TP 5 : Compilation séparée

Exercice 1 : Fonctions et repère orthonormé

Nous allons manipuler dans ce TP des points dans l'espace (en trois dimensions donc) et des vecteurs (au sens mathématique). Nous allons pour ce faire utiliser des **vecteurs** pour représenter ces structures dans un repère orthonormé `(O,i,j,k)` tridimensionnel.

Un point et un vecteur seront donc représentés par une structure que vous définirez.

Vous créerez dans cet exercice un fichier **maLibMath.h** contenant les prototypes des fonctions suivantes et **maLibMath.c** contenant l'implantation des fonctions (vous êtes libres d'utiliser des pointeurs ou non).

1) Ecrire une fonction ***`Point Pt=Demande_Point()`*** qui créé un « *point 3D* » à partir des entrées de l'utilisateur
2) Ecrire une fonction **`Point Pt=`*`Creer_Point(float x, float y, float z)`*** qui créé un « *point 3D* » à partir des coordonnées `(x,y,z)` données en paramètre.

3) Ecrire une fonction **`Vecteur V=`*`Vectorise(Point Pt1,Point Pt2)`*** qui définit le vecteur $\overrightarrow{AB}$ à partir des points `A` et `B`.
4) Ecrire une fonction **`Vecteur PV=`*`Produit_Vectoriel(Vecteur V1,Vecteur V2)`*** qui renvoie le vecteur résultat du produit vectoriel de 2 vecteurs.
5) Ecrire une fonction **`float PS=`*`Produit_Scalaire(Vecteur V1,Vecteur V2)`*** qui renvoie le produit scalaire de 2 vecteurs.
6) Ecrire une fonction **`float N=`*`Norme(Vecteur V)`*** qui calcule la norme (distance) d'un vecteur.

7) Ecrire une fonction **`Equation Eq=`*`Equation_Plan(Point Pt1,Point Pt2,Point Pt3)`*** qui, à partir de 3 points passés en paramètre renvoie l'équation du plan correspondant sous la forme de 4 coefficients `[a, b, c, d]` représentant l'équation $ax + by + cz + d = 0$.

   On stockera les 4 coefficients dans une seule et même structure (que vous définirez).

8) Ecrire une fonction Affiche_***`Equation_Plan(Equation Eq)`*** qui affiche de manière « lisible » une équation de plan.

Nota : vous pouvez ajouter autant de fonctions internes utiles à vos calculs !

Compilez le fichier C. Qu'obtenez-vous ?

Exercice 2 : Un programme pour les utiliser toutes !

Ecrire un programme en C utilisant la librairie définie plus haut qui permet :

1) à l'utilisateur de rentrer les coordonnées de 3 points dans l'espace.
2) de déterminer les coordonnées du centre de gravité `G` du triangle dont les sommets correspondent aux 3 points définis par l'utilisateur et de l'afficher
3) de calculer et d'afficher le périmètre et l'aire du triangle.
4) de calculer l'équation du plan passant par ces 3 points et l'afficher

Exercice 3 : compilation séparée

L'utilité de la compilation séparée est triple :

- la programmation est modulaire, donc plus compréhensible ;
- la séparation en plusieurs fichiers produit des listings plus lisibles ;

- la maintenance est plus facile car seuls les modules modifiés sont recompilés.

La compilation séparée se fait traditionnellement avec la commande **make**. Cette commande :

- assure la compilation séparée grâce à la commande **gcc** ;
- utilise des macro-commandes et des variables ;
- permet de ne recompiler que le code modifié ;
- et permet d'utiliser des commandes du shell.

La commande **make** requiert pour son fonctionnement un fichier nommé **Makefile**. Le fichier **Makefile** indique à la commande make comment exécuter les instructions nécessaires à l'installation d'un logiciel ou d'une bibliothèque. Le fichier **Makefile** doit se trouver dans le répertoire courant lorsque l'on appelle la commande **make** à l'invite du shell et les instructions contenues dans ce fichier doivent respecter une syntaxe particulière expliquée ci-après.

Une règle est une suite d'instructions qui seront exécutées pour construire une cible, mais uniquement si des dépendances sont plus récentes (par rapport à la dernière construction de la cible). La syntaxe d'une règle est la suivante :

```
cible: dépendances
	commandes
```

La cible est généralement le nom d'un fichier qui va être généré par les commandes qui vont suivre. Les dépendances sont les fichiers ou les règles nécessaires à la création de la cible.
Les commandes sont des commandes shell qui seront exécutées au moment de la construction de la cible. La tabulation avant les commandes est obligatoire et si la commande dépasse une ligne, il est nécessaire de signaler la fin de ligne avec un caractère antislash "\".

Mon premier Makefile

```
# construction du TP4

all: maLibMath.o TP4.o

	gcc –o TP4 maLibMath.o TP4.o

maLibMath.o : maLibMath.c MaLibMath.h

	gcc -c maLibMath.c -Wall -o MaLibMath.o

TP4.o: TP4.c

	gcc -c TP4.c -Wall -o TP4.o
```

Les lignes commençant par le caractère **#** sont des lignes de commentaires.

Le **Makefile** ci-dessus a trois règles destinées à construire les cibles *all*, *maLibMath.o* et *TP4.o*. La cible principale all nécessite les fichiers *maLibMath.o* et *TP4.o* pour être exécutée afin d'obtenir le fichier exécutable *TP4*.

La commande **make** peut être exécutée sans argument. Dans ce cas, elle exécute la première règle rencontrée.

On peut aussi utiliser des variables (macro-commandes dans un fichier **Makefile**). La déclaration se fait par la règle **NOM = VALEUR**. La valeur affectée à la variable peut comporter n'importe quels caractères. Elle peut être aussi une autre variable.
Par exemple, la syntaxe de l'appel d'une variable est la suivante : **$(NOM)**

```
# $(BIN) est le nom du fichier binaire généré
BIN = TP5
# $(OBJECTS) sont les objets qui seront générés après la compilation
OBJECTS = TP4.o maLibMath.o
# $(CC) est le compilateur utilisé
CC = gcc
# all est la première règle à être exécutée (1ère règle)
all: $(OBJECTS)
$(CC) $(OBJECTS) -o $(BIN)
…
```

Attention : il existe des conventions de nommage sur des noms des d'exécutables et leurs arguments (Ex : **CC** pour le compilateur) ou pour les noms de cible (Ex : **all**). Nous y reviendrons prochainement.

Exercice : Ecrire un fichier **Makefile** qui permette de compiler le code de l'exercice à l'aide de variables.