Exercices - Programmation C

Olivier De Jonckère, Nicolas Le Borgne

1 Environnement

Listing 1: Premier programme

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

int main(void) {
    printf("Welcome␣to␣the␣IG3␣C␣Programming␣Course\n");
    return EXIT_SUCCESS;
}
```

Exercice 1:

- Créer un dossier de travail pour les exercices et ouvrir ce dossier dans VS Code
- Depuis VS Code, créer un nouveau fichier "**welcome.c**"
- Remplir le fichier avec le code du Listing 1
- Depuis le terminal de VS Code, compiler avec "**gcc welcome.c**"
- Exécuter le programme avec "**./a.out**"

Exercice 2:

Changer les options de **gcc** afin que le programme généré s'appelle "**welcome**" à la place de "**./a.out**".

Exercice 3:

Depuis le terminal de VS Code :

- Exécuter "**./welcome**", puis exécuter "**echo $?**" et noter la valeur affichée
- Remplacer "**EXIT_SUCCESS**" par "**23**" dans le code
- Recompiler et exécuter le nouveau programme
- Exécuter "**echo $?**" et noter la valeur affichée

2 Variables, Types, Mémoire, Instructions

Pour chaque exercice, créer un nouveau fichier C.

Exercice 4:

À l'aide des fonctions **printf** et **sizeof**, déterminer la taille des types suivants sur votre machine : **char**, **int**, **unsigned int**, **short**, **long**, **float** et **double**.

Exercice 5:

L'objectif de cet exercice est d'apprendre à explorer la documentation. Pour rappel, la documentation des fonctions de la bibliothèque standard C est disponible directement dans le terminal via la commande "**man 3 nom_de_la_fonction**". Pensez à ouvrir un terminal en dehors de VS Code pour avoir plus d'espace visuel. Une documentation est également disponible en ligne ici : `https://en.cppreference.com/w/c.html`. Vous pouvez utiliser la barre de recherche du site pour accéder directement à une fonction en particulier.

Compléter le code du programme suivant avec **printf** et les variables déclarées au début de la fonction *main* pour obtenir l'affichage présenté plus bas.

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

int main() {
  int a = 7;
  int b = -28;
  int c = 134;

  double pi = M_PI;
  double d = 299792458.0;

  char e = 'e';

  /* Completer la suite */


  return 0;
}
```

Le programme doit afficher :

```
a = 7
a =   7
a = 007
b =  -28
c = +134
pi = 3.1416
pi = 3.1415926535897931
pi = 3.14159265358979311600
d = 2.997925e+08
d = 2.9979e+08
d = 2.9979245800e+08
d =      +2.9979e+08
e = e
e = 101
e = 0x65
```

3 Blocs, Fonctions, Opérateurs, Conditions

Pour chaque exercice, créer un nouveau fichier C.

Exercice 6:

On souhaite écrire un programme permettant de calculer l'addition, la soustraction, la multiplication et la division de deux nombres entiers. L'utilisateur commence par entrer le premier nombre, puis l'opérateur désiré parmi '+', '-', '*' et '/', et enfin le second nombre. Le programme affiche le calcul et son résultat.

Exemple d'exécution du programme :

```
# ./calculatrice
12
*
3
12 * 3 = 36
```

Compléter le programme suivant:

Listing 2: calculatrice.c

```c
#include <stdio.h>
int main()
{
    int a;
    int b;
    char op;
    scanf("%d", &a);
    scanf("␣%c", &op);
    scanf("%d", &b);

}
```

Exercice 7:

Écrire un programme permettant de prendre 2 entiers sur l'entrée standard (avec scanf) et d'afficher le plus grand des deux. Le code source doit contenir une fonction responsable de retourner le plus grand nombre.

Exercice 8:

Écrire un programme permettant de prendre 2 entiers sur l'entrée standard (avec scanf). Le code source doit présenter une fonction permettant d'afficher si le premier entier est divisible par le second.

Exercice 9:

Écrire une fonction **maybe_init** qui prend un booléen **init** (un entier) en argument et ne renvoie rien. La fonction commence par déclarer 3 entiers **a**, **b** et **c** mais ne les initialise pas. Ensuite, si **init** est vrai, la fonction assigne les valeurs 42, 2077, et 2029 à **a**, **b** et **c**. Avant le retour, et peu importe la valeur de **init**, la fonction affiche **a**, **b** et **c**. Dans la fonction main, faire 3 appels successifs à **maybe_init** avec comme arguments faux, vrai, puis faux pour la valeur d'**init**, dans cet ordre.

Que se passe-t-il ?

4 Tableaux, Boucles, Chaînes de caractères

Pour chaque exercice, créer un nouveau fichier C.

Exercice 10:

Écrire un programme qui prend un entier sur l'entrée standard (scanf), et affiche tous les entiers entre 1 et 100 divisibles par cet entier.

Exercice 11:

Écrire un programme qui affiche les tables de multiplication de 1 à 9 sous la forme suivante:

```
1x1=1 1x2=2 1x3=3 1x4=4 1x5=5 1x6=6 1x7=7 1x8=8 1x9=9
2x1=2 2x2=4 2x3=6 2x4=8 2x5=10 2x6=12 2x7=14 2x8=16 2x9=18
3x1=3 3x2=6 3x3=9 3x4=12 3x5=15 3x6=18 3x7=21 3x8=24 3x9=27
4x1=4 4x2=8 4x3=12 4x4=16 4x5=20 4x6=24 4x7=28 4x8=32 4x9=36
5x1=5 5x2=10 5x3=15 5x4=20 5x5=25 5x6=30 5x7=35 5x8=40 5x9=45
6x1=6 6x2=12 6x3=18 6x4=24 6x5=30 6x6=36 6x7=42 6x8=48 6x9=54
7x1=7 7x2=14 7x3=21 7x4=28 7x5=35 7x6=42 7x7=49 7x8=56 7x9=63
8x1=8 8x2=16 8x3=24 8x4=32 8x5=40 8x6=48 8x7=56 8x8=64 8x9=72
9x1=9 9x2=18 9x3=27 9x4=36 9x5=45 9x6=54 9x7=63 9x8=72 9x9=81
```

Exercice 12:

La conjecture de Syracuse, également appelée conjecture 3n+1 ou problème de Collatz, énonce que pour tout nombre entier positif n, la séquence définie par la règle suivante finit toujours par atteindre 1 : si n est pair, on le divise par 2 ; si n est impair, on le multiplie par 3 et on ajoute 1.

1. Écrire un programme qui calcule la séquence à partir d'un nombre fournit par l'entrée standard jusqu'à atteindre 1 et affiche le temps de vol (le nombre d'itérations) et l'altitude maximale (la plus grande valeur atteinte).
2. Écrire un programme qui accepte deux nombres sur l'entrée standard : le premier nombre de la séquence et un temps de vol. Le programme doit afficher l'altitude à la fin du temps de vol. Le code source doit contenir une fonction qui retourne l'altitude pour la dernière itération ou -1 si la séquence se termine avant la fin du temps de vol.

Exercice 13:

Dans la fonction main, initialiser un tableau de 10 entiers :

```
int  tab [10] = { 1, 43, 2, 23,  -23, 123, 0, 12,  -4, 7 }
```

Puis créer 3 fonctions. **Attention**, les fonctions doivent fonctionner peu importe la taille du tableau en argument (qu'on admet non vide):

- **print** pour afficher le tableau en ligne (avec un saut de ligne à la fin).
- **add_val** pour ajouter une valeur unique prise en argument à chaque élément du tableau.
- **max** pour rendre la plus grande valeur du tableau.

Aide : vous pouvez utiliser la syntaxe `int arg[]` pour passer un tableau en argument d'une fonction.

Faire ensuite un appel à **print**, un appel à **add_val**, un nouvel appel à **print**, puis récupérer la valeur max et l'afficher.

Exercice 14:

Sur Moodle, télécharger les fichiers **simple_words.c** et **simple_words.in**. Compiler avec `gcc simple_words.c -o simple_words`. Exécuter `./simple_words < simple_words.in`. Le programme devrait afficher une liste de mot.

1. Écrire une fonction **int count_letters(char word[128])** qui retourne le nombre de lettres dans la variable **word**, sans utiliser *strlen*. Afficher le nombre de lettres de chaque mot dans la boucle principale de la fonction **main**.
2. Écrire une fonction **int count_letter_e(char word[128])** qui retourne le nombre d'occurences de la lettre 'e' dans la variable **word**. Afficher le nombre d'occurences de 'e' pour chaque mot dans la boucle principale.
3. Modifier la fonction **int count_letter_e(char word[128])** en ajoutant un second argument afin de compter n'importe quelle lettre. Grâce à cette variante, afficher le nombre de 'a' et de 'i' pour chaque mot.

Exercice 15:

Écrire un programme qui affiche le contenu de la table ASCII (128 éléments) sur 8 lignes en alignant correctement les colonnes. Les 32 premiers éléments ainsi que le dernier élément sont des codes spéciaux qui ne sont pas affichables sous forme de caractère. On affichera leur valeur hexadécimale (par exemple "0x0a") grâce au format *printf* "0x%02x". Pour les autres on affichera leur forme lisible.

Bon à savoir : la valeur décimale d'une variable de type **char** correspond à son code ASCII et on peut effectuer des opérations mathématiques sur **char** comme sur n'importe quel nombre.

Exercice 16:

Sur Moodle, télécharger le fichier **matrix_product.c**.

1. Compléter la fonction **print_matrix** pour afficher la matrice 3x3
2. Compléter la fonction **matmul_vec** pour calculer et afficher le produit matriciel entre la matrice *mat* et le vecteur *vec* en arguments
3. *(bonus)* Compléter la fonction **matmul_mat** pour calculer et afficher le produit matriciel entre deux matrices passées en arguments

Note pour les prochains exercices : Il est possible de lire les arguments passés en paramètre du programme au moment où il est lancé depuis le terminal. Par exemple `./mon_programme califourchon` démarre le programme **mon_programme** avec l'argument **califourchon**. En C, on peut accéder à ces arguments de la façon suivante :

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


int main(int argc, char *argv[])
{
    char * str;
    if (argc < 2) {
        printf("Usage: $ %s <some_arg>\n", argv[0]);
        return EXIT_FAILURE;
    } else {
        printf("Working with: %s\n", argv[1]);
        str = argv[1];
    }
    // Work here with str

    return EXIT_SUCCESS;
}
```

1. **argc** contient le nombre d'arguments passés au programme
2. **argv** est un tableau de chaînes de caractères, chacune correspondant à un argument
3. **argc** est toujours ≥ 1 et **argv[0]** correspond au chemin utilisé pour démarrer le programme (par exemple "`./mon_programme`").

Exercice 17:

Reprogrammer les fonctions **strlen**, **strcpy** et **strcmp**. On peut omettre la gestion du signe ainsi que le mot clef const.

Exercice 18:

Écrire une fonction permettant de renverser une chaîne de caractère.

Exercice 19:

Écrire une fonction qui vérifie si une chaîne de caractère est un palindrome.

- Une première version peut utiliser la fonction renverser. Attention : si votre fonction reverse modifie la chaîne en place, copiez là avant !
- Trouvez une version plus efficace.

5 Pointeurs et Structures

Exercice 20:

Reprendre l'exercice avec **maybe_init**, et compléter la fonction avec l'affichage des adresses de **a**, **b**, et **c**. Compiler une seul fois, mais executer plusieurs fois le programme. Que conclure sur la tentative de réutilisation d'une adresse affichée lors d'une exécution précédente du programme ?

Exercice 21:

Recoder la fonction **memcpy**. Puis utiliser **memcpy** pour créer une fonction concat, qui permet de concaténer une chaîne à la suite d'une autre. Attention aux conditions nécessaires et aux cas d'erreur. On initialisera un grand tableau pour le résultat, et on prendra 2 chaînes à copier dans ce tableau (la deuxième concaténée à la première).

Exercice 22:

Écrire une fonction **swap** qui échange les valeurs de deux entiers en utilisant des pointeurs. Afficher le contenu des variables avant et après l'appel de la fonction **swap**.

Exercice 23:

Écrire une fonction qui multiplie par 2 chaque élément d'un tableau d'entiers en utilisant l'arithmétique des pointeurs.

Exercice 24:

- Déclarez une structure nommée `character` contenant :
  - Un champ `name` (tableau de 50 `char` pour le nom)
  - Un champ `level` (entier pour le niveau)
  - Un champ `health` (entier pour les points de vie)
- Dans la fonction `main`, déclarez et initialisez une variable de type `character` avec :
  - Nom : `"Hero"`
  - Niveau : `1`
  - Points de vie : `100`
- Écrivez la fonction `print_character` qui prend un `character` en paramètre et affiche ses informations sous la forme :

```
Name: Hero
Level: 1
Health: 100
```

- Appelez la fonction dans `main`

Exercice 25:

Écrire une fonction qui échange deux pointeurs vers des entiers, en utilisant des pointeurs de pointeurs.

- Dans la fonction `main` :
  - Déclarez deux variables entières `sensor1` et `sensor2` avec des valeurs distinctes (par exemple 42 et 100).
  - Déclarez deux pointeurs `ptr1` et `ptr2` qui pointent respectivement vers `sensor1` et `sensor2`.
  - Affichez les adresses contenues dans `ptr1` et `ptr2`, ainsi que valeurs pointées, avant l'échange.

- Écrivez une fonction `swap_pointers` qui prend en paramètre deux pointeurs de pointeurs (`int**`) et échange les pointeurs qu'ils contiennent.
- Utilisez cette fonction dans le `main` pour échanger le contenu de `ptr1` et `ptr2`.
- Affichez à nouveau les adresses et valeurs pointées par `ptr1` et `ptr2`.

Exercice 26:

Écrire des fonctions en C qui mettent à jour une structure de type `character` (définie précédemment) en utilisant le passage par valeur et par adresse.

- Écrivez une fonction `level_up_by_value` qui :
  - Prend un `character` **par valeur**
  - Incrémente son niveau de 1
  - Affiche le nouveau niveau **à l'intérieur de la fonction**
- Écrivez une fonction `heal_by_address` qui :
  - Prend un `character` **par adresse** (pointeur)
  - Ajoute 20 points de vie au personnage
- Dans le `main` :
  - Créez un personnage avec les valeurs initiales de votre choix
  - Appelez `level_up_by_value` et observez le résultat
  - Appelez `heal_by_address` et vérifiez que les points de vie ont bien été mis à jour
  - Affichez l'état final du personnage

Questions de compréhension :

- Pourquoi la modification du niveau dans `level_up_by_value` n'affecte-t-elle pas le personnage original ?
- Quelle est la taille totale des données ajoutées sur la pile pour l'argument de la fonction `level_up_by_value` ?
- Et pour l'argument de la fonction `heal_by_address` ?

Exercice 27:

- Déclarez une structure `Point2D` contenant deux champs `x` et `y` (coordonnées réelles)
- Déclarez une structure `Rectangle` contenant deux champs `top_left` et `bottom_right`
- Écrivez une fonction `compute_area` qui prend un `Rectangle` en paramètre et calcule et retourne son aire (largeur $\times$ hauteur).
- Dans le `main` :
  - Créez un rectangle avec les coordonnées de votre choix
  - Appelez `compute_area` et affichez le résultat
- Écrivez une fonction `scale_rectangle` qui prend un `Rectangle` et un facteur d'agrandissement en paramètres. La fonction doit modifier le rectangle et ne rien retourner.
- Dans le `main`, faites un agrandissement de facteur 2 sur le rectangle, afficher le résultat de `compute_area` sur le rectangle modifié.

6 Exercice de mi-parcours

Exercice 28:

Objectif

Implémenter une version complète du jeu Puissance 4 pour deux joueurs en mode terminal. Le programme devra gérer l'affichage du plateau, les tours de jeu, la détection des victoires et les entrées utilisateur.

Spécifications techniques

1. **Structures de données** :
   - Définissez une structure `Board` contenant :
     - Un tableau 2D (6x7) pour représenter le plateau
     - Un champ indiquant le joueur courant (1 ou 2)
   - Définissez une structure `Player` contenant :
     - Un nom (chaîne de 20 caractères max)
     - Un symbole ('X' ou 'O')
     - Un score
2. **Fonctionnalités requises** :
   - `initialize_board` : Initialise un plateau vide
   - `print_board` : Affiche le plateau dans le terminal avec numérotation des colonnes
   - `make_move` : Place un jeton dans la colonne choisie (avec gestion des erreurs)
   - `check_win` : Vérifie si le dernier coup est gagnant (horizontale, verticale, diagonale)
   - `switch_player` : Change de joueur courant
   - `play_game` : Boucle principale du jeu
3. **Règles du jeu** :
   - Plateau de 6 lignes $\times$ 7 colonnes
   - Les jetons "tombent" dans la colonne choisie
   - Victoire si 4 jetons alignés (toutes directions)
   - Match nul si le plateau est plein
   - Le premier joueur alterne à chaque partie
4. **Interface utilisateur** :
   - Choix du nom pour chacun des joueurs au début du programme
   - À la fin de chaque partie, affichage du score de chaque joueur et démarrage automatique d'une nouvelle partie
   - Affichage clair du plateau avec numérotation des colonnes (1-7)
   - Saisie sécurisée du numéro de colonne
   - Messages pour :
     - Tour du joueur
     - Coup invalide
     - Victoire
     - Match nul
     - Proposition de rejouer

Exemple d'exécution

```
Puissance 4 - Joueur 1 (X) vs Joueur 2 (O)

 1 2 3 4 5 6 7
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

Joueur 1 (X), choisissez une colonne (1-7): 4
```

Contraintes techniques

- Utilisation obligatoire des structures définies
- Gestion des entrées invalides (colonnes pleines, numéros invalides)
- Pas d'allocation dynamique (tableau statique)
- Code modulaire avec fonctions séparées
- Commentaires explicites pour chaque fonction

7 Allocation dynamique

Exercice 29:

Écrire une fonction prenant un entier positif **length**, et un entier **default_value**. La fonction alloue un tableau de *length* éléments initialisés avec la valeur par défaut. Traiter les cas d'erreur. Renvoyer le tableau.

Exercice 30:

Écrire une fonction prenant deux entiers positifs **X** et **Y**, et allouer un tableau à 2 dimensions (**X** colonnes, et **Y** lignes). Traiter les cas d'erreur. Renvoyer le tableau.

Programmation de structure de données usuelles

Pour chaque exercice, écrire une structure, et des fonctions pour interagir avec elle. Quelles fonctions sont susceptibles de générer des erreurs ? Proposer à l'utilisateur de ces fonctions une façon de traiter ces erreurs.

Exercice 31:

Stack (pile): elle fonctionne avec un tableau dynamique, les données sont alignées en mémoire. Prévoir les fonctions :

- init
- push (empiler un élément)
- pop (dépiler un élément)
- length
- lookup (renvoyer, sans le dépiler, le premier élément)
- free_stack (libère la pile)

Utiliser le type int pour les éléments. Trouver un moyen pour résoudre le problème de la gestion des cas d'erreur sans renvoyer une structure.

Exercice 32:

Vector (tableau dynamique): les données sont alignées en mémoire. Prévoir les fonctions :

- init
- push_back (append)
- push_front (insérer à l'indice 0)
- insert (insérer à l'indice x)
- pop (supprimer l'élément à l'indice x)
- length
- get (récupérer l'élément à l'index x)
- clear (supprimer tous les éléments)
- free_vector (libère le Vector)

Utiliser le type int pour les éléments. Trouver un moyen pour résoudre le problème de la gestion des cas d'erreur sans renvoyer une structure.

Note: La liste python est en réalité un Vector (tableau dynamique), alors qu'en général, on parle de "List" pour parler de **listes chaînées**.

Exercice 33:

LinkedList (liste chaînée): elle propose un séquence d'élément par chaînage. On fera trois versions:

1. Une version simple, les éléments sont des maillons. On imagine que le type qu'on veut chaîner est "struct LinkedElement". Le début de la liste est un pointeur vers le premier élément, et l'élément comprend un pointeur vers le suivant.
2. Une vraie data-structure "LinkedList", avec une structure dédiée Element (maillon). Les maillons possèdent un pointeur vers le maillon suivant, et un pointeur vers la donnée.
3. Une version étendue de la "LinkedList" nommée "DoubleLinkedList", les maillons possèdent aussi un membre "précédent".

Pour chaque version, prévoir les fonctions:

- init
- push_back (append)
- push_front (insérer à l'indice 0)
- insert (insérer à l'indice x)
- pop (supprimer l'élément à l'indice x)
- length
- get (récupérer l'élément à l'index x)
- clear (supprimer tous les éléments)
- free_linkedlist (libère la liste)

Utiliser le type int pour les éléments. Trouver un moyen pour résoudre le problème de la gestion des cas d'erreur sans renvoyer une structure.

Exercice 34:

Déclarer une structure `string` qui supportera les opérations classiques sur les chaînes de caractères. Vous devrez gérer dynamiquement la mémoire et éviter les fuites ou les accès invalides.

Implémenter les fonctions suivantes pour manipuler votre structure `string` :

1. `int string_make(struct string* str, char *s)` : Initialise une instance de `string` à partir d'une chaîne C classique (`char *`). Attention à bien allouer la mémoire pour la chaîne et à copier son contenu.
2. `void string_destroy(struct string *str)` : Libère la mémoire occupée par la chaîne.
3. `int string_clone(struct string* dest, struct string *src)` : Clone la chaîne `src` dans la chaîne `dest`. Vérifiez que modifier le clone ne modifie pas l'original.
4. `int string_append(struct string *dest, struct string *src)` : Concatène la chaîne `src` à la fin de `dest`. Pensez à réallouer la mémoire si nécessaire.

5. `int string_compare(struct string *str1, struct string *str2)` : Compare lexicographiquement `str1` et `str2`. Retourne un entier négatif, nul ou positif si `str1` est respectivement inférieure, égale ou supérieure à `str2`.

6. `void string_print(struct string *str)` : Affiche la chaîne sur la sortie standard.

Exercice 35:

(Difficile) On souhaite allouer dynamiquement un tableau à deux dimensions (tab[X][Y]), mais en n'utilisant qu'un seul appel à malloc.

En effet, lorsqu'on alloue dynamiquement un tableau à deux dimensions avec des pointeurs, il faut habituellement :

- allouer un tableau de pointeurs (pour représenter les lignes ou les colonnes),
- puis allouer chaque ligne séparément.

Ici, le défi est d'éviter ces multiples allocations et d'obtenir directement, avec une seule zone mémoire, une variable qui permette d'accéder à tab[i][j] comme si c'était un vrai tableau 2D.

Conseils pour la réflexion:

- Commencer par calculer la taille totale à allouer.
- Est-il plus intéressant de mettre les adresses avant ou après les données?
- Si des pointeurs sont en jeux, quelles sont les valeurs de ces adresses?

Indications pour la réflexion:

- Calculez la taille totale de mémoire nécessaire. Celle-ci doit contenir :
  - d'abord les pointeurs (un pour chaque ligne ou colonne),
  - puis toutes les données du tableau (les int, short, etc.).
- Demandez-vous s'il est plus pratique de mettre les pointeurs avant les données ou l'inverse.
- Rappelez-vous que les pointeurs devront pointer vers les zones correctes dans la partie données .

Chercher l'erreur...

Pour chaque exercice, télécharger le fichier source correspondant, identifier l'erreur et la corriger.

Exercice 36:

Voir **allocation_error_1.c**.

Exercice 37:

Voir **allocation_error_2.c**.

Exercice 38:

Voir **allocation_error_3.c**.

Exercice 39:

Voir **allocation_error_4.c**.

Exercice 40:

Voir **allocation_error_5.c**.