3/01/2023 - 9h30-11h30

Langage Objet Avancé - C++

Consignes :

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention : il vaut mieux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.
- Les 4 premiers exercices sont un peu du type QCM (il faut quand même prendre le temps de la réflexion), les 2 derniers exercices sont plus ouverts, il vous faudra écrire du code.

Exercice 1 [3 points]

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3    const int value;
   public :
5    X(int v) : value {v} { value += 10; }
     const int &getValue () { return value; }
7  friend void f ();
   };
9
   void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
   };
19
   int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
   };
```

Exercice 2 [2 points]

Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```

Exercice 3 [3 points]

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() + " by f() in E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;  // (1)
 cout << y.h() << endl;  // (2)
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

Exercice 4 [3 points]

Voici deux situations qu'on veut absolument représenter à l'aide de l'**héritage multiple** :

- Au niveau des Animaux on stocke un attribut pour mémoriser un ADN. Parmi les Animaux on distingue la sous-classe des Carnivores, et celle des Herbivores. Les Omnivores sont eux à la fois des Carnivores et des Herbivores. Vous partirez de :

```
class Animal {
 public :
  const string adn;
  Animal (string x) :adn {x} {}
};
```

Ecrivez les classes Carnivore, Herbivore, Omnivore ainsi qu'un exemple dans un main de construction et d'affichage de l'adn unique d'un omnivore particulier.

- Au niveau des Composants on stocke un attribut pour mémoriser un numéro de série. Parmi les Composants on distingue les sous-classes Clavier et Ecran. Par héritage un Ordi est à la fois un Clavier et un Ecran. Vous partirez de :

```
class Composant {
 private :
  static int NB;
 public :
  const string num;
  Composant ():num{to_string(NB++)}{}
};
int Composant::NB=0;
```

qui permet une numérotation automatique à partir d'un compteur statique.

Ecrivez les classes Ecran, Clavier, Ordi ainsi qu'un exemple dans un main de construction d'un ordi et d'affichage des numéros de série différents de son clavier et de son écran.

Exercice 5 [5 points]

On rappelle rapidement comment fonctionne un itérateur sur les collections en général en l'illustrant sur la classe vector. Chaque collection définit deux classes internes iterator et reverse_iterator. Dans l'exemple suivant, v.begin(),v.end() et v.rbegin(), v.rend() en sont respectivement des instances.

```
vector<int> v {1,2,3};  // pour l'exemple
// pour une iteration de gauche a droite :
for(vector<int>::iterator i{v.begin()} ; i != v.end(); ++i) cout << *i;
cout << endl;
// pour une iteration de droite a gauche :
for(vector<int>::reverse_iterator i{v.rbegin()} ; i != v.rend(); ++i) cout << *i;
cout << endl;
```

L'affichage résultant est

```
123
321
```

D'un autre coté nous avons aussi la possibilité d'écrire :

```
for (int i: v) cout << i;
```

la syntaxe du for avec les deux points est une sorte de macro syntaxique qui est traduite vers la première des formes que nous avons données ci dessus, ligne 2-3 : le type iterator implicitement utilisé est celui du type interne de la collection parcourue, begin() et end() seront les méthodes invoquées pour obtenir les bornes. Ce qui est affiché est ici : 123, dans le sens "gauche-droite".

On souhaite introduire une forme syntaxique maison pour parcourir simplement notre vecteur dans l'autre sens, mais il n'y a pas de moyen de redéfinir ce qu'il se passe avec le symbole ":" on penche donc pour une solution qui aurait cette forme :

```
for (int i:wrap{v}) { cout << i ;}
```

Proposez une solution en ne vous occupant que du cas des vecteurs d'entiers (c.à d ne cherchez pas à faire de templates).

Indications :

- wrap est une nouvelle classe, qu'il vous faut écrire
- pour que le code soit lisible sur votre copie, écrivez définition et déclaration ensemble (ne séparez pas hpp et cpp)
- soyez rassurés : la macro for (:) se comportera avec wrap de la même façon qu'elle le fait pour les vector ou pour une autre une collection.

Exercice 6 [5 points]

Afin de mettre en pratique le patron Décorateur, nous allons concevoir une application qui permet de gérer la vente de desserts. Elle doit permettre d'afficher le nom complet du dessert (avec ses options) et d'obtenir son prix total. Les clients (qu'on ne représentera pas) ont potentiellement le choix entre plusieurs desserts de base (disons au moins crêpes ou gaufres). Sur chaque base ils peuvent ajouter un nombre quelconque d'ingrédients, par exemple du chocolat ou de la chantilly. Nous dirons qu'une crêpe (nature) coûte 1.50 euros et une gaufre (nature) 1.80 euros. L'ajout de chocolat sera facturé 0.20 euros et 0.50 euros pour de la chantilly.

1. Reprenez l'UML de figure 1 en l'adaptant aux desserts. Faites apparaître en particulier : les classes et attributs qui vous semblent indispensables (précisez leur visibilité), les constructeurs significatifs, les méthodes permettant d'obtenir nom et prix avec leurs spécifications exactes (virtuelle, abstraite, const ...).
2. Ecrivez le code correspondant (Ne séparez pas hpp et cpp pour que cela reste lisible. Pour info ma correction fait une trentaine de courtes lignes)
3. Ecrivez un main vous permettant d'afficher la description et le prix d'une crêpe au chocolat et à la chantilly avec un second supplément chocolat.

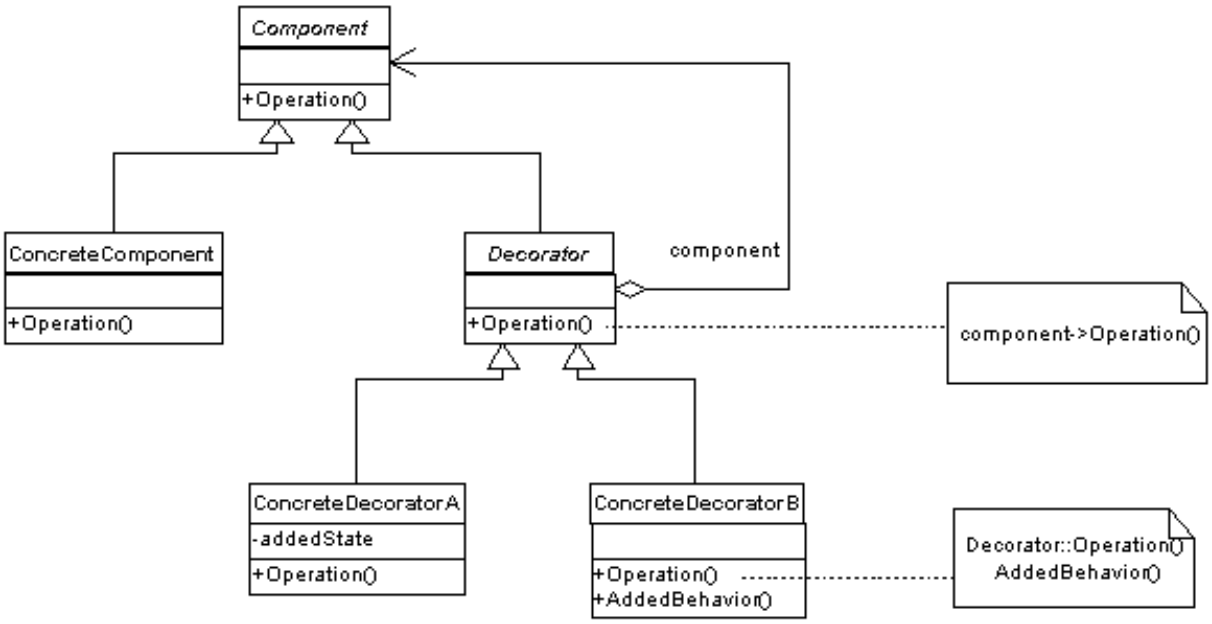

Figure 1: Rappel UML du patron décorateur.