Les employés [hm03] - Exercice

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

C++ - Les employés (Solution)

Mots-Clés Héritage multiple ■
Requis Classes, Classes (suite), Pointeurs, Héritage, Polymorphisme, Modèles, Classes abstraites ■
Fichiers dtpersonnel.txt ■
Difficulté • ○ ○ (1 h 30) ■

Objectif
Cet exercice conçoit une hiérarchie de classes d'employés. Il sert de révision pour les notions d'héritage, de polymorphisme et de classes abstraites.

1 Les employés / pgpersonnel

1.1 Classe Employe

Le directeur d'une entreprise de produits chimiques souhaite gérer les salaires et primes de ses employés.

Un employé est caractérisé par son nom, son prénom et sa date d'entrée en service dans l'entreprise. Définissez une classe `Employe` dotée des attributs nécessaires.

Fournissez un constructeur prenant en paramètres les attributs nécessaires.

Écrivez une méthode `getInfo` qui renvoie une chaîne de caractères obtenue en concaténant le nom, prénom et date (séparés par un espace).

Écrivez une méthode abstraite `salaire` qui renvoie le salaire mensuel de l'employé.

La classe étant une classe abstraite, écrivez le destructeur virtuel vide.

1.2 Classes concrètes

Le salaire mensuel dépend de la catégorie de l'employé. On distingue ceux affectés à :

- La *Vente*. Le salaire mensuel est 20% du chiffre d'affaires qu'ils réalisent mensuellement, plus 400€.
- La *Représentation*. Le salaire mensuel est également 20% du chiffre d'affaires qu'ils réalisent mensuellement, plus 800€.
- La *Production*. Le salaire vaut le nombre d'unités produites mensuellement multipliées par 5.
- La *Manutention*. Le salaire vaut leur nombre d'heures de travail mensuel multipliées par 65€.

Réalisez une hiérarchie de classes pour les employés en respectant les conditions suivantes :

- La super-classe de la hiérarchie doit être la classe `Employe`.
- N'hésitez pas à introduire des classes intermédiaires pour éviter au maximum les redondances d'attributs et de méthodes dans les sous-classes.

Écrivez vos classes et vos méthodes.

- Les classes doivent contenir les attributs qui leur sont spécifiques ainsi que le codage approprié des méthodes `salaire` et `getInfo` en précisant la catégorie correspondante.
- Chaque sous-classe est dotée d'un constructeur prenant en paramètre les attributs nécessaires et d'un destructeur.

1.3 Employés à risques

Certains employés des secteurs *Production* et *Manutention* sont appelés à fabriquer et manipuler des produits dangereux. Après plusieurs négociations syndicales, ces derniers parviennent à obtenir une prime de risque mensuelle.

Ajoutez une nouvelle super-classe pour les « employés à risques » permettant de leur associer un attribut « prime mensuelle ». Cette classe sera dotée d'un constructeur initialisant la prime mensuelle à 150€. Fournissez également un destructeur virtuel vide.

Complétez alors votre hiérarchie en introduisant deux nouvelles sous-classes d'employés. Ces sous-classes désigneront les employés des secteurs *Production* et *Manutention* travaillant avec des produits dangereux.

Écrivez vos classes et vos méthodes.

Validez vos classes et vos méthodes avec la solution.

Solution C++ @[Employes.hpp] @[Employes.cpp]

```cpp
#ifndef PEMPLOYES_CLASS
#define PEMPLOYES_CLASS
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

/**
  Classe abstraite des employes
*/
class Employe
{
  public:
    Employe(const string& n, const string& p, const string& d);
    virtual ~Employe();
    virtual double salaire() const = 0;
    virtual string getInfo() const;

  protected:
    string m_nom;       // nom de la personne
```

```
    string m_prenom;    // prenom de la personne
    string m_date;      // date d'entree dans la societe
};

/**
  Represente des commerciaux (vendeur et representant)
*/
class Commercial : public Employe
{
  typedef Employe super;

  public:
    Commercial(const string& n, const string& p,
               const string& d, double ca);

  protected:
    double m_ca; // chiffre d'affaire
};

/**
  Represente des vendeurs
*/
class Vendeur : public Commercial
{
  typedef Commercial super;

  public:
    Vendeur(const string& n, const string& p,
            const string& d, double ca);
    virtual double salaire() const;
    virtual string getInfo() const;
};

/**
  Represente des representants
*/
class Representant : public Commercial
{
  typedef Commercial super;

  public:
    Representant(const string& n, const string& p,
                 const string& d, double ca);
    virtual double salaire() const;
    virtual string getInfo() const;
};

/**
  Represente des techniciens
*/
class Technicien : public Employe
{
  typedef Employe super;

  public:
    Technicien(const string& n, const string& p,
               const string& d, unsigned u);
    virtual double salaire() const;
```

```
    virtual string getInfo() const;

  protected:
    unsigned m_unites; // nombre d'unites
};

/**
  Represente des manutentionnaires
*/
class Manutentionnaire : public Employe
{
  typedef Employe super;

  public:
    Manutentionnaire(const string& n, const string& p,
                     const string& d, unsigned h);
    virtual double salaire() const;
    virtual string getInfo() const;

  protected:
    unsigned m_heures; // nombre d'heures
};

/**
  Represente des employes a-risque
*/
class ARisque
{
  public:
    ARisque(double p);
    virtual ~ARisque();

  protected:
    double m_prime; // montant de la prime
};

/**
  Represente des techniciens a-risque
*/
class TechnARisque : public Technicien,
                     public ARisque
{
  typedef Technicien super;

  public:
    TechnARisque(const string& n, const string& p,
                 const string& d, unsigned u, double prime);
    virtual double salaire() const;
    virtual string getInfo() const;
};

/**
  Represente des manutentionnaires a-risque
*/
class ManutARisque : public Manutentionnaire,
                     public ARisque
{
  typedef Manutentionnaire super;
```

```
  public:
    ManutARisque(const string& n, const string& p,
                 const string& d, unsigned h, double prime);
    virtual double salaire() const;
    virtual string getInfo() const;
};
#include "Employes.cpp"
#endif
```

```
/**
  Constructeur normal
*/
Employe::Employe(const string& n, const string& p, const string& d)
: m_nom(n), m_prenom(p), m_date(d)
{}

/**
  Destructeur virtuel
*/
Employe::~Employe()
{}

/**
  Methode retournant la chaine d'infos de l'employe
*/
string Employe::getInfo() const
{
  return m_nom + " " + m_prenom + " " + m_date;
}

/**
  Constructeur normal
*/
Commercial::Commercial(const string& n, const string& p,
           const string& d, double ca)
: super(n, p, d), m_ca(ca)
{}

/**
  Constructeur normal
*/
Vendeur::Vendeur(const string& n, const string& p,
        const string& d, double ca)
: super(n, p, d, ca)
{}

/**
  Methode de calcul du salaire
*/
double Vendeur::salaire() const
{
  return (0.2 * m_ca) + 400;
}

/**
  Chaine d'infos de l'employe
*/
```

```
string Vendeur::getInfo() const
{
  return "Vendeur " + super::getInfo();
}

/**
  Constructeur normal
*/
Representant::Representant(const string& n, const string& p,
             const string& d, double ca)
: super(n, p, d, ca)
{}

/**
  Methode de calcul du salaire
*/
double Representant::salaire() const
{
  return (0.2 * m_ca) + 800;
}

/**
  Chaine d'infos de l'employe
*/
string Representant::getInfo() const
{
  return "Representant " + super::getInfo();
}

/**
  Constructeur normal
*/
Technicien::Technicien(const string& n, const string& p,
           const string& d, unsigned u)
: super(n, p, d), m_unites(u)
{}

/**
  Methode de calcul du salaire
*/
double Technicien::salaire() const
{
  return 5.0 * m_unites;
}

/**
  Chaine d'infos de l'employe
*/
string Technicien::getInfo() const
{
  return "Technicien " + super::getInfo();
}

/**
  Constructeur normal
*/
Manutentionnaire::Manutentionnaire(const string& n, const string& p,
                 const string& d, unsigned h)
```

```
: super(n, p, d), m_heures(h)
{}

/**
  Methode de calcul du salaire
*/
double Manutentionnaire::salaire() const
{
  return 65.0 * m_heures;
}

/**
  Chaine d'infos de l'employe
*/
string Manutentionnaire::getInfo() const
{
  return "Manutentionnaire " + super::getInfo();
}

/**
  Constructeur normal
*/
ARisque::ARisque(double p)
: m_prime(p)
{}

/**
  Destructeur virtuel
*/
ARisque::~ARisque()
{}

/**
  Constructeur normal
*/
TechnARisque::TechnARisque(const string& n, const string& p,
             const string& d, unsigned u, double prime)
: super(n, p, d, u), ARisque(prime)
{}

/**
  Methode de calcul du salaire
*/
double TechnARisque::salaire() const
{
  return super::salaire() + m_prime;
}

/**
  Chaine d'infos de l'employe
*/
string TechnARisque::getInfo() const
{
  return "TechnARisque " + super::getInfo();
}

/**
  Constructeur normal
```

```
*/
ManutARisque::ManutARisque(const string& n, const string& p,
             const string& d, unsigned h, double prime)
: super(n, p, d, h), ARisque(prime)
{}

/**
  Methode de calcul du salaire
*/
double ManutARisque::salaire() const
{
  return super::salaire() + m_prime;
}

/**
  Chaine d'infos de l'employe
*/
string ManutARisque::getInfo() const
{
  return "ManutARisque " + super::getInfo();
}
```

1.4 Collection d'Employés

Satisfait de la hiérarchie proposée, notre directeur souhaite maintenant l'exploiter pour calculer le salaire de tous ses employés ainsi que le salaire moyen.

Définissez une classe `Personnel` contenant une collection « hétérogène » (polymorphique) d'`Employe`. Fournissez :

- Le constructeur par défaut.
- Le destructeur afin de libérer proprement les instances dynamiques.
- Un accesseur `size` de la taille de la collection.

Écrivez une méthode `ajouter(e)` qui ajoute un (pointeur sur un) `Employe e` à la collection.

Écrivez une méthode `charger(fn)` qui charge une collection depuis un nom de fichier `fn` (chaine de caractères).

Écrivez une méthode `calculer` qui calcule et affiche le salaire de chacun des employés ainsi que le salaire moyen des employés de la collection.

Validez votre classe et vos méthodes avec la solution.

Solution C++ @[Personnel.hpp] @[Personnel.cpp]

```
#ifndef PERSONNEL_CLASS
#define PERSONNEL_CLASS
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

/**
  Collection d'employes
*/
#include "Employes.hpp"
class Personnel
{
  public:
    Personnel();
    ~Personnel();
    unsigned size() const;
    bool charger(const string& fn);
    void ajouter(Employe* e);
    void calculer() const;
    void vider();

  protected:
    template<typename T>
    bool ajouterCommercial(istream& is);

    template<typename T>
    bool ajouterEmploye(istream& is);

    template<typename T>
    bool ajouterArisque(istream& is);

  private:
    vector<Employe*> m_staff;

    Personnel(const Personnel&); // interdit la recopie
    Personnel& operator=(const Personnel&); // interdit l'affectation
};
#include "Personnel.cpp"
#endif
```

```
/**
  Constructeur par defaut
*/
Personnel::Personnel()
: m_staff()
{}

/**
  Destructeur: il faut le surdefinir afin de liberer
  proprement les instances dynamiques
*/
Personnel::~Personnel()
{
  vider();
}
```

```
/**
  Taille de la collection
*/
unsigned Personnel::size() const
{
  return m_staff.size();
}

/**
  Charge les donnees depuis un fichier
  @return Vrai si l'operation a ete effectuee, Faux sinon
*/
bool Personnel::charger(const string& fn)
{
  ifstream is(fn.c_str());
  if (!is)
  {
    return false;
  }
  // Effectue la lecture du fichier, tantque c'est possible
  char c;
  while (is >> c)
  {
    // Decode les donnees
    switch (c)
    {
      case 'V': case 'v':
        ajouterCommercial<Vendeur>(is);
        break;

      case 'R': case 'r':
        ajouterCommercial<Representant>(is);
        break;

      case 'T': case 't':
        ajouterEmploye<Technicien>(is);
        break;

      case 'M': case 'm':
        ajouterEmploye<Manutentionnaire>(is);
        break;

      case 'U': case 'u':
        ajouterArisque<TechnARisque>(is);
        break;

      case 'N': case 'n':
        ajouterArisque<ManutARisque>(is);
        break;

      default:
        cerr << "\aOUPS... erreur de donnees!" << endl;
    }

    // Ignore le reste de la ligne
    is.ignore(100, '\n');
  }
```

```
  // On pose que c'est OK
  return true;
}

/**
  Ajoute un nouvel employe a la collection
*/
void Personnel::ajouter(Employe* e)
{
  m_staff.push_back(e);
}

/**
  Calcule et affiche l'etat du personnel
*/
void Personnel::calculer() const
{
  // Verifie que le calcul est possible
  if (m_staff.empty())
  {
    return;
  }

  // Somme des salaires
  double sigma = 0.0;

  // Affiche les employes
  const unsigned n = size();
  for (unsigned ix = 0; ix < n; ++ix)
  {
    cout<<m_staff[ix]->getInfo();

    double s = m_staff[ix]->salaire();
    cout<<", salaire = "<<s<<endl;
    sigma += s;
  }

  // Calcule et affiche le salaire moyen
  cout<<"Total des salaires = "<<sigma<<endl;
  cout<<"# d'employes       = "<<n<<endl;
  cout<<"Salaire moyen      = "<<sigma/n<<endl;
}

/**
  Vide la collection des employes
*/
void Personnel::vider()
{
  // Libère chacune des instances dynamiques:
  for (unsigned k = 0; k < size(); ++k)
  {
    delete m_staff[k];
    m_staff[k] = 0;
  }

  // Purge le conteneur
  m_staff.clear();
```

```
}

/**
  Lit les caracteriques d'un commercial
  @param[in,out] is - un flux d'entree
  @return le status du flux
*/
template<typename T>
bool Personnel::ajouterCommercial(istream& is)
{
  Employe *xx;
  string n, p, d; double ca;
  if (is>>n>>p>>d>>ca)
  { ajouter( xx = new T(n, p, d, ca) ); }

  return is.good();
}

/**
  Lit les caracteriques d'un Technicien ou Manutentionnaire
  @param[in,out] is - un flux d'entree
  @return le status du flux
*/
template<typename T>
bool Personnel::ajouterEmploye(istream& is)
{
  Employe *xx;
  string n, p, d; unsigned u;
  if (is>>n>>p>>d>>u)
  {
    ajouter( xx = new T(n, p, d, u) );
  }

  return is.good();
}

/**
  Lit les caracteriques d'un employe a-risque
  @param[in,out] is - un flux d'entree
  @return le status du flux
*/
template<typename T>
bool Personnel::ajouterArisque(istream& is)
{
  Employe *xx;
  string n, p, d; unsigned u; double prime;
  if (is>>n>>p>>d>>u>>prime)
  {
    ajouter( xx = new T(n, p, d, u, prime) );
  }

  return is.good();
}
```

1.5 Programme de test

Écrivez un programme qui instancie une gestion d'employés, demande le nom du fichier des données et charge les données puis calcule et affiche l'état des salaires.

Téléchargez le fichier des données :
@[dtpersonnel.txt]

```
V Business Pierre 1960 30000 #vendeur
R Vendtout Leon 1980 20000   #representant
T Bosseur Yves 1975 1000     #technicien
M Stocketout Jeanne 1970 45  #manutentionaire
U Flippe Jean 1975 1000 200  #techARisque
N Abordage Alain 1970 45 120 #manutARisque
```

Testez. Résultat d'exécution :

```
Nom du fichier des donnees? dtpersonnel1.txt
Vendeur Business Pierre 1960, salaire = 6400
Representant Vendtout Leon 1980, salaire = 4800
Technicien Bosseur Yves 1975, salaire = 5000
Manutentionnaire Stocketout Jeanne 1970, salaire = 2925
TechnARisque Technicien Flippe Jean 1975, salaire = 5200
ManutARisque Manutentionnaire Abordage Alain 1970, salaire = 3045
Total des salaires = 27370
# d'employes       = 6
Salaire moyen      = 4561.67
```

Validez votre programme avec la solution.

Solution C++ @[pgpersonnel.cpp]

```cpp
#include <iostream>
#include <fstream>
using namespace std;
#include "Employes.hpp"
#include "Personnel.hpp"

int main()
{
  // Instancie une gestion d'employes
  Personnel p;

  // Charge les donnees
  string fn;
  cout<<"Nom du fichier des donnees? ";
  cin>>fn;
  if (not p.charger(fn))
  {
    cerr<<"\aOUPS... chargement fichier"<<endl;
```

```
  }
  else
  {
    // Calcule et affiche l'etat des salaires
    p.calculer();

    // Libere les instances dynamiques
    p.vider();
  }
}
```

2 Références générales

Comprend [Chappelier-CPP1 :c13 :ex64] ■