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 Objets

- 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.

auto

- Préféré `auto x = 5` à `int x =5`.
- *auto* comprendra `auto x={1,2,3,4,5}` ; et x sera une `std::initializer_list<int>`. Cependant pour la déduction avec des templates, cela est impossible. Pour pouvoir passer a une fonction f({1,2,3}), il faut que f soit de la forme

```
template<typename T>
void f(std::initializer_list<T>)
```

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 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;
}
```

    - *Bonus:* Creer une boucle infinie `Class( Class other );`

- Le compilateur va générer pour nous un constructeur par copie, ou pr assignement. Si l'on veut empecher cela, et ainsi empecher l'assignement entre classe, on déclarera le l'opérateur d'assignement dans le header auquel on assignera le mot clé `= delete`. Ce mot clé peut être utilisé pour interdire l'usage d'un template pour un type défini : `template<>double fonction<double>(const double&) = delete`.
- La règle de trois indique que si l'on définint un constructeur par copie,un par assignement et un destructeur, il faut définir les trois. Si l'on génère un cpc,alors un constructeur par déplacement ne sera pas généré automatiquement. Il en de même si un destructeur est redéfinit. Pour indiquer au compilateur, qu'il peut générer quand même un constructeur, le mot clé `default` est utilisé.

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
    }
```

Fonction

- 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 l'objet qui appelle la méthode (le const se met ç la fin de la déclaration de la fonction, et dans les parametres si lors d'un passage par référence on est sur de pas modifier la valeur**. Le mot clé **decltype** permet d'extraire le type d'une variable ou d'une expression.
    - L'attribut `constexpr` permet de définir de véritables constantes qui seront fixés à la compilation. (const quand à lui peut être modifié par l'attribut `mutable`)
- Pour éviter les problèmes d'héritages de méthodes, déclarer une fonction purrement virtuelle avec `=0`,et toute fonction qui overide une méthode de la classe mère par `override` à la fin de la méthode. Ainsi si jamais la fonction n'est pas défini dans la classe mère, le compilateur nous l'indiquera.

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'`
    - Unique_ptr: Un unique_ptr ne partage pas son pointeur. Il ne peut pas être copié vers un autre unique_ptr, passé par valeur à une fonction, ni utilisé dans un algorithme STL (Standard Template Library) qui nécessite d'effectuer des copies.Un unique_ptr peut uniquement être déplacé. Cela signifie que la propriété de la ressource mémoire est transférée à un autre unique_ptr et que le unique_ptr d'origine ne le possède plus. Cela peut être utile lorsque l'on crée un objet dans une fonction et qu'on veut le passer en retour.
        - On peut affecter à des unique_ptr des destructeurs (si il en existe pas déja). Pour cela l'utilisation des fonctions lambda est pratique.

```
    std::unique_ptr<X,decltype(deletion)> F1(new X(26),deletion);

    auto deletion = [](X* x)
    {
    std::cout <<"L'objet est supprime";
    delete x;
    };
```

- On peut `reset` un unique_ptr.

- Shared_ptr Deux fois plus gros qu'un unique_ptr: contient un bloc de contrôle. Un shared_ptr est utilisable de deux bonnes manières. **Une mauvaise habitude est de déclarer un shared_ptr avec un raw pointeur qui a déja été définit/déclarer**. Chaque shared_ptr pointe vers un bloc de controle qui enregistre le nombre de pointeurs alloué... Un mauvais usage des shared_ptr entraine la création de plusieurs blocs de controle pour un même objet. Les trois manières de déclarer un objet sont:

```
    shared_ptr<A> a(new A()) // en utilisant new
    shared_ptr<A> b = make_shared(parametre constructeur de A) // en uti
lisant make_shared
    // make_shared ne supporte pas de définir un déconstructeur
    shared_ptr<A> c(a) // en copiant d'un autre shared ptr
    shared_ptr<A> c(a, deconstructeurlambda) // suffit (plus simple que
unique_ptr)
```

- Comme un unique_ptr, il est possible de définir une fonction lambda comme déstructeur
- Ne pas utiliser des shared_ptr de classe et des fonctions appelant this qui est un raw pointeur car cela crée un nouveau bloc de controle.
- Weak_ptr:

STL

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 ),container.end )`.
- 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.

> *Voici un exemple pour afficher toutes les valeurs d'un vector, en utilisant des **fonctions lambda***

```
std::for_each(t.begin(),t.end(),[](const auto& x)
    {
        std::cout << x << " ";
    });
```

> *Voici un autre exemple ou l'on est obligé de retourner une valeur:*

```
std::transform(v.begin(), v.end(), v.begin(),
    [](double d) -> double {
        if (d < 0.0001) {
            return 0;
        } else {
            return d;
}});
```

> *Il est aussi possible de capturer des variables dans une fonction lambda en les appelant par leur nom par référence [&variable], ou par valeur [=,variable]. N Lors de la capture par défaut [=],[&], seul les valeurs nécéssaires sont capturées. La capture par défaut permet de récupérer toutes les variables locales, il faut faire attention à ce que les variables soient accessibles à tout instant. Par exemple dans une classe, au sein d'une fonction menbre, la capture d'un attribut privée de la classe n'est pas possible. Pour pallier ce problème, il faut faire une copie locale de l'attribut dans la fonction puis de le capturer[].*

> *Il est aussi de déplacer des objets dans la fermeture grâce à `C++14`.*

- 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().
    - Initialiser le vector en incrémentant la valeur `iota(myvector.begin(),myvector.end(),valeur)`
    - Verifier si tous les éléments respectent un prédicat: `all_of`, si au moins `any_of` ou aucun `none_of`: renvoit un booleen.

Unordered_Map

Comme une Map mais les éléments ne sont pas triés en utilisant un comparateur <. Ainsi l'accès à une valeur passe de O(log n) (recherche dichotomique), à O(1) en utilisant une fonction de hash.Voici un exemple pour implémenter sa fonction de hash.

```
class A {
   int X; string Y;}

struct RHash{
   size_t operator()(const &A a){
       return hash<int>()(a.X) ^ hash<string>()(a.Y);
   }
}   // et enfin
std::unordered_map<A,RHash> map;
```

MultiMap

- Inserer dans une `multimap<int,int>` t avec la fonction `insert(pair<int,int>()).
- Pour récuperer toutes les valeurs d'une clé:

```
for(auto range=t.equal_range(0) ; range.first != range.second ;++range.first)
    {
        std::cout << range.first->second;
    }
// Ici auto représente pair<multimap<int,int>::iterator,       multimap<int,int>::iterator>
```

Tuple : *"pair aggrandis"*

- Creer : `auto tuple = make_tuple("hello",0,0.05)`
- Recuperer un objet : `get<0>(t)`

Template

Les templates sont des méchanismes qui ont lieu lors de la compilation.

Algorithme

Sommer tous les éléments d'un vecteur de unique_ptr de classe contenant un uint_32:

```
// 0 car on ajoute aucune valeur
std::accumulate(t.begin(),t.end(),0,
            [](int sum,const std::unique_ptr<A>& y)
                                    {
                                       return sum + y->i_;
                                    });
```

Déduction des templates

- Cas d'utilisations des références et de l'utilisation des constantes lors de la déduction des templates: `template<typename T>`
    - void f(T param)
        - Dans ce cas tout objet, meme si il est déclaré constant, est copié lors pour la fonction.
        - Si T est un tableau , il est percu comme un pointeur constant sur un tableu
    - void fonction(T& attri)
        - Si l'objet passé est déja une référence, alors l'objet perd la référence
        - Sinon l'objet garde ses attributs et est passé par référence.
        - Si param est un tableau alors contrairement au cas au dessus, il sera percu comme un tableau. Voici un code pour obtenir la taille d'un tableau

```
template<typename T, std::size_t N>
constexpr taille(T (&)[N]) noexcept
{
    return N;
}
```

    - void f(const T& param);
        - Dans ce cas là, tout parametre est passé par référence et devient const
    - void f(T&& param)
        - Si une déduction doit avoir lieu (utilisation de templates) alors param est une *référence universelle*. Cependant lorsque il n'y a pas de déduction éfféctué. Il s'agit d'une r-value. Ainsi **&& peut être ou une référence universelle ou une r-value**.

- **Pour obtenir des informations de déduction de type ou de variable auto**, on peut utiliser typeid(objet).name (deux objet seront différents si ils sont différents. Différence entre pointeur et objet. Cependant il ne détecte pas la différence entre une référence, un objet constant, et une valeur simple). On peut aussi utiliser la fonction `std::is_same<decltype(x),decltype(y)>::value` qui retourne un booleun suivant que x et y ont le même type. Cette fonction différencie référence, constante…

- Utilisation des fonctions variadiques, pour passer plusieurs arguments différents. Voici un exemple pour afficher en sortie plusieurs éléments:

```
template<class T>
void print(const T& l)
{
    std::cout << l;
}
template<typename T,typename ...R>
void print(const T& l,const R& ...p)
{
    std::cout << l;
    print(p...);
}
```

Sémantique de C++11

1. std::move et std::forward

`std::move` transforme n'importe quel argument qui lui est passé en une r-value, il ne déplace rien. L'objet casté en r-value est ensuite éligible à être déplacé. Ce n'est pas le cas tout le temps. Si l'objet est constant, alors std::move transformera l'objet en r-value mais le constructeur de copie sera appelé. Conséquence: ne jamais appelé std::move sur un objet constant lors de déplacement:

```
    void function(const std::string text)
    :   m_text(std::move(text)) // ne déplace pas, mais copie
    {
    }
```

std::forward est comme std::move, sauf qu'il ne caste en r-value que quand le parametre appelé a été initialisé comme r-value

```
    function(std::move(w));
    template<typename T>
    void function(T&& param){
        call_other_function(std::forward<T>(param));
    }
```

2. Distinguer entre r-value et référence universelle

Les références universelles ne sont présentes que lorsqu'il y a une déduction de types:

```
    auto&& objet \\ référence universelle

    template<class T> \\ référence universelle
    void function(T&& objet)

    template <class T>          \\ r-value
    void function(const T&& objet)

    template<typename T>        \\ Pas une r-value
    class Vector{               \\ car la déduction se fait
                                \\ quand le vecteur est crée
        void push_back(T&& objet)
    }
    void function(std::vector<T>&& vec) \\ r-value
```

3.

- Appliquez std::forward() à des références universelles la dernière fois que vous l'utilisez, et std::move() pour des références de r-value la dernière fois que vous utilisez la référence.
- Appliquez std::forward pour des références universelles retournés par valeur d'une fonction. Appliquez std::move() pour des références r-value retournés par valeur d'une fonction.
- Ne jamais appliquez forward ou move pour des objets locaux retournés par valeur. Cela supprime l'optimisation des compilateurs, la RVO.