

06A : Lire des chaînes et gérer leur stockage

420-3A6-ST : Programmation système

Dany Gagnon

Cégep de Saint-Jean-sur-Richelieu

Mercredi 30 septembre 2026

Une chaîne occupe aussi une case pour sa fin

```
char mot[] = "lac";  
indice      0      1      2      3  
contenu     'l'    'a'    'c'    '\0'
```

La longueur du texte est 3 ; le tableau réserve 4 octets. Un accès doit rester dans les cases du tableau.

Une longueur n'est pas une capacité

```
char mot[12] = "lac";  
printf("%zu\n", strlen(mot)); /* 3 */  
printf("%zu\n", sizeof mot); /* 12 */
```

La capacité détermine ce qu'on peut écrire. L'octet nul détermine où le texte courant se termine.

Lire une adresse, puis l'objet désigné

```
int compteur = 3;  
int *adresse = &compteur;  
*adresse = 4;
```

- `&compteur` obtient son adresse.
- `adresse` contient cette adresse.
- `*adresse` accède à l'entier ; `compteur` vaut ensuite 4.

Un pointeur vers un tableau ne conserve pas sa taille

```
char texte[12] = "lac";  
char *debut = texte;  
printf("%s\n", debut);
```

Le pointeur permet d'accéder au contenu. La capacité doit être conservée ou transmise séparément ; `sizeof debut` ne redonne pas 12.

Regrouper les valeurs qui décrivent le même état

```
typedef struct {  
    int lignes;  
    int caracteres;  
} Bilan;
```

```
Bilan bilan = {0, 0};  
Bilan *ptr = &bilan;  
ptr->lignes++;
```

La structure regroupe deux compteurs. `bilan.lignes` et `ptr->lignes` désignent ici le même membre.

```
ptr->lignes  
// équivaut à :  
(*ptr).lignes
```

Les parenthèses regroupent `*ptr` avant l'accès au membre avec `..`

malloc

```
#include <stdlib.h>

typedef struct {
    int lignes;
    int caracteres;
} Bilan;

int main(void) {
    Bilan* ptr = malloc(sizeof *ptr);

    if (ptr == NULL) {
        return EXIT_FAILURE;
    }

    // malloc n'initialise pas les champs.
    ptr->lignes = 0;
    ptr->caracteres = 0;

    ptr->lignes = 5;

    free(ptr); // Libère la mémoire allouée.
    return 0;
}
```

Deux objets peuvent avoir des durées différentes

Objet	Fin de sa durée de vie
un tableau automatique local	sortie du bloc
un bloc obtenu avec <code>malloc</code>	appel de <code>free</code>

Quitter la fonction ne libère pas automatiquement une allocation. Retourner l'adresse d'un tableau local ne prolonge pas sa durée de vie.

Allouer un bloc avec malloc

« La fonction `malloc()` alloue taille octets et renvoie un pointeur sur la mémoire allouée. »

Manuel Linux, malloc(3), traduction française

Réserver 64 octets :

```
#include <stdlib.h>

/* Dans main : */
char *texte = malloc(64);
if (texte == NULL) {
    return EXIT_FAILURE;
}
texte[0] = '\0'; // Chaîne vide.
/* utiliser le bloc */
free(texte);
```

- La taille est en **octets**. `sizeof *ptr` donne la taille d'un objet pointé.
- Le contenu n'est pas initialisé : écrire les valeurs avant de les lire.
- En cas d'échec, `malloc` retourne `NULL`. Vérifier avant d'utiliser le pointeur.

Allouer, vérifier, utiliser, libérer

```
size_t capacite = 64;
char *texte = malloc(capacite);
if (texte == NULL) {
    perror("malloc");
    return 1;
}
/* écrire avant de lire : le contenu initial est indéterminé */
free(texte);
```

Une allocation doit avoir un propriétaire responsable de sa libération sur chaque chemin de sortie.

Libérer un bloc avec free

« La fonction `free()` libère l'espace mémoire pointé par `ptr` [...] »

Manuel Linux, `free(3)`, traduction française

Quand le bloc n'est plus utile :

```
/* texte provient d'un malloc réussi. */  
free(texte);  
texte = NULL;
```

- Libérer un bloc alloué dynamiquement **une seule fois**, pas un tableau local.
- Après `free`, ne plus lire ni écrire dans ce bloc.
- `free` ne remet pas le pointeur à zéro.
L'affectation `texte = NULL` le fait explicitement ; `free(NULL)` ne fait rien.

Redimensionner un bloc avec realloc

« La fonction `realloc()` modifie la taille du bloc de mémoire pointé par `ptr` à taille octets. »

Manuel Linux, `realloc(3)`, traduction française

Passer de 64 à 128 octets :

```
/* texte pointe déjà vers 64 octets alloués. */
char *nouveau = realloc(texte, 128);
if (nouveau == NULL) {
    perror("realloc");
    free(texte);
    return 1;
}
texte = nouveau;
/* utiliser le bloc agrandi */
free(texte);
```

- La taille est en **octets**. Le bloc peut changer d'adresse.
- Ici, les 64 premiers octets sont conservés ; les nouveaux ne sont pas initialisés.
- En cas d'échec, l'ancien bloc reste valide. Le pointeur temporaire permet de le libérer ou de le réutiliser.

Lire /proc/<pid>/maps

Chaque ligne décrit une région de mémoire virtuelle d'un processus.

```
cat /proc/$$/maps    # $$ : PID du shell courant
```

Extrait d'exemple du manuel :

adresse	droits	offset	dev	inode	chemin
00400000-00452000	r-xp	00000000	08:02	173521	/usr/bin/dbus-daemon
00e03000-00e24000	rw-p	00000000	00:00	0	[heap]
7fffb2c0d000-7fffb2c2e000	rw-p	00000000	00:00	0	[stack]

- **Adresses** : début et fin de la région, en hexadécimal.
- **Droits** : **r** lecture, **w** écriture, **x** exécution ; **p** privé, **s** partagé.
- **Chemin** : programme, bibliothèque, **[heap]** pour le tas ou **[stack]** pour la pile.
- **Autres colonnes** : position dans le fichier (**offset**), périphérique (**dev**), numéro d'inode.

Référence : manuel Linux, `proc_pid_maps(5)`

fgets laisse une place pour l'octet nul

```
char ligne[64];  
if (fgets(ligne, sizeof ligne, stdin) != NULL) {  
    printf("%s", ligne);  
}
```

Une lecture conserve au plus 63 octets de contenu et ajoute l'octet nul. Le saut de ligne est conservé s'il entre dans le tampon. Sinon, le reste attend une prochaine lecture.

Comparer le contenu, pas les adresses

```
if (strcmp(ligne, "stop") == 0) {  
    puts("Arret demande");  
}
```

- `strcmp` compare deux chaînes terminées par un octet nul.
- Le résultat vaut 0 si leur contenu est égal.
- `ligne == "stop"` compare des adresses.

Le saut de ligne est aussi un caractère

```
ligne[strcspn(ligne, "\n")] = '\0';
```

Si la chaîne contient un saut de ligne, cette instruction le remplace par l'octet nul. Sinon, elle réécrit l'octet nul déjà présent.

La chaîne "stopper" ne doit pas être confondue avec "stop".

Conserver la capacité après malloc

```
size_t capacite = 64;
char *ligne = malloc(capacite);
if (ligne == NULL) return 1;
if (fgets(ligne, (int)capacite, stdin) != NULL) {
    printf("%s", ligne);
}
free(ligne);
```

`sizeof ligne` mesure ici le pointeur, pas les 64 octets alloués.

Atelier : lire jusqu'à une commande

- Écrire un programme qui alloue un tampon de 64 octets.
- Lire et afficher les lignes jusqu'à la commande exacte `stop`.
- Libérer le tampon après la commande, après EOF et après une erreur.
- Annoncer le traitement choisi pour une ligne trop longue.

Vérifier quatre chemins distincts

```
printf 'une phrase\nstop\nignore\n' | ./saisie  
printf 'stopper\n' | ./saisie  
printf 'stop' | ./saisie  
./saisie < /dev/null
```

Ajoutez un essai à la capacité maximale et un essai trop long. Expliquez où l'allocation est libérée dans chaque cas.

Un outil peut montrer l'accès qui dépasse

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address \  
    saisie.c -o saisie-asan  
./saisie-asan
```

Tester une frontière et lire le premier diagnostic. Une exécution sans erreur signalée doit aussi être expliquée par les bornes du code.

Relier la forme, l'adresse et la durée

- Quelle capacité contient l'allocation ?
- Le pointeur vise-t-il encore un objet vivant ?
- Où la chaîne reçoit-elle son octet nul ?
- Qui libère le bloc lorsque la lecture se termine ?

À la prochaine séance

Le 7 octobre, les données viendront d'un fichier choisi en argument.

La lecture et ses bornes resteront reconnaissables ; nous ajouterons l'ouverture, le choix du mode et la fermeture du flux.