TP Test Unitaire et Pointeurs Dynamiques

Prérequis

Commencer par créer un répertoire Git public et m'envoyer le lien par Teams.

Créer un dossier en local par exemple TP_POINTEUR_DYN, dedans ouvrir un terminal et taper :
git clone « le chemin de votre repo git donné par github ».

Copié-collé le fichier .gitinore fourni dans le dossier (utile principalement pour ceux sous VisualStudio).

Dans le terminal taper :
git add .gitignore
git commit -m « ajout gitignore »
git push

Vérifier dans Github que votre commit a bien été effectué.

Maintenant créer un nouveau projet, pour cela, placez-vous bien dans le dossier TP_POINTEUR_DYN !!

Vous êtes prêt à attaquer le TP.

Commiter régulièrement !

Test Unitaire

Les tests unitaires permettent de tester chaque fonction d'un programme unitairement.
Cela permet de vérifier que notre application fonctionne correctement à tout moment du développement. Si une régression (problème qui n'existait pas auparavant) apparaît, on est directement informé.

Il existe plusieurs framework de tests en C.
Nous allons utiliser MinUnit qui est assez minimaliste mais il est simple et facile à intégrer.
Que ce soit sous Windows, Lunix ou Mac, on lancera les tests comme on lance un programme classique.

1. Macros

MinUnit utilise des **macros.**

Une **macro** est un bout de code que le préprocesseur remplace par un autre fragment de code avant que le compilateur ne commence à compiler.
Les macros sont principalement définies à l'aide de la directive du préprocesseur **#define**.

Exemples :

```
#define PI 3.14159
```

Le préprocesseur remplacera PI, partout dans le code où il le trouvera, par 3.14159, avant que le code soit compilé.

```
#define MAX(a,b) ( (a)> (b) ? (a) : (b) )
```

Cette macro renvoie le maximum entre les deux arguments passés en paramètre.

Si dans votre code vous utilisez MAX(x,y) le préprocesseur remplacer par ((x)> (y) ? (x) : (y))

2. Assertion

Une assertion est une instruction utilisée pour vérifier que des conditions spécifiques sont respectées à un moment donné dans le code.
Si ce que l'assertion évalue renvoie « faux » alors le programme signale une erreur ou peut se terminer.

Afin d'effectuer nos tests unitaires, nous allons créer une macro pour faire les assertions :

```
1. #define mu_assert(message, test) do { if (!(test)) return message; } while (0)
```

Que fait cette macro à votre avis ?

3. Lancer les tests

Pour lancer les tests, les comptabilisés et récupérer les messages d'erreur, nous allons créer une autre macro :

```
#define mu_run_test(test) do { char *message = test(); tests_run++; \
                               if (message) return message; } while (0)
```

4. Premier test unitaire

Habituellement le code de votre application et les tests sont dans deux fichiers séparés.

Pour cet exercice vous pouvez les mettre ensemble.

Créer un fichier tests_tp2.c

Commencer par inclure les librairies habituelles.
Puis définir les 2 macros ci-dessus.

Copier la fonction basique suivante :

```
int carre(int a)
{
    return a * a;
}
```

Nous allons créer une fonction pour faire notre premier test unitaire.
Une bonne pratique est de le nommer *test_nomDeLaFonctionTestee()*.
La valeur de retour sera de type *char **, afin de permettre de retourner un message d'erreur en cas d'échec du test.

Autre particularité, nous allons rendre cette fonction statique, cela signifie que la fonction ne pourra pas être utilisée en dehors du fichier.
Comme mentionné plus haut, à part pour cet exemple, vos tests unitaires devront être dans des fichiers séparés. Mettre les fonctions de test statique permet de s'assurer qu'elles ne seront pas utilisées par mégarde dans votre programme.

Le prototype du premier test sera donc :

```
static char * test_carre() ;
```

Qu'allons-nous mettre dans cette fonction de test ?

Nous allons maintenant créer une nouvelle fonction qui permettra de lancer tous les tests, elle sera également statique :

```
static char * all_tests()
{
  mu_run_test(test_carre);
  // lister tous les tests ici quand il y en aura d'autres
  return 0;
}
```

Et enfin nous allons faire le ***main*** qui lancera tout ça :

```
int main()
{
  char *result = all_tests(); // on lance tous les tests
  if (result != 0)   // il y a eu une erreur
  {
    printf("%s\n", result);  // on affiche le message d'erreur
  }
  else
  {
    printf("All tests passed.\n");
  }
  printf("Tests run: %d\n", tests_run); // on affiche le nombre de tests lancés
  return result != 0;
}
```

Exercice 1

Rajouter deux tests unitaires pour tester la fonction carre.

Exercice 2

Faire les tests unitaires pour cette fonction :

```
Const char* categorize_age(int age)
{
  if (age < 0) {
    return "Invalid age";
  }
  else if (age < 13) {
    return "Child";
  }
  else if (age < 20) {
     return "Teenager";
  }
  else if (age < 65) {
     return "Adult";
  }
  else {
     return "Senior";
  }
}
```

Simulateur de Piano Virtuel

Développer un simulateur de piano virtuel pour jouer des sons et détecter les interactions clavier.

Chaque touche correspondra à une note spécifique du piano, et la fréquence de chaque note sera déterminée par un tableau prédéfini de fréquences.
On se limitera à 12 notes pour cet exercice.

Fonctionnalités à implémenter

1. Structures de Données :
    - Définir une structure **Note** pour stocker la *fréquence* et la *durée* de chaque note de musique.

2. Création et Gestion des Notes :
    - Écrire une fonction *creerNote* pour allouer et initialiser une note. La fonction retourne une *Note*.
    - Écrire une fonction *libererNotes* pour libérer toutes les notes allouées dynamiquement à la fin de la simulation.

3. Simulation de Piano :

   - Écrire une fonction *simulerPiano* pour écouter les entrées du clavier et jouer la note correspondante lorsque les touches spécifiées sont pressées.

     Cette fonction prendra en paramètre un **tableau dynamique de pointeurs** de *Note* ainsi que le nombre de notes (ici max 12).
     Lorsque l'on écrit : *Note * note;* cela représente un tableau de Note.
     Si l'on souhaite avoir un tableau de pointeur de Note, il faudra écrire : *Note **notes ;*

     Mappez chaque note à une touche spécifique sur le clavier.
     Implémentez une boucle infinie qui attend l'entrée de l'utilisateur et joue la note correspondante jusqu'à ce que 'q' soit pressé pour quitter.

     Pour ceux sous Windows utiliser Beep pour jouer les sons et _kbhit/_getch pour détecter les touches pressées (comme pour le projet Snake).
     Pour ceux sous Linux et Mac réutiliser les fonctions utilisées pour le projet Snake pour l'appui des touches. Pour le son cela est plus compliqué donc faites seulement un affichage du nom des notes plutôt que de faire réellement le son.

4. Gestion de la Mémoire :

   - Assurer que tout l'espace mémoire alloué pour les notes est correctement libéré à la fin de la simulation pour éviter les fuites de mémoire. Faire une fonction *libererNotes* pour cela.

5. Main :

   - Utiliser le tableau suivant pour les fréquences à donner à la fonction Beep pour simuler le piano :
     float frequencies[12] = { 261.63, 277.18, 293.66, 311.13, 329.63, 349.23, 369.99, 392.00, 415.30, 440.00, 466.16, 493.88 };

   - Créer un **tableau dynamique de pointeurs** vers ces structures *Note*.
     Créer une *Note* pour chacune des fréquences, les stocker dans le *tableau de pointeur de Note*.
     Simuler le Piano avec ce tableau de Notes.
     Libérer la mémoire.

6. Tests unitaires :

   Faites les tests unitaires de vos différentes fonctions.