ZZ1 – C – TP n°2

Manipulation de pointeurs, Allocation dynamique de mémoire, fichiers texte et le débogueur

Le but de ce TP est de se familiariser avec les pointeurs et l'allocation dynamique de mémoire via le calcul vectoriel et matriciel.

A : Produit scalaire de deux vecteurs

Soient X et Y deux vecteurs d'ordre n, le produit scalaire de X et Y est $X \cdot Y = \sum_{i=0}^{n-1} X_i Y_i, \quad X, Y \in \Re^n$.

Supposons que chaque vecteur est stocké dans un fichier texte du format suivant:

3

1.0 2.0 3.0

La première valeur est l'ordre du vecteur, les valeurs suivantes sont les composantes du vecteur.

A.1. Ecrire une fonction de lecture de fichier de vecteur. Le nom de fichier doit être passé en paramètre. L'ordre et les composantes du vecteur peuvent être soit passés en paramètre, soit retournés comme résultat de fonction. La mémoire nécessaire au vecteur sera allouée dynamiquement après la lecture de l'ordre de vecteur. (suggestion : mettez l'ordre de vecteur en paramètre et retournez le vecteur)

Nous demandons d'utiliser un pointeur pour le parcours de vecteur. Dans le cas où l'ordre et les composantes du vecteur sont passés en paramètre, l'emploi du passage de paramètre par adresse est nécessaire afin que le résultat de lecture puisse être conservé.

A.2. Ecrire une fonction d'affichage de vecteur. Le vecteur et son ordre seront passés en paramètre.

A.3. Ecrire la fonction principale pour tester vos fonctions avec des vecteurs d'ordre différent.

A.4. Ajouter une fonction de calcul du produit scalaire de deux vecteurs et tester à nouveau.

B: Produit Matrice-Vecteur

Soient A une matrice carrée des réels d'ordre n et X un vecteur d'ordre n. Nous allons calculer $Y = AX$, avec $Y_i = \sum_{j=0}^{n-1} A_{ij} X_j,\ i = 0, 1, ..., n-1; X, Y \in \Re^n$.

Supposons que la matrice est aussi stockée dans un fichier texte du format suivant:

3

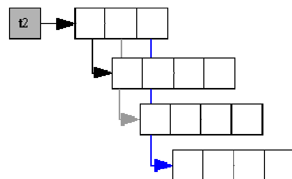
1.0 2.0 3.0

2.0 3.0 4.0

3.0 4.0 5.0

La première valeur est l'ordre de matrice carrée, les suivantes sont les éléments de matrice.

B.1. Ecrire une fonction de lecture de fichier de matrice en respectant les mêmes consignes que A.1. La structure suivante sera utilisée dans l'allocation dynamique de mémoire pour la matrice, qui se réalise en 2 étapes : nous allouons d'abord un tableau de n pointeurs (n étant le nombre de lignes de la matrice), puis nous faisons une allocation de n éléments (ici, n est le nombre de colonnes de la matrice) pour chaque pointeur. De cette manière, chaque pointeur du tableau pointe sur une ligne de matrice.



B.2. Ecrire une fonction d'affichage de matrice. La matrice et son ordre seront passés en paramètre.

B.3. Modifier la fonction principale afin de tester ces fonctions.

B.4. Ecrire une fonction de multiplication de matrice-vecteur en utilisant la fonction de produit scalaire écrite en A.4.

B.5. Modifier la fonction principale et tester à nouveau.

Utilisation du débogueur :

Pendant la programmation, en dehors des erreurs syntaxiques et sémantiques que les compilateurs peuvent détecter, on peut aussi commettre d'autres erreurs : comme des erreurs de segmentation fault qui interrompent l'exécution d'un programme ou celles qui nuisent au résultat d'exécution. Ces erreurs sont plus difficiles à détecter, leurs corrections demandent plus d'attention et de minutie. Dans ce cas, l'utilisation d'un débugger est plus efficace que les **printf**s entre les lignes de code.

Un debugger permet de positionner des points d'arrêt, d'exécuter un programme pas à pas, de vérifier les valeurs des variables et la pile d'exécution. Dans ce TP, vous pouvez utiliser **gdb/ddd** si vous travaillez sous Linux, ou **dbx/dbxtool** si vous travaillez sous Unix. **ddd** et **dbxtool** sont les interfaces graphiques respectives de **gdb** et **dbx**, ils ne sont pas forcément installés sur tous les systèmes.

1. Compilation

Afin qu'un programme puisse être utilisé par un débugger, il faut le compiler avec l'option **–g**, qui permet de produire des informations supplémentaires nécessaires aux débuggers. Exemple :

`$ gcc –g votrePrg.c –o votrePrgExec` ou `$ gcc votrePrg.c –o votrePrgExec -ggdb`

2. Exécution

`$ nomDuDebugger votreProgExec`

`nomDuDebugger` doit être remplacé par le nom du debugger que vous avez choisi. Par exemple : **gdb/ddd** ou **dbx/dbxtool**. Une fois cette commande exécutée, on est entré dans l'environnement d'exécution du débugger. Qu'importe le debugger de votre choix, les premières étapes d'exécution et les commandes de base sont identiques.

- On peut commencer par positionner un ou plusieurs points d'arrêt (stop) dans le code via la commande **stop** ou **break**. Exemple :
  - **dbx** : **stop at numero_ligne**
  - **gdb** : **break nom_fonction**
  - **ddd** : positionner le curseur au début d'une ligne puis cliquer sur le bouton **break**
- Suppression d'un point d'arrêt :
  - **dbx** : **clear numero_ligne**
  - **gdb** : **clear nom_fonction**
  - **ddd** : positionner le curseur sur un point d'arrêt puis cliquer sur le bouton **clear**
- On lance l'exécution du programme avec la commande (ou le bouton) **run**. L'exécution s'arrête au premier arrêt.
- Ensuite, on peut continuer l'exécution du programme étape par étape en utilisant les commandes **next** ou **step**. **next** exécute complètement l'instruction courante, **step** entre à l'intérieur de l'instruction si celle-ci est une fonction utilisateur.
- Les valeurs des variables au cours d'une exécution peuvent être visualisées avec
  - **dbx** : la commande **print** : **print nom_variable** ; **print * pointeur** affiche la valeur pointée par (le contenu de la case mémoire pointée par) un pointeur.
  - **gdb** : **print nom_variable** ;
  - **ddd** : **sélection d'une variable** puis **cliquer sur le bouton display**
- **cont** permet de continuer l'exécution du programme jusqu'au prochain point d'arrêt (stop) ou terminer l'exécution s'il n'y a plus de stop.
- **quit** permet de quitter l'exécution du programme.
- **bt** : permet d'afficher la trace de la pile d'exécution.
- Les commandes concernant la pile d'exécution sont : **where** (ou bouton **stack**), **up** et **down.**
- La commande **help** donne la liste de toutes les commandes. **help** suivi du nom d'une commande ou du nom d'une famille de commandes explique l'utilisation de la commande ou celle de la famille de commandes.

Essayez le débogueur sur le programme suivant :

```
int main()
{
      char * chaine ;

      scanf("%s", chaine) ;
      printf(chaine) ;
}
```