1SN_LangageC_C2

January 20, 2021

1 Langage C - Notebook C2

1.1 Allocation dynamique

Katia Jaffrès-Runser, Xavier Crégut Toulouse INP - ENSEEIHT,

1ère année, Dept. Sciences du Numérique, 2020-2021.

1.2 ## 1. Déroulement du cours

Ce cours se déroule sur 6 séances de TP.

- Lors des trois premières séances, vous avez suivi le sujet C1 sous la forme d'un notebook Jupyter.
- Lors des trois dernières séances, vous suivrez deux autres notebook Jupyter, C2 et C3, à votre rythme.

Chaque sujet, C1, C2 et C3, se termine par un exercice Bilan à rendre via votre dépot SVN. Les échéances sont indiquées sur Moodle. Les 3 exercices bilans sont notés, et leur moyenne fournit une note d'exercices.

Vous aurez, en fin de cours, un QCM d'une heure. La note finale est une moyenne des deux notes (QCM et exercices rendus).

1.3 2. Objectifs

Ce cours, sous la forme de notebooks Jupyter et d'un ensemble d'exercices à réaliser en TP, a pour objectif de vous présenter les spécificités de la programmation en langage C. Il se base sur vos acquis du cours de Programmation Impérative en algorithmique et vous détaille les éléments du langage C nécessaires à la production d'un programme en C.

Un support de cours PDF vous est également fournit sur Moodle : Cours C.

1.4 ## 3. Plan du sujet C2.

Ce sujet se focalise sur l'allocation dynamique de mémoire en C. Il vous présente :

- Les principaux allocateurs de mémoire, et leur utilisation.
- La libération de mémoire, et son utilisation.
- La distinction entre manipulation d'un tableau statique et dynamique en C
- Le sous-programme de réallocation de mémoire

1.5 —

1.6 Rappel : Jupyter notebook

Le support de cours que vous lisez est un notebook Jupyter. Pour visualiser le notebook, lancer l'editeur web avec la commande
> `jupyter-notebook`

et rechercher le fichier dans l'arborescence. Le fichier est édité dans votre navigateur Web par défaut. L'enregistrement est automatique (`CTRL S` pour le forcer).

Pour fermer votre fichier, il faut fermer le navigateur et terminer le processus serveur qui s'exécute dans le terminal (`CTRL C`, puis `y`).

> **Important** : - Pour faire fonctionner le kernel C de jupyter notebook, il faut, avant une **première utilisation** de Notebook, lancer la commande suivante dans un `Terminal` :
>
> `install_c_kernel --user`

Ce notebook se compose de cellules présentant soit : - Des éléments de cours, au format Markdown. Ce langage est traduit en HTML pour un affichage aisé quand on clique sur la flèche `Exécuter (run)` et que la cellule est active. - Du code en Langage C (ou Python, ou autre..). Pour compiler et exécuter le code écrit dans la cellule active, on clique sur la flèche `Exécuter (run)`. Si la compilation se déroule sans erreur ni avertissement, le programme est exécuté et les sorties sont affichées en bas de la cellule. Si ce n'est pas le cas, les avertissements et warnings sont affichés en bas de la cellule.

En double-cliquant sur une cellule, on peut éditer son contenu. Vous pouvez ainsi : - Editer une cellule markdown pour y intégrer vos propres notes. - Modifier les programmes pour répondre aux questions et exercices proposés.

Il est possible d'exporter votre travail en PDF, HTML, etc. Il est aussi possible d'afficher les numéros de ligne dans le menu **Affichage**.

Le programme dans la cellule suivante s'exécute sans erreur. Vous pouvez - le tester en l'exécutant. - y introduire une erreur (suppression d'un point-virgule par exemple) pour observer la sortie du compilateur.

[1]:
```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    printf("*******************************\n");
    printf("******** Langage C ***********\n");
    printf("*******************************\n");
    return EXIT_SUCCESS;
}
```

1.7 —

1.8 4. Allocation dynamique de mémoire

1.8.1 4.1 Structure de la mémoire

La mémoire vive de votre ordinateur est structurée en différentes parties : - **Le mémoire statique** Zone de la mémoire où sont stockées les données qui ont la même durée de vie que le programme (variables globales). - **La mémoire automatique** Zone de la mémoire appelée **pile d'exécution** où sont stockés les blocs d'activation, paramètres et variables locales des sous-programmes. Cette mémoire est gérée automatiquement par le compilateur (réservation et libération). La mémoire est contigüe (sans trous). - **La mémoire dynamique** Zone de la mémoire aussi appelée **tas** dans laquelle le programmeur peut explicitement réserver (allouer) de la place. Il devra la libérer explicitement. Cette zone est fragmentée (trous).

Utilisation de la mémoire dynamique L'enregistrement d'une donnée dans une zone de la mémoire dynamique nécessite une **demande d'allocation explicite** de ladite zone. Cette zone mémoire est référencée à traver **un pointeur**. Ainsi, l'écriture et la lecture de la donnée se fait exclusivement par ce pointeur.

Quand la zone mémoire n'est plus utile, il faut **demander explicitement la libération** de l'espace mémoire en C. Attention, quand on réalise cette opération, il faut s'assurer qu'aucun pointeur ne référence plus cette zone mémoire un fois désallouée.

1.8.2 4.2 Allocation de mémoire

La bibliothèque `stdlib.h` (ou `malloc.h`) offre 3 procédures d'allocation de mémoire : `malloc`, `calloc` et `realloc`. Elle définit aussi une procédure de libération de mémoire `free`.

L'allocateur `malloc` C'est l'allocateur utilisé le plus couramment : > `void* malloc(size_t taille);`

Ici on trouve : - Le type de retour `void *` qui représente un type pointeur générique sur une zone mémoire. Le pointeur retourné vaut `NULL` si l'allocation échoue (manque d'espace mémoire contigüe). - Le type `size_t` qui est un alias de `unsigned int` et représente la **taille en octets** de la zone mémoire réservée.

L'opérateur `sizeof`

- Pour obtenir la taille en octets d'une variable ou d'un type, on utilise la fonction `sizeof()` : > `sizeof(ma_variable)` ou `sizeof(type)`

Exemples d'allocations :

```
char* un_char = malloc(sizeof(char));
char* autre_char = malloc(sizeof(*autre_char)); //taille du type pointé

enum genre = {H, F, NC};
enum genre * il = malloc(sizeof(enum genre));
```

L'allocateur `calloc` C'est une variante de malloc : > `void* calloc(size_t nombre, size_t taille_element);`

Ici, la taille de la zone mémoire allouée est décrite avec deux paramètres : - La taille d'un élément avec `taille_element` - Et le nombre d'éléments de cette taille avec `nombre`. Ainsi, on alloue `nombre * taille_element` octets.

> **Note :** À la différence de malloc, tous les bits de la zone allouée sont positionnés à zéro.

Libérer la mémoire avec `free` : Désallouer se fait avec la fonction : > `void free(void* pointeur);`

L'unique paramètre est le pointeur qui désigne l'adresse de la mémoire à désallouer. > **Attention** la libération ne modifie pas l'adresse enregistrée dans le pointeur. Il faut explicitement oublier l'adresse non-valide en initialisant le pointeur à `NULL` :

```
free(ptr_int);
ptr_int = NULL;
```

Voici un exemple d'allocation, utilisation et libération de la mémoire (cf. fichier **`Exemple4_2.c`**) :

[2]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void exemple_dynamique(){
    //Allouer dynamiquement un entier
    unsigned int taille = sizeof(int);
    int *mon_entier = malloc(taille);
    //Vérifier le succès de la demande d'allocation
    assert(mon_entier != NULL);

    //Initialiser la donnée à travers le pointeur mon_entier
    *mon_entier = 10;
    //Accéder à la donnée
    printf("Donnée enregistrée : %d\n", *mon_entier);

    //Libérer la mémoire dynamique
    free(mon_entier);
    //Oublier l'adresse mémoire
    mon_entier = NULL;
}

int main() {
    exemple_dynamique();
    return EXIT_SUCCESS;
}
```

```
Donnée enregistrée : 10
```

On observe les éléments suivants :

- L'allocation est ici réalisée avec la fonction `malloc(taille)` qui retourne un pointeur. Elle demande l'allocation de `taille` octets. Si l'allocation est réalisée avec succès, elle retourne l'adresse de la zone mémoire via le pointeur. Sinon, elle retourne `NULL`.
- L'accès à la donnée est réalisé via le pointeur `mon_entier`
- La libération de la mémoire utilise la fonction `free(mon_entier)`.

  Attention la libération ne modifie pas l'adresse enregistrée dans le pointeur. Il faut explicitement oublier l'adresse non-valide en initialisant le pointeur à `NULL`.

1.8.3 4.3 Allocation d'un tableau avec `malloc`

Pour allouer un tableau de N éléments dynamiquement, il suffit de demander l'espace mémoire pour contenir les N éléments :

```
// Allocation d'un tableau de 10 entiers
int* mon_tableau = malloc(10*sizeof(int));
```

On peut alors utiliser la notation `mon_tableau[..]` pour accéder aux éléments du tableau. En effet, les 10 entiers sont enregistrés dans une portion de mémoire dynamique contigüe. `mon_tableau` est un pointeur qui comporte l'adresse de la première case du tableau, et l'accès à la 3ème case se fait par simple décalage d'indice avec l'opérateur `[]`.

```
mon_tableau[2] = 20;
```

Attention : `sizeof(mon_tableau)` retourne uniquement la taille du pointeur `mon_tableau` alloué dynamiquement ! Par contre, si un tableau est alloué statiquement, `sizeof` retourne la taille totale du tableau.

Exemple Voici un exemple d'allocation de tableaux dynamiques et statiques, et de la valeur retournée par `sizeof`. Vous pouvez manipuler le fichier **`Exemple4_3.c`**.

[5]:
```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

int main(){
    int taille_entier = sizeof(int);
    printf("Taille d'un entier : %d\n", taille_entier); // int Sur 4 bits
    int taille_pointeur = sizeof(int*);
    printf("Taille d'un pointeur : %d\n", taille_pointeur); // Addr Sur 8 bits

    // Allouer un tableau de 10 entiers
    int* mon_tableau = malloc(10*taille_entier);
    int taille_dynamique = sizeof(mon_tableau);
    printf("Taille tableau dynamique : %d\n", taille_dynamique); // Addr ducoup
    assert(taille_dynamique == taille_pointeur);

    // Declarer un tableau statique de 10 entiers
```

```
    int mon_tab[10];
    int taille_statique = sizeof(mon_tab);
    assert(taille_statique == 10*taille_entier);

    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Taille d'un entier : 4
Taille d'un pointeur : 8
Taille tableau dynamique : 8
Bravo ! Tous les tests passent.
```

Conséquence sur la définition d'un type tableau dynamique Il n'est pas possible d'utiliser `sizeof` pour connaitre la taille d'un tableau dynamique.

> **Bonne pratique :** Il convient donc d'enregistrer dans une variable la capacité actuelle du tableau à l'aide d'une enregistrement

```
struct tab {
    int* tableau; //Le tableau, alloué dynamiquement à l'initialisation
    int capacite; //La capacité
};
typedef struct tab tab;
```

1.9 —

1.9.1 Exercice 1a : Manipuler les allocateurs

Cet exercice a pour but de vous faire manipuler l'allocateur `malloc` et de libérer la mémoire avec `free`. Vous pouvez compléter le fichier **`Exercice1a.c`** si besoin.

[25]:
```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

// Consignes pour une obtenir une exécution sans erreur :
//     - compléter les instruction **** TODO ****
// Attention : toutes les variables sont ici allouées et libérées dynamiquent

int main(){

    int* ptr_int; //un entier en mémoire dynamique
    // Allocation et initialisation à la valeur 100;
    ptr_int = malloc(sizeof(int));
    *ptr_int = 100;
```

```
    assert(*ptr_int == 100);


    //**** TODO ****
    //Libérer toute la mémoire dynamique
    ptr_int = NULL;
    free(ptr_int);
    assert(!ptr_int);

    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```

1.9.2 Exercice 1b : Manipuler les allocateurs

Cet exercice a pour but de vous faire manipuler l'allocateur `calloc` et de libérer la mémoire avec `free`. Vous pouvez compléter le fichier **`Exercice1b.c`** si besoin.

[26]:
```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

// Consignes pour une obtenir une exécution sans erreur :
//     - compléter les instruction **** TODO ****
// Attention : toutes les variables sont ici allouées et libérées dynamiquent

int main(){


    float* ptr_float; //un réel en mémoire dynamique
    // Allocation du réel avec CALLOC;
    ptr_float = calloc(1, sizeof(float));
    *ptr_float = 0.0;

    assert(*ptr_float == 0.0);

    //**** TODO ****
    //Libérer toute la mémoire dynamique
    ptr_float = NULL;
    free(ptr_float);
    assert(!ptr_float);

    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```

1.9.3 Exercice 1c : Manipuler les allocateurs

Cet exercice a pour but de vous faire manipuler l'allocateur `calloc` et de libérer la mémoire avec `free`. Vous pouvez compléter le fichier **`Exercice1c.c`** si besoin.

[29]:
```c
#define XXX 1

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

// Consignes pour une obtenir une exécution sans erreur :
//     - Remplacer XXX par le bon résultat dans la suite.
// Attention : toutes les variables sont ici allouées et libérées dynamiquent

int main(){

    enum chat {SIAMOIS, CALICO, PERSAN, TABBY};
    enum chat * my_cat;
    my_cat = calloc(1, sizeof(enum chat));
    *my_cat = XXX;
    assert(*my_cat == XXX);

    my_cat = NULL;
    free(my_cat);
    //Libérer toute la mémoire dynamique

    assert(!my_cat);

    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```

1.9.4 Exercice 1.d : Manipuler les allocateurs

Cet exercice a pour but de vous faire allouer de la mémoire pour enregistrer un tableau de caractères. Vous pouvez completer le fichier **`Exercice1d.c`** si besoin.

Pour rappel, en C, une chaine de caratères est un tableau de caractères qui termine par le caractère `\0`. Cet exercice utilise la bibliothèque `string.h` qui offre des sous-programmes permettant de manipuler des chaines de caractère.

[31]:
```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
```

```c
#include <string.h>

// Consignes pour une obtenir une exécution sans erreur :
//     - compléter les instruction **** TODO ****
// Attention : toutes les variables sont ici allouées et libérées dynamiquent

int main(){
    char* chaine; //une chaine de caractère dynamique
    chaine = calloc(9, sizeof(char));
    // Allocation pour pouvoir y copier la chaine constante "LANGAGE_C"
    // à l'aide de la procédure strcpy() de string.h

    strcpy(chaine, "LANGAGE_C");
    assert(strcmp(chaine, "LANGAGE_C")==0);
    assert(chaine[0] == 'L');
    assert(chaine[9] == '\0');

    //**** TODO ****
    //Libérer toute la mémoire dynamique
    chaine = NULL;
    free(chaine);

    assert(!chaine);

    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```

1.10 —

1.10.1 Exercice 2 : Allocation dynamique et statique d'un tableau.

Dans l'exercice suivant, il faut compléter des sous-programmes permettant d'initialiser et manipuler les structures de données nécessaires à la réalisation d'une version simplifiée du jeu de UNO.

Dans ce jeu, il y a 10 cartes de 4 couleurs différentes (jaune, rouge, vert et bleu), numérotées entre 0 et 9. Une main de 7 cartes est distribuée à 2 joueurs. Le premier joueur à avoir posé toutes ses cartes est le vainqueur. Une carte ne peut être jouée que si elle présente le même numéro OU la même couleur que la précédente. Si un joueur ne peut poser une carte, il doit piocher une carte dans le tas de cartes restantes.

Cet exercice a pour but de vous faire pratiquer la manipulation des tableaux dynamiques et statiques. Les consignes précises sont décrites dans le fichier ci-après. Vous pouvez completer le fichier **`Exercice2.c`** si besoin.

L'objectif de l'exercice est de réaliser une exécution sans erreur du programme de test proposé

(`test_preparer_jeu_UNO`). Ce programme de test permet de vérifier la bonne préparation du jeu, et donc des étapes suivantes : - la création du jeu de 4*10 cartes, - la création de la main des deux joueurs. Chaque main comporte 7 cartes. - la création de la derniere carte posée pour démarrer le jeu.

[55]:
```c
#define XXX 10

// Consignes :
//  1. Remplacer XXX par le bon résultat dans la suite.
//  2. Compléter avec les instructions nécessaires en lieu et place de
//     **** TODO ****

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

#define NB_VALEURS XXX
#define NB_CARTES 4*NB_VALEURS

//Définition du type enseigne
enum couleur {JAUNE, ROUGE, VERT, BLEU};
typedef enum couleur couleur;

//Tableau de caractères représentant les couleurs
char C[4] = {'J', 'R', 'V', 'B'};

//Définition du type carte
struct carte {
    couleur couleur;
    int valeur; //valeur >= 0 && valeur < NB_VALEURS
    bool presente; // la carte est-elle presente dans le jeu ?
};
typedef struct carte carte;

//Définition du type jeu complet pour enregistrer NB_CARTES cartes.
typedef carte jeu[NB_CARTES];

//Définition du type t_main, capable d'enregistrer un nombre variable de cartes.
struct main {
    carte* main; //tableau des cartes dans la main.
    int nb; //monbre de cartes
};
typedef struct main t_main;
```

```
/**
 * \brief Initialiser une carte avec une couleur et une valeur.
 * \param[in] c couleur de la carte
 * \param[in] v valeur de la carte
 * \param[in] ej booléen presente
 * \param[out] la_carte
 */
void init_carte(carte* la_carte, couleur c, int v, bool pr){
    la_carte->couleur = c;
    la_carte->valeur = v;
    la_carte->presente = pr;
}

/**
 * \brief Vérifie si la valeur de la carte est conforme à l'invariant.
 * \param[in] c la carte
 * \return bool vrai si la valeur est conforme, faux sinon.
 */
 bool est_conforme(carte c){
    return (c.valeur>=0 && c.valeur<NB_VALEURS);
}

/**
 * \brief Initialiser une main.
 * \param[in] N nombre de cartes composant la main.  Précondition : N <=␣
↪(NB_CARTES - 1) div 2
 * \param[out] la_main créée
 * \return true si l'initialisation a échouée.
 */
bool init_main(t_main* la_main, int N){
    assert(N <= (NB_CARTES-1)/2);
    // ***** TODO *****
    // Corriger l'initialisation du tableau main
    la_main->main = malloc(N*sizeof(carte));
    la_main->nb = N;
    return (la_main==NULL); //allocation réussie ?
}

/**
 * \brief Initialiser le jeu en ajoutant toutes les cartes possibles au jeu.
 * \brief Chaque carte est alors présente dans le jeu.
 * \param[out] le_jeu tableau de cartes avec les 4 couleurs et NB_VALEURS␣
↪valeurs possibles
 */
void init_jeu(jeu le_jeu){
    int k=0;
    for (int i=0 ; i<4 ; i++){
```

```
        for (int j=0 ; j<NB_VALEURS ; j++){
            init_carte(&(le_jeu[k]), i, j, true);
            k++;
        }
    }
}


/**
 * \brief Copie les valeurs de la carte src dans la carte dest.
 * \param[in] src carte à copier
 * \param[out] dest carte destination de la copie
 */
void copier_carte(carte* dest, carte src){
    dest->couleur = src.couleur;
    dest->valeur = src.valeur;
    dest->presente = src.presente;
}

/**
 * \brief Afficher une carte.
 * \param[in] cte carte à afficher
 */
void afficher_carte(carte cte){
    printf("(%c;%d;%d)\t", C[cte.couleur], cte.valeur, cte.presente);
}


/**
 * \brief Afficher le jeu.
 * \param[in] le_jeu complet
 */
void afficher_jeu(jeu le_jeu){
    // ***** TODO *****
    // Afficher le jeu complet. Les carte sont listées sur une même ligne,
    // et séparées par une tabulation \t
    for (int i = 0; i< NB_CARTES; i++) {
        afficher_carte(le_jeu[i]);
    }
    printf("\n");
}

/**
 * \brief Afficher une main.
 * \param[in] la_main la main a afficher
 */
void afficher_main(t_main la_main){
```

```
    // ***** TODO *****
    // Afficher le jeu complet. Les carte sont listées sur une même ligne,
    // et séparées par une tabulation \t
    for (int i = 0; i< la_main.nb; i++) {
        afficher_carte(la_main.main[i]);
    }
    printf("\n");
}

/**
 * \brief mélange le jeu.
 * \param[in out] le_jeu complet mélangé
 */
void melanger_jeu(jeu le_jeu){
    for (int k=0; k<1000; k++){
        // Choisir deux cartes aléatoirement
        int i = rand()%NB_CARTES;
        int j = rand()%NB_CARTES;
        // Les échanger
        // ***** TODO ****
        carte carte_cop;
        copier_carte(&carte_cop, le_jeu[i]);
        copier_carte(&le_jeu[i], le_jeu[j]);
        copier_carte(&le_jeu[j], carte_cop);
    }
}

/**
 \brief Distribuer N cartes à chacun des deux joueurs, en alternant les joueurs.
 * \param[in out] le_jeu complet.
 *       Si la carte c est distribuée dans une main, c.presente devient faux.
 * \param[in] N nombre de cartes distribuées à chaque joueur.  Précondition : N␣
↪<= (NB_CARTES - 1) div 2
 * \param[out] m1 main du joueur 1.
 * \param[out] m2 main du joueur 2.
 */
void distribuer_mains(jeu le_jeu, int N, t_main* m1, t_main* m2){
    assert(N <= (NB_CARTES-1)/2);

    //Initialiser les mains de N cartes
    bool errA = init_main(m1, N);
    bool errB = init_main(m2, N);
    assert(!errA && !errB);

    //Distribuer les cartes
    for (int i=0; i<N; i++){
        // ajout d'une carte dans la main m1
```

```
        copier_carte(&(m1->main[i]), le_jeu[2*i]);
        // ajout d'une carte dans la main m2
        copier_carte(&(m2->main[i]), le_jeu[2*i+1]);
        //mise à jour de presente à false dans le_jeu
        le_jeu[2*i].presente = false;
        le_jeu[2*i+1].presente = false;
    }
}

/**
 * \brief Initialise le jeu de carte, les mains des joueurs et la carte 'last'.
 * \param[out] le_jeu complet avec les 4 couleurs et 10 valeurs possibles.
 *                Ce jeu est mélangé.
 *                Si la carte est inclue dans une main ou est la derniere carte␣
 ↪jouée,
 *                Alors carte.presente vaut faux.
 * \param[in] N nombre de cartes par main.  Precondition : N <= (NB_CARTES-1)/
 ↪2);
 * \param[out] main_A main du joueur A.
 * \param[out] main_B main du joueur B.
 * \param[out] last la derniere carte jouée par un des joueurs.
 */
int preparer_jeu_UNO(jeu le_jeu, int N, t_main* main_A, t_main* main_B, carte*␣
 ↪last){
    assert(N <= (NB_CARTES-1)/2);

    //Initialiser le générateur de nombres aléatoires
    time_t t;
    srand((unsigned) time(&t));

    //Initialiser le jeu
    init_jeu(le_jeu);

    //Melanger le jeu
    melanger_jeu(le_jeu);

    //Distribuer N cartes à chaque joueur
    distribuer_mains(le_jeu, N, main_A, main_B);

    //Initialiser last avec la (2N+1)-ème carte du jeu.
    copier_carte(last, le_jeu[2*N]);
    le_jeu[2*N].presente = false; //carte n'est plus presente dans le_jeu

    return EXIT_SUCCESS;
}

void test_preparer_jeu_UNO(){
```

```
    //Déclarer un jeu (tableau statique), les deux mains (tableaux dynamiques)␣
 ↪et
    //la carte last.
    jeu le_jeu;
    t_main main_A, main_B;
    carte last;

    //Préparer le jeu, les deux mains de 7 cartes et la carte last
    int retour = preparer_jeu_UNO(le_jeu, 7, &main_A, &main_B, &last);
    printf("\n Le jeu mélangé avec les cartes presentes (c ; v ; p) : \n");
    afficher_jeu(le_jeu);
    printf("\n Les deux mains : \n");
    afficher_main(main_A);
    afficher_main(main_B);
    printf("\n La carte last : ");
    afficher_carte(last);
    printf("\n");

    //Vérifier le jeu et les mains.
    assert(retour == EXIT_SUCCESS);
    assert(main_A.nb == 7 && main_B.nb == 7);
    assert(main_A.main != NULL && main_B.main != NULL);
    assert(est_conforme(main_A.main[0]));
    assert(est_conforme(main_B.main[0]));
    assert(est_conforme(last));

    //Détruire la mémoire allouée dynamiquement
    main_A.main = NULL;
    free(main_A.main);
    main_B.main = NULL;
    free(main_B.main);

    assert(main_A.main == NULL);
    assert(main_B.main == NULL);

}

int main(void) {

    test_preparer_jeu_UNO();

    printf("%s", "\n Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Le jeu mélangé avec les cartes presentes (c ; v ; p) :
```

```
(J;0;0) (J;1;0) (J;2;0) (J;3;0) (J;4;0) (J;5;0) (J;6;0) (J;7;0) (J;8;0) (J;9;0)
(V;0;0) (R;1;0) (R;2;0) (R;3;0) (R;4;0) (R;5;1) (R;6;1) (R;7;1) (R;8;1) (R;9;1)
(R;0;1) (V;1;1) (V;2;1) (V;3;1) (V;4;1) (V;5;1) (V;6;1) (V;7;1) (V;8;1) (V;9;1)
(B;0;1) (B;1;1) (B;2;1) (B;3;1) (B;4;1) (B;5;1) (B;6;1) (B;7;1) (B;8;1) (B;9;1)

 Les deux mains :
(J;0;1) (J;2;1) (J;4;1) (J;6;1) (J;8;1) (V;0;1) (R;2;1)
(J;1;1) (J;3;1) (J;5;1) (J;7;1) (J;9;1) (R;1;1) (R;3;1)

 La carte last : (R;4;1)

 Bravo ! Tous les tests passent.
```

1.11 —

1.11.1 4.4 La réallocation avec `Realloc`

En C, il est possible de réallouer une variable dynamique avec la procédure : > `void* realloc(void* ptr_mem, size_t taille)`

Elle prend en paramètres : - le pointeur `ptr_mem` sur la zone mémoire dont on veut modifier la taille, - la nouvelle `taille` de la zone mémoire.

Elle retourne : - un pointeur sur la zone mémoire allouée. **Si cette réallocation échoue, elle retoune `NULL`**.

> **Note :** La réallocation copie également les données enregistrées dans la zone mémoire initiale vers la nouvelle zone mémoire.

On peut se servir de `realloc` pour : - Augmenter la taille mémoire allouée à l'origine. - Réduire la taille mémoire allouée à l'origine. - Libérer la mémoire. Dans ce cas, le paramètre `taille` vaut 0. Ce comportement est équivalement à `free`. - Allouer une nouvelle zone mémoire. Dans ce cas, le paramètre `ptr_mem` vaut `NULL`. Ce comportement est équivalent à `malloc`.

Voici quelques exemples d'utilisation.

1.11.2 Exemple d'une *mauvaise* utilisation de `realloc`

Vous pouvez manipuler le fichier **`Exemple4_4_mauvaise.c`** si besoin.

[56]:
```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

#define TAILLE 10

int main(){

    // Allouer un tableau de TAILLE entiers
```

```
    int* tableau = malloc(TAILLE*sizeof(int));
    assert(tableau); //allocation réussie ?

    // Initialiser les éléments à 1
    for (int i=0; i<TAILLE; i++){
        tableau[i]=1;
    }

    // Augmenter la taille du tableau pour enregistrer TAILLE entiers␣
↪supplémentaires.
    tableau = realloc(tableau, (TAILLE+TAILLE)*sizeof(int));
    assert(tableau); // ré-allocation réussie ?

    //test des 5 premiers éléments
    assert(tableau[0]==1 && tableau[1]==1 && tableau[2]==1 && tableau[3]==1 &&␣
↪tableau[4]==1);

    for (int i=TAILLE; i<TAILLE+TAILLE; i++){
        tableau[i]=2;
    }

    //test de 5 nouveaux éléments
    assert(tableau[TAILLE]==2 && tableau[TAILLE+1]==2 && tableau[TAILLE+2]==2␣
↪&& tableau[TAILLE+3]==2 && tableau[TAILLE+4]==2);

    printf("%s", "\n Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
 Bravo ! Tous les tests passent.
```

Observation : Ici, l'allocation et la réallocation se sont déroulées avec succès et les données présentes avant la réallocation sont toujours présentes après l'allocation. De plus, l'utilisation de l'espace supplémentaire se fait normalement.

Dans l'exemple suivant, on vous propose d'augmenter la constante pré-processeur `INC` pour observer l'échec de la demande de réallocation. Vous pouvez manipuler le fichier **`Exemple4_4_mauvaise_bis.c`** si besoin.

[61]:
```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

#define TAILLE 1000000
#define INC 1e10
//#define INC 1e15 // Plus grand erreur !
```

```c
int main(){

    // Allouer un tableau de TAILLE entiers.
    int* tableau = malloc(TAILLE*sizeof(int));
    assert(tableau); //allocation réussie ?

    // Initialiser les éléments à 1
    for (int i=0; i<TAILLE; i++){
        tableau[i]=1;
    }

    // Augmenter la taille du tableau pour enregistrer INC entiers␣
 ↪supplémentaires.
    tableau = realloc(tableau, (TAILLE+INC)*sizeof(int));
    assert(tableau);

    // Initialiser l'élément d'indice 0 à 2
    tableau[0]=2;

    printf("%s", "\n Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
 Bravo ! Tous les tests passent.
```

Observations :

N.B. : Pour lire les obsevations, il faut afficher les numéros de ligne du programme. Pour cela, il faut se rendre dans le menu 'Affichage' et sélectionner l'option 'Afficher/Masquer les numéros de ligne'.

Voici les principales observations :

- L'assertion de la ligne 21 n'est pas vérifiée quand `INC` devient trop grand : le pointeur tableau est donc `NULL`.
- Si on commente l'instruction de la ligne 21, et que l'on ré-exécute le programme, on obtient une erreur `Executable exited with code -11`, ce qui correspond à un accès à une adresse mémoire non valide (i.e. un segmentation fault). L'adresse non valide est l'adresse `NULL` car à la ligne 24, on accède au premier élément du tableau. Le premier élément n'est plus disponible ici car la réallocation a échoué, et le pointeur tableau a été mis à `NULL` à la ligne 20 par `realloc`.

********* Ici, on a perdu l'accès aux données présentes dans le tableau d'origine avant réallocation *********

> On observe ici **une double peine** : - La réallocation a échoué, - Les données présentes dans le tableau avant l'appel à `realloc` sont définitivement perdues.

1.11.3 Exemple d'une *bonne* utilisation de `realloc`.

Vous pouvez manipuler le fichier **`Exemple4_4_bonne.c`** si besoin.

[62]:
```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

#define TAILLE 1e6
#define INC 1e14

int main(){

    // Allouer un tableau de TAILLE entiers initialisés à 1.
    int* tableau = malloc(TAILLE*sizeof(int));
    assert(tableau); //allocation réussie ?

    // Initialisation à 1
    for (int i=0; i<TAILLE; i++){
        tableau[i]=1;
    }

    // Augmentater la taille du tableau pour enregistrer 10 entiers.
    int* nouveau = realloc(tableau, (TAILLE+INC)*sizeof(int));
    if (nouveau) {
        //recopie de l'adresse uniquement si succès
        tableau = nouveau;
    }
    assert(tableau[0]==1);

    printf("%s", "\n Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
 Bravo ! Tous les tests passent.
```

Observations :

Un appel convenable à `realloc` est illustré à la ligne 20 : ici, l'éventuel échec d'allocation mettra le pointeur `nouveau` à `NULL`, et non le pointeur `tableau`. Si et seulement si `nouveau` n'est pas `NULL`, on recopie la nouvelle adresse dans le pointeur `tableau`. On évite ainsi la double peine.

1.12 —

1.12.1 Exercice 3 : Ré-allocation et jeu de Uno.

Toujour dans le jeu de Uno, on cherche à tester la procédure qui permet de piocher une carte parmi les cartes qui ne sont pas en jeu. Une carte est en jeu si elle appartient à une des deux mains, ou si elle est la dernière jouée.

Dans le tableau statique `le_jeu` qui contient toutes les cartes du jeu, on référence une carte comme 'présente' si elle n'est pas en jeu. Quand on pioche, on cherche la première carte présente dans `le_jeu` et on l'ajoute à la main courante. La carte piochée du tableau `le_jeu` y est indiquée comme non-présente.

L'objet de cet exercice est d'exécuter les tests de la procédure `carte * piocher(jeu le_jeu, t_main* main)` qui : - retourne un pointeur sur la carte piochée dans le jeu - null si elle aucune carte ne peut être piochée ou si la réallocation de mémoire échoue.

L'essentiel du travail est à réaliser au niveau de cette procédure `piocher`. Vous pouvez manipuler le fichier **`Exercice3.c`** si besoin.

[89]:
```c
// Consignes :
//  1. Compléter avec les instructions requises en lieu et place de *** TODO ***

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

#define NB_VALEURS 10
#define NB_CARTES 4*NB_VALEURS

//Définition du type enseigne
enum couleur {JAUNE, ROUGE, VERT, BLEU};
typedef enum couleur couleur;

//Tableau de caractères représentant les couleurs
char C[4] = {'J', 'R', 'V', 'B'};

//Définition du type carte
struct carte {
    couleur couleur;
    int valeur; // Invariant : valeur >= 0 && valeur < NB_VALEURS
    bool presente; // la carte est-elle presente dans le jeu ?
};
typedef struct carte carte;

//Définition du type jeu complet pour enregistrer NB_CARTES cartes.
typedef carte jeu[NB_CARTES];

//Définition du type t_main, capable d'enregistrer un nombre variable de cartes.
struct main {
```

```
    carte * main; //tableau des cartes dans la main.
    int nb; //monbre de cartes
};
typedef struct main t_main;


/**
 * \brief Initialiser une carte avec une couleur et une valeur.
 * \param[in] c couleur de la carte
 * \param[in] v valeur de la carte
 * \param[in] ej booléen presente
 * \param[out] la_carte
 */
void init_carte(carte* la_carte, couleur c, int v, bool pr){
    la_carte->couleur = c;
    la_carte->valeur = v;
    la_carte->presente = pr;
}

/**
 * \brief Vérifie si la valeur de la carte est conforme à l'invariant.
 * \param[in] c la carte
 * \return bool vrai si la valeur est conforme, faux sinon.
 */
 bool est_conforme(carte c){
    return (c.valeur>=0 && c.valeur<NB_VALEURS);
}

/**
 * \brief Copie les valeurs de la carte src dans la carte dest.
 * \param[in] src carte à copier
 * \param[out] dest carte destination de la copie
 */
void copier_carte(carte* dest, carte src){
    dest->couleur = src.couleur;
    dest->valeur = src.valeur;
    dest->presente = src.presente;
}


/**
 * \brief Afficher une carte.
 * \param[in] cte carte à afficher
 */
void afficher_carte(carte cte){
    printf("(%c;%d;%d)\t", C[cte.couleur],cte.valeur, cte.presente);
}
```

```
/**
 * \brief Initialiser une main.
 * \param[in] N nombre de cartes composant la main.  Précondition : N <=␣
 ↪(NB_CARTES - 1) div 2
 * \param[out] la_main créée
 * \return true si l'initialisation a échouée.
 */
bool init_main(t_main* la_main, int N){
    assert(N <= (NB_CARTES-1)/2);
    la_main->main = malloc(N*sizeof(carte));
    la_main->nb = N;
    return (la_main==NULL); //allocation réussie ?
}

/**
 * \brief Initialiser le jeu en ajoutant toutes les cartes possibles au jeu.
 * \param[out] le_jeu tableau de cartes avec les 4 couleurs et NB_VALEURS␣
 ↪valeurs possibles
 */
void init_jeu(jeu le_jeu){
    int k=0;
    for (int i=0 ; i<4 ; i++){
        for (int j=0 ; j<NB_VALEURS ; j++){
            init_carte(&(le_jeu[k]), i, j, true);
            k++;
        }
    }
}

/**
 * \brief Afficher le jeu.
 * \param[in] le_jeu complet avec les 4 couleurs et 910valeurs possibles
 */
void afficher_jeu(jeu le_jeu){
    for (int k=0; k<NB_CARTES; k++){
        afficher_carte(le_jeu[k]);
    }
    printf("\n");
}

/**
 * \brief Afficher une main.
 * \param[in] la_main la main a afficher
 */
void afficher_main(t_main la_main){
    for (int k=0; k<la_main.nb; k++){
```

```
        afficher_carte(la_main.main[k]);
    }
    printf("\n");
}

/**
 * \brief mélange le jeu.
 * \param[in out] le_jeu complet
 */
void melanger_jeu(jeu le_jeu){
    for (int i=0; i<1000; i++){
        // Choisir deux cartes aléatoirement
        int i = rand()%NB_CARTES;
        int j = rand()%NB_CARTES;
        // Les échanger
        carte mem;
        copier_carte(&mem, le_jeu[i]);
        copier_carte(&(le_jeu[i]), le_jeu[j]);
        copier_carte(&(le_jeu[j]), mem);
    }
}


/**
 \brief Distribuer N cartes à chacun des deux joueurs, en alternant les joueurs.
 * \param[in out] le_jeu complet.
 *       Si la carte c est distribuée dans une main, c.presente devient faux.
 * \param[in] N nombre de cartes distribuées à chaque joueur.  Précondition : N␣
↪<= (NB_CARTES - 1) div 2
 * \param[out] m1 main du joueur 1.
 * \param[out] m2 main du joueur 2.
 */
void distribuer_mains(jeu le_jeu, int N, t_main* m1, t_main* m2){
    assert(N <= (NB_CARTES-1)/2);

    //Initialiser les mains de N cartes
    bool errA = init_main(m1, N);
    bool errB = init_main(m2, N);
    assert(!errA && !errB);

    //Distribuer les cartes
    for (int i=0; i<N; i++){
        // ajout d'une carte dans la main m1
        copier_carte(&(m1->main[i]), le_jeu[2*i]);
        // ajout d'une carte dans la main m2
        copier_carte(&(m2->main[i]), le_jeu[2*i+1]);
        //mise à jour de presente à false dans le_jeu
```

```
        le_jeu[2*i].presente = false;
        le_jeu[2*i+1].presente = false;
    }
}

/**
 * \brief Vérifie si les cartes c1 et c2 on la même couleur et la même valeur.
 * \param[in] c1 carte
 * \param[in] c2 carte
 * \return bool Vrai si les deux cartes ont même valeur et couleur.
*/
bool est_egale(carte c1, carte c2){
    return ((c1.couleur == c2.couleur) && (c1.valeur == c2.valeur));
}

/**
 * \brief Vérifie si la carte c est présente dans la main.
 * \param[in] main main d'un joueur
 * \param[in] c carte
 * \return bool Vrai si la carte est presente dans la main, faux sinon.
*/
bool est_presente_main(t_main main, carte c){
    int i = 0;
    while (i < main.nb && !est_egale(main.main[i], c)) {
        i++;
    }
    return !(i == main.nb);
}



/**
 * \brief Piocher une carte dans le jeu et l'inclure dans la main en paramètre.
 * \param[in out] le_jeu complet avec les 4 couleurs et 10 valeurs possibles.
 *                Ce jeu est mélangé.
 *                Si la carte est inclue dans une main ou est la derniere carte␣
 ↪jouée,
 *                Alors carte.presente vaut faux.
 * \param[in out] main main d'un joueur
 * \return carte * un pointeur sur la carte piochee dans le_jeu en paramètre.
 * Ce pointeur vaut NULL si aucune carte ne peut être piochée ou si␣
 ↪l'allocation de mémoire échoue.
*/
carte* piocher(jeu le_jeu, t_main* main){
    // Recherche une carte presente dans le jeu.
    carte *carte_piochee = le_jeu;
    int i = 0;
    while(i < NB_CARTES && carte_piochee->presente == false){
```

```
        carte_piochee = carte_piochee + 1;
        i++;
    }
    if (i == NB_CARTES) {
        carte_piochee = NULL;
    } else {
        // Inserer la carte dans la main
        //*** TODO *** ;
        // Reallouer la mémoire pour enregistrer une carte de plus dans la main.
        carte* nouv_main = realloc(main->main, (main->nb + 1)*sizeof(carte)) ;
        // Penser à l'echec de la reallocation
        if (nouv_main) {
            // Copier la carte_piochee dans la main
            copier_carte(&nouv_main[main->nb-1], *carte_piochee);
            carte_piochee->presente = false;
            // Indiquer que carte_piochee n'est plus presente dans le_jeu
        }


    }
    return carte_piochee;
}

/**
 * \brief Initialise le jeu de carte, les mains des joueurs et la carte 'last'.
 * \param[out] le_jeu complet avec les 4 couleurs et 10 valeurs possibles.
 *                Ce jeu est mélangé.
 *                Si la carte est inclue dans une main ou est la derniere carte␣
 ↪jouée,
 *                Alors carte.presente vaut faux.
 * \param[in] N nombre de cartes par main.  Precondition : N <= (NB_CARTES-1)/
 ↪2);
 * \param[out] main_A main du joueur A.
 * \param[out] main_B main du joueur B.
 * \param[out] last la derniere carte jouée par un des joueurs.
 */
int preparer_jeu_UNO(jeu le_jeu, int N, t_main* main_A, t_main* main_B, carte*␣
 ↪last){
    assert(N <= (NB_CARTES-1)/2);

    //Initialiser le générateur de nombres aléatoires
    time_t t;
    srand((unsigned) time(&t));

    //Initialiser le jeu
    init_jeu(le_jeu);
```

```
    //Melanger le jeu
    melanger_jeu(le_jeu);

    //Distribuer N cartes à chaque joueur
    distribuer_mains(le_jeu, N, main_A, main_B);

    //Initialiser last avec la (2N+1)-ème carte du jeu.
    copier_carte(last, le_jeu[2*N]);
    le_jeu[2*N].presente = false; //carte n'est plus presente dans le_jeu

    return EXIT_SUCCESS;
}

void test_piocher(){
    jeu le_jeu; // le jeu de cartes
    t_main main_A, main_B; // les deux mains
    carte last; // la derniere carte posee

    //Préparer le jeu, les deux mains de 7 cartes et la carte last
    int retour = preparer_jeu_UNO(le_jeu, 7, &main_A, &main_B, &last);
    printf("\n Les deux mains : \n");
    afficher_main(main_A);
    afficher_main(main_B);

    int mem_taille = main_A.nb;

    //Le joueur A pioche une carte dans le_jeu
    carte *c_piochee = piocher(le_jeu, &main_A);

    // Une carte a-t-elle été piochée ?
    assert(c_piochee);
    assert(c_piochee->presente==false); // absence du jeu ?
    assert(est_presente_main(main_A, *c_piochee));
    assert(main_A.nb = mem_taille + 1);

    // Affichage
    printf("\n\n ***** APRES la pioche : ");
    printf("\n La carte piochee : ");
    afficher_carte(*c_piochee);
    printf("\n La nouvelle main A après pioche : \n");
    afficher_main(main_A);
    printf("\n Le nouveau jeu après pioche : \n");
    afficher_jeu(le_jeu);

    //Détruire la mémoire allouée dynamiquement
    free(main_A.main);
    free(main_B.main);
```

```
    main_A.main = NULL;
    main_B.main = NULL;
}

int main(void) {

    test_piocher();

    printf("%s", "\n Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
 Les deux mains :
(R;0;1) (J;2;1) (J;4;1) (J;6;1) (J;8;1) (B;0;1) (R;2;1)
(J;1;1) (J;3;1) (J;5;1) (J;7;1) (J;9;1) (R;1;1) (R;3;1)


 ***** APRES la pioche :
 La carte piochee : (R;5;0)
 La nouvelle main A après pioche :
(R;0;1) (J;2;1) (J;4;1) (J;6;1) (J;8;1) (B;0;1) (R;5;1) (J;0;0)

 Le nouveau jeu après pioche :
(R;0;0) (J;1;0) (J;2;0) (J;3;0) (J;4;0) (J;5;0) (J;6;0) (J;7;0) (J;8;0) (J;9;0)
(B;0;0) (R;1;0) (R;2;0) (R;3;0) (R;4;0) (R;5;0) (R;6;1) (R;7;1) (R;8;1) (R;9;1)
(V;0;1) (V;1;1) (V;2;1) (V;3;1) (V;4;1) (V;5;1) (V;6;1) (V;7;1) (V;8;1) (V;9;1)
(J;0;1) (B;1;1) (B;2;1) (B;3;1) (B;4;1) (B;5;1) (B;6;1) (B;7;1) (B;8;1) (B;9;1)

 Bravo ! Tous les tests passent.
```

1.13 —

1.14 ## BILAN sur l'allocation dynamique. (à rendre)

1.14.1 String : Chaines de caractères à la Java / C++

L'objectif de ces exercices est de définir un vrai type « chaîne de caractères » appelé String par analogie avec le type correspondant des langages Java ou C++. Outre les opérations de haut niveau disponibles sur ce type, sa caractéristique essentielle est de décharger l'utilisateur de la gestion des problèmes de capacité de la chaîne. Si elle est trop petite pour faire une opération d'ajout, elle est agrandie de manière transparente pour l'utilisateur.

Les opérations disponibles sur une String sont : - **create** : initialiser une variable de type String à partir d'une chaîne de caractères classique (tableau de caractères terminé par le caractère nul) ; - **destroy** : détruire un variable de type String. Elle ne pourra plus être utilisée (sauf à être de nouveau initialisée) ; - **length** : obtenir le nombre de caractères de la chaîne ; - **get** : obtenir le

caractère à la position `i` de la chaîne. Le premier caractère a la position 0 ; - **replace** : remplacer le caractère à la position `i` de la chaîne par un nouveau caractère ; - **add** : ajouter un nouveau caractère à la fin de la chaîne. Sa longueur est donc augmentée de 1 ; - **append** : ajouter une chaîne de caractères à la fin de d'une chaîne ; - **insert** : ajouter un nouveau caractère en position `i` de la chaîne `str`. La longueur de la chaîne est donc augmentée de 1. La valeur de `i` doit être comprise entre 0 et `length(str)`. Si `i` vaut `length(str)`, alors `insert` se comporte comme `add` ; - **delete** : supprimer le caractère à la position `i`. - **substring** : retourne une nouvelle String initialisée avec la partie de la chaîne comprise entre les indices début et fin, début inclu et fin exclu.

Une chaine de caractères sera définie dans ce travail comme un enregistrement d'un tableau de caractères dynamique et d'une taille.

Question

> Dans cet exercice bilan, il faut compléter l'implantation des sous-programmes `create`, `add`, `delete`, `length` et `destroy` de façon à ce que les tests s'exécutent avec succès.

RENDU Le rendu de cet exercice Bilan est attendu dans le fichier **`1SN_LangageC_C2_Bilan.c`** via SVN.

[163]:
```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include <string.h>

// Consignes :
//      - Compléter les instructions pour réaliser les fonctions et
//        procédures de ce fichier de façon à exécuter les tests avec succès.
// Vous pouvez utiliser les sous-programmes de la bibliothèque string.h pour␣
 ↪réaliser
// les principales opérations (copie, recherche, etc.)

struct string
{
    char *str; // tableau de caracteres. Doit se terminer par `\0`.
    int N;     // nombre de caractères, `\0` inclus.
};
typedef struct string string;

/**
 * \brief Initialiser à partir d'une chaîne de caractères classique
 * (tableau de caractères terminé par le caractère nul)
 * \param[out] chaine_dest string initialisé
 * \param[in] chaine_src chaine conventionnelle
 */
void create(string *chaine_dest, char *chaine_src)
{
    int taille = strlen(chaine_src) + 1;
```

```
    chaine_dest->N = taille;
    chaine_dest->str = malloc(taille * sizeof(char));
    if (chaine_dest->str)
    {
        strcpy(chaine_dest->str, chaine_src);
    }
}

/**
 * \brief obtenir le nombre de caractères de la chaîne
 * \param[in] str chaine
 */
int length(string chaine)
{
    // ****** TODO *******
    return chaine.N - 1;
}

/**
 * \brief ajouter un nouveau caractère à la fin de la chaîne. Sa longueur est␣
 ↪donc augmentée de 1.
 * \param[inout] chaine
 * \param[in] c le caractère à ajouter en fin de chaine.
*/
void add(string *chaine, char c)
{
    chaine->N++;
    char *nouvelle_str = realloc(chaine->str, (chaine->N + 1) * sizeof(char));
    if (nouvelle_str)
    {
        chaine->str = nouvelle_str;
        chaine->str[chaine->N - 1] = chaine->str[chaine->N - 2];
        chaine->str[chaine->N - 2] = c;
    }
    nouvelle_str = NULL;
    free(nouvelle_str);
}

/**
 * \brief supprimer le caractère à la position i.
 * \param[inout] chaine
 * \param[in] i position du caractere dans la chaine
 * (attention à la precondition).
*/
void delete (string *chaine, int i)
{
    //UNSE_ ->UNEE_ -> UNE
```

```
    for (int j = i; j <= (chaine->N - 2); j++)
    {
        chaine->str[j] = chaine->str[j + 1];
    }
    chaine->str[chaine->N - 2] = chaine->str[chaine->N - 1];
    char *char_suppr = &(chaine->str[chaine->N - 1]);
    char *nouvelle_str = realloc(chaine->str, (chaine->N - 1) * sizeof(char));
    chaine->N--;
    if (nouvelle_str)
    {
        nouvelle_str = NULL;
        free(nouvelle_str);
    }
}

/**
 * \brief détruire, elle ne pourra plus être utilisée (sauf à être de nouveau␣
 ↪initialisée)
 * \param[in] chaine chaine à détruire
*/
void destroy(string *chaine)
{
    for (int j = 0; j <= (chaine->N - 1); j++)
    {
        chaine->str[j] = '\0';
    }
    chaine->str = NULL;
    free(chaine->str);
}

void test_create()
{
    string ch, ch1, ch2;
    create(&ch, "UN");
    assert(ch.N == 3);
    assert(ch.str[0] == 'U');
    add(&ch, 'S');
    create(&ch1, "DEUX");
    assert(ch1.N == 5);
    assert(ch1.str[4] == '\0');
    create(&ch2, "");
    assert(ch2.N == 1);
    assert(ch2.str[0] == '\0');
    destroy(&ch);
    destroy(&ch1);
    destroy(&ch2);
}
```

```
void test_length()
{
    string ch, ch1;
    create(&ch, "UN");
    assert(strlen("UN") == length(ch));
    create(&ch1, "");
    assert(length(ch1) == strlen(""));
    destroy(&ch);
    destroy(&ch1);
}

void test_add()
{
    string ch1;
    create(&ch1, "TROI");
    add(&ch1, 'S');
    assert(length(ch1) == 5);
    assert(ch1.str[4] == 'S');
    add(&ch1, '+');
    assert(length(ch1) == 6);
    assert(ch1.str[5] == '+');
    destroy(&ch1);
}

void test_delete()
{
    string ch1;
    create(&ch1, "TROIS");
    delete (&ch1, 0); //ROIS
    assert(length(ch1) == 4);
    assert(ch1.str[0] == 'R');
    delete (&ch1, 2); //ROS
    assert(length(ch1) == 3);
    assert(ch1.str[2] == 'S');
    delete (&ch1, 2); //RO
    assert(length(ch1) == 2);
    assert(ch1.str[1] == 'O');
    delete (&ch1, 0); //O
    delete (&ch1, 0); //_
    assert(length(ch1) == 0);

    destroy(&ch1);
}

void test_destroy()
{
```

```
    string ch, ch1;
    create(&ch, "UN");
    destroy(&ch);
    assert(ch.str == NULL);

    create(&ch1, "TROI");
    add(&ch1, 'S');
    destroy(&ch1);
    assert(ch1.str == NULL);
}

int main()
{
    test_create();
    test_length();
    test_add();
    test_delete();
    test_destroy();

    printf("%s", "\n Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```