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 a une certaine valeur (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 argvalue=-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 :"<<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 :"<<this<<std::endl;
```

4. 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
5. Définissez le constructeur par copie, permettant de copies les données membres de l'objet passé en argument (p) dans l'objet actuel.
6. 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 ["
              << this << "]" << std::endl;
// constructeur par copie
    std::cout << "constructeur de capteur par copie depuis [" << &p << "]
      vers le nouvel objet [" << this << "] " << std::endl;
```

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. Dans la partie *B* vous avez rajouté des instructions commencçant par "std : :cout", quel est leur intérêt pratique lors de l'éxécution du programme ?
7. Que constatez vous entre le nombre d'appel aux destructeurs et le nombre d'appel aux constructeurs ? Et concernant l'ordre des appels ?
8. Les donnés membres actuelles de la classe sont-elles définies de manière statique ou dynamique ? En conséquence, devons nous faire quelque chose de particulier dans le destructeur associé de la classe ?

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.

   — Utilisez dans la fonction main la fonction membre get pour afficher les valeurs des capteurs B et E.

— Utilisez dans la fonction main la fonction set pour modifier les valeurs des points C et E.
— La modification de la valeur du capteur C a-t-elle entrainé une modification de la valeur du capteur D ? Expliquez.
— et du capteur B ? Expliquez aussi
— 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 ;)
— 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 ?
— 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);
```

F) Gestion de la mémoire et tableaux

Nous allons voir que en C++, il existe au moins 3 manières de déclarer des tableaux, de manière statique, dynamique ou via les "vectors". Pour les syntaxes associées, regardez l'annexe 1.

1. Dans la fonction main(...), créez un tableau statique F de 3 capteurs. Un constructeur est il appellé ? et un destructeur en fin de programme ?
2. Dans la fonction main(...), créez un tableau dynamique G de 3 capteurs (avec new) et détruisez le avec l'opérateur "delete []". Que remarquez-vous ?
3. Pour vérifier si la mémoire a bien été libérée, déclarez un nouveau tableau dynamique ***H*** de 3 capteurs après l'appel au destructeur de G.
4. Affichez l'adresse de ***H***, de son 1ier élément ainsi que celle des éléments G[0], G[1] et G[2], dans le cas où G 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] :"<<&(F[idx])<<std::endl
   ```

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

   ```
   std::cout<<"adresse de F :"<<F<<std::endl
   ```

5. Dans la fonction main(...), créez un tableau T de 3 Capteurs (avec les vectors). L'utilisation de la notion de vectors entraîne l'appel de combien de constructeurs et destructeurs ?
6. Au niveau de l'utilisation, si vous deviez choisir entre vector et allocation dynamique, que choisiriez-vous ? Justifiez brièvement votre choix.

G) Nouvelle fonctionnalité

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

1. ajoutez une variable membre pour stocker le nombre de modifications. Quel doit-être le type d'accès à cette variable membre : public, private, protected ?
2. Modifiez les éléments nécessaires de la classe afin que le nombre de modifications soit incrémenté chaque fois que cela est nécessaire. Remarque : considérez que l'initialisation d'une variable constitue une modification.
3. Ajoutez une fonction permettant de renvoyer le nombre de modifications.

Partie 2 : Notion d'héritage

H) 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, représenté par un nombre décimal, et le capteur digital une valeur en bits, représenté par un entier.
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 :

- 0-256 pour randint
- 0-100 pour randdouble

Remarque générale :

- Pour cette partie, changez la partie *private* en *protected* dans la déclaration de la classe ***Capteur*** utilisée dans la partie 1.
- 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 ["
            << 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.

I) Déclaration de la classe CapteurAnalogique et de ses membres

1. Définissez les constructeurs suivants dans le fichier *Classe_ derived.cpp* :
   - Un constructeur par défaut
   - 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. Définissez le destructeur

J) 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%.

K) Création de capteurs analogiques

1. Vous créerez dans votre fonction principale *main* (située dans le fichier *main_ derived.cpp*) plusieurs capteurs afin d'utiliser les fonctions définies dans vos nouvelles classes :
   — 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.
   — Pour chacun des cas précédent, vous afficherez la valeur mesurée.
   — 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 ?
   — Répondez à la même question que précédement concernant les destructeurs de capteurs analogiques.

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
   — 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.

   Dans chaque constructeur, toutes les données membre doivent être initialisées.
2. Définissez le destructeur. Répondez à la même question que précédement concernant les destructeurs cette fois.
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 valeur de *bitsmax* de 1024, correspondant à des valeurs physiques de 0% à 100%, une valeur digitale de 256 correspond ainsi à une valeur physique de 25%.

M) Création de capteurs digitaux

1. Vous créerez dans votre fonction principale *main* (située dans le fichier *main_ derived.cpp*) plusieurs capteurs afin d'utiliser les fonctions définies dans vos nouvelles classes :
   — 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 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 ***CapteurAnalogique*** ?
   — Répondez à la même question que précédement concernant les destructeurs de capteurs digitaux.

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 implementer, aussi bien pour les déclarations que pour les définitions.
- Pour répondre aux différentes questions demandées, vous pouvez être ammenées à déclarer de nouvelles données membres et/ou de nouvelles fonctions membres.

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. Vous devrez dans ce cas simuler la lecture des valeurs des capteurs dès le constructeur.
   - 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 renvoie le nombre total de capteurs.
   - Une fonction membre qui affiche les valeurs mesurées par tous les capteurs.
   - Une fonction membre qui permet d'ajouter N capteurs analogiques et une fonction membre qui permet d'ajouter N capteurs digitaux
   - Une fonction membre qui affichera la température moyenne et l'humidité moyenne.

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. Les valeurs renvoyées par ces capteurs doivent être différentes des valeurs par défaut possibles.
   - en créant 2 tableaux dynamiques que vous passerez en argument. Les valeurs renvoyées par ces capteurs doivent être différentes des valeurs par défaut possibles.

— 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 : Tableaux en C++

Les tableaux, en informatique, font partie de la grande famille des structures de données. Ces structures de données sont des façons particulières d'organiser et de stocker des données. Les tableaux sont conçus pour stocker des données de même type et ayant des points communs : notes d'un élève à l'école, titres de livres, âge d'utilisateurs. Dès que vous pensez « liste de... », dès que des éléments ont des points communs et qu'il fait sens de les regrouper, alors l'usage d'un tableau peut s'envisager. Un tableau peut se voir comme un ensemble d'éléments de même type, stocké de manière contigu en mémoire et ayant une taille donnée.

1 tableaux statiques

Le cas le plus simple concerne les tableaux qualifiés de statiques, tableaux pour lesquels la taille est fixe et connue au moment de la compilation. La syntaxe générale est la suivante :

```
type_de_donnée nom[TAILLE];
```

L'argument TAILLE doit être obligatoirement un nombre entier. Exemple, pour déclarer un tableau de 10 double, qui va s'appeler montableau, nous devons écrire :

```
double montableau[10];
```

Ces tableaux sont faciles à déclarer mais ils ont 3 inconvénients :
- La taille allouée ne peut pas être modifiée sans recompiler le programme
- Le tableau ne peut pas être redimensionné
- Il n'y a pas de moyen de connaître la taille du tableau.

Dans un programme ou des tableaux sont utilisés, il est possible d'accéder à la valeur de chaque élément comme si il s'agissait d'une variable normale afin de pouvoir la lire et ou la modifier. Le format est le suivant :

```
nom_tableau[ indice_element ];
```

Pour revenir à notre tableau example, pour stocker la valeur 4.2 dans la case 1, il faut écrire :

```
montableau[1] = 4.2;
```

Remarque : Comme en C, les indices des tableaux commencent à 0. Ainsi un tableau de 10 éléments pourra être accédé via les indices $0, 1, \ldots, 9$.

2 tableaux dynamiques

Afin de compenser la limitation associée à l'allocation statique, il est possible d'effectuer une demande d'allocation mémoire de taille définie lors de l'éxécution du programme (et non plus lors de la phase de compilation). La taille de l'allocation corresponds à la taille du tableau désiré. Afin de créer le tableau en mémoire on utilisera l'opérateur ***new***. Cet opérateur est chargé d'allouer une zone mémoire pouvant contenir notre tableau et renvoie un pointeur vers cette zone mémoire du même type.
Ainsi si nous voulons, comme dans le cas précédent, déclarer et allouer un tableau contenant 10 doubles, nous devons faire :

```
double *nom_tableau=NULL;
nom_tableau = new double[10];
```

De manière générale, pour allouer un tableau de N éléments d'un certain type, nous devrons faire :

```
type_donnee *nom_tableau=NULL;
nom_tableau = new type_donnee[10];
```

Contrairement aux allocations statiques, si nous allouons la mémoire via l'utilisation de l'opérateur *new*, il faut ensuite la libérer via l'utilisation de l'opérateur delete[].

```
type_donnee *nom_tableau = new type_donnee[TAILLE];
//code
delete[] nom_tableau;
```

Comme pour les tableaux statiques, pour un tableau de taille N, les indices iront de 0 à $N-1$.

3 les vectors

Pour pallier les défauts inhérents à la rigidité des tableaux de taille fixe (built-in array), la librairie (générique) standard de C++ fournit un type de donnée dénommée vector (vecteur), offrant au programmeur un moyen très efficace pour construire des structures de données permettant de représenter des tableaux de tailles variables (i.e. tableaux dynamiques). La taille de ces «tableaux» n'est pas obligatoirement prédéfinie, et peut donc varier en cours d'utilisation. Pour pouvoir utiliser ces vecteurs dans un programme, il faut, comme dans le cas des entrées-sorties, importer les prototypes et définitions contenus dans la librairie, au moyen de la directive d'inclusion :

```
#include <vector>
```

Un vecteur peut être déclaré selon la syntaxe suivante :

```
vector < type_de_donnee > nom_de_la_variable(TAILLE, valeur_initiale)
```

Avec ***type_de_donnee*** qui correspond au type de base du tableau. Vous remarquez que la taille du vector peut-être indiquée à postériori. Exemple, pour un tableau de 10 doubles avec toutes les cellules initialisées avec la valeur 4.2.

```
std::vector<double> tableau_vector(10, 4.2);
```

ou alors

```
std::vector<double> tableau_vector(10);
nom_tableau = 4.2;
```

Comme pour les tableaux dynamiques ou statiques, il est possible d'utiliser les crochets pour accéder aux valeurs.

```
tableau_vector[3] = 5.4;
```

Le 4ieme élément du tableau (portant l'indice 3) aura la valeur 5.4.
Les vectors ont d'autres avantages, ainsi il est possible de rajouter des éléments à la fin de notre vector sans avoir à s'occuper explicitement de l'allocation mémoire associée et laisser cette partie gérée de manière interne par le programme. Pour cela il faut utiliser la fonction membre ***push_back*** pour ajouter un nouvel élément en fin de vecteur. Ainsi toujours en reprenant notre tableau de 10 doubles nous pouvons tout à fait le faire de la manière suivante :

```
#include <iostream>
#include <vector>

int main(void)
{
int N = 10;
std::vector<double> vector_array;

for(int i=0; i<N; i++)
{
vector_array.push_back(4.2);
}
 return 0;
}
```

Finalement nous pouvons comparer les 3 approches pour initialiser le tableau et afficher les différentes valeurs

```
#include <iostream>
#include <vector>

int main(void)
{
double static_array[10];
int N = 10;
double *dynamic_array = new double[N];
std::vector<double> vector_array(N, 4.2);

for(int i=0; i<N; i++)
{
static_array[i] = 4.2;
dynamic_array[i] = 4.2;
}

for(int i=0; i<N; i++)
{
std::cout<<static_array[i]<<std::endl;
std::cout<<dynamic_array[i]<<std::endl;
}

for(int i=0; i<vector_array.size(); i++)
{
std::cout<<vector_array[i]<<std::endl;
}



// version encore plus simple en C++ 11
// auto permet au compilateur de "deviner" le bon type pour la variable
// à n'utiliser que dans les boucles
// 'element': nom de la variable qui va representer chaque element du vector
```

```
for(auto element: vector_array)
    {
        std::cout<<element<<std::endl;
    }

delete[] dynamic_array;
return 0;
}
```

Annexe 2 : 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 3 : 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".
- avoir fini la partie 1 du TP.

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
- Initialiser toutes les variables lors des déclarations.
- Faire une allocation dynamique plutôt que déclarer des tableaux statiques de taille disproportionnée.
- Avoir fini la déclaration et la définition du capteur analogique.
- Libérer la mémoire (via **delete** ou **delete[]**) pour un bloc mémoire alloué via **new**, **new []**.

Niveau 3 : 15-17
- Initialisez les données membres des classes version C++11 (lors de la déclaration des données membres ou des constructeurs)
- avoir fini les parties 1 et 2 du TP
- utiliser au maximum les éléments de la STL (en particulier les fonctions associées aux std : :vector).
- Spécifier explicitement le constructeur utilisé par la classe mère lors de la définition du constructeur de la classe fille.

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...
**/
```