ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation Orientée Objet, sections MA et PH
Chappelier J.-C.

Programmation Orientée Objet (SMA/SPH) : Correction examen final

17 août 2020

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre examen annulé dans le cas contraire.

1. Vous disposez de deux heures pour faire cet examen (9h15 – 11h15).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.
   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
   Vous disposez, si nécessaire, de pages blanches supplémentaires en fin de sujet.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. L'examen comporte cinq exercices indépendants, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 107) :
   1. question courte : 13 points ;
   2. conception : 22 points ;
   3. programmation : 28 points ;
   4. déroulement de programme (1) : 26 points ; et
   5. déroulement de programme (2) : 18 points.

   Tous les exercices comptent pour la note finale.

Question 1 – Gestion de données [13 points]

Considérez la classe suivante :

```
class DataList
{
private:
    vector<Data*> content;
};
```

① **[5 points]** Implémentez une méthode `add()` qui reçoit une référence constante sur un objet `Data`, et en ajoute une copie au contenu de `DataList`. Chaque `DataList` est responsable des `Data` qu'elle contient et ne les partage en aucune façon avec aucune autre partie du code.

Vous pouvez faire toutes les suppositions *raisonnables* dont vous avez besoin sur la classe `Data`, mais veuillez les expliquer brièvement.

② **[8 points]** Implémentez ensuite l'opérateur d'affectation pour la classe `DataList` (qui doit être copiable).

Réponses :

Le fait que « *chaque* `DataList` *est responsable des* `Data` *qu'elle contient et ne les partage en aucune façon avec aucune autre partie du code* » impose donc une copie.

Le problème est que dans le cas le plus général `Data` pourrait être une classe dont on attend un comportement polymorphique (p.ex. une classe abstraite). On supposera donc que `Data` possède une méthode de copie *polymorphique* : `Data* clone() const;` ou alors, mais c'est plus restrictif (moins général), que `Data` n'est pas une super-classe et est copiable (p.ex. avec un constructeur de copie).

```
void DataList::add(Data const& d)
{
  content.push_back(d.clone());
}
```

Pour l'opérateur d'affectation, le point clé est de faire une *copie profonde*. On peut soit procéder comme dans le cours avec un passage par valeur et un `swap()`, mais il est alors impératif de définir le constructeur de copie, soit coder directement cette copie dans l'opérateur d'affectation.

Dans la version avec copie et échange (*swap*), il faudrait aussi redéfinir la fonction `swap` (sinon la version par défaut `std::swap()` va faire une boucle infinie vue qu'elle appelle justement `operator=()`).

Et dans les 2 cas, il faudrait faire la destruction des copies (`delete`) dans le destructeur.

(Codes solutions au dos.)

Version avec `swap` :

```
// constructeur de copie avec copie profonde
DataList::DataList(DataList const& other)
{
  for (auto ptr : other.content) add(*ptr);
}

// copy-swap idiom
DataList& DataList::operator=(DataList other)
{
  swap(*this, other);
  return *this;
}

// fonction swap() pour éviter une boucle infinie
void swap(DataList& a, DataList& b)
{
  swap(a.content, b.content);
}
```

Version « locale » :

```
DataList& DataList::operator=(DataList const& other)
{
  if (this != &other) {
    empty(); // assume DataList::empty() exists and frees all content
    for (auto ptr : other.content) add(*ptr);
  }
  return *this;
}
```

Commentaires de correction : de manière générale, cet exercice n'a pas vraiment été bien réussi. Les principales erreurs ont été :

- q1.2 : ne pas savoir ce qu'est l'opérateur d'affectation ;
- q1.1 : ne pas formuler les hypothèses nécessaires sur `Data` ;
- q1.1 : copie non-polymorphique ;
- q1.1 : `push_back` de `unique_ptr` (alors que la collection contient explicitement des pointeurs à la C) ;
- q1.1 : ajout de pointeur vers une variable locale ;
- q1.2 : copie de surface ;
- q1.2 : pour ceux qui n'utilisent pas le `swap`, oubli du test d'auto-affectation ;
- q1.2 : pour ceux qui n'utilisent pas le `swap`, très peu pensent à libérer le contenu avant de faire l'ajout ;
- q1.2 : retour d'une copie (type de retour `DataList`, sans référence).

Question 2 – Conception [22 points]

On souhaite modéliser informatiquement une boîte d'envois postaux, dans laquelle on peut placer différents courriers. Les courriers stockés peuvent être de deux types :

- des lettres, dont le tarif d'affranchissement dépend du caractère d'urgence de chaque lettre : en tarif normal, on affranchit les lettres au coût de 3 francs l'unité et en courrier rapide à 5 francs l'unité ;
- des colis, dont l'affranchissement est de 10 francs par unité plus 2 francs par kilo (poids).

Les lettres et colis ont par ailleurs d'autres spécificités propres non explicitées ici, lesquelles les différencient. Ces spécificités propres sont simplement ignorées dans cet examen.

On souhaite créer une méthode `affranchir()` qui affranchit automatiquement l'ensemble des courriers contenus dans la boîte en fonction de leur type précis.

① **[18 points] Conception**

Sans donner tout le détail du code complet (on ne demande ici qu'une *conception*), écrivez **en C++** les classes, les relations d'héritage, les attributs et les méthodes des classes, les droits d'accès et les éventuelles fonctions (externes) que vous utiliseriez pour implémenter un tel programme.

Précisez les types des attributs et les prototypes des méthodes/fonctions, mais ne donnez pas leur définition (on répète : il s'agit ici de la partie *conception*, pas de l'implémentation ; c.-à-d. les prototypes, pas les définitions des méthodes).

Indiquez également les constructeurs et les destructeurs *lorsque nécessaire* (sans leur définition).

② **[4 points] Programmation**

Implémentez (c.-à-d. définissez) la méthode `affranchir()`. Commentez votre solution en insistant sur les principaux aspects et explicitez son fonctionnement.

Suite des réponses à la Question 2 :

Voici *une* réponse possibe :

```
class Courrier {
  // courrier en général; pour factoriser le code commun

public:
  virtual ~Courrier() = default;
  virtual void affranchir() = 0; // voir commentaires

  // attributs: voir commentaires
};

class Lettre : public Courrier {
public:
  Lettre(bool urgence = false);
  virtual void affranchir() override;

private:
  bool urgent; // par exemple
};

class Colis : public Courrier {
public:
  Colis(double poids);

  virtual void affranchir() override;

private:
  double poids_;
};

class Boite {
private:
  vector<Courrier*> contenu; // pourrait être un unique_ptr
public:
  void ajoute(Courrier*); // ou par valeur
  void affranchir() const;
}
```

J'ai mis tous les attributs en `private`, mais on peut les tolérer en `protected`.

Pour toutes les classes : s'il y a des attributs, il faut le constructeur pour les initialiser ; s'il n'y a pas d'attrbut, le contructeur par défaut par défaut suffit, **sauf** pour les sous-classes dont la super-classe a un constructeur spécifique.

Pour la classe `Courrier` :

— il n'est pas forcément nécessaire d'avoir des attributs, mais un attribut `tarif`, p.ex. peut se concevoir ;
cela peut aussi être une méthode (pour fixer le tarif), mais cela semble un peu faire double emploi avec `affranchir()`, mais on peut le tolérer (séparation du fait de fixer le tarif et le

fait d'affranchir) ;
si `tarif` est un attribut ici, on attend alors un constructeur ;
- la méthode `affranchir()` peut se nommer autrement pour les courriers (mais doit être celle utilisée dans le corps de `Boite::affranchir()`, sous-question ②) ;
- la méthode `affranchir()` **doit** être `virtual` ; elle n'est pas contre pas forcément nécessaire virtuelle pure ;
  cette méthode *peut* aussi être `const`, mais ce n'est pas obligé (au fond quand on affranchit, on modifie la lettre : on colle le timbre dessus `;-)`)
- il faut un destructeur virtuel puisque l'on attend un comportement polymorphique de cette classe ;
  du coup, pour les plus avancés, cela pourrait signifier devoir remettre toute la copie (et le déplacement) ; mais ceci est totalement optionnel pour le niveau de ce cours (mais ne doit pas être considéré comme faux si fait).

Pour les classes `Lettre`/`Colis` :
- l'héritage pourrait être virtuel si l'on imagine pouvoir avoir des colis qui sont aussi des lettres, mais ceci est un peu tordu et doit donc absolument avoir été expliqué, sinon on considèrera un héritage virtuel comme faux ;
- la méthode `affranchir()` doit être strictement la même que pour `Courrier` ; `virtual` et `override` sont optionnels ;
- les `Lettres` doivent avoir quelque chose permettant de fixer le tarif ;
- les `Colis` doivent avoir quelque chose permettant de connaitre le poids ;
- ces classes ayant un attribut, on attend (au moins) un constructeur ; la valeur par défaut donné ici pour l'urgence des lettres est totalement optionnelle, juste fournie comme illustration possible.

Pour la classe `Boite` :
- la `Boite` peut ne pas être propriétaire de son contenu mais simplement un moyen de lister des courrier gérés par ailleurs (c.-à.d. que l'on peut se contenter d'avoir des pointeurs à la C dans `add`) ;
  ces pointeurs peuvent même être `const` ; (si l'affranchissement des courriers est `const`) ;
- il n'est pas nécessaire d'avoir un constructeur ;
- la méthode `affranchir()` **doit** être `const` ici ;
- le type de retour de la méthode `affranchir()` est assez libre tant que ça reste clair et fait sens, par exemple retourner la somme des affranchissement, ou juste rien comme ici ;
- rien besoin de faire pour la copie (ni l'affectation) car un clône de la boîte pourrait pointer sur les mêmes courriers et donc la copie de surface offerte par défaut pourrait suffire ;
- *par contre*, si des `unique_prt` ont été choisis, il est nécessaire d'avoir une copie polymorphique dans les `Courrier`.

Commentaires de correction : question globalement bien réussie. Les principales erreurs ont été :
- destructeur de `Colis` non virtuel ;
- oubli de constructeurs ou incomplets ;
- oubli de la `Boite` elle-même ;
- oubli de la méthode `Boite::add()` ;
- pas de `const` pour `Boite::affranchir()` ;
- en faire trop en rajoutant d'autres choses inutiles (constructeur de copie, opérateur d'affectation, etc.) ;

— utilisation d'un `int` ou `unsigned int` pour le poids d'un colis.

② version minimale sans sophistication :

```
void Boite::affranchir() const {
  for (auto courrier : contenu)
    courrier->affranchir();
}
```

Le point de fond est que la collection doit être hétérogène et la méthode `affranchir()` de la classe `Courrier` doit être virtuelle.

Explicitation du fonctionnement : polymorphisme (ou plus exactement résolution dynamique des liens) par pointeurs et méthode virtuelle.

Question 3 – Programmation [28 points]

Dans cette question, on vous demande de complèter les parties manquantes d'un programme C++ dont le but est de modéliser différents types de comptes bancaires : comptes de base, compte épargne et compte à prestations facturées (dits « comptes payants ») :

- les comptes de base possèdent un identificateur unique, ainsi qu'un certain montant (solde du compte) ; on peut de plus ajouter ou retirer de l'argent de ces comptes et afficher leurs informations ;
- les comptes épargne sont des comptes de base qui en plus rémunèrent leur argent suivant un certain taux ;
- les comptes payants sont des comptes de base pour lesquels chaque opération (dépôt ou retrait) coûte de l'argent.

Dans le code fourni ci-dessous, certaines portions sont manquantes et on vous demande de les écrire, à l'endroit des commentaires « `// COMPLÉTER ICI ...` » :

- les méthodes `deposer()` et `retirer()` de `Compte` ; les sommes déposées ou retirées doivent être strictement positives (sinon il ne se passe simplement rien) ;
  de plus la méthode `retirer()` doit indiquer en retour si le compte est à découvert ou non ;
- la première ligne de la définition de chacune des classes `CompteEpargne` et `ComptePayant` ;
- le constructeur de chacune des classes `CompteEpargne` et `ComptePayant` ;
  on vous demande d'obliger, lors de la construction, de fournir les valeurs des attributs spécifiques supplémentaires (taux, taxe) ;
- la méthode `affiche()` de chacune des classes `CompteEpargne` et `ComptePayant` ;
  ces méthodes doivent afficher les informations communes à tous les comptes exactement comme la classe `Compte`, puis afficher ensuite toutes les informations supplémentaires des classes concernées ;
- les méthodes `deposer()` et `retirer()` de la classe `ComptePayant`.

Répondez directement sur la donnée.

```
class Compte { // Un compte en banque
private:
  unsigned int const id; // numéro de compte
  double solde_;         // état du compte (solde)

public:
  Compte(unsigned int numero) : id(numero), solde_(0.0) {}

  virtual ~Compte() = default;

  double solde()  const { return solde_; }
  double numero() const { return id;     }

  // Méthode pour tester si le compte est à découvert ou non
  bool decouvert() const { return (solde() < 0.0); }

  // ... suite du code sur la page de droite ...
```

```
  // Méthode pour déposer de l'argent sur le compte
  virtual void deposer(double somme) {
    if (somme > 0.0) solde_ += somme;
  }


  // Méthode pour retirer de l'argent du compte
  virtual bool retirer(double somme) {
    if (somme > 0.0) solde_ -= somme;
    return decouvert();
  }

  // affichage
  virtual void affiche() const {
    cout << "Etat du compte ";
    affiche_nom();
    cout << " #" << numero() << " : " << solde() << endl;
  }

private:
  virtual void affiche_nom() const { cout << "de base"; }
};

// ------------------------------------------------------------
class CompteEpargne : public Compte {
private:
  double interet; // taux d'intérêt

public:
  CompteEpargne (unsigned int numero, double taux)
    : Compte(numero), interet(taux) {
    /* Optionnel: rend les choses crédibles si mal utilisées.
     * On pourrait à la place lever des exceptions.         */
    if (interet < 0.0) interet = -interet;
    if (interet > 1.0) {
      if (interet <= 100.0) {
        // l'utilisateur l'a peut être donné en %
        interet /= 100.0;
      } else {
        interet = 1.0;
      }
    }
  }

  virtual ~CompteEpargne() = default;

  double taux() const { return interet; }
  void taux(double t) { interet = t; }

  // ... suite du code au dos ...
```

suite au dos ☞

```
  // calcul des intérêts
  double calcul_interets() const { return taux() * solde(); }

  // ajout des intérêts
  void credite_interets() { deposer(calcul_interets()); }

  // affichage
  void affiche() const {
    Compte::affiche();
    cout << "Taux d'intérêts : " << taux() << endl;
  }

private:
  virtual void affiche_nom() const override {
    cout << "épargne";
  }
};

// ------------------------------------------------------------
class ComptePayant : public Compte {
private:
  double taxe_; // taxe sur les opérations

public:
  ComptePayant(unsigned int numero, double taxe)
    : Compte(numero), taxe_(taxe) {}

  virtual ~ComptePayant() = default;

  double taxe() const { return taxe_; }
  void taxe(double t) { taxe_ = t; }

  // méthode de dépot d'argent
  void deposer(double somme) {
    Compte::deposer(somme);
    Compte::retirer(taxe()); // voir commentaire ci-dessous
  }

  // ... suite du code sur la page de droite ...
```

Concernant la taxe sur les comptes payants, il n'a pas été précisé si elle est fixe (additive ; comme dans la solution ci-dessus) ou proportionnelle (multiplicative : `taxe() * somme`, donc). Les deux solutions sont acceptables (`taxe() * somme / 100.` est aussi acceptable).

```
  // Méthode pour retirer de l'argent du compte
  bool retirer(double somme) {
    Compte::retirer(somme);
    Compte::retirer(taxe());
    return decouvert();
  }

  // affichage
  void affiche() const {
    Compte::affiche();
    cout << "Taxe : " << taxe() << endl;
  }

private:
  virtual void affiche_nom() const override {
    cout << "payant";
  }
};
```

Commentaires de correction : les constructeurs et l'héritage sont bien compris. Les principales erreurs ont été :

- beaucoup de copié-collé pour les fonctions `affiche()`, `retirer()` et `deposer()` ;
- type de retour `void` pour la méthode `retirer()` et confusion entre affichage et valeur de retour ;
- oubli du `virtual` pour `Compte::deposer()` et `retirer()` ;
- utiliser `int` dans les prototypes (bien que `solde_` soit un `double`) ;
- changer le prototype d'une methode lorsque redéfinie dans une sous-classe.

Question 4 – Déroulement de programme [26 points]

On considère le début de programme suivant, qui, même s'il ne suit pas tous les conseils du cours, compile et s'exécute sans erreurs :

```
1   #include <iostream>
2   using namespace std;
3
4   class A {
5   private:
6     int x;
7   protected:
8     int y, z;
9   public:
10    int t;
11
12    A() : x(0), z(0), t(0)
13    { cout << "A()" << endl; }
14    A(bool inutile) : z(5) { y = 7; }
15    virtual ~A() {
16      cout << "bye A" << endl;
17    }
18    int calcul() {
19      int z(42);
20      return 101;
21    }
22  };
23
24  class B : public A {
25    int w;
26  public:
27    B() { cout << "B()" << endl; }
28    B(int u) : A(false), w(3) {
29      cout << "B(" << u << ')' << endl;
30      z = z + u;
31    }
32    ~B() {
33      cout << "bye B" << endl;
34    }
35  };
36
37  class C : public B {
38  protected:
39    int s;
40  public:
41    C() : B(4), s(7) {
42      cout << "C()" << endl;
43    }
44    ~C() {
45      cout << "bye C" << endl;
46    }
47  };
48
49  class D : public A {
50    int r;
51  public:
52    D() : r(2) {
53      cout << "D:D()" << endl;
54    }
55    D(bool p) : A(p) {
56      cout << "D(" << p << ')' << endl;
57    }
58    virtual ~D() {
59      cout << "bye D" << endl;
60    }
61    int calcul() { return r; }
62  };
63
64  class E : public D {
65  public:
66    int q;
67    E(int r) {
68      cout << "E(" << r << ')' << endl;
69      y = calcul();
70      if   (r == 2) { q =  8; }
71      else          { q = 10; }
72    }
73    virtual ~E() {
74      cout << "bye E" << endl;
75    }
76  };
77
78  int main() {
79    A a;
80    B b;
81    C c;
82    D d(true);
83    E e(11);
84
85    // *** ICI
86    return 0;
87  }
```

① **[15 points]** Il vous est demandé d'indiquer, dans le tableau suivant, la valeur des attributs `x, y, z, t, w, s, r` et `q` pour chacun des objets `a, b, c, d` et `e` au moment de l'éxécution de la ligne correspondant au commentaire « `// *** ICI` » (ligne 85).

Si un des attributs n'existe pas pour l'objet en question, indiquez-le par un tiret « — » à la place de la valeur. Si une valeur n'est pas connue (non initialisée), indiquez le par un point d'interrogation « ? ». Si, par contre, *vous*, vous ne connaîssez pas la réponse, laissez en blanc ! **Toute mauvaise valeur sera pénalisée** : −0.25 point par réponse incorrecte contre 0.5 point par réponse non-tiret correcte, 0.15 si c'est un tiret.

		attribut							
		x	y	z	t	w	s	r	q
variable	a	0	?	0	0	—	—	—	—
	b	0	?	0	0	?	—	—	—
	c	?	7	9	?	3	7	—	—
	d	?	7	5	?	—	—	?	—
	e	0	2	0	0	—	—	2	10

② **[11 points]** Dans cette sous-question, on s'intéresse aux droits d'accès pour chacune des classes.

Imaginez pour cela que chaque classe possède une méthode `f()` de la forme

```
void X::f(Y const& autre) {
  m = 33;
  cout << autre.m;
}
```

où *X* représente la classe en question (`A`, `B`, `C`, `D` ou `E`), *Y* une autre classe (possiblement la même : `A`, `B`, `C`, `D` ou `E`), et *m* un attribut possible (`x`, `y`, `z`, `t`, `w`, `s`, `r` ou `q`).

Pour les différents cas possibles considérés dans le tableau ci-contre, indiquez par `oui` ou par `non` dans chacune des cases du tableau, si la ligne de code correspondante est acceptée sans erreur à la compilation.

Par exemple, pour *X*=`A`, *Y*=`A` et *m*=`t`, la méthode `f()` considérée est :

```
void A::f(A const& autre) {
  t = 33;
  cout << autre.t;
}
```

dont les deux lignes compilent et nous avons donc répondu « `oui` » dans les deux cases correspondantes.

X	*Y*	*m*	*m* = 33;	cout << autre.*m*;
A	A	t	oui	oui
		x	oui	oui
		z	oui	oui
	B	z	oui	oui
B	B	x	**NON**	**NON**
		w	oui	oui
C	A	t	oui	oui
		x	**NON**	**NON**
		z	oui	**NON**
	B	z	oui	**NON**
		w	**NON**	**NON**
E	B	z	oui	**NON**

En cas de doute, laissez un blanc, car **toute mauvaise valeur sera pénalisée** : −0.5 point par réponse incorrecte contre 0.5 point par réponse correcte.

Commentaires de correction : les principales erreurs ont été :

- 1.`c` : `z` vaut 5 au lieu de 9
- 1.`e` : `y` vaut 101 au lieu de 2
- 1.`e` : `q` vaut 8 au lieu de 10
- 2.`A` : la réponse en bas à droite de cette question (`B-z-autre.m`) est souvent fausse ou pas faite ;
- 2.`C` : `A-z-autre.m` et `B-z-autre.m` sont aussi souvent faux (complément du point ci-dessus) ;
- 2.`E` : cette dernière ligne a souvent été complétée comme oui-oui ou non-non.

Question 5 – Déroulement de programme [18 points]

Le programme ci-dessous compile et s'exécute sans erreurs. Indiquez l'affichage exact qu'il produit puis *justifiez* votre réponse en expliquant les points *importants* (on ne vous demande pas ici de paraphraser le code, ni de justifier chaque affichage individuellement, mais bien de montrer que vous avez compris ce qui se passe !)
Vous pouvez répondre à droite du code et, si nécessaire, annoter le code lui-même.

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   class Legume {
6   public:
7     Legume() : qualite(7), poids(1.5) {
8       cout << "Encore un légume" << endl;
9     }
10    ~Legume() {
11      cout << "Et un légume de moins" << endl;
12    }
13    virtual void affiche () const {
14      cout << "Qualité       : " << qualite << endl;
15      cout << "Poids (masse) : " << poids   << endl;
16    }
17  protected:
18    int qualite;
19  private:
20    double poids;
21  };
22
23  class Poireau : public Legume {
24  public:
25    Poireau(double h) : hauteur(h) {
26     cout << "C'est un poireau" << endl;
27    }
28    virtual ~Poireau() {
29      cout << "Un poireau à la poubelle" << endl;
30    }
31    virtual void affiche() const {
32      cout << "Hauteur : " << hauteur << endl;
33    }
34  private:
35    double hauteur;
36  };
37
38    // ... suite du code sur la page de droite ...
```

```
38  class Courgette : public Legume {
39  public:
40    Courgette(double L) : longueur(L) {
41     cout <<"C'est une courgette" << endl;
42    }
43    virtual ~Courgette(){
44      cout << "La courgette se ratatine" << endl;
45    }
46    void affiche () const {
47      cout << "Longueur : " << longueur << endl;
48    }
49  private:
50    double longueur;
51  };
52
53  class Navet : public Legume {
54  public:
55    Navet(double r) : rayon(r) {
56     cout << "C'est un navet" << endl;
57    }
58    ~Navet() {
59      cout << "Le navet s'en va" << endl;
60    }
61    virtual void affiche () const {
62      cout << "Rayon : " << rayon << endl;
63    }
64  private:
65    double rayon;
66  };
67
68  int main() {
69    vector<Legume*> panier({
70        new Navet(4.5),
71        new Courgette(12.3),
72        new Poireau(25.6)
73    });
74
75    for (auto x : panier) {
76      x->affiche();
77      cout << "----" << endl;
78    }
79    for (auto x : panier) {
80      delete x;
81    }
82    return 0;
83  }
```

CORRECTION AU DOS

```
Encore un légume
C'est un navet
Encore un légume
C'est une courgette
Encore un légume
C'est un poireau
Rayon : 4.5
----
Longueur : 12.3
----
Hauteur : 25.6
----
Et un légume de moins
Et un légume de moins
Et un légume de moins
```

Constructeurs : il y a chaque fois appel du constructeur correspondant ; pour les sous-classes, même s'il n'est pas explicité, il y a appel au constructeur par défaut de la super-classe (puisqu'il existe).

`affiche()` : la méthode `affiche()` est virtuelle et on l'appelle via des pointeurs, il y a donc *polymorphisme* (résolution dynamique des liens).

Destructeurs : le destructeur de `Legume` n'étant pas virtuel, c'est lui qui est appelé (et non pas le destructeur de la sous-classe) ; que le destructeur de la sous-classe ait été défini comme virtuel ou non, cela n'y change rien.

Commentaires de correction : globalement, l'exercice a été compris et bien réussi. Les principales erreurs ont été :

- ne pas remarquer (ou pas comprendre l'impact) que le destructeur de `Legume` n'est pas virtuel ;
- quelques-uns ont affiché le poids et la qualité de la `Courgette` au lieu de sa longueur (car sa fonction `affiche()` est virtuelle).