CY TECH

PRÉ-ING1
SEMESTRE 2 - 2023/2024

DS N° 1 - INFORMATIQUE II

Calculatrice et documents non autorisés

> *Une fonction / procédure peut être écrite en plusieures fonction si besoin.*
> *Si vous ne parvenez pas à répondre à une question, vous pouvez passer à la suivante en supposant que la réponse précédente a été fournie correctement).*
> *Les fonctions déjà écrites peuvent être ré-utilisées dans les questions suivantes.*
> *On ne vous demande pas d'écrire la fonction principale du programme.*
> **Vos programmes doivent être rédigés en C !**
> **Pretez attention à la ROBUSTESSE de vos programmes !!!**

Exercice 1 (*JO de Cergy 2024 : villes olympiques*) *(12.0 pts)*

Nous allons écrire des fonctions permettant de comparer différents stades multi-sports à travers l'agglomération de Cergy.

1. Un `stade` est défini par :
   — son nom (chaîne de caractères),
   — sa capacité maximale de spectateurs,
   — sa surface de terrain utile (valeur réelle en m^2),
   — son année de construction,
   — sa ville,
   — son type.

   avec un `type` qui est une valeur parmi une liste fixe de valeurs :
   — FOOTBALL (le stade accueille principalement des matchs de football),
   — PATINOIRE (le stade contient une patinoire pour des matchs de hockey et du patinage artistique),
   — OMNISPORT (le stade accueille plusieurs événements sportifs distincts les uns des autres).
   — PISCINE (le stade est une piscine ;-) il accueille donc des compétitions ... aquatiques).

   et une `ville` définie par :
   — son nom (chaîne de caractères),
   — son code postal,
   — sa population.

   Définir la ou les structures nécéssaire(s) pour gerer des `stade`.

2. Écrire une fonction/procédure `afficheVille(...)` qui prendra en paramètre une `ville` et qui affichera toutes les informations contenues.
3. Écrire une fonction/procédure `calculerRatio(...)` qui prendra en paramètre un `stade` et qui retournera le rapport entre le nombre d'habitants de sa ville et le nombre maximal de spectateurs qu'il peut accueillir.
4. Écrire une fonction/procédure `compareStade(...)` qui prendra en paramètre deux `stade` et qui retournera une valeur entière comme résultat de comparaison.
   La comparaison renverra la valeur +1 si le premier `stade` est plus grand que le deuxième, renverra -1 si il est plus petit, et 0 sinon.
   La comparaison se fait d'abord sur la capacité, et si les capacités sont équivalentes, alors la comparaison se fera ensuite sur la surface. (La valeur retournée 0 signifiera que les 2 `stade` ont une capacité ET une surface équivalentes).
5. Écrire une fonction/procédure `memeVille(...)` qui prendra en paramètre deux `ville` et qui retournera 1 si elles sont les mêmes, 0 sinon. La comparaison se fera à la fois sur le nom ET sur le code postal.
   Si la comparaison ne peut pas se faire, la valeur retournée sera -1.
6. Le comité olympique impose certains critères sur les lieux des événements pour pouvoir les homologuer :
   — un stade omnisport doit pouvoir accueillir au moins 1% de la population de la ville ET avoir une surface d'au moins $10000m^2$
   — un stade de football doit pouvoir accueillir au moins 1% de la population de la ville
   — une patinoire doit faire au moins $1800m^2$

— une piscine doit faire au minimum 1250m²

Écrire une fonction/procédure `stadeValide(...)` qui prendra en paramètre un `stade` et renverra la valeur 1 si les caractéristiques du stade sont en accord avec les critères du comité olympique, et 0 sinon.

7. Écrire une fonction/procédure `villeOlympique(...)` qui prendra en paramètres :
   — une `ville`
   — un tableau de `stade`
   — la taille du tableau

   et qui retournera 1 si cette ville contient au moins 1 `stade` de chaque `type` qui répond aux critères du comité olympique, et 0 sinon.
8. Écrire une fonction/procédure `villesCandidates(...)` qui prendra en paramètres :
   — un tableau de `ville`
   — le nombre de villes dans le tableau
   — un tableau de `stade`
   — le nombre de stades dans le tableau

   et qui affichera toutes les informations des villes qui sont éligibles à l'accueil des futures jeux olympiques.

Exercice 2 (*Gestion de score d'un jeu vidéo multi-joueurs* ***(8.0 pts)***)

`Forknife` est un jeu en ligne de type battle royale. Bien que l'objectif du joueur soit d'être le dernier en vie à la fin de la partie, un système de score est tout de même utilisé pour indiquer la performance d'un joueur. Dans cet exercice, nous allons utiliser un tableau d'entiers permettant de gérer le score de chaque joueur. Un joueur est identifié par son numéro. Ainsi, la première case du tableau permet de gérer le score du joueur 1, la seconde du joueur 2, etc.

1. **Création du tableau :** Implémenter une fonction qui va permettre de créer le tableau de scores. Cette fonction prend en paramètre le nombre de joueurs présents dans le lobby (inscrits à la partie).
   — Si le nombre de joueurs présents n'atteint pas 50, la partie ne peut pas se faire et on doit alors **quitter le programme** (pas uniquement la fonction).
   — Sinon, un tableau contenant un nombre de cases approprié doit être créé.
   — Le score de chaque joueur doit être initialisé à 0.
   — La fonction doit retourner le tableau créé.
2. **Mise à jour des scores :** Écrire une fonction/procedure `majScore` qui va prendre en paramètre le tableau de score, le numéro du joueur ainsi que la valeur à ajouter au score de ce joueur. Cette fonction doit mettre à jour le score du joueur concerné *mais ne doit utiliser que le formalisme pointeur (pas d'utilisation des symboles 'crochets' [ ]).*
3. **Réallocation du tableau :** de nouveaux joueurs peuvent rejoindre une partie en cours.
   (a) Quel peut être le problème dans ce cas ?
   (b) Quelles pourraient être les étapes à réaliser pour que le score des nouveaux joueurs puisse être pris en compte ?
   (c) Implémenter une fonction/procedure `reallocScore` qui prendra en paramètre le tableau original des scores ainsi que le nombre de nouveaux joueurs à ajouter, et qui permet de résoudre le problème de l'arrivée de ces nouveaux joueurs. *L'utilisation de la fonction realloc() n'est pas autorisée dans cette question.*
4. **High score :** Écrire une fonction/procedure `highScore` qui prend en paramètre le tableau de score et sa taille et qui va communiquer au reste du programme le numéro du joueur ayant le score le plus élevé mais également son score.

Annexe : quelques fonctions utiles

string.h

strlen

Prototype : `size_t strlen(char *s);`
Description : La fonction `strlen` calcule la longueur d'une chaîne de caractères, sans compter le caractère nul '\0'.
Paramètre :
— `s` : Pointeur vers la chaîne dont la longueur est calculée.
Retour : Le nombre de caractères dans la chaîne pointée par `s`.

strcpy

Prototype : `char *strcpy(char *dest, char *src);`
Description : La fonction `strcpy` copie la chaîne pointée par `src`, y compris le caractère nul '\0', dans le tableau pointé par `dest`.
Paramètres :

— **dest** : Pointeur vers le tableau de destination.
— **src** : Pointeur vers la chaîne de caractères à copier.
Retour : Un pointeur vers la chaîne de destination **dest**.

strcmp

Prototype : `int strcmp(char *s1, char *s2);`
Description : La fonction **strcmp** compare les deux chaînes **s1** et **s2**.
Paramètres :
— **s1** : Première chaîne à comparer.
— **s2** : Seconde chaîne à comparer.
Retour : 0 si **s1** et **s2** sont égales, une valeur négative si **s1** est inférieure à **s2**, et une valeur positive si **s1** est supérieure à **s2**.

strcat

Prototype : `char *strcat(char *dest, char *src);`
Description : La fonction **strcat** ajoute la chaîne **src** à la fin de **dest**. Elle suppose que le tableau **dest** a assez d'espace pour contenir les deux chaînes.
Paramètres :
— **dest** : Pointeur vers la chaîne de destination.
— **src** : Chaîne à ajouter à **dest**.
Retour : Un pointeur vers la chaîne de destination **dest**.

---

stdlib.h

malloc

Prototype : `void *malloc(size_t size);`
Description : La fonction **malloc** alloue un bloc de mémoire de taille spécifiée et renvoie un pointeur vers le début du bloc.
Paramètre :
— **size** : La taille en octets de la mémoire à allouer.
Retour : Un pointeur vers le bloc de mémoire alloué ou NULL si l'allocation échoue.

calloc

Prototype : `void *calloc(size_t nmemb, size_t size);`
Description : La fonction **calloc** alloue un bloc de mémoire pour un tableau de **nmemb** éléments de **size** octets chacun. La mémoire est initialisée à zéro.
Paramètres :
— **nmemb** : Nombre d'éléments.
— **size** : Taille de chaque élément.
Retour : Un pointeur vers le bloc de mémoire alloué ou NULL si l'allocation échoue.

atof

Prototype : `double atof(const char *str);`
Description : La fonction **atof** convertit la chaîne de caractères **str** en un nombre à virgule flottante de type double.
Paramètre :
— **str** : Chaîne représentant le nombre à convertir.
Retour : La valeur convertie en double. Si **str** ne contient pas un nombre valide, zéro est retourné.