MOOC Intro POO C++

Corrigés semaine 6

Les corrigés proposés correspondent à l'ordre des apprentissages : chaque corrigé correspond à la solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante.

Exercice 20 : Animaux en peluche

> Cet exercice correspond à l'exercice n°63 (pages 160 et 351) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

La classe `Animal` ne présente aucune difficulté :

```
class Animal {
public:
   void affiche() const;
protected:
   string nom;
   string continent;
};

void Animal::affiche() const {
   cout << "Je suis un " << nom << " et je vis en " << continent << endl;
}
```

pas plus que les deux autres classes :

```
class EnDanger {
public:
   void affiche() const;
protected:
   unsigned int nombre;
};

void EnDanger::affiche() const {
    cout << "Il ne reste que " << nombre
         << " individus de mon espece sur Terre" << endl;
}

class Gadget {
public:
   void affiche() const;
   void affiche_prix() const;
protected:
   string nom;
   double prix;
};

void Gadget::affiche() const {
  cout << "Mon nom est " << nom << endl;
}

```

```
void Gadget::affiche_prix() const {
  cout << "Achetez-moi pour " << prix
       << " francs et vous contribuerez a me sauver!" << endl;
}
```

L'ajout des constructeurs et destructeurs se fait aussi trivialement :

```
class Animal {
public:
   Animal(string, string);
   ~Animal();
   void affiche() const;
protected:
   string nom;
   string continent;
};

void Animal::affiche() const {
  cout << "Je suis un " << nom << " et je vis en " << continent << endl;
}

Animal::Animal(string nom, string continent)
  : nom(nom), continent(continent)
{
  cout << "Nouvel animal protege" << endl;
}

Animal::~Animal() {
  cout << "Je ne suis plus protege" << endl;
}

// ************************************************************************
class EnDanger {
public:
   void affiche() const;
   EnDanger(unsigned int);
   ~EnDanger();
protected:
   unsigned int nombre;
};

EnDanger::EnDanger(unsigned int nombre)
  : nombre(nombre)
{
  cout << "Nouvel animal en danger" << endl;
}

EnDanger::~EnDanger() {
  cout << "ouf! je ne suis plus en danger" << endl;
}

void EnDanger::affiche() const {
  cout << "Il ne reste que " << nombre
       << " individus de mon espece sur Terre" << endl;
}

// ************************************************************************
class Gadget {
public:
   void affiche() const;
   void affiche_prix() const;
   Gadget(string, double);
   ~Gadget();
```

```
protected:
   string nom;
   double prix;
};

Gadget::Gadget(string nom, double prix)
    : nom(nom), prix(prix)
{
    cout << "Nouveau gadget" << endl;
}

Gadget::~Gadget() {
  cout << "Je ne suis plus un gadget" << endl;
}

void Gadget::affiche() const {
  cout << "Mon nom est " << nom << endl;
}

void Gadget::affiche_prix() const {
  cout << "Achetez-moi pour " << prix
       << " francs et vous contribuerez a me sauver !"
       << endl;
}
```

Définissez une classe `Peluche` *héritant des classes* `Animal`, `EnDanger` *et* `Gadget` :

```
class Peluche : public Animal, public EnDanger, public Gadget {
};
```

Dotez votre classe `Peluche` *d'une méthode* `etiquette` *[...] codée au moyen des méthodes* `affiche` *et* `affiche_prix` *des super-classes* :

```
class Peluche : public Animal, public EnDanger, public Gadget {
public:
   void etiquette() const;
};

void Peluche::etiquette() const {
  cout << "Hello," << endl;
  Gadget::affiche();
  Animal::affiche();
  EnDanger::affiche();
  affiche_prix();
  cout << endl;
}
```

puis les constructeurs et destructeurs :

```
class Peluche : public Animal, public EnDanger, public Gadget {
public:
   void etiquette() const;
   Peluche(string, string, string, unsigned int, double);
   ~Peluche();
};

Peluche::Peluche(string nom_animal, string nom_gadget,
                 string continent, unsigned int nombre, double prix)
   : Animal(nom_animal, continent), EnDanger(nombre),
     Gadget(nom_gadget, prix)
{ cout << "Nouvelle peluche" << endl; }

Peluche::~Peluche() {
  cout << "Je ne suis plus une peluche" << endl;
```

```
}
```

Voici le code complet :

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

// ***************************************************************************
class Animal {
public:
   Animal(string, string);
   ~Animal();
   void affiche() const;
protected:
   string nom;
   string continent;
};

void Animal::affiche() const {
  cout << "Je suis un " << nom << " et je vis en " << continent << endl;
}

Animal::Animal(string nom, string continent)
  : nom(nom), continent(continent)
{
  cout << "Nouvel animal protege" << endl;
}

Animal::~Animal() {
  cout << "Je ne suis plus protege" << endl;
}

// ***************************************************************************
class EnDanger {
public:
   void affiche() const;
   EnDanger(unsigned int);
   ~EnDanger();
protected:
   unsigned int nombre;
};

EnDanger::EnDanger(unsigned int nombre)
  : nombre(nombre)
{
  cout << "Nouvel animal en danger" << endl;
}

EnDanger::~EnDanger() {
  cout << "ouf! je ne suis plus en danger" << endl;
}

void EnDanger::affiche() const {
  cout << "Il ne reste que " << nombre
       << " individus de mon espece sur Terre" << endl;
}

// ***************************************************************************
class Gadget {
public:
  void affiche() const;
   void affiche_prix() const;
```

```
   Gadget(string, double);
   ~Gadget();
protected:
   string nom;
   double prix;
};

Gadget::Gadget(string nom, double prix)
  : nom(nom), prix(prix)
{
  cout << "Nouveau gadget" << endl;
}

Gadget::~Gadget() {
  cout << "Je ne suis plus un gadget" << endl;
}

void Gadget::affiche() const {
  cout << "Mon nom est " << nom << endl;
}

void Gadget::affiche_prix() const {
  cout << "Achetez-moi pour " << prix
       << " francs et vous contribuerez a me sauver !"
       << endl;
}

// ************************************************************************
class Peluche : public Animal, public EnDanger, public Gadget {
public:
   void etiquette() const;
   Peluche(string, string, string, unsigned int, double);
   ~Peluche();
};

Peluche::Peluche(string nom_animal, string nom_gadget, string continent,
   unsigned int nombre, double prix)
  : Animal(nom_animal, continent), EnDanger(nombre), Gadget(nom_gadget, prix)
{
  cout << "Nouvelle peluche" << endl;
}

Peluche::~Peluche() {
  cout << "Je ne suis plus une peluche" << endl;
}

void Peluche::etiquette() const
{
  cout << "Hello," << endl;
  Gadget::affiche();
  Animal::affiche();
  EnDanger::affiche();
  affiche_prix();
  cout << endl;
}

// ************************************************************************
int main()
{
    Peluche panda ("Panda", "Ming", "Asie", 200, 20.0);
    Peluche serpent("Cobra", "ssss", "Asie", 500, 10.0);
    Peluche toucan ("Toucan", "Bello", "Amerique du Sud", 1000, 15.0);

```

```
123     panda.etiquette();
124     serpent.etiquette();
125     toucan.etiquette();
126 
127     return 0;
128 }
```

Exercice 21 : Employés

Cet exercice correspond à l'exercice n°64 (pages 162 et 354) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Codez une classe abstraite `Employe` [...] :

```
class Employe {
public:
   virtual double calculer_salaire() const = 0;
protected:
   string prenom;
   string nom;
   unsigned int age;
   string date;
};
```

Dotez également votre classe d'un constructeur [...] et d'un destructeur virtuel vide.

```
class Employe {
public:
   Employe(string prenom, string nom, unsigned int age, string date)
     : nom(nom), prenom(prenom), age(age), date(date) {}
   virtual ~Employe() {}
   virtual double calculer_salaire() const = 0;
protected:
   string prenom;
   string nom;
   unsigned int age;
   string date;
};
```

Calcul du salaire

Codez une hiérarchie de classes pour les employés en respectant les conditions suivantes [...]

La première constatation que l'on peut faire est que les deux commerciaux (vendeur et représentant) ont une base commune de calcul. On peut créer une super-classe commune à ces deux classes.

Appelons-la par exemple `Commercial`. Elle reste une classe abstraite et il n'y a qu'à lui affecter l'attribut nécessaire (`chiffre_affaire`) et le constructeur qui va avec :

```
class Commercial: public Employe {
public:
   Commercial(string prenom, string nom, unsigned int age, string date,
        double chiffre_affaire)
      : Employe(prenom, nom, age, date), chiffre_affaire(chiffre_affaire)
   {}
   ~Commercial() {}
protected:
   double chiffre_affaire;
};
```

On peut alors écrire les deux classes qui en héritent : `Vendeur` et `Representant` :

```
class Vendeur: public Commercial {
public:
  Vendeur(string prenom, string nom, unsigned int age,
          string date, double chiffre_affaire)
     : Commercial(prenom, nom, age, date, chiffre_affaire)
  {}
  ~Vendeur() {}
```

```
  double calculer_salaire() const;
  string get_nom() const;
};

class Representant: public Commercial {
public:
   Representant(string prenom, string nom, unsigned int age,
                string date, double chiffre_affaire)
      : Commercial(prenom, nom, age, date, chiffre_affaire)
   {}
   ~Representant() {}
   double calculer_salaire() const;
   string get_nom() const;
};
```

Il n'y a plus qu'à spécifier leurs méthodes spécifiques :

```
double Vendeur::calculer_salaire() const {
  return (0.2 * chiffre_affaire) + 400;
}

string Vendeur::get_nom() const {
  return "Le vendeur " + prenom + ' ' + nom;
}

double Representant::calculer_salaire() const {
  return (0.2 * chiffre_affaire) + 800;
}

string Representant::get_nom() const {
  return "Le representant " + prenom + ' ' + nom;
}
```

Remarque. Pour les programmeurs plus avancés, il serait préférable de ne pas coder les constantes 400 et 800 «en dur», mais de définir des constantes statiques dans la classe. Pour un programme de petite taille cela ne change pas grand-chose, mais pour un projet d'envergure cela facilite la maintenance.

Les classes `Technicien` et `Manutentionnaire` se font de la même façon, sans difficulté :

```
class Technicien: public Employe {
public:
   Technicien(string prenom, string nom, unsigned int age, string date,
              unsigned int unites)
      : Employe(prenom, nom, age, date), unites(unites)
   {}
   ~Technicien() {}
   double calculer_salaire() const;
   string get_nom() const;
protected:
   unsigned int unites;
};

double Technicien::calculer_salaire() const {
   return 5.0 * unites;
}

string Technicien::get_nom() const {
  return "Le technicien " + prenom + ' ' + nom;
}

class Manutentionnaire: public Employe {
public:
  Manutentionnaire(string prenom, string nom, unsigned int age,
```

```
                  string date, unsigned int heures)
      : Employe(prenom, nom, age, date), heures(heures)
   {}
   ~Manutentionnaire() {}
   double calculer_salaire() const;
   string get_nom() const;
protected:
   unsigned int heures;
};

double Manutentionnaire::calculer_salaire() const {
  return 65.0 * heures;
}

string Manutentionnaire::get_nom() const {
  return "Le manut. " + prenom + ' ' + nom;
}
```

Employés à risques

Voici finalement l'héritage multiple. Mais avant il faut définir la super-classe d'employé à risque :

```
class ARisque {
public:
   ARisque(double prime = 100) : prime(prime) {}
   virtual ~ARisque() {}
protected:
   double prime;
};
```

Concernant l'ordre d'héritage, il semble ici évident que ces employés sont avant tout des employés (avant d'être « à risque »). On a donc naturellement :

```
class TechnARisque: public Technicien, public ARisque {
};
class ManutARisque: public Manutentionnaire, public ARisque {
};
```

qu'il suffit ensuite de compléter des éléments habituels :

```
class TechnARisque: public Technicien, public ARisque {
public:
   TechnARisque(string prenom, string nom, unsigned int age,
                string date, unsigned int unites, double prime)
      : Technicien(prenom, nom, age, date, unites), ARisque(prime)
   {}
   double calculer_salaire() const;
};

double TechnARisque::calculer_salaire() const {
   return Technicien::calculer_salaire() + prime;
}

class ManutARisque: public Manutentionnaire, public ARisque {
public:
   ManutARisque(string prenom, string nom, unsigned int age,
                string date, unsigned int heures, double prime)
      : Manutentionnaire(prenom, nom, age, date, heures), ARisque(prime)
   {}
   double calculer_salaire() const;
};

```

```
double ManutARisque::calculer_salaire() const {
  return Manutentionnaire::calculer_salaire() + prime;
}
```

Collection d'employés

Il s'agit ici de quelque chose de très similaire à ce que nous avons fait la semaine dernière. Voici donc la solution :

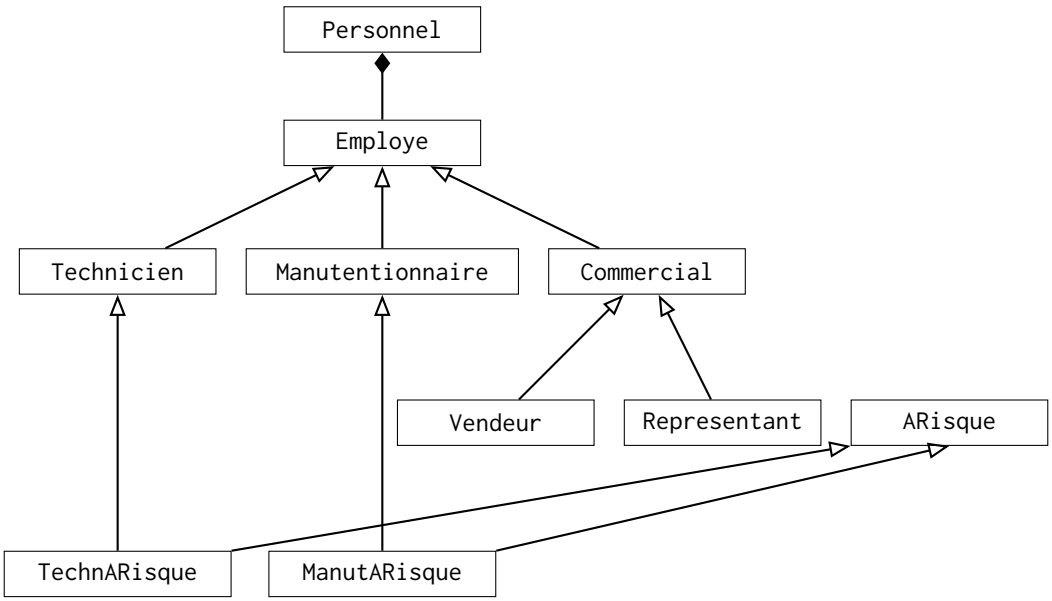
```
class Personnel {
public:
   void ajouter_employe(Employe* newbie) { staff.push_back(newbie); }
   void licencie();
   void afficher_salaires() const;
   double salaire_moyen() const;
protected:
   vector<Employe*> staff;
};

void Personnel::licencie() {
  for (auto p : staff) delete p;
  staff.clear();
}

double Personnel::salaire_moyen() const {
  double somme(0.0);
  for (auto p : staff) {
    somme += p->calculer_salaire();
  }
  return somme / staff.size();
}

void Personnel::afficher_salaires() const {
  for (auto p : staff) {
    cout << p->get_nom() << " gagne "
         << p->calculer_salaire() << " francs."
         << endl;
  }
}
```

Voici pour résumer, le diagramme d'héritage (et encapsulation) de cet exercice :



et le code source complet :

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

/* ************************************************************************
 * La classe Employe
 */
class Employe {
public:
   Employe(string prenom, string nom, unsigned int age, string date)
     : nom(nom), prenom(prenom), age(age), date(date) {}
   virtual ~Employe() {}
   virtual double calculer_salaire() const = 0;
   virtual string get_nom() const;
protected:
   string nom;
   string prenom;
   unsigned int age;
   string date;
};

string Employe::get_nom() const { return "L'employe " + prenom + ' ' + nom; }

/* ************************************************************************
 * La classe Commercial (factorise Vendeur et Representant)
 */
class Commercial: public Employe {
public:
   Commercial(string prenom, string nom, unsigned int age, string date,
        double chiffre_affaire)
      : Employe(prenom, nom, age, date), chiffre_affaire(chiffre_affaire)
   {}
   ~Commercial() {}
protected:
   double chiffre_affaire;
};

/* ************************************************************************
 * La classe Vendeur
 */
class Vendeur: public Commercial {
public:
   Vendeur(string prenom, string nom, unsigned int age, string date,
    double chiffre_affaire)
      : Commercial(prenom, nom, age, date, chiffre_affaire)
   {}
   ~Vendeur() {}
   double calculer_salaire() const;
   string get_nom() const;
};

double Vendeur::calculer_salaire() const {
  return (0.2 * chiffre_affaire) + 400;
}

string Vendeur::get_nom() const { return "Le vendeur " + prenom + ' ' + nom; }

/* ************************************************************************
 * La classe Representant
 */
```

```cpp
class Representant: public Commercial {
public:
  Representant(string prenom, string nom, unsigned int age, string date,
        double chiffre_affaire)
     : Commercial(prenom, nom, age, date, chiffre_affaire)
  {}
  ~Representant() {}
  double calculer_salaire() const;
  string get_nom() const;
};

double Representant::calculer_salaire() const {
  return (0.2 * chiffre_affaire) + 800;
}

string Representant::get_nom() const { return "Le representant " + prenom + ' ' + nom; }

/* ************************************************************************
 * La classe Technicien (Production)
 */
class Technicien: public Employe {
public:
   Technicien(string prenom, string nom, unsigned int age, string date,
       unsigned int unites)
      : Employe(prenom, nom, age, date), unites(unites)
   {}
   ~Technicien() {}
   double calculer_salaire() const;
   string get_nom() const;
protected:
   unsigned int unites;
};

double Technicien::calculer_salaire() const {
   return 5.0 * unites;
}

string Technicien::get_nom() const { return "Le technicien " + prenom + ' ' + nom; }

/* ************************************************************************
 * La classe Manutentionnaire
 */
class Manutentionnaire: public Employe {
public:
   Manutentionnaire(string prenom, string nom, unsigned int age, string date,
       unsigned int heures)
      : Employe(prenom, nom, age, date), heures(heures)
   {}
   ~Manutentionnaire() {}
   double calculer_salaire() const;
   string get_nom() const;
protected:
   unsigned int heures;
};

double Manutentionnaire::calculer_salaire() const {
  return 65.0 * heures;
}

string Manutentionnaire::get_nom() const { return "Le manut. " + prenom + ' ' + nom; }

/* ************************************************************************
 * La classe d'employes a risque
```

```
 */
class ARisque {
public:
   ARisque(double prime = 100) : prime(prime) {}
   virtual ~ARisque() {}
protected:
   double prime;
};

/* ************************************************************************
 * Une premiere sous-classe d'employe a risque
 */
class TechnARisque: public Technicien, public ARisque {
public:
   TechnARisque(string prenom, string nom, unsigned int age, string date,
         unsigned int unites, double prime)
      : Technicien(prenom, nom, age, date, unites), ARisque(prime)
   {}
   double calculer_salaire() const;
};

double TechnARisque::calculer_salaire() const {
   return Technicien::calculer_salaire() + prime;
}

/* ************************************************************************
 * Une autre sous-classe d'employe a risque
 */
class ManutARisque: public Manutentionnaire, public ARisque {
public:
   ManutARisque(string prenom, string nom, unsigned int age, string date,
         unsigned int heures, double prime)
      : Manutentionnaire(prenom, nom, age, date, heures), ARisque(prime)
   {}
   double calculer_salaire() const;
};

double ManutARisque::calculer_salaire() const {
  return Manutentionnaire::calculer_salaire() + prime;
}

/* ************************************************************************
 * La classe Personnel
 */
class Personnel {
public:
    void ajouter_employe(Employe* newbie) { staff.push_back(newbie); }
   void licencie();
   void afficher_salaires() const;
   double salaire_moyen() const;
protected:
   vector<Employe*> staff;
};

void Personnel::licencie() {
  for (unsigned int i(0); i < staff.size(); i++) {
    delete staff[i];
  }
  staff.clear();
}

double Personnel::salaire_moyen() const {
  double somme(0.0);
```

```cpp
  for (unsigned int i(0); i < staff.size(); i++) {
    somme += staff[i]->calculer_salaire();
  }
  return somme / staff.size();
}

void Personnel::afficher_salaires() const {
  for (unsigned int i(0); i < staff.size(); i++) {
    cout << staff[i]->get_nom() << " gagne "
         << staff[i]->calculer_salaire() << " francs."
         << endl;
  }
}

// ====================================================================
int main () {
  Personnel p;
  p.ajouter_employe(new Vendeur("Pierre", "Business", 45, "1995", 30000));
  p.ajouter_employe(new Representant("Leon", "Vendtout", 25, "2001", 20000));
  p.ajouter_employe(new Technicien("Yves", "Bosseur", 28, "1998", 1000));
  p.ajouter_employe(new Manutentionnaire("Jeanne", "Stocketout", 32, "1998", 45));
  p.ajouter_employe(new TechnARisque("Jean", "Flippe", 28, "2000", 1000, 200));
  p.ajouter_employe(new ManutARisque("Al", "Abordage", 30, "2001", 45, 120));

  p.afficher_salaires();
  cout << "Le salaire moyen dans l'entreprise est de "
       << p.salaire_moyen() << " francs." << endl;

  // liberation memoire
  p.licencie();
}
```

Exercice 22 : Jeu de cartes

Cet exercice correspond à l'exercice n°65 (pages 164 et 358) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

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

class Couleur {
public:
   typedef enum { ROUGE, VERT, BLEU, BLANC, NOIR } Choix;
   Couleur(Choix c) : valeur(c) {}
   virtual ~Couleur() {}
   Choix valeur;
   void affiche(ostream&, bool feminin = false) const;
};

void Couleur::affiche(ostream& out, bool feminin) const {
  switch (valeur) {
  case ROUGE: out << "rouge"; break;
  case VERT:
    out << "vert";
    if (feminin) out << 'e';
    break;
  case BLEU:
    out << "bleu";
    if (feminin) out << 'e';
    break;
  case BLANC:
    out << "blanc";
    if (feminin) out << "he";
    break;
  case NOIR:
    out << "noir";
    if (feminin) out << 'e';
    break;
  }
}

// ------------------------------------------------------------------
class Carte {
public:
  Carte(unsigned int cost = 0) : cost(cost) {}
  virtual ~Carte(){}
  virtual void afficher(ostream& out) const {
    out << "de cout " << cost; }

protected:
   unsigned int cost;
};

ostream& operator<<(ostream& out, const Carte& c) {
  c.afficher(out);
  return out;
}

// ------------------------------------------------------------------
class Terrain : public virtual Carte {
public:
```

```
  Terrain(Couleur c) : couleur(c) {
    cout << "Un nouveau terrain." << endl;
   }
   virtual ~Terrain() {}
   void afficher(ostream&) const;
protected:
   Couleur couleur;
};

void Terrain::afficher(ostream& out) const {
  out << "Un terrain ";
  couleur.affiche(out);
  out << "." << endl;
}

// ----------------------------------------------------------------------
class Creature : public virtual Carte {
public:
   Creature(unsigned int cost, string nom, unsigned int attaque,
            unsigned int defense) :
     Carte(cost), nom(nom), attaque(attaque), defense(defense) {
     cout << "Une nouvelle creature." << endl;
   }
   virtual ~Creature() {}
   void afficher(ostream&) const;
protected:
   string nom;
   unsigned int attaque;
   unsigned int defense;
};

void Creature::afficher(ostream& out) const {
  out << "Une creature " << nom << ' ' << attaque << "/"
      << defense << ' ';
  Carte::afficher(out);
  out << endl;
}

// ----------------------------------------------------------------------
class Sortilege : public virtual Carte {
public:
   Sortilege(unsigned int cost, string nom, string desc) :
     Carte(cost), nom(nom), description(desc) {
     cout << "Un sortilege de plus." << endl;
   }
   virtual ~Sortilege() {}
   void afficher(ostream&) const;
protected:
   string nom;
   string description;
};

void Sortilege::afficher(ostream& out) const {
  out << "Un sortilege " << nom << ' ';
  Carte::afficher(out);
  out << endl;
}

// ----------------------------------------------------------------------
class CreatureTerrain : public Creature, public Terrain {
public:
  CreatureTerrain(unsigned int cost, string nom, unsigned int attaque,
                  unsigned int defense, Couleur couleur)
```

```
    : Carte(cost), Creature(cost, nom, attaque, defense),
      Terrain(couleur)
  {
    cout << "Houla, une creature/terrain." << endl;
  }
  virtual ~CreatureTerrain() {}
  void afficher(ostream&) const;
};

void CreatureTerrain::afficher(ostream& out) const {
  out << "Une creature/terrain ";
  couleur.affiche(out, true);
  out << ' ' << nom << ' ' << attaque << "/" << defense << ' ';
  Carte::afficher(out);
  out << endl;
}

// ----------------------------------------------------------------------
class Jeu {
public:
   Jeu(){ cout << "On change de main" << endl; }
   virtual ~Jeu(){}
   void jette();
   void ajoute(Carte* carte) { contenu.push_back(carte); }
private:
   vector<Carte*> contenu;
friend ostream& operator<<(ostream&, const Jeu&);
};

ostream& operator<<(ostream& out, const Jeu& j) {
  for (auto carte : j.contenu)
    out << " + " << *carte;
  return out;
}

void Jeu::jette() {
  cout << "Je jette ma main." << endl;
  for (auto& carte : contenu) { delete carte; }
  contenu.clear();
}

// ----------------------------------------------------------------------
int main()
{
  Jeu mamain;

  mamain.ajoute(new Terrain(Couleur::BLEU));
  mamain.ajoute(new Creature(6, "Golem", 4, 6));
  mamain.ajoute(new Sortilege(1, "Croissance Gigantesque",
          "La creature ciblee gagne +3/+3 jusqu'a la fin du tour"));
  mamain.ajoute(new CreatureTerrain(2, "Ondine", 1, 1, Couleur::BLEU));

  cout << "La, j'ai en stock :" << endl;
  cout << mamain;

  mamain.jette();

  return 0;
}
```