enseignant : Philippe Reitz
année universitaire : 2001-2002

Programmation

unité de Travaux Pratiques

TP n°3 : C++

[Corrigé]

Exercice 1 : Une classe pointeur **1**

Exercice 2 : Constructeur de recopie - affectation **1**

Exercice 3 : membres statiques **5**

Exercice 4 : les animaux **7**

Exercice 1 : Une classe pointeur

Définir une classe générique `Pointeur` telle qu'un objet de type `Pointeur<T>`, où `T` est un type quelconque, ait les mêmes fonctionnalités qu'un objet de type `T*` (pointeur sur `T`), c'est à dire, en supposant que `o` soit de type `Pointeur<T>` :

- `o` est initialisable avec une valeur de type `T*`.
- `*o` a la même signification que l'opérateur pointeur associé.

Correction

```
#include <iostream.h>

template <class T> class Pointeur {
  T* ptr;
public:
  Pointeur(T*v=0) : ptr(v) {};
public:
  inline T& operator*() { return *ptr; };
//  inline T& operator->() { return *ptr; };
};

int main() {
  int i = 1;
  Pointeur<int> p(&i);
  *p += 2;
  cout << i << endl; };
```

Exercice 2 : Constructeur de recopie - affectation

Observer le comportement du programme suivant :

```
#include <iostream.h>
#define msg(m) cout << m << endl

class Objet {
  int a;
public:
  Objet() : a(-1) { msg("Objet::Objet()"); };
  Objet(int v) : a(v) { msg("Objet::Objet(" << a << ")"); };
  Objet(const Objet& o) : a(o.a) { msg("Objet::Objet(Objet(" << a << "))"); };
  ~Objet() { msg("Objet::~Objet(" << a << ")"); };
public:
  Objet& operator=(const Objet& o) {
    msg("Objet(" << a << ") = Objet(" << o.a << ")"); a = o.a; return *this; };
  Objet operator+(const Objet& o) { return Objet(a+o.a); };
};

Objet fonc1(Objet o) { return o+o; };

Objet& fonc2(Objet& o) { o = o+o; return o; };

int main () {          msg("---");
  Objet o1 = Objet(1); msg("---");
  Objet o2;            msg("---");
  Objet o3 = o1+o2;    msg("---");
  o3 = o1+o2;          msg("---");
  o3 = fonc1(o1);      msg("---");
  o3 = fonc2(o2);      msg("---"); };
```

Expliquer la séquence des messages obtenue.

Correction

Les messages affichés sur le terminal ne sont pas nécessairement les mêmes selon le compilateur C++ exploité. Ce corrigé s'appuie donc sur une session[1] dont les messages sont les suivants :

```
---
Objet::Objet(1)
---
Objet::Objet()
---
Objet::Objet(0)
---
Objet::Objet(0)
Objet(0) = Objet(0)
Objet::~Objet(0)
---
Objet::Objet(Objet(1))
Objet::Objet(2)
Objet::~Objet(1)
Objet(0) = Objet(2)
Objet::~Objet(2)
---
Objet::Objet(-2)
Objet(-1) = Objet(-2)
Objet::~Objet(-2)
Objet(2) = Objet(-2)
---
Objet::~Objet(-2)
Objet::~Objet(-2)
Objet::~Objet(1)
```

Nous omettons dans ce corrigé d'expliquer l'affichage des `---`, puisqu'ils sont affichés après chaque instruction de la fonction `main`.

- `Objet o1 = Objet(1)`

  La fonction `main` commence par définir une variable `o1` de type `Objet` ; la construction de cet objet s'effectue via le constructeur `Objet(int)`, d'où l'affichage de (`o1.a` vaut 1) :

  ```
  Objet::Objet(1)
  ```

- `Objet o2`

  Un deuxième objet `o2` est défini, via le constructeur `Objet()` (`o2.a` vaut -1) :

  ```
  Objet::Objet()
  ```

- `Objet o3 = o1+o2`

  Un troisième objet `o3` est défini à partir de la somme des 2 premiers. Cette définition s'effectue ainsi : dans la définition de l'opérateur `+`, le compilateur remarque que le résultat retourné est un nouvel objet dont le champ `a` est la somme des champs `a` des 2 objets en paramètre. Puisque ce résultat doit être, dans notre cas, rangé dans l'objet `o3` à définir, le compilateur optimise les opérations en faisant en sorte que le nouvel objet à retourner en résultat de `+` soit l'objet `o3`, d'où le seul message :

  ```
  Objet::Objet(0)
  ```

  Si cette optimisation n'avait pas existé, il est probable que l'affichage comporterait les messages suivants :

  ```
  Objet::Objet(0)
  Objet::Objet(Objet(0))
  Objet::~Objet(0)
  ```

  Autrement dit :

  1. allocation d'un objet temporaire afin de recevoir le résultat de l'opérateur `+`

1. compilateur GNU C++ 2.7.1

2. allocation de l'objet `o3` avec initialisation par recopie de l'objet temporaire
3. désallocation de l'objet temporaire, devenu inutile.

- `o3 = o1+o2`

  Pour l'affectation qui suit, la série de messages affichée est la suivante :

```
Objet::Objet(0)
Objet(0) = Objet(0)
Objet::~Objet(0)
```

  Cette séquence de trois messages s'explique ainsi :

1. tout d'abord, allocation d'un objet temporaire de type `Objet` dans lequel sera rangé le résultat de `o1+o2` ; la somme de ces 2 objets est un objet dont le champ `a` vaut la somme de `o1.a` et `o2.a`.
2. ensuite, affectation (via notre opérateur =) de cet objet temporaire dans l'objet `o3` → tous les champs de l'objet temporaire sont recopiés dans les champs de `o3`.
3. l'objet temporaire, devenu inutile, est détruit (désalloué).

- `o3 = fonc1(o1)`

  La trace observée commence à être plus complexe :

```
Objet::Objet(Objet(1))
Objet::Objet(2)
Objet::~Objet(1)
Objet(0) = Objet(2)
Objet::~Objet(2)
```

  L'instruction consiste à faire un appel à la fonction `fonc1` puis à affecter le résultat dans l'objet `o3`.

  Commençons par l'appel à la fonction : dans la définition de `fonc1`, le paramètre `o` est passé par valeur ; autrement dit, toute modification du paramètre dans le corps de la fonction ne doit affecter en rien l'objet passé en paramètre (ici `o1`).

  Lors de l'appel, un nouvel objet `o` est donc alloué sur la pile, initialisé par recopie de l'objet passé en paramètre lors de l'appel, soit `o1`. D'où le premier message :

```
Objet::Objet(Objet(1))
```

  Le corps de la fonction est alors exécuté ; il s'agit de retourner le résultat de la somme de 2 objets, qui lui même est un objet. Un nouvel objet est donc alloué pour recevoir temporairement le résultat, d'où le message :

```
Objet::Objet(2)
```

  L'appel étant terminé, tout objet alloué lors de cet appel est détruit ; c'est le cas du paramètre `o`, d'où :

```
Objet::~Objet(1)
```

  L'affectation dans `o3` peut alors être effectuée :

```
Objet(0) = Objet(2)
```

  Une fois l'affectation terminée, l'objet alloué temporairement pour recevoir le résultat de l'appel à la fonction est détruit :

```
Objet::~Objet(2)
```

- `o3 = fonc2(o2)`

  La trace observée est la suivante :

```
Objet::Objet(-2)
Objet(-1) = Objet(-2)
Objet::~Objet(-2)
Objet(2) = Objet(-2)
```

Il s'agit encore une fois d'un appel à une fonction, presque en tous points semblable au précédent, excepté que le passage du paramètre s'effectue cette fois-ci par référence.

Lors de l'appel, puisque le passage est du type *passage par référence*, aucun nouvel objet n'est alloué ; l'objet `o` n'existe pas en tant que tel : travailler avec `o`, c'est travailler avec l'objet passé lors de l'appel, soit ici `o2`. Aucun message n'apparait donc lors de cette phase.

Le corps de la fonction est alors exécuté. La première instruction consiste à calculer la somme de 2 objets et à affecter l'objet résultant dans un objet cible. Nous savons désormais que trois messages sont affichés : le premier lors de l'allocation d'un objet temporaire recevant le résultat, un second pour l'affectation proprement dite, et un troisième pour la désallocation de l'objet temporaire, d'où :

```
Objet::Objet(-2)
Objet(-1) = Objet(-2)
Objet::~Objet(-2)
```

La fonction se termine en retournant son résultat ; ce dernier est lui aussi une référence sur un objet existant, et donc aucun objet n'a à être alloué pour recevoir le résultat. Ce dernier est affecté à l'objet `o3`, d'où le message :

```
Objet(2) = Objet(-2)
```

- fin de `main`

  La fonction `main` se terminant, les trois objets alloués dans son corps sont détruits :

```
Objet::~Objet(-2)
Objet::~Objet(-2)
Objet::~Objet(1)
```

Avec un autre compilateur [2], nous avons obtenu la trace suivante :

```
---
Objet::Objet(1)
---
Objet::Objet()
---
Objet::Objet(0)
---
Objet::Objet(0)
Objet(0) = Objet(0)
---
Objet::Objet(Objet(1))
Objet::Objet(2)
Objet::~Objet(1)
Objet(0) = Objet(2)
---
Objet::Objet(-2)
Objet(-1) = Objet(-2)
Objet::~Objet(-2)
Objet(2) = Objet(-2)
---
Objet::~Objet(2)
Objet::~Objet(0)
Objet::~Objet(-2)
Objet::~Objet(-2)
Objet::~Objet(1)
```

Les différences résident simplement dans le fait que ce dernier compilateur détruit les objets alloués temporairement le plus tard possible.

2. Sun C++ 3.0.1

Exercice 3 : membres statiques

Écrire deux classes `A` et `B` telles que, pour chacune de ces deux classes, il est possible de connaître, via un appel de fonction, le nombre d'instances (d'objets) de leur type ayant été créées (allouées) au moment de l'appel.

Correction

Une première solution consiste à associer à chaque classe une variable globale représentant le compteur d'instances, d'où un code ressemblant à celui-ci :

```
int cpt_A = 0;

class A {
  int x;
public:
  A(int=0);
  ~A();
};

A::A(int v) : x(v) { cpt_A++; };

A::~A() { cpt_A--; };
```

Idem pour la classe `B`.

Cette solution possède un gros défaut : rien n'empêche l'usager de modifier lui-même la valeur de ces compteurs.

Un remède permet de pallier à ce défaut : si ces classes sont destinées à être offertes à une communauté de programmeurs, il est alors possible d'écrire dans deux fichiers séparés d'une part la définition de la classe (partie spécification, dans un fichier du genre `exo.h`), et d'autre part l'implantation des méthodes (partie implantation, dans un fichier du genre `exo.cpp`) ; nous aurions alors :

- fichier `exo.h`

```
class A {
  int x;
public:
  A(int=0);
  ~A();
};
```

- fichier `exo.cpp`

```
int cpt_A = 0;

A::A(int v) : x(v) { cpt_A++; };

A::~A() { cpt_A--; };
```

Les compteurs, bien que variables globales, ne sont pas référencées dans le fichier `exo.h`, en principe le seul qui est présenté à l'usager de nos classes → l'usager ne peut donc pas les exploiter directement.

La solution présentée permet bien d'associer à chaque classe une variable globale, mais il faudrait que cette variable ne puisse pas être modifiée autrement que par un ensemble de fonctions parfaitement identifiées.

La réponse propre à ce problème en `C++` consiste à exploiter les membres statiques :

```
class A {
  static int cpt;
  int x;
public:
  A(int=0);
  ~A();
public:
  static int nb();
};
```

```
int A::cpt = 0;
```

Notons l'initialisation du membre statique, que ne peut être réalisée ni dans la définition de la classe (interdiction de spécifier des valeurs par défaut pour les membres données d'une classe lors de sa définition), ni par un constructeur (sinon ce membre serait initialisé à chaque fois qu'un objet serait créé via ce constructeur ; or notre membre donnée ne doit être initialisé qu'une seule fois).

Le membre donnée `cpt` étant **private**, il n'est donc accessible qu'aux seules méthodes associées à la classe `A` ; de plus, le membre étant statique, il sera défini indépendamment des objets de la classe, bien qu'accessible par tous ces objets comme un membre donnée classique.

La définition des différentes fonctions membres ne pose pas de difficultés particulières :

```
A::A(int v) : x(v) { cpt++; };

A::~A() { cpt--; };

int A::nb() { return cpt; };
```

Tout constructeur doit incrémenter le compteur, et le destructeur le décrémenter.

Notons que la fonction membre `nb` étant elle-même statique, sa syntaxe d'appel est celle d'un appel de fonction classique, i.e. sans passer par un objet receveur. Cette remarque prend tout son sens lors de l'exploitation de cette fonction membre (voir la fonction `main` donnée en exemple plus loin).

Nous retrouvons un code similaire pour la seconde classe :

```
class B {
  static int cpt;
  int x;
public:
  B(int=0);
  ~B();
public:
  static int nb();
};

int B::cpt = 0;

B::B(int v) : x(v) { cpt++; };

B::~B() { cpt--; };

int B::nb() { return cpt; };
```

Voici pour finir un exemple d'exploitation :

```
int main () {
  A o1(1);
  A o2(2);
  cout << A::nb() << ',' << B::nb() << endl;
  { A o3(3);
    B o4(4);
    cout << A::nb() << ',' << B::nb() << endl; };
  cout << A::nb() << ',' << B::nb() << endl;
};
```

et les messages affichés :

```
2,0
3,1
2,0
```

Exercice 4 : 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

Rappelons que le concept de *fonction virtuelle* en `C++` n'est pas encore connu lorsque cet exercice est posé ; ce dernier sera reproposé dans un TP ultérieur, sachant que ce concept aura alors été étudié en cours.
Considérons que tout animal qui s'exprime est identifié par son nom, s'il en porte un, soit par son type.

```
class unAnimal {
protected:
  char* nom;
public:
  unAnimal(char* ="un animal");
public:
  void sexprimer() const;
  char* sonNom() const;
};
```

Le constructeur se contente d'initialiser l'attribut adéquat :

```
unAnimal::unAnimal(char* n) : nom(n) {};
```

La méthode permettant à un animal de s'exprimer est appelée `sexprimer`. Le code de cette méthode est le même pour tous les animaux :

```
void unAnimal::sexprimer() const { cout << "un cri"; };
```

Cette méthode imprime un cir par défaut, commun à tous les animaux : *un cri*.
La méthode `sonNom()` retourne le nom de l'animal. Sa définition est :

```
char* unAnimal::sonNom() const { return nom; };
```

Un mamifère est un cas particulier d'animal :

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

Puisqu'un mamifère est un animal, l'initialiser en tant que mamifère implique de l'initialiser en tant qu'animal ; il nous faut donc définir un constructeur pour la classe `unMamifere` qui précise quoi faire pour initialiser l'objet en tant qu'animal (ici, il suffit de passer le nom au constructeur `unAnimal`), d'où sa définition :

```
unMamifere::unMamifere(char* n) : unAnimal(n) {};
```

Un chat est un cas particulier de mamifère dont le cri est parfaitement défini :

```
class unChat
: public unMamifere {
public:
  unChat(char* ="un chat");
public:
  void sexprimer() const;
};
```

avec :

```
unChat::unChat(char* n) : unMamifere(n) {};

void unChat::sexprimer() const { cout << "Miaou"; };
```

Les autres classes se définissent de la même façon :

```
class uneSouris
: public unMamifere {
public:
  uneSouris(char* ="une souris");
public:
  void sexprimer() const;
};


uneSouris::uneSouris(char* n) : unMamifere(n) {};

void uneSouris::sexprimer() const { cout << "Couic"; };
```

et :

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


unOiseau::unOiseau(char* n) : unAnimal(n) {};

class unCanari
: public unOiseau {
public:
  unCanari(char* ="un canari");
public:
  void sexprimer() const;
};


unCanari::unCanari(char* n) : unOiseau(n) {};

void unCanari::sexprimer() const { cout << "Cuicui"; };
```

Testons ce petit programme avec, entre autres, les animaux suggérés dans l'énoncé :

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

  cout << GrosMinet.sonNom() << " : ";
    GrosMinet.sexprimer(); cout << endl;
  cout << Tom.sonNom() << " : ";
    Tom.sexprimer(); cout << endl;
  cout << Jerry.sonNom() << " : ";
    Jerry.sexprimer(); cout << endl;
  cout << Titi.sonNom() << " : ";
```

```
    Titi.sexprimer(); cout << endl;

  uneSouris      s = uneSouris();
  unMamifere   Rox = unMamifere("Rox");

  cout << s.sonNom() << " : ";
    s.sexprimer(); cout << endl;
  cout << Rox.sonNom() << " : ";
    Rox.sexprimer(); cout << endl;

  cout << "----------" << endl;

  unAnimal* lesAnimaux[6] = { &GrosMinet, &Tom, &Jerry, &Titi, &s, &Rox };
  for (int i = 0; i<6; i++) {
    cout << lesAnimaux[i]->sonNom() << " : ";
    lesAnimaux[i]->sexprimer();
    cout << endl; }
};
```

À l'exécution, nous obtenons les messages suivants :

```
GrosMinet : Miaou
Tom : Miaou
Jerry : Couic
Titi : Cuicui
une souris : Couic
Rox : un cri
----------
GrosMinet : un cri
Tom : un cri
Jerry : un cri
Titi : un cri
une souris : un cri
Rox : un cri
```

Notons que, lorsqu'il est demandé à chaque animal de s'exprimer, les messages sont corrects ; par contre, lorsque tous les animaux sont pointés dans le tableau `lesAnimaux`, leur méthode d'expression est identique : c'est celle par défaut.

L'explication est la suivante : le tableau possède des composantes de type `unAnimal*` ; pour chacune des composantes, nous appelons la méthode `sexprimer`, et c'est donc celle de la classe `unAnimal` qui sera appelée, quelque soit le type réel de l'objet pointé.

Ce comportement n'est pas conforme à ce que nous sommes en droit d'attendre, i.e. la méthode d'expression devrait être celle de l'animal pointé ; nous verrons dans un prochain exercice que ce problème ne peut être résolu que par les fonctions membres virtuelles.