TD1 à 3 Résumé

M2103 Programmation Orientée Objets

Dept Info. IUT Bordeaux - 2015-2016

Table des matières

1 TD1 - Utiliser des objets, définir une classe 1

1.1 Étude d'un exemple SFML 1

1.1.1 Structure du code 1

1.1.2 Les objets utilisés 2

1.1.3 Actions sur un objet 2

1.2 Résumé 2

2 TD 1 - Exemple de classe : Totalisateur 2

2.1 Code client 2

2.2 Services attendus de l'objet 2

2.3 Etat interne de l'objet 3

2.4 Code `Totalisateur.h` 3

2.5 Code `Totalisateur.cpp` 3

2.6 Exercices 3

3 TD 2 - Programmer avec une classe 3

3.1 Rappel de la terminologie C++ 3

3.2 Étude de cas : une classe "Journal" . . 3

3.2.1 Exemple d'utilisation 3

3.2.2 Partie publique de l'entête 4

3.2.3 La variable "fichier de sortie" . . 4

3.2.4 Une fonction auxiliaire 4

3.2.5 La définition des fonctions 4

3.3 Étude du code "Grille" 4

4 TD 2 - Structure générale d'une classe 5

4.1 Constructeur par défaut 5

4.2 Constructeur de copie 5

4.3 Constructeurs délégués 5

4.4 Affectation 6

4.5 Le destructeur 6

5 TD3 - Ecriture d'une classe Robot 6

5.1 Enoncé 6

5.2 Complément : retour de référence. . . . 7

6 TD 3 - Classe Complexe 7

6.1 Étude du source 7

6.1.1 Ce qui est écrit 7

6.1.2 Ce qui est inutile d'écrire 7

6.2 Quelques ajouts 7

6.2.1 Accesseurs, mutateurs 7

6.2.2 Addition et autres opérations . . 7

6.3 Surcharge des opérations +, *, ... 8

6.3.1 Comme fonction membre 8

6.3.2 Comme fonction libre 8

6.3.3 Surcharge de operator<< 8

1 TD1 - Utiliser des objets, définir une classe

1.1 Étude d'un exemple SFML

Source étudié : `exemple-sfml-1.cc`.

1.1.1 Structure du code

```
19 - créer la fenêtre,
23   fixer la vitesse de réaffichage

25 - préparer des formes (carré, rond, texte)
50

54  boucle de travail tant que fenêtre ouverte
59  | traiter les événements reçus
    | depuis le dernier affichage
    |  |  selon évenement :
    |  |  - si Echappement ou demande de fermeture
    |  |       => fermer la fenêtre
    |  |  - si touche X
71  |  |       => échanger les positions du
```

```
    |  |         du rond et du carré
    |
86  | mise à jour de l'affichage
    | - effacement
90  | - positionnement et tracé des formes
99  | - affichage
```

1.1.2 Les objets utilisés

Les noms des types SFML sont préfixés par `sf`, qui est un **espace de noms** (*namespace* `std`).

Exemples :

```
sf::RenderWindow
sf::RectangleShape
sf::CircleShape
```

1.1.3 Actions sur un objet

Déclaration d'une variable La ligne 41

```
sf::text texte;
```

déclare une variable contenant un objet de type `sf::text` (les textes affichables), et demande (implicitement) son initialisation par le **constructeur par défaut** de la classe `sf::Text`

Vocabulaire :

- `sf::text` est ce qu'on appelle une **classe**
- `texte` est une **instance** de `sf::text`

après quoi vous avez constaté que son contenu était initialisé à vide. Alors que la déclaration d'un variable d'un type de base (booléens, réels, caractères et pointeurs)

```
int compteur;
```

n'initialise pas sa valeur : son contenu initial est aléatoire.

Remarque : on peut expliciter l'appel au constructeur par défaut

```
sf::text texte {};  // liste vide de paramètres
```

Fonctions membres Exemple :

```
texte.setString (L"Un carré et un cercle");
texte.setFont   (police);
texte.setColor  (sf::Color::Yellow);
```

Forme générale :

```
objet . nomDeFonction (param1, param2, ....);
```

`setString` est une **fonction membre** de la classe `sf::Text` (= une fonction qui peut être exécutée par les instances de cette classe).

Constructeur avec paramètres Exemple

```
sf::CircleShape rond { rayon_cercle };
```

Ancienne notation : avec des parenthèses au lieu des accolades.

1.2 Résumé

- classe,
- instance,
- fonction membre,
- constructeur par défaut
- constructeur avec paramètre

2 TD 1 - Exemple de classe : Totalisateur

On a besoin de faire des totaux sur des séries de nombre, des moyennes, des écarts-types etc. On définit une classe 'Totalisateur" qui s'en occupe

2.1 Code client

Le **code client** utilise la classe `Totalisateur`. Exemple :

```
Totalisateur t;
cout << "Tapez des nombres >= 0" << endl;
bool continuer = true;
while (continuer) {
    double x;
    cin >> x;
    if (x < 0) {
       continuer = false;
    } else {
       t.ajouter(x);
    }
}
cout << "moyenne = " << t.moyenne()
     << endl;
```

2.2 Services attendus de l'objet

- s'initialiser (constructeur)
- prendre en compte un nombre (ajouter)
- nous indiquer la moyenne

2.3 Etat interne de l'objet

L'objet contient

- un compteur de nombres
- un total

mis à jour à chaque appel de `ajouter`.

2.4 Code `Totalisateur.h`

```
#ifndef TOTALISATEUR_H
#define TOTALISATEUR_H

class Totalisateur
{
private:
    double m_somme;
    int m_nombre;
public:
    Totalisateur();
    void ajouter(float f);
    double moyenne()      const;
};
#endif
```

Note : on utilise systématiquement une variable de préprocesseur comme "garde" pour éviter l'effet des inclusions multiples,

2.5 Code `Totalisateur.cpp`

```
#include "Totalisateur.h"

Totalisateur::Totalisateur()
{
  m_somme = 0.0;
  m_nombre = 0;
  m_somme_carres = 0.0;
}

void Totalisateur::ajouter(float f)
{
  m_somme        += f;
  m_nombre       += 1;
}

double Totalisateur::moyenne()  const
{
    return m_somme / m_nombre;
}
```

2.6 Exercices

1. Ajouter des fonctions *accesseurs* qui retournent la valeur de la somme et le nombre d'éléments.
2. Doivent-elle être `const` ?
3. Pour pouvoir réutiliser le Totalisateur, il faut une fonction qui le remet à zéro.
4. Que faut-il ajouter pour que notre Totalisateur calcule aussi la variance ?

(variance = moyenne des carrés moins carré de la moyenne, d'après le théorême de Konig-Huygens).

3 TD 2 - Programmer avec une classe

3.1 Rappel de la terminologie C++

- Classe, instance (= objet)
- donnée membre (attribut, champ)
- fonction membre (méthode)
- constructeur

Attention : dans le jargon C++, les *membres* d'une classe sont les données et fonctions membres. Pas les instances.

Visibilité :

- membres publics
- membres privés (visibles seulements par les fonctions membres de la classe)

3.2 Étude de cas : une classe "Journal"

3.2.1 Exemple d'utilisation

```
# include "Journal.h"

int main(int argc, char **argv)
{
  Logfile f("/tmp/log.txt");
  // ..
  f.log("C'est parti");
  //...
  f.log("et voila");
  return 0;
}
```

Fichier `/tmp/log.txt` qui en résulte :

```
* OPENED AT 12:34:56
12:34:57 C'est parti
12:35:12 et voila
* CLOSED AT 12:35:12
```

3.2.2 Partie publique de l'entête

1. écrire la partie publique de la casse `Journal`

```
class Journal
{
public:
   // constructeur
   Journal  (const std::string
                   & filename);

   // fonction membre
   void log (const std::string message);
   ...
```

2. le premier et le dernier message sont produits respectivement :

— par le **constructeur** appelé lors de la déclaration de `f`
— par le **destructeur** qui s'exécute quand la variable `f` cesse d'exister (au moment du `return`).

```
class Journal
{
public:
  Journal  (const std::string
                      & filename);
  void log (const std::string message);

  // destructeur
  ~Journal ();
  ...
```

3.2.3 La variable "fichier de sortie"

Le fichier de sortie est une **donnée membre privée**

```
class Journal
{
private:
    std::ofstream m_out;
    ...
```

Il est ouvert par le constructeur, de préférence dans la **liste d'initialisation** :

```
Journal::Journal( const std::string
                               & filename )
   : m_out { filename }
{
   m_out << "* DEBUT " <<...
}
```

3.2.4 Une fonction auxiliaire

On a besoin, dans le constructeur, dans la fonction `log`, et dans le destructeur, de faire écrire l'heure courante.

On confie ce travail à une fonction *privée* `printCurrentTime`, appelée à ces trois endroits.

La déclaration de la classe serait donc :

```
class Journal
{
public:
    Logfile(const std::string &filename);
    void log(const std::string &message);
    ~Logfile();

private:
    std::ofstream m_out;
    void          printCurrentTime();

};
```

3.2.5 La définition des fonctions

laissée en exercice.

3.3 Étude du code "Grille"

Feuille `td2-exemple-grille.pdf`

La notion de classe permet de mieux organiser le code. Le programme principal crée une instance d'`Appli` et lui demande de tourner.

```
Appli a;
a.run();
```

Dans la classe `Appli` Le code est découpé en fonctions, les données membres sont les variables partagées

— la fenêtre, les formes pour le dessin, ...
— les données du "modèle" : le tableau de booléens
— les données de l'interface : les coordonnées de la case courante.

Ainsi le code est beaucoup plus clair, avec des responsabilités bien définies

— le constructeur crée et initialise les éléments
— la fonction `run` lance la boucle de l'application : traitement des événements reçus, et affichage
— la fonction `traiter_evenement` exécute les actions sur le modèle et l'interface (changer de case, changer son contenu)
— la fonction affichage traduit la situation en dessin.

Pour des programmes plus élaborés, nous pousserons plus loin la distinction entre interface et modèle, en les représentant par des objets distincts.

4 TD 2 - Structure générale d'une classe

Prenons par exemple la classe `A` déclarée ainsi, à partir de deux types `X`, `Y` existants

```
class A
{
   X x;
   Y y;
};
```

Le compilateur génère, sous certaines conditions, des constructeurs, destructeurs etc. dit implicites.

4.1 Constructeur par défaut

La terminologie est trompeuse, le **constructeur par défaut** est celui qui n'a pas de paramètres. Il est appelé au moment de la déclaration des variables, exemples :

```
A a;       // constructeur par défaut
A b{};     // idem, notation c++11
A t[10];   // appelé 10 fois
A *p = new A{};
```

Il initialise les données membres, dans l'ordre de déclaration, avec leurs constructeurs par défaut respectifs

Si on l'explicitait, on le déclarerait ainsi

```
class A {

public:
 A();     // déclaration
};
```

et il serait défini par

```
A::A()
  : x {}
  , y {}
   {
   // corps vide
   }
}
```

A savoir : si d'autres constructeurs sont déclarés, ce constructeur n'est pas généré automatiquement. On peut insister pour l'avoir

```
class A {
   A() = default;
};
```

4.2 Constructeur de copie

Initialise un **nouvel** objet à partir d'un objet existant. Appelé lors d'une déclaration avec initialisation, et quand une fonction reçoit un **paramètre par valeur**, par exemple :

```
void f(A p) {
...
}

int main() {
   A a;
   A b{a}  // 1
   f(a);   // 2
}
```

Le constructeur de copie implicite initialise les champs de `b` (ou `p`) à partir de ceux de `a`.

Redéfinition : Il reçoit en paramètre une référence (souvent constante) à une instance du même type.

```
class A
{
  A (const A & a);      // déclaration
};

A::A (const A & a)      // définition
   : x {a.x}
   , y {a.y}
{
  // corps vide
}
```

A savoir : il y a 3 notations équivalentes pour la déclaration d'une variable avec inititalisation

```
              // notation
A a2 {a1};    // uniforme moderne C++11
A a2 (a1);    // ancienne C++
A a2 = a1;    // historique compatible avec C
```

Malgré les apparences, la dernière n'est pas une affectation.

4.3 Constructeurs délégués

Si il y a plusieurs constructeurs, l'un pourra appeler à l'autre (notion de délégation). Exemple :

```
class Position
{
    ....
    Position();                 // déclaration
    Position(int x, int y);
};
```

```
Position::Position(int x, int y)
  : m_x{x}
  , m_y{y}
{}

Position::Position()  // constructeur par défaut
  : Position{0,0}       // délégation
{}
```

4.4 Affectation

L'affectation a lieu entre deux variables *existantes*. Elle est générée implicitement.

```
  A a1, a2;
  ...
  a1 = a2;
  a1.operator=(a2);  // notation équivalente
```

L'affectation implicite, générée par le compilateur, affecte les données membres, une par une.

Si on veut un autre comportement que le comportement par défaut, on redéfinit la fonction membre `operator =` :

```
class A
{
  A &  operator=  (const A & b);  // décl.
};

A &  A::operator=  (const A & b); // déf.
{
  x = b.x;
  y = b.y;
  return *this;
}
```

l'affectation retourne une référence à l'objet destinataire (`this` est un pointeur sur cet objet) ce qui permet le chainage d'affectations (a = b = c).

Précaution

Dans certaines circonstances (on verra plus tard) on peut avoir des problèmes en cas d'auto-affectation (`a = a`).

Une précaution classique (totalement inutile ici) est de tester que le paramètre n'est pas l'objet lui-même, en comparant leurs adresses.

```
Thing & Thing::operator= (const Thing & other)
{
  if (this == &other)
     return;       // rien à faire
  ...
}
```

Une meilleure conception du code rend généralement cette précaution inutile.

4.5 Le destructeur

Le destructeur de la classe `A` est appelé

- A la fin du bloc où une variable automatique de classe `A` a été déclarée.
- lors d'une libération par `delete` d'une variable allouée dynamiquement.

```
voir foo()
{
  A a1;
  A a2;

  if (x == 2) {
    A  a3;
    ....
  }            // destruction de a3
  ...
}              // destruction de a2, puis a1
```

Pour expliciter le destructeur on écrit

```
class A
{
    ~A();       // déclaration
};

A::~A() {      // définition
   // corps vide
}
```

L'appel aux destructeurs de `y` et `x` (ordre inverse de déclaration) est fait automatiquement.

5 TD3 - Ecriture d'une classe Robot

5.1 Enoncé

Ecrire une classe `Robot` qui permette d'exécuter le code suivant

```
#include <iostream>
#include "Robot.h"

using namespace std;

int main()
{
    Robot r (100,200, Robot::NORTH);
    std::cout << "départ  : "
              << r.state()  // (100 200 N)
              << std::endl;
    r.move(50);
    r.turnLeft();
```

```
    r.move(10);
    r.turnRight();
    r.move(20);
    r.turnRight();
    std::cout << "arrivée : "
              << r.state()  // (90 45 E)
              << std::endl;
    return 0;
}
```

On ajoutera quelques accesseurs et mutateurs, comme `getX()`, `setDirection(int)`, etc.

Indications

- le robot est censé se déplacer dans une grille 2D à coordonnées entières.
- conventions classiques : les x croissent vers l'est, le y vers le nord.
- 4 directions : `NORTH WEST SOUTH EAST`.
- l'accesseur `getDirection()` retournera toujours un entier de 0 (=NORTH) à 3 (=EAST).
- le mutateur `setDirection(int)` acceptera n'importe quel entier, qui sera normalisé modulo 4. Par exemple, après un `setDirection(443)`, un appel à `getDirection()` retournera 3.
- fabrication d'une chaine de caractères : la fonction `Robot::state()` retourne une chaîne qui décrit l'état du robot. Utiliser `to_string`

```
return "(" + std::to_string(m_x)
     + " " + std::to_string(m_y)
     + ...;
```

- Dans la fonction `Robot::state()`, il faut produire une chaine qui indique la direction. Utiliser un tableau de chaines constantes :

```
std::string Robot::state() const {

static const std::string noms
          { "N", "W", "S", "E" };
   ....

   .... noms[direction]....
```

5.2 Complément : retour de référence.

Pour pouvoir chainer les opérations :

```
r .move(50) .turnLeft() .move(10) .turnRight()
  .move(20) .turnRight();
```

les opérations retournent une référence à l'objet.

```
class Robot {
...
   Robot &  turnLeft();
   ...
};
```

```
Robot & Robot::turnLeft()
{
   m_dir = (m_dir+1) % 4;
   return *this;
}
```

6 TD 3 - Classe Complexe

6.1 Étude du source

(voir feuille distribuée)

6.1.1 Ce qui est écrit

1. Représentation de nombres complexes
2. Parties réelles et imaginaires en données privées
3. Deux constructeurs : par défaut et avec paramètres. Le const. par défaut fait appel à l'autre (**constructeur délégué**).
4. Une fonction pour construire une chaine de caractères représentant le nombre. Pourrait être améliorée (quand partie réelle ou imaginaire nulle, ou partie imaginaire négative).
5. Comparaison de deux complexes.

6.1.2 Ce qui est inutile d'écrire

- Constructeur par copie
- Affectation
- Destructeur

Les versions implicites suffisent.

6.2 Quelques ajouts

6.2.1 Accesseurs, mutateurs

- ajouter accesseurs et mutateurs pour les parties réelles et imaginaires.

6.2.2 Addition et autres opérations

L'addition de nombres complexes, en tant qu'objets, peut être comprise de deux façons assez différentes.

1. soit un Complexe reçoit l'ordre de s'augmenter, avec un paramètre disant de combien. Et il sera modifié.
2. soit il reçoit l'ordre de fournir le résultat de la somme de sa valeur et de celle d'un autre complexe. Ca ne le modifie pas.

```
Complexe c1, c2, c3;

c1.ajouter(c2);     // modifie c1
c3 = c1.plus(c2);   // ne modifie pas c1
```

Analogies - Dans le premier cas, c'est comme le robot, dont l'état est modifié par l'action move(). - Dans le second cas, quand on demande à 2 ce que vaut sa somme avec 3, ça renvoie 5, mais ne modifie pas 2.

Il y a donc deux fonctions très différentes, que nous appelons respectivement `ajouter` (choix d'un verbe pour exprimer une action, procédurale) et `somme` (pour une forme fonctionnelle, parce que ça retourne une somme).

```
class Complexe {
...
   void     ajouter(const Complexe & autre);
   Complexe somme  (const Complexe & autre) const;
..
};
```

Exercice : écrire ces deux fonctions membres.

Exercice : améliorer la définition d'`ajouter` pour qu'on puisse faire

```
c1.ajouter(c2).ajouter(c3);
```

Exercice : Même chose pour la soustraction, multiplication etc.

6.3 Surcharge des opérations +, *, ...

La notation infixe des opérateurs binaires +, , ... *est une* commodité d'écriture* pour l'appel de fonctions qui se nomment `operator+`, `operator*`, etc.

Quand on écrit

```
Complexe c1, c2, c3;
c1 = c2 + c3;
```

la somme `c2+c3` peut correspondre à deux choses

- soit l'appel d'une **fonction membre** `Complexe::operator+` à un paramètre,
- soit l'appel d'une **fonction libre** `operator+` à deux paramètres.

```
c1 = c2.operator+(c3);
c1 = operator+(c2,c3);
```

6.3.1 Comme fonction membre

Si on choisit de représenter l'addition par une fonction membre, on ajoute dans Complexe :

```
class Complexe {

Complexe operator+
     (const Complexe & autre) const;
};
```

Avec le même code que `plus`, seul le nom a changé en `operator+`.

6.3.2 Comme fonction libre

Une fonction libre sera par, par définition, déclarée **hors** de la classe (mais souvent dans le fichier .h de celle-ci)

```
Complexe operator+ (const Complexe & c1,
                    const Complexe & c2);
```

6.3.3 Surcharge de operator<<

De même ce fichier est le bon endroit pour déclarer la fonction libre

```
std::ostream &
    operator<< (std::ostream & o,
                const Complexe & c);
```

qui permet d'afficher un Complexe sur `cout`

```
cout << "les racines sont " << r1
     << " et " << r2 << endl;
```

Elle est définie ainsi

```
ostream &  operator<< (ostream & o,
           const Complexe & c)
{
  o << c.toString();
  return o;
}
```