EFREI

L2 TP3 C++

Relations de composition et d'héritage public, polymorphisme

PARTIE I: Dessine moi une image

- On souhaite modéliser un ensemble de figures géométriques.
- Parmi les figures géométriques, on s'intéresse plus particulièrement aux lignes droites et aux cercles.
- On souhaite pouvoir placer les figures géométriques sur une image (les doublons sont autorisés).
- Une image peut également contenir des images.

Compléter et éventuellement modifer les sources des classes fournies :

Figure, Image, Ligne, Cercle, Point.

PARTIE II: Vérification de conditions

Classe abstraite Condition

Créer une classe abstraite **Condition** avec une seule méthode booléenne **verif** qui prend une forme géométrique en paramètre.

Classe EstPetite

- Créer une classe **EstPetite** dérivée de la classe **Condition**
- Ecrire le constructeur mémorisant un **seuil** réel que la **surface** de la forme ne devra pas dépasser afin que la méthode **verif** renvoie **true**.

Classe EstUn

- Créer une classe **EstUn** dérivée de la classe **Condition**
- Ecrire le constructeur mémorisant une **forme** "témoin" dont le type constaté devra être identique à celui de la forme passée à la méthode **verif** pour que celle-ci renvoie **true**.

  indication: Comment trouvé le type d'un objet en C++

Classe Non

- Créer une classe **Non** dérivée de la classe **Condition**
- Ecrire le constructeur mémorisant une **condition** dont la négation sera renvoyée par la méthode **verif**.

Classe Et

- Créer une classe **Et** dérivée de la classe **Condition**
- Ecrire le constructeur mémorisant 2 conditions dont la conjonction (ET logique) sera renvoyée par la méthode **verif.**

PARTIE III: Classe de Filtrage

- Créer une classe **Filtrage** sans attribut qui ne contiendra que des **méthodes de classe statiques.**
- Ecrire la méthode **getUneForme** à 2 paramètres (les coordonnées du centre) qui crée une forme et la retourne. La forme est créée aléatoirement parmi les 3 possibles avec des paramètres également alétoires.
- Ecrire la méthode **creerFormes** à un paramètre n de type entier (le nombre de formes) qui doit créer, remplir et retourner une liste de n formes créées par la méthode getUneForme et situées sur une diagonale (x = y = 10 x le n° de la forme).
- Ecrire la méthode **compterSi** qui retourne le nombre d'éléments de la **collection** passée en premier paramètre vérifiant la **condition** passée en second paramètre.
- Ecrire la méthode **supprimerSi** qui supprime dans la **collection** passée en premier paramètre les éléments qui vérifient la **condition** passée en second paramètre. Cette méthode retourne **true** si au moins un élément a été supprimé. On programmera 2 versions de supprimerSi : uen suppression superficielle puis une supression profonde.
- Ecrire la méthode **essai** qui devra, en utilisant les méthodes ci-dessus:
  1) déclarer et créer une collection aléatoire de 10 formes
  2) afficher cette collection en une seule ligne
  3) déclarer et créer la condition 1 "n'est pas un cercle"
  4) déclarer et créer la condition 2 "est plus petite que 1 000 et n'est pas un cercle"
  5) afficher le nombre de formes de la collection vérifiant la condition 1 puis la 2
  6) supprimer dans la collection les formes vérifiant la condition2
  7) afficher à nouveau la collection

PARTIE IV: Retour sur Image

- Si ce n'est déjà fait, faites en sorte que la surcharge de l'opérateur << ne soit présent que dans la classe Figure, où il appelle une fonction virtuelle d'affichage.
- La classe Image a un tableau de pointeurs en attribut. Par suite, un constructeur de copie est à écrire. Au préalable, on peut envisager d'écrire une fonction virtuelle de copie dans toutes les classes de la hiérarchie.

PARTIE V: Les tableaux de pointeurs aux oubliettes! Vive les classes containers de la STL!

- L'utilisation d'un tableau de pointeurs de type Figure est quelque peu lourde à gérer.
  Remplacer ce tableau par une liste générique de la STL et modifier votre code en conséquence. Il est conseillé de réaliser cela dans un nouveau projet.

ANNEXE 1 : typeid, dynamic_cast, RTTI

The typeid operator provides a program with the ability to retrieve the actual derived type of the object referred to by a pointer or a reference. This operator, along with the dynamic_cast operator, are provided for runtime type identification (RTTI) support in C++.

The typeid operator requires runtime type information (RTTI) to be generated, which must be explicitly specified at compile time through a compiler option.

The typeid operator returns an lvalue of type const std::type_info that represents the type of expression *expr*. You must include the standard template library header <typeinfo> to use the typeid operator.

If *expr* is a reference or a dereferenced pointer to a polymorphic class, typeid will return a type_info object that represents the object that the reference or pointer denotes at run time. If it is not a polymorphic class, typeid will return a type_info object that represents the type of the reference or dereferenced pointer. The following example demonstrates this:

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

struct A { virtual ~A() { } };
struct B : A { };

struct C { };
struct D : C { };

int main() {
  B bobj;
  A* ap = &bobj;
  A& ar = bobj;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;

  D dobj;
  C* cp = &dobj;
  C& cr = dobj;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
}
```

The following is the output of the above example:

```
ap: B
ar: B
cp: C
cr: C
```

Classes A and B are polymorphic; classes C and D are not. Although cp and cr refer to an object of type D, typeid(*cp) and typeid(cr) return objects that represent class C.

ANNEXE 2 : La classe list de la STL (standart template library)

Cet exemple montre comment insérer les valeurs 4, 5, 4, 1 dans une liste et comment afficher son contenu :

```
#include <list>
#include <iostream>

int main() {
        std::list<int> ma_liste;
        ma_liste.push_back(4);
        ma_liste.push_back(5);
        ma_liste.push_back(4);
        ma_liste.push_back(1);

        std::list<int>::const_iterator
          lit (ma_liste.begin()),
          lend(ma_liste.end());

        for(;lit!=lend;++lit) std::cout << *lit << ' ';
        std::cout << std::endl;
        return 0;
}
```

Utilisation d'une boucle for_each

```
#include <list>
#include <algorithm>

// le foncteur de la list : le paramètre est de type celui des éléments de la liste
void affiche(Figure * f)
{
   f->afficher(); // fonction virtuelle de la classe abstraite Figure
   cout << endl;
}

void testCreerFigures()
{
   list<Figure *> figures = Filtrage::creerFigures(10);

   // FOREACH ITERATOR
   list<Figure *>::const_iterator
   lit (figures.begin()),
   lend(figures.end());

   for( ; lit != lend; ++lit)    {
      cout << (*lit) << endl; // affiche des adresses
   }

   std::for_each(figures.begin(), figures.end(), affiche);
}
```