Master Ingénierie Mathématique (M2) — D5MA2J1 Informatique

Université Paris–Sud — Centre d'Orsay

AN-EDP — Année 2012 – 2013

© *Jean-Baptiste APOUNG KAMGA <jean-baptiste.apoung@math.u-psud.fr>*

Examen

Lundi 17 Décembre 2012 - Durée : 3 heures

Aucun document n'est autorisé.

Les Thème-5 et Thème-6 sont au choix.

Thème - 1 *Question de cours : Paradigme de programmation*

Donner un avantage et un inconvénient de chacun des paradigmes de programmation listé ci-dessous.
On précisera un langage de programmation identifiant chaque paradigme.

- Paradigme impératif.
- Paradigme structuré.
- Paradigme orienté objet.
- Paradigme fonctionnel.

Thème - 2 *Syntaxe en C++*

Les programmes C++ suivants comportent chacun *une seule* erreur de syntaxe. À l'aide des messages d'erreur produits

par le compilateur, trouvez les et écrivez une version corrigée de chacun des programmes.

Exercice-1 : Trouvez et corrigez l'erreur de syntaxe contenue dans le programme suivant :

Listing 1 – R2Sommet.cpp

```
1  #include <iostream>
2  class R2 {
3  protected:
4    double m_x;
5    double m_y;
6  public:
7    R2(const double& x, const double
       & y) : m_x(x),m_y(y) {}
8    virtual ~R2() {}
9  };
10
11 class Sommet : public R2{
12   int m_reference;
13 public:
14   Sommet(const double& x, const
       double& y, const int& reference
       )
15     :m_x(x), m_y(y), m_reference(
         reference){}
16 friend std::ostream& operator << (
     std::ostream& os, const Sommet& s
     ){
17     os << '(' << s.m_x << ',' << s
         .m_y << ')';
```

```
18      return os;
19    }
20     ~Sommet() {}
21  };
```

Voici le message donné par le compilateur

```
R2Sommet.cpp: In constructor '↩
  Sommet::Sommet(const double&, ↩
  const double&, const int&)':
R2Sommet.cpp:15:6: error: class '↩
  Sommet' does not have any field ↩
  named 'm_x'
R2Sommet.cpp:15:14: error: class '↩
  Sommet' does not have any field ↩
  named 'm_y'
R2Sommet.cpp:15:43: error: no ↩
  matching function for call to 'R2↩
  ::R2()'
R2Sommet.cpp:15:43: note: ↩
  candidates are:
R2Sommet.cpp:7:3: note: R2::R2(↩
  const double&, const double&)
R2Sommet.cpp:7:3: note:   ↩
  candidate expects 2 arguments, 0 ↩
  provided
R2Sommet.cpp:2:7: note: R2::R2(↩
  const R2&)
R2Sommet.cpp:2:7: note:   ↩
  candidate expects 1 argument, 0 ↩
  provided
```

Exercice-2 :

L' une des instructions du **Listing 2** ci-dessous empêchera la compilation. Préciser laquelle et justifier.

Listing 2 – increment.cpp

```
1 #include <iostream>
2 int main(int argc, char** argv){
3 int un = 1, deux = 2;
4 int& suivantUn   = un++ ;
5 int& suivantDeux = ++deux;
6 return 0;
7 }
```

Exercice-3 : Qu'affiche le code (**Listing 3**) ci-dessous ? Justifier.

Listing 3 – constref.cpp

```
#include <iostream>
int main(int argc, char** argv){
int  i = 3;
const int& j =i;
int &k = i; k++;
std::cout<< i <<"  " <<j <<"  " <<↩
   k <<std::endl;
return 0;
}
```

Exercice-1 : Orientée Objet : dérivation

Etablir la visibilité des données ou des fonctions membres d'une classe suivant le type de dérivation, en complétant la table du (**Listing 4**), suivant la légende : INA(inaccessible) , PRI (privée), PUB (publique) et PRO (protégée).

Listing 4 – Dérivation et accessibilité des données

	Inaccesibles	Privées	Protégées	Publiques
Héritage Privée				
Héritage Protégée				
Héritage Publique				

Exercice-2 : Orientée Objet : composition

On considère le code suivant (**Listing 5**) ainsi que le message du compilateur associé :

Listing 5 – composition.cpp

```
1 #include <iostream>
2 //↩
    ----------------------------------------
3 class VariableAleatoire{
4 unsigned m_dim ;
5 public:
6 ~VariableAleatoire(){}
7 VariableAleatoire(unsigned dim):↩
    m_dim(dim){}
8 VariableAleatoire() = delete;
```

```
9  };
10 //↩
     ----------------------------------------

11 class MouvementBrownien{
12 VariableAleatoire m_va;
13 public:
14 ~MouvementBrownien(){}
15 MouvementBrownien(const ↩
     VariableAleatoire& va):m_va(va){}
16 MouvementBrownien(){;}
17 };
18 //↩
     ----------------------------------------

19 void test(){
20  VariableAleatoire va(10);
21  MouvementBrownien brownien(va);
22 }
```

Voici le message donné par le compilateur

```
composition.cpp: In constructor ↩
  MouvementBrownien::↩
  MouvementBrownien():
composition.cpp:16:20: error: use ↩
  of deleted function ↩
  VariableAleatoire::↩
  VariableAleatoire()
composition.cpp:8:1: error: ↩
  declared here
```

1. Expliquez pourquoi ce code ne compile pas.
2. Que se passerait-il si l'on supprimait la ligne 16 ?

Exercice-3 : Orientée Objet : délégation

On considère le code suivant, dites s'il réalise la fonctionnalité attendue.

Listing 6 – delegation.cpp

```
1  #include <iostream>
2  class Action{
3  public:
4  virtual void evolue()const{std::cout<<"Action \n";}
5  };//----------------------------------------

6   class ActionEuro:public Action{
7   public:
8    void evolue()const/*override*/{ std::cout<<"ActionEuro \n"; }
9   };//----------------------------------------

10  class ActionAsiatique: public Action{
11  public:
12   void evolue()const /*override*/{std::cout<<"ActionAsiatique \n";}
13  };//----------------------------------------
```

```
14 class Option{
15 const Action& m_action;
16 public:
17 Option(const Action action):↩
     m_action(action){}
18 void evolue() const { m_action.↩
     evolue();}
19 }; //↩
     ----------------------------------------

20 int main(int argc, char** argv){
21 ActionEuro action_euro;
22 ActionAsiatique action_asi;
23 Option option_euro (action_euro);
24 Option option_asi (action_asi);
25 option_euro.evolue();
26 option_asi.evolue();
27 return 0;
28 }
```

Thème - 4 *Ecriture d'une classe en C++*

L'objectif de ce problème est d'écrire une classe permettant de modéliser des nombres complexes.

Écrire une classe `Complexe` possèdant les caractéristiques suivantes :

1. Définir un constructeur qui prend en argument la partie réelle et la partie imaginaire d'un `Complexe`, un constructeur par copie et un desctructeur.
2. Écrire les fonctions d'accès renvoyant la partie réelle et la partie imaginaire d'un `Complexe`.
3. Écrire les fonctions calculant le module et l'argument d'un nombre complexe.
4. Surcharger l'opérateur = pour permettre l'affectation de nombres complexes.
5. Surcharger les opérateurs + et − pour pouvoir sommer et soustraire des nombres complexes.
6. Écrire par surcharges d'opérateurs le produit d'un `Complexe` par un `double` à gauche et à droite.
7. En utilisant à nouveau la surcharge d'opérateurs, écrire le produit de deux nombres complexes.
8. Permettre l'affichage d'un nombre complexe de manière à afficher $a + ib$ dans le cas général et a si la partie imaginaire est nulle.

Thème - 5 *Programmation générique en C++*

Exercice-1 **: Fonction générique**

Trois individus s'amusent à fournir une fonction générique *optimisée* qui retourne le maximum entre deux objets supportant l'opérateur de comparaison ($<$). Malheureusement dans leur soucis d'optimisation ils commettent chacun une erreur. Identifier la.

Listing 7 – individuI.cpp

```
template<typename T>
const T& myMax(const T& a, const T&
  & b){
return a<b? a: b;
}
```

Listing 8 – individuII.cpp

```
template<typename T>
T const& myMax(T const a, T const
  b){
return a<b? b: a;
}
```

Listing 9 – individuIII.cpp

```
template<typename T>
const T& myMax(const T a, const T
  b){
static T res = a<b? b: a;
return res;
}
```

Exercice-2 : Métaprogrammation

Soit n un entier positif et x un nombre réel, écrire un programme calculant, **durant la compilation** la valeur x^n.

Exercice-1 : On souhaite disposer d'une classe **`VarAleatoire`** qui modélise une variable aléatoire. On voudrait que cette classe soit extensible dans le futur sans nécessiter sa recompilation. On concours d'architecte est ouvert et les deux propositions suivantes nous sont parvenues. (*On n'a fait figurer ici que les éléments essentiels des ces classes*)

Conception proposée par l' architecte **A**

Listing 10 – Architecte A

```
class VarAleatoire{
public:
template <typename TypeManip>
void transform(const TypeManip& ↩
  unManip)
{
 unManip.execute(*this);
}
};
```

Listing 11 – Architecte A

```
class CalculMoyenne{
public:
void
execute(VarAleatoire& vaCible) ↩
  const
{
    "code ici"
}
};
```

Conception proposée par l'architecte **B**

Listing 12 – Architecte B

```
class ManipVarAleatoire;
class VarAleatoire{
ManipVarAleatoire* m_manipVA;
public:
void
chargeManip(ManipVarAleatoire* ↩
  unManip)
{
 unManip->chargeCible(*this);
 m_manipVA = unManip;
}
void transform()
{
      m_manipVA->execute();
}
};
```

Listing 13 – Architecte B

```
class ManipVarAleatoire{
VarAleatoire* m_vaCible;
public:
void chargeCible(VarAleatoire& ↩
  unVA);
virtual void execute() = 0;
};
```

Listing 14 – Architecte B

```
class CalculMoyenne
    : public ManipVarAleatoire{
public:
void execute(){ "code ici" }
};
```

- Ces conceptions sont-elles susceptibles de résoudre le problème ?
- Laquelle préférez-vous ? Justifiez ?
- Expliquez comment ajouter une opération d'écriture d'une **`VarAleatoire`** dans un fichier, avec chaque conception.