Université de Paris
Master 1

Langages à Objets Avancés
Année 2022-2023

TP n°1

Introduction à C++

1 Prise en main

Exercice 1 (Un programme Simple)
Créez un répertoire `ObjetsAvances/TP1/Calculette`, puis écrivez (dans un éditeur de votre choix) un programme `calculette.cpp`. Le programme affichera deux entiers tirés au hasard et demandera à l'utilisateur d'en donner la somme tant qu'il n'a pas la réponse exacte. Il comptera le nombre d'échecs et l'indiquera à la sortie.
Indications :

- pour tirer un nombre au hasard :
```
#include <cstdlib>
#include <ctime>
...
srand(time(nullptr));
int x { rand() };
```
- vous compilerez avec `g++ --std=c++11 -Wall`

Exercice 2 (Makefile)
En c++ la compilation est séparée, de sorte qu'il est inutile de recompiler des codes qui n'ont pas été modifiés. Nous utiliserons l'outil "`make`" pour nous y retrouver. La section 4 de ce TP vous présente d'autres environnements de travail que vous pourrez préférer sur votre machine, mais il vous faut de toutes façons savoir écrire un `Makefile` car lorsque vous nous rendrez un travail nous ne voudrons **absolument pas** quelque chose dépendant de votre machine ou de vos outils.

La paire (commande `make` / fichier `Makefile`) permet de faire une série de compilations à l'aide d'une seule commande. `Makefile` est un fichier texte qui contient des règles de la forme :

```
cible : dependances
        commandes
```

les dépendances sont d'autres cibles, séparées par des espaces, ou bien un nom de fichier pour indiquer le cas où celui ci aurait été modifié. (Attention à la syntaxe : avant "commandes" il doit y avoir une tabulation).

Dans le répertoire qui contient le fichier `Makefile`, la commande `make cible` se chargera d'exécuter les commandes de votre cible, après avoir rafraîchit les dépendances si c'est nécessaire.

1. Dans le répertoire de votre calculette créez un fichier `Makefile` contenant :

```
CPP=g++ --std=c++11 -Wall

all :  calculette

calculette : calculette.o
     $(CPP) -o calculette calculette.o
calculette.o : calculette.cpp
     $(CPP) -c calculette.cpp
clean :
     rm *.o
```

   puis à la console faites `make all` (ou simplement `make`).

   Vous pourrez alors executer `./calculette`.

2. Faites ensuite `make clean` pour appliquer la règle qui efface les fichiers intermédiaires produits.

Remarques :
— en général lors de l'édition de liens, il faut lister tous les fichiers d'extentions `.o` utilisés en les séparant par un espace.
— et lors de la compilation d'un fichier `.cpp`, les dépendances portent à la fois sur le fichier d'en-tête `.hpp` et sur le fichier de code `.cpp`

Exercice 3 (fichier d'en-tête)
On rappelle que le sinus de la bibliothèque `cmath` est défini pour un angle exprimé en radians et que les valeurs des angles se calculent proportionnellement en fonction de l'unité : $[0; 2\pi[$ en radians, $[0; 360[$ en degrés, $[0; 400[$ en grades.

Vous pouvez utiliser la constante `M_PI` de la bibliothèque `cmath`.

1. Dans un fichier `trigo.cpp`, écrivez une fonction `sinus(double, char)` à deux arguments permettant de calculer le sinus d'un angle donné dans une unité parmi {degrés, radians, grades}.
2. Comment garantir que les paramètres passés à la fonction `sinus` ne seront pas modifiés ?
3. Vous pouvez paramétrer votre fonction pour que la valeur par défaut soit le radian. Vérifiez en appelant `sin(M_PI/2)`.
4. On veut pouvoir appeler les fonctions définies dans le fichier `trigo.cpp` depuis des fonctions définies dans d'autres fichiers. Pour cela, créez un fichier d'en-tête `trigo.hpp`.

   Pour éviter des problèmes de compilation dus à des inclusions multiples, commencez le fichier d'en-tête par la directive de compilation suivante :

```
#ifndef TRIGO
#define TRIGO
```

   Et finissez par

```
#endif
```

5. Dans un fichier `triangle.cpp`, écrivez une fonction `autreCoté` qui prend en argument un angle $(\overrightarrow{AB}, \overrightarrow{AC})$, une unité d'angle dont la valeur par défaut est le radian et une longueur AB. On supposera le triangle rectangle en B. Votre fonction retournera la longueur BC. (Naturellement vous utiliserez la fonction *sinus* précédente) Pensez à préciser si les arguments peuvent être considérés constant.

Exercice 4 Environnements de travail

1. **Avec emacs :** Sous `emacs` en appuyant simultanément sur Alt et x, puis en complétant le mini-menu avec `compile`, vous pourrez ensuite lancer la commande `make` sans avoir à aller dans la console.
2. **Avec codeblocks :** Pour une compilation simple, en reprenant le premier exercice, lancez `codeblocs calculette.cpp &` puis appuyez sur l'icone de compilation/exécution (flèche verte avec un engrenage). Vous pouvez aussi créer un nouveau projet, puis y créer de nouvelles classes. Reprenez votre second exercice comme si vous l'aviez développé avec Codeblocs. (Il est possible qu'il vous faille paramétrer le compilateur pour utiliser la version c++11)
3. **Avec netbeans :** Netbeans est également installé sur nos machines. Reprenez encore le second exercice pour vous faire la main avec netbeans en créant un nouveau projet reprenant l'exercice 2.
4. Après cet aperçu vous pourrez avoir une préférence pour travailler. Vous pouvez naturellement préférer d'autres environnement, mais nous ne fournirons de support pour aucun. Pour mémoire : si vous devez rendre un devoir celui ci doit être compilable avec un make adapté à toute machine ...

2 Un peu de pratique

Exercice 5 (Classes)
Vous pouvez vous inspirer de la classe `Point` vue en exemple dans le cours.

On cherche à écrire une classe `TimeStamp` qui modélise une durée dans le format heure, minute, seconde.

1. Ecrivez son en-tête `TimeStamp.hpp` en utilisant 3 attributs. Dans `Test.cpp` vous aurez un `main` qui testera la création d'un objet et son affichage. Prévoyez les méthodes utiles ainsi qu'un constructeur à trois arguments. Assurez vous d'écrire séparément dans `TimeStamp.cpp` les réalisations. Vous utiliserez un fichier `Makefile` pour effectuer la compilation. Pensez à `#include <iostream>`, et éventuellement à `using namespace std;`
2. Rendez le constructeur robuste dans le sens où les valeurs transmises pour les minutes et secondes puissent être non naturelles (c.a.d hors de l'intervalle [0..60[) Vous passerez donc d'abord toutes ces valeurs en secondes avant de refaire le calcul modulo 60 etc ... Si la durée globale est négative vous la réduirez à 0.

   On ne doit pas pouvoir modifier les valeurs des attributs des objets : plus que privés, ils doivent être déclarés constant. Cette contrainte complique un peu l'initialisation,

le but de cette question est que vous vous en rendiez compte (et vous pouvez vous en sortir en utilisant une fonction intermédiaire)

Exercice 6 Pour cet exercice, **n'utilisez pas** `using namespace std;`, qui peut induire un comportement inattendu.
Dans tout l'exercice, les éléments qui sont additionnés doivent être déclarés constants en argument des fonctions.

1. Écrivez deux fonctions appelées `plus` permettant de calculer :
   (a) la somme de deux `int` et renvoyant un `int`,
   (b) la somme de deux `double` et renvoyant un `double`.

   Appelez `plus` en passant : deux `int`, un `int` et un `short`, deux `float`, deux `double`, un `int` et un `double`. Que dit le compilateur ? Pourquoi ?
2. Écrivez deux fonctions appelées `somme` qui permettent de calculer la somme des éléments d'un vecteur d'`int` pour l'une et de `double` pour l'autre. Appelez la fonction `somme` en passant : un vecteur d'`int`, un vecteur de `short`, un vecteur de `double`. Que dit le compilateur ? pourquoi ?
3. Faites-vous le même constat si vous définissez deux fonctions `somme` supplémentaires qui permettent de calculer la somme des éléments d'un tableau d'`int` pour l'une et de `double` pour l'autre ?

A faire chez vous

Installez l'un des environnements C++ sur votre ordinateur personnel, assurez vous d'avoir le standard c++11.