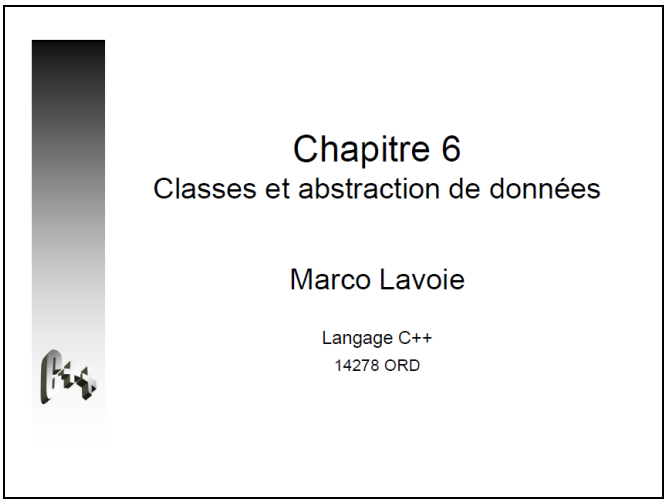
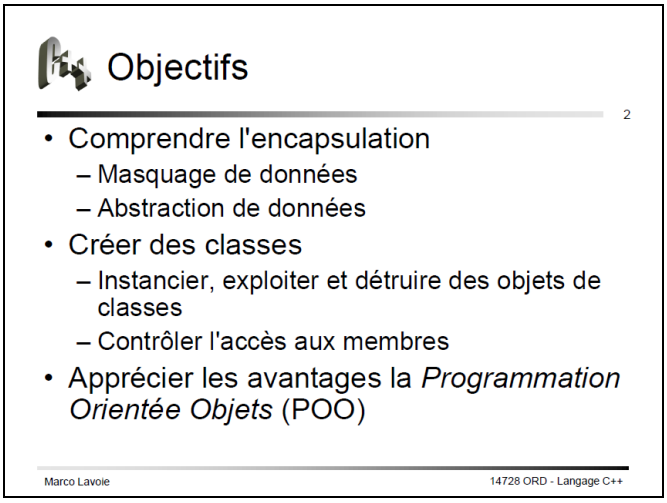


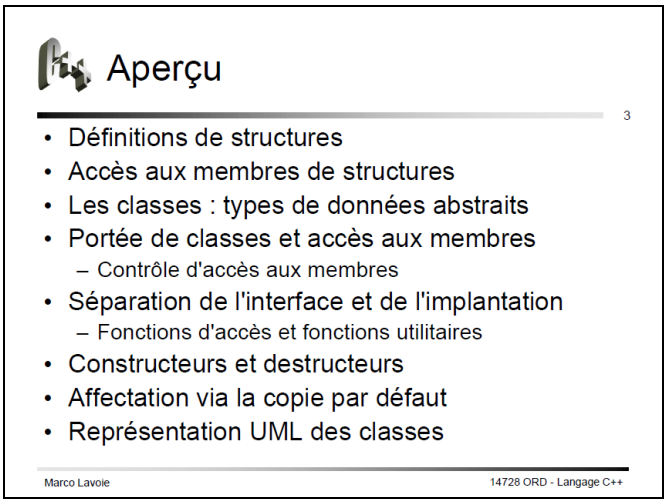
Ce chapitre aborde l'introduction à la *programmation orientée objet* (POO) en C++. La POO *encapsule* les données (*attributs membres*) et les fonctionnalités (*fonctions membres*) à l'intérieur d'ensembles appelés *classes*, les données et fonctions d'une classe étant intimement liées les unes aux autres.

Une classe ressemble à un plan : comme un ouvrier qui peut construire une maison à partir d'un plan, un programmeur pour créer un objet à partir d'une classe. De même qu'un plan qui peut être utilisé plusieurs fois pour fabriquer de nombreuses maisons, une classe peut être utilisée à maintes reprises pour créer plusieurs objets.



Les classes offrent la propriété du *masquage des informations*, ce qui signifie que, même si les objets de classes peuvent communiquer entre eux par le biais d'*interfaces* bien définies, on ne permet normalement pas aux classes de connaître les détails d'implantation des autres classes; ces détails d'implantation sont masqués à l'intérieur des classes elles-mêmes.

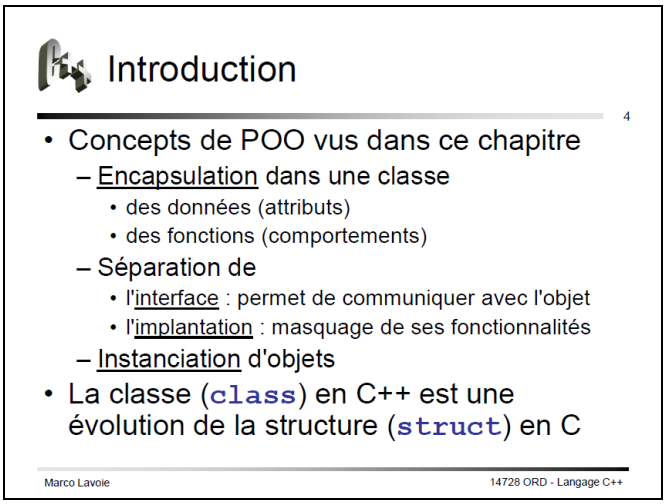
Chaque classe contient des données dans ses attributs membres, ainsi qu'une série de fonctions membres manipulant ces données. Les fonctions membres sont aussi communément appelées des *méthodes*.



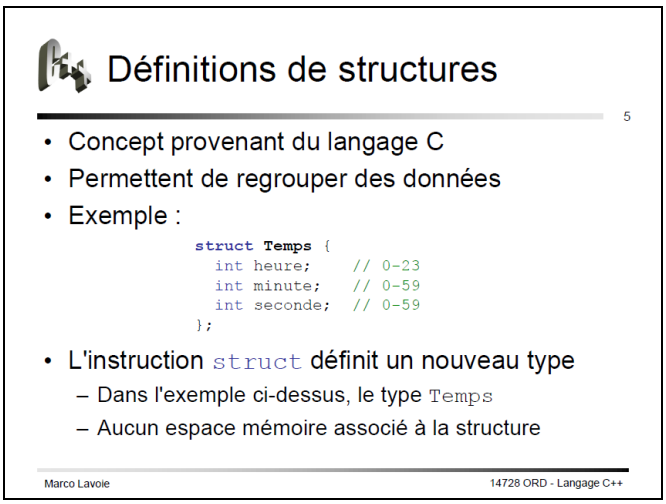
Dans ce premier chapitre portant sur la programmation orientée objet en C++, nous abordons les principes de base de la POO, soit l'encapsulation des données et de fonctionnalités dans la classe, le contrôle d'accès à ces données et fonctionnalités, ainsi que les composants essentiels à la manipulation d'objets créés à partir d'une classe, les *constructeurs* (qui contrôlent la création d'objets) et les *destructeurs* (qui en contrôle la destruction).

Nous verrons aux chapitres suivants qu'une classe peut disposer de plusieurs *opérateurs* facilitant la manipulation d'objets. L'*opérateur d'affectation* en est un dont nous verrons les fondements dans ce chapitre.

Finalement, en fin de chapitre nous aborderons le schéma de représentation graphique le plus utilisé pour la modélisation des systèmes orientés objets : la *modélisation UML*. Par modélisation il faut comprendre la représentation des classes sous forme de modèles graphiques.

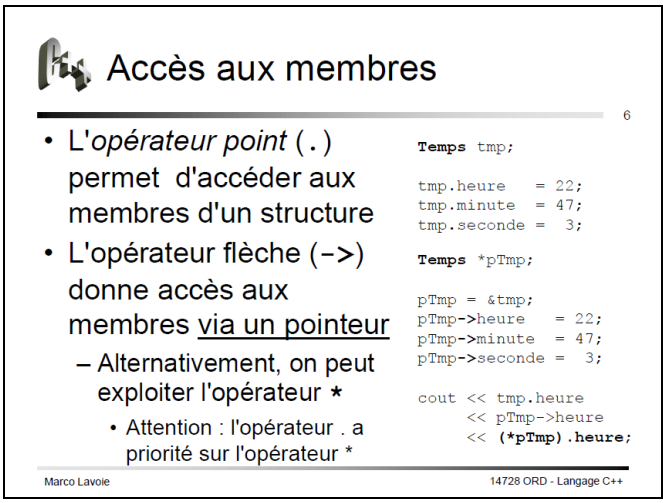


Les classes du C++ représentent une évolution naturelle de la notion de structure que le langage C désigne sous le mot-clé **`struct`**. Avant d'aborder les particularités du développement des classes en C++, nous allons discuter des structures et construire un type défini par l'utilisateur, basé sur une structure. Les faiblesses que nous relèverons dans cette approche nous aiderons à motiver la naissance de la notion de classe.



Les structures sont construites en regroupant des éléments d'autres types, y compris d'autres structures. Analysons la définition de structure ci-contre. Le mot-clé **`struct`** introduit une définition de structure, **`Temps`** représente l'*identificateur de la structure*, qui dénomme la définition de la structure, utilisée à son tour pour déclarer des variables du *type de la structure*. Dans cet exemple, le nom du nouveau type est identifié par **`Temps`** et les noms, déclarés entre accolades dans la définition de structure, représentent les *attributs membres* de cette dernière. Les membres d'une même structure doivent posséder des noms uniques, bien que, deux structures différentes puissent contenir des membres de même nom sans qu'il y ait de conflit. Toute définition de structure doit se terminée par un point-virgule.

Une définition de structure ne réserve aucun espace en mémoire; elle crée plutôt un nouveau type de données utilisable pour déclarer des variables qui, elles, auront un espace mémoire attribué.



On accède aux attributs membres d'une structure en utilisant les opérateurs d'accès aux membres : l'*opérateur point* (`.`) et l'*opérateur flèche* (`->`). L'opérateur point accède à un membre d'une structure par le biais du nom d'une variable du type de cette structure (ou d'une référence à une telle variable), alors que l'opérateur flèche (constitué du signe moins, `-`, suivi du signe plus grand, `>`, sans espace intermédiaire), permet d'accéder à un membre de structure via un pointeur vers une variable du type de cette structure.

Comme nous l'avons vu au chapitre précédent, les opérateurs de déréférencement (`*`) et point (`.`), accompagnés des parenthèses (pour contrecarrer les priorités relatives de ces deux opérateurs), peuvent se substituer à l'opérateur flèche pour accéder aux membres d'une structure.

Fonctionnalités pour structures

7

- 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" );
}
```

Marco Lavoie 14728 ORD - Langage C++

L'exemple ci-contre présente deux fonctions permettant de manipuler les variables de type de structure **`Temps`** : **`afficherMilitaire`** et **`afficherStandard`**. Ces deux fonctions « encapsulent » les détails d'affichage d'un argument de type **`Temps`**. Ainsi, dans le programme principal, on peut afficher le contenu d'une variable de type **`Temps`** sans connaître les détails intrinsèques de la structure **`Temps`**, tout simplement en invoquant une des deux fonctions ci-contre tout en leur passant la variable de type **`Temps`** en argument (dont le paramètre **`t`** de la fonction sera un alias puisque les deux fonctions implantent l'*appel par référence via référence*).

Fonctionnalités (suite)

8

```
// 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;
}
```

Marco Lavoie 14728 ORD - Langage C++

Notez la distinction au niveau de la déclaration du paramètre **`t`** entre la fonction **`lireMilitaire`** et les deux fonctions d'affichage précédentes. Les fonctions **`afficherMilitaire`** et **`afficherStandard`** déclarent **`t`** de type **`const Temps`** alors que **`lireMilitaire`** déclare **`t`** de type **`Temps`** (c.à.d. sans le qualificatif **`const`**). La raison est que **`lireMilitaire`** change le contenu de son paramètre **`t`** (qui est en fait un alias de l'argument **`tempsSouper`** de **`main`**) alors que les deux fonctions d'affichage ne modifient pas le contenu de leur paramètre **`t`**.

Notez que les deux fonctions d'affichage pourraient exploiter l'appel par valeur (c.à.d. sans **`&`**) avec le même résultat :

```
void afficherMilitaire(const Temps t);
void afficherStandard(const Temps t);
```

mais l'appel par référence apporte une efficacité accrue en évitant le transfert du contenu de l'espace mémoire de l'argument vers celui du paramètre.

Fonctionnalités (suite)

9

- 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**

Marco Lavoie 14728 ORD - Langage C++

Comme mentionné précédemment, le programme **`main`** de l'exemple actuel manipule une variable de type **`Temps`** sans manipuler directement les attributs membres de cette variable, grâce aux trois fonctions accompagnant la structure **`Temps`**. Ces trois fonctions assurent le *masquage* des attributs membres de la structure **`Temps`** : elles en permettent l'exploitation sans avoir à connaître les « détails » du contenu de la structure.

Ce principe de masquage des membres d'une structure est nommé *encapsulation* en programmation orientée objet.

Encapsulation dans une classe

10

- Contrairement à la structure (**`struct`**) 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*

Marco Lavoie 14728 ORD - Langage C++

Les classes permettent au programmeur de modeler des objets possédant des *attributs* (représentés par des *attributs membres*) et des comportements ou opérations (représentés par des *fonctions membres*). En C++, les structures contenant des attributs membres et des fonctions membres se définissent grâce au mot-clé **`class`**.

Dans certains autres langages de programmation orientés objets, les fonctions membres sont désignées parfois par le terme *méthodes* et répondent à des *messages* transmis vers un objet. Dans le contexte du C++, un tel message correspond à une invocation de fonction membre de la classe du type de l'objet destinataire de ce message.

Encapsulation dans une classe (suite)

11

- 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*

Marco Lavoie 14728 ORD - Langage C++

En plus de définir des attributs membres et des fonctions membres, les classes peuvent aussi définir des *opérateurs* (tels que **`+`**, **`<<`** et **`[]`**) pour adapter le comportement des objets issus de la classe à ces opérateurs (p.ex. afficher une variable de type **`Temps`** via l'opérateur de flux **`<<`**, ou additionner une heure à cette même variable via l'opérateur **`+`**).

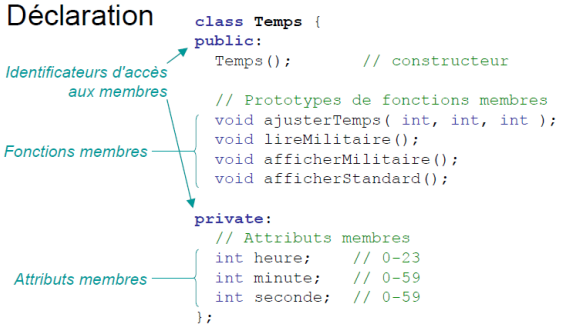
Une définition de classe contient aussi généralement un ou plusieurs *constructeurs d'objets* qui permettent de paramétrer la création d'objets du type de la classe, ainsi qu'un *destructeur* permettant de paramétrer leur destruction. On pourrait par exemple intégrer un constructeur à la classe **`Temps`** afin d'initialiser les variables de ce type à l'heure de l'ordinateur.

Finalement, une classe classifie ses membres (attributs, fonctions et opérateurs) selon trois *identificateurs d'accessibilité* : **`private`**, **`protected`** et **`public`**. Ces identificateurs permettent de contrôler l'accès aux membres de la classe.

Classe `Temps`

12

- Déclaration




```
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
};
```

Marco Lavoie 14728 ORD - Langage C++

La définition de la classe **`Temps`** ci-contre dispose de trois attributs membres ainsi que trois fonctions membres. Déclaré à la suite de l'identificateur d'accès **`public`** (et avant l'identificateur d'accès suivant), tout membre d'attribut ou de fonction est accessible à tout endroit où le programme permet d'accéder à un objet de type **`Temps`**. Pour leur part, tout membre déclaré après un identificateur d'accès **`private`** (et jusqu'à l'identificateur d'accès suivant) n'est accessible qu'aux fonctions membres de la classe **`Temps`**.

Identificateurs d'accessibilité

13

- 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)

Marco Lavoie 14728 ORD - Langage C++

Les identificateurs d'accès aux membres sont toujours suivis d'un deux-points (**`:`**) et peuvent apparaître autant de fois que nécessaire dans une définition de classe, dans n'importe quel ordre.

Tel que mentionné précédemment, les membres d'accès **`private`** sont seulement accessibles dans le corps des fonctions membres de la classe (p.ex. dans la classe **`Temps`** de la page précédente, seules les fonctions membres **`ajusterTemps`**, **`lireMilitaire`**, **`afficherMilitaire`** et **`afficherStandard`** ainsi que le constructeur **`Temps()`** ont accès aux attributs membres **`heure`**, **`minute`** et **`seconde`**. On dit que la classe **`Temps`** *encapsule* (ou masque) ces membres au monde extérieur.

Contrairement aux attributs membres, le constructeur et les fonctions membres de la classe sont déclarés d'accès **`public`**, ce qui signifie qu'ils sont accessibles au code de programme autre que celui des fonctions membres de la classe. Ces membres représentent ainsi l'*interface* de la classe pour le monde extérieur.

Fonctions membres

14

- 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" );
}
```

Marco Lavoie 14728 ORD - Langage C++

La définition d'une classe regroupe les déclarations des attributs membres et des fonctions membres de cette classe. Les déclarations des fonctions membres se présentent sous forme des prototypes de fonctions que nous avons évoqués aux chapitres précédents. Les fonctions membres peuvent être définies à l'intérieur de la classe, mais une bonne pratique de programmation consiste à les placer en dehors de la définition de la classe elle-même.

Dans l'exemple ci-contre (qui poursuit la définition de la classe **`Temps`**), les définitions des fonctions membres de la classe sont intègrent le nom de la classe suivi de l'*opérateur de portée* (**`::`**) en préfixe du nom de la fonction. Puisque différentes classes peuvent avoir des fonctions membres identiques (i.e. de même signature), l'opérateur de portée relie le nom de la fonction membre à sa classe, afin d'identifier de manière unique les fonctions membres d'une classe particulière.

Fonctions membres (suite)

15

- 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( Temps & ) {
  char sep;
  cin >> t.heure >> sep >> t.minute >> sep >> t.seconde;
}
```

  - La fonction s'exécute pour une instance

Marco Lavoie 14728 ORD - Langage C++

Il est intéressant de noter que les fonctions membres **`lireMilitaire`**, **`afficherMilitaire`** et **`afficherStandard`** ne prennent aucun argument. Ceci s'explique par le fait que les fonctions membres savent implicitement qu'elles manipuleront les attributs membres d'un objet de type **`Temps`** particulier pour lequel elles seront invoquées. Puisqu'un objet de type **`Temps`** est requis pour invoquer ses fonctions membres, ces dernières n'ont pas à recevoir cet objet en paramètre, ayant implicitement accès à l'objet en question.

Les appels de fonctions membres offrent donc une plus grande concision que les appels de fonctions conventionnelles associées à la programmation structurée (telles que celles accompagnant la structure **`Temps`** vue précédemment).

Instanciation d'une classe

16

- 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;
}
```

Marco Lavoie 14728 ORD - Langage C++

Analysons une utilisation de la classe **`Temps`**. Le programme ci-contre crée ou *instancie* un objet, appelé **`tempsSouper`**, de la classe **`Temps`**. Lors de la création de l'objet, le constructeur **`Temps()`** de la classe Temps est implicitement invoqué afin d'initialiser l'objet **`tempsSouper`** à *0h0m0s* (voir la page suivante). L'heure est ensuite lue du clavier via une invocation de la fonction membre **`lireMilitaire`** pour l'objet **`tempsSouper`**. Puisque les fonctions membres de la classe **`Temps`** doivent être invoqués par un objet de cette classe, l'opérateur point (.) est exploité pour associer l'invocation de **`lireMilitaire`** à l'objet **`tempsSouper`**. Ainsi, **`lireMilitaire`** modifie les attributs de **`tempsSouper`**.

L'heure stockée dans **`tempsSouper`** est ensuite affichée à l'écran en format militaire, puis en format standard.

Constructeurs

17

- 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;
}
```

Marco Lavoie 14728 ORD - Langage C++

Un constructeur d'une classe se présente sous la forme d'une fonction membre spéciale, qui initialise les attributs membres d'un objet de la classe et qui est invoquée automatiquement lors de la création d'un objet de cette classe. Comme nous le verrons, il est courant de retrouver plusieurs constructeurs pour une même classe; c'est ce que permet la *surcharge de fonctions*. Le constructeur ne comportant aucun paramètre (une classe ne peut en avoir plus qu'un) est généralement appelé le *constructeur par défaut*; c'est celui qui est implicitement invoqué lorsqu'on crée un objet du type de la classe (les constructeurs autres que celui par défaut doivent être explicitement invoqués, comme nous le verrons plus tard).

Notez que l'on ne spécifie aucun type de retour pour un constructeur.

Constructeurs (suite)

18

- 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

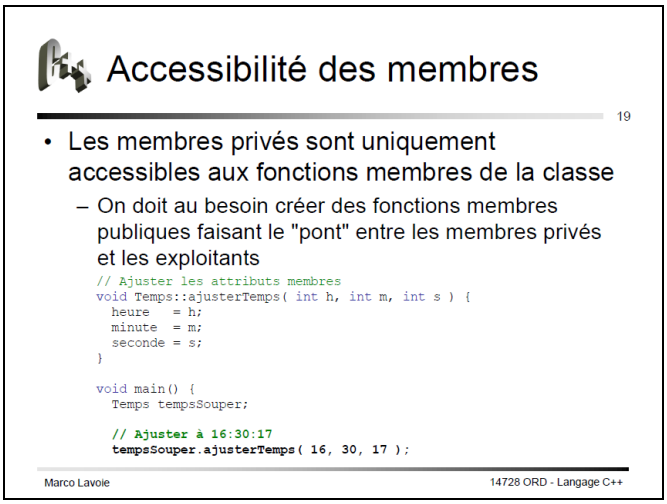
```
class Temps {
public:
  Temps();

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

private:
  int heure  = 0;
  int minute = 0;
  int seconde = 0;
};
```

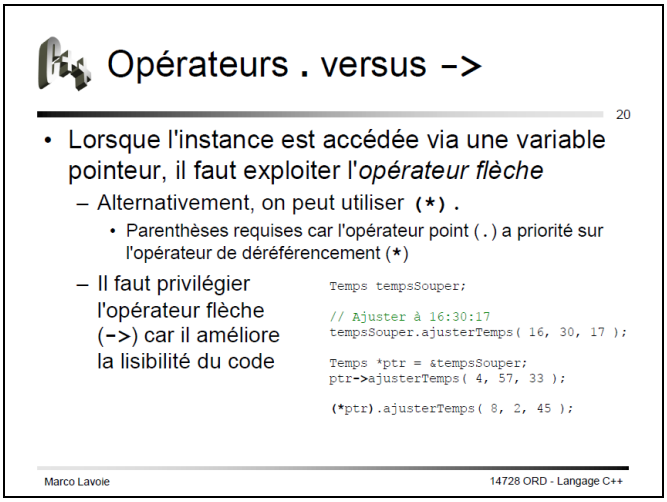
Marco Lavoie 14728 ORD - Langage C++

Notez que, contrairement à d'autres langages de programmation, il n'est pas permis en C++ d'initialiser les attributs membres d'une classe à l'emplacement de leur déclaration, dans le corps de la classe. Ces membres doivent être initialisés par un constructeur de la classe ou recevoir des valeurs par l'intermédiaire des fonctions membres de cette classe.



Les membres (attributs et fonctions) d'une classe appartiennent à la *portée* (on dit aussi *étendue*) de cette classe. Les fonctions non membres (c.à.d. conventionnelles) se définissent sous la portée du fichier.

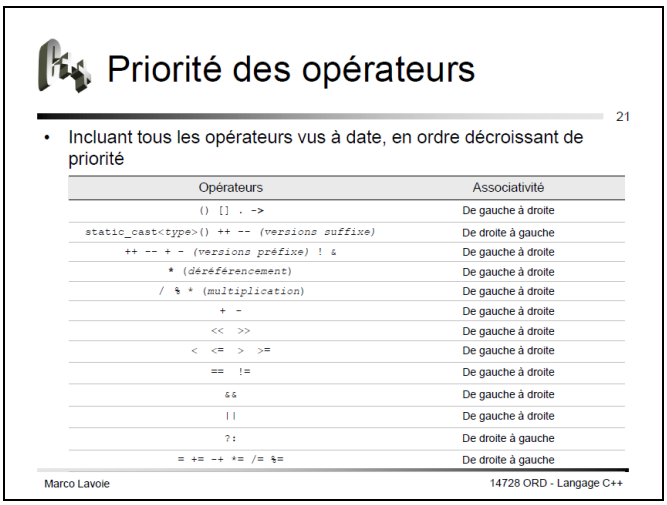
Sous la portée d'une classe, les membres de la classe deviennent immédiatement accessibles par toutes les fonctions membres de la classe et peuvent être référencés par leur nom. En dehors de la portée de la classe, la référence aux membres s'effectue par le biais de l'un des identificateurs d'objet : une variable d'objet, une référence ou un pointeur vers l'objet. Le chapitre 7 montrera que le compilateur insère un identificateur implicite (explicitement identifié par le mot-clé **`this`**) pour chaque référence à un membre de la classe dans les fonctions membres de cette dernière.



Les opérateurs utilisés pour accéder aux membres d'une classe sont les mêmes à ceux utilisés pour accéder aux attributs d'une structure. L'*opérateur point* (**`.`**) se combine avec le nom d'une variable d'objet (ou d'une référence à un objet) afin d'accéder aux membres de l'objet. L'*opérateur flèche* (**`->`**), combiné à un pointeur vers un objet, permet l'accès aux membres de cet objet.

Le programme ci-contre instancie la variable **`tempsSouper`** et l'initialise à *16h30m17s*. La variable pointeur **`ptr`**, initialisée pour pointer vers **`tempsSouper`**, est ensuite utilisée avec l'opérateur flèche pour modifier les attributs membres de **`tempsSouper`** à *4h57m33s*.

Comme mentionné à maintes reprises, l'opérateur flèche peut être remplacé par l'opérateur point (**`.`**) accompagné de l'opérateur de déréférencement (**`*`**). Ceci constitue cependant une mauvaise pratique de programmation.



Voici à nouveau la liste des opérateurs étudiés à date, en ordre décroissant de priorité.

Séparation de l'interface et l'implantation

22

- 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`)

Marco Lavoie 14728 ORD - Langage C++

Un des principes fondamentaux d'une bonne conception de logiciels consiste à séparer l'interface de l'implantation pour faciliter la modification des programmes. En ce qui concerne les *clients* de la classe (c.à.d. les programmes exploitant des objets issus de la classe), les changements dans l'implantation de la classe n'affectent en rien ces clients en autant que ces changements soient confinés à l'implantation des fonctions membres non privées de la classe, c'est-à-dire que les attributs membres et les prototypes des fonctions membres désignés **`protected`** ou **`public`** ne soient pas modifiés.

Une stratégie d'implantation couramment appliquée en génie logiciel consiste à séparer la déclaration de la classe (c.à.d. la définition **`class`** et son corps) et la définition des fonctions membres dans des fichiers distinctes. Cette approche constitue une bonne pratique de programmation en C++.

Séparation : fichier en-tête

23

- 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
```

Marco Lavoie 14728 ORD - Langage C++

L'exemple ci-contre divise le programme précédent en plusieurs fichiers. Lors de la construction d'un programme en C++, chaque définition de classe est normalement placée dans un *fichier d'en-tête* et les définitions des fonctions membres de cette classe le sont dans des *fichiers de code source* portant le même nom mais pas la même extension de nom. Le nom du fichier contenant la définition de la classe comporte généralement l'extension **`.h`** (p.ex. **`temps.h`**) alors que le fichier du même nom contenant les définitions de fonctions membres de la classe comporte l'extension **`.cpp`** (p.ex. **`temps.cpp`**).

Séparation : fichier source

24

- 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
```

Marco Lavoie 14728 ORD - Langage C++

Le fichier d'en-tête est incorporé, par un **`#include`**, dans chaque fichier utilisant la classe, tandis que le fichier source de la classe est compilé et lié avec celui du programme principal. C'est l'environnement de développement *Visual Studio* qui s'occupe de compiler et lier un programme constitué de plusieurs fichiers d'en-têtes et sources.

L'exemple ci-contre est le fichier source contenant les définitions des fonctions membres de la classe **`Temps`**, dont la définition de classe est contenue dans le fichier **`temps.h`**. Ainsi, l'instruction **`#include "temps.h"`** indique au compilateur où retrouver la définition de la classe **`Temps`** dont les fonctions membres sont définies plus bas.

Séparation : programme principal

25

- 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();
}
```

Marco Lavoie 14728 ORD - Langage C++

Tel que mentionné précédemment, tout programme exploitant une classe dont la définition est séparée en deux fichiers doit spécifiquement inclure le fichier d'en-tête de cette classe, le compilateur prenant en charge la fusion des fichiers sources afin de produire un exécutable.

Le fichier main.cpp contient le code source du programme principal. Puisque ce dernier exploite la classe **`Temps`**, le fichier d'en-tête **`temps.h`** est inclus dans **`main.cpp`**.

Séparation : pourquoi `#ifndef` ?

26

- 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
#define TEST_H

class Temps {
  ...
};

#endif
```

  - **`#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

Marco Lavoie 14728 ORD - Langage C++

Lorsque nous commencerons à élaborer de plus gros programmes, nous placerons d'autres définitions et d'autres déclarations dans les fichiers d'en-tête. L'utilisation des directives de précompilations **`#ifndef`**, **`#define`** et **`#endif`** telles qu'illustré ci-contre empêchent l'inclusion du code situé entre **`#ifndef`** (« *if not defined* ») et **`#endif`** si le nom de *constante symbolique* **`TEST_H`** est déjà défini. Si l'en-tête n'a pas été inclus auparavant dans un fichier, le nom **`TEST_H`** est défini grâce à la directive **`#define`**, permettant l'ajout des instructions du fichier d'en-tête. Par contre, si l'en-tête a été inclus auparavant, **`TEST_H`** est déjà défini et interdit une nouvelle inclusion du fichier d'en-tête.

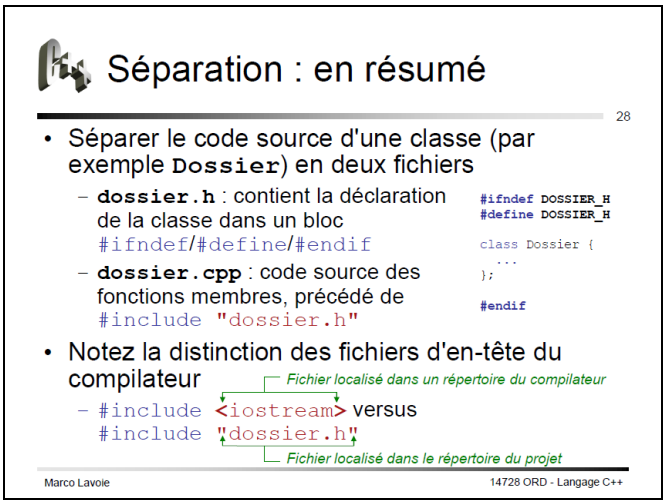
Séparation : pourquoi `#ifndef` ?

27

- 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

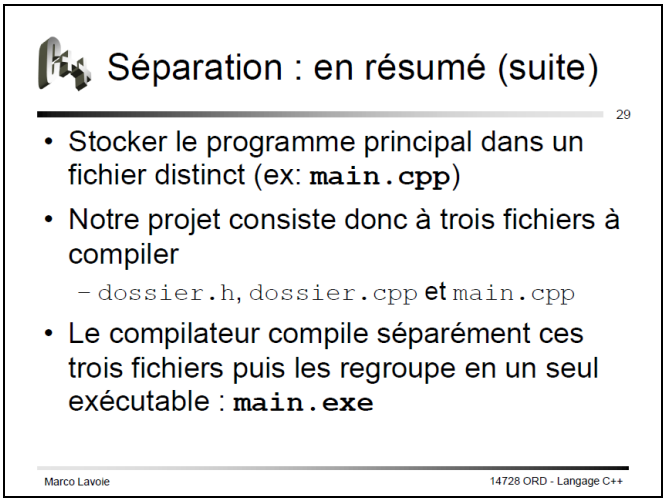
Marco Lavoie 14728 ORD - Langage C++

Les tentatives d'inclure plusieurs fois un fichier d'en-tête, par inadvertance ou par obligation, se produisent habituellement dans les programmes volumineux, contenant plusieurs fichiers d'en-tête pouvant à leur tour inclure d'autres fichiers d'en-tête. Les risques que le compilateur ait à compiler un fichier d'en-tête à de multiples reprises (ce qui constitue une erreur de syntaxe) étant plus grandes, l'exploitation des directives de précompilation **`#ifndef`**, **`#define`** et **`#endif`** avec constantes symboliques préviennent de telles situations.



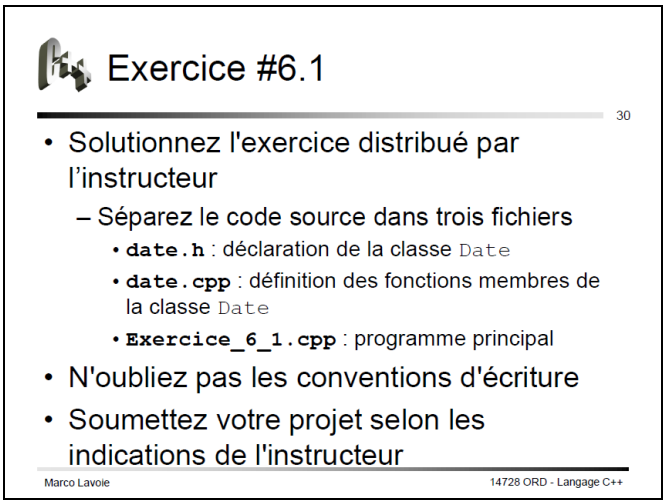
La convention adoptée en programmation C++ consiste à utiliser le nom du fichier pour le nom de constante symbolique dans les directives de précompilation, remplaçant le point par un souligné (p.ex. la constante **`TEMPS_H`** est exploitée dans le fichier **`temps.h`**).

Une autre bonne pratique de programmation consiste à appliquer cette stratégie aux fichiers sources également, au cas où un même ficher serait compilé plus d'une fois par le compilateur lors de la compilation des fichiers du programme. Par exemple, la constante **`TEMPS_CPP`** sera exploitée dans le fichier **`temps.cpp`**.



Dans un programme exploitant plusieurs classes, chaque classe devrait être implantée dans deux fichiers distincts : un fichier d'en-tête contenant la définition de la classe, et un fichier source contenant les définitions des fonctions membres de la classe. Chaque fichier source comportera ainsi une commande d'inclusion (**`#include`**) pour chaque classe exploitée dans le fichier.

C'est le rôle de l'environnement de développement (p.ex. *Visual Studio*) de compiler et lier ensembles tous les fichiers du programme afin d'en faire un seul fichier exécutable.



Pour solutionner cet exercice, vous devez exploiter les fonctionnalités de *Visual Studio* pour créer une solution comportant plusieurs fichiers. L'instructeur vous démontrera comment procéder pour ajouter des nouveaux fichiers à une solution (il existe plus d'une façon d'y parvenir).

N'oubliez pas d'exploiter une constante symbolique dans les fichiers **`date.h`** et **`date.cpp`**, afin d'éviter la compilation multiple de leur contenu.

Contrôle d'accès aux membres

31

- 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

Marco Lavoie 14728 ORD - Langage C++

Les identificateurs d'accès aux membres **private** et **public** (et **protected** que nous verrons au chapitre 9) sont utilisés pour contrôler l'accès aux attributs membres ainsi qu'aux fonctions membres.

Les membres **private** d'une classe ne sont accessibles que par les fonctions membres (et par les **friend**, mot-clé signifiant « ami » que nous verrons au chapitre 7) de cette classe. Cependant, toute fonction du programme (telle que **main**) a accès aux membres **public** d'une classe.

Le C++ favorise l'indépendance des programmes vis-à-vis de l'implantation. Lorsque l'implantation d'une classe change alors qu'elle est utilisée par du code indépendant de l'implantation (i.e. n'ayant pas accès aux membres privés de la classe), ce code n'a nul besoin de subir des modifications. En revanche, si une partie de l'interface de la classe est modifiée, le code indépendant de l'implantation de la classe doit être adapté à ces modifications.

Contrôle d'accès aux membres

32

- 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

Marco Lavoie 14728 ORD - Langage C++

Pour les classes, le mode d'accès par défaut est **private**, de sorte que tous les membres insérés avant le premier identificateur d'accès sont d'office privés. Après chaque identificateur d'accès, le mode invoqué s'applique aux membres aux suivent jusqu'au prochain identificateur d'accès, ou jusqu'à l'accolade de terminaison de définition de la classe.

On peut répéter les identificateurs d'accès **public**, **private** et **protected** dans la classe, bien qu'un tel usage soit peu courant et qu'il puisse porter à confusion.

Fonctions membres utilitaires

33

- 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() );
}
```

Marco Lavoie 14728 ORD - Langage C++

Le seul fait que les attributs d'une classe soient **private** n'indique pas nécessairement que les clients ne pourront effectuer des modifications à leur contenu. Leurs valeurs peuvent être modifiées par les fonctions membres ou par les amis de cette classe. Si de telles fonctions sont d'accès **public**, les clients peuvent modifier le contenu des attributs membres privés de l'objet via les fonctions membres publiques de la classe.

Alors que les fonctions membres publiques font parties de l'*interface* de la classe (c.à.d. accessibles au code client de la classe), les fonctions membres privées de la classe constituent les *fonctions utilitaires* de celle-ci, apportant un support aux opérations des fonctions publiques de la classe. Les fonctions utilitaires ne sont pas conçues pour être directement employées par les clients de la classe.

Fonctions membres d'accès

34

- 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;
}
```

Marco Lavoie 14728 ORD - Langage C++

Comme mentionné précédemment, les fonctions membres publiques font parties de l'interface de la classe car elles sont accessibles au code client de la classe. Le rôle principal de ces fonctions consiste à présenter les différents *services* (comportements) de la classe à ses clients. Cette série de services permettent aux clients de les utiliser sans voir à se préoccuper de la manière dont la classe accomplit ses tâches.

Tant les membres **`private`** d'une classe que les définitions de ses fonctions membres **`public`** ne sont pas accessibles aux clients de la classe. Ces composants forment l'*implantation* de la classe.

Constructeurs

35

- 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

Marco Lavoie 14728 ORD - Langage C++

Lorsqu'un objet d'une classe est créé, ses attributs membres peuvent être initialisés par une fonction *constructeur* de cette classe. Un constructeur constitue une fonction membre de classe du même nom que cette dernière. Le programmeur de la classe fournit le constructeur et il est invoqué automatiquement par la suite, chaque fois qu'un objet de cette classe est instancié (c.à.d. créé).

On peut surcharger les constructeurs pour fournir une variété de moyens d'initialisation des objets instanciés d'une classe. Les attributs membres doivent être initialisés dans un constructeur de la classe car il n'est pas possible en C++ (contrairement à d'autres langages comme le C#) de leur affecter une valeur directement dans la définition de la classe.

Les constructeurs se distinguent des autres fonctions membres d'une classe par leur nom (étant le même que celui de la classe) et par l'absence d'une valeur de retour.

Constructeurs par défaut

36

- 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:\>
```

Marco Lavoie 14728 ORD - Langage C++

Le constructeur sans paramètres est désigné *constructeur par défaut* de la classe. C'est le constructeur qui est automatiquement invoqué lorsqu'un objet de classe est instancié sans argument (nous verrons à la prochaine page comment instancier avec arguments).

C'est une bonne pratique de programmation de fournir un constructeur par défaut à toute classe pour assurer l'initialisation approprié de chaque objet instancié avec des valeurs explicites. Les attributs membres de type pointeurs en particulier doivent toujours être initialisés à des valeurs adéquates ou à 0.

Constructeurs paramétrés

37

- 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

Marco Lavoie 14728 ORD - Langage C++

On peut surcharger les constructeurs d'une classe pour fournir une variété de moyens d'initialisation des objets instanciés de la classe. En autant que chaque constructeur ait une *signature* distincte des autres constructeurs de la classe, le compilateur peut compiler la définition de la classe ainsi que tout code client de celle-ci sans confusion.

Lors de l'instanciation d'objets d'une classe, le compilateur identifie la surcharge de constructeur invoqué en fonction de la signature de l'invocation. Si la création de l'objet comporte des arguments, par exemple **`t2(3, 47, 19)`** dans l'exemple ci-contre, le compilateur identifie le constructeur correspondant à l'instanciation, soit le deuxième constructeur dans le même exemple..

Constructeurs paramétrés (suite)

38

- 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

Marco Lavoie 14728 ORD - Langage C++

Comme les fonctions membres de classe ou fonctions conventionnelles, les constructeurs de classe peuvent contenir des arguments par défaut. L'exemple ci-contre définit un constructeur **`Temps`** afin d'inclure des arguments par défaut. Ils définissent à zéro chacun des paramètres. La fourniture d'arguments par défaut au constructeur garantit l'initialisation de l'objet en un état cohérent, même si aucune valeur n'est fournie lors de l'instanciation.

Un constructeur, fourni par le programmeur, qui impose une valeur par défaut à tous ses paramètres ne requiert explicitement aucun argument et constitue donc un constructeur par défaut, c'est-à-dire un constructeur dont l'invocation peut être effectué sans argument.

Dans une classe il ne peut y avoir qu'un seul constructeur par défaut, sinon le compilateur ne saurait lequel invoqué lorsqu'une instanciation ne fournirait aucun argument.

Destructeur

39

- 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

Marco Lavoie 14728 ORD - Langage C++

Un destructeur est une fonction spéciale, membre d'une classe, dont le nom consiste en un caractère *tilde* (~), suivi du nom de la classe. Cette convention de nom présente un attrait intuitif puisque, comme nous le verrons au prochain chapitre, l'opérateur tilde est l'*opérateur binaire inverse* et, de ce fait, le destructeur représente le complément du constructeur.

Le destructeur d'une classe est invoqué à la destruction d'un objet de cette classe, c'est-à-dire lorsque l'exécution du programme quitte la portée dans laquelle un objet de cette classe a été instancié.

Un destructeur ne reçoit aucun paramètre et ne retourne aucune valeur. Une classe ne peut donc pas posséder plus d'un destructeur puisque ceux-ci auraient une signature commune. La surcharge de destructeur n'est donc pas permise dans les classes.

Destructeur (suite)

40

- 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:\>
```

Marco Lavoie 14728 ORD - Langage C++

Notez que le destructeur fourni à **`Temps`** dans l'exemple ci-contre n'est d'aucune utilité. Nous verrons au chapitre 8 que les destructeurs se révèlent particulièrement appropriés pour les classes dont les objets contiennent de la mémoire allouée dynamiquement (p.ex. des tableaux et des chaînes), alors qu'au chapitre 7 nous verrons comment allouer dynamiquement de la mémoire.

Destructeur (suite)

41

- 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   ✗
}
```

Marco Lavoie 14728 ORD - Langage C++

En réalité, le destructeur ne détruit pas l'objet; il effectue un *nettoyage de terminaison* avant que le système ne réclame la mémoire allouée à l'objet, afin de libérer celle-ci pour être ultérieurement allouée aux objets futurs.

Fonctions accesseurs et mutateurs

42

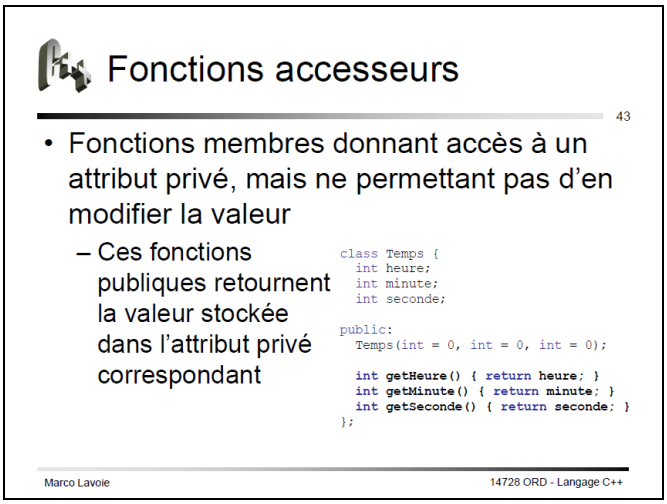
- En anglais : *getters* et *setters*
- Catégories de fonctions membres contrôlant l'accès aux attributs membres privés
  - Certains langages intègrent le principe des *getters* et *setters* dans leur syntaxe (p.ex. C# et VB : **`property`**), mais pas le C++
  - Le principe peut cependant être émulé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

Marco Lavoie 14728 ORD - Langage C++

Même si un attribut membre est déclaré **`private`**, il est parfois nécessaire de permettre aux clients de la classe un accès contrôlé à cet attribut. C'est le rôle des fonctions accesseurs et mutateurs :

- La fonction membre accesseur retourne la valeur stockée dans un attribut membre privé de la classe.
- Une fonction membre mutateur permet de modifier la valeur stockée dans un attribut membre privé de la classe.

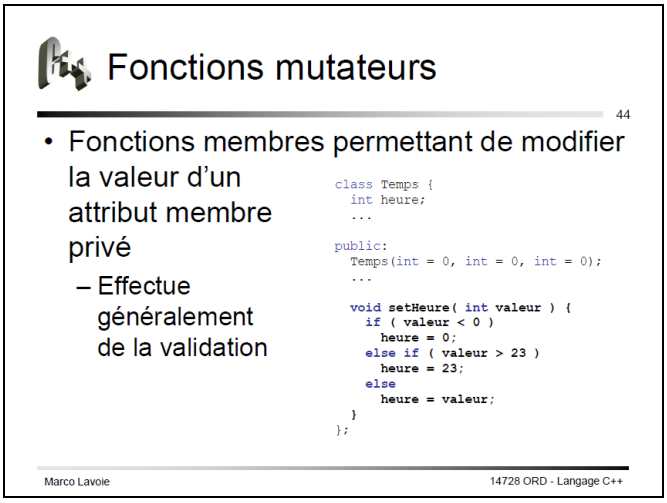
Ces fonctions d'accès **`public`** permettent donc aux clients d'accéder à et modifier la valeur d'un attribut membre privé, sans pour autant donnée un accès direct à l'attribut. Le client ne peut ainsi pas modifier de façon erronée le contenu de l'attribut, via une affectation par exemple.



La fonction accesseur, généralement déclarée *inline* (c.à.d. dans la définition de la classe), retourne la valeur de l'attribut membre correspondant. Le code client peut ainsi obtenir la valeur de l'attribut membre privé d'un objet du type de la classe, sans pour autant avoir la possibilité de modifier cette valeur dans l'attribut.

L'alternative à l'emploi d'une fonction accesseur est de déclarer les attributs membres correspondants **`public`** afin que les clients puissent avoir accès à leurs contenus. Cette alternative est cependant risquée car elle permet aussi aux clients d'attribuer des valeurs invalides à ces attributs. Par exemple, si l'attribut **`heure`** de la classe **`Temps`** était d'accès **`public`**, les clients pourraient lui attribuer des valeurs comme **`-1`** ou **`234`**.

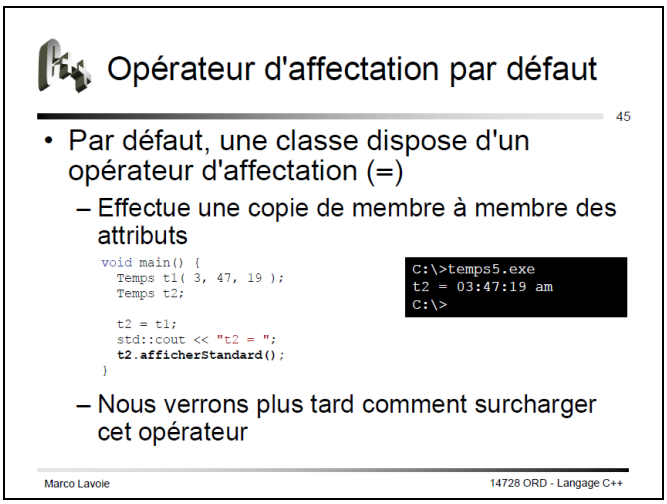
En C++, une fonction membre déclarée à l'intérieur de la définition de la classe est considérée *inline* par le compilateur. Il n'est pas nécessaire d'insérer en préfixe le mot-clé **`inline`** pour que le compilateur considère la fonction membre ainsi.



Le complément de la fonction membre accesseur est la fonction membre mutateur, qui permet aux clients de la classe une modification contrôlée du contenu d'un attribut membre privé. L'exemple ci-contre en fait la démonstration : la fonction mutateur **`setHeure`** permet de modifier le contenu de l'attribut privé **`heure`**, mais seulement avec une valeur valide. Toute valeur invalide fournie à la fonction **`setHeure`** est ramenée dans l'intervalle de valeurs considérées valides par la classe **`Temps`** pour son attribut **`heure`**, soit entre *0h* (c.à.d. minuit) et *23h*.

Par convention, les fonctions accesseurs et mutateurs sont nommées selon le nom de leur attribut membre privé correspondant, précédé du préfixe **`get`** pour les accesseurs et **`set`** pour les mutateurs. Ces deux préfixes sont issus des désignations anglaises *getter* et *setter*.

Les accesseurs et mutateurs sont généralement d'accès **`public`**, mais il y a parfois des exceptions.



L'opérateur d'affectation (=) peut servir à affecter un objet à un autre objet de même type. Par défaut, une telle affectation se fait par *copie de membre à membre*; chaque attribut membre d'un objet est copié (affecté) individuellement dans l'attribut membre correspondant de l'autre objet.

Cette copie de membre à membre peut engendrer de sérieux problèmes si on l'utilise avec une classe dont des attributs membres font référence à de la mémoire allouée dynamiquement. Le chapitre 8, *Surcharge des opérateurs*, expose ces problèmes et illustre une manière de les contourner via la surcharge de l'opérateur d'affectation.

Exercice #6.2

46

- 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

Marco Lavoie 14728 ORD - Langage C++

Cet exercice poursuit le travail commencé sur la classe **`Date`** à l'exercice 6.1. Vous devez ajouter des accesseurs et mutateurs à la classe afin de maintenir des attributs membres privés valides en tout temps.

Vous devez de plus ajouter une nouvelle fonction membre publique à la classe permettant d'incrémenter la date stockée dans un objet **`Date`** d'une journée. Cette opération ne se résume pas uniquement à incrémenter l'attribut **`jour`** puisque, la date résultante devant toujours être valide, l'incrémentation peut mener à une modification du mois, et éventuellement de l'année. Par exemple, la date du *31 décembre 2003* incrémentée d'une journée passe au *1 janvier 2004*, où les trois attributs membres privés de l'objet de classe **`Date`** sont modifiés.

Diagrammes UML

47

- **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*)

Marco Lavoie 14728 ORD - Langage C++

UML (*Unified Modeling Language*) est une méthode de modélisation orientée objet utilisée pour décrire un programme exploitant une ou plusieurs classes.

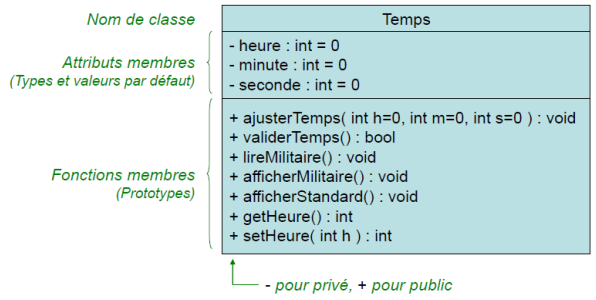
UML est une méthode utilisant une représentation graphique. L'usage d'une représentation graphique est un complément excellent à celui de représentions textuelles. En effet, l'une comme l'autre sont ambigües mais leur utilisation simultanée permet de diminuer les ambigüités de chacune d'elle. Un dessin permet bien souvent d'exprimer clairement ce qu'un texte exprime difficilement et un bon commentaire permet d'enrichir une figure.

Dans UML, il existe plusieurs « modèles » ou formalismes : modèles *de classes*, *d'états*, *des cas d'utilisation*, *des interactions*, etc. Dans ce chapitre nous introduisons le modèle de classes.

Représentation UML de classes

48

- Chaque classe représentée par une boîte



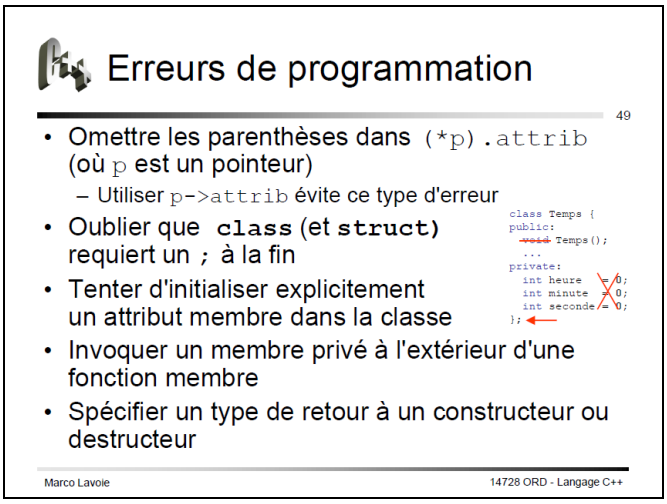
Marco Lavoie 14728 ORD - Langage C++

Le *diagramme de classe* permet de modéliser une classe sous forme de boîte énumérant les membres de la classe. Cette boîte est divisée verticalement en trois sections :

1. La première section présente l'identificateur (c.à.d. le nom) de la classe.
2. La deuxième section présente les attributs membres de la classe, chacun avec sont type et sa valeur par défaut correspondante.
3. La troisième section présente les prototypes des fonctions membres de la classe.

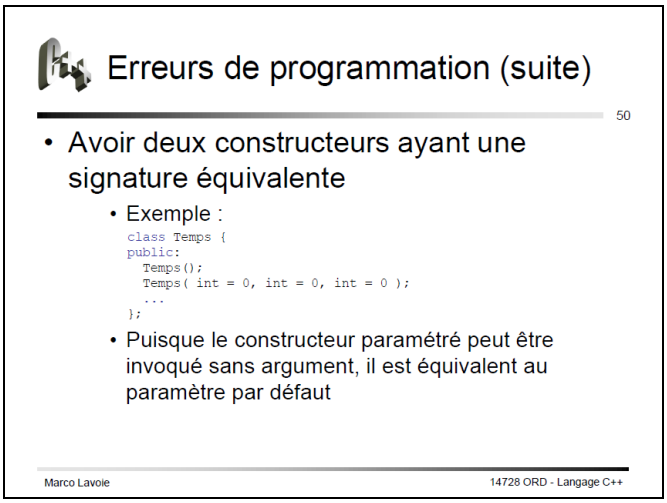
Les membres de la classe sont tous précédés d'un symbole indiquant sous quel identificateur d'accès est déclaré le membre : **`-`** pour un membre **`private`**, **`+`** pour un membre **`public`**.

Nous poursuivrons l'étude des modèles UML aux chapitres subséquents.

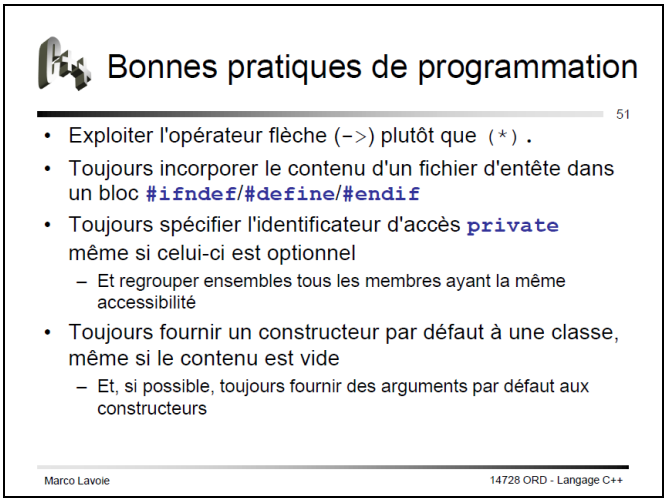


Voici d'autres erreurs de syntaxe fréquemment produite par les programmeurs en C++ :

- Le fait de spécifier un type de valeur de retour pour un constructeur, ou d'omettre un type de valeur de retour pour une fonction membre n'étant pas un constructeur. Seuls les constructeurs ne doivent pas avoir de type de valeur de retour dans une classe. Si une fonction membre de classe ne doit pas retourner un résultat, il faut indiquer **`void`** comme type de valeur de retour pour cette fonction membre.
- Lors de la définition d'une fonction membre à l'extérieur de sa classe, l'omission du nom de classe et de l'opérateur de portée (`::`) en préfixe du nom de la fonction constitue une erreur de syntaxe, car le compilateur ne peut savoir alors à quelle classe correspond cette fonction membre.

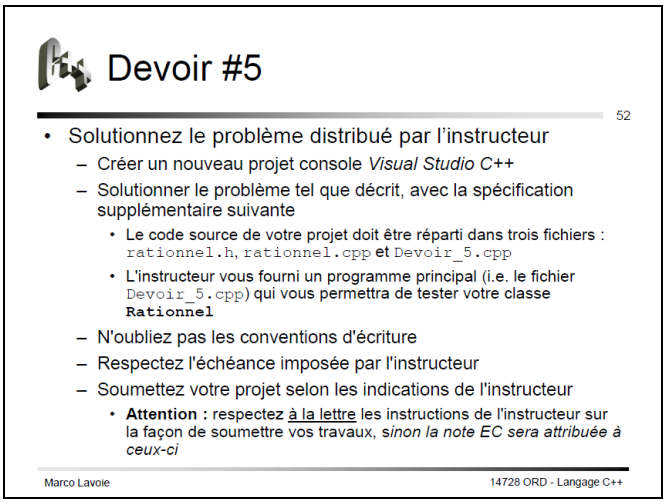


C'est aussi une erreur de syntaxe de tenter de définir plus d'un destructeur dans une classe, ou de définir un destructeur avec paramètres. Il est en effet inutile qu'un destructeur ait des paramètres puisque, contrairement aux constructeurs, il est impossible pour un client de la classe d'invoquer explicitement son destructeur à la fin de portée d'un objet de cette classe.



Voici d'autres bonnes pratiques de programmation en C++ :

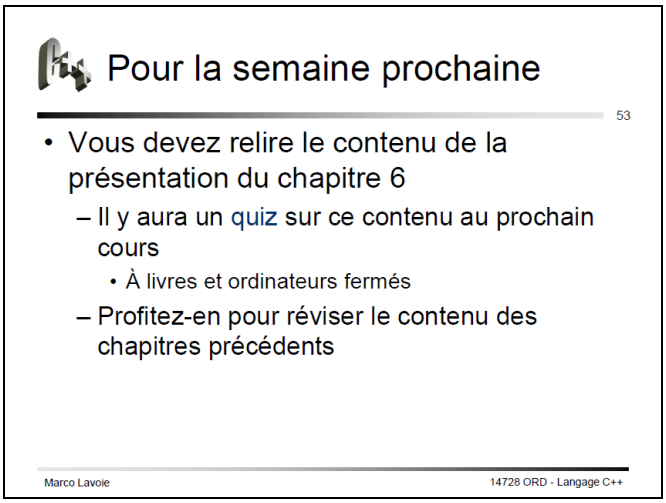
- Pour la clarté et la meilleure lisibilité d'un programme, n'utilisez chaque identificateur d'accès qu'une seule fois dans une définition de classe. Placez les membres **`public`** en premier, car le lecteur est généralement plus intéressé à connaître l'interface de la classe que son fonctionnement internes (c.à.d. ses membres privés).
- Utilisez le nom du fichier avec un caractère de soulignement en remplacement du point comme constante symbolique dans les directives de précompilation **`#ifndef`** et **`#define`** d'un fichier d'en-tête ou de code source d'une classe.
- Toujours fournir un accesseur ainsi qu'un mutateur à la classe pour chacun de ses attributs membres privés.



Ce devoir exige de votre part beaucoup de travail, autant du point de vu de la compréhension des spécifications que de la programmation de sa solution. Conséquemment, débutez votre travail le plus tôt possible.

Vous devez implanter une classe nommée **`Rationnel`** représentant les nombres rationnels (c.à.d. avec un numérateur et un dénominateur). Votre classe doit donc disposer de deux attributs membres privés, **`numerateur`** et **`denominateur`**, ainsi que d'une fonction membre privée réduisant la fraction à son dénominateur le plus petit via l'algorithme d'Euclide. Les mutateurs des attributs membres privés peuvent faire appel à cette fonction privée lorsque leur attribut correspondant est modifié par le client (c.à.d. le programme principal ou une autre fonction membre de la classe).

N'hésitez pas à demander de l'aide à l'instructeur au besoin.



En début de classe la semaine prochaine vous aurez à répondre à des questions sur le C++ sans consultation du matériel pédagogique. Vous êtes donc fortement encouragé à relire les notes de cours du chapitre 6.

Profitez-en pour réviser le contenu des chapitres précédents.