Grands Moulins

9/01/2024 - 13h-15h

Master 1 - Langage Objet Avancé - C++

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention.
- Le barème est indicatif. **Aucun document n'est autorisé**.

Exercice 1 [3 points] Ce sont des questions faciles dans le contexte d'une compilation séparée. Le barème tiens compte essentiellement du temps qu'il vous faut pour les lire et pour y répondre proprement:

- La séquence suivante est-elle acceptable ? Si elle ne l'est pas, expliquez et proposez une solution.

```
// fichier File1.hpp
  #include "File2.hpp"
  void functionA();

// fichier File2.hpp
  #include "File1.hpp"
  void functionB();

// fichier Main.cpp
  #include "File1.hpp"
```

- La séquence suivante est-elle acceptable ? Si elle ne l'est pas, expliquez et proposez une solution.

```
// fichier File1.hpp
  void functionA() {};

// fichier File2.hpp
  #include "File1.hpp"
  void functionB() {};

// fichier Main.cpp
  #include "File1.hpp"
  #include "File2.hpp"
```

- La séquence suivante est-elle acceptable ? Si elle ne l'est pas, expliquez et proposez une solution.

```
// Makefile
CC = g++
CFLAGS = -Wall -c

all: main

main: main.o functions.o
  $(CC) -o main main.o functions.o

main.o: main.cpp functions.hpp
  $(CC) $(CFLAGS) main.cpp

functions.o: functions.cpp
  $(CC) $(CFLAGS) functions.cpp

clean:
  rm -f *.o main
```

- La séquence suivante est-elle acceptable ? Si elle ne l'est pas, expliquez et proposez une solution.

```
// fichier File1.hpp
  void greet();

// fichier File1.cpp
  #include <iostream>
  using namespace std;
  void greet() { cout << "Hello " << endl;}

  void display() {
    greet();
    cout << " World !" << endl;
  }

// fichier main.cpp
  #include "File1.hpp"
  int main() {
   greet();
   display();
  }
```

Exercice 2 [4 points] Quels sont les affichages produits par le main suivant ?
(S'il y a une erreur dites laquelle, et passez au test suivant)

```
class A {
  public :
  A() {cout << "Ctor A; " ;}
  A (const A& a) {cout << "Copy A; " ;}
  virtual ~A() {cout << "Dtor A ; " ;}
};

class X : public A {
  public :
  X() {cout << "Ctor X; ";}
  X (const X& a) {cout << "Copy X; " ;}
  virtual ~X() {cout << "Dtor X; " ;}
};

void f(A a) {cout << "f(A); " ;}
void f(A *a) { cout << "f(A*); " ;}

class B {
  public :
  B() {cout << "Ctor B; " ;}
  B (const B& b) {cout << "Copy B; " ;}
  virtual ~B() {cout << "Dtor B; " ;}
};

class Y : public A, public B {
  public :
  Y() {cout << "Ctor Y; " ;}
  Y (const Y& y) {cout << "Copy Y; " ;}
  virtual ~Y() {cout << "Dtor Y; " ;}
};

void g(Y y) { f(&y); cout << "g(Y); " ;}
void g(Y *y) { f(*y); cout << "g(Y*); " ;}

```

```
int main(){
  int i=0;
  cout << " resultat " << ++i << " : " ;  A a; f(a); cout << endl;
  cout << " resultat " << ++i << " : " ;  A a2; f(&a2); cout << endl;
  cout << " resultat " << ++i << " : " ;  X x; f(x); cout << endl;
  cout << " resultat " << ++i << " : " ;  X* x2 = new X; f(x2); delete x2; cout << endl;
  cout << " resultat " << ++i << " : " ;  A* a3 = new X; f(a3); delete a3; cout << endl;
  cout << " resultat " << ++i << " : " ;  Y* y = new Y;  g(y); delete y; cout << endl;
  cout << " resultat " << ++i << " : " ;  B* b = new Y; g(b); cout << endl;
  cout << " resultat " << ++i << " : " ;  A* a4 = new Y; f(*a4); cout << endl;
  // ne pas afficher les destructions qui suivent la fin du bloc
}
```

Exercice 3 [7 points] On vous donne le code suivant :

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class A {
   static vector<A*> all;
   int val=0;
 public :
    A(int x=0) :val{x} {}
    A next() { return val+1;}
    static void process(A& ref) { // on ajoute son pointeur a all, sans doublons
        auto it = std::find(all.begin(), all.end(), &ref);
        if (it == all.end()) all.push_back(&ref);
    }
    virtual ~A() {
        cout << "destruction de " << val << endl;
        auto it = std::find(all.begin(), all.end(), this);
        if (it != all.end()) delete (*it);
    }
};

vector<A*> A::all;

int main() {
    A a;
    A::process(a);
    A::process( *(new A()));
    cout << A::all.size() << endl;
    for (int i = 0; i < 2; ++i) {
        A b = a.next();
        A::process(b);
    }
    cout << A::all.size() << endl;
}
```

1. Expliquez pourquoi la ligne 11 compile (c'est en particulier l'absence d'un mot clé qui rend les choses possibles, on attend dans vos explications que vous précisiez duquel il s'agit).

2. D'ordinaire on n'aime pas trop voir `auto` utilisé, mais on peut le tolérer si on sait vraiment à quel type il se substitue ici. Pouvez vous le dire ?

3. Il y a une erreur à la ligne 19 qui intervient sur l'exécution de delete (*it). On vous donne l'affichage obtenu. Expliquez clairement de quoi il s'agit.

```
...
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1
destruction de 1

RUN FINISHED; Segmentation fault; core dumped;
```

4. On remplace `delete(*it)` par `all.erase (it)`, ce qui permet de retirer l'élément trouvé par find. Il n'y a plus d'erreur. Expliquez pourquoi les affichages sont les mêmes en ligne 29 et en ligne 34 (ils valent 2).

5. Lorsqu'on quitte vraiment le `main`, c'est à dire juste après la ligne 35, le vecteur `all` n'est pas vide, il a une taille 1. Il manque toute la prise en charge de la destruction de ce qu'il contient.

   (a) Expliquez à quoi correspond ce 1

   (b) Expliquez pourquoi cette destruction ne peut pas intervenir au niveau de la ligne 34

   (c) On veut une solution avec un type personnel, une nouvelle classe `MyVec` pour `all`, où la gestion de la destruction sera mieux adaptée . On change la ligne 7 en `static MyVec all;` et la ligne 23 en `MyVec A::all;` et rien d'autre au code fourni.

      i. Proposez un `MyVec.hpp` (sans code donc) minimal

      ii. Ecrivez son destructeur

Exercice 4 [6 points]

Dans cet exercice, on mélange un cas particulier du Pattern Composite avec le Pattern Visitor. Techniquement les choses s'écrivent simplement, il faut prendre le temps de comprendre ce que l'on vous demande en recoupant les informations fournies. Notez qu'il est absolument **interdit** d'utiliser des instructions de cast.

On vous donne (figure 1) une structure composite arborescente dont la sémantique (ce qu'elle représente) est indéfinie. Ses feuilles portent des entiers, ses noeuds internes sont soit unaires soit binaires. Cette structure est "fermée" au sens où on est censé la prendre telle quelle est : il vous est impossible d'y écrire davantage de code, et elle n'est pas sensée évoluer (n'en écrivez pas de sous classes).

Le concepteur a cependant prévu une "ouverture" en mettant en place le pattern `Visitor` (uml en figure 2). On vous fourni également l'exemple du code d'un Client (figure 3)

1. Combien y a t'il d'éléments concrets dans notre structure ?

2. Dans le code (Figure 1) on retrouve dans chaque sous-classe des versions de la méthode accept qui sont identiques au mot près, ce n'est pas une erreur. Expliquez pourquoi on ne peut pas se contenter de ne l'écrire qu'une seule fois dans Node et enlever les autres.

```
#include "Visitor.hpp"
// rq : les champs sont public pour s'epargner l'ecriture des accesseurs/modifieurs
class Node {
  public :
    virtual void accept(Visitor &)=0;
};
class Binaire : public  Node {
  public :
    Node *left, *right;
    Binaire (Node *g, Node *d):left{g},right{d}{}
    void accept(Visitor &v) { v.visit(this); }
};
class Unaire : public Node {
  public :
    Node * next;
    Unaire (Node * fils):next{fils} {}
    void accept(Visitor &v) { v.visit(this); }
};
class Feuille : public Node {
  public :
    int val;
    Feuille (int v=0):val{v}{}
    void accept(Visitor &v) { v.visit(this); }
};
```

Figure 1: code de la structure

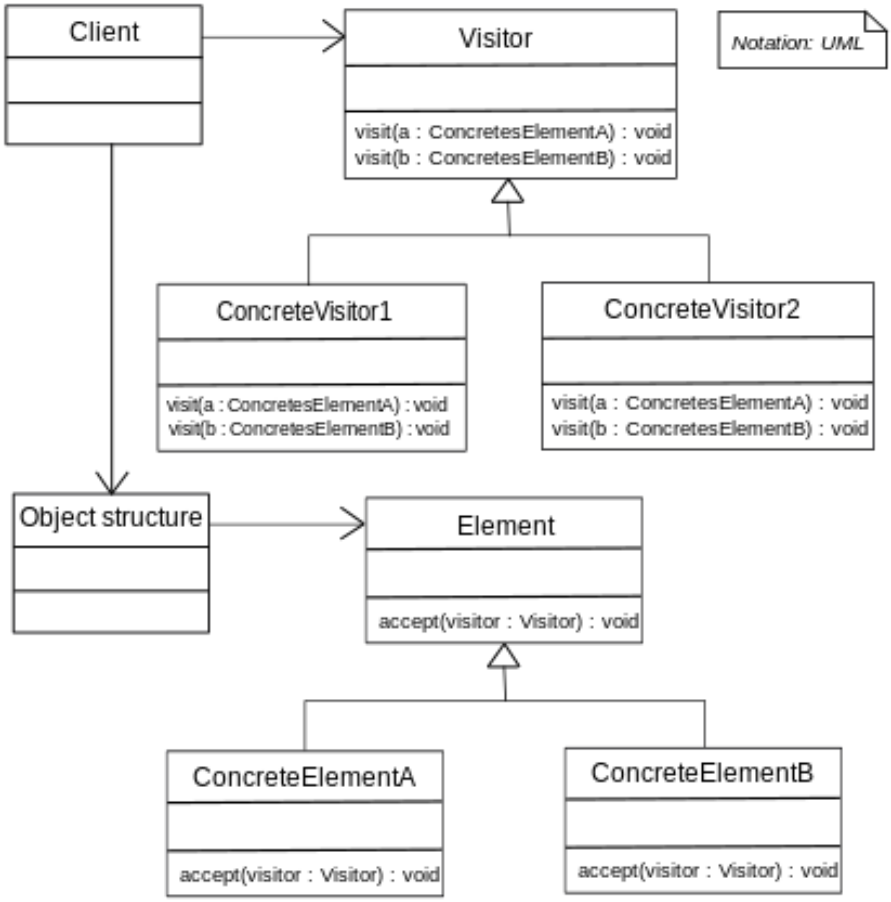

Figure 2: uml du patron visiteur.

```
class Client {
  int main() {
    Feuille f1{1},f2{2}, f3{3},f4{4};
    Binaire n1{&f1,&f2}, n2{&f3,&f4};
    Unaire n3{&n2};
    Binaire all{&n3,&n1};

    Display v1; // un visitor
    all.accept(v1);

    EtNot v2; // un autre visitor
    all.accept(v2);
    cout << v2.getResult() << endl;
  }
};
```

Figure 3: Exemple d'utilisation

3. Sachant que la classe Visitor est abstraite pure, écrivez la. (Si, en croisant les information du diagramme uml et le code de la figure 1, vous avez un doute sur des signatures : fiez vous au code, qui est plus précis)

4. Dans l'exemple (Figure 3) Display est un Visitor qui permet en ligne 9 de produire l'affichage : `binaire unaire binaire feuille feuille binaire feuille feuille`

   Il correspond à un parcours préfixe de l'arbre all. Ecrivez la classe `Display` [1].

5. EtNot (Figure 3, ligne 11) est un second Visitor qui interprète les noeud binaires comme étant un "et" logique, les noeuds unaires comme étant une négation, alors qu'une feuille sera interprétée comme true ssi elle est impaire. Ecrivez la classe EtNot qui calcule la valeur représentée par l'expression de l'arbre all. (Remarquez qu'elle stocke un résultat, accessible avec la méthode getResult)

6. Ecrivez l'un des visiteurs suivant [2] :

   - Clone : un clone que l'on récupère grâce à getResult()
   - Symetrie qui, sur place, inverse le sens des branches de la structure pendant la visite

[1] Pour rester concis, ne distinguez pas le hpp du cpp et écrivez tout ensemble, sur le modèle de la figure 1

[2] on vous laisse le choix mais n'en écrivez qu'un seul