enseignant : Philippe REITZ
année universitaire : 2001-2002

Programmation

unité de Travaux Pratiques

TP n°4 : C++

[Corrigé]

Exercice 1 : les animaux **1**

Exercice 2 : les figures **3**

Exercice 3 : un simulateur de processeur **12**

Exercice 1 : les animaux

Concevoir le code permettant de représenter l'univers de concepts suivant :

> Un *animal* porte un *nom* et *s'exprime*. Un *mamifère* est un animal. Un *chat* est un mamifère, et s'exprime par un *miaou*. *Sylvestre* et *Tom* sont des chats. Une *souris* est un animal qui s'exprime en faisant *couic*. *Jerry* est une souris. Un *oiseau* est un animal. Un *canari* est un oiseau qui s'exprime par un *cui-cui*. *Titi* est un canari.

Le programme principal se contente de demander à tous les animaux nommés ci-dessus de s'exprimer, i.e. un message s'affiche à l'écran ; ce message doit préciser le nom de l'animal.

Correction

Commençons par une première correction soulignant les difficultés de l'exercice.
Tout animal s'exprime par un cri, ce dernier lui étant spécifique :

```
class unAnimal
{
public:
  virtual void s_exprimer() const = 0;
};
```

Notons que la méthode `s_exprimer` est virtuelle : en effet, si une variable `v` est un pointeur ou une référence sur un objet de classe `unAnimal`, le cri affiché n'est pas celui défini dans cette classe (pour peu qu'il existe), mais est spécifique de l'animal correspondant exactement à l'objet.

Pour exprimer le fait que nous ne sommes pas capables de définir le cri par défaut d'un animal, nous indiquons que la méthode `s_exprimer` est pure → la classe `unAnimal` devient du coup une classe *abstraite*, qui ne pourra donc plus être instanciée, i.e. nous ne pourrons pas en user pour construire des objets.

Les autres classes sont alors triviales à définir :

```
class unMamifere
: public unAnimal
{};

class unChat
: public unMamifere
{
public:
  void s_exprimer() const { cout << "Miaou" << endl; };
};

class uneSouris
: public unMamifere
{
public:
  void s_exprimer() const { cout << "Couic" << endl; };
};

class unOiseau
: public unAnimal
{};

class unCanari
: public unOiseau
{
public:
  void s_exprimer() const { cout << "Cuicui" << endl; };
};
```

Afin de tester ces classes avec les animaux décrits dans l'énoncé, nous pourrions écrire :

```
int main() {
  unChat Sylvestre,
         Tom;
  uneSouris Jerry;
  unCanari Titi;

  unAnimal* mesAnimaux[4] = { &Sylvestre, &Titi, &Tom, &Jerry };

  for (int i = 0; i < 4; i++) mesAnimaux[i]->s_exprimer();
};
```

qui affiche, à l'écution :

```
Miaou
Cuicui
Miaou
Couic
```

Cette première version possède au moins un défaut : lorsqu'un animal s'exprime, nous ne savons pas duquel il s'agit précisemment. Modifions notre classe des animaux afin que tout animal porte un nom ; lorsqu'il s'exprime, l'affichage du cri de l'animal est alors précédé de son nom. Les classes deviennent alors :

```
class unAnimal
{
protected:
  char* sonNom;
public:
  unAnimal(char* n = "un animal") : sonNom(n) {};
protected:
  virtual char* son_cri() const = 0;
public:
  virtual void s_exprimer() const {
    cout << sonNom << " : " << son_cri() << endl; };
};

class unMamifere
: public unAnimal
{
public:
  unMamifere(char* n = "un mamifere") : unAnimal(n) {};
};

class unChat
: public unMamifere
{
public:
  unChat(char* n = "un chat") : unMamifere(n) {};
protected:
  char* son_cri() const { return "Miaou"; };
};

class uneSouris
: public unMamifere
{
public:
  uneSouris(char* n = "une souris") : unMamifere(n) {};
protected:
  char* son_cri() const { return "Couic"; };
};
```

```
class unOiseau
: public unAnimal
{
public:
  unOiseau(char* n = "un oiseau") : unAnimal(n) {};
};

class unCanari
: public unOiseau
{
public:
  unCanari(char* n = "un canari") : unOiseau(n) {};
public:
  char* son_cri() const { return "Cuicui"; };
};
```

La méthode publique `s_exprimer` exploite une méthode protégée (définie pour l'occasion) `son_cri` qui retourne la chaine de caractère indiquant le cri de l'animal. Cette dernière méthode est donc nécessairement virtuelle, puisque caractéristique de chaque animal.

L'exécution de :

```
int main() {
  unChat Sylvestre("Sylvestre"),
         Tom      ("Tom");
  uneSouris Jerry ("Jerry");
  unCanari Titi   ("Titi");

  unAnimal* mesAnimaux[4] = { &Sylvestre, &Titi, &Tom, &Jerry };

  for (int i = 0; i < 4; i++) mesAnimaux[i]->s_exprimer();
};
```

provoque l'affichage de :

```
Sylvestre : Miaou
Titi : Cuicui
Tom : Miaou
Jerry : Couic
```

Exercice 2 : les figures

1. Écrire une classe `Figure` permettant de représenter des figures du plan. Toute figure est caractérisée par au moins un point de référence, et peut être déplacée (indication d'un déplacement pour chaque coordonnée). Étant donnée une figure, il est possible d'en connaitre la taille ou de lui demander de se décrire (messages sur la console).
   Écrire alors une classe `FigurePlane`, spécialisation de `Figure`, pour qui la taille correspond à sa surface.
2. Écrire une classe `Cercle` permettant de représenter un cercle. Un cercle est une figure plane qui se caractérise par son rayon. Tous les services offerts par un cercle sont ceux du point 1 (taille, surface et description).
   Faire de même avec les rectangles (hauteur et largeur), les triangles (hauteur et base) et les carrés (coté).
3. Définir la classe qui permet de représenter un segment de droite (sa taille est sa longueur).
4. Définir la classe des rectangles évidés par des cercles (nous supposerons que le cercle intérieur est toujours inclus dans le rectangle qu'il évide).
   Généraliser la classe afin que toute figure plane puisse être évidée par une autre figure plane.

Correction

Une figure est simplement définie par :

```
class uneFigure
```

```
{
protected:
  float x, y;
public:
  uneFigure(float=0.0, float=0.0);
protected:
  virtual char* intitule_genre() const {
    return "une figure"; };
  virtual char* intitule_taille() const {
    return "Ma taille"; };
  virtual char* intitule_reference() const {
    return "Mon point de reference"; };
public:
  virtual void se_decrire() const;
  virtual void deplacer(float=0.0, float=0.0);
  virtual float taille() const = 0;
};
```

Ainsi toute figure doit savoir répondre au message lui demandant de calculer sa taille, ainsi que de se décrire. Pour la méthode décrivant l'objet, nous faisons appel à trois fonctions locales :

- `intitule_genre` retourne le genre, en français, de la figure : *un carré*, *un triangle*, etc. Par défaut, cette méthode retourne le genre *une figure*.
- `intitule_taille` retourne la nature, en français, de la taille (c'est une *surface* pour une figure plane, une *longueur* pour une figure linéïque, etc). Par défaut, cette méthode retourne l'intitulé *ma taille*.
- `intitule_reference` retourne l'intitulé, en français, du point de référence. Par défaut, cette méthode retourne l'intitulé *Mon point de référence*.

L'implantation de cette méthode consiste en :

```
uneFigure::uneFigure(float X, float Y)
: x(X), y(Y)
{};

void uneFigure::se_decrire() const {
  cout << "Je suis " << intitule_genre() << endl;
  cout << intitule_taille() << " vaut " << taille() << endl;
  cout << intitule_reference() << " est en (" << x << ", " << y << ")" << endl;
};

void uneFigure::deplacer(float dx, float dy) {
  x += dx; y += dy; };
```

Notons que toute figure affiche au moins trois lignes de texte en se décrivant : la première affiche le genre de la figure, la seconde sa taille, la troisième le point de référence. Le texte de chaque ligne est adapté par chaque type de figure ; ainsi, nous avons par défaut :

```
Je suis une figure
Ma taille vaut ...
Mon point de référence est en (..., ...)
```

Si un carré se décrit, il affichera simplement, en tant que figure :

```
Je suis un carré
Ma surface vaut ...
Mon coin inférieur gauche est en (..., ...)
```

alors qu'un segment de droite affichera :

```
Je suis un segment
Ma longueur vaut ...
Ma première extrémité est en (..., ...)
```

Puisque le code des fonctions `intitule_...` est spécifique de l'objet qui se décrit, ces fonctions sont impérativement virtuelles. Le même argument conduit à ce que les fonctions `taille` et `se_decrire` le soient aussi.

Passons aux figures planes :

```
class uneFigurePlane
: public uneFigure
{
public:
  uneFigurePlane(float=0.0, float=0.0);
public:
  virtual float surface() const = 0;
  float taille() const;
protected:
  char* intitule_genre() const;
  char* intitule_taille() const;
};
```

Cette classe redéfinit la méthode `taille`, puisque sa définition est parfaitement connue.

L'implantation de ces méthodes nous conduit à:

```
uneFigurePlane::uneFigurePlane(float X, float Y)
: uneFigure(X, Y)
{};

float uneFigurePlane::taille() const { return surface(); };

char* uneFigurePlane::intitule_genre() const { return "une figure plane"; };

char* uneFigurePlane::intitule_taille() const { return "Ma surface"; };
```

Un cercle est un cas particulier de figure plane, d'où :

```
class unCercle
: public uneFigurePlane
{
protected:
  float rayon;
public:
  unCercle(float=0.0, float=0.0, float=1.0);
public:
  float surface() const;
  void se_decrire() const;
protected:
  char* intitule_genre() const;
  char* intitule_reference() const;
};
```

Le code associé ne pose pas de difficulté particulière :

```
unCercle::unCercle(float X, float Y, float R)
: uneFigurePlane(X, Y), rayon(R)
{};

float unCercle::surface() const { return 3.1415927*rayon*rayon; };

char* unCercle::intitule_genre() const { return "un cercle"; };

char* unCercle::intitule_reference() const { return "Mon centre"; };

void unCercle::se_decrire() const {
  uneFigurePlane::se_decrire();
  cout << "Mon rayon vaut " << rayon << endl;
};
```

Les définitions des autres figures sont du même style :

– pour les rectangles :

```
class unRectangle
: public uneFigurePlane
{
protected:
  float longueur, hauteur;
public:
  unRectangle(float=0.0, float=0.0, float=1.0, float=1.0);
public:
  float surface() const;
  void se_decrire() const;
protected:
  char* intitule_genre() const;
  char* intitule_reference() const;
  virtual void decrire_longueur_hauteur() const;
};
```

dont l'implantation est :

```
unRectangle::unRectangle(float X, float Y, float L, float H)
: uneFigurePlane(X, Y), longueur(L), hauteur(H)
{};

float unRectangle::surface() const { return longueur*hauteur; };

char* unRectangle::intitule_genre() const { return "un rectangle"; };

char* unRectangle::intitule_reference() const {
  return "Mon coin inferieur gauche"; };

void unRectangle::se_decrire() const {
  uneFigurePlane::se_decrire();
  decrire_longueur_hauteur();
};

void unRectangle::decrire_longueur_hauteur() const {
  cout << "Ma longueur vaut " << longueur
       << ", ma hauteur " << hauteur
       << endl;
};
```

– pour les carrés, considérés comme des cas particuliers de rectangles :

```
class unCarre
: public unRectangle
{
public:
  unCarre(float=0.0, float=0.0, float=1.0);
protected:
  char* intitule_genre() const;
  void decrire_longueur_hauteur() const;
};
```

dont l'implantation est :

```
unCarre::unCarre(float X, float Y, float C)
: unRectangle(X, Y, C, C)
{};
```

```
char* unCarre::intitule_genre() const { return "un carre"; };

void unCarre::decrire_longueur_hauteur() const {
  cout << "Mon cote vaut " << longueur
       << endl;
};
```

Notons que notre choix d'implantation consistant à exploiter le lien d'héritage pour représenter la relation *est un cas particulier de* entre `unCarre` et `unRectangle` peut être critiqué ; en effet, s'il semble justifié conceptuellement (un carré est bien un cas particulier de rectangle), il est plus criticable quand à la consommation d'espace mémoire exigée pour stocker les informations d'un carré : deux membres (`longueur` et `hauteur`) sont exploités, alors qu'un seul aurait suffit.
Cette remarque montre clairement que le lien d'héritage offert par `C++` est bien exploité au niveau de la réalisation, pas de la conceptualisation du problème → établir un lien d'héritage entre deux classes relève d'un choix d'implantation.

– pour les triangles :

```
class unTriangle
: public uneFigurePlane
{
protected:
  float base, hauteur;
public:
  unTriangle(float=0.0, float=0.0, float=1.0, float=1.0);
public:
  float surface() const;
  void se_decrire() const;
protected:
  char* intitule_genre() const;
  char* intitule_reference() const;
};
```

dont l'implantation est :

```
unTriangle::unTriangle(float X, float Y, float B, float H)
: uneFigurePlane(X, Y), base(B), hauteur(H)
{};

float unTriangle::surface() const { return base*hauteur/2; };

char* unTriangle::intitule_genre() const { return "un triangle"; };

char* unTriangle::intitule_reference() const {
  return "Mon coin inferieur gauche"; };

void unTriangle::se_decrire() const {
  uneFigurePlane::se_decrire();
  cout << "Ma base vaut " << base
       << ", ma hauteur " << hauteur
       << endl;
};
```

Un segment de droite n'est pas une figure plane, puisque sa taille n'est pas sa surface mais sa longueur. Nous définissons donc une classe des figures linéïques, dont la taille est donnée par leur longueur :

```
class uneFigureLineique
: public uneFigure
{
public:
  uneFigureLineique(float=0.0, float=0.0);
```

```
public:
  virtual float longueur() const = 0;
  float taille() const;
protected:
  char* intitule_genre() const;
  char* intitule_taille() const;
};
```

dont l'implantation est :

```
uneFigureLineique::uneFigureLineique(float X, float Y)
: uneFigure(X, Y)
{};

float uneFigureLineique::taille() const { return longueur(); };

char* uneFigureLineique::intitule_taille() const { return "Ma longueur"; };

char* uneFigureLineique::intitule_genre() const { return "une figure lineique"; };
```

Dès lors, un segment se définit simplement par :

```
class unSegment
: public uneFigureLineique
{
protected:
  float xB, yB;
public:
  unSegment(float=0.0, float=0.0, float=1.0, float=1.0);
public:
  float longueur() const;
  void se_decrire() const;
protected:
  char* intitule_genre() const;
  char* intitule_reference() const;
};
```

avec :

```
unSegment::unSegment(float XA, float YA, float XB, float YB)
: uneFigureLineique(XA, YA), xB(XB), yB(YB)
{};

float unSegment::longueur() const {
  return sqrt((x-xB)*(x-xB)+(y-yB)*(y-yB)); };

char* unSegment::intitule_genre() const { return "un segment"; };

char* unSegment::intitule_reference() const {
  return "Ma premiere extremite"; };

void unSegment::se_decrire() const {
  uneFigureLineique::se_decrire();
  cout << "Ma deuxieme extremite est en ("
       << xB << ", " << yB << ")" << endl;
};
```

Un rectangle évidé par un cercle est un cas particulier de rectangle, d'où notre choix :

```
class unRecEPCer
: public unRectangle
{
```

```
private:
  bool      detruire;
protected:
  unCercle& cercle;
public:
  unRecEPCer(float=0.0, float=0.0, float=1.0, float=1.0, float=0.5);
  unRecEPCer(unCercle&, float=0.0, float=0.0, float=1.0, float=1.0);
  ~unRecEPCer();
public:
  float surface() const;
  void se_decrire() const;
protected:
  char* intitule_genre() const;
};
```

Nous nous donnons deux moyens pour spécifier le cercle intérieur :

- soit nous en précisons le rayon
- soit nous passons au constructeur le cercle qui évide

En principe, nous devrions contrôler que le cercle qui évide est bien complètement inclu dans le rectangle ; nous ne le faisons pas, afin de ne pas alourdir le code.

Certains pourraient se demander pourquoi la classe `unRecEPCer` n'hérite pas à la fois de `unRectangle` et `unCercle`.

Dans un premier temps, il semble que cette solution convienne : l'héritage nous permet de récupérer tout le code associé aux cercles, nous évitant, en apparence, de pénibles récritures. Ce choix n'est pourtant pas à retenir, bien que réalisable (en effet, rien n'interdit de l'adopter en `C++`), pour les raisons suivantes :

> depuis le début de cet exercice, nous avons exploité l'héritage pour implanter une relation du type *est un cas particulier de*. Si cette relation existe naturellement entre un rectangle évidé par un cercle et un rectangle, elle n'existe pas entre un rectangle évidé par un cercle et un cercle (un tel rectangle **n'est pas** un cas particulier de cercle). Par contre, un tel rectangle *contient* un cercle → il s'agit cette fois d'une relation de *composition*.
> Choisir un héritage multiple reviendrait donc à exploiter le lien d'héritage pour implanter deux relations de nature distincte : la relation de spécialisation et la relation de composition.
> La communauté de la programmation objet a appris que ce genre d'amalgamme conduit à concevoir des bibliothèques de classes qui, rapidement, ne pourront plus être enrichies : en dérivant une classe existante, le concepteur hérite de méthodes dont il ne souhaite surtout pas hériter.

L'implantation des différentes fonctions membres est la suivante :

```
unRecEPCer::unRecEPCer(float X, float Y, float L, float H, float R)
: unRectangle(X, Y, L, H),
  cercle(*(new unCercle(X+L/2, Y+H/2, R))),
  detruire(true)
{};

unRecEPCer::unRecEPCer(unCercle& C, float X, float Y, float L, float H)
: unRectangle(X, Y, L, H),
  cercle(C),
  detruire(false)
{};

unRecEPCer::~unRecEPCer() { if (detruire) delete &cercle; };

float unRecEPCer::surface() const {
  return unRectangle::surface() - cercle.surface(); };

char* unRecEPCer::intitule_genre() const {
  return "un rectangle evide par un cercle"; };

void unRecEPCer::se_decrire() const {
```

```
  unRectangle::se_decrire();
  cout << "Le cercle qui m'evide se decrit maintenant :" << endl;
  cout << "<<<<" << endl;
  cercle.se_decrire();
  cout << ">>>>" << endl;
};
```

Notons que les 2 constructeurs se différencient par le fait que le cercle intérieur est soit créé dynamiquement, auquel cas il devra être explicitement détruit lorsque le rectangle évidé lui-même sera détruit, soit passé en paramètre, auquel cas ce n'est pas au rectangle évidé de le détruire.

Notre choix d'implantation de la classe des rectangles évidés par un cercle n'est, à l'évidence, aucunement généralisable à une figure plane évidée par une autre figure plane ; en effet, il faudrait créer un nombre infini de classes :

1. création des classes X évidé par Y, où X et Y sont au choix l'une des classes `unCercle`, `unRectangle`, `unCarre` ou `unTriangle`.
2. un triangle peut être évidé par (un rectangle évidé par un carré), par exemple ; cette sorte de triangle créée peut maintenant servir pour évider d'autres figures planes.
3. etc

Un chemin conduisant à une solution consiste à remarquer que toute figure plane évidée par une autre figure plane fait intervenir deux figures planes :

- la figure plane qui est évidée, que nous appellerons la *figure extérieure*
- la figure plane qui évide la première, appelée la *figure intérieure*

Une telle figure plane est donc composée de deux figures planes → la relation qui lie ces deux figures à la figure évidée est donc du genre *composition*.

Cette solution a l'avantage de permettre d'évider n'importe quelle figure plane par n'importe quelle autre figure plane, et ceci à partir d'une seule classe.

Nous sommes amenés à adopter la représentation suivante :

```
class uneFigurePlaneEvidee
: public uneFigurePlane
{
private:
  bool detruire;
protected:
  uneFigurePlane &exterieure, &interieure;
public:
  uneFigurePlaneEvidee(uneFigurePlane&, uneFigurePlane&, bool=false);
  ~uneFigurePlaneEvidee();
public:
  float surface() const;
  void se_decrire() const;
protected:
  char* intitule_genre() const;
};
```

avec l'implantation suivante :

```
uneFigurePlaneEvidee::uneFigurePlaneEvidee(uneFigurePlane& E, uneFigurePlane& I, bool D)
: exterieure(E), interieure(I), detruire(D)
{};

uneFigurePlaneEvidee::~uneFigurePlaneEvidee() {
  if (detruire) {
    delete &exterieure;
    delete &interieure; }
};

float uneFigurePlaneEvidee::surface() const {
```

```
  return exterieure.surface() - interieure.surface(); };

char* uneFigurePlaneEvidee::intitule_genre() const {
  return "une figure plane evidee par une autre figure plane"; };

void uneFigurePlaneEvidee::se_decrire() const {
  uneFigurePlane::se_decrire();
  cout << "Ma figure exterieure (qui est evidee) se decrit maintenant :" << endl;
  cout << "<<<<" << endl;
  exterieure.se_decrire();
  cout << ">>>>" << endl;
  cout << "Ma figure interieure (qui m'evide) se decrit maintenant :" << endl;
  cout << "<<<<" << endl;
  interieure.se_decrire();
  cout << ">>>>" << endl;
};
```

L'essentiel des classes de cet exercice est testé dans le programme suivant :

```
int main() {
  unCercle             o1(1,2,3);
  unRectangle          o2(1,2,3,4);
  unTriangle           o3(1,2,3,4);
  unCarre              o4(1,2,3);
  unSegment            o5(1,2,3,4);
  unRecEPCer           o6(0,0,2,2,1);
  uneFigurePlaneEvidee o7(o2, o4);

  const int n = 7;
   uneFigure* figures[n] = { &o1, &o2, &o3, &o4, &o5, &o6, &o7 };

  for (int i=0; i<n; i++) {
    cout << "-----" << endl;
    figures[i]->se_decrire(); }
};
```

À l'exécution, nous obtenons :

```
-----
Je suis un cercle
Ma surface vaut 28.2743
Mon centre est en (1, 2)
Mon rayon vaut 3
-----
Je suis un rectangle
Ma surface vaut 12
Mon coin inferieur gauche est en (1, 2)
Ma longueur vaut 3, ma hauteur 4
-----
Je suis un triangle
Ma surface vaut 6
Mon coin inferieur gauche est en (1, 2)
Ma base vaut 3, ma hauteur 4
-----
Je suis un carre
Ma surface vaut 9
Mon coin inferieur gauche est en (1, 2)
Mon cote vaut 3
-----
Je suis un segment
Ma longueur vaut 2.82843
```

```
Ma premiere extremite est en (1, 2)
Ma deuxieme extremite est en (3, 4)
-----
Je suis un rectangle evide par un cercle
Ma surface vaut 0.858407
Mon coin inferieur gauche est en (0, 0)
Ma longueur vaut 2, ma hauteur 2
Le cercle qui m'evide se decrit maintenant :
<<<<
Je suis un cercle
Ma surface vaut 3.14159
Mon centre est en (1, 1)
Mon rayon vaut 1
>>>>
-----
Je suis une figure plane evidee par une autre figure plane
Ma surface vaut 3
Mon point de reference est en (0, 0)
Ma figure exterieure (qui est evidee) se decrit maintenant :
<<<<
Je suis un rectangle
Ma surface vaut 12
Mon coin inferieur gauche est en (1, 2)
Ma longueur vaut 3, ma hauteur 4
>>>>
Ma figure interieure (qui m'evide) se decrit maintenant :
<<<<
Je suis un carre
Ma surface vaut 9
Mon coin inferieur gauche est en (1, 2)
Mon cote vaut 3
>>>>
```

Exercice 3 : un simulateur de processeur

1. Écrire les classes permettant de réaliser un petit simulateur d'un processeur. Les concepts essentiels du problème sont les suivants :

   - la *mémoire* est un tableau de *cases mémoire* indexé par des *adresses* (a priori représentées par des entiers naturels). Chaque case mémoire possède un *contenu* qui peut être au choix (exclusif) :
     - une *donnée* (en pratique, un entier relatif)
     - une *instruction* (voir ci-après)
   - un *processeur* possède 2 *registres* (notés A et B) qui sont 2 cases mémoire particulières ne pouvant contenir que des données. Il possède aussi un *compteur ordinal* (noté PC) qui lui permet de connaître l'adresse de la prochaine instruction à exécuter.
   - une *instruction* est de la forme :

     Instruction de transfert ou calcul

`LDI reg, don`	écrit la donnée *don* dans le registre *reg*
`LDD reg, adr`	écrit la donnée de la case d'adresse *adr* dans le registre *reg*
`ST reg, adr`	écrit le contenu du registre *reg* dans la case d'adresse *adr*
`ADD reg, adr`	ajoute au registre *reg* le contenu de la case d'adresse *adr*

     Instruction de contrôle

`NEQ reg, adr`	si le registre *reg* est non nul, poursuivre le programme à l'adresse *adr*
`JMP adr`	poursuivre le programme à l'adresse *adr*
`HALT`	arréter l'exécution du programme

   Ces classes doivent permettre au moins :

   - d'écrire une donnée ou une instruction dans une case mémoire de la mémoire dont l'adresse est spécifiée.

- d'exécuter un programme en commançant à partir d'une case mémoire d'adresse spécifiée.
  Exécuter un programme à partir d'une adresse a, c'est, pour le processeur, exécuter l'instruction correspondante codée en a. Une fois cette instruction exécutée, l'exécution se poursuit à l'adresse spécifiée par son compteur ordinal, sauf s'il s'agit de l'instruction `HALT`, auquel cas l'exécution s'arrête.
  Toute instruction qui ne modifie pas explicitement le compteur ordinal se contente de lui ajouter 1 après exécution (cas des instructions de transfert ou de calcul). Une instruction de contrôle modifie ce compteur comme indiqué dans le tableau des instructions.

2. tester ces classes en simulant l'exécution du programme suivant (débute à l'adresse 1) :

Adresse	Contenu
	des instructions
1	`LDD A, 8`
2	`LDI B, 0`
3	`ADD A, 10`
4	`ADD B, 9`
5	`NEQ A, 3`
6	`ST B, 8`
7	`HALT`
	des données
8	10
9	2
10	-1

Correction

Commençons par définir le type le plus simple, à savoir celui des adresses :

```
class uneAdresse
{
protected:
  int valeur;
public:
  uneAdresse(int);
public:
  inline operator int() const;
};
```

avec l'implantation des méthodes suivante :

```
uneAdresse::uneAdresse(int a)
: valeur(a)
{ if (a<0); };

uneAdresse::operator int() const {
  return valeur; };
```

Notons qu'il peut être objecté à notre définition d'être trop compliquée, puisqu'apparemment il s'agit simplement de définir ce type comme équivalent au type **int** ; autrement dit, pourquoi ne pas s'être contenté de :

```
typedef int uneAdresse;
```

Deux raisons essentielles contrent cette objection :

- notre type est un type à part entière, pour lequel nous pouvons définir des fonctionnalités propres.
- ce type étant effectivement autonome, nous pouvons donc surcharger sans souçi d'ambiguïté une fonction avec soit **int**, soit `uneAdresse` ; cela n'était pas possible avec un simple **typedef**.

Notons que nous avons défini le strict nécessaire afin que les conversions avec le type **int** se fassent sans problème, de la propre initiative du compilateur :

- l'opérateur `(int)` permet la conversion `uneAdresse` vers **int**.

– le constructeur `uneAdresse(int)` permet la conversion inverse.

Une case mémoire a pour contenu soit une donnée, soit une instruction.
Commençons par définir ce qu'est un contenu :

```
class unContenu
{
public:
  virtual bool estExecutable() const;
};
```

avec :

```
bool unContenu::estExecutable() const { return false; };
```

La méthode `estExecutable` permet de tester si le contenu est exécutable ou non. Cette propriété est importante pour la méthode d'exécution, définie plus loin.
Dès lors, une donnée est simplement définie à partir de :

```
class uneDonnee
: public unContenu
{
public:
  int valeur;
public:
  uneDonnee(int);
public:
  inline operator int() const;
};
```

avec :

```
uneDonnee::uneDonnee(int a)
: valeur(a)
{};

uneDonnee::operator int() const {
  return valeur; };
```

Notons que, comme le type `uneAdresse`, le strict minimum permet des conversions faciles entre les types **int** et `uneDonnee`, selon les besoins du compilateur.
Pour les instructions, nous définissons une classe qui porte les services minimaux que doivent rendre les instructions, à savoir s'identifier comme exécutable, puis produire les effets d'une exécution.

```
class uneInstruction
: public unContenu
{
public:
  bool estExecutable() const;
  virtual void executer(unProcesseur&, uneMemoire&) const = 0;
};
```

avec :

```
bool uneInstruction::estExecutable() const { return true; };
```

Dès lors, la mémoire se définit comme un tableau de cases mémoires indexé par des adresses, d'où :

```
class uneMemoire
{
protected:
  int taille;
  unContenu* *cases;
public:
```

```
  uneMemoire(int=10);
  ~uneMemoire();
public:
  uneDonnee& lireDonnee(const uneAdresse&) const;
  uneInstruction& lireInstruction(const uneAdresse&) const;
  void ecrire(const uneAdresse&, const unContenu&);
};
```

avec :

```
uneMemoire::uneMemoire(int t)
: taille(t), cases(new unContenu*[t])
{
  for (int i=0; i<taille; i++) cases[i] = NULL;
};

uneMemoire::~uneMemoire() {
  if (cases) delete [] cases; };

uneDonnee& uneMemoire::lireDonnee(const uneAdresse& a) const {
  if (! cases[a]->estExecutable()) return (uneDonnee&)*cases[a];
  //  sinon erreur a traiter
};

uneInstruction& uneMemoire::lireInstruction(const uneAdresse& a) const {
  if (cases[a]->estExecutable()) return (uneInstruction&)*cases[a];
  //  sinon erreur a traiter
};

void uneMemoire::ecrire(const uneAdresse& a, const unContenu& c) {
  cases[a] = &c; };
```

Notons que le constructeur attend un entier spécifiant la capacité (nombre de cases) de la mémoire. Deux fonctions permettent l'accès aux cases mémoire via leur adresse. Nous n'avons pas exploité l'opérateur `[]` : les deux compilateurs essayés [1] indiquaient un problème d'ambiguïté.

Nous pouvons enfin définir les propriétés générales d'un processeur :

```
class unProcesseur
{
protected:
  uneAdresse CO;
  bool halted;
public:
  unProcesseur(uneAdresse);
public:
  void executer(const uneAdresse&, uneMemoire&);
  virtual void executer(uneMemoire&);
};
```

avec :

```
unProcesseur::unProcesseur(uneAdresse a)
: CO(a), halted(false)
{};

void unProcesseur::executer(const uneAdresse& a, uneMemoire& m) {
  CO = a;
  executer(m); };

void unProcesseur::executer(uneMemoire& m) {
```

1. GNU C++ 2.7.2 et Sun CC 3.0.1

```
while (! halted)
  m.lireInstruction(CO).executer(*this, m); };
```

Tout processeur se caractérise au moins par son compteur ordinal `CO`, et est capable d'exécuter un programme en mémoire.

Les deux méthodes définies diffèrent simplement par l'indication explicite ou non (dans ce cas, usage du compteur ordinal) de l'adresse de la prochaine instruction à exécuter.

Pour la simulation proprosée dans l'exercice, il suffit de définir complètement les caractéristiques propres de notre processeur :

```
class ProcEX0
: public unProcesseur
{
protected:
  uneDonnee regA, regB;
public:
  enum unRegistre {A, B};

  class uneInstructionProcEX0;
  class LDI;
  class LDD;
  class ADD;
  class ST;
  class NEQ;
  class JMP;
  class HALT;
public:
  ProcEX0(uneAdresse);
protected:
  uneDonnee& operator[](unRegistre) const;
friend LDI;
friend LDD;
friend ADD;
friend ST;
friend NEQ;
friend JMP;
friend HALT;
};
```

Notons que la classe encapsule tout ce qui est spécifique du processeur :

- les registres
- les instructions

L'implantation de ses méthodes est :

```
ProcEX0::ProcEX0(uneAdresse a)
: unProcesseur(a), regA(0), regB(0)
{};

uneDonnee& ProcEX0::operator[](unRegistre r) const {
  switch (r) {
  case A:
    return regA;
  case B:
    return regB; }};
```

Toutes les instructions sont définies par :

```
class ProcEX0::uneInstructionProcEX0
  : public uneInstruction
  {
```

```
  public:
    inline void executer(unProcesseur&, uneMemoire&) const;
  protected:
    virtual void exec(ProcEX0&, uneMemoire&) const = 0;
  };

class ProcEX0::LDI
  : public ProcEX0::uneInstructionProcEX0
  {
  private:
    ProcEX0::unRegistre reg;
    uneDonnee           don;
  public:
    LDI(ProcEX0::unRegistre, uneDonnee);
  public:
    void exec(ProcEX0&, uneMemoire&) const;
  };

class ProcEX0::LDD
  : public ProcEX0::uneInstructionProcEX0
  {
  private:
    ProcEX0::unRegistre reg;
    uneAdresse          adr;
  public:
    LDD(ProcEX0::unRegistre, uneAdresse);
  public:
    void exec(ProcEX0&, uneMemoire&) const;
  };

class ProcEX0::ADD
  : public ProcEX0::uneInstructionProcEX0
  {
  private:
    ProcEX0::unRegistre reg;
    uneAdresse          adr;
  public:
    ADD(ProcEX0::unRegistre, uneAdresse);
  public:
    void exec(ProcEX0&, uneMemoire&) const;
  };

class ProcEX0::ST
  : public ProcEX0::uneInstructionProcEX0
  {
  private:
    ProcEX0::unRegistre reg;
    uneAdresse          adr;
  public:
    ST(ProcEX0::unRegistre, uneAdresse);
  public:
    void exec(ProcEX0&, uneMemoire&) const;
  };

class ProcEX0::NEQ
  : public ProcEX0::uneInstructionProcEX0
  {
  private:
    ProcEX0::unRegistre reg;
```

```
    uneAdresse          adr;
  public:
    NEQ(ProcEX0::unRegistre, uneAdresse);
  public:
    void exec(ProcEX0&, uneMemoire&) const;
  };

class ProcEX0::JMP
  : public ProcEX0::uneInstructionProcEX0
  {
  private:
    uneAdresse adr;
  public:
    JMP(uneAdresse);
  public:
    void exec(ProcEX0&, uneMemoire&) const;
  };

class ProcEX0::HALT
  : public ProcEX0::uneInstructionProcEX0
  {
  public:
    HALT();
  public:
    void exec(ProcEX0&, uneMemoire&) const;
  };
```

avec l'implantation suivante :

```
void ProcEX0::uneInstructionProcEX0::executer(unProcesseur& p, uneMemoire& m) const {
  exec((ProcEX0&)p, m); };

ProcEX0::LDI::LDI(unRegistre r, uneDonnee d)
: reg(r), don(d)
{};

void ProcEX0::LDI::exec(ProcEX0& p, uneMemoire&) const {
  p[reg] = don;
  p.CO = p.CO+1;
};

ProcEX0::LDD::LDD(unRegistre r, uneAdresse a)
: reg(r), adr(a)
{};

void ProcEX0::LDD::exec(ProcEX0& p, uneMemoire& m) const {
  p[reg] = m.lireDonnee(adr);
  p.CO = p.CO+1;
};

ProcEX0::ST::ST(unRegistre r, uneAdresse a)
: reg(r), adr(a)
{};

void ProcEX0::ST::exec(ProcEX0& p, uneMemoire& m) const {
  m.ecrire(adr, p[reg]);
  p.CO = p.CO+1;
};

ProcEX0::ADD::ADD(unRegistre r, uneAdresse a)
```

```
: reg(r), adr(a)
{};

void ProcEX0::ADD::exec(ProcEX0& p, uneMemoire& m) const {
  p[reg] = p[reg]+m.lireDonnee(adr);
  p.CO = p.CO+1;
};

ProcEX0::NEQ::NEQ(unRegistre r, uneAdresse a)
: reg(r), adr(a)
{};

void ProcEX0::NEQ::exec(ProcEX0& p, uneMemoire&) const {
  if (p[reg] != 0)
    p.CO = adr;
  else
    p.CO = p.CO+1;
};

ProcEX0::JMP::JMP(uneAdresse a)
: adr(a)
{};

void ProcEX0::JMP::exec(ProcEX0& p, uneMemoire&) const {
  p.CO = adr;
};

ProcEX0::HALT::HALT()
{};

void ProcEX0::HALT::exec(ProcEX0& p, uneMemoire&) const {
  p.halted = true;
};
```

La fonction `main` permettant de tester le programme donné en exemple se limite alors à :

```
int main () {
  uneMemoire   m(10);
  ProcEX0      p(1);

  m.ecrire(1, ProcEX0::LDD  (ProcEX0::A, 8));
  m.ecrire(2, ProcEX0::LDI  (ProcEX0::B, 0));
  m.ecrire(3, ProcEX0::ADD  (ProcEX0::A, 10));
  m.ecrire(4, ProcEX0::ADD  (ProcEX0::B, 9));
  m.ecrire(5, ProcEX0::NEQ  (ProcEX0::A, 3));
  m.ecrire(6, ProcEX0::ST   (ProcEX0::B, 8));
  m.ecrire(7, ProcEX0::HALT ());
  m.ecrire(8, uneDonnee(10));
  m.ecrire(9, uneDonnee(2));
  m.ecrire(10,uneDonnee(-1));

  p.executer(1, m);
};
```