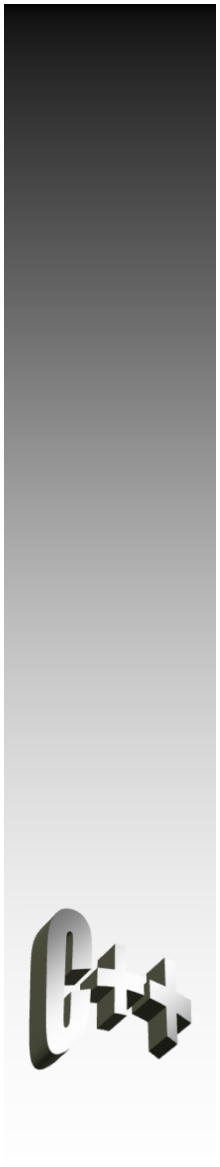

Chapitre 6
Classes et abstraction de données

Auteur : Marco Lavoie

Adaptation : Sébastien Bois

Langage C++

25908 IFM

Objectifs

- Comprendre l'encapsulation
  - Masquage de données
  - Abstraction de données
- Créer des classes
  - Instancier, exploiter et détruire des objets de classes
  - Contrôler l'accès aux membres
- Apprécier les avantages la *Programmation Orientée Objets* (POO)

Aperçu

- Définitions de structures
- Accès aux membres de structures
- Les classes : types de données abstraits
- Portée de classes et accès aux membres
  - Contrôle d'accès aux membres
- Séparation de l'interface et de l'implantation
  - Fonctions d'accès et fonctions utilitaires
- Constructeurs et destructeurs
- Affectation via la copie par défaut
- Représentation UML des classes

Introduction

- Concepts de POO vus dans ce chapitre
  - Encapsulation dans une classe
    - des données (attributs)
    - des fonctions (comportements)
  - Séparation de
    - l'interface : permet de communiquer avec l'objet
    - l'implantation : masquage de ses fonctionnalités
  - Instanciation d'objets
- La classe (`class`) en C++ est une évolution de la structure (`struct`) en C

Définitions de structures

- Concept provenant du langage C
- Permettent de regrouper des données
- Exemple :

```
struct Temps {
  int heure;     // 0-23
  int minute;    // 0-59
  int seconde;   // 0-59
};
```

- L'instruction `struct` définit un nouveau type
  - Dans l'exemple ci-dessus, le type `Temps`
  - Aucun espace mémoire associé à la structure

Accès aux membres

- L'*opérateur point* (**.**) permet d'accéder aux membres d'un structure
- L'opérateur flèche (**->**) donne accès aux membres via un pointeur
  - Alternativement, on peut exploiter l'opérateur *****
    - Attention : l'opérateur . a priorité sur l'opérateur *

```
Temps tmp;

tmp.heure   = 22;
tmp.minute  = 47;
tmp.seconde =  3;

Temps *pTmp;

pTmp = &tmp;
pTmp->heure   = 22;
pTmp->minute  = 47;
pTmp->seconde =  3;

cout << tmp.heure
     << pTmp->heure
     << (*pTmp).heure;
```

Fonctionnalités pour structures

- Des fonctions sont requises pour "encapsuler" les fonctionnalités d'une structures

```
// Afficher temps en format hh:mm:ss
void afficherMilitaire( const Temps &t ) {
  cout << ( t.heure   < 10 ? "0" : "" ) << t.heure   << ":"
       << ( t.minute  < 10 ? "0" : "" ) << t.minute  << ":"
       << ( t.seconde < 10 ? "0" : "" ) << t.seconde;
}

// Afficher temps en format hh:mm:ss am/pm
void afficherStandard( const Temps &t ) {
  cout << ( ( t.heure == 0 || t.heure == 12 ) ? 12 : t.heure % 12 )
       << ":" << ( t.minute  < 10 ? "0" : "" ) << t.minute
       << ":" << ( t.seconde < 10 ? "0" : "" ) << t.seconde
       << " " << ( t.heure < 12 ? "am" : "pm" );
}
```

Fonctionnalités (suite)


```
// Lecture du temps en format hh:mm:ss
void lireMilitaire( Temps &t ) {
  char sep;
  cin >> t.heure >> sep >> t.minute >> sep >> t.seconde;
}

// Programme de manipulation de la structure temps
int main() {
  Temps tempsSouper;

  cout << "Entrez l'heure du souper en format militaire: ";
  lireMilitaire( tempsSouper );

  cout << "\nHeure du souper en format militaire: ";
  afficherMilitaire( tempsSouper );

  cout << "\nHeure du souper en format standard: ";
  afficherStandard( tempsSouper );

  return 0;
}
```

Fonctionnalités (suite)

- Exécution de l'exemple :

```
C:\>temps.exe
Entrez l'heure du souper en format militaire: 18:30:17
Heure du souper en format militaire: 18:30:17
Heure du souper en format standard: 6:30:17 pm
C:\>
```

- Remarques sur l'exemple précédent
  - L'exploitation de la structure `Temps` se fait via les trois fonctions : `lireMilitaire()`, `afficherMilitaire()` et `afficherStandard()`
  - C'est le principe de masquage : le programmeur n'a pas à connaître les attributs de la structure
    - Synonyme de masquage : **encapsulation**

Encapsulation dans une classe

- Contrairement à la structure (`struct`) en C qui ne peut contenir que des attributs membres, la classe (`class`) peut aussi contenir des fonctions membres
  - Si on exploite une classe dans l'exemple précédent, les trois fonctions peuvent être encapsulées dans la classe
- En POO, les fonctions membres de classes sont aussi appelées *méthodes*

Encapsulation dans une classe (suite)

- La classe permet, entre autres, de
  - Définir les *attributs membres*
  - Définir les *fonctions membres*
  - Contrôler l'accessibilité aux membres via les *identificateurs d'accessibilité*
  - Gérer la création d'instances via les *constructeurs*
  - Gérer la destruction d'instances via les *destructeurs*

Classe **Temps**

- Déclaration

Identificateurs d'accès aux membres

Fonctions membres

Attributs membres

```
class Temps {
public:
   Temps();          // constructeur

   // Prototypes de fonctions membres
   void ajusterTemps( int, int, int );
   void lireMilitaire();
   void afficherMilitaire();
   void afficherStandard();

private:
   // Attributs membres
   int heure;     // 0-23
   int minute;    // 0-59
   int seconde;   // 0-59
};
```

Identificateurs d'accessibilité

- Contrôlent l'accès aux membres
  - **`private`** : membres accessibles aux fonctions membres de la classe uniquement (encapsulation)
  - **`public`** : membres accessibles à tous (interface)
- Dans la classe **`Temps`**
  - Le constructeur et les fonctions membres sont accessibles à tous
  - Les attributs membres sont uniquement accessibles aux membres de la classe (le constructeur et les trois fonctions)

Fonctions membres

- Seuls leurs prototypes sont énumérés dans le bloc `class`
  - Les fonctions complètes sont définies plus loin dans le code, avec l'opérateur de portée (`::`)

```
// Afficher temps en format hh:mm:ss
void Temps::afficherMilitaire() {
  cout << ( heure   < 10 ? "0" : "" ) << heure   << ":"
       << ( minute  < 10 ? "0" : "" ) << minute  << ":"
       << ( seconde < 10 ? "0" : "" ) << seconde;
}

// Afficher temps en format hh:mm:ss am/pm
void Temps::afficherStandard() {
  cout << ( ( heure == 0 || heure == 12 ) ? 12 : heure % 12 )
       << ":" << ( minute  < 10 ? "0" : "" ) << minute
       << ":" << ( seconde < 10 ? "0" : "" ) << seconde
       << " " << ( heure < 12 ? " m" : "pm" );
}
```

Fonctions membres (suite)

- Suite de l'exemple

```
// Lecture du temps en format hh:mm:ss
void Temps::lireMilitaire() {
  char sep;
  cin >> heure >> sep >> minute >> sep >> seconde;
}
```

- À noter les attributs membres exploités dans les fonctions membres

```
void lireMilitaire( [redacted] ) {
  char sep;
  cin >> [redacted]heure >> sep >> [redacted]minute >> sep >> [redacted]seconde;
}
```

  – La fonction s'exécute pour une instance

Instanciation d'une classe

- Instancier = créer un objet d'une classe
  - L'opérateur point (**.**) permet d'invoquer un membre de la classe pour une instance

```
// Programme de manipulation de la structure temps
int main() {
  Temps tempsSouper;   // Instanciation

  cout << "Entrez l'heure du souper en format militaire: ";
  tempsSouper.lireMilitaire();

  cout << "\nHeure du souper en format militaire: ";
  tempsSouper.afficherMilitaire();

  cout << "\nHeure du souper en format standard: ";
  tempsSouper.afficherStandard();

  return 0;
}
```

Constructeurs

- Pour déclarer un constructeur membre
  - Fonction publique ayant le même nom que la classe
  - Aucune valeur de retour (pas même `void`)
- Un constructeur sert généralement à initialiser l'instance
  - Constructeur par défaut : constructeur sans paramètres

```
// Constructeur par défaut
Temps::Temps() {
  heure   = 0;
  minute  = 0;
  seconde = 0;
}
```

Constructeurs (suite)

- Essentiels pour initialiser les attributs membres car il n'est pas possible d'initialiser ceux-ci dans la définition de la classe
  - Puisque aucune mémoire est associée à la définition de classe, il est impossible d'initialiser les attributs directement dans la définition

```
class Temps {
public:
  Temps();

  void ajusterTemps( int, int, int );
  void lireMilitaire();
  void afficherMilitaire();
  void afficherStandard();

private:
  int heure   ;
  int minute  ;
  int seconde ;
};
```

Accessibilité des membres

- Les membres privés sont uniquement accessibles aux fonctions membres de la classe
  - On doit au besoin créer des fonctions membres publiques faisant le "pont" entre les membres privés et les exploitants

```
// Ajuster les attributs membres
void Temps::ajusterTemps( int h, int m, int s ) {
  heure   = h;
  minute  = m;
  seconde = s;
}

void main() {
  Temps tempsSouper;

  // Ajuster à 16:30:17
  tempsSouper.ajusterTemps( 16, 30, 17 );
```

Opérateurs . versus –>

- Lorsque l'instance est accédée via une variable pointeur, il faut exploiter l'*opérateur flèche*
  - Alternativement, on peut utiliser **`(*).`**
    - Parenthèses requises car l'opérateur point (`.`) a priorité sur l'opérateur de déréférencement (**`*`**)
  - Il faut privilégier l'opérateur flèche (**`–>`**) car il améliore la lisibilité du code

```
Temps tempsSouper;

// Ajuster à 16:30:17
tempsSouper.ajusterTemps( 16, 30, 17 );

Temps *ptr = &tempsSouper;
ptr->ajusterTemps( 4, 57, 33 );

(*ptr).ajusterTemps( 8, 2, 45 );
```

Priorité des opérateurs

- Incluant tous les opérateurs vus à date, en ordre décroissant de priorité

Opérateurs	Associativité
`() [] . ->`	De gauche à droite
`static_cast<type>() ++ -- (versions suffixe)`	De droite à gauche
`++ -- + - (versions préfixe) ! &`	De gauche à droite
`*` (*déréférencement*)	De gauche à droite
`/ % *` (*multiplication*)	De gauche à droite
`+ -`	De gauche à droite
`<< >>`	De gauche à droite
`< <= > >=`	De gauche à droite
`== !=`	De gauche à droite
`&&`	De gauche à droite
`||`	De gauche à droite
`?:`	De droite à gauche
`= += -+ *= /= %=`	De droite à gauche

Séparation de l'interface et l'implantation

- Séparer la déclaration de la classe et la définition des fonctions membres
  - On verra plus tard comment distribuer la déclaration (i.e. *interface*) sans distribuer les définitions (i.e. *implantation*)
- Fichiers séparés
  - La déclaration de la classe stockée dans un fichier en-tête (ex: `temps.h`)
  - Les définitions de fonctions membres stockées dans un fichier CPP (ex: `temps.cpp`)

Séparation : fichier en-tête

- Contenu de **`temps.h`** :

```
// Empêcher les inclusions multiples
#ifndef TEMPS_H   // Est-ce que la variable de précompilation existe?
#define TEMPS_H   // Sinon, définir la variable de précompilation

// Définition de la classe
class Temps {
public:
  Temps();        // constructeur
  void ajusterTemps( int, int, int );  // ajuste heure, minute, seconde
  void lireMilitaire();                // lire selon le format hh:mm:ss
  void afficherMilitaire();            // affiche en format hh:mm:ss
  void afficherStandard();             // affiche en format hh:mm:ss am/pm
private:
  int heure;     // 0-23
  int minute;    // 0-59
  int seconde;   // 0-59
};

#endif   // Fin du bloc de précompilation #ifndef précécent
```

Séparation : fichier source

- Contenu (incomplet) de **`temps.cpp`** :

```
#include <iostream>
using std::cout;

// Requiert la classe Temps
#include "temps.h"

// Constructeur par défaut
Temps::Temps() {
  heure = minute = seconde = 0;
}

// Ajuster les attributs membres
void Temps::ajusterTemps( int h, int m, int s ) {
  heure   = ( h < 0 ? 0 : (h > 23 ? 23 : h ) );
  minute  = ( m < 0 ? 0 : (m > 59 ? 59 : m ) );
  seconde = ( s < 0 ? 0 : (s > 59 ? 59 : s ) );
}

// Insérer les autres fonctions membres ici, mais PAS LE MAIN
```

Séparation : programme principal

- Dans un fichier distinct (ex: **`main.cpp`**) :

```
#include <iostream>
using std::cout;
using std::endl;

// Requiert la classe Temps
#include "temps.h"

// Programme de manipulation de la structure temps
int main() {
  Temps tempsSouper;   // Instanciation

  cout << "Entrez l'heure du souper en format militaire: ";
  tempsSouper.lireMilitaire();

  cout << "\nHeure du souper en format militaire: ";
  tempsSouper.afficherMilitaire();
}
```

Séparation : pourquoi `#ifndef` ?

- Un fichier en-tête (ex: `test.h`) inclus toujours son contenu dans un bloc **`#ifndef`**/**`#define`**/**`#endif`** afin que celui-ci ne soit pas compilé plus d'une fois
  - **`#ifndef TEST_H`** : la variable de précompilation **`TEST_H`** est-elle non définie?
    - Si c'est le cas, alors le code qui suit est compilé jusqu'au **`#endif`**
    - **`#define TEST_H`** définie la variable de sorte que si jamais le fichier `test.h` est compilé plus d'une fois par le compilateur, les compilations subséquentes ignoreront ce code

```
#ifndef TEST_H
#define TEST_H

class Temps {
  ...
};

#endif
```

Séparation : pourquoi `#ifndef` ?

- Et pourquoi un fichier en-tête serait-il compilé plus d'une fois ?
  - Si le code source du projet est répartit dans plusieurs fichiers et que plusieurs d'entre eux incluent le fichier en-tête
  - Que signifie `#include "fichier.h"` ?
    - Équivalent à du copier-coller : insérer le contenu du fichier `ficher.h` à la place de la ligne `#include`
    - Le bloc `#ifndef`/`#define`/`#endif` évite l'erreur de compilation : déclarations multiples d'une même classe

Séparation : en résumé

- Séparer le code source d'une classe (par exemple **`Dossier`**) en deux fichiers
  - **`dossier.h`** : contient la déclaration de la classe dans un bloc `#ifndef`/`#define`/`#endif`
  - **`dossier.cpp`** : code source des fonctions membres, précédé de `#include "dossier.h"`

```
#ifndef DOSSIER_H
#define DOSSIER_H

class Dossier {
  ...
};

#endif
```

- Notez la distinction des fichiers d'en-tête du compilateur
  - `#include <iostream>` versus `#include "dossier.h"`

Fichier localisé dans un répertoire du compilateur

Fichier localisé dans le répertoire du projet

Séparation : en résumé (suite)

- Stocker le programme principal dans un fichier distinct (ex: **`main.cpp`**)
- Notre projet consiste donc à trois fichiers à compiler
  - `dossier.h`, `dossier.cpp` et `main.cpp`
- Le compilateur compile séparément ces trois fichiers puis les regroupe en un seul exécutable : **`main.exe`**

Exercice #6.1

- Solutionnez l'exercice distribué par l'instructeur
  - Séparez le code source dans trois fichiers
    - **`date.h`** : déclaration de la classe `Date`
    - **`date.cpp`** : définition des fonctions membres de la classe `Date`
    - **`Exercice_6_1.cpp`** : programme principal
- N'oubliez pas les conventions d'écriture
- Soumettez votre projet selon les indications de l'instructeur

Contrôle d'accès aux membres

- Rappel : *identificateurs d'accessibilité*
  - **`private`** : membres accessibles aux fonctions membres de la classe uniquement (encapsulation)
  - **`public`** : membres accessibles à tous (interface)

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

int main() {
  Temps t;    // Instanciation

  // Erreur : Temps::heure est privé
  t.heure = 7;

  // Pas d'erreur Temps::afficherMilitaire() est publique
  t.afficherMilitaire();
}
```

Le compilateur indique l'erreur lors de la compilation

Contrôle d'accès aux membres

- Par défaut, l'accessibilité des membres est `private`

Considérées privées

```
class Temps {
  int heure;
  int minute;
  int seconde;
public:
  Temps();
  void ajusterTemps( int, int, int );
  void lireMilitaire();
  void afficherMilitaire();
  void afficherStandard();
};
```

`private:`

- Il est toujours préférable d'indiquer explicitement l'accessibilité
  - Facilite la compréhension du code source

Fonctions membres utilitaires

- Généralement privées, elles servent à faciliter l'implantation d'autres fonctions

```
// Fonction utilitaire (privée)
bool Temps::validerTemps() {
  return ( heure   >= 0 && heure   <= 23 ) &&
         ( minute  >= 0 && minute  <= 59 ) &&
         ( seconde >= 0 && seconde <= 59 );
}

// Lecture du temps en format hh:mm:ss (publique)
void Temps::lireMilitaire() {
  char sep;
  do
    cin >> heure >> sep >> minute >> sep >> seconde;
  while ( ! validerTemps() );
}
```

Fonctions membres d'accès

- Généralement pubiques, elles servent à donner accès aux attributs privés

```
// Fonction d'accès en lecture à l'attribut heure (publique)
int Temps::getHeure() {
  return ( heure );
}

// Fonction d'accès en écriture à l'attribut heure (publique)
int Temps::setHeure( const int h ) {
  if ( h < 0 )
    heure = 0;
  else if ( h > 23 )
    heure = 23
  else
    heure = h;

  return heure;
}
```

Constructeurs

- Servent à initialiser les instances d'une classe
  - Une classe peut disposer de plusieurs constructeurs
    - En autant qu'ils aient chacun une signature distincte
- Un constructeur est une fonction membre dont
  - Le nom est le même que celui de la classe
  - Aucun type de valeur de retour n'est spécifié

Constructeurs par défaut

- Aucun paramètre
- Celui exécuté lorsqu'une instance est créée sans arguments

```
class Temps {
public:
  Temps();  // constructeur par défaut
  ...
};

Temps::Temps() {
  heure = minute = seconde = 0;
  std::cout << "Constructeur par défaut\n";
}

int main() {
  Temps t;   // Instanciation
  ...
}
```

Aucun type de valeur de retour

Aucun paramètre

```
C:\>temps2.exe
Constructeur par défaut
C:\>
```

Constructeurs paramétrés

- Une classe peut disposer de constructeurs avec paramètres
  - En plus du constructeur par défaut

```
class Temps {
public:
  Temps();  // constructeur par défaut
  Temps( const int, const int, const int );
  ...
};

Temps::Temps( const int h, const int m, const int s ) {
  ajusterTemps ( h, m, s );
  std::cout << "Constructeur paramétré\n";
}

int main() {
  Temps t1, t2( 3, 47, 19 );   // Instanciations
  ...
}
```

```
C:\>temps3.exe
Constructeur par défaut
Constructeur paramétré
C:\>
```

Deux instances créées

Constructeurs paramétrés (suite)

- Peuvent fournir des arguments par défaut

```
class Temps {
public:
  Temps( const int = 0, const int = 0, const int = 0 );
  ...
};

int main() {
  Temps t1,                  // h = 0, m = 0 et s = 0
        t2( 3 ),             // h = 3, m = 0 et s = 0
        t3( 3, 47 ),         // h = 3, m = 47 et s = 0
        t4( 3, 47, 19 );     // h = 3, m = 47 et s = 19
  ...
}
```

- Dans l'exemple ci-dessus, le constructeur par défaut est celui paramétré
  - Car il est invocable sans argument

Destructeur

- Alter ego du constructeur par défaut
  - Le destructeur est automatiquement invoqué à la destruction d'une instance de la classe
- Même format que le constructeur par défaut, mais son nom est précédé d'un tilde (**~**)
  - Aucun paramètre permis
- La restriction ci-dessus (i.e. *aucun paramètre*) fait en sorte qu'une classe ne peut disposer que d'un seul destructeur

Destructeur (suite)

- Exemple

```
class Temps {
public:
  Temps( const int = 0, const int = 0, const int = 0 );
  ~Temps();
  ...
};

Temps::Temps( const int h, const int m, const int s ) {
  ajusterTemps ( h, m, s );
  std::cout << "Constructeur invoqué\n";
}

Temps::~Temps() {
  std::cout << "Destructeur invoqué\n";
}

void main() {
  Temps t;    // Création de t
}             // Destruction de t
```

```
C:\>temps4.exe
Constructeur invoqué
Destructeur invoqué
C:\>
```

Destructeur (suite)

- Le destructeur est automatiquement invoqué lorsque l'instance est détruite (selon la portée de l'instance)
  - Le programmeur ne peut pas explicitement l'invoquer

```
void main() {
  Temps t( 3, 47, 19 );

  t.~Temps();  // ERREUR   ✘
}
```

  - Réciproquement, on ne peut pas invoquer un constructeur après la création de l'instance

```
void main() {
  Temps t( 3, 47, 19 );

  t.Temps( 14, 56, 4 );  // ERREUR   ✘
}
```

Fonctions accesseurs et mutateurs

- En anglais : *getters* et *setters*
- Catégories de fonctions membres contrôlant l'accès aux attributs membres privés
  - Certains langages intègrent le principe des *getters* et *setters* dans leur syntaxe (p.ex. C# et VB : **`property`**), mais pas le C++
  - Le principe peut cependant être émulé en C++
    - Ça constitue une bonne pratique de programmation

Fonctions accesseurs

- Fonctions membres donnant accès à un attribut privé, mais ne permettant pas d'en modifier la valeur
  - Ces fonctions publiques retournent la valeur stockée dans l'attribut privé correspondant

```
class Temps {
  int heure;
  int minute;
  int seconde;

public:
  Temps(int = 0, int = 0, int = 0);

  int getHeure() { return heure; }
  int getMinute() { return minute; }
  int getSeconde() { return seconde; }
};
```

Fonctions mutateurs

- Fonctions membres permettant de modifier la valeur d'un attribut membre privé
  - Effectue généralement de la validation

```
class Temps {
  int heure;
  ...

public:
  Temps(int = 0, int = 0, int = 0);
  ...

  void setHeure( int valeur ) {
    if ( valeur < 0 )
      heure = 0;
    else if ( valeur > 23 )
      heure = 23;
    else
      heure = valeur;
  }
};
```

Opérateur d'affectation par défaut

- Par défaut, une classe dispose d'un opérateur d'affectation (=)
  - Effectue une copie de membre à membre des attributs

```
void main() {
  Temps t1( 3, 47, 19 );
  Temps t2;

  t2 = t1;
  std::cout << "t2 = ";
  t2.afficherStandard();
}
```

```
C:\>temps5.exe
t2 = 03:47:19 am
C:\>
```

  - Nous verrons plus tard comment surcharger cet opérateur

Exercice #6.2

- Solutionnez l'exercice distribué par l'instructeur
  - Poursuivez le travail sur le projet console *Visual Studio C++* créé lors de l'exercice 6.1
  - Solutionner le problème tel que décrit
  - N'oubliez pas les conventions d'écriture
  - Soumettez votre projet selon les indications de l'instructeur

Diagrammes UML

- **UML** = *Unified Modelling Language*
  - Représentation graphique de classes et de leurs interactions en POO
    - Facilite la formulation et la compréhension d'architectures de logiciels
  - Indépendant du langage de programmation
- Vous étudierez en détails le UML à l'étape 6 de TGI (*ORD11167 - Analyse et conception de systèmes*)

Représentation UML de classes

- Chaque classe représentée par une boîte

Nom de classe

Attributs membres
(Types et valeurs par défaut)

Fonctions membres
(Prototypes)

Temps
- heure : int = 0 - minute : int = 0 - seconde : int = 0
+ ajusterTemps(int h=0, int m=0, int s=0) : void + validerTemps() : bool + lireMilitaire() : void + afficherMilitaire() : void + afficherStandard() : void + getHeure() : int + setHeure(int h) : int

- *pour privé,* + *pour public*

Erreurs de programmation

- Omettre les parenthèses dans `(*p).attrib` (où `p` est un pointeur)
  - Utiliser `p->attrib` évite ce type d'erreur
- Oublier que **`class`** (et **`struct`**) requiert un `;` à la fin
- Tenter d'initialiser explicitement un attribut membre dans la classe

```
class Temps {
public:
  void Temps();
  ...
private:
  int heure   = 0;
  int minute  = 0;
  int seconde = 0;
};
```

- Invoquer un membre privé à l'extérieur d'une fonction membre
- Spécifier un type de retour à un constructeur ou destructeur

Erreurs de programmation (suite)

- Avoir deux constructeurs ayant une signature équivalente
  - Exemple :

```
class Temps {
public:
  Temps();
  Temps( int = 0, int = 0, int = 0 );
  ...
};
```

  - Puisque le constructeur paramétré peut être invoqué sans argument, il est équivalent au paramètre par défaut

Bonnes pratiques de programmation

- Exploiter l'opérateur flèche (`->`) plutôt que `(*).`
- Toujours incorporer le contenu d'un fichier d'entête dans un bloc **`#ifndef`**/**`#define`**/**`#endif`**
- Toujours spécifier l'identificateur d'accès **`private`** même si celui-ci est optionnel
  - Et regrouper ensembles tous les membres ayant la même accessibilité
- Toujours fournir un constructeur par défaut à une classe, même si le contenu est vide
  - Et, si possible, toujours fournir des arguments par défaut aux constructeurs

Devoir #5

- Solutionnez le problème distribué par l'instructeur
  - Créer un nouveau projet console *Visual Studio C++*
  - Solutionner le problème tel que décrit, avec la spécification supplémentaire suivante
    - Le code source de votre projet doit être réparti dans trois fichiers : `rationnel.h`, `rationnel.cpp` et `Devoir_5.cpp`
    - L'instructeur vous fourni un programme principal (i.e. le fichier `Devoir_5.cpp`) qui vous permettra de tester votre classe **`Rationnel`**
  - N'oubliez pas les conventions d'écriture
  - Respectez l'échéance imposée par l'instructeur
  - Soumettez votre projet selon les indications de l'instructeur
    - **Attention :** respectez à la lettre les instructions de l'instructeur sur la façon de soumettre vos travaux, s*inon la note EC sera attribuée à ceux-ci*

Pour la semaine prochaine

- Vous devez relire le contenu de la présentation du chapitre 6
  - Il y aura un quiz sur ce contenu au prochain cours
    - À livres et ordinateurs fermés
  - Profitez-en pour réviser le contenu des chapitres précédents