Université de Nice-Sophia Antipolis — POLYTECH
ELEC4 — 2021–2022

Examen de Programmation C++

Durée : 2h
Aucun document autorisé
Mobiles interdits Nom : Prénom :

---

1 La classe abstraite matrice

▶ 1. Écrivez la classe abstraite *générique* `matrice`. On appellera `T` le type générique des éléments de la matrice. Cette classe contient :

- 3 variables membres protégées, `nbElem`, `nbL` et `nbC`, qui sont respectivement le nombre d'éléments contenus dans la matrice, son nombre de lignes et de colonnes ;
- un constructeur public avec 2 paramètres (le nombre de lignes et de colonnes, par défaut égaux à 0) pour initialiser les 3 variables membres ;
- un constructeur de copie public ;
- le destructeur public (déclaré **virtual**) ;
- la méthode publique virtuelle pure `get` qui renvoie l'élément de type `T` en (i, j) ;
- la méthode publique virtuelle pure `set` qui met un l'élément `x` de type `T` en (i, j) ;
- la méthode publique virtuelle pure `toString` qui renvoie la représentation sous forme de `std::string` de l'objet courant ;
- la surcharge publique de l'opérateur `<<` pour écrire une `matrice<T>` sur un `std::ostream` ;

Fichier matrice.hpp :

```
#pragma once

#include <cassert>

template<typename T>
class matrice {
protected:
  int nbElem;
  int nbL, nbC;

public:
  matrice(const int m, const int n) :nbL(m), nbC(n), nbElem(m*n) {
    assert(m>0 && n>0);
  }
  matrice(const matrice<T> &m) : nbL(m.nbL), nbC(m.nbC), nbElem(m.nbElem) {}
  virtual ~matrice() {};
  virtual T &get(const int i, const int j) const =0;
  virtual void set(const int i, const int j, const T& x) =0;
  virtual std::string toString() const =0;
  friend std::ostream&operator<<(std::ostream &f, const matrice<T>& m)
  {
    return f << m.toString();
  }
};
```


.....

2 La classe matriceDouble

On souhaite manipuler des matrices de réels **double**. Pour cela, vous allez définir la classe `matriceDouble` qui hérite de la classe générique `matrice` précédente, en fixant le type des éléments au type **double**. Les réels **double** de la `matriceDouble` seront placés dans un tableau à une dimension appelé `mat` et alloué **dynamiquement**,

▶ 2. Écrivez la classe `matriceDouble` avec :

- la variable protégée `mat` de type **double** `*` (je vous aide) ;
- un constructeur public pour initialiser une `matriceDouble` $m \times n$ à une valeur `v` (par défaut 0.0) ;
- le constructeur public de copie ;
- le destructeur public ;
- l'implémentation des méthodes virtuelles `get`, `set` (vérifiez la validité des indices) et `toString`.
- la surcharge publique de l'opérateur d'affectation ;
- la surcharge publique de l'opérateur `<<`, si c'est utile. Oui ou non, vous expliquez.

Fichier matriceDouble.hpp :

```
#pragma once

#include <iostream>
#include <sstream>
#include <cassert>
#include <iomanip>

#include "matrice.hpp"

class matriceDouble : public matrice<double> {

protected:
  double *mat;

  void dupliquer(const matriceDouble &m) {
    this->mat = new double[matrice::nbElem = m.nbElem];
    matrice::nbL = m.nbL;
    matrice::nbC = m.nbC;
    for (int i=0; i<m.nbElem; i++)
      this->mat[i] = m.mat[i];
  }

public:
  matriceDouble(const int m, const int n, const double v=0.0) :
     matrice<double>(m,n), mat(new double[m*n])  {
     for (int i=0; i<matrice::nbElem; i++)
      this->mat[i] = v;
  }

  matriceDouble(const matriceDouble &m) : matrice<double>(m.nbL,m.nbC) {
     this->dupliquer(m);
```

```
  }

  ~matriceDouble() override {
    delete [] this->mat;
  }

  double &get(const int i, const int j) const override {
    assert(i>=0 and i<matrice::nbL);
    assert(j>=0 and j<matrice::nbC);
    int index = matrice::nbC*i+j;
    return this->mat[index];
  }

  void set(const int i, const int j, const double &x) override {
    assert(i>=0 and i<matrice::nbL);
    assert(j>=0 and j<matrice::nbC);
    int index = matrice::nbC*i+j;
    this->mat[index] = x;
  }

  const matriceDouble &operator=(const matriceDouble &m) {
    delete [] this->mat;
    this->dupliquer(m);
    return *this;
  }

  std::string toString() const override {
   std::ostringstream s;
   int k=0;
   for (int i=0; i<matrice::nbL; i++) {
     for (int j=0; j<matrice::nbC; j++)
       s << std::setw(3) << this->mat[k++] <<  " ";
     //
     s << "\n";
    }
    return s.str();
  }
};
```

.....

▶ 3. Est-il possible d'écrire dans la classe `matrice` la surcharge publique de l'opérateur `()` pour accéder à un élément d'une `matriceDouble` ? Si on a une `matriceDouble m`, on pourra par exemple écrire `m(2,1)` pour accéder au réel double de la 3ème ligne et de la 2ème colonne. Si c'est possible (expliquez), écrivez cette surcharge.

Fichier matrice.hpp :

```
T &operator() (int i, int j) const {
  return this->get(i,j);
}
```

.....

▶ 4. Dans la classe `matrice`, ajoutez la déclaration de la méthode publique virtuelle pure `subMat` qui possède 4 paramètres entiers `i1`, `i2`, `j1` et `j2` (avec `i1`⩽`i2` et `j1`⩽`j2`), et qui renvoie un pointeur

sur un objet de type `matrice<T>` formé des lignes `i1` à `i2` et des colonnes `j1` à `j2`.

Fichier matrice.hpp :

```
virtual matrice<T> *subMat(const int i1, const int i2,  const int j1, const int j2) c
```

.....

▶ 5. Dans la classe `matriceDouble`, écrivez l'implémentation de la méthode publique `submat` (sans oublier son en-tête). Si les indices ne sont pas valides, vous émettrez l'exception `IndexInvalide`. Vous déclarerez la classe `IndexInvalide` héritière de `std::exception`. L'exception indiquerera le ou les indices invalides.

Fichier IndexInvalide.hpp :

```
class IndexInvalide : public std::exception {
protected:
  std::string msg;
public:
  IndexInvalide(const int i) {
    this->msg = "Indice invalide : " + std::to_string(i);
  }
  IndexInvalide(const int i1, const int i2, const int j1, const int j2) {
    this->msg = "Indices invalides : " + std::to_string(i1) + "<=" + std::to_string(i2) +
      std::to_string(j1) + "<=" + std::to_string(j2);
  }
  IndexInvalide(const IndexInvalide &e) : msg(e.msg) {}
  // redéfinition de what
  virtual const char* what() const noexcept override {
    return msg.c_str();
  }
};
```

Fichier matriceDouble.hpp :

```
matriceDouble *subMat(const int i1, const int i2, const int j1, const int j2) const c
{
    if (i1<0 or i1>=matrice::nbL) throw new IndexInvalide(i1);
    if (i2<0 or i2>=matrice::nbL) throw new IndexInvalide(i2);
    if (j1<0 or j1>=matrice::nbC) throw new IndexInvalide(j1);
    if (j2<0 or j2>=matrice::nbC) throw new IndexInvalide(j2);
    if (i1>i2 or j1>j2) throw new IndexInvalide(i1, i2, j1, j2);
    int nbl = i2-i1+1;
    int nbc = j2-j1+1;
    matriceDouble *m = new matriceDouble(nbl, nbc);
    // copier les éléments de this dans m
    int o_i = i1;
    for (int i=0; i<nbl; i++, o_i++)  {
      int o_j = j1;
      for (int j=0; j<nbc; j++, o_j++)
        m->set(i,j, this->get(o_i, o_j));
    }
    return m;
  }
```

.....

▶ 6. Écrivez la fonction `main` qui :
   — déclare une `matricedDouble m1` 3×6 dont les éléments sont initialisés à 1.0 ;
   — écrit sur la sortie standard `m1`
   — déclare `matricedDouble m2` initialisée à partir de `m1` ;
   — crée et écrit sur la sortie standard la sous-matrice prise entre les lignes 1 et 2 et les colonnes 1 et 3 de `m2`. Vous gérerez l'exception.

Fichier test.cpp :

```
int main() {

  matriceDouble m1(3,6, 1.0);
  std::cout << m1;
  matriceDouble m2(m1);
  try {
    std::cout << *(m2.subMat(1,2,1,3)) << std::endl;
  }
  catch (const IndexInvalide &e) {
    std::cerr << e.what() << std::endl;
  }
  return EXIT_SUCCESS;
}
```

.....