Programmation Avancée - Langage C

Table des matières

1 Gestion de la Mémoire

1.1 Mémoire d'un programme

En C, deux espaces mémoires principaux sont utilisés :
— **Pile (Stack)** : Mémoire à durée limitée, utilisée pour les variables locales et les appels de fonction.
— **Tas (Heap)** : Mémoire disponible pendant l'exécution du programme. La gestion de cette mémoire est à la charge du programmeur (allocation/libération).

1.1.1 Pile (Stack)

— Mémoire réservée pour l'exécution d'une fonction.
— Disparaît une fois la fonction exécutée.
— Les paramètres de fonction sont passés en **copie** (sauf pour les tableaux).
— Utilisée pour les **variables simples**.

1.1.2 Tas (Heap)

— Mémoire allouée pendant l'exécution, persiste jusqu'à libération.
— Seule l'adresse est communiquée, pas la donnée.
— La gestion de la mémoire est plus complexe : **Déclaration, Allocation, Libération**.

1.2 Allocation de mémoire

En C, différentes fonctions sont disponibles pour allouer de la mémoire dynamique :

1.2.1 malloc()

Alloue une zone mémoire :

```
void * malloc(size_t taille_totale);
```

— Renvoie un pointeur sur la zone allouée ou `NULL` en cas d'échec.

1.2.2 calloc()

Similaire à `malloc()` mais initialise à 0.

1.2.3 realloc()

Redimensionne une zone mémoire déjà allouée.

1.3 Connaître la taille d'un type

La fonction `sizeof` permet de déterminer la taille en octets d'un type :

```
size_t sizeof(type);
```

— Fonctionne pour tous les types (int, float, struct, etc.) ainsi que pour les pointeurs.

1.4 Manipulation de pointeurs

Un exemple basique de manipulation de pointeur :

```
int a = 114;
int *b = NULL;
b = &a;
*b = 14;
```

— `b` pointe vers `a`. En modifiant `*b`, la valeur de `a` devient 14.

1.5 Gestion de la mémoire dynamique

1.5.1 Allocation simple

Exemple d'allocation d'un entier sur le tas :

```
int *p = NULL;
p = malloc(sizeof(int));
if(p == NULL) { return -1; }
*p = 25;
```

— `malloc` alloue l'espace pour un entier.
— Il est impératif de vérifier si l'allocation a réussi (via `NULL`).

1.5.2 Libération de mémoire

La fonction `free()` libère la mémoire allouée :

```
void free(void *pointer);
```

— Utiliser pour éviter les fuites de mémoire.
Exemple avec une correction :

```
int *a = malloc(sizeof(int));
int *b = malloc(sizeof(int));
*(a) = 10;
*(b) = 15;
free(a);
a = b;
```

— Il faut libérer la mémoire avant d'écraser le pointeur.

1.6 Tableaux Dynamiques

1.6.1 Tableau à une dimension

Création et utilisation :

```
int taille = 25;
int *T = malloc(taille * sizeof(int));
if(T == NULL) { return -1; }
T[5] = 16;  // Accès via les crochets
*(T + 5) = 16;  // Accès via pointeur
```

1.6.2 Tableau à deux dimensions

Allocation dynamique d'un tableau 2D :

```
int **T = NULL;
T = malloc(5 * sizeof(int *));
for(int i = 0; i < 5; i++) {
    T[i] = malloc(4 * sizeof(int));
}
```

— Chaque ligne est un tableau alloué dynamiquement.

1.6.3 Libération d'un tableau 2D

Pour libérer un tableau à deux dimensions :

```
for(int i = 0; i < 5; i++) {
    free(T[i]);
}
free(T);
T = NULL;
```

1.7 Allocation Dynamique de struct

Exemple avec une structure `coord` :

```
struct coord { int x; int y; };
struct coord *point = malloc(sizeof(struct coord));
point->x = 42;
point->y = 566;
free(point);
```

— Accès aux attributs :
  — Notation avec `*` : `(*point).x = 25;`
  — Notation avec `->` : `point->x = 42;`

2 Entrées-Sorties : Stdio.h

2.1 Manipulation de fichiers

Pour manipuler des fichiers en C, la bibliothèque `stdio.h` propose plusieurs fonctions, regroupées en quatre catégories :

- Ouverture
- Lecture
- Écriture
- Fermeture

2.2 Ouverture de fichier

Avant de manipuler un fichier, il est nécessaire de l'ouvrir à l'aide de la fonction `fopen()`, qui permet d'obtenir un objet de type `FILE *`. Ce pointeur est utilisé pour interagir avec le fichier.

2.2.1 Le type FILE *

Le pointeur `FILE *` permet d'accéder au fichier, en maintenant un index pour la lecture et/ou l'écriture. Il est également possible d'ouvrir un fichier plusieurs fois pour différentes opérations simultanées. Les flux standards associés au type `FILE *` sont :

- `stdin` : entrée standard (terminal/clavier).
- `stdout` : sortie standard (terminal).
- `stderr` : sortie d'erreur (terminal).

Ces flux sont ouverts par défaut pour chaque programme et permettent une gestion unifiée des entrées et sorties, qu'elles proviennent d'un fichier ou d'un terminal.

2.2.2 Ouverture avec `fopen()`

La fonction `fopen()` renvoie un pointeur `FILE *` ou `NULL` en cas d'erreur. Sa signature est :

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

- Le premier paramètre correspond au nom du fichier (avec le chemin, si nécessaire).
- Le second paramètre définit le mode d'ouverture.

2.2.3 Modes d'ouverture

Voici les principaux modes d'ouverture de fichiers :

- `"r"` : Lecture seule.
- `"w"` : Écriture seule (crée ou écrase le fichier).
- `"a"` : Ajout à la fin du fichier.
- `"rb"`, `"wb"`, `"ab"` : Modes en binaire (exécutables, compressés).
- `"r+"` : Lecture/écriture avec curseur au début.
- `"w+"` : Lecture/écriture, création du fichier et remise à zéro de sa taille.
- `"a+"` : Lecture/écriture avec ajout à la fin (crée le fichier si absent).

2.2.4 Échec de `fopen()`

La fonction `fopen()` peut échouer pour plusieurs raisons :
— Le fichier n'existe pas.
— Les droits d'accès ne sont pas compatibles avec le mode choisi.
— Le système a atteint la limite du nombre de fichiers ouverts.
En cas d'échec, `fopen()` renvoie `NULL` et met à jour la variable globale `errno`. Pour afficher l'erreur, la fonction `perror()` peut être utilisée.

2.2.5 Exemple d'ouverture de fichier

```
FILE * fd = fopen("test.txt", "r");
if (fd == NULL) {
    perror("Problème d'ouverture de fichier");
    exit(-1);
}
```

2.3 Fermeture du fichier

Une fois la manipulation du fichier terminée, il est nécessaire de le fermer avec la fonction `fclose()`, qui prend en paramètre le pointeur `FILE *`. La fermeture permet de libérer les ressources associées au fichier. `fclose()` renvoie 0 en cas de succès ou `EOF` en cas d'erreur.

2.4 Écritures dans les fichiers

2.4.1 Fonction `fprintf()`

La fonction `fprintf()` permet d'écrire des données formatées dans un fichier, de manière similaire à `printf()`. Sa signature est :

```
int fprintf(FILE * stream, const char * format, ...);
```

Elle renvoie le nombre d'octets écrits ou une valeur négative en cas d'erreur.

2.4.2 `fputc()`

La fonction `fputc()` permet d'écrire un seul caractère dans un fichier. Sa signature est :

```
int fputc(int char, FILE * stream);
```

Elle avance le curseur d'une position et renvoie le caractère écrit ou `EOF` en cas d'échec.

2.4.3 `fputs()`

La fonction `fputs()` écrit une chaîne de caractères dans un fichier. Sa signature est :

```
int fputs(const char * str, FILE * stream);
```

Elle renvoie le nombre d'octets écrits ou `EOF` en cas d'erreur.

2.4.4 `fwrite()`

La fonction `fwrite()` est utilisée pour écrire des données brutes dans un fichier. Elle ne formate pas les données et est adaptée à l'écriture de buffers ou de tableaux. Sa signature est :

```
size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream);
```

2.5 Lecture depuis un fichier

2.5.1 `fscanf()`

La fonction `fscanf()` se comporte comme `scanf()` mais pour lire à partir d'un fichier. Sa signature est :

```
int fscanf(FILE * stream, const char * format, ...);
```

Elle renvoie le nombre de variables lues avec succès.

2.5.2 `fgetc()`

La fonction `fgetc()` lit un caractère à la fois depuis le fichier. Sa signature est :

```
int fgetc(FILE * stream);
```

Elle renvoie le caractère lu ou `EOF` si la fin du fichier est atteinte ou s'il y a une erreur.

2.5.3 `fgets()`

La fonction `fgets()` permet de lire une chaîne de caractères dans un fichier. Elle prend en paramètres un tableau de caractères déjà alloué, la taille du tableau, et le flux :

```
char * fgets(char * str, int n, FILE * stream);
```

Elle renvoie un pointeur sur la chaîne lue ou `NULL` en cas d'erreur ou de fin de fichier.

2.5.4 `fread()`

La fonction `fread()` lit des données brutes à partir d'un fichier, de la même manière que `fwrite()`. Sa signature est :

```
size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream);
```

2.6 Déplacement dans un fichier

2.6.1 `ftell()`

La fonction `ftell()` permet de connaître la position actuelle du curseur dans un fichier. Sa signature est :

```
long int ftell(FILE * stream);
```

Elle renvoie la position actuelle ou `-1L` en cas d'erreur.

2.6.2 `fseek()`

La fonction `fseek()` permet de déplacer le curseur dans le fichier à une position donnée. Sa signature est :

```
int fseek(FILE * stream, long int offset, int whence);
```

- `SEEK_SET` : Début du fichier.
- `SEEK_CUR` : Position courante du curseur.
- `SEEK_END` : Fin du fichier.

Elle renvoie 0 en cas de succès ou une valeur différente de 0 en cas d'erreur.

2.7 Autres fonctions utiles

2.7.1 `fflush()`

La fonction `fflush()` permet de forcer l'écriture immédiate des données en attente dans un buffer. Sa signature est :

```
int fflush(FILE * stream);
```

Elle renvoie 0 si tout s'est bien passé.

2.7.2 `feof()`

La fonction `feof()` permet de savoir si la fin d'un fichier a été atteinte. Sa signature

3 Entrées-Sorties (suite)

3.1 Manipulation externe de fichiers

La bibliothèque `stdio.h` fournit des fonctions utiles pour manipuler les fichiers, y compris la suppression, le renommage et la création de fichiers temporaires.

3.1.1 Suppression de fichiers

On peut supprimer un fichier (à condition d'avoir les droits nécessaires) à l'aide de la fonction `remove`. Sa signature est la suivante :

```
int remove(const char *nom_fichier);
```

Elle renvoie 0 en cas de succès et -1 en cas d'erreur, cette dernière pouvant être consultée via `perror`.

3.1.2 Renommage de fichier

La fonction `rename` permet de renommer un fichier. Sa signature est la suivante :

```
int rename(const char *old, const char *new);
```

Elle renvoie 0 en cas de succès, et -1 sinon.

3.1.3 Création de fichier temporaire

Il est possible de créer des fichiers temporaires pour stocker des résultats intermédiaires :
— Créer explicitement un fichier avec `fopen` et le supprimer ensuite avec `remove`.
— Utiliser la fonction `tmpfile`, qui crée un fichier temporaire automatiquement supprimé lors de la fermeture du flux avec `fclose`.

Fonction `tmpname` Cette fonction permet de générer un nom unique pour un fichier temporaire. Sa signature est :

```
char * tmpname(char *str);
```

Elle renvoie un pointeur sur une chaîne de caractères contenant le nom de fichier, ou `NULL` en cas de problème.

Fonction `tmpfile` La fonction `tmpfile` crée un fichier temporaire et renvoie un pointeur vers une structure `FILE*`. Ce fichier est automatiquement supprimé lorsque le flux est fermé avec `fclose`.

```
FILE * tmpfile(void);
```

3.2 Fonctions `sscanf`, `sprintf`, et Variantes

Les fonctions de la famille `*scanf` permettent de lire et de convertir des chaînes de caractères en différentes valeurs. Deux fonctions principales ont été vues précédemment :
— `scanf` pour lire depuis le terminal.
— `fscanf` pour lire depuis un fichier.

3.2.1 Lecture depuis une chaîne de caractères : `sscanf`

La fonction `sscanf` permet de lire des valeurs formatées à partir d'une chaîne de caractères. Sa signature est la suivante :

```
int sscanf(const char *str, const char *format, ...);
```

Elle renvoie le nombre d'affectations correctes effectuées.

3.2.2 Écriture dans une chaîne de caractères : `sprintf`

La fonction `sprintf` permet d'écrire des valeurs formatées dans une chaîne de caractères. Sa signature est similaire à celle de `printf` :

```
int sprintf(char *str, const char *format, ...);
```

Elle renvoie le nombre d'octets écrits, ou un nombre négatif en cas d'erreur.

3.3 Fonctions Variadiques

Une fonction variadique est une fonction qui accepte un nombre non déterminé de paramètres. L'exemple le plus célèbre est `printf`, où seul le premier paramètre (le format) est obligatoire.

3.3.1 Syntaxe d'une fonction variadique

Une fonction variadique doit comporter au moins un paramètre fixe, suivi d'une ellipse `...`, comme illustré ci-dessous :

```
int max(int nombre, ...);
```

Dans cet exemple, `nombre` représente le nombre de paramètres qui suivront.

3.3.2 Utilisation de `stdarg.h`

La bibliothèque `stdarg.h` permet de manipuler les paramètres variadiques à l'aide des macros suivantes :

- `va_start` pour initialiser la liste des paramètres.
- `va_arg` pour récupérer les paramètres un par un.
- `va_end` pour fermer la liste des paramètres.

Exemple : Fonction `max`

```
int max(int nombre, ...) {
    int resultat, i, m = 0;
    va_list ap;
    va_start(ap, nombre);
    for (i = 0; i < nombre; i++) {
        resultat = va_arg(ap, int);
        if (resultat > m) {
            m = resultat;
        }
    }
    va_end(ap);
    return m;
}
```

3.4 Fonctions utiles de `stdlib.h`

3.4.1 Fonction `exit`

La fonction `exit` permet de quitter le programme en fermant tous les fichiers ouverts et en retournant un statut au système d'exploitation :

```
void exit(int status);
```

Le paramètre `status` est un entier que l'on peut récupérer dans le terminal à l'aide de la variable `$?`.

Exemple : Utilisation de `exit`

```
void f() {
    exit(1);
}

int main() {
    f();
    return 0;
}
```

Depuis le terminal :

```
$ ./exe
$ echo $?
1
```

3.4.2 Fonction `abort`

La fonction `abort` arrête le programme en cas d'erreur grave. Elle génère un fichier `core dump` qui peut être utilisé pour déboguer.

3.4.3 Fonction `system`

La fonction `system` permet d'exécuter des commandes du système. Elle renvoie 0 en cas de succès ou une valeur différente de 0 en cas d'erreur :

```
int r = system("ls -l");
```

4 Pointeurs de Fonctions et Unions

4.1 Pointeurs de Fonctions

Les pointeurs de fonctions permettent d'apporter de la généricité dans le code en permettant de manipuler des fonctions comme des données.

4.1.1 Utilisation des Pointeurs de Fonctions

Par exemple, si nous disposons d'une fonction qui parcourt un tableau d'entiers pour calculer le maximum, et que nous souhaitons également calculer le minimum, nous devrions réécrire l'intégralité du code. Cependant, en utilisant des pointeurs de fonctions, nous pouvons changer le critère de sélection sans avoir à tout réécrire.

4.1.2 Syntaxe

Pour déclarer un pointeur de fonction, on utilise la syntaxe suivante :

```
int (*ma_fonction)(int, int);
```

Ici, `*ma_fonction` indique qu'il s'agit d'un pointeur de fonction qui prend deux entiers en paramètres.

4.1.3 Utilisation d'un Pointeur de Fonction

Une fois le pointeur déclaré, il peut pointer vers une fonction existante. Par exemple, si nous avons une fonction `max` qui renvoie le maximum entre deux entiers, nous pouvons l'appeler directement :

```
printf("Max(1,2): %d\n", max(1,2));
```

Ou bien en utilisant le pointeur :

```
ma_fonction = &max;
printf("%d\n", ma_fonction(1,2));
```

4.1.4 Remarques

Un pointeur de fonction n'est pas une définition de fonction. La déclaration du pointeur définit son type de retour ainsi que le type et le nombre des paramètres. Toute fonction ayant les mêmes propriétés peut être pointée par ce pointeur. De plus, un paramètre d'une fonction peut également être un pointeur de fonction.

4.1.5 Exemple : Fonction de Comparaison

Considérons une fonction de comparaison :

```
int cmp(int a, int b) {
    if (a < b) return -1;
    if (a == b) return 0;
    return 1;
}
```

Exemple d'Utilisation Nous pouvons avoir une fonction `extract` qui extrait un maximum basé sur un critère de comparaison passé en paramètre :

```
int extract(int *T, int n, int (*comp)(int,int)) {
    int i, m = 0;
    for (i = 1; i < n; i++) {
        if (comp(T[m], T[i]) > 0) {
            m = i;
        }
    }
    return m;
}
```

Exemple d'Utilisation (suite) Voici deux fonctions de comparaison :

```
int Max(int a, int b) {
    return a < b ? +1 : -1;
}
```

```
int Min(int a, int b) {
    return a > b ? +1 : -1;
}
```

Pour extraire les indices des maximum et minimum :

```
int A = extract(T, n, &Max); // Va extraire l'indice du maximum.
int B = extract(T, n, &Min);  // Va extraire l'indice du minimum.
```

4.1.6 Fonction `qsort` de `stdlib`

La bibliothèque `stdlib` fournit une fonction de tri générique qui implémente l'algorithme Quicksort. Sa signature est :

```
void qsort(void *Base, size_t Nmemb, size_t Size, int (*compar)(const void*, const void*));
```

Elle permet de trier n'importe quel tableau.

4.1.7 Utilisation de `qsort`

Pour utiliser `qsort`, vous devez définir une fonction de comparaison :

```
int MaxBis(int *a, int *b) {
    return *a < *b ? +1 : -1;
}
```

Ensuite, vous pouvez l'appeler comme suit :

```
qsort(T, n, sizeof(int), &MaxBis);
```

4.2 Unions

Une union est similaire à une structure, mais tous ses membres partagent la même zone mémoire, ce qui permet de faciliter l'accès à des données de bas niveau.

4.2.1 Déclaration

Une union se déclare de la même manière qu'une structure. Par exemple :

```
union Simple {
    int x;
    float y;
};
```

4.2.2 Instanciation

Voici comment instancier une union :

```
union Simple a;
a.x = 259;
// La valeur a.y est maintenant modifiée, car c'est la même zone mémoire.
```

Dans cet exemple, la taille de l'union `Simple` est de 4 octets (la taille de l'entier ou du flottant).

4.2.3 Unions Complexes

Considérons une union plus complexe :

```
union Complexe {
    double a;  // 8 octets
    int b;     // 4 octets
    short int c; // 2 octets
    char d;    // 1 octet
};
```

La taille de cette union est de 8 octets, en raison de l'attribut de taille maximum (le `double`).

4.2.4 Accès aux Membres

Si l'on souhaite pouvoir accéder à tous les octets de plusieurs manières, on peut utiliser des tableaux et/ou des structures :

```
union X {
    double a;
    struct { int ia; int ib; } b; // structure anonyme
    char d[8];
};
```

Vous pouvez ensuite accéder à ces membres :

```
union X instance;
instance.a = 256000;
instance.b.ia = 999;
instance.d[7] = 89;
```

4.2.5 Limites des Unions

Les unions sont d'un usage limité, souvent utilisées pour des opérations de bas niveau, comme la modification individuelle de bits. Il est essentiel de ne pas les confondre avec les structures.

4.3 Alignements des Données / Padding

Les tailles de données peuvent varier selon l'architecture. Par exemple, considérons une structure contenant deux entiers et deux entiers courts :

```
struct s1 {
    int a;         // 4 octets
    int b;         // 4 octets
    short int c;   // 2 octets
    short int d;   // 2 octets
};
```

La taille de cette structure serait de 12 octets, car il n'y a pas de perte d'espace.

4.3.1 Scénario Alternatif

Pour une autre structure :

```
struct s2 {
    short int c;   // 2 octets
    int a;         // 4 octets
    int b;         // 4 octets
    short int d;   // 2 octets
};
```

Dans ce cas, la taille totale pourrait être de 16 octets, avec 4 octets perdus à cause de l'alignement.

4.3.2 Problème de Sérialisation

La sérialisation d'une structure consiste à l'écrire en mémoire. Par exemple :

```
struct s2 instance;
//...
fwrite(&instance, sizeof(struct s2), 1, fd);
```

Cela écrit l'intégralité de l'instance dans un fichier.

5 Entrées/Sorties : Fonctions printf, scanf et la bibliothèque string.h

5.1 Introduction à stdio.h

Le fichier d'en-tête `stdio.h` est essentiel dans le langage C pour effectuer des opérations d'entrée/sortie. Il contient les déclarations de plusieurs fonctions qui permettent de lire des données depuis le terminal ou des fichiers, ainsi que d'afficher des informations à l'écran.

5.2 La fonction printf

La fonction `printf` est utilisée pour afficher du texte ou des variables sur la sortie standard, généralement la console. Sa syntaxe est la suivante :

```
int printf(const char *format, ...);
```

Le paramètre `format` est une chaîne de caractères qui peut contenir du texte à afficher ainsi que des spécificateurs de format qui indiquent le type de données à afficher. Les spécificateurs de format courants incluent :

- `%d` ou `%i` : Pour afficher un entier signé. - `%u` : Pour afficher un entier non signé. - `%o` : Pour afficher un entier en notation octale. - `%x` : Pour afficher un entier en notation hexadécimale (en minuscules). - `%X` : Pour afficher un entier en notation hexadécimale (en majuscules). - `%f` : Pour afficher un nombre flottant. - `%s` : Pour afficher une chaîne de caractères. - `%c` : Pour afficher un caractère. - `%p` : Pour afficher l'adresse d'un pointeur. - `%%` : Pour afficher un symbole de pourcentage (%).

Par exemple, pour afficher un message simple, on peut utiliser :

```
printf("Bonjour !\n");
```

Pour afficher des variables, on peut procéder comme suit :

```
int i = 10;
char c = 'A';
printf("i vaut %d, c vaut %c\n", i, c);
```

5.3 La fonction scanf

La fonction `scanf` est utilisée pour lire des données saisies par l'utilisateur. Sa syntaxe est la suivante :

```
int scanf(const char *format, ...);
```

Le paramètre `format` contient des spécificateurs qui indiquent le type de données à lire. Par exemple, pour lire un entier, on peut utiliser :

```
int a;
printf("Entrez un entier : ");
scanf("%d", &a);
```

Il est essentiel de vérifier si l'entrée est valide, car une saisie incorrecte peut provoquer un comportement inattendu.

5.4 La bibliothèque string.h

Le fichier d'en-tête `string.h` fournit un ensemble de fonctions utiles pour manipuler des chaînes de caractères et des blocs de mémoire. Voici un aperçu des principales fonctions disponibles :

La fonction `memccpy` permet de copier un bloc de mémoire dans un second bloc en s'arrêtant après la première occurrence d'un caractère. Sa syntaxe est :

```
void *memccpy(void *dest, const void *src, int c, size_t n);
```

La fonction `memchr` recherche la première occurrence d'une valeur dans un bloc de mémoire. Sa syntaxe est :

```
void *memchr(const void *s, int c, size_t n);
```

La fonction `memcmp` compare le contenu de deux blocs de mémoire et renvoie un entier indiquant leur ordre. Sa syntaxe est :

```
int memcmp(const void *s1, const void *s2, size_t n);
```

La fonction `memcpy` copie un bloc de mémoire dans un second bloc, sans considérer les chevauchements. Sa syntaxe est :

```
void *memcpy(void *dest, const void *src, size_t n);
```

La fonction `memmove` permet de copier un bloc de mémoire même si les deux blocs se chevauchent. Sa syntaxe est :

```
void *memmove(void *dest, const void *src, size_t n);
```

La fonction `memset` remplit une zone mémoire avec une valeur précise. Sa syntaxe est :

```
void *memset(void *s, int c, size_t n);
```

La fonction `strcat` ajoute une chaîne de caractères à la suite d'une autre. Sa syntaxe est :

```
char *strcat(char *dest, const char *src);
```

La fonction `strchr` recherche la première occurrence d'un caractère dans une chaîne de caractères. Sa syntaxe est :

```
char *strchr(const char *str, int character);
```

La fonction `strcmp` compare deux chaînes de caractères pour déterminer si elles sont égales, ou si l'une est inférieure ou supérieure à l'autre. Sa syntaxe est :

```
int strcmp(const char *str1, const char *str2);
```

La fonction `strcoll` compare deux chaînes de caractères en tenant compte des règles de localisation. Sa syntaxe est :

```
int strcoll(const char *str1, const char *str2);
```

La fonction `strcpy` permet de copier une chaîne de caractères d'une variable à une autre. Sa syntaxe est :

```
char *strcpy(char *dest, const char *src);
```

La fonction `strcspn` renvoie la longueur de la plus grande sous-chaîne, en partant du début de la chaîne, ne contenant aucun des caractères spécifiés dans une liste de rejet. Sa syntaxe est :

```
size_t strcspn(const char *s, const char *reject);
```

La fonction `strdup` duplique la chaîne de caractères passée en paramètre. Sa syntaxe est :

```
char *strdup(const char *s);
```

La fonction `strerror` renvoie la chaîne de caractères associée à un code d'erreur stocké dans la variable entière `errno`. Sa syntaxe est :

```
char *strerror(int errnum);
```

La fonction `strlen` calcule la longueur d'une chaîne de caractères, sans compter le caractère nul ‘
0‘. Sa syntaxe est :

```
size_t strlen(const char *str);
```

La fonction `strncat` ajoute une chaîne de caractères à la suite d'une autre, en limitant le nombre maximum de caractères copiés. Sa syntaxe est :

```
char *strncat(char *dest, const char *src, size_t n);
```

La fonction `strncmp` compare deux chaînes de caractères, mais uniquement jusqu'à la taille spécifiée. Sa syntaxe est :

```
int strncmp(const char *str1, const char *str2, size_t n);
```

La fonction `strncpy` copie, au plus, les n premiers caractères d'une chaîne de caractères dans une autre. Sa syntaxe est :

```
char *strncpy(char *dest, const char *src, size_t n);
```

La fonction `strndup` duplique au plus n caractères de la chaîne passée en paramètre. Sa syntaxe est :

```
char *strndup(const char *s, size_t n);
```

La fonction `strpbrk` recherche dans une chaîne de caractères la première occurrence d'un caractère parmi une liste de caractères autorisés. Sa syntaxe est :

```
char *strpbrk(const char *s, const char *accept);
```

La fonction `strrchr` recherche la dernière occurrence d'un caractère dans une chaîne de caractères. Sa syntaxe est :

```
char *strrchr(const char *str, int character);
```

La fonction `strspn` renvoie la longueur de la plus grande sous-chaîne ne contenant que des caractères spécifiés dans une liste de caractères acceptés. Sa syntaxe est :

```
size_t strspn(const char *s, const char *accept);
```

La fonction `strstr` recherche la première occurrence d'une sous-chaîne dans une chaîne principale. Sa syntaxe est :

```
char *strstr(const char *haystack, const char *needle);
```

La fonction `strtok` permet d'extraire, un à un, tous les éléments syntaxiques (tokens) d'une chaîne de caractères, en utilisant un ensemble de délimiteurs. Sa syntaxe est :

```
char *strtok(char *str, const char *delim);
```

Enfin, la fonction `strxfrm` transforme les n premiers caractères d'une chaîne source en tenant compte de la localisation en cours, et les place dans la chaîne de destination. Sa syntaxe est :

```
size_t strxfrm(char *dest, const char *src, size_t n);
```

Ces fonctions constituent une base solide pour la manipulation de chaînes et de mémoire en C, offrant une flexibilité et des capacités étendues pour le traitement des données.