Aix-Marseille Université
Master 2 F.I.

Introduction au C++ - TP

A) Préambule
Ce TP a pour but de vous faire manipuler des classes C++ ainsi que les différentes notions associées expliquées en cour. Pour cela vous allez partir d'une classe de base à laquelle vous aurez à ajouter diverses fonctionnalités. Dans un second temps, vous verrez comment étendre cette classe et la faire communiquer avec d'autres classes que vous développerez aussi.
L'ensemble des sources, ainsi que les fichiers "projets" utilisés par `Code::Blocks` (**TP1.cbp**, **TP2.cbp** et **TP3.cbp**), nécessaires à ce TP peuvent être récupérés à l'adresse :

`https://github.com/GFuhr/M2_FI`

Il faut ensuite cliquer sur le bouton vert "clone or download" à droite puis sur "Download ZIP".

Partie 1 : La classe de base - la classe *Capteur*

B) Introduction
La classe ***Capteur*** est une classe simple représentant un capteur quelconque qui mesure une grandeur physique (par exemple un capteur de température, humidité etc...) ainsi qu'un chiffre représentant son identité (afin de différencier les capteurs entre eux). La déclaration de la classe est la suivante :

```
//classe Capteur
class Capteur {
    int  id ;
    double  value;

public:
    Capteur(void) ;
    Capteur(const int iid, const double dvalue=-1);
    Capteur(const Capteur &p);
    ~Capteur(void);

    void set(const double val);
    double get(void) const;
    double ecart(const Capteur &p) const;
    void cout_id(void) const;
};
```

La déclaration de la classe se trouve dans le fichier *Classe_ capteur.h*.

C) Définition des fonctions membres de la classe
Remarque : Votre code source ne pourra être compilé qu'une fois la fin de cette partie atteinte. La définition des divers constructeurs, fonctions... de la classe ***Capteur*** seront à faire dans le fichier **classe_capteur.cpp**.

1. Définissez le constructeur par défaut tel que la valeur *value* soit égale à -1 et l'id à -1 aussi.
2. Ajoutez ensuite à ce constructeur la ligne suivante :

```
std::cout << "appel du constructeur de capteur par defaut" << std::endl;
std::cout << "pour l'objet :"<<(long)this<<std::endl;
```

3. Ajoutez la ligne suivante au corps du destructeur :

```
std::cout << "appel du destructeur de capteur " << std::endl;
std::cout << "pour l'objet :"<<(long)this<<std::endl;
```

4. Dites ce que permettent ces 2 instructions et quels sont les intérêts pratique lors de l'éxécution du programme ?
5. Définissez le constructeur avec argument, permettant de spécifier le valeur de l'identité via la variable *iid* ainsi que, au choix, une valeur initiale pour la mesure du capteur
6. Définissez le constructeur par copie, permettant de copies les données membres de l'objet passé en argument (p) dans l'objet actuel.
7. Ajoutez pour chaque constructeur (constructeur avec arguments et constructeur par copie) une ligne permettant d'afficher à l'écran quel constructeur est appelé :

```
// constructeur avec arguments
    std::cout << "constructeur avec argument pour la classe capteur de l'objet ["
              << (long)this << "]" << std::endl;
// constructeur par copie
    std::cout << "constructeur de capteur par copie depuis [" << (long)&p << "]
      vers le nouvel objet [" << (long)this << "] " << std::endl;
```

8. Les donnés membres actuelles de la classe sont-elles définies de manière statique ou dynamique ? En conséquence, devons nous faire autre chose dans le destructeur associé de la classe ?

D) Utilisation dans la fonction principale ***main***

Maintenant que nous avons déclaré et défini notre classe, nous allons voir comment l'utiliser. Pour cela nous allons nous intéresser à notre fonction *main* afin de créer plusieurs capteurs. La fonction *main* à modifier se trouve dans le fichier **main_Capteurs.cpp**. Écrivez votre fonction ***main*** de telle sorte qu'elle contiendra 5 capteurs.

1. un capteur A qui utilisera le constructeur par défaut
2. un capteur B qui utilisera le constructeur avec arguments
3. un capteur C qui utilisera le constructeur par copie pour faire une copie de B
4. Un capteur D qui sera une référence sur le capteur C
5. Un capteur E qui sera défini via un pointeur sur un capteur et l'opérateur new
6. Que constatez vous entre le nombre d'appel aux destructeurs et le nombre d'appel aux constructeurs ? Et concernant l'ordre des appels ?

E) Fonctions d'interface

1. Définissez les fonctions : ***get()*** qui renvoie la valeur *_value*, et ***set()*** qui permet de modifier la valeur de *value*

2. Définissez la fonction ***cout_id()*** qui permet d'afficher la donnée membre ***id*** du capteur

3. Définissez la fonction ***ecart()*** qui permet de calculer l'écart de valeurs entre 2 capteurs.

F) Utilisation des fonctions membres

1. Utilisez dans la fonction ***main()*** la fonction membre ***get()*** pour afficher les valeurs des capteurs B et E.
2. Utilisez dans la fonction main la fonction ***set*** pour modifier les valeurs des capteurs C et E.
3. La modification de la valeur du capteur C a-t-elle entrainé une modification de la valeur du capteur D ? Expliquez.
4. et du capteur B ? Expliquez aussi
5. Utilisez dans la fonction ***main()*** la fonction membre ***ecart()*** pour calculer les écarts entre les capteurs suivants : A et B, C et D, E et A, E et D. Affichez les résultats obtenus et vérifiez que les valeurs sont correctes.
6. Que devez vous changer dans la syntaxe d'appel de votre fonction membre lorsque vous l'utilisez depuis l'objet E ? Devez vous changer la syntaxe d'appel lorsque vous utilisez une fonction membre depuis l'objet D ?
7. Si, dans la déclaration et la définition de votre fonction ***ecart***, vous supprimez l'utilisation des références, cela va-t-il changer le code de la définition de votre fonction ? et au niveau de l'utilisation, remarquez vous une différence ?

   ```
   double ecart(const Capteur &p);
   ```

   devient,

   ```
   double ecart(const Capteur p);
   ```

G) Gestion de la mémoire et tableaux

1. Dans la fonction ***main***, créez un tableau F de 3 capteurs (avec *new*) et détruisez le avec l'opérateur "[] delete". Que remarquez-vous ?
2. Pour vérifier si la mémoire a bien été libérée, déclarez un nouveau tableau G de 3 capteurs après l'appel au destructeur de F.
3. Affichez l'adresse de ce tableau, de son 1ier élément ainsi que celle des éléments F[0], F[1] et F[2], dans le cas où F a été détruit via *// delete*.
   Remarque : pour afficher l'adresse de l'élément F[idx] utilisez l'opérateur (&) de la manière suivante avec *cout*

   ```
   std::cout<<"adresse de F[idx] :"<<(long)&(F[idx])<<std::endl
   ```

   Et pour afficher l'adresse de F, utilisez la syntaxe :

   ```
   std::cout<<"adresse de F :"<<(long)F<<std::endl
   ```

4. Dans la fonction ***main***, créez un tableau H de 3 ***Capteur*** (avec les vectors). Que remarquez-vous, en particulier au niveau des constructeurs/destructeurs ? Déduisez-en comment se passe la construction du vector de capteurs.

H) Nouvelle fonctionnalité

Nous allons chercher à ajouter un autre élément à notre classe, afin de savoir le nombre de lectures de la valeur du capteur.

1. ajoutez une variable membre pour stocker le nombre de lectures. Quel doit-être le type d'accès à cette variable membre : public, private, protected ?
2. Ajoutez une fonction membre permettant de renvoyer le nombre de lectures.

Partie 2 : Notion d'héritage

I) Préambule

Dans cette seconde partie, nous allons partir de cette classe ***Capteur*** pour générer 2 classes filles correspondant à des capteurs analogiques et digitaux. Pour différencier ces 2 nouveaux types, nous supposerons simplement que un capteur analogique va lire une tension en Volts, représentée par un nombre décimal, et le capteur digital une valeur en bits, représentée par un entier. Ces nouveaux capteurs étant dérivé de la classe ***Capteur***, il faut ensuite utiliser ces valeurs lues pour les convertir dans la valeur correspondant pour la donnée membre ***value*** de la classe ***Capteur***. Nous allons dans cette partie déclarer ces nouvelles classes ainsi que les fonctions associées. Afin de simuler la lecture de valeurs, vous pouvez utiliser les 2 fonctions *randint* et *randdouble* qui renvoient chacunes une valeur aléatoire de type *int* pour la première et *double* pour la seconde. Ces valeurs sont comprises entre 0 et la valeur mise en argument, ou bien par défaut :

- 0-1024 pour randint
- 0-100 pour randdouble

Remarque générale :

- Pour cette partie, vous utiliserez le projet TP2 dans `Code::Blocks`.
- Pour chaque constructeur/destructeur que vous définirez dans la suite, vous ajouterez une ligne permettant d'afficher à l'écran quel est le constructeur employé, comme pour la partie 1.
  Exemple pour le constructeur par défaut de la classe ***CapteurDigital***, vous ajouterez :
  ```
  std::cout << "constructeur avec argument pour la classe capteur digital ["
            << (long)this << "]" << std::endl;
  ```
- Toutes les définitions devront être faites dans le fichier **Classe_derived.cpp** et les déclarations des fonctions membres supplémentaires dans le fichier **Classe_derived.h**

Rôle des données membres :

1. CapteurAnalogique
   (a) ***volts*** : valeur "lue"
   (b) ***vmin*** : valeur minimale possible
   (c) ***vmax*** : valeur maximale possible
   (d) ***range*** : plage de valeur correspondant.
2. CapteurDigital
   (a) ***bits*** : valeur "lue" en bits
   (b) ***bitsmax*** : valeur maximale possible
   (c) ***range*** : plage de valeur correspondant.

J) Déclaration de la classe CapteurAnalogique et de ses membres

1. Définissez les constructeurs suivants dans le fichier **Classe_derived.cpp** :
   - Un constructeur par défaut pour lequel la plage de valeur sera de 10 pour une tension comprise entre 0V et 3.5V.
   - Un constructeur pour lequel on spécifiera l'id du capteur, la plage de valeurs ainsi que les valeurs minimales et maximales possibles.
   - Un constructeur par copie.

   Dans chaque constructeur, toutes les données membre doivent être initialisées.
2. Est ce qu'un appel au constructeur de la classe ***CapteurAnalogique*** entraîne automatiquement un appel à un des constructeur de la classe ***Capteur*** ?Si oui lequel ? Pourquoi ?

3. Définissez le destructeur.

K) Déclarations et Définitions des fonctions annexes

1. Définissez la fonction ***read*** qui permet de simuler une lecture du capteur, et aussi de calculer la valeur en conséquence pour la donnée lue. Par exemple, pour un capteur analogique avec une plage de valeurs possibles de 0 à 5V correspondant à des valeurs physiques de 0% à 100%, une valeur lue de 2V correspond ainsi à une valeur physique de 40%. Ainsi, si dans la fonction ***read*** vous lisez une valeur de 2V, alors la donnée membre ***value*** associée doit alors valoir :$2 * 200/5 = 40$.

L) Déclaration et définition de la classe ***CapteurDigital*** et de ses membres

1. Définissez les constructeurs suivants dans le fichier **Classe_derived.cpp** :
   — Un constructeur par défaut pour lequel la plage de valeurs possible sera de 10 pour un bus de 8bits, donc une valeur ***bitsmax*** de 256.
   — Un constructeur pour lequel on spécifiera l'id du capteur, la plage de valeurs ainsi que la valeur maximale possible en lecture.
   — Un constructeur par copie qui utilisera le constructeur par copie de la classe capteur.
   Dans chaque constructeur, toutes les données membre doivent être initialisées.
2. Définissez le destructeur
3. Définissez la fonction ***read*** qui permet de simuler une lecture du capteur, et aussi de calculer la valeur en conséquence pour la donnée lue. Par exemple, pour un capteur digital avec une plage de valeurs possibles de 0 à 256 correspondant à des valeurs physiques de 0°C à 100°C, une valeur lue de 128 correspond ainsi à une valeur physique de 50°C. Ainsi, si dans la fonction ***read*** vous lisez une valeur de 128, alors la donnée membre ***value*** associée doit alors valoir :$128 * 100/256 = 50$.

M) Création de capteurs

1. Vous créerez dans votre fonction principale ***main*** (située dans le fichier **main_capteurs_derived.cp** plusieurs capteurs afin d'utiliser les fonctions définies dans vos nouvelles classes :
   — Un capteur générique utilisant la classe ***Capteurs*** de base.
   — Un ***CapteurAnalogique*** utilisant le constructeur par défaut
   — Un ***CapteurAnalogique*** représentant un capteur de température pouvant mesurer des valeurs de 0 à $80°C$ avec une plage de valeurs de $0V$ à $5V$.
   — Un ***CapteurAnalogique*** fait par copie du capteur précédent.
   — Un ***CapteurDigital*** utilisant le constructeur par défaut
   — Un ***CapteurDigital*** représentant un capteur d'humidité pouvant mesurer des valeurs de 0% à 100% utilisant un bus $10bits$ donc une valeur maximale possible de 1024.
   — Un ***CapteurDigital*** fait par copie du capteur précédent.
   — Pour chacun des cas précédent, vous ferez une lecture du capteur et afficherez la valeur mesurée.
   — Est ce qu'un appel au constructeur de la classe ***CapteurDigital*** entraîne automatiquement un appel à un des constructeur de la classe ***Capteur*** ?et à un constructeur de la classe ***CapteurDigital*** ? Expliquez pourquoi.
   — Et concernant les destructeurs ?

Partie 3 : Tableaux et Composition

N) Classe Infrastructure

En général dans le cadre d'une utilisation type domotique par exemple, il n'y aura pas qu'un seul capteur utilisé mais plusieurs et de divers types : température, humidité, lumière etc... Nous allons considérer ici le cas d'un ensemble de capteurs de température et d'humidité, le nombre sera à définir au sein d'une classe ***Infrastructure***.

Cette classe contiendra plusieurs capteurs de température et d'humidité, représenté par 2 vectors, ***temperature*** et ***humidity***. De manière pratique, pour cette dernière partie, vous utiliserez le projet TP3 dans `Code::Blocks`.

Remarque : Contrairement aux parties précédentes, l'utilisation de la notion de constantes (mot-clé "const") et des références est à indiquer, aussi bien pour les déclarations que pour les définitions.

1. Déclarez les données membres de la classe ***Infrastructure*** :
   - un vector de ***CapteurAnalogique*** pour stocker les diverses valeurs d'humidité, dans une donnée membre ***humidity***.
   - un vector de ***CapteurDigital*** pour stocker les diverses valeurs de température, dans une donnée membre ***temperature***.
   - une variable ***nombre_ca*** contenant le nombre de capteurs analogiques
   - une variable ***nombre_cd*** contenant le nombre de capteurs digitaux
2. Définissez ensuite les divers constructeurs pour la classe :
   - Un constructeur par défaut qui correspond au cas d'une installation sans aucuns capteurs.
   - Un constructeur surchargé pour lequel on spécifiera le nombre de capteurs digitaux et analogiques.
   - Un constructeur surchargé pour lequel on copiera les capteurs depuis 2 tableaux de capteurs analogiques et digitaux, en spécifiant aussi la taille de chaque tableau.
   - Un constructeur surchargé pour lequel on copiera les capteurs depuis 2 vectors transmis en argument.
   - Le constructeur par copie
   - Un destructeur
   - Une fonction membre qui affichera la température moyenne et l'humidité moyenne.
   - Une fonction membre qui renvoie le nombre total de capteurs.
   - Une fonction membre qui affiche les valeurs mesurées par tous les capteurs.

O) Utilisation

1. Une fois cette classe définie, créez une infrastructure contenant 4 capteurs de température et 2 capteurs d'humidité de 3 manières différentes (vous aurez donc 3 variables) :
   - en spécifiant le nombre d'éléments lors de l'appel au constructeur.
   - en créant 2 vectors (1 pour le capteur analogique et 1 pour le capteur digital) que vous passerez en argument.
   - en créant 2 tableaux dynamiques que vous passerez en argument.
   - Sur ces 3 méthode, laquelle vous semble la plus simple à utilisez ? Dites pourquoi selon-vous.
2. Affichez les valeurs mesurées pour l'ensemble des éléments.
3. Affichez les valeurs moyennes et vérifiez qu'elles correspondent aux valeurs mesurées.
4. Créez une nouvelle infrastructure par copie de la précédente, et affichez les valeurs moyennes des capteurs. Avez vous les mêmes valeurs que précédemment ? Expliquez.

Annexe 1 : Vector et STL

Soit la variable $varV$ correspondant à un vector d'un type T .

```
std::vector<T> varV;
```

- spécification de la taille du tableau lors de la déclaration de la variable :
  `std::vector<T> varV(taille_tableau);`
- nombre d'elements dans le tableau associé :
  `varV.size();`
- modifier la taille du tableau associé :
  `varV.resize(new_size);`
- accéder à l'élément idx :
  `varV.at(idx);`
  `varV[idx];`
- ajouter ŭn élément "value" en fin de "vector" idx :
  `varV.push_back( value );`
- insérer plusieurs éléments à partir de la position idx :

```
std::vector<T> new_elements;
// inserer tous les éléments de new_elements dans le vector varV
varV.insert(varV.begin()+idx, new_elements.begin(), new_elements.end() );
// avec un tableau comme en C
// pour inserer tous les elements
T old_array[5];
varV.insert(varV.begin()+idx, old_array, old_array+4 );
```

Annexe 2 : Notation des sources

P) Préambule
Ce document liste les points importants qui serviront à évaluer les programmes écrits lors des TP. En priorité, le code doit compiler et s'éxecuter pour fournir les resultats attendus. Tous ces critères correspondent aux normes et bonnes pratiques de développement C++ pour la production de codes fonctionnels et réduisants le risque d'erreurs.
Ces critères vont être classés en 4 niveaux, avec un niveau de validé uniquement si toutes les conditions liées à ce niveau le sont.

Minimum requis pour la moyenne
- avoir un code qui compile sans erreurs ni warnings.
- ne pas utiliser de variables globales et/ou de "#define".

Niveau 1 : 10-12
- Utiliser les bons types d'accès pour les fonctions/données membres d'une classe : public, private ou protected.
- Utiliser les équivalent C++ de fonctions C *"classiques"*.

Niveau 2 : 12-14
- avoir fini la partie 2.
- Initialiser toutes les variables lors des déclarations.
- Faire une allocation dynamique plutôt que déclarer des tableaux statiques de taille disproportionnée.
- Commenter le code de manière utile.
- Libérer la mémoire (via **delete** ou **delete[]**) pour un bloc mémoire alloué via **new**, **new []**.

Niveau 3 : 15-17
- avoir fini le TP intégralement.
- utiliser au maximum les éléments de la STL (en particulier les fonctions associées aux std : :vector).
- Utiliser les références autant que possible, en particulier lors des déclarations de fonctions/-constructeurs.

Niveau 4 : 18-20
- Faire de tests sur les fonctions globales/les fonctions membres pour s'assurer des résultats fournis.
- S'assurer de la "const correctness".

Rappel : commentaire d'une fonction

```
/**
 * definition :  type_retour nom_fonction(type_1 const param1, type_2 param2);
 * description :  But de  fonction
 *
 * param  : param1: variable de type type_1 qui va servir a...,
                    non modifie par la fonction
 * param  : param2: variable de type type_2 qui va servir a...,
                   modifie par la fonction pour stocker tel r\'esultat
```

```
* valeur renvoyee : variable de type, type_retour, qui  contiendra telle valeur.
                    Il peut s'agir d'un code erreur, du resultat d'un calcul...
**/
```