Merci de respecter les cadres pour les réponses. Tout débordement ne sera pas pris en compte et pourra être pénalisé de 1 point maximum par exercice.

Question de cours (4 points)

1) Quelle est la différence entre une classe et un objet ? (0,5 pt)

Objet est une instance d'une classe
Une classe = nouveau type et l'objet est la variable

2) Qu'est-ce que le principe d'encapsulation ? (0,5 pt)

Attributs en privés

3) Définir 2 propriétés d'une *Map* ? (1 pt)

Ordonnée par clés / une clé = 1 valeur (associatif) / doublons non autorisés pour les clés mais autorisés pour les valeurs

4) Quels sont les avantages de la composition ? (1 pt)

Durée de vie : quand le composite est détruit, le composant/agrégé est détruit aussi.
Exclusivité : une classe agrégée ne peut l'être que par une seule classe composite

5) Donner 2 relations possibles entre des classes. Expliquer en une phrase ces relations. (1 pt)

Agregation = relation faible entre 2 classes : pas de durée de vie liée
Composition = relation forte avec notion d'exclusivité (cf slides)

Exercice 1 : Etude d'une université (4 points)

Une université est composée de divers départements (sciences, technos, art…) regroupant des professeurs. On y trouve aussi des salles de classes comme par exemple la « UV-212 » ou encore « UV-424 ». Les salles peuvent être plus ou moins grandes et avoir 1 ou 2 portes. Je connais une amie qui a étudié à l'université Paris-Dauphine et y a rencontré M. Dupont, un professeur de maths.

Identifier sous forme de tableau : les classes, objets et attributs

Classes	*Objets*	*Attributs*
Universite	Paris dauphine	Nom …
Départements		Nom, taille
Profs	M. Dupont	Nom, age, sexe…
Salles	UV-212 et UV-424	nbPortes, taille

Exercice 2 (5 points) – 1 point par question

Soit la classe suivante :

```
class Data
{
private:
    std::list<int> m_list;

public:
    Data(int _size); // remplir aléatoirement la liste
    ~Data(); // destructeur
    float moyenne(); // calcul la moyenne de la liste
    void afficher(); // affiche le contenu de la liste
};
```

1) Implémenter en C++ le constructeur qui remplit aléatoirement la liste avec des entiers

```
MyData ::MyData(int size)
{
    For (unsignedint i=0 ; i<size ; ++i)
      m_list.push_back(rand()%200) ;
}
// ou alors push_front() pour des ajouts en tête. Accepter aussi insert() si bien utilisé
```

2) Implémenter en C++ la méthode *moyenne*()

```
float MyData ::moyenne()
{
   int sum = 0 ;
   For (const auto& elem : m_list)
      sum += elem ;

   float moy = sum/(float)m_list.size() ;
   return moy ;
}

// Cast obligatoire ! -0.5 point si oubli
// accepter static_cast<float> pour ceux qui connait
// accepter la version par itérateur si elle est bien utilisée. Il faut déférencer l'itérateur pour accéder à l'élément par exemple.
```

3) Implémenter en C++ la méthode *afficher*()

```
Void MyData ::afficher()
{
    For (const auto& elem : m_list)
       std ::cout << elem << std ::endl ;
}

// ou alors version itérateur
// ne pas pénaliser « const auto& » , on accepte « auto » ou « auto& »
```

4) Implémenter en C++ le destructeur

```
MyData ::~MyData() {}

// Ne surtout pas libérer le vecteur avec delete !! 0 point dans ce cas !
```

5) Ecrire les lignes du *main* qui créé un objet de type *Data* puis appelle la méthode *moyenne* et enfin la méthode *afficher*. Merci de libérer si nécessaire la mémoire allouée…

```
MyData D(50);
D.moyenne() ;
D.afficher() ;

// pas besoin de libérer la mémoire sauf si version avec des pointeurs et new…

----------  Autre version avec pointeurs ----------

MyData* D = new MyData(50);
D->trier() ;
D->afficher() ;

delete D ; // pénaliser de 0.5 point si oubli du delete !
```

Exercice 3 : (2 points) – 0.5 point par ligne

```
class Test
{
public:
    Test() { std::cout << "default ctor" << std::endl; }
    Test(int a) { std::cout << "overload ctor" << std::endl; }
    Test(int a, float b) { std::cout << "overload2 ctor" << std::endl; }
    ~Test() { std::cout << "default dtor" << std::endl;}
    int foo() { std::cout << "inside foo" << std::endl; }
};

int main()
{
    Test* t = new Test(0);
    Test t2;
    t->foo();
    return 0;
}
```

Ecrivez la sortie du programme (ce qui sera affiché à l'écran lors de son exécution).

overload ctor
defaut ctor
inside foo
default dtor

Exercice 4 (5 points)

Ecrivez une classe *Compteur* qui comporte les méthodes suivantes :

1. un constructeur par défaut
2. un constructeur surchargé
3. une méthode d'incrémentation du compteur
4. une méthode de décrémentation du compteur
5. un accesseur pour récupérer la valeur du compteur

On implémentera les méthodes directement dans la classe pour plus de simplicité.

```
// 0.5 pour la structure de la classe
class Compteur
{
private:
   // 1 point si en privé sinon 0 point si l'attribut est en publique. Initialisation facultative.
   int m_value = 0;

public:
   // 0.5 point
   Compteur() : m_value(0) {

   }

   // 0.5 point
   Compteur(int value) : m_value(value) {

   }

   // Facultatif
   ~Compteur() {
      // vide rien à faire
   }

   // 0.5 point
   void increment() {
      setValue(getValue() + 1);
   }

   // 0.5 point
   void decrement() {
      setValue(getValue() - 1);
   }

   // 0.5 point (le const est facultatif)
   int getValue() const { return m_value; }
```

```
  // 1 point ou alors c'est fait dans la méthode decrement()
  void setValue(int value) {
     if (getValue() >= 0)
     {
        m_value = value;
     }
     else
     {
        // ce que vous voulez ou pas de else...
        std::cout << "error !" << std::endl;
        exit(-1);
     }
  }
};
```

Bon courage !

Annexe des fonctions pour le conteneur *liste* contenant des entiers :

void push_front(int valeur) ; // ajoute un élément en tête de liste
void push_back(int valeur) ; // ajoute un élément en fin de liste
void pop_front() ; // supprime le premier élément de la liste
void pop_back() ; // supprime le dernier élément de la liste
void insert(iterator position, int valeur) ; // ajoute un élement à une certaine position (itérateur)
iterator erase(iterator position) ; // supprime un élement à une certaine position et retourne un itérateur sur l'élément suivant
void clear() ; // supprime tous les éléments de la liste
int front() ; // retourne le premier élément de la liste
int back() ; // retourne le dernier élément de la liste