IFT159
Analyse et programmation
Thème 8 — Introduction aux types abstraits

Gabriel Girard

Département d'informatique
UNIVERSITÉ DE SHERBROOKE

9 novembre 2015

Thème 8 — Introduction aux types abstraits

Introduction aux types abstraits

Types d'abstraction

- Abstraction procédurale
  On utilise une fonction sans se préoccuper de son implémentation.
  Nous avons fait de l'abstraction procédurale.

Types d'abstraction

- Abstraction procédurale
  On utilise une fonction sans se préoccuper de son implémentation.
  Nous avons fait de l'abstraction procédurale.
- Abstraction de données
  On utilise les données sans se préoccuper de son implémentation
  Nous avons utilisé des types abstraits (`float, int, string`).

Approches pour résoudre un problème

- Abstraction selon le traitement

Approches pour résoudre un problème

- Abstraction selon le traitement
- Abstraction selon les données à manipuler
  Pourquoi ?
  La solution d'un problème n'est-elle pas un processus qui consiste à transformer des données en entrée en données en sortie.

Approche vue jusqu'à maintenant

- On extrait de la spécification

Approche vue jusqu'à maintenant

- On extrait de la spécification
  - Les données en entrée et en sortie (représentées en termes de *int*, *float*, *char*, agrégats, ...)

Approche vue jusqu'à maintenant

- On extrait de la spécification
  - Les données en entrée et en sortie (représentées en termes de *int*, *float*, *char*, agrégats, ...)
  - Les modules qui opèrent sur les données

Approche vue jusqu'à maintenant

- On extrait de la spécification
  - Les données en entrée et en sortie (représentées en termes de *int*, *float*, *char*, agrégats, ...)
  - Les modules qui opèrent sur les données
- **Problème**
  Avec la complexité croissante des logiciels, cette approche, utilisée seule, n'est plus toujours adéquate.
  Exemple : `etudiant.cours[i].trav[j]`

Abstraction de données

- On veut créer des nouveaux types pour lesquels les détails de l'implantation sont cachés.

Abstraction de données

- On veut créer des nouveaux types pour lesquels les détails de l'implantation sont cachés.
- Nous avons vu :

Abstraction de données

- On veut créer des nouveaux types pour lesquels les détails de l'implantation sont cachés.
- Nous avons vu :
  - Les types énumérés (enum).

Abstraction de données

- On veut créer des nouveaux types pour lesquels les détails de l'implantation sont cachés.
- Nous avons vu :
  - Les types énumérés (enum).
  - Les types structurés (struct).

Abstraction de données

- On veut créer des nouveaux types pour lesquels les détails de l'implantation sont cachés.
- Nous avons vu :
  - Les types énumérés (enum).
  - Les types structurés (struct).
- C'est insuffisant ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !

Abstraction de données

- Insuffisant car cela ne cache pas la structure complexe de certains types de données.
  Exemple : `etudiant[k].cours[i].trav[j]`

Abstraction de données

- Insuffisant car cela ne cache pas la structure complexe de certains types de données.
  Exemple : `etudiant[k].cours[i].trav[j]`
- Il faut que :

Abstraction de données

- Insuffisant car cela ne cache pas la structure complexe de certains types de données.
  Exemple : `etudiant[k].cours[i].trav[j]`
- Il faut que :
  - les données soient vues comme un tout ;

Abstraction de données

- Insuffisant car cela ne cache pas la structure complexe de certains types de données.
  Exemple : `etudiant[k].cours[i].trav[j]`
- Il faut que :
  - les données soient vues comme un tout ;
  - les opérations soient intégrées (les détails de l'implantation sont cachés).

Abstraction de données

- Insuffisant car cela ne cache pas la structure complexe de certains types de données.
  Exemple : `etudiant[k].cours[i].trav[j]`
- Il faut que :
  - les données soient vues comme un tout ;
  - les opérations soient intégrées (les détails de l'implantation sont cachés).
- On doit faire de l'encapsulation.

Définition

- Définition : abstraction de données.
  C'est le processus qui permet d'identifier dans un problème les données nécessaires, leurs propriétés ainsi que leurs opérations.

Définition

- Définition : abstraction de données.
  C'est le processus qui permet d'identifier dans un problème les données nécessaires, leurs propriétés ainsi que leurs opérations.
- Cela permet d'isoler ce qui concerne une donnée et d'implémenter un modèle comme une entité séparée.

Définition

- Définition : abstraction de données.
  C'est le processus qui permet d'identifier dans un problème les données nécessaires, leurs propriétés ainsi que leurs opérations.
- Cela permet d'isoler ce qui concerne une donnée et d'implémenter un modèle comme une entité séparée.
- On développe une vue logique des données.

Définition

- Définition : abstraction de données.
  C'est le processus qui permet d'identifier dans un problème les données nécessaires, leurs propriétés ainsi que leurs opérations.
- Cela permet d'isoler ce qui concerne une donnée et d'implémenter un modèle comme une entité séparée.
- On développe une vue logique des données.
- On devra ensuite implanter la « vue physique » .

Exemple 1 : le type « float »

- Les membres :

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.
  - La caractéristique (exposant)

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.
  - La caractéristique (exposant)
  - Le signe de la mantisse

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.
  - La caractéristique (exposant)
  - Le signe de la mantisse
- Les opérations :

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.
  - La caractéristique (exposant)
  - Le signe de la mantisse
- Les opérations :
  - $+, -, *, /$

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.
  - La caractéristique (exposant)
  - Le signe de la mantisse
- Les opérations :
  - $+, -, *, /$
  - $=$

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.
  - La caractéristique (exposant)
  - Le signe de la mantisse
- Les opérations :
  - $+, -, *, /$
  - $=$
  - $==, <=, >=, <, >, !=$

Exemple 2 : une « Collection »

- Les membres :

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.
  - Les éléments.

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.
  - Les éléments.
  - La capacité maximale de la collection.

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.
  - Les éléments.
  - La capacité maximale de la collection.
- Les opérations :

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.
  - Les éléments.
  - La capacité maximale de la collection.
- Les opérations :
  - Ajouter.

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.
  - Les éléments.
  - La capacité maximale de la collection.
- Les opérations :
  - Ajouter.
  - Retirer.

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.
  - Les éléments.
  - La capacité maximale de la collection.
- Les opérations :
  - Ajouter.
  - Retirer.
  - Taille.

Exemple 2 : une « Collection »

- Les membres :
  - Nombre d'éléments.
  - Les éléments.
  - La capacité maximale de la collection.
- Les opérations :
  - Ajouter.
  - Retirer.
  - Taille.
  - ...

Type abstrait de données

- Les opérations représentent la fenêtre d'accès (ce sont des fonctions du langage).

Type abstrait de données

- Les opérations représentent la fenêtre d'accès (ce sont des fonctions du langage).
- Les membres sont généralement inaccessibles directement.

Type abstrait de données

- Les opérations représentent la fenêtre d'accès (ce sont des fonctions du langage).
- Les membres sont généralement inaccessibles directement.
- Un **type abstrait de données** se divise donc en deux parties distinctes.

Type abstrait de données

- Les opérations représentent la fenêtre d'accès (ce sont des fonctions du langage).
- Les membres sont généralement inaccessibles directement.
- Un **type abstrait de données** se divise donc en deux parties distinctes.
  - La spécification (domaine et opérations permises).

Type abstrait de données

- Les opérations représentent la fenêtre d'accès (ce sont des fonctions du langage).
- Les membres sont généralement inaccessibles directement.
- Un **type abstrait de données** se divise donc en deux parties distinctes.
  - La spécification (domaine et opérations permises).
  - L'implantation.

Introduction aux types abstraits

Type abstraits en C++

- En C++, on défini un type abstrait grâce aux classes (`class`).

Type abstraits en C++

- En C++, on défini un type abstrait grâce aux classes (`class`).
- Les classes permettent de définir de nouveaux types.

Type abstraits en C++

- En C++, on défini un type abstrait grâce aux classes (`class`).
- Les classes permettent de définir de nouveaux types.
- Une variable définie à partir d'un de ces nouveaux types s'appelle un **objet** (ou instance).

Type abstraits en C++

- En C++, on défini un type abstrait grâce aux classes (`class`).
- Les classes permettent de définir de nouveaux types.
- Une variable définie à partir d'un de ces nouveaux types s'appelle un **objet** (ou instance).
- Une opération d'une classe s'appelle une **méthode.**

Exemple 1 : le type « Compteur »

- Exemple 1 :
  On veut définir un type de donnée abstrait qui est un compteur.

Exemple 1 : type Compteur

Analyse/conception

- Information

Exemple 1 : type Compteur

Analyse/conception

- Information
  - La valeur du compteur (entier)

Exemple 1 : type Compteur

Analyse/conception

- Information
  - La valeur du compteur (entier)
  - Des bornes d'utilisation

Exemple 1 : type Compteur

Analyse/conception

- Information
  - La valeur du compteur (entier)
  - Des bornes d'utilisation
- Opérations

Exemple 1 : type Compteur

Analyse/conception

- Information
  - La valeur du compteur (entier)
  - Des bornes d'utilisation
- Opérations
  - Initialisation

Exemple 1 : type Compteur

Analyse/conception

- Information
  - La valeur du compteur (entier)
  - Des bornes d'utilisation
- Opérations
  - Initialisation
  - Incrémentation

Exemple 1 : type Compteur

Analyse/conception

- Information
  - La valeur du compteur (entier)
  - Des bornes d'utilisation
- Opérations
  - Initialisation
  - Incrémentation
  - Décrémentation

Exemple 1 : type Compteur

Analyse/conception

- Information
  - La valeur du compteur (entier)
  - Des bornes d'utilisation
- Opérations
  - Initialisation
  - Incrémentation
  - Décrémentation
  - Accès à la valeur courante

Type Compteur – Implantation – compteur.h

```
class Compteur
{
      const int MAX_CPT = 100 ;
      const int MIN_CPT = -100 ;
      int valeur ;
   public:
      //initialisation du compteur
      void initialise() ;
      //incremente le compteur
      void incremente() ;
      //decremente le compteur
      void decremente() ;
      // valeur courante du compteur
      int valeur_courante() ;
};
```

Type Compteur – Implantation – compteur.cpp

```
/*****************************************
  Implantation du type abstrait Compteur
*****************************************/
#include "Compteur.h"
#include <iostream>

//Initialisation du compteur
void Compteur::initialise()
{
    valeur = 0 ;
}
```

Type Compteur – Implantation

```
// Incrementation du compteur
void Compteur::incremente()
{
   if (valeur < MAX_CPT)
   {
      valeur++ ;
   }
   else
   {
      cerr << "Debordement du compteur. " ;
      cerr << "Incrementation ignoree" << endl ;
   }
}
```

Type Compteur – Implantation

```
// Decrementation du compteur
void Compteur::decremente()
{
    if (valeur > MIN_CPT)
    {
        valeur-- ;
    }
    else
    {
        cerr << "Debordement du compteur. " ;
        cerr << "Decrementation ignoree" << endl ;
    }
}
```

Type Compteur – Implantation

```
//Valeur courante du compteur
int Compteur::valeur_courante()
{
    return valeur ;
}

// Fin du fichier Compteur.cpp
```

Introduction aux types abstraits

Syntaxe des classes

Syntaxe de la partie définition :

```
class nom_de_la_classe
{
```

- partie privée de la classe

```
public:
```

```
};
```

Syntaxe des classes

Syntaxe de la partie définition :

`class` nom_de_la_classe
{

- partie privée de la classe
- déclaration des types, variables et constantes

`public:`

};

Syntaxe des classes

Syntaxe de la partie définition :

```
class nom_de_la_classe
{
```

- partie privée de la classe
- déclaration des types, variables et constantes
- déclaration des méthodes utilitaires

```
public:




};
```

Syntaxe des classes

Syntaxe de la partie définition :

`class` nom_de_la_classe
{

- partie privée de la classe
- déclaration des types, variables et constantes
- déclaration des méthodes utilitaires

`public:`

- partie publique de la classe

};

Syntaxe des classes

Syntaxe de la partie définition :

`class` nom_de_la_classe
{

- partie privée de la classe
- déclaration des types, variables et constantes
- déclaration des méthodes utilitaires

`public:`

- partie publique de la classe
- interface de la classe

};

Syntaxe des classes

Syntaxe de la partie définition :

`class` nom_de_la_classe
{

- partie privée de la classe
- déclaration des types, variables et constantes
- déclaration des méthodes utilitaires

`public:`

- partie publique de la classe
- interface de la classe
- déclaration des types, variables, constantes et méthodes

};

Terminologies et restrictions

- Fonctions membres (méthodes)

Terminologies et restrictions

- Fonctions membres (méthodes)
- Données membres (membres ou attributs)

Terminologies et restrictions

- Fonctions membres (méthodes)
- Données membres (membres ou attributs)
- Les données privées membres ne sont accessibles qu'aux méthodes membres.

Terminologies et restrictions

- Fonctions membres (méthodes)
- Données membres (membres ou attributs)
- Les données privées membres ne sont accessibles qu'aux méthodes membres.
- Une méthode non membre ne peut accéder directement aux données membres (si elles sont privées).

Syntaxe des classes

Syntaxe de la partie implantation (méthode membre) :

type nom_de_la_classe : :*nom_de_la_méthode(par. formels)*
{
corps de la méthode
}

- Chaque méthode est préfixée du nom de la classe

Syntaxe des classes

Syntaxe de la partie implantation (méthode membre) :

type nom_de_la_classe : :*nom_de_la_méthode(par. formels)*
{
corps de la méthode
}

- Chaque méthode est préfixée du nom de la classe
- Opérateur de résolution de portée

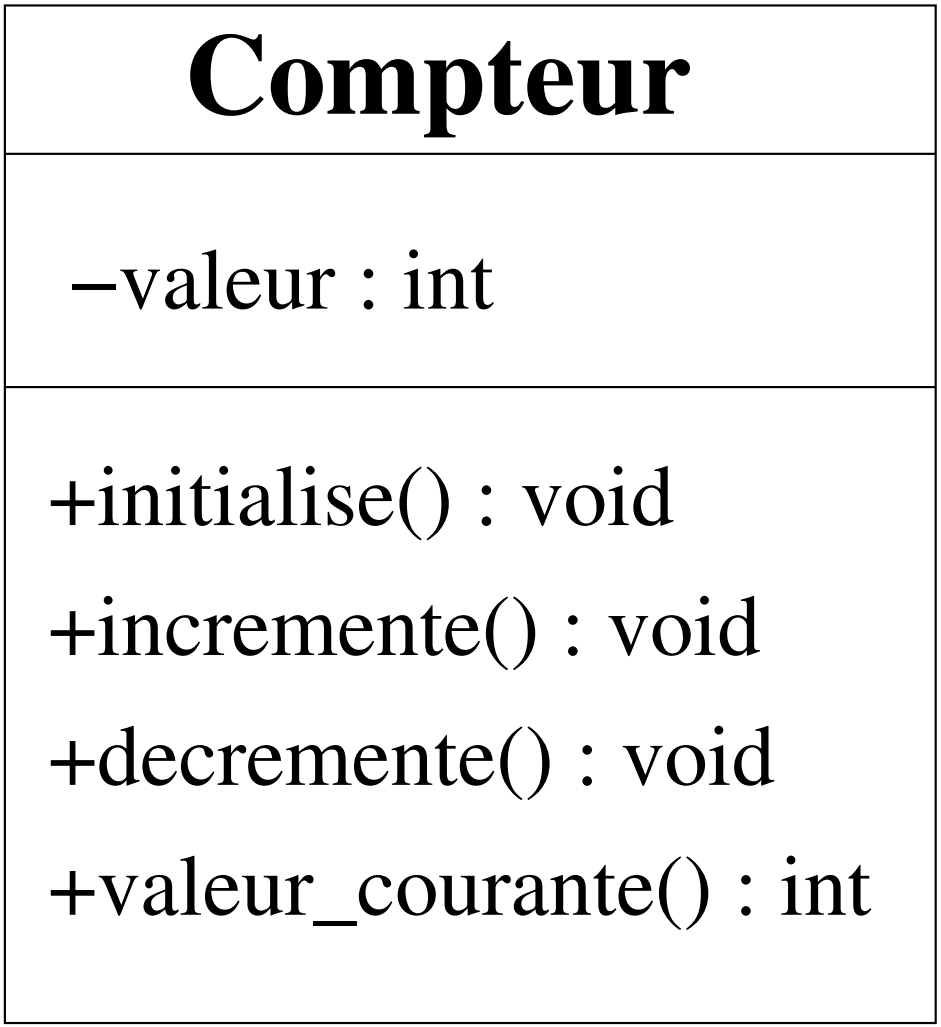
Syntaxe des classes

Syntaxe de la partie implantation (méthode membre) :

type nom_de_la_classe : :*nom_de_la_méthode(par. formels)*
{
corps de la méthode
}

- Chaque méthode est préfixée du nom de la classe
- Opérateur de résolution de portée
- Une méthode membre a accès à la partie privée

Diagramme de classe (UML) : classe Compteur



Utilisation de la classe Compteur

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

int main()
{
    //variables locales
    Compteur cpt1 ;
    Compteur cpt2 ;

    cpt1.initialise() ;
    cpt2.initialise() ;
```

Utilisation de la classe Compteur — suite

```
    for (int i = 0 ; i < 10 ; i++)
    {
        cpt1.incremente() ;
        cpt2.decremente() ;
        cout << "Valeur du premier compteur: "
             << cpt1.valeur_courante() << endl
             << "Valeur du second compteur: "
             << cpt2.valeur_courante() << endl <<
                endl ;
    }

    return EXIT_SUCCESS;
}
```

Terminologies et conventions

- On déclare deux objets (variables) de type Compteur.

Terminologies et conventions

- On déclare deux objets (variables) de type Compteur.
- L'appel d'une méthode est préfixé du nom de l'objet.

Terminologies et conventions

- On déclare deux objets (variables) de type Compteur.
- L'appel d'une méthode est préfixé du nom de l'objet.
- L'opérateur d'accès « . » sert de délimiteur.

Terminologies et conventions

- On déclare deux objets (variables) de type Compteur.
- L'appel d'une méthode est préfixé du nom de l'objet.
- L'opérateur d'accès « . » sert de délimiteur.
- L'utilisation d'une méthode demeure la même que celle des fonctions normales.

Terminologies et conventions

- On déclare deux objets (variables) de type Compteur.
- L'appel d'une méthode est préfixé du nom de l'objet.
- L'opérateur d'accès « . » sert de délimiteur.
- L'utilisation d'une méthode demeure la même que celle des fonctions normales.
- Par convention :

Terminologies et conventions

- On déclare deux objets (variables) de type Compteur.
- L'appel d'une méthode est préfixé du nom de l'objet.
- L'opérateur d'accès « . » sert de délimiteur.
- L'utilisation d'une méthode demeure la même que celle des fonctions normales.
- Par convention :
  - Le fichier « .h » contient la définition de la classe.

Terminologies et conventions

- On déclare deux objets (variables) de type Compteur.
- L'appel d'une méthode est préfixé du nom de l'objet.
- L'opérateur d'accès « . » sert de délimiteur.
- L'utilisation d'une méthode demeure la même que celle des fonctions normales.
- Par convention :
  - Le fichier « .h » contient la définition de la classe.
  - Le fichier « .cpp » contient l'implantation des méthodes membres.

Terminologie

- Objet = instance d'une classe

Terminologie

- Objet = instance d'une classe
- Par abus de langage on utilise « variable » .

Introduction aux types abstraits

Multiples définitions

- Risque de charger plusieurs fois le même fichier.

Multiples définitions

- Risque de charger plusieurs fois le même fichier.
- Risque de définitions multiples.

Multiples définitions

- Risque de charger plusieurs fois le même fichier.
- Risque de définitions multiples.
- Pour éviter la recompilation et la redéfinition : on utilise des instructions au précompilateur.

Multiples définitions

- Risque de charger plusieurs fois le même fichier.
- Risque de définitions multiples.
- Pour éviter la recompilation et la redéfinition : on utilise des instructions au précompilateur.
- #ifndef et #endif accompagné par #define.

Exemple : la type Compteur - compteur.h

```
#ifndef COMPTEUR_H // nom du fichier (classe)
#define COMPTEUR_H

class Compteur
{
      int valeur ;
  public:
      //initialisation du compteur
      void initialise() ;
      //incremente le compteur
      void incremente() ;
      //decremente le compteur
      void decremente() ;
      // valeur courante du compteur
      int valeur_courante() ;
};
#endif // COMPTEUR_H
```

Introduction aux types abstraits

Exemple 1

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

int main()
{
  Compteur cpt1, cpt2 ;

  cpt1.initialise() ;
  cpt2.initialise() ;

  cpt1.valeur = 10 ;  // erreur d'acces
  cpt2.valeur = -10 ; // erreur d'acces
  for (int i = 0 ; i < 10 ; i++)
  {
    cpt1.incremente() ;
    cpt2.decremente() ;
    cout << "cpt1 = " << cpt1.valeur_courante()
         << "cpt2 = " << cpt2.valeur_courante();
  }
} // Fin du main
```

Exemple 2

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

int main()
{
  void atteindre(int, Compteur&) ;
  Compteur cpt1, cpt2 ;

  cpt1.initialise() ;
  cpt2.initialise() ;

  atteindre(10, cpt1) ;
  atteindre(-10, cpt2) ;
  for (int i = 0 ; i < 10 ; i++)
  {
    cpt1.incremente() ;
    cpt2.decremente() ;
    cout << "cpt1 = " << cpt1.valeur_courante()
         << "cpt2 = " << cpt2.valeur_courante();
   }
}
```

Exemple 2

```
void atteindre(int init, Compteur& cpt)
{
    int nombre ;
    nombre = cpt.valeur_courante() ;
    if(nombre < init)
    {
        while(nombre < init)
        {
            cpt.incremente() ;
            nombre = cpt.valeur_courante() ;
        }
    }
    else
    {
        while(init < nombre)
        {
            cpt.decremente() ;
            nombre = cpt.valeur_courante() ;
        }
    }
}
```

Introduction aux types abstraits

Comparaison classe – structure

- Une structure et une classe peuvent contenir des fonctions et des données membres.

Comparaison classe – structure

- Une structure et une classe peuvent contenir des fonctions et des données membres.
- Les membres peuvent être privés ou publiques.

Comparaison classe – structure

- Une structure et une classe peuvent contenir des fonctions et des données membres.
- Les membres peuvent être privés ou publiques.
- Par défaut tout est publique dans une structure.

Comparaison classe – structure

- Une structure et une classe peuvent contenir des fonctions et des données membres.
- Les membres peuvent être privés ou publiques.
- Par défaut tout est publique dans une structure.
- Par défaut tout est privé dans une classe.

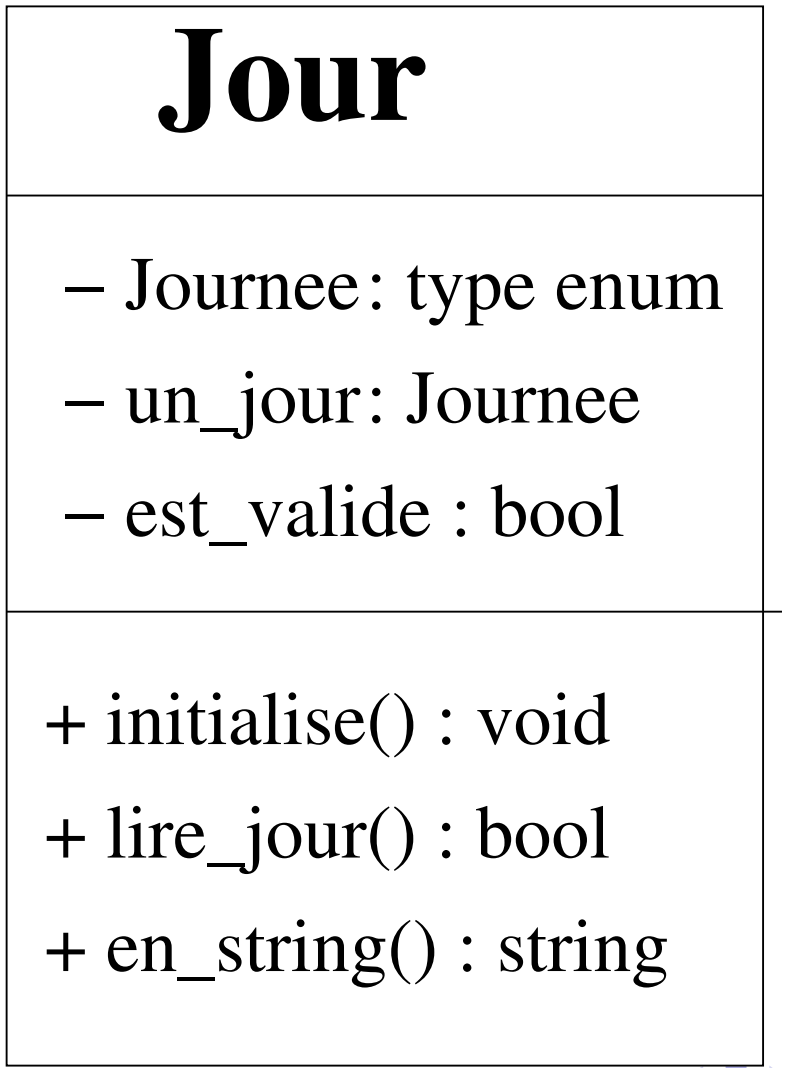
Introduction aux types abstraits

Exemple : le type abstrait « Jour »

On veut définir le type de donnée abstrait jour.

Nous avions déjà défini un type énuméré jour.
On ne peut cependant pas lire ou écrire des variables de ce type.

Type Jour : diagramme de classe



Type jour (Utilisation)

```
#include <iostream>
#include <string>
#include "jour.h"

int main()
{
    Jour aujourdhui ;
    bool est_un_jour_valide ;

    aujourdhui.initialise() ;

    cout << "Quel jour sommes nous? " ;
    est_un_jour_valide = aujourdhui.lire_jour() ;
```

Type jour (Utilisation)

```
    if (! est_un_jour_valide)
         cerr << "Jour incorrect" << endl ;
    else
    {
    cout << "Aujourd'hui, nous sommes "
         << aujourdhui.en_string()
         << "." << endl ;
    }

    return 0;
}
```

Type jour (définition)

```
// fichier jour.h
#ifndef JOUR_H
#define JOUR_H
#include <string>
#include <iostream>
#include <cctype>

class Jour
{
   enum Journee { aucun=-1, dimanche, lundi, mardi,
                  mercredi, jeudi, vendredi,
                  samedi} ;
   Journee un_jour ;
   bool est_valide ;
```

Type jour (définition)

```
  public:
    void initialise() ;

    // Cette fonction lit les 2 premiers caracteres
    // au clavier et en fait une journee
    bool lire_jour() ;

    // Cette fonction convertit le nom d'une
        journee
    // en une chaine de caractere.
    string en_string() ;
} ;
#endif
```

Type jour (implantation)

```
// fichier jour.cpp
#include "jour.h"

void Jour::initialise()
{
    est_valide = false ;
}
```

Type jour (implantation)

```
bool Jour::lire_jour()
{
    char lettre1, lettre2 ;

    cout << "Donner les 2 premiers caracteres: ";
    cin >> lettre1 >> lettre2 ;

    //Conversion en majuscules
    lettre1 = toupper(lettre1) ; // necessite cctype
    lettre2 = toupper(lettre2) ;

    est_valide = true ;
```

Type jour (implantation)

```
switch (lettre1)
{
    case 'D' : un_jour = dimanche ;
               break ;
    case 'L' : un_jour = lundi ;
               break ;
    case 'J' : un_jour = jeudi ;
               break ;
    case 'V' : un_jour = vendredi ;
               break ;
    case 'S' : un_jour = samedi ;
               break ;
```

Type jour (implantation)

```
        case 'M': switch (lettre2)
                  {
                   case 'A' : un_jour = mardi ;
                              break ;
                   case 'E' : un_jour = mercredi ;
                              break ;
                   default  : est_valide = false ;
                              un_jour = aucun ;
                  }
                   break ;
        default : est_valide = false ;
                  un_jour = aucun ;
      }
    return valide ;
}
```

Type jour (implantation)

```
string Jour::en_string() {
  string jour;
  if (est_valide)
     switch (un_jour) {
      case dimanche: jour = "Dimanche"; break ;
      case lundi :   jour = "Lundi";    break ;
      case mardi :   jour = "Mardi";    break ;
      case mercredi: jour = "Mercredi"; break ;
      case jeudi :   jour = "Jeudi";    break ;
      case vendredi: jour = "Vendredi"; break ;
      case samedi :  jour = "Samedi";
      }
  else {
     cerr << "Jour invalide." << endl ;
     jour = "*** non defini ***";
  }

  return jour ;
}
```

Introduction aux types abstraits

Initialisation automatique : constructeur

Pour initialiser un objet, on peut utiliser

1. Une fonction d'initialisation.
   (ex. : `cpt.initialise()`).

Initialisation automatique : constructeur

Pour initialiser un objet, on peut utiliser

1. Une fonction d'initialisation.
   (ex. : `cpt.initialise()`).
2. Un constructeur.

Initialisation automatique : constructeur

Pour initialiser un objet, on peut utiliser

1. Une fonction d'initialisation.
   (ex. : `cpt.initialise()`).
2. Un constructeur.
   - C'est une fonction membre de la classe dont le nom est **obligatoirement** celui de la classe.

Initialisation automatique : constructeur

Pour initialiser un objet, on peut utiliser

1. Une fonction d'initialisation.
   (ex. : `cpt.initialise()`).
2. Un constructeur.
   - C'est une fonction membre de la classe dont le nom est **obligatoirement** celui de la classe.
   - Cette fonction est appelée automatiquement.

Initialisation automatique : constructeur

Pour initialiser un objet, on peut utiliser

1. Une fonction d'initialisation.
   (ex. : `cpt.initialise()`).
2. Un constructeur.
   - C'est une fonction membre de la classe dont le nom est **obligatoirement** celui de la classe.
   - Cette fonction est appelée automatiquement.
   - Elle remplace la fonction d'initialisation.

Classe Compteur : définition

```
// Fichier compteur.h
#ifndef COMPTEUR_H
#define COMPTEUR_H
class Compteur
{
      const int MAX_CPT = 100 ;
      const int MIN_CPT = -100 ;
      int valeur ;
  public:
      Compteur() ;
      void incremente() ;
      void decremente() ;
      int valeur_courante() ;
} ;
#endif
```

Classe Compteur : implantation

```
// Fichier compteur.cpp
#include "Compteur.h"

//Constructeur
Compteur::Compteur()
{
    valeur = 0 ;
}
//etc...
```

```
// Programme utilisant le compteur
#include "Compteur.h"

int main()
{
  Compteur cpt1;

  ...
```

Introduction aux types abstraits

Multiples constructeurs

1. Il est possible qu'une classe ait plusieurs constructeurs.

Multiples constructeurs

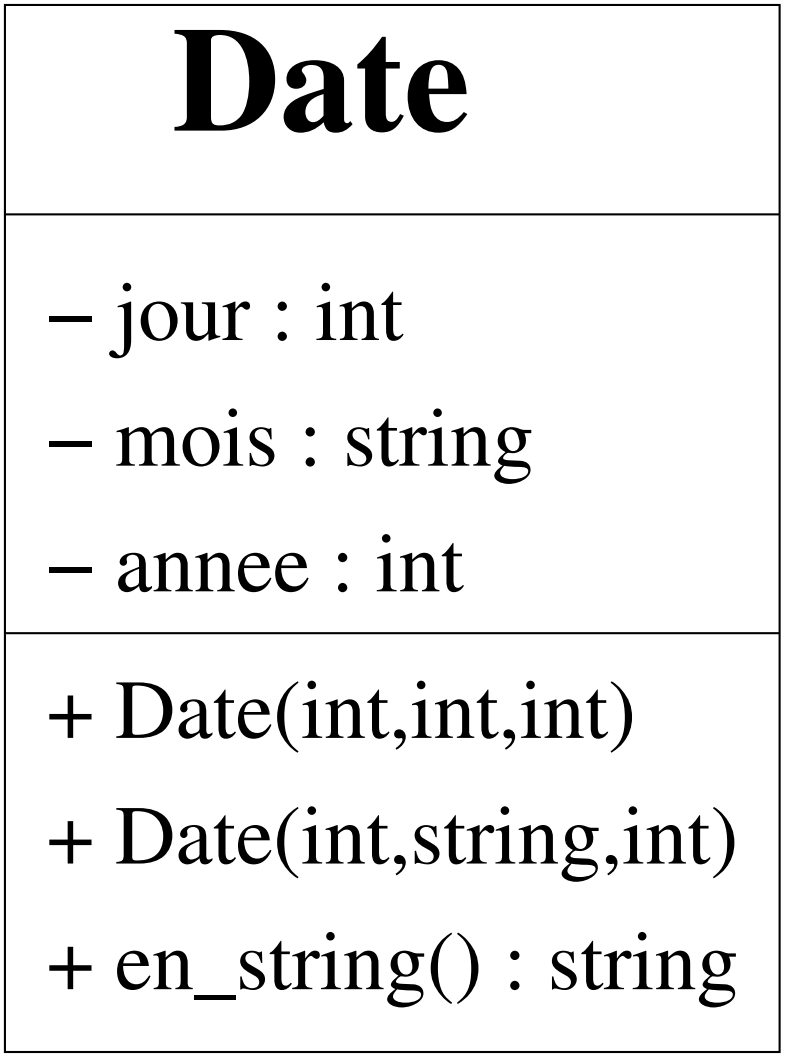
1. Il est possible qu'une classe ait plusieurs constructeurs.

2. Pourquoi ? ? ? ?
   Un objet possède une seule représentation interne mais peut posséder plusieurs représentations externes.
   Exemple : la date (23/10/2002 ou 23 octobre 2002)

Exemple : la classe « Date »

- Exemple 3 :
  On veut définir le type de donnée abstrait date.

Type Date : diagramme de classe



Classe Date : utilisation

```
int main()
{
    Date hier (12, 11, 2015)
    Date demain (14, "novembre", 2015);
    Date noel {25, "decembre", 2015};

    cout << "Hier etait le " << hier.en_string();
    cout << endl << endl;

    cout << "Demain sera le " << demain.en_string();
    cout << endl << endl;

    return EXIT_SUCCESS;
}
```

Classe Date : définition - date.h

```
#include <iostream>
#include <string>

class Date
{
        int jour ;
        string mois;
        int annee ;
  public:
        Date(int, int, int) ;
        Date(int, string, int) ;
        string en_string() ;
} ;
```

Classe date : implantation

```
Date::Date(int j, int m, int a)  {
   jour = j ;
   switch (m)
   {
      case 1  : mois = "Janvier" ;    break ;
      case 2  : mois = "Fevrier" ;    break ;
      case 3  : mois = "Mars" ;       break ;
      case 4  : mois = "Avril" ;      break ;
      case 5  : mois = "Mai" ;        break ;
      case 6  : mois = "Juin" ;       break ;
      case 7  : mois = "Juillet" ;    break ;
      case 8  : mois = "Aout" ;       break ;
      case 9  : mois = "Septembre" ; break ;
      case 10 : mois = "Octobre" ;    break ;
      case 11 : mois = "Novembre" ;   break ;
      case 12 : mois = "Decembre" ;   break ;
      default : cerr << "Mois invalide" << endl ;
                mois = "?????????" ;
   }
   annee = a ;}
```

Classe date : implantation

```
Date::Date(int j , string m, int a)
{
        jour = j ;
        mois = m ;
        annee = a ;
}

string Date::en_string()
{
   return convertir_entier_en_string(jour) +
          " " + mois + " " +
          convertir_entier_en_string(annee) ;
}
```

Multiples constructeurs

- Il est impératif de pouvoir distinguer entre les constructeurs.

Multiples constructeurs

- Il est impératif de pouvoir distinguer entre les constructeurs.
- Les signatures des constructeurs doivent être différentes, i.e. :

Multiples constructeurs

- Il est impératif de pouvoir distinguer entre les constructeurs.
- Les signatures des constructeurs doivent être différentes, i.e. :
  - un nombre de paramètres différents ou

Multiples constructeurs

- Il est impératif de pouvoir distinguer entre les constructeurs.
- Les signatures des constructeurs doivent être différentes, i.e. :
  - un nombre de paramètres différents ou
  - des types des paramètres différents.

Multiples constructeurs

- Il faut avoir un constructeur sans paramètre si l'on veut définir un objet date sans spécifier de valeurs d'initialisation.

```
// Dans date.h ...
Date() ;

// Dans date.cpp
Date::Date()
{
      jour = 1 ;
      mois = "Janvier" ;
      annee = 1900 ;
}

// Declaration
Date siecle ;
```

Introduction aux types abstraits

Destruction automatique : destructeur

- Il est exécuté automatiquement lorsque l'on sort de la portée de l'objet.

Destruction automatique : destructeur

- Il est exécuté automatiquement lorsque l'on sort de la portée de l'objet.
- Utile principalement lorsque lorsqu'on utilise des ressources qui doivent être libérées manuellement.

  Exemple : mémoire allouée dynamiquement

Destruction automatique : destructeur

- Il est exécuté automatiquement lorsque l'on sort de la portée de l'objet.
- Utile principalement lorsque lorsqu'on utilise des ressources qui doivent être libérées manuellement.

  Exemple : mémoire allouée dynamiquement
- Donc peu utile pour nous (pour le moment).

Destruction automatique : destructeur

- Il est exécuté automatiquement lorsque l'on sort de la portée de l'objet.
- Utile principalement lorsque lorsqu'on utilise des ressources qui doivent être libérées manuellement.

  Exemple : mémoire allouée dynamiquement
- Donc peu utile pour nous (pour le moment).
- Nom de la classe précédé d'un « $\sim$ » .

Exemple de destructeur

```
// Dans date.h ...
~Date() ;

// Dans date.cpp
Date::~Date()
{
      // Liberation des ressources...
}
```