

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, c'est à dire que chaque portion de code est compilée de son côté, en laissant ouverts/suspendus les liens vers le code des autres modules. L'exécutable est produit lors d'une dernière opération de synthèse, où les liens peuvent être établis par le compilateur, puisque les portions de code concernées sont alors disponibles.

L'intérêt est qu'il sera inutile de recompiler des modules 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 voudrions absolument pas quelque chose dépendant de votre machine ou de vos outils. Vous nous rendrez donc un Makefile écrit par vos soins, compréhensible.**

On rappelle que 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 calculatrice créez un fichier **Makefile** contenant :

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

all :   calculatrice

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

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

Vous pourrez alors exécuter **./calculatrice**.

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.
- lors de la compilation d'un fichier **XX.cpp**, les dépendances portent à la fois sur le fichier d'en-tête **XX.hpp** et sur le fichier de code **XX.cpp**, mais aussi sur les **YY.hpp** que vous incluez !
- un exemple plus approfondi a été présenté dans le cours, avec une syntaxe un peu plus évoluée, des astuces de debugages ... Vous arriverez petit à petit à cette forme plus avancée au fur et à mesure de vos besoins.

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 **d** : pour degrés, **r** : pour radians, **g** : pour grades.
2. Comment indiquer (pour optimisation) que les paramètres passés à la fonction **sinus** ne seront pas modifiés par son code ?
3. Vous pouvez paramétrer votre fonction pour que la valeur par défaut soit le radian, c'est à dire qu'il ne sera pas obligatoire de préciser l'angle. 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, on veut mettre en place l'architecture d'une bibliothèque : créez un fichier `trigo.hpp` d'en-tête.

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 AC . 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

Vous aurez chacun vos préférences (différentes) et il nous est impossible de vous fournir un support.

Dans les dépôts de vos travaux, pour la communication avec vos enseignants, nous ne prendrons en compte que des codes associés à un Makefile simple et compréhensible. Imaginez que nous utiliserons emacs et tout ira bien.

Pensez à paramétrer le compilateur associé pour utiliser la version 11 de c++

1. **Avec emacs :** Tips : 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. **codeblocks, netbeans, eclipse, intellij ...** les deux premiers sont installés sur les machines de l'ufr, pour les autres c'est moins sûr. Ces IDE vous permettent de vous concentrer sur le code, ils simplifient la gestion des projets, la correction de la syntaxe etc. Mais ils produisent leur propre Makefile, alors si vous les utilisez ayez bien à l'esprit qu'il vous faudra écrire un Makefile intelligible.
3. L'an dernier un sondage en TP a montré que beaucoup d'entre vous utilisent **vscode**. Voici quelques petit conseils :
 - Pour contourner en partie la télémétrie de microsoft probablement qu'il est plus raisonnable d'utiliser **vscodium**
 - Installez ensuite les extensions : "C/C++ runner" et "Make support and task provider".
 - Naturellement vous serez tentés d'installer une IA comme **codeium**. Soyez conscient que ces IAs vous donnent un faux sentiment de puissance. En ce qui vous concerne, pour ces TPs, il faut toujours que vous vous demandiez si sans elle vous seriez capable d'écrire le code demandé. Si ce n'est pas le cas c'est que votre apprentissage est superficiel. Le code doit sortir de votre esprit et pas être soufflé d'on ne sait où. A l'écrit vous n'aurez pas cet oracle. A l'oral

vous ne l'aurez pas non plus, et quel que soit la qualité du code d'un projet rendu ce sera la défense que vous en ferez qui rendra crédible votre niveau, et c'est cela que nous évaluerons... Bref ne l'utilisez pas en TP.

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. Écrivez 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 car la fonction "plus" y est déjà définie ...

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 (à la c) d'`int` pour l'une et de `double` pour l'autre ?