Classes (suite) [cm]
Exercices de cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 20 mai 2018

Table des matières

C++ - Exercices de cours (TP)

1 Appréhender le cours

1.1 Membres partagés / qzstatic1

Créez une classe `KStatic` muni d'un attribut partagé `cpt` de type `Entier`.

Que devez-vous obligatoirement faire concernant un attribut partagé.

Instanciez l'attribut partagé à zéro.

Ajoutez une méthode `m1` et une méthode partagée `m2`.
Écrivez un corps vide pour chacune.

Déclarez une instance `o` de type `K` puis écrivez tous les appels licites de méthodes.

1.2 Classe Dessin / pgdessin

1.2.1 Classe Dessin

Écrivez une classe `Dessin` contenant :

- Un attribut `n` (entier) du nombre de replicatats.
- Un attribut `motif` (chaîne de caractères) du motif à reproduire.
- Un attribut partagé public `total` (entier) du nombre d'instances.

Initialisez l'attribut partagé à zéro.

Écrivez un constructeur normal à deux paramètres qui :

- Initialise les attributs.
- Et incrémente le `total` du nombre d'instances.

Écrivez une méthode `afficher` qui affiche une ligne de `n` tirets suivi du `motif`.

1.2.2 Programme de test

On considère le programme suivant :

Programme C++ @[qzdessin.cpp]

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

#include "Dessin.hpp"

void afficher(int n)
{
  Dessin d(n,"*");
  d.afficher();
}

void afficherDessin(int n)
{
  for (int k = 0; k < n; ++k)
  {
    afficher(k);
  }
  for (int k = n; k >= 0; --k)
  {
    afficher(k);
  }
}

int main()
```

```
{
  cout<<"Voici un dessin"<<endl;
  afficherDessin(5);
  cout<<"Je viens d'afficher "<<Dessin::total<<" motifs"<<endl;
}
```

Sans l'exécuter, qu'affiche-t-il ?

Exécutez-le afin de confirmer vos résultats.

On supprime le qualificatif `static` de l'attribut partagé `total`.

Quelles seront les modifications à apporter au programme ? Justifiez.

1.3 Variables statiques / qzstatic2

Téléchargez le programme suivant :

C++ @[qzstatic2.cpp]

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

class Figure
{
  public:
    Figure()
    : csurf("blanc"), ccont("jaune")
    {}

    Figure(const string& cs, const string& cc)
    : csurf(cs), ccont(cc)
    {
      if (ccont == "noir")  { ++nfgCN; }
      if (csurf == "blanc") { ++nfgSB; }
      ++nfigs;
    }

    void afficher() const
    { cout << "csurf = " << csurf << ", ccont = " << ccont << endl; }

    void setCC(const string& x)
    { ccont = x; }

    void setCS(const string& x)
    { csurf = x; }

    static unsigned nfigs;      // # de figures
    static unsigned nfgCN;      // # de figures Contour Noir
    static unsigned nfgSB;      // # de figures Surface Blanc

  private:
    string csurf;               // couleur de la surface
    string ccont;               // couleur du contour
};

unsigned Figure::nfigs = 0;
unsigned Figure::nfgCN = 0;
unsigned Figure::nfgSB = 0;

int main(int, char*[])
{
  // Instanciation des objets
  Figure f1("rouge", "noir");
  Figure f2("bleu", "blanc");
  Figure f3;
  Figure f4("violet", "orange");

  // Exécution des traitements
  f2.setCC("noir");

  // Affichage des attributs:
```

```
  cout << "f3: "; f3.afficher();
  cout << "f2: "; f2.afficher();

  cout << "Figure::nfigs = " << Figure::nfigs << endl;
  cout << "f2.nfigs = " << f2.nfigs << endl;
  cout << "f1.nfgCN = " << f1.nfgCN << endl;
  cout << "f4.nfgSB = " << f4.nfgSB << endl;
}
```

Sans l'exécuter, indiquez l'affichage exact produit par son exécution.

Aide simple

Il s'agit de faire attention à ce qui est **partagé** par la classe de ce qui ne l'est pas. Pour ce genre d'exercices, aidez-vous de petits schémas associés à vos objets et décrivant leur contenu, puis déroulez le programme pas à pas en mettant à jour le contenu des objets dans vos schémas.

Exécutez-le afin de vérifier vos résultats.
Résultat d'exécution :

```
f3: csurf = blanc, ccont = jaune
f2: csurf = bleu, ccont = noir
FigureP1::nfigs = 3
f2.nfigs = 3
f1.nfgCN = 1
f4.nfgSB = 0
```

1.4 Les tirelires / qztirelire

Toto achète deux tirelires et les remplit avec le même montant. Il écrit ensuite un programme orienté objet pour simuler la gestion de ses tirelires.

Voici le programme : @[qztirelire1.cpp]

```
#include <iostream>
using namespace std;
#include "Tirelire1.hpp"
#include "Tirelire2.hpp"

int main(int, char*[])
{
  gestion1();
  gestion2();  //<- AJOUT
  gestion3();  //<- AJOUT
}
```

Voici le Fichier Implémentation : @[Tirelire1.hpp]

```
#ifndef TIRELIRE_CLASS
#define TIRELIRE_CLASS
#include <iostream>
#include <vector>
using namespace std;

// Représente des tirelires
class Tirelire1
{
  public:
    // Constructeur normal
    Tirelire1()
    : m_v()
    {}

    // Ajoute le montant à la tirelire
    void gagner(double montant)
    { m_v.push_back(montant); }

    // Retire le montant de la tirelire
    void depenser(double montant)
    {
      double somme = 0.;
      do{
        somme += m_v.back();
        m_v.pop_back();
      } while (somme < montant);

      if (somme > montant)
      { m_v.push_back(somme - montant); }
    }

    // Retourne la somme contenue dans la tirelire
    double eval() const
    {
      if (m_v.empty()) { return 0.; }

      double s = 0.;
```

```
      for (unsigned ix = 0; ix < m_v.size(); ++ix)
      { s += m_v[ix]; }

      return s;
    }

  private:
    vector<double> m_v;
};

// Remplit une tirelire
void remplir(Tirelire1& b)
{
  b.gagner(10.0);
  b.gagner(5.0);
  b.gagner(2.0);
  b.gagner(0.5);
  b.gagner(100.0);
}

// Retire d'une tirelire
void retirer1(Tirelire1& b)
{ b.depenser(15.0); }

// Retire d'une tirelire
void retirer2(Tirelire1& b)
{
  retirer1(b);
  b.depenser(10.0);
}

// Effectue une gestion de tirelires
void gestion1()
{
  typedef Tirelire1 Tirelire;

  // Instancie une Tirelire
  Tirelire b1;

  // Effectue quelques opérations
  remplir(b1);

  // Une autre Tirelire
  Tirelire &b2 = b1;
  retirer1(b1);
  retirer2(b2);

  // Affiche les sommes contenues dans les tirelires
  cout << "Gestion1 :\n";
  cout << "b1 contient " << b1.eval() << " euros\n";
  cout << "b2 contient " << b2.eval() << " euros\n";
}
#endif
```

Son code compile et s'exécute sans message d'erreur.
Qu'affiche-t-il ? Justifier votre réponse en **explicitant** les principales étapes.

Il y a deux problèmes importants avec ce programme : l'un dans la classe `Tirelire1` et l'autre dans le programme principal. Proposez des corrections à la fois sous forme d'explications en français **et** de code.

Proposez une méthode `depenser` plus simple et plus efficace (texte en français sur comment on pourrait la coder autrement). Implémentez votre méthode et/ou nouvelle classe `Tirelire2`.

Testez. Exemple d'exécution :

```
Gestion1 :
b1 contient 77.5 euros
b2 contient 77.5 euros

Gestion2 :
b1 contient 102.5 euros
b2 contient 92.5 euros

Gestion3 :
b1 contient 10 euros
OUPS... montant negatif !
b1 contient 5 euros
b1 contient 4.5 euros
OUPS... montant negatif !
b1 contient 6 euros
OUPS... montant negatif !
b1 contient -4 euros
```

2 Appliquer le cours

2.1 Classe Hasard / pghasard

Créez une classe `Hasard` modélisant des générateurs de nombres pseudo-aléatoires.

Écrivez un constructeur par défaut.

Écrivez une méthode partagée `randomize` qui initialise le générateur.

Écrivez une méthode partagée `random` qui renvoie un entier pseudo-aléatoire.

Écrivez une méthode partagée `random(n)` qui renvoie un entier pseudo-aléatoire dans `[0..n[`.

Écrivez une méthode partagée `frandom` qui renvoie un réel pseudo-aléatoire.

Écrivez un programme qui saisit la borne supérieure des entiers pseudo-aléatoires dans un entier `vmax` et le nombre de tirages dans un entier `n`.

Instanciez un générateur de générateur de nombres pseudo-aléatoires.

Effectuez `n` tirages d'entiers dans l'intervalle `[0..vmax[`.

Testez. Résultats d'exécution montrant que le caractère pseudo-aléatoire :

```
Borne superieure vmax? 10
Nombre de tirages? 20
3 0 7 8 4 3 0 8 8 3 2 5 4 1 6 5 4 1 5 5
```

2.2 Jeu de la fourchette OO / pgjeufourche

Objectif
Cet exercice réalise une classe du jeu de la fourchette : l'algorithme tire un entier au hasard entre 1 et 100 puis demande à l'utilisateur de le trouver.

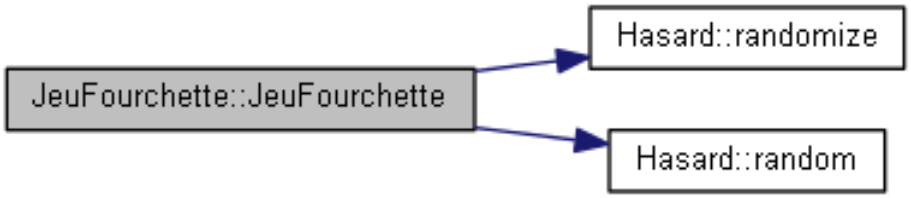
2.2.1 Classe JeuFourchette

Soit la classe `Hasard` modélisant des générateurs de nombres pseudo-aléatoires.

C++ @[Hasard.hpp] @[Hasard.cpp]

Écrivez une classe `JeuFourchette` comprenant l'entier `mystere` à trouver.

Écrivez un constructeur à un paramètre `vmax` qui tire au hasard un entier dans `[0..vmax[` pour initialiser son attribut.



Écrivez un accesseur `getMystere` de l'entier à trouver.

Écrivez une méthode `compare(nombre)` qui teste et renvoie -1, 0 ou 1 selon que l'entier mystère est inférieur, égal, supérieur à un entier `nombre`.

Écrivez une méthode `afficher` qui affiche l'entier mystère.

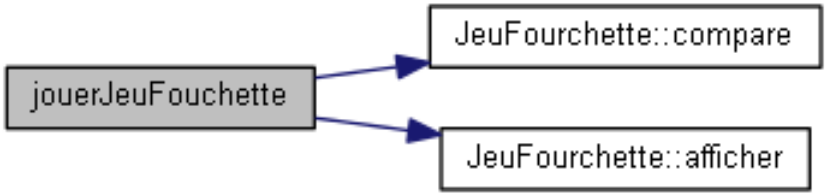
2.2.2 Programme de test

Écrivez une procédure `jouerJeuFourchette(nmax,maxessais)` qui réalise le jeu de la fourchette en tirant un entier entre 1 et `nmax` et demande de le trouver en au plus `maxessais` tentatives. Dans le cas où l'utilisateur trouve le nombre mystère, affichez le message :

```
==> Bravo, Vous avez trouve en [nessais] essai(s)
```

sinon affichez le message :

```
==> Désolé, Mystère = [mystere]
```

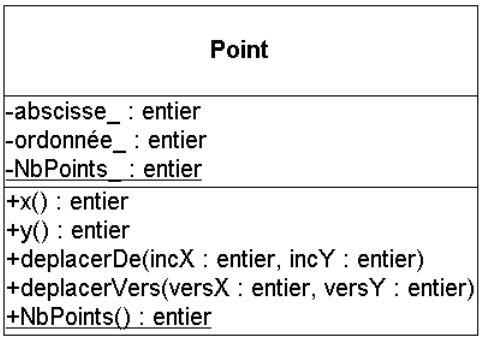


Testez. Exemple d'exécution avec `nmax=100` et `maxessais=7` :

```
Trouvez un entier dans [1..100] en 7 tentatives max
Votre entier? 50
C'est moins
Votre entier? 25
C'est plus
Votre entier? 40
C'est plus
Votre entier? 45
C'est moins
Votre entier? 42
==> Bravo, Vous avez trouve en 5 essai(s)
```

2.3 Classe Point (avec compteur d'instances)

Définissez une classe `Point` telle qu'explicitée sur la figure ci-dessous :



Écrivez un programme de test.

Testez.

3 Approfondir le cours