

TP n°1

Introduction à C++

1 Prise en main

Exercice 1 (Un programme Simple)

Créez un répertoire `ObjetsAvances/TP1/Calcullette`, puis écrivez (dans un éditeur de votre choix) un programme `calcullette.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 2 de ce TP vous présente d'autres environnements de travail que vous pourrez préférer plus tard, mais il vous faut de toutes façons savoir configurer un **Makefile**.

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 rafraîchi les dépendances si c'est nécessaire.

1. Dans le répertoire de votre calcullette 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 exécuter `./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'extensions `.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, créer un fichier d'en-tête `trigo.hpp` (ou `trigo.h`).

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 `hauteur` 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 et retourne la hauteur du triangle ABC issue du sommet B . La fonction `hauteur` invoquera la fonction `sinus`.
Pensez aux arguments constant.

2 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 `codeblocks calcullette.cpp &` puis appuyez sur l'icône 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 Codeblocks. (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 également le second exercice pour vous faire la main avec netbeans en créant un nouveau projet reprenant l'exercice 2.

3 compatibilités de types

Exercice 4 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 ?

4 Si vous avez le temps...

Exercice 5 (Classes)

Un fichier d'**en-tête** d'une classe est un fichier `MaClasse.hpp`, qui ne contient que les inclusions de bibliothèque, les déclarations des attributs et les prototypes des méthodes; la **définition**, c'est à dire le code des méthodes, sera mise dans un fichier `MaClasse.cpp`.

1. On cherche à écrire une classe qui modélise un point du plan par ses coordonnées. Écrivez trois fichiers : `Point.hpp` d'en-tête, `Point.cpp` de réalisation, et `Test.cpp` où vous aurez un `main` pour tester la création d'un objet et son affichage. Pour cette question votre classe se réduira à :
 - un constructeur à deux arguments,
 - des accesseurs,
 - des modifieurs,
 - une méthode d'affichage.

Vous utiliserez un fichier `Makefile` pour effectuer la compilation.

Pensez au couple `#include <iostream>`, `using namespace std;`

2. Ajoutez une méthode `distance(Point)` qui retourne la distance entre deux points (indication : vous utiliserez `#include <cmath>`). Vérifiez en calculant la distance du point de coordonnées 1,1 à l'origine.

A faire chez vous

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