Programmation en C et structures de données

guillaume.revy@univ-perp.fr

CT, Lundi 8/01/2024 de 14h à 16h

Aucun document n'est autorisé.
Les calculatrices, ordinateurs, et téléphones portables sont interdits.

Exercice 1. Importance de l'indentation, malgré tout — $0.75 + 1.25 = 2$ *pts*

Contrairement à certains autres langages, l'indendation d'un programme C n'a pas d'impact sur le résultat de son exécution. Mais elle a une importance pour sa lisibilité.

► **1.** Que fait le programme ci-dessous ?

```
int main(void){float ct,cc1,cc2,note;scanf("%f",&ct);scanf("%f",&cc1);scanf("%f",&cc2);
    note=.25f*cc1+.25f*cc2+.5f*ct;printf("Resultat=%f\n",note);if(note<10){printf("...
    non validee\n");}else{printf("... validee\n");}return 0;}
```

► **2.** Écrire un programme qui lit deux valeurs entières au clavier, puis qui indique si leur produit est positif, négatif, ou nul (en distinguant les 3 cas), sans le calculer explicitement.

Exercice 2. Approximation d'une constante mathématique — $2 + 1 + 1 = 4$ *pts*

La constante mathématique $e \approx 2.178$ peut être approchée de la manière suivante :

$$e \approx \sum_{i=0}^{n} 1/i!,$$

où $i! = 1 \times 2 \times \cdots \times (n-1) \times n$ est le factoriel de i et $0! = 1$.

► **1.** Écrire un programme qui lit un entier n au clavier, puis qui calcule puis affiche une approximation de la constante e en utilisant que des boucles **`for`** et aucun appel à une autre fonction.

► **2.** Écrire une fonction `factoriel` qui calcule et retourne le factoriel d'un entier $i \geq 0$ passé en paramètre.

► **3.** Écrire un nouveau programme pour utiliser, cette fois, une (des) boucle(s) **`while`**, et un (des) appel(s) à la fonction `factoriel`.

Exercice 3. Affichage d'un tas de crêpes — $2 + 0.5 = 3$ *pts*

Soit T un tableau de taille 10, modélisant un tas de crêpes, et où chaque case i correspond au rayon strictement positif de la i-ème crêpe du tas (la crêpe d'indice 0 est celle du dessous du tas).

► **1.** Écrire une fonction `crepes` qui prend en paramètre le tableau T, puis qui affiche le tas de crêpes de manière graphique, comme illustré ci-dessous pour le tableau $T = \{8, 6, 9, 5, 2, 10, 1, 7, 4, 3\}$.

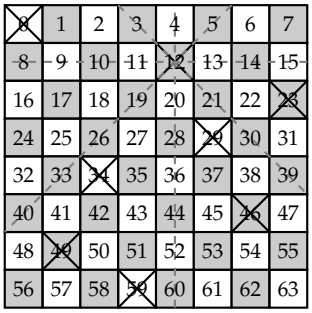
```
          ***|***
         ****|****
       *******|*******
             *|*
    **********|**********
           **|**
        *****|*****
      *********|*********
         ******|******
       ********|********
    -----------+----------
```

► **2.** Illustrer l'utilisation de cette fonction sur un tableau T de taille 10, dont le contenu est lu au clavier.

Exercice 4. Problème des huit reines

$0.5 + 1 + 1 + 2 + 0.5 = 5$ *pts*

Le problème des huit reines consiste à placer huit reines sur un échiquier de taille 8×8, sans qu'aucune d'entre elles ne menace une des sept autres reines. Pour résoudre ce problème, nous numérotons les cases de l'échiquier de 0 à 63, comme illustré ci-dessous.



Pour rappel, une reine menace les pions sur la même ligne, sur la même colonne et sur les deux diagonales passant pas la case sur laquelle elle se trouve, comme illustré en pointillé pour la reine en case 12 sur la figure ci-dessus. Les huit reines peuvent par exemple être placées sur les cases $\{0, 12, 23, 29, 34, 46, 49, 59\}$.

Dans cet exercice, nous représentons la position des huit reines par un tableau T contenant le numéro des cases sur lesquelles elles sont posées.

► **1.** Quelle est la taille de ce tableau ?

Le tableau est dans un premier temps initialisé avec des valeurs -1 :
- $T[i] = -1$: la i-ème reine n'a pas encore été posée,
- $T[i] \neq -1$: la i-ème reine est en case $T[i]$.

► **2.** Écrire une fonction `initialisation` qui initialisa le tableau T passé en paramètre avec des valeurs -1, indiquant qu'aucune reine n'a encore été posée.

► **3.** Écrire une fonction `print` qui affiche le contenu du tableau passé en paramètre.

► **4.** Écrire une fonction `check` qui prend en paramètre le tableau T et l'indice i de la reine que l'on vient de poser, qui vérifie si elle peut être posée en case $T[i]$, puis qui retourne 1 si cela est possible, et 0 sinon.

Pour résoudre le problème des huit reines, l'algorithme utilisé pourra être le suivant :

1. Nous commençons par la première reine ($i = 0$) : nous la posons sur la première case ($T[i] = 0$), puis nous passons à la reine suivante.
2. Dès lors que nous ne pouvons pas poser la i-ème reine sur une case k :
   - soit nous essayons de la poser sur la case suivante $k + 1$ si cela est possible,
   - soit ce n'est pas possible, et nous revenons à la reine $i - 1$.

► **5.** Écrire enfin un programme principal qui permet de résoudre le problème des huit reines, et d'afficher la première configuration trouvée.

Exercice 5. À l'aide !

$1 + 1 + 1 = 3$ *pts*

Un étudiant vous envoie le mail suivant :

"*Je ne comprends pas, mon programme C compile sans erreur avec gcc-12, mais il plante (Segmentation fault : 11). Il est censé allouer et initialiser un tableau de taille 10 dans la fonction foo, puis de l'afficher dans la programme principal. Mais je vous assure, il n'y a pas d'erreur !*".

Pour l'aider, vous lui demandez de vous envoyer son programme (ci-dessous).

```c
int* foo(int n) {
  int *T = (int*)malloc(n*sizeof(int));
  for(int i = 0; i < n; i++)
    T[i] = i;
  int tmp = T[0];
  return &tmp;
}
```

```c
int main(void) {
  int *R = foo(10);
  for(int i = 0; i < 10; i++)
    printf("%d ", R[i]);
  printf("\n");
  free(R);
  return 0;
}
```

► **1.** Proposer une explication, en vous aidant éventuellement d'un dessin.

► **2.** Corriger le programme, sans modifier le prototype des fonctions.

► **3.** Réécrire la fonction `foo` de telle sorte que son retour soit de type **`void`** et expliquer précisément son fonctionnement en vous aidant d'un dessin.

Exercice 6. À vos marches ! $1 + 1 = 2$ *pts*

Soit T un tableau d'entiers positifs de taille n. Ce tableau modélise un ensemble de marches, où la i-ème case représente la hauteur de la i-ème marche. On positionne une bille sur la case d'indice 0. Ensuite la bille peut se déplacer de la case i vers la case $i + 1$ dans deux cas :

1. si la marche i est plus élevée que la marche $i + 1$,
2. ou si la marche $i - 1$ est plus élevée que la marche i, et que la différence de hauteur entre la marche $i - 1$ et la marche i est strictement plus élevée que celle entre la marche i et la marche $i + 1$.

► **1.** Écrire une fonction `marche` qui prend en paramètre le tableau T, puis qui détermine et retourne l'indice de la case sur laquelle la bille s'arrêtera.

► **2.** Illustrer l'utilisation de cette fonction sur un tableau de taille n lu au clavier, et dont les valeurs ont été tirées aléatoirement dans l'intervalle $[0, 20]$.

Exercice 7. Triangle de Pascal $1.5 + 1 + 0.5 = 3$ *pts*

Nous avons vu en TD que le triangle de Pascal est une présentation des coefficients binomiaux dans un tableau triangulaire de n lignes. Il est de la forme suivante, pour $n = 5$.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Pour éviter de gaspiller de la mémoire, nous n'utilisons pas ici un tableau de taille $n \times n$, mais plutôt un tableau de n lignes où chaque ligne $i \in \{0, \cdots, n-1\}$ contient exactement $i + 1$ colonnes. Ensuite, pour le remplir, nous utilisons les relations suivantes :

— $t[i][0] = t[i][i] = 1$ pour toute valeur $i \in \{0, \cdots, n-1\}$,
— $t[i][j] = t[i-1][j-1] + t[i-1][j]$ pour toutes valeurs $i \in \{1, \cdots, n-1\}$ et $j \in \{1, \cdots, i-1\}$.

Pour construire cette structure, nous utilisons l'allocation dynamique.

► **1.** Écrire une fonction `allouer_triangle`, dont le retour est de type **`void`**, qui prend en paramètre la valeur de n, puis qui alloue et remplit le tableau permettant de stocker les n premières lignes du triangle de Pascal comme décrit ci-dessus.

► **2.** Écrire les fonctions `afficher` qui affiche un triangle, et `liberer_triangle` qui libère la mémoire allouée pour un triangle de Pascal en mettant explicitement le pointeur à NULL.

► **3.** Illustrer leur utilisation dans un programme principal.