Exercice III

Compilation de code sur plusieurs fichiers

Pour compiler un programme écrit sur plusieurs fichiers, il sera plus simple d'utiliser la commande `make` avec le fichier `Makefile`.
(http://www.gnu.org/software/make/manual/make.html)
Le fichier `Makefile` permet de décrire par le biais de cibles dans un fichier, une succession de commandes qui s'exécutent en tenant compte d'un graphe de dépendances.
Avec `make`, uniquement les fichiers nécessaires (modifiés) seront (re-)compilés.
La présence d'un Makefile permet de simplifier la compilation de programme dont le code source est réparti en plusieurs unités (morceaux) compilées séparément.

Appliquons cela au programme de l'exercice 2 sur la structure `Point`.
Le code source est composé des fichiers: Point.hpp, Point.cpp et PointTest.cpp.
Appelons notre programme: **test-point**.

1. Premier Makefile

Ecrivez le `Makefile` suivant dans le même répertoire où se trouvent les fichiers de notre programme.

```
#
# Cible de l'exécutable : édition des liens des fichiers objet
#
test-point: Point.o PointTest.o
	g++ -o test-point Point.o PointTest.o

#
# Cibles de dépendances : compilation des sources en fichiers objet
#
# Note: on a fait le choix ici de compiler avec les options de débogage: -g -O0
#
Point.o: Point.cpp Point.hpp
	g++ -std=C++11 -Wall -g -O0 -c Point.cpp

PointTest.o: PointTest.cpp catch.hpp
```

```
	g++ -std=C++11 -Wall -g -O0 -c PointTest.cpp

#
# Cibles habituelles
#
cleaner: clean
	rm -f test-point

clean:
	rm -f *.o
```

1.a. Vérifions que le `Makefile` est bien écrit.
Dans le terminal, en étant dans le même répertoire, exécutez la commande `make clean`
Si tout se passe bien, cette commande devrait effacer les fichiers .o déjà présent dans le répertoire suite au précédent exercice.

1.b. Exécutez la commande `make`
Si tout se passe bien, la compilation devrait se lancer avec succès. Vérifiez que le programme test-point a bien généré et exécutez-le. Il devrait donner les mêmes résultats qu'au précédent exercice.

2. Variable cible, variable liste de dépendances

2.a. Modifiez le `Makefile` de la façon suivante :
Pour la cible de l'exécutable, remplacez dans la commande de compilation `test-point` par `$@`

```
#
# Cible de l'exécutable : édition des liens des fichiers objet
#
test-point: Point.o PointTest.o
	g++ -o $@ Point.o PointTest.o

...
```

`$@` est ce qu'on appelle une variable prédéfinie. Elle contient la valeur de la cible.

Re-exécutez les commandes de la partie 1. Vérifiez qu'on obtient le même résultat.

2.b. Modifiez à nouveau le `Makefile` de la façon suivante :
Pour la cible de l'exécutable, remplacez dans la commande de compilation, la liste d'objets par `$^`

```
#
# Cible de l'exécutable : édition des liens des fichiers objet
#
test-point: Point.o PointTest.o
	g++ -o $@ $^

...
```

`$^` est aussi une variable prédéfinie. Elle contient la liste de dépendances définie sur la cible courante.

Re-exécutez les commandes de la partie 1. Vérifiez qu'on obtient le même résultat.

3. Variable première dépendance

3.a. Modifiez le `Makefile` de la façon suivante :
Pour les cibles de dépendances, remplacez dans la commande de compilation le nom du fichier source par `$<`

```
...
# Cibles de dépendances : compilation des sources en fichiers objet
#...
Point.o: Point.cpp Point.hpp
	g++ -std=C++11 -Wall -g -O0 -c $<

PointTest.o: PointTest.cpp catch.hpp
	g++ -std=C++11 -Wall -g -O0 -c $<

...
```

`$<` est une autre variable prédéfinie. Elle contient la valeur de la première dépendance.
En C/C++, il faut généralement mettre l'unité de compilation (le fichier source) comme première dépendance de la cible de fichier objet.

Re-exécutez les commandes de la partie 1. Vérifiez qu'on obtient le même résultat.

4. Variables définies par le développeur

On peut définir des variables dans un `Makefile`, pour encore simplifier son écriture.

4.a. Modifiez le `Makefile` de la façon suivante :
En début de fichier, déclarez les variables `SRC`, `INC` et `OBJ`

```
#
# Variables
#
SRC = Point.cpp PointTest.cpp
INC = Point.hpp
OBJ = $(SRC:.cpp=.o)
...
```

Pour la cible de l'exécutable, remplacez les dépendances par `$(OBJ)`
Pour les cibles des dépendances, remplacez le fichier d'en-tête Point.hpp par `$(INC)`

```
...
# Cible de l'exécutable : édition des liens des fichiers objet
#
test-point: $(OBJ)
	g++ -o $@ $^

#
# Cibles de dépendances : compilation des sources en fichiers objet
#...
Point.o: Point.cpp $(INC)
	g++ -std=C++11 -Wall -g -O0 -c $<

PointTest.o: PointTest.cpp catch.hpp
	g++ -std=C++11 -Wall -g -O0 -c $<

...
```

Note:

- l'affectation d'une variable est aussi simple que ça. On peut concaténer des valeurs à une variable avec l'opérateur `+=` (ajout à droite avec un espace)
- l'expansion d'une variable se fait par l'utilisation de `$(...)`
- la possibilité de substitution pendant l'expansion. (utilisée pour la variable `OBJ`)

Re-exécutez les commandes de la partie 1. Vérifiez qu'on obtient le même résultat.

5. Commande par défaut

La commande `make` possède des règles de dépendances par défaut qu'on peut utiliser pour simplifier l'écriture de notre `Makefile`.
Par exemple, `make Point.o` lance automatiquement la commande:

```
g++ Point.cpp -o Point
```

sans qu'on ait rien écrit dans le `Makefile`. C'est le résultat :

- d'une dépendance implicite : si Point.cpp existe dans le répertoire, alors le potentiel exécutable **Point** dépend de Point.cpp
- d'une commande par défaut : pour fabriquer un exécutable (sans extension) à partir d'un fichier source C++ (fichier.cpp <--> [nom_fichier].[extension]), la commande de compilation adaptée à lancer est `g++ ...` si rien d'autre n'est précisé

5.a. Pour notre `Makefile`, tout en utilisant les commandes par défaut, on veut garder les options de compilation `-std=C++11 -Wall -g -O0` pour les compilations séparées. Modifiez le fichier en y ajoutant au début la variable `CXXFLAGS` de la façon suivante :

```
#
# Variables
#
CXXFLAGS =  -std=C++11 -Wall -g -O0
SRC = Point.cpp PointTest.cpp
INC = Point.hpp
OBJ = $(SRC:.cpp=.o)
...
```

La variable `CXXFLAGS` est une variable prédéfinie qui ne contient rien par défaut, mais qui est utilisée dans la commande de compilation par défaut. Ici, on vient d'effectuer une surcharge de la variable prédéfinie.

Effacez également les commandes au niveau des cibles de dépendances pour que la commande par défaut soit utilisée :

```
...
#
# Cibles de dépendances : compilation des sources en fichiers objet
#...
Point.o: Point.cpp $(INC)
PointTest.o: PointTest.cpp catch.hpp


...
```

Re-exécutez les commandes de la partie 1. Vérifiez qu'on obtient le même résultat.