NOM : Prénom :

Examen Méthodologie Orientée Objets (IF3)

Durée 1h, documents autorisés, annales et appareils communiquant interdits

→Répondre sur une nouvelle copie double ←

Exercice : Traitement de petit « big data » (45 minutes)

Le but de cet exercice est de réaliser une hiérarchie de classes permettant d'effectuer des traitements rapides sur des données volumineuses. Cette hiérarchie de classes doit permettre une gestion précise de la mémoire, l'optimisation de la vitesse des traitements n'est pas demandée (5GE).

Classe Data

Voici la déclaration de la classe ***Data*** permettant de stocker, sous forme de matrice, des *double*:

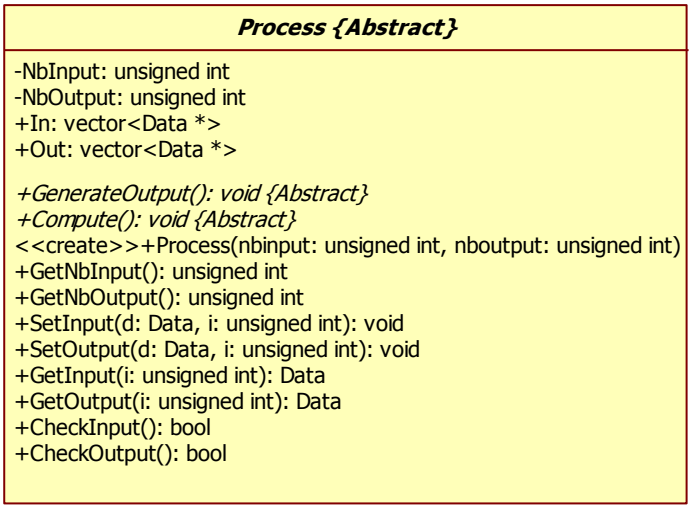
```
class Data
{       unsigned int L;  // nombre de lignes
        unsigned int C;  // nombre de colonnes
        double **Pt;
        bool IsAllocated; // l'espace mémoire es Alloué
        Data& operator=(const Data &d) { return *this; }
        Data(const Data &d) { }

public:
        bool GetIsAllocated() { return IsAllocated;}
        unsigned int GetL()   { return L; }
        unsigned int GetC()   { return C; }

        void SetLC( unsigned int l, unsigned int c );
        double GetPt(unsigned int i, unsigned int j) const
                { return Pt[i][j]; }
        void SetPt(const double &v, unsigned int i, unsigned int j)
                { Pt[i][j] = v;    }
        double& operator()(unsigned int i, unsigned int j)
                { return Pt[i][j]; }
        void Allocate(); // Alloue la mémoire pour Pt à partir de L et C
        void Destroy();  // Libère la mémoire de Pt à partir de L et C
        Data():L(0),C(0) { Pt = 0; IsAllocated =  false; }
        Data(unsigned int l, unsigned int c):L(l),C(c)  { Allocate() ; }
        virtual ~Data()  { Destroy(); }
        void Print() const;
};
```

1) Combien de constructeurs possède cette classe et quels sont leur rôles ?
2) Donner l'UML de la classe ***Data***.
3) *Allocate()* et *Destroy()* sont les fonctions réalisant respectivement l'allocation dynamique et la libération de mémoire du champ ***Pt***.
    a. Dans ce genre de classe C++, quelles sont les méthodes à surcharger pour garantir une gestion mémoire correcte ?
    b. Est-ce fait correctement ici ?
    c. Quel est l'intérêt de la solution proposée ici?
4) En vous appuyant sur *Allocate()* et *Destroy()*, donner le code C++ de la fonction *SetLC()* qui permettra de modifier le nombre de lignes et de colonnes et de garantir une bonne gestion de la mémoire.

Classe Process

La classe ***Process*** est une classe abstraite fixant les règles générales de fonctionnement des traitements et des interactions avec les objets de type ***Data***. Ces règles permettent d'optimiser la consommation en mémoire et la vitesse d'exécution des traitements.
Typiquement, un objet de type ***Process*** disposera d'entrées ***In*** et de sorties ***Out*** et permet de réaliser des seuillages, des calculs de moyenne, de gradient, …

Avant de lancer le traitement par la fonction ***Compute()***, la fonction ***GenerateOutput()*** devra être appelée afin d'allouer les espaces mémoires nécessaires en sortie.
Voici le diagramme UML de la classe ***Process***.



NbInput et *NbOutput* correspondent aux nombres d'entrées et de sorties nécessaires au traitement.
In et *Out* seront typés comme précisé de manière abusive dans l'UML : vector<Data*>.
SetInput() et *SetOutput()* (et respectivement les *Get*…) permettent de lier des objets ***Data*** existants au traitement.
GenerateOutput() et *Compute()* sont des fonctions virtuelles pures.
Le code du constructeur est le suivant :

```
Process(unsigned int nbinput=1, unsigned int nboutput=1)
                        :NbInput(nbinput), NbOutput(nboutput)
        {
        In.resize(NbInput);
        Out.resize(NbOutput);
        for(unsigned int i=0; i<NbInput; i++)
                In[i] = NULL;
        for(unsigned int i=0; i<NbOutput; i++)
                Out[i] = NULL;
        }
```

CheckInput() et *CheckOutput()* permettent de vérifier si les entrées et les sorties nécessaires à l'objet ont bien toutes été liées à des variables par les méthodes *SetInput()* et *SetOutput()*.

5) Modéliser par relations externes la classe ***Process***. On se limitera à la modélisation des champs. **Justifier** les relations utilisées. (reportez-vous aussi aux codes suivants)

6) Voici le code de la classe ***Duplicate*** qui permet de dupliquer une entrée sur une sortie, avec un extrait de code d'utilisation.

```
class Duplicate: public Process
{
public:
  Duplicate():Process(1,1) {}
  virtual void GenerateOutput()
        {
        if( CheckInput() && CheckOutput() )
                Out[0]->SetLC( In[0]->GetL(), In[0]->GetC() );
        }
  virtual void Compute()
        {
        if( CheckInput() && CheckOutput() )
                for( unsigned int i=0; i<In[0]->GetL(); i++)
                        for( unsigned int j=0; j<In[0]->GetC(); j++)
                                Out[0]->SetPt( In[0]->GetPt(i,j), i, j);
        }
};
```

Extrait du main :

```
        Data in(5,4);
        Data out;
        Duplicate dupli;
        dupli.SetInput(  &in  );
        dupli.SetOutput( &out );
        dupli.GenerateOutput();
        dupli.Compute();
        cout << "Duplication " << endl;
        out.Print();
```

a. En respectant le modèle de ***Process***, écrire la classe ***Mean***, qui calcule la valeur moyenne d'une seule entrée (***In[0]***) et stocke cette valeur dans la sortie 0 (***Out[0]*** est un scalaire, c'est-à-dire une matrice de taille 1x1).

7) On dispose des classes ***Mean***, ***Substract*** (qui réalise pour **2 entrées de même taille** « Out[0] = In[0] – In[1] ») et ***Square*** (qui réalise « Out[0] = $In[0]^2$ »). Donner le code d'une fonction main calculant la variance d'une entrée X et consommant le moins possible de mémoire. On rappelle :

$$Var[X] = E[(X - E[X])^2], \text{ avec } E[.] \text{ l'espérance mathématique}$$

8) On souhaite disposer de nouvelles classes de type ***Data*** constituées non plus de ***double*** mais de ***complex***. Que proposez-vous de mettre en place ? Que faut-il redévelopper ? Justifier.

Exercice : Trouver les erreurs (15 minutes)

1) Les programmes suivants possèdent chacun ***une*** erreur de conception (*i.e.* une erreur lors de la compilation, de l'exécution, résultat incohérent). Pour *chaque* programme, **indiquer** quelle est l'erreur et **corriger** la.

```cpp
#include <iostream>
using namespace std;
int main()
{ double tab[]={1,2,3,4,5};
  cout << "valeurs : " << endl;
  for(int i=0; i<5; i++ );
     cout << tab[i] << " ";
  return 0;
}
```

Programme 1

```cpp
void AllocDyn(double *tab, int t)
{       tab = new double[t]; }

int main()
{       double *tab=NULL;
        AllocDyn( tab, 5);
        tab[0]=0;
        return 0;
}
```

Programme 2

```cpp
int main()
{
        double a = 3.14;
        double *n = &a;
        double r = 1.0/*n;
        return 0;
} // ne compile pas !
// d'après compilateur, il y a une
// erreur ligne 5 « double r =… »
```

Programme 3

```cpp
#include <iostream>
using namespace std;
float MyFunc(float a)
{return (1/2)*a;}

int main()
{  cout << MyFunc(3.1415);
   return 0;
}// affiche 0  ☹
```

Programme 4

2) **Modifier les classes B** suivantes pour que les programmes compilent, s'exécutent et produisent un résultat cohérent. Il n'y a qu'**une** erreur par **classe B**.

```cpp
#include <iostream>
using namespace std;

class B
{
public:
        double X;
        double& GetX2()
         {return X*X;}
};

int main()
{
        B b;
        b.X = 2;
        cout << b.GetX2();
        return 0;
}
// Erreur de compilation ligne
//   {return X*X;}
```

Programme 5

```cpp
#include <iostream>
using namespace std;
class B
{   double X;
public:
   void SetX(double x) { X=x; }
   double GetX()    { return X; }
};
class C : public B
{public:
        void Affiche() const
         { cout << GetX(); }
};
int main()
{       C c;
        c.SetX(3);
        c.Affiche();
        return 0;
} // Erreur de compilation ligne
//   { cout << GetX(); }
```

Programme 6

Fin.