# Exercice 1: Questions de cours (1+0.5x5+1+1+1=6 points)

1. Les extensions d’un fichier source C++ sont : .cpp et .h.
2. Definitions des concepts:
   * **Programmation orientée objet** c’est un paradigme de programmation inventé au début des années 1960 consitant en la définition et l’interaction de briques logicielles appelées objets.
   * **Un objet est une structure informatique regroupant** :
     + des variables, caractérisant l’état de l’objet,
     + des fonctions, caractérisant le comportement de l’objet.
   * **Classe**: Une classe est un ensemble d’ojets de même type.
   * **Polymorphisme**: c’est la capacité d’un l’objet à posséder plusieurs formes.
   * **Héritage**: C’est un mécanisme de la programmation orientée objet qui permet de créer des classes dites filles à partir des caractéristique et méthodes des classes existantes dites parentes.
3. Signification des mots clés:
   * **Virtual**: signifie que toute fonction-membre de la classe de base doit être surchargée (c’est-à-dire redéfinie) dans une classe dérivée.
   * **Private**: signifie que les propriétés et méthodes d’une classe sont privées à cette classe et inaccéssible depuis l’exterieur.
4. La sortie du code est: **15**.

# Solution de l’exercice 2 (choisir la(les) bonne(s) reponse(s)):(0.25x8=2pts)

1. c)une instance / la classe
2. b) type dynamique / type statique
3. b)Square/Shape
4. a)\*p = a;
5. a)Avant
6. b)après
7. peut contenir au moins constructeurs—Il n’est pas nécessaire de définir explicitement un constructeur ; cependant, toutes les classes doivent avoir au moins un constructeur. Un constructeur vide par défaut sera généré si vous n’en fournissez pas .
8. d)du nombre ou du type de leurs paramètres.

# Problème : (0.75+0.75+0.75+0.75+0.75\*4+0.75x2+0.75+0.75+0.75+0.75+0.75x2=12 points)

1. Classe Complexes permettant de représenter des nombres complexes.

* class Complexe  
  {  
    
  private:  
   double re;  
   double im;  
  };

1. Définition d’un constructeur par défaut sans paramètre permettant d’initialiser les deux parties du nombre à 0.

* public:  
   Complexe()  
   {  
   re = 0.0;  
   im = 0.0;  
   }

1. Définition du constructeur d’initialisation pour la classe.

* public:  
   Complexe(double pre, double pim)  
   {  
   re = pre;  
   im = pim;  
   }

1. Définition d’un constructeur public **Complexe(Complexe c)** permettant de créer une copie du Complexe passé en argument.

* public:  
   // User defined Copy constructor  
   Complexe(const Complexe &c)  
   {  
   re = c.re;  
   im = c.im;  
   }

1. Écriture des méthodes publiques *public Complexe plus(Complexe c),public Complexe fois(Complexe c),public Complexe divise(Complexe c)*, et *public double module()* qui implémentent les opérations algébriques classiques sur les nombres complexes (la racine carrée est donnée par *sqrt(double d)*).

* public:  
   Complexe plus(Complexe c)  
   {  
    
   return Complexe(re + c.re, im + c.im);  
   }  
    
  public:  
   Complexe fois(Complexe c)  
   {  
   double re\_temp = re \* c.re - im \* c.im;  
   double im\_temp = re \* c.im + im \* c.re;  
    
   return Complexe(re\_temp, im\_temp);  
   }  
    
  public:  
   Complexe divise(Complexe c)  
   {  
    
   double num\_re = re \* c.re + im \* c.im;  
    
   double num\_im = im \* c.re - re \* c.im;  
    
   double den = pow(c.re, 2) + pow(c.im, 2);  
    
   return Complexe(num\_re / den, num\_im / den);  
   }  
    
  public:  
   double module()  
   {  
   return sqrt(pow(re, 2) + pow(im, 2));  
   }

1. Ajoutons des méthodes **plus**, et **fois** qui prennent des **double** en paramètres.

* public:  
   Complexe plus(double pre, double pim)  
   {  
    
   return Complexe(re + pre, im + pim);  
   }  
    
  public:  
   Complexe fois(double pre, double pim)  
   {  
   double re\_temp = re \* pre - im \* pim;  
   double im\_temp = re \* pim + im \* pre;  
    
   return Complexe(re\_temp, im\_temp);  
   }

1. Écrivons une méthode **afficher()** qui donne une représentation d’un nombre complexe comme suit : .

* public:  
   void const afficher()  
   {  
   cout << re << "+" << im << "i" << endl;  
   }

1. Écrivons une méthode **bool egal(Complexe c)** permettant de comparer 2 complexes. Utiliser **"=="** pour faire une comparaison

* public:  
   bool egal(Complexe c)  
   {  
   return (re == c.re & im == c.im);  
   }

1. Ajoutons une méthode **toString()** renvoyant une représentation sous forme de chaine de caractère du Complexe courant.

* public:  
   string const toString()  
   {  
   return to\_string(re) + "+" + to\_string(im) + "i";  
   }

1. Écrivons une méthode **void swap(Complexe c1, Complexe c2)** permettant de permuter c1 et c2. Par exemple, on voudrait que le code suivant : Complexe c1=new Complexe(1, 1); Complexe c2=new Complexe(2,2); Complexe.swap(c1,c2) Complexe.affiche() affichera

* public:  
   void static swap(Complexe &c1, Complexe &c2)  
   {  
   Complexe temp = c1;  
   c1 = c2;  
   c2 = temp;  
   }
* **Oui**, nous pouvons écrire une telle méthode pour permuter des entiers.

1. Écrivons des méthodes **conjugue()** et **inverse()** qui transforment un complexe en son conjugué ou en son inverse. NB : ces méthodes ne retournent rien : elles modifient juste le Complexe sur lequel elles sont appelées.

* public:  
   void conjugue()  
   {  
   im = -im;  
   }  
    
  public:  
   void inverse()  
   {  
   double deno = pow(re, 2) + pow(im, 2);  
   re = re / deno;  
   im = -im / deno;  
   }  
  };