TP n° 1 de C++ - 2 séances

Partie 1 : ++C

- Ecrire un petit programme qui affiche « hello world » sur la console avec **cout** défini dans l'entête iostream. Utiliser le compilateur g++ (et non gcc).
- Utiliser le prototype de main() incluant les paramètres classiques pour récupérer les arguments de la ligne de commande. Compiler avec les options –Wall –ansi –pedantic -Wextra pour afficher tous les warnings et s'assurer que l'on n'est pas prétentieux par rapport au standard ANSI C++.
- Enlever les deux warnings en utilisant les **paramètres muets** du C++.
- Ecrire une méthode swap() qui permet d'échanger le contenu de deux variables de type entier en utilisant le passage de paramètre par adresse. Tester et valider dans le main().
- Ecrire une autre méthode swap() qui permet d'échanger le contenu de deux variables entières passées **par référence**… Remarquez que la fonction peut porter le même nom grâce à la **surcharge**. Tester et valider dans le main().
- Ecrire une dernière fonction swap() permettant d'échanger par référence deux variables réelles. Tester et valider dans le main(). On verra (beaucoup) plus tard un moyen d'écrire une telle fonction d'échange pour tout type d'opérandes.

Partie 2 : POO

Cette partie permet de tester les concepts basiques de la POO en C++

- Ecrire une classe mère *Mere* sans attribut. Instancier un objet dans le main().
- Ecrire le constructeur par défaut de la classe. Le constructeur affiche à l'écran un message.
- Doter la classe d'un attribut entier *id*, modifier le constructeur pour que la valeur *id* soit donnée en paramètre et que le constructeur affiche *id*. Instancier deux objets avec des valeurs différentes.
- Doter le constructeur d'une valeur par défaut : 0 par exemple.
- Vérifier la portée de *id* (utilisation directe dans le main) en le rendant successivement public, protected et private.
- Pour respecter l'encapsulation, placer *id* en private/protected suivant le dosage d'encapsulation que vous souhaitez et ajouter un « getter » et un « setter ».
- Ajouter un compteur d'instance *compteur* (attribut de classe) initialisé à 0 et incrémenté à chaque nouvelle création d'instance. Vérifier que le comportement est conforme à celui attendu.
- Ecrire une méthode de classe qui renvoie la valeur de *compteur* pour appliquer le principe de l'encapsulation.
- Ecrire une nouvelle classe : *Fille* qui hérite de *Mere*.
- Vérifier qu'un objet de classe Fille permet d'accéder aux méthodes définies précédemment.
- Doter les classes d'une méthode whoami(), qui affiche le nom de la classe. Vérifier que l'affichage est correct.

Partie 3 : Constructeurs et destructeurs

Voici un petit programme. Le tester et étudier ce qu'il affiche exactement à l'écran. Repérer les lignes qui interviennent à chaque fois.

```
#include <iostream>

class A
{
   public:
      A(void)         { std::cout << "A()"  << std::endl; }
      A(const A &)    { std::cout << "A(A)" << std::endl; }
     ~A(void)         { std::cout << "~A()" << std::endl; }
 };

class B : public A
{
   public:
      B(void)         { std::cout << "B()"  << std::endl; }
      B(const B &)    { std::cout << "B(B)" << std::endl; }
     ~B(void)         { std::cout << "~B()" << std::endl; }
 };

int main(int, char **)
{
   A a1;
   B b1;
   A a2(a1);
   A a3 = a1;
   B b2(b1);
   A a3(b1);
   A ta[5];
   B tb[3];

   std::cout << "Ouf ! Mais est-ce fini ? ! ?" << std::endl;

   return 0;
}
```

Partie 4 : Héritage multiple et en diamant

Donner la trace du programme suivant :

```
#include<iostream.h> // on peut aussi
using namespace std; // facilité

class A
{
   public:
      A() { cout << "A ";}
     ~A() { cout << "~A ";}
};

class B : public A
{
   public:
      B() { cout << "B "; }
     ~B() { cout << "~B ";}
};

class C : public A {
 public:
   C() { cout << "C ";}
  ~C() { cout << "~C ";}
};
```

```
class D : public B, public C
{
   public:
      D() { cout << "D ";}
     ~D() { cout << "~D ";}
};

int main(int, char**)
{
   D d;

   cout << "c'est fini !";

   return 0 ;
}
```

Que faut-il faire pour que le constructeur de A soit appelé un nombre minimal de fois ?

Partie 5 (Obligation : prévoir 2 fichiers pour chaque classe)

1) Concevez une classe Voiture caractérisée par, au minimum, sa marque, sa couleur, sa puissance (din), son régime moteur maximal (trs/min), sa consommation (urbaine, et extra-urbaine au 100 km), la capacité de son réservoir, le contenu du réservoir et son émission de C02 (en g/km).
   Définissez des méthodes pour accéder aux attributs et proposez quelques méthodes pour : faire le plein, faire rouler une voiture sur une distance donnée et sur un type de parcours (ville, route de campagne, autoroute).

2) Prévoir un programme principal pour instancier plusieurs voitures : saisir (et donc préciser les attributs de chaque voiture) et faire rouler les voitures pendant un nombre donné de kilomètres (ville et zone urbaine), puis afficher ce qu'il restera dans leur réservoir en fin de programme ainsi que la quantité de CO2 émise par l'ensemble de ces voitures.

3) Dans le cas de votre programme sur les voitures, quelle serait la solution pour obtenir simplement le nombre de voitures créées (et pour le mémoriser). Implémenter votre solution.

4) Réutilisation de classes. Considérons une extension de ce programme : ajouter une classe Parking que vous concevrez et où vous stockerez les voitures. Considérez une allocation statique et une allocation dynamique pour stocker les voitures. Reconsidérez la question précédente sachant que vous disposez de la classe Parking. Dans le programme principal, stocker les voitures créées dans le parking lors de leur création, puis lister toutes les voitures présentes dans ce parking avec leurs principaux attributs.

Partie 6 – Prévoir 5 fichiers : 2 pour chaque classes plus un pour le programme principal. Compilation séparée avec un petit makefile (suivre l'exemple du poly).

Envoi de messages (appel de méthodes) entre 2 classes – cet exercice suppose que l'on consulte les conseils de codage en C++ du polycopié.

Décrire deux classes **A** et **B**. La classe **A** possède un entier i, et la classe **B** un entier j. Ces deux classes ont chacune une méthode « `exec` » et une méthode « `send` » qui leur permet d'envoyer un message à un objet de l'autre classe. La méthode « `send` » de la classe **A** accepte un pointeur sur un objet de classe **B** et réciproquement. La méthode « `exec` » de chaque classe accepte un entier en paramètre et ajoute la valeur de cet entier aux attributs i ou j selon la classe de l'objet concerné (**A** ou **B**). L'exécution du corps d'une méthode « `send` » lance un « `exec` » sur l'objet *distant* avec une constante de votre choix. Ainsi `unA.send(&B)` active la méthode « `send` » de la classe **A** qui lance la méthode `exec` de la classe **B**. Indépendamment des imbrications de méthodes, le but est de mettre en évidence l'utilisation systématique des déclarations anticipatives dans les entêtes des classes communicantes.

OPTIONNEL :

Pour les informaticiens qui s'ennuient trop avec les 5 parties précédentes…

Adaptation et réutilisation du code sur les voitures :

Considérons une classe Paddock qui possède un ensemble de véhicules de type formule 1, mais radiocommandée et sans boite de vitesse. Chaque véhicule précise de manière tabulée (tous les 1000 tr/min) la vitesse obtenue en fonction du régime moteur. (Ex : 1000 – 10 km/h ; 2000 – 40 km/ h ; … 14 000 – 350 km/h). On considère non plus une consommation pour 100km mais une consommation en nombre de litres à l'heure. Considérons également une classe Circuit qui comporte un nom et une distance en mètres ainsi que ses caractéristiques sous forme tabulée. Le tableau donne des couples du type : (distance en mètres, régime maximal), la somme des distances de tous les couples donne la distance totale du circuit. Considérons maintenant une classe Course qui simule une course avec l'ensemble des véhicules présents dans la classe Paddock pendant une durée fixe (en minutes) et sur un circuit donné, en fonction de leurs performances (chaque véhicule présentant des caractéristiques différentes). Le programme principal effectuera une compétition sur au moins 2 circuits avec au moins 2 véhicules différents. Vous prendrez les choix de conception qui vous sembleront pertinents (prise en compte des pannes d'essence, classement des véhicules, évolution du temps, etc…)