

```
int num, ret;  
char chaine[128];  
chaine[0] = '\0';  
ret = scanf("%d W %s",  
           &num, chaine);  
printf("%d %d %s\n",  
       ret, num, chaine);
```

standard un entier, un caractère 'W',  
puis une chaîne de caractères.

```
1 25_W_abc\n  
→ 2 25 abc
```

Ces instructions lisent sur l'entrée

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

```
int num, ret;  
char chaine[128];  
chaine[0] = '\0';  
ret = scanf("%d W %s",  
           &num, chaine);  
printf("%d %d %s\n",  
       ret, num, chaine);
```

standard un entier, un caractère 'W',  
puis une chaîne de caractères.

1 25\_W\_abc\n  
→ 2 25 abc

2 25\_W\_W\_abc\n  
→ 2 25 abc

Ces instructions lisent sur l'entrée

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

```
int num, ret;  
char chaine[128];  
chaine[0] = '\0';  
ret = scanf("%d W %s",  
           &num, chaine);  
printf("%d %d %s\n",  
       ret, num, chaine);
```

Ces instructions lisent sur l'entrée

standard un entier, un caractère 'W',  
puis une chaîne de caractères.

1 25\_W\_abc\n  
→ 2 25 abc

2 25\_W\_W\_abc\n  
→ 2 25 abc

3 25Wabc\n  
→ 2 25 abc

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

```
int num, ret;  
char chaine[128];  
chaine[0] = '\0';  
ret = scanf("%d W %s",  
           &num, chaine);  
printf("%d %d %s\n",  
       ret, num, chaine);
```

Ces instructions lisent sur l'entrée

standard un entier, un caractère 'W',  
puis une chaîne de caractères.

- |   |              |   |          |
|---|--------------|---|----------|
| 1 | 25_W_abc\n   | 4 | 25_abc\n |
|   | → 2 25 abc   |   | → 1 25   |
| 2 | 25_W_W_abc\n |   |          |
|   | → 2 25 abc   |   |          |
| 3 | 25Wabc\n     |   |          |
|   | → 2 25 abc   |   |          |

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

```
int num, ret;
char chaine[128];
chaine[0] = '\0';
ret = scanf("%d W %s",
           &num, chaine);
printf("%d %d %s\n",
       ret, num, chaine);
```

Ces instructions lisent sur l'entrée

standard un entier, un caractère 'W',  
puis une chaîne de caractères.

1 25\_W\_abc\n  
→ 2 25 abc

2 25\_W\_W\_abc\n  
→ 2 25 abc

3 25Wabc\n  
→ 2 25 abc

4 25\_abc\n  
→ 1 25

5 xy\_W\_abc\n  
→ 0 ?

# Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une **lecture formatée** sur l'entrée standard `stdin`.

Cette fonction renvoie le nombre d'éléments lus correctement assignés.

```
int num, ret;  
char chaine[128];  
chaine[0] = '\0';  
ret = scanf("%d W %s",  
           &num, chaine);  
printf("%d %d %s\n",  
       ret, num, chaine);
```

Ces instructions lisent sur l'entrée

standard un entier, un caractère 'W',  
puis une chaîne de caractères.

1 25\_W\_abc\n  
→ 2 25 abc

2 25\_W\_\_abc\n  
→ 2 25 abc

3 25Wabc\n  
→ 2 25 abc

4 25\_abc\n  
→ 1 25

5 xy\_W\_abc\n  
→ 0 ?

6 25\_w\_abc\n  
→ 1 25

## Exemple d'utilisation de scanf

Ce programme lit sur l'entrée standard une chaîne d'au plus sept caractères (format `%Ns`), une espace, puis un entier.

## Exemple d'utilisation de scanf

Ce programme lit sur l'entrée standard une chaîne d'au plus sept caractères (format `%Ns`), une espace, puis un entier.

```
ret = scanf("%7s %d", prenom, &age);
```

## Exemple d'utilisation de scanf

Ce programme lit sur l'entrée standard une chaîne d'au plus sept caractères (format `%Ns`), une espace, puis un entier.

```
ret = scanf("%7s %d", prenom, &age);
while (ret != 2) {
    printf("%lu\n", strlen(prenom));
    ret = scanf("%7s %d", prenom, &age);
}
```

## Exemple d'utilisation de scanf

Ce programme lit sur l'entrée standard une chaîne d'au plus sept caractères (format `%Ns`), une espace, puis un entier.

```
#include <stdio.h>
#include <string.h>

int main() {
    char prenom[8];
    int age, ret;

    ret = scanf("%7s %d", prenom, &age);
    while (ret != 2) {
        printf("%lu\n", strlen(prenom));
        ret = scanf("%7s %d", prenom, &age);
    }
    return 0;
}
```

# Lecture caractère par caractère

La fonction

```
int getchar();
```

permet de lire un caractère sur l'entrée standard.

# Lecture caractère par caractère

La fonction

```
int getchar();
```

permet de lire un caractère sur l'entrée standard.

Cette fonction renvoie le caractère lu (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) lorsque la fin de fichier est détectée (touche `Ctrl + D`).

# Lecture caractère par caractère

La fonction

```
int getchar();
```

permet de lire un caractère sur l'entrée standard.

Cette fonction renvoie le caractère lu (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) lorsque la fin de fichier est détectée (touche Ctrl + D).

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

Ces instructions affichent chaque caractère lu en entrée, tant que `EOF` n'est pas rencontré.

# Le tampon d'entrée

Considérons les instructions

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

# Le tampon d'entrée

Considérons les instructions

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

Il est possible de saisir plusieurs caractères avant le '`\n`' (touche Entrée).

# Le tampon d'entrée

Considérons les instructions

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

Il est possible de saisir plusieurs caractères avant le `'\n'` (touche Entrée).

Avant d'être lus par le `getchar` de la boucle, ils sont situés temporairement dans le **tampon d'entrée**.

# Le tampon d'entrée

Considérons les instructions

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

Il est possible de saisir plusieurs caractères avant le `'\n'` (touche Entrée).

Avant d'être lus par le `getchar` de la boucle, ils sont situés temporairement dans le **tampon d'entrée**.

Le tampon d'entrée se comporte comme un tableau de caractères.

# Le tampon d'entrée

Considérons les instructions

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

Il est possible de saisir plusieurs caractères avant le '`\n`' (touche Entrée).

Avant d'être lus par le `getchar` de la boucle, ils sont situés temporairement dans le **tampon d'entrée**.

Le tampon d'entrée se comporte comme un tableau de caractères.

Chaque appel à `getchar` considère le tampon d'entrée et :

- s'il est vide, on attend la saisie d'un caractère de l'entrée standard ;
- s'il contient au moins un élément, il le considère et le supprime du tampon d'entrée.

- 6 Entrées et sorties
  - Sortie
  - Entrée
  - **Fichiers**
  - Fichiers binaires

# Descripteurs de fichiers et tête de lecture/écriture

Tout fichier est manipulé par un pointeur sur une variable de type `FILE`.

C'est un type structuré déclaré dans `stdio.h` qui contient diverses informations nécessaires et relatives au fichier qu'il adresse.

Toute variable de type `FILE *` est un **descripteur de fichier**.

# Descripteurs de fichiers et tête de lecture/écriture

Tout fichier est manipulé par un pointeur sur une variable de type `FILE`.

C'est un type structuré déclaré dans `stdio.h` qui contient diverses informations nécessaires et relatives au fichier qu'il adresse.

Toute variable de type `FILE *` est un **descripteur de fichier**.

La lecture écriture dans un fichier se fait par l'intermédiaire d'une **tête de lecture**.

Celle-ci désigne un caractère du fichier considéré comme le **caractère observé** à un instant donné.

Les fonctions de lecture/écriture ont pour effet (en particulier) de mettre à jour cette tête de lecture en avançant dans son positionnement dans le fichier.

# Tête de lecture/écriture

Il existe trois fonctions de `stdio.h` pour **déplacer manuellement la tête de lecture** ou obtenir des informations à son sujet :

- `int ftell(FILE *f)`; qui renvoie l'indice de la tête de lecture du fichier pointé par `f`.

Cette fonction est sans effet de bord ;

# Tête de lecture/écriture

Il existe trois fonctions de `stdio.h` pour **déplacer manuellement la tête de lecture** ou obtenir des informations à son sujet :

- `int ftell(FILE *f)`; qui renvoie l'indice de la tête de lecture du fichier pointé par `f`.

Cette fonction est sans effet de bord ;

- `int fseek(FILE *f, int decalage, int mode)`; qui décale la tête de lecture du fichier pointé par `f` de `decalage` caractères selon `mode`.

Ce paramètre explique à quoi le décalage est relatif (`SEEK_SET` : début du fichier, `SEEK_CUR` : position courante, `SEEK_END` : fin du fichier).

Cette fonction renvoie `0` si elle s'est bien exécutée et `-1` sinon ;

# Tête de lecture/écriture

Il existe trois fonctions de `stdio.h` pour **déplacer manuellement la tête de lecture** ou obtenir des informations à son sujet :

- `int ftell(FILE *f)` ; qui renvoie l'indice de la tête de lecture du fichier pointé par `f`.

Cette fonction est sans effet de bord ;

- `int fseek(FILE *f, int decalage, int mode)` ; qui décale la tête de lecture du fichier pointé par `f` de `decalage` caractères selon `mode`.

Ce paramètre explique à quoi le décalage est relatif (`SEEK_SET` : début du fichier, `SEEK_CUR` : position courante, `SEEK_END` : fin du fichier).

Cette fonction renvoie `0` si elle s'est bien exécutée et `-1` sinon ;

- `void rewind(FILE *f)` ; place la tête de lecture au début du fichier. L'appel `void rewind(f)` ; est ainsi équivalent à `int fseek(f, 0, SEEK_SET)` ;.

# Ouverture de fichiers

La fonction

```
FILE *fopen(const char *chemin, const char *mode);
```

permet d'**ouvrir un fichier**.

# Ouverture de fichiers

La fonction

```
FILE *fopen(const char *chemin, const char *mode);
```

permet d'**ouvrir un fichier**.

Cette fonction renvoie un descripteur de fichier sur le fichier de chemin **chemin** (relatif par rapport à l'exécutable).

# Ouverture de fichiers

La fonction

```
FILE *fopen(const char *chemin, const char *mode);
```

permet d'**ouvrir un fichier**.

Cette fonction renvoie un descripteur de fichier sur le fichier de chemin `chemin` (relatif par rapport à l'exécutable).

La tête de lecture est positionnée sur son 1<sup>er</sup> caractère.

# Ouverture de fichiers

La fonction

```
FILE *fopen(const char *chemin, const char *mode);
```

permet d'**ouvrir un fichier**.

Cette fonction renvoie un descripteur de fichier sur le fichier de chemin `chemin` (relatif par rapport à l'exécutable).

La tête de lecture est positionnée sur son 1<sup>er</sup> caractère.

Le paramètre `mode` désigne le mode d'ouverture désiré.

# Ouverture de fichiers

La fonction

```
FILE *fopen(const char *chemin, const char *mode);
```

permet d'**ouvrir un fichier**.

Cette fonction renvoie un descripteur de fichier sur le fichier de chemin **chemin** (relatif par rapport à l'exécutable).

La tête de lecture est positionnée sur son 1<sup>er</sup> caractère.

Le paramètre **mode** désigne le mode d'ouverture désiré.

**Attention** : **fopen** renvoie **NULL** si l'ouverture s'est mal passée. Il faut donc toujours tester la valeur de retour de **fopen**.

# Modes d'ouverture

Il existe plusieurs **modes d'ouverture**. Chacun répond à un besoin particulier :

- **"r"** : lecture seule.
- **"w"** : écriture seule. Si le fichier n'existe pas, il est créé.
- **"a"** : écriture en ajout. Permet d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.
- **"r+"** : lecture et écriture.
- **"w+"** : lecture et écriture avec suppression préalable du contenu du fichier. Si le fichier n'existe pas, il est créé.
- **"a+"** : lecture et écriture en ajout. Permet de lire et d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.

# Fermeture de fichiers

La fonction

```
int fclose(FILE *f);
```

permet de fermer un fichier.

# Fermeture de fichiers

La fonction

```
int fclose(FILE *f);
```

permet de **fermer un fichier**.

Cette fonction permet de mettre à jour le fichier pointé par `f` de sorte que toutes les modifications effectuées soient prises en compte. Le pointeur `f` n'est plus alors utilisable pour accéder au fichier.

# Fermeture de fichiers

La fonction

```
int fclose(FILE *f);
```

permet de **fermer un fichier**.

Cette fonction permet de mettre à jour le fichier pointé par `f` de sorte que toutes les modifications effectuées soient prises en compte. Le pointeur `f` n'est plus alors utilisable pour accéder au fichier.

Cette fonction renvoie `0` si la fermeture s'est bien déroulée et `EOF` dans le cas contraire.

# Fermeture de fichiers

La fonction

```
int fclose(FILE *f);
```

permet de **fermer un fichier**.

Cette fonction permet de mettre à jour le fichier pointé par `f` de sorte que toutes les modifications effectuées soient prises en compte. Le pointeur `f` n'est plus alors utilisable pour accéder au fichier.

Cette fonction renvoie `0` si la fermeture s'est bien déroulée et `EOF` dans le cas contraire.

**Attention** : toute ouverture d'un fichier lors de l'exécution d'un programme doit s'accompagner tôt ou tard de la fermeture future du fichier en question.

# Écriture dans un fichier

La fonction

```
int fprintf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** dans le fichier pointé par `f`.

# Écriture dans un fichier

La fonction

```
int fprintf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** dans le fichier pointé par **f**.

Cette fonction se comporte comme **printf**, à la différence que cette dernière travaille sur le fichier **stdout**.

# Écriture dans un fichier

La fonction

```
int fprintf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** dans le fichier pointé par `f`.

Cette fonction se comporte comme `printf`, à la différence que cette dernière travaille sur le fichier `stdout`.

```
FILE *f;
int ret;
f = fopen("fic.txt", "w");
if (f == NULL)
    exit(EXIT_FAILURE);
fprintf(f, "abc\n");
ret = fclose(f);
```

```
if (ret == EOF)
    exit(EXIT_FAILURE);
```

Ces instructions déclarent un pointeur sur le fichier `fic.txt` et écrivent `"abc"` dans `fic.txt`.

# Lecture dans un fichier

La fonction

```
int fscanf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **lecture formatée** depuis le fichier pointé par `f`.

# Lecture dans un fichier

La fonction

```
int fscanf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **lecture formatée** depuis le fichier pointé par `f`.

Cette fonction se comporte comme `scanf`, à la différence que cette dernière travaille sur le fichier `stdin`.

- 6 Entrées et sorties
  - Sortie
  - Entrée
  - Fichiers
  - Fichiers binaires

# Ouverture de fichiers en mode binaire

On utilise les modes d'ouverture habituels avec un **b** en plus pour signaler l'ouverture en binaire.

- **"rb"** : lecture binaire seule.
- **"wb"** : écriture binaire seule. Si le fichier n'existe pas, il est créé.
- **"ab"** : écriture binaire en ajout. Permet d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.
- **"rb+"** : lecture binaire et écriture binaire .
- **"wb+"** : lecture binaire et écriture binaire avec suppression préalable du contenu du fichier. Si le fichier n'existe pas, il est créé.
- **"ab+"** : lecture binaire et écriture binaire en ajout. Permet de lire et d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.

# Écriture dans un fichier binaire

On utilise la fonction

```
int fwrite(void *ptr, int taille, int nb, FILE *f);
```

pour **écrire** dans un fichier pointé par **f** ouvert en **mode binaire**.

# Écriture dans un fichier binaire

On utilise la fonction

```
int fwrite(void *ptr, int taille, int nb, FILE *f);
```

pour **écrire** dans un fichier pointé par **f** ouvert en **mode binaire**.

Cette fonction écrit les **nb** valeurs de taille **taille** octets pointées par le pointeur **ptr** dans le fichier pointé par **f**.

# Écriture dans un fichier binaire

On utilise la fonction

```
int fwrite(void *ptr, int taille, int nb, FILE *f);
```

pour **écrire** dans un fichier pointé par **f** ouvert en **mode binaire**.

Cette fonction écrit les **nb** valeurs de taille **taille** octets pointées par le pointeur **ptr** dans le fichier pointé par **f**.

Cette fonction renvoie le nombre de valeurs écrites.

## Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

# Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
typedef struct {  
    unsigned char rouge;  
    unsigned char bleu;  
    unsigned char vert;  
} Couleur;
```

# Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    return 0;
}
```

# Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");

    return 0;
}
```

# Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");
    fwrite(&coul, sizeof(Couleur), 1, f);

    return 0;
}
```

# Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");
    fwrite(&coul, sizeof(Couleur), 1, f);
    fclose(f);

    return 0;
}
```