Des tableaux au tokenizer

Tableaux, mémoire, chaînes, fonctions, structures et allocation dynamique

Programmation en C99 – GNU/Linux

Objectifs et organisation

L'objectif final est de réaliser un *tokenizer* : un programme qui découpe une chaîne en mots *en la modifiant*, puis renvoie des pointeurs sur le début de chaque mot. Avec un objectif principal : ne jamais utiliser plus de mémoire !

Le TP progresse ainsi :

1. tableaux et débordements
2. pile, tas et durées de stockage
3. chaînes de caractères et littéraux
4. boucles `for` et `while`
5. fonctions et arguments
6. structures
7. vue de chaîne et format `%.*s`
8. tokenizer sur un tableau modifiable
9. allocation dynamique, `malloc` et `memcpy`

Jusqu'à la partie sur les fonctions, tous les traitements sont écrits dans `main`. Si vous avez du mal il peut être pertinent de faire des schéma représentant l'état du programme et sa mémoire.

Compilation

```
gcc -std=c99 -Wall -Wextra -Wpedantic -g programme.c -o programme
```

Pour rechercher des erreurs mémoire pendant les tests :

```
gcc -std=c99 -Wall -Wextra -Wpedantic -g -fsanitize=address,undefined -fno-omit-frame
    -pointer programme.c -o programme
```

L'absence de diagnostic d'un outil ne prouve pas qu'un programme est correct. Pour un programme qui a un comportement indéfini, ne déduisez aucune règle à partir d'un affichage observé. Les programmes volontairement incorrects doivent être placés dans des fichiers séparés.

0 Norme C99 et compilation

Rappel

- L'option `-std=c99` demande au compilateur de suivre la norme C99.
- `-Wall -Wextra -Wpedantic` activent des diagnostics supplémentaires.
- Ils ne remplacent pas la lecture attentive du code.
- Trois situations sont à distinguer :
  - une *erreur* : le programme n'est pas produit
  - un *avertissement* : le programme est produit, mais le code est suspect
  - un *comportement indéfini* : la norme n'impose aucun résultat, même si le programme compile.
- La macro `__STDC_VERSION__` renseigne la version de la norme utilisée.

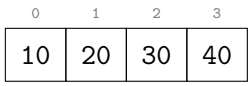
Documentation : Comportement indéfini.

1 Tableaux et débordements

Rappel

- Un tableau contient des éléments contigus de même type
- Pour un tableau de quatre éléments, les indices valides sont `0`, `1`, `2` et `3`
- Le langage ne vérifie pas les bornes de chaque accès.
- Nombre d'éléments d'un tableau déclaré dans le même bloc :

```c
size_t n = sizeof(t) / sizeof(t[0]);
```



`int t[4]` : quatre `int` contigus

Documentation : Tableaux.

Exercice 1. Lire et modifier un tableau

```c
#include <stdio.h>

int main(void)
{
    int t[4] = {10, 20, 30, 40};

    t[1] = t[0] + t[2];
    t[3] = t[1] - t[0];

    printf("%d %d %d %d\n", t[0], t[1], t[2], t[3]);

    return 0;
}
```

a) Donner l'affichage exact avant d'exécuter.
b) Quel est le dernier indice valide ?
c) Que se passerait-il, du point de vue de la norme, avec l'instruction `t[4] = 0;` ?

Exercice 2. Nombre d'éléments et taille en octets

```c
int t[] = {4, 8, 15, 16, 23, 42};
```

a) Afficher `sizeof(t)` et `sizeof(t[0])` avec le format `%zu`.
b) Calculer le nombre d'éléments sans écrire `6`.
c) Ajouter deux valeurs dans l'initialisation : quelle modification du calcul est nécessaire ?
d) Quelle est la différence entre une taille en octets et un nombre d'éléments ?

Exercice 3. Observer un débordement avec AddressSanitizer

```c
#include <stdio.h>

int main(void)
{
    int t[3] = {10, 20, 30};

    for (int i = 0; i <= 3; i++) {
        printf("%d\n", t[i]);
    }

    return 0;
}
```

a) Identifier la première instruction qui accède hors du tableau
b) Compiler avec `-fsanitize=address,undefined` et relever le diagnostic
c) Pourquoi un affichage plausible de la quatrième valeur ne démontre-t-il pas que le programme est valide ?
d) Corriger la boucle

Exercice 4. Parcourir, sommer, chercher

Écrire un programme qui, à partir de :

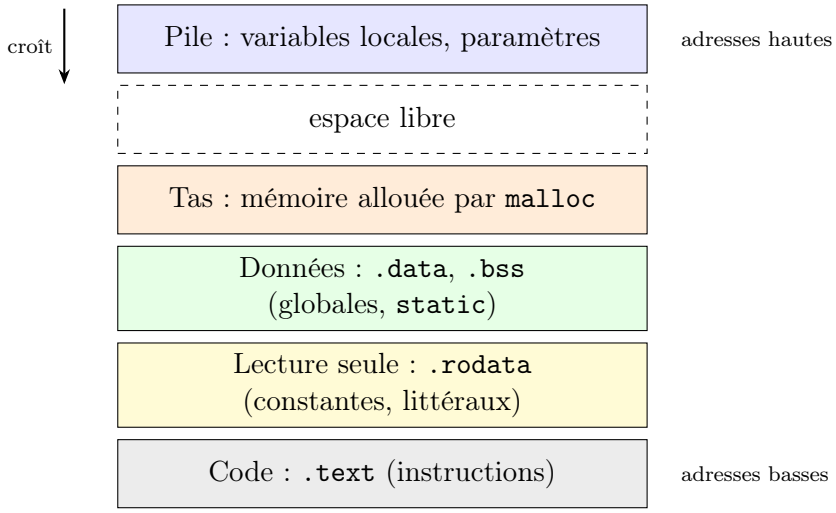
```c
int t[] = {5, -2, 7, -2, 7};
```

a) calcule la somme des éléments
b) détermine le minimum et le maximum
c) affiche l'indice de la première occurrence de chacun
d) compte les occurrences de la valeur `7`
e) explique pourquoi initialiser le minimum à zéro serait incorrect dans le cas général

2 Pile, tas et durées de stockage

Rappel

- Un objet C possède une *durée de stockage* : automatique, statique ou allouée
- Durée *automatique* : variables locales et paramètres, créés à l'entrée du bloc et détruits à sa sortie Dans les implémentations usuelles, ils se trouvent sur la *pile*.
- Durée *statique* : variables globales, variables `static` et objets des littéraux de chaînes Leur durée de vie couvre toute l'exécution (tant que le programme est en vie)
- Durée *allouée* : objets obtenus avec `malloc`, qui vivent jusqu'à `free` Dans les implémentations usuelles, ils se trouvent sur le *tas*.
- Accéder à un objet hors de sa durée de vie est un comportement indéfini
- Le schéma ci-dessous est celui d'une organisation courante. Le C n'impose ni cette disposition, ni ces noms de sections.



Documentation : Durées de stockage.

Documentation : Durée de vie des objets.

Exercice 5. Où se trouvent les objets ?

```c
#include <stdio.h>

int truc = 1;

int main(int argc, char *argv[])
{
    int data = 2;
    static int bidule = 3;
    const char *litteral = "texte";
    char tableau[] = "texte";

    printf("argc     : %p\n", (void *)&argc);
    printf("argv     : %p\n", (void *)argv);
    printf("truc     : %p\n", (void *)&truc);
    printf("bidule : %p\n", (void *)&bidule);
    printf("data      : %p\n", (void *)&data);
    printf("tableau    : %p\n", (void *)tableau);
    printf("litteral   : %p\n", (void *)litteral);

    return 0;
}
```

a) Les valeurs changent-elles d'une exécution à l'autre ?

b) Quels objets ont des adresses proches les unes des autres ?

c) Quels objets ont la durée de stockage automatique ? Statique ?

d) `tableau` et `litteral` contiennent les mêmes caractères. Ont-ils une nature differente ?

e) Pourquoi ne faut-il pas écrire dans un programme une adresse obtenue lors d'une exécution précédente ?

Exercice 6. Variable locale ou variable statique ?

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++) {
        static int n = 0;
        int m = 0;

        n++;
        m++;

        printf("%d %d\n", n, m);
    }

    return 0;
}
```

a) Donner l'affichage exact avant d'exécuter.

b) Pourquoi l'initialisation de `n` n'a-t-elle lieu qu'une seule fois ?
c) Pourquoi `m` recommence-t-il à zéro à chaque tour ?

Exercice 7. Durée de vie d'un bloc

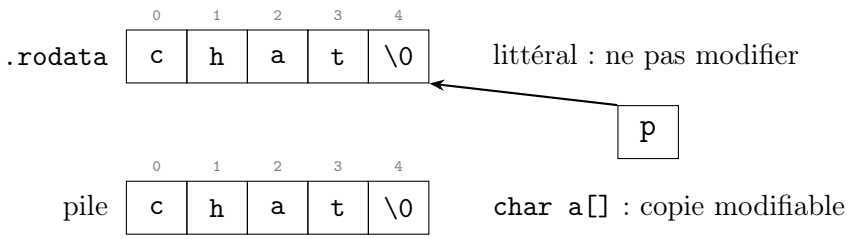
```c
#include <stdio.h>

int main(void)
{
    int *p;

    {
        int x = 5;
        p = &x;
        printf("%d\n", *p);
    }

    printf("%d\n", *p);

    return 0;
}
```

a) Quelle instruction est valide sans condition ?
b) Quelle instruction accède à un objet dont la durée de vie est terminée ?
c) Pourquoi la valeur de `p` reste-t-elle utilisable comme adresse, mais pas comme moyen d'accès à `x` ?
d) Corriger le programme en déplaçant l'utilisation de `p`.

3 Chaînes de caractères et littéraux

Rappel

— Une chaîne C est une suite de caractères terminée par `'\0'`, que `printf("%s", s)` utilise pour savoir où s'arrêter.
— `char a[] = "chat"` crée un tableau de cinq cases, initialisé avec une copie des caractères et du terminateur. Ses éléments sont modifiables.
— `const char *p = "chat"` crée un pointeur vers un littéral. Modifier le contenu du littéral est un comportement indéfini.
— Les littéraux sont souvent placés dans une zone en lecture seule, généralement `.rodata`. Une écriture dans une page protégée provoque souvent une *segmentation fault*, mais la norme ne la garantit pas.



Documentation : Littéraux de chaînes.

Exercice 8. Terminateur et longueurs

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

int main(void)
{
    char a[] = "chat";
    char b[10] = "chat";

    printf("%zu %zu\n", sizeof(a), strlen(a));
    printf("%zu %zu\n", sizeof(b), strlen(b));

    return 0;
}
```

a) Donner l'affichage exact avant d'exécuter

b) Pourquoi `sizeof(a)` et `strlen(a)` diffèrent-ils ?

c) Combien de cases de `b` contiennent le caractère nul ?

d) Pourquoi l'instruction suivante est-elle incorrecte comme chaîne de `printf` ?

```c
char c[4] = {'c', 'h', 'a', 't'};
printf("%s\n", c);
```

Exercice 9. Placer un terminateur au milieu

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

int main(void)
{
    char s[] = "Hello World";

    s[5] = '\0';

    printf("%s\n", s);
    printf("%s\n", s + 6);
    printf("%zu %zu\n", strlen(s), sizeof(s));

    return 0;
}
```

a) Estimer l'affichage avant d'exécuter

b) Les caractères situés après `s[5]` ont-ils été supprimés du tableau ?

c) Pourquoi `sizeof(s)` ne change-t-il pas ?

Exercice 10. Copier un littéral dans un tableau local

Sans fonction de bibliothèque de copie :

```c
const char *source = "Bonjour";
char copie[8];
```

a) Copier les caractères de `source` dans `copie` avec une boucle `while`, terminateur compris.
b) Modifier le premier caractère de `copie`.
c) Afficher `source` et `copie`.
d) Pourquoi le tableau doit-il avoir huit cases ?
e) Que se passerait-il avec un tableau de sept cases ?

Exercice 11. Inspecter l'exécutable

```
readelf -S programme
readelf -p .rodata programme
objdump -d programme
```

a) Rechercher la chaîne `"Bonjour"` dans les données en lecture seule, si elle y est présente.
b) Distinguer les instructions de `.text` et les données de `.rodata`.
c) Pourquoi le langage C ne garantit-il pas le nom d'une section ELF particulière ?
d) Distinguer une règle du langage, un choix de l'outil de compilation et une protection appliquée à l'exécution.

4 Boucles `for` et `while`

Rappel

— Boucle `for` :

```c
for (size_t i = 0; i < n; i++) {
    /* corps */
}
```

— Boucle `while` :

```c
while (condition) {
    /* corps qui fait progresser le traitement */
}
```

— La condition est évaluée avant chaque exécution du corps.
— Un type non signé ne devient jamais négatif : il repart vers sa valeur maximale.

Documentation : Boucle for.

Documentation : Boucle while.

Exercice 12. Parcourir une chaîne

a) Avec une boucle `while` et un indice, afficher chaque caractère de `"Hello"` sans afficher le terminateur.

b) Compter les caractères sans utiliser `strlen`.

c) Réécrire le comptage avec un pointeur `const char *p`.

d) Compter les espaces dans `"un deux  trois"`.

Exercice 13. Compter les mots sans modifier la chaîne

Avec `const char *s = "  un deux   trois ";` :

a) Sauter les espaces initiaux avec une boucle `while`.

b) Compter les mots, un mot étant une suite maximale de caractères différents de l'espace.

c) Tester une chaîne vide, une chaîne d'espaces et une chaîne sans espace.

d) Pourquoi faut-il tester la fin de chaîne avant d'examiner le caractère suivant ?

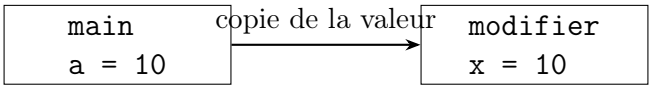
5 Fonctions et arguments

Rappel

— Une fonction possède un type de retour, un nom et des paramètres (pas aucun) :

```c
int carre(int x)
{
    return x * x;
}
```

— Les arguments sont passés par valeur : le paramètre est une copie de l'argument.
— Passer l'adresse d'un objet permet à la fonction de modifier cet objet.



Documentation : Définition de fonctions.

Exercice 14. Définir et appeler une fonction

a) Définir `int affine(int a, int x, int b)` qui renvoie $ax + b$.
b) L'appeler depuis `main` et afficher le résultat.
c) Placer la définition après `main`, sans prototype, et relever le diagnostic en C99.
d) Ajouter le prototype avant `main` c'est a dire sa définition.

Exercice 15. Passage par valeur

```c
#include <stdio.h>

void modifier(int x)
{
    x = 99;
}

int main(void)
{
    int a = 10;

    modifier(a);
    printf("%d\n", a);

    return 0;
}
```

a) Estimer l'affichage avant d'exécuter.
b) L'affectation dans `modifier` change-t-elle `a` ?
c) Réécrire la fonction avec un pointeur pour modifier `a`.

Exercice 16. Échanger deux valeurs

a) Écrire `void swap(int *a, int *b)`.

b) La tester sur deux entiers

Exercice 17. Tableau en argument

```c
#include <stdio.h>

size_t taille(int t[])
{
    return sizeof(t) / sizeof(t[0]);
}

int main(void)
{
    int t[] = {1, 2, 3, 4, 5, 6};

    printf("%zu\n", sizeof(t) / sizeof(t[0]));
    printf("%zu\n", taille(t));

    return 0;
}
```

a) Compiler et lire les avertissements.

b) Pourquoi les deux calculs ne sont-ils pas équivalents ?

c) Quel est le type réel du paramètre `t` ?

d) Corriger la fonction en ajoutant un paramètre de taille.

6 Structures

Rappel

— Une structure regroupe des champs :

```c
typedef struct point
{
    int x;
    int y;
} point;
```

— Accès par une variable : `p.x`.
— Accès par un pointeur : `q->x`, équivalent à `(*q).x`.
— Copier une structure copie ses champs. Si un champ est un pointeur, c'est l'adresse qui est copiée, pas l'objet pointé.

Documentation : Structures.

Exercice 18. Définir et utiliser une structure

a) Définir `point` et créer une variable initialisée avec $(2, 3)$.
b) Afficher ses champs.
c) Créer un pointeur vers la variable et modifier `x` avec l'opérateur `->`.
d) Écrire `void deplacer(point *p, int dx, int dy)`.

Exercice 19. Copier une structure contenant un pointeur

```c
#include <stdio.h>

typedef struct nom { char *texte; size_t taille; } nom;

int main(void)
{
    char s[] = "chat";
    nom a = {s, 4};
    nom b = a;

    b.texte[0] = 'C';
    b.taille = 1;

    printf("%s %zu\n", a.texte, a.taille);
    printf("%s %zu\n", b.texte, b.taille);

    return 0;
}
```

a) La copie `b = a` duplique-t-elle les caractères ?
b) Pourquoi `a.taille` ne change-t-il pas ?
c) Qui est responsable de la durée de vie du tableau `s` ?

7 Vue de chaîne et format %.*s

Rappel

— Il est parfois utile de désigner une portion de chaîne (ou d'un tableau) sans avoir à la copier. Pour cela, on peut utiliser une structure contenant l'adresse du premier caractère et le nombre de caractères concernés :

```c
typedef struct string_view {
    const char *ptr;
    size_t len;
} string_view;
```

— Une telle structure permet de manipuler une portion de chaîne sans en posséder les caractères : elle se contente de référencer les caractères existants. Elle ne les copie pas et ne permet pas de les modifier.

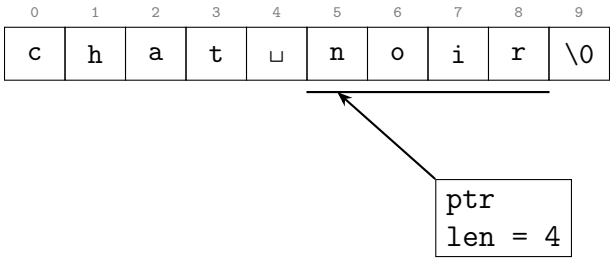
— Pour afficher les `N` premiers caractères référencés par une variable `v`, on peut utiliser :

```c
printf("%.*s\n", (int)v.len, v.ptr);
```

— Dans le format `%.*s`, le caractère `*` indique que la précision doit être fournie comme argument. Cette précision est de type `int` et doit précéder la chaîne dans la liste des arguments.

— L'affichage s'arrête au premier des deux événements suivants : l'affichage de `N` caractères ou la rencontre du caractère nul `'\0'`.

— Enfin, les caractères référencés doivent rester valides pendant toute la durée d'utilisation de la structure. La validité de celle-ci dépend donc de la durée de vie de la chaîne d'origine. C'est a dire sont espace de stockage dois rester valide.



Documentation : printf : largeur et précision.

Exercice 20. Créer et afficher une vue

a) Définir `stringe_view`, que j'écris dans la suite`sv`.

b) Écrire `sv sv_from_cstr(const char *s)` qui renvoie une vue sur toute la chaîne

c) Afficher la vue avec `%.*s`.

d) Afficher une vue vide

Exercice 21. Extraire sans copier

Avec `const char *s = "Hello World";` :

a) Construire une vue sur `"Hello"`

b) Construire une vue sur `"World"` en utilisant `s + 6` et une longueur

c) Afficher les deux vues, puis afficher `s`

d) Écrire `int sv_equals(sv a, sv b)` sans `strcmp` qui permet de comparer deux `sv` et les chaînes associées.

Exercice 22. Durée de vie d'une vue

```
sv fabriquer(void)
{
    char data[] = "bonjour";
    sv v = {data, 3};
    return v;
}
```

a) Pourquoi la vue renvoyée est-elle invalide ?

b) Que contient exactement la structure renvoyée ?

c) Proposer une version valide à partir d'un paramètre `const char *`.

8 Tokenizer sur un tableau modifiable

Principe

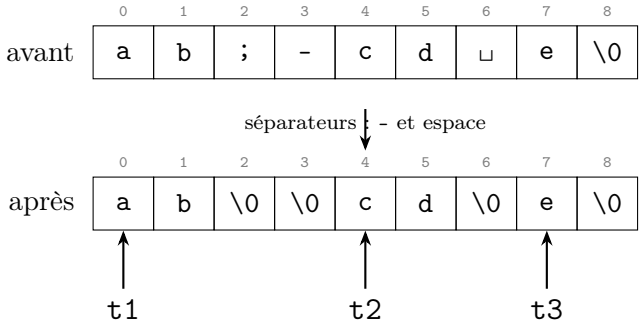
Un *tokenizer* est un programme qui découpe une chaîne de caractères en plusieurs morceaux, appelés *tokens*, en utilisant certains caractères comme séparateurs.

Il existe plusieurs façons d'implémenter un tokenizer. Une approche consiste à créer une nouvelle chaîne de caractères pour chaque token. Cette solution nécessite des allocations mémoire et des copies de caractères, ce qui peut prendre du temps et consommer de la mémoire supplémentaire.

Dans cet exercice, nous allons étudier une autre approche : modifier directement la chaîne d'origine, sans créer de nouvelles chaînes. Le tokenizer remplacera les séparateurs par le caractère nul `'\0'`, puis permettra de parcourir successivement les différents tokens.

Cette approche évite les allocations mémoire et les copies. Elle illustre une idée importante en programmation : il est souvent préférable de travailler directement sur les données existantes plutôt que de multiplier les allocations et les copies.

Chaque token se termine donc par `'\0'` : on l'affiche directement avec `%s`.



Sur cet exemple, les trois tokens sont `"ab"`, `"cd"` et `"e"`.

Résultat attendu

Pour :

```
char data[] = "Hello World!u   ;+ pouet";
char separator[] = " ,;:?./*-+!";
```

le programme doit afficher :

```
Hello
World
u
pouet
```

Chaque caractère de `separator` est un séparateur indépendant. Un token est une suite maximale non vide de caractères qui ne sont pas des séparateurs.

Restrictions

— Ne pas utiliser `strtok`, `strtok_r`, `strsep`, `strchr`, `strpbrk`, `strspn` ni `strcspn`.
— Le découpage modifie le tableau d'origine : aucune copie des tokens n'est effectuée.

— Pas de variable globale pour mémoriser la progression.
— Pas d'allocation dynamique dans cette partie : la structure `token` est une variable locale de `main`.

Squelette

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

typedef struct token
{
    const char *separator;
    char *data;
    size_t size;
} token;

static inline bool isSeparator(const token *tk, char value)
{
    /* A completer. */
    return false;
}

void initializeTokenizer(
    token *tk,
    char *data,
    const char *separator
)
{
    /* A completer. */
}

char *getNextToken(const token *tk, char *current)
{
    /* A completer. */
    return NULL;
}

char *moveToNextToken(char *current)
{
    while (*current != '\0') {
        current++;
    }

    return current;
}

int main(void)
{
    char data[] = "Hello World!u   ;+ pouet";
    char separator[] = " ,;:?./*-+!";

    token tk;
    initializeTokenizer(&tk, data, separator);

    char *current = tk.data;

```

```c
    while ((current = getNextToken(&tk, current)) != NULL) {
        printf("%s\n", current);
        current = moveToNextToken(current);
    }

    return 0;
}
```

Exercice 23. Comprendre le contrat

a) Quel est le rôle de chaque champ de `token` ?
b) Pourquoi `size` mémorise-t-il la longueur *initiale* ?
c) Pourquoi `strlen(tk->data)` ne convient-il plus après le découpage ?
d) Pourquoi `separator` est-il un `const char *` ?
e) Pourquoi `data` doit-il désigner un tableau modifiable ?

Exercice 24. Reconnaître un séparateur

a) Compléter `isSeparator` avec une boucle sur `tk->separator`.
b) Tester un caractère situé au début de la liste, à la fin, puis absent.
c) Que doit renvoyer la fonction si la liste est vide ?
d) Pourquoi `static inline` est-il adapté à une fonction aussi courte, appelée dans une boucle ?
e) Comparer le code obtenu avec `-O0` et `-O2` à l'aide de `gcc -S`.

Exercice 25. Initialiser et découper

a) Mémoriser les deux adresses reçues.
b) Calculer la longueur de `data` avant toute modification.
c) Parcourir exactement `size` caractères.
d) Remplacer chaque séparateur par `'\0'`.
e) Pourquoi faut-il calculer la longueur *avant* le premier remplacement ?

Exercice 26. Trouver le prochain token

Préconditions : `current` appartient au tableau et se situe entre `tk->data` et `tk->data + tk->size`, bornes incluses.

a) Calculer l'adresse de fin des données initiales.
b) Sauter les caractères nuls tant que la fin n'est pas atteinte.
c) Renvoyer `NULL` s'il ne reste aucun token.
d) Sinon, renvoyer l'adresse du premier caractère du token.
e) Expliquer l'ordre des deux tests de la condition de boucle.

Exercice 27. Se placer à la fin du token courant

a) À quel caractère s'arrête `moveToNextToken` ?

b) Pourquoi renvoie-t-elle le terminateur du token, et non le début du suivant ?
c) Pourquoi `getNextToken` peut-elle ensuite repartir de cette adresse ?
d) Que se passerait-il si la boucle de `main` ne contenait pas l'appel à `moveToNextToken` ?

Exercice 28. Tester les cas limites

Utiliser l'espace, le point-virgule et le point d'exclamation comme séparateurs.

Entrée	Cas testé
`""`	Chaîne vide
`"abc"`	Aucun séparateur
`" abc"`	Séparateur initial
`"abc "`	Séparateur final
`"a;;;b"`	Séparateurs consécutifs
`" ;!!; "`	Seulement des séparateurs
`"a b;c!d"`	Séparateurs différents
`"a"`	Token d'un caractère

a) Prévoir les tokens avant chaque test.
b) Tester une liste de séparateurs vide.
c) Tester mille espaces avant un token.
d) Vérifier qu'aucun token vide n'est affiché.
e) Compiler avec AddressSanitizer et relancer les tests.

Exercice 29. Analyser la complexité

On note n la longueur de `data`, m celle de `separator`, ℓ la longueur du token courant et r le nombre de caractères nuls sautés par un appel à `getNextToken`.

a) Donner le meilleur et le pire cas de `isSeparator` en fonction de m.
b) Pourquoi peut-on dire que c'est $O(1)$ quand la liste est fixée et courte, mais pas en général ?
c) Donner le coût de l'initialisation en fonction de n et m.
d) Donner le coût de `getNextToken` en fonction de r.
e) Donner le coût de `moveToNextToken` en fonction de ℓ.
f) Donner le coût total de l'énumération des tokens.
g) Dans l'exemple, la liste contient onze séparateurs. Pourquoi ce nombre borne-t-il le coût de `isSeparator` mais pas celui de `getNextToken` ?
h) Quelle mémoire supplémentaire est utilisée ?

Exercice 30. Lire le tableau après découpage

```
1 char data[] = "a;;bc d";
2 token tk;
3
4 initializeTokenizer(&tk, data, " ;");
```

a) Donner `tk.size`.
b) Donner l'affichage :

```c
printf("%s\n", data);
printf("%s\n", data + 3);
printf("%s\n", data + 6);
```

c) Donner les résultats des appels, sous la forme `data + indice` ou `NULL` :

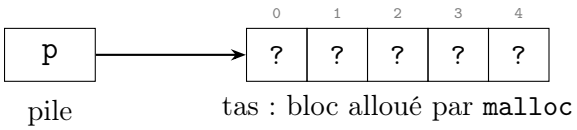
```c
getNextToken(&tk, data);
getNextToken(&tk, data + 1);
getNextToken(&tk, data + 5);
getNextToken(&tk, data + 7);
```

d) Pourquoi le dernier appel, à l'adresse `data + 7`, est-il autorisé par la précondition ?

9 Allocation dynamique, `malloc` et `memcpy`

Rappel

- `malloc(n)` réserve `n` octets sur le tas et renvoie leur adresse, ou `NULL` en cas d'échec.
- La mémoire renvoyée n'est pas initialisée.
- Chaque bloc obtenu doit être libéré une seule fois avec `free`, sauf si le pointeur est `NULL`.
- Après `free`, le pointeur garde sa valeur mais ne désigne plus un objet valide.
- `memcpy(dest, src, n)` copie `n` octets. Les zones ne doivent pas se chevaucher ; sinon, il faut utiliser `memmove`.
- Pour dupliquer une chaîne, il faut copier aussi le terminateur : `strlen(s) + 1` octets.
- Une adresse issue de `malloc` ne doit jamais être donnée à `free` si elle ne vient pas de l'allocateur, par exemple celle d'un tableau local.



Documentation : malloc.

Documentation : free.

Documentation : memcpy.

Exercice 31. Premier bloc alloué

a) Allouer un tableau de dix `int` avec `malloc(10 * sizeof(int))`.
b) Tester le résultat avant toute utilisation.
c) Remplir le tableau avec les carrés des indices et l'afficher.
d) Libérer le bloc.
e) Écrire l'allocation avec `sizeof *p` et expliquer l'intérêt de cette écriture.
f) Reprendre l'exercice sur les adresses en ajoutant l'adresse du bloc alloué.

Exercice 32. Dupliquer une chaîne

```c
const char *source = "Bonjour";
```

a) Calculer la longueur de `source`.
b) Allouer `longueur + 1` octets et tester le résultat.
c) Copier tous les octets nécessaires avec `memcpy`.
d) Modifier le premier caractère de la copie.
e) Afficher `source` et la copie, puis libérer le bloc.
f) Pourquoi le `+ 1` est-il indispensable ?
g) Pourquoi cette copie est-elle modifiable alors que `source` ne l'est pas ?

Exercice 33. Tokeniser une copie allouée

```c
const char *source = "Hello World!u   ;+ pouet";
const char *separator = " ,;:?./*-+!";
```

a) Créer une copie modifiable de `source` avec `malloc` et `memcpy`.
b) Tokeniser cette copie avec les fonctions de la partie précédente.
c) Vérifier que `source` n'a pas changé.
d) Libérer la copie une seule fois.
e) Pourquoi ne pouvait-on pas passer directement `source` au tokenizer ?
f) Dans quel cas cette copie est-elle inutile ?

Exercice 34. Allouer la structure `token`

Reprendre la version d'origine :

```c
token *initializeTokenizer(char *data, const char *separator);
```

a) Allouer la structure, tester le résultat, puis l'initialiser.
b) Que doit renvoyer la fonction en cas d'échec ?
c) Écrire le code de `main` qui gère cet échec.
d) Qui doit libérer la structure ?
e) `free(tk)` libère-t-il aussi `tk->data` ?
f) Quelles libérations sont nécessaires quand la structure et la copie des données ont toutes deux été allouées ? Dans quel ordre les effectuer ?

Exercice 35. Lire des erreurs d'allocation

Les fragments sont indépendants. Ne pas donner de valeur d'exécution aux comportements indéfinis.

```c
/* Fragment A */
char *p = malloc(10);
p = malloc(20);
free(p);

/* Fragment B */
```

```c
char *p = malloc(10);
free(p);
free(p);

/* Fragment C */
char *p = malloc(10);
free(p);
p[0] = 'a';

/* Fragment D */
const char *s = "abc";
char *d = malloc(strlen(s));
memcpy(d, s, strlen(s) + 1);

/* Fragment E */
char *d = malloc(strlen(s) + 1);
memcpy(d, s, strlen(s));
printf("%s\n", d);

/* Fragment F */
char t[] = "abcdef";
memcpy(t + 1, t, 4);

/* Fragment G */
char t[] = "abc";
free(t);
```

a) Identifier le problème de chaque fragment.

b) Distinguer fuite, double libération, accès après libération, dépassement et libération invalide.

c) Proposer une correction pour chaque fragment.

d) Pourquoi mettre un pointeur à `NULL` après `free` peut-il aider, sans suffire à corriger tous les problèmes ?

10 Bilan

Exercice 36. Classer les objets

Pour chaque objet, donner sa durée de stockage et indiquer qui est responsable de sa fin de vie :

```c
int global;
const char *litteral = "abc";
char tableau[] = "abc";
static int compteur;
char *copie = malloc(4);
token tk;
```

a) Répondre pour chaque objet.

b) Lesquels peuvent être modifiés sans comportement indéfini ?

c) Lesquels nécessitent un appel à `free` ?

Exercice 37. Rédiger les contrats du tokenizer

Pour chacune des fonctions `isSeparator`, `initializeTokenizer`, `getNextToken` et `moveToNextToken`, rédiger un contrat précisant :

- les arguments attendus ;
- les objets qui doivent être modifiables ;
- les objets modifiés ;
- les préconditions sur les adresses ;
- la signification du résultat ;
- le comportement en cas d'échec ;
- le responsable de chaque libération éventuelle ;
- la durée de validité des pointeurs renvoyés.

Exercice 38. Synthèse

Répondre en quelques phrases :

a) Pourquoi le tokenizer fonctionne-t-il directement avec un `char []` et sans `malloc` ?

b) Pourquoi la même opération échoue-t-elle avec un littéral ?

c) Pourquoi le terminateur `'\0'` permet-il d'utiliser directement `printf("%s")` sur chaque token ?

d) Pourquoi un pointeur de token devient-il invalide lorsque le tableau qui contient les caractères disparaît ?

e) Quelle différence y a-t-il entre un token renvoyé par le tokenizer et une vue de chaîne ?

Documentation

- Tableaux
- Durées de stockage
- Durée de vie
- Littéraux de chaînes
- Boucle for
- Boucle while
- Fonctions
- Structures
- printf
- strlen
- memcpy
- malloc
- free