Références :
    Dans une déclaration, le symbole & signifie "Je veux créer une référence".
    Partout ailleurs, le symbole & signifie "Je veux obtenir l'adresse de cette variable".
    * Règle 1 : une référence doit être initialisée dès sa déclaration.
    * Règle 2 : une fois initialisée, une référence ne peut plus être modifiée.

    Exemple avec pointeur :
```
        int age = 21;
        int *pointeurSurAge = &age;

        cout << *pointeurSurAge;
```
    même exemple avec référence :
```
        int age = 21;
        int &referenceSurAge = age;

        cout << referenceSurAge;
```
    Pour les 2 exemples suivants :
```
        Arme(const Arme &arme); //définition dans le Arme.h

        Exemple 1 avec passage d'agument :
            Arme *m_armeSec = new Arme();
            Arme m_arme = new Arme(*m_armeSec);

        Exemple 2 :
            Arme &m_armeSec = new Arme();
            Arme m_arme = new Arme(m_armeSec);
```
    Donc dans tous les cas &a en C++ <=> *a en C
    Sauf en paramètre de méthode : &a en C++ <=> &a en C

Surcharge des opérateur == :
```
    bool Duree::operator==(const Duree &duree)
    {
        if (m_heures == duree.m_heures && m_minutes == duree.m_minutes && m_secondes == duree.m_secondes)
            return true;
        else
            return false;
    }
```
Surcharge des opérateur =:
```
    //le constructeur de copie qui est appelé (affecter immédiatement la valeur d'un autre objet)
    Objet copieObjet = monObjet;

    //pour les autres =, c'est la méthode operator= qui sera appelée
    copieObjet = monObjet;
```
Tableau en C++ :
```
    doube t[] = {1.,2.,3.,4.,5.};
```
Delete :
```
    char* MyChar = new char[15];
    delete[] MyChar; //appel du destructeur d'Arme pour chaque Arme du tableau

    Arme* arme = new Arme();
    delete arme; //appel du destructeur d'Arme
```
Pour utiliser un template :

    A lire :
    template <class Tye> et template<typename T> sont équivalent,
    sauf que typename à été introduit après car on peut mêtre un type primitif, ce qui est != class.
    On préferera donc template<typename T>

    Important : on ne peut pas séparer le prototype du corps, toute la méthode se trouve dans le .h !!!

    Exemple 1 : on peut ajouter un paramètre à la méthode qui n'est pas de type T
```
        #include <vector>
    template<typename T>
    T moyenne(const std::vector<T>& tableau, int b)
    {
       T somme(0);
       for(int i(0); i < tableau.size(); ++i)
        somme += tableau[i];

       return somme/tableau.size();
    }
```
Exemple 2 : //exemple du prof avec "class" au lieu de "typename"
```
    template <class T>
    public class VectorTrie : public Vector<T> {

        //passage en paramètre
        VectorTrie( Vector<Type> ){
            addCollection()
        }

        //avec référence
        void add( const Type& oType ){
            Vector<Type>:: push_back ( oType ) ;
            sort();
        };


        Type& operator[ ] ( const unsigned int index ) const
```

```
        {
            return Vector<Type>:: operator[ ] ( index ) ;
        }

        //définition
    }

Exemlpe 3 : T pourrait être int, float, double,...
template <typename T>
class Rectangle{
    public:
        Rectangle(T gauche, T droite, T haut, T bas){}
    private:
        //Les cotes du Rectangle
       T m_gauche;
       T m_droite;
       T m_haut;
       T m_bas;
}

//appel
Rectangle<double> monRectangle(1.0, 4.5, 3.1, 5.2);


Exemple 4: Redéfinition de la classe Vector pour des string
template <>
class Vector< std::String >
{
    public:
        void Swap( const unsigned int idx1 , const unsigned int idx2 ) ;
    private :
        std::string m_oData ;
} ;
```

Classe Abstraite :

Une méthode "virtuelle pure" est une méthode qui est déclarée mais non définie dans une classe. Elle est définie dans une des classes dérivées de cette classe.

Une classe "abstraite" est une classe comportant au moins une méthode virtuelle pure.

Étant donné que les classes abstraites ont des méthodes non définies, il est impossible d'instancier des objets pour ces classes. En revanche, on pourra les référencer avec des pointeurs.
Si un des fils de la classe Forme définit toutes ses méthodes, alors il sera instanciable (et non instanciable dans le cas contraire).

Classe abstraite :

```
    class Forme
    {
        public:
            virtual void sePresenter() const
            {
                cout << "Je suis une Forme." << endl;
            }

            //Méthode virtuelle pure, on ajoute un "= 0" à la fin de la méthode.
            //Ne pas confondre avec le "const" qui indique seulement que l'élément de retour est constant
            virtual double surface() const = 0;

        virtual ~Forme()
        {}
};

class Cercle: public Forme
{
    public:
        virtual void sePresenter() const
        {
           cout << "Je suis un Cercle." << endl;
        }

        virtual double surface() const
        {
           return M_PI * m_rayon * m_rayon;  // Ici, pas de problème.
        }

        virtual ~Cercle()
        {}

    private:
        double m_rayon;    // Le rayon du cercle.
};


int main()
{
    Forme* ptr = 0;    // Un pointeur sur une forme.
    Cercle rond;         // On crée un Cercle, ceci est autorisé puisque toutes les fonctions ont un corps.

    ptr = &rond;        // On fait pointer le pointeur sur le Cercle.

    cout << ptr->surface() << endl;  // Dans la classe fille surface() existe donc ceci est autorisé.

    return 0;
}
```