S3A2a

Modélisation 1D des matrices et autres tenseurs

Licence Informatique : Programmation Nécessaire

Stéfane

Table des Matières

1 Objectif

On évite au tant que faire ce peut de modéliser les types abstraits bidimensionnels avec de la mémoire en 2D[1]*. On utilisera de préférence la mémoire 1D – sous forme de vecteurs donc – pour modéliser les* TA *2D.*

Par exemple une matrice $A_{M \times N}$ *sera représentée par un vecteur de taille* $M * N$ *où les valeurs seront celles de* A *rangées ligne par ligne (ou colonne par colonne).*

2 Le TA `matrix`

Dans `include`*, créez un fichier* `matrix.h` *contenant la déclaration du type abstrait des matrices ainsi que les déclarations des fonctions publiques.*

Le type abstrait d'une matrice

```
1  typedef struct matrix {
2      int nbLig;
3      int nbCol;
4    double * values;
5  } matrix_t;
6  matrix_t * consMatrix ( int nlig, int ncol );
7  matrix_t * cpyMatrix( matrix_t * M );
8  void freeMatrix( matrix_t ** ptrM );
9  void printMatrix( matrix_t * M, char * entete );
10 matrix_t * scanMatrix();
11 matrix_t * addMatrix ( matrix_t *  A, matrix_t *  B );
12 matrix_t * multMatrix ( matrix_t *  A, matrix_t *  B );
```

a) `matrix_t * consMatrix ( int nlig, int ncol );`
Initialise les champs `nbLig` *et* `nbCol` *et alloue à* `values` *un vecteur de* `nbLig*nbCol` *réels.*

b) `matrix_t * cpyMatrix( matrix_t * M );`
Duplique la matrice `M`*.*

[1] *En l'état actuel des modèles de machines existants, la mémoire 2D n'existe pas.*

c) `void freeMatrix( matrix_t ** ptrM );`
Libère toute la mémoire occupée par `*ptrM` *et lui assigne* `NULL`.

d) `void printMatrix( matrix_t * M, char * entete );`
Visualise la matrice `M` *après avoir affiché son* `entête`. *(Une chaîne de caractères)*

e) `matrix_t * scanMatrix();`
Saisit au clavier les données.

f) `matrix_t * addMatrix ( matrix_t * A, matrix_t * B );`
dont la définition est:

$$(A)_{n\times m} + (B)_{n\times m} = (a_{i,j}) + (b_{i,j}) = (c_{i,j}) = (C)_{n\times m}$$

g) `matrix_t * multMatrix ( matrix_t * A, matrix_t * B );`
dont la définition est:

$$(A)_{n\times m} \cdot (B)_{m\times p} = \left(\sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}\right) = (c_{i,j}) = (C)_{m\times p}$$

Dans `src`, *créer le fichier* `matrix.c` *pour y ranger les définitions des fonctions déclarées dans* `matrix.h`.

3 La fonction principale

Récupérer les fichiers d'entête et source des vecteurs du précédent TP et placez les dans leurs dossiers respectifs.

Créez le `Makefile` *qui devra ressembler à ceci :*

Le `Makefile` des vecteurs et matrices.

```
IDIR = include
ODIR = obj
SDIR = src
BDIR = bin

CC = gcc
CFLAGS = -Wall -I$(IDIR) -c
LDFLAGS = -lm

PROG=$(BDIR)/mat

_DEPS = vector.h matrix.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJS= vector.o matrix.o main.o
OBJS = $(patsubst %,$(ODIR)/%,$(_OBJS))

.PHONY : run all dirs clean delete

run : all
  ./$(PROG)

all : dirs $(OBJS)
  $(CC) -o $(PROG) $(OBJS) $(LDFLAGS)

$(ODIR)/%.o : $(SDIR)/%.c $(DEPS)
  $(CC) -o $@ $< $(CFLAGS)

dirs :
  @mkdir -p $(ODIR) $(BDIR)

clean :
  rm -rf $(ODIR) core

delete : clean
  rm -rf $(BDIR)
```

Créez un fichier `main.c` qui contenant uniquement la fonction `main` qui devra :

a) Appeler la fonction `A = scanMatrix();`
b) Appeler la fonction `B = scanMatrix();`
c) Visualiser ces deux matrices
d) En fonction de la compatibilité des dimensions des deux matrices proposez: l'addition, la multiplication ou bien les deux;
e) Appelez la fonction souhaitée par l'utilisateur et visualiser le résultat.

Ensuite vous pouvez changer cette fonction principale pour qu'elle propose à l'utilisateur un menu où toutes les fonctions possibles des vecteurs et des matrices sont proposées (ainsi que la possibilité de quitter l'application)