Programme C++

Header

- Pour chaque fichier .cpp associé un fichier .h.
- Chaque HEADER doit démarre par un `#IFNDEF FOO_BAR_BAZ_H_` si le header se trouve dans foo/src/bar/baz.h

- Ordre des inclusions d'header:
    1. header correspondant au .cpp
    2. C/C++ headers
    3. Autre librairie .h (boost)
    4. Autre header du projet

Namespace:

- Ne pas mettre de namespace sans nom dans un Header
- Dans un .cpp il est autorisé de mettre des namespace sans nom
- Pas de namespace inline
- Utiliser des raccourcis `namespace baz = ::foo::bar::baz`
- Privilégier les fonctions dans un namespace plutot que des fonctions statiques dans une classe

```
namespace project{
    namespace fs{    // Correct
        void foo1();}
    class fs{
        static void foo1();
    }
}
```

Variables et Objet

- Declarer les variables/objets au plus prêt de leur utilisation

- `std::move` a utilisé quand l'on veut déplacer un objet
- Préféré int8_t, int16_t, int32_t et int64_t (ou leur version non signé uint64_t) a short, int, long
- Eviter les **macros** autant que possibles (préférer const,inline function…). Utiliser `#DEFINE` le plus pret de l'endroit de l'utilisation de la macro, pensez à `#UNDEF` juste après.

Classe

- Eviter d'appeler des fonctions virtuelles dans le constructeur
- Faire des initialisation arbitraire des attributs d'un objets
- Ordre dans lequel une classe est rangé (public, protected, private) :
    - using declaration
    - static constant
    - constructeurs
    - deconstructeur
    - méthodes
    - attributs
- Utiliser `explicit` quand cela est nécessaire et proscrire `implicit`

```
class Object{
    /* explicit */ Object(int i)
}
int k=0;
Object objet = K; // est correcte tant que
                  // le constructeur n'est pas déclaré comme explicit
```

Constructeur

- **Les 5 constructeurs utiles**
    - Constructeur par défaut: ne prend pas de paramètres, instancie les attributs
    - Constructeur par copie. Il en est généré un de base par le compilateur. Cependant, il faut le définir si certains objets ont des pointeurs comme attributs. Dans ce cas le constructeur par défaut se contentera de copier le pointeur. Si l'on appelle le destructeur sur l'objet copié après, alors il y aura

une erreur dans notre nouvel objet. Voici les 4 syntaxes possibles:

```
Class( const Class& other );
Class( Class& other );
Class( volatile const Class& other );
Class( volatile Class& other );
```

- Constructeur de copie par assignement :

```
T& T::operator=(T arg)
{
    swap(arg); //si la classe T a un attribut n,on a swap(n,arg.n)
   return *this;
}
```

- Destructeur
- *C++11 Style*: Constructeur par déplacement. L'objet passé en parametre est déplacé dans le nouveau Objet et le constructeur par défaut. Voici la syntaxe `Class& C::operator=(C&& other)`. Les 4 étapes sont:
  - supprimer les attributs actuels de la classe
  - copier les attributs de l'objet passé en paramètre, dans la classe
  - mettre les attributs de l'objet passé en parametre à leur valeur par défault
  - renvoyer un pointeur *this

```
 T& operator=(T&& other) // move assignment
{
    assert(this != &other); // exit si ==
    delete[] mArray;        // supprime
    mArray = other.mArray;  // deplace
    other.mArray = nullptr; // met a défaut
    return *this;
}
```

- Creer une boucle infinie `Class( Class other );`

Avant C++11, si une classe fille ne définissait pas de constructeur, alors le constructeur

appelé lors de la création de l'objet était celui par défaut de la classe mère.En ajoutant dans la classe fille B la ligne `using A::A;`, c'est **l'héritage de constructeur**. Une autre astuce pour éviter les répétitions est la délégation de constructeur : un constructeur en appelle un autre

Opérateurs

Les utiliser que si cela est vraiment utile. *Quelques exemples*:

- **Flux**:

```
std::ostream& operator<<(std::ostream& os, const T& obj)
{
    // write obj to stream
    return os;
}
```

- **Comparaison**:

```
// Mettre cette fonction comme friend car elle a besoin d'acceder a des paramet
re privés de Class
friend bool operator<(const Class& l, const Clas&s r)
    {
        return std::tie(l.name, l.floor, l.weight)
             < std::tie(r.name, r.floor, r.weight); // compare tous les attribu
ts
    }
```

Function

- Inliner seulement les petites fonctions (tous les accesseurs par exemple)
- Un passage par référence en C++ correspond a un passage de pointeur, la variable est modifié, il n'y a pas de copie locale de la variable.
- Trailing Return :Permet de trouver le type de la sortie d'une fonction. Voici un exemple avec `C++11` et le mot clés `auto` : `template <class T, class U> auto`

`add(T t, U u) -> decltype(t + u);` versus la version C++98 `template <class T, class U> decltype(declval<T&>() + declval<U&>()) add(T t, U u);`.

- Utiliser au maximum l'attribut const, pour tous les accesseurs, pour toutes les méthodes ne modifiant pas la classe, et dans les parametres si lors d'un passage par référence on est sur de pas modifier la valeur.
    - L'attribut `constexpr` permet de définir de véritables constantes qui seront fixés à la compilation. (const peut être modifié par l'attribut `mutable`

Dynamic Allocation

- Pointeurs Pour des pointeurs, préféré `nullptr` à `NULL`, pour des integers utilisez 0, pour des réels 0.0 et pour des chars `'\0'`
    - Shared_ptr

```
void f()
{
    typedef std::shared_ptr<MyObject> MyObjectPtr; // nice short alias
    MyObjectPtr p1; // Empty
    {
        MyObjectPtr p2(new MyObject());
        // There is now one "reference" to the created object
        p1 = p2; // Copy the pointer.
        // There are now two references to the object.
    } // p2 is destroyed, leaving one reference to the object.
} // p1 is destroyed, leaving a reference count of zero.
// The object is deleted.
```

    - Unique_ptr déplace l'objet…
    - Caste
        - Utiliser static_cast<T> plutot que la version C
        - Utiliser const_cast<T> pour transformer des éléments non constants en éléments constant.

STL

- Lambda Expression, introduite par `C++11` : utile pour les fonctions de la STL. Voici un exemple:

```

```

Vector

De tous les structures de données de la STL, **vector est la "meilleure"** ("By default use vector when you need a container" de Bjarne Stroustrup).

- Privilégier les vector et les string aux tableaux et aux pointeurs de char. Evite les erreurs et rend le code plus portable.Dans le mode *release*, l'utilisation d'un vector est presque aussi rapide, et beaucoup plus sur, que celle d'un tableau style C, ne pas s'en priver.Quand la taille est fixe dès le début, il est équivalent d'utiliser un tableau *style C* ou le tableau de la STL: `std::array<int, 3>`. (ajout des méthodes `fill`, `empty`, et `at` qui vérifie les bornes...). Voici un exemple d'utilisation de std::array pour une matrice.

```
template <class T, size_t LIGNE, size_t COL>
using Matrice = std::array<std::array<T, COL>, LIGNE>;
Matrice<float, 3, 4> mat;
```

- Privilégier `push_back` a toutes les autres insertions dans un vector. Eviter les insertions répétés uniques: préférer `insert(vector.begin(), tableau , tableau + 50)` pour ajouter plusieurs valeurs en mêmet temps.
- Pour supprimer **vraiment** des éléments d'un conteneur, utiliser `container.erase( remove(container.begin(),container.end(), value ) )`.
- Privilégier les algorithmes de la STL, qu'une boucle itérant sur les valeurs d'un vector: exemple de la fonction `foreach()`.
- Itérer a la recherche d'élements dans un container avec find(if) et count(if) pour des containeurs non triés, et binary_search() pour des élements trié.sp
- Quelques astuces:
    - Trier les valeurs selon un prédicat. Les valeurs respectants le prédicat sont mises avant celle qui ne le respecte pas. Renvoit un pointeur sur le premier élement qui est faux pour la condition. `bound = std::partition`

`(myvector.begin(), myvector.end(), predicat);`.

- Récuperer les n premiers élément dans une liste bien plus grande,ici n=20 `nth_element(vector.begin(),vector.begin()+20,vector.end(), comparateur: optionnel)`. Pour que, **en plus** ces éléments soient triés, utiliser partial_sort().