IFT159
Analyse et programmation
Chapitre 10 — Introduction aux types abstraits

Gabriel Girard

Département d'informatique
UNIVERSITÉ DE SHERBROOKE

11 novembre 2008

Chapitre 10 — 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.
- 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
- 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
  - 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'implémentation 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]`
- Il faut que :
  - les données soient vues comme un tout ;
  - les opérations soient intégrées (les détails de l'implémentation 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.
- Cela permet d'isoler ce qui concerne une donnée et d'en implémenter unAbstraction de données modèle comme une entité séparée.
- On développe une vue logique des données.
- On devra ensuite implémenter la « vue physique » .

Exemple 1 : le type « float »

- Les membres :
  - La mantisse.
  - La caractéristique.
- Les opérations :
  - $+, -, *, /$
  - $=$
  - $==, <=, >=, <, >, !=$

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).
- 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'implémentation.

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**.
- 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
  - La valeur du compteur (entier)
  - Des bornes d'utilisation
- Opérations
  - Initialisation
  - Incrémentation
  - Décrémentation
  - Obtention de la valeur courante

Type Compteur – Implémentation - compteur.h

```
/*  Definition du type abstrait Compteur */

#define MAX_INT 100
#define MIN_INT -100
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() ;
};
```


Type Compteur – Implémentation - compteur.cpp

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

//Initialisation du Compteur

void Compteur::initialise()
{
        valeur = 0 ;
}
```

Type Compteur – Implémentation

```
//Incrementation du Compteur

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


Type Compteur – Implémentation

```
//Decrementation du Compteur

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


Type Compteur – Implémentation

```
//Valeur courante du Compteur

int Compteur::valeur_courante()
{
        return valeur ;
}

// Fin du fichier Compteur.cpp
```


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)
- Données membres (membres)
- 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).

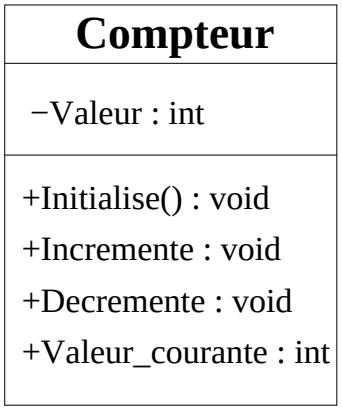
Syntaxe des classes

Syntaxe de la partie implémentation (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.h>
#include "Compteur.h"

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

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

Utilisation de la classe « Compteur »

```
  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 ;
  }
}
```


Termonologies 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 « .cc » ou « .cpp » contient l'implémentation des méthodes membres.

Terminologie

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

Multiples définitions

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

Exemple : la type Compteur - compteur.h

```
#ifndef COMPTEUR_H // nom de la 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
```


Exemple 1

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

void main()
{
   Compteur cpt1 ;
   Compteur cpt2 ;

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

   cpt1.valeur = 10 ;
   cpt2.valeur = -10 ;
```


Exemple 1

```
   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 ;
   }

} // Fin du main
```


Exemple 2

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

void main()
{
  void position_cpt(int, Compteur&) ;

  Compteur cpt1 ;
  Compteur cpt2 ;

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

Exemple 2

```
    position_cpt(10, cpt1);
    position_cpt(-10, cpt2) ;

    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 ;
    }

  } // Fin du main
```


Exemple 2

```
void position_cpt(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() ;
       }
}
```


Comparaison classe – structure

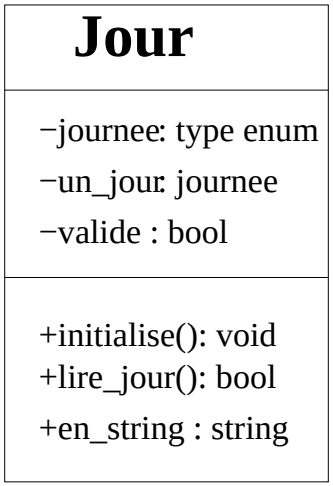
- Une structure peut contenir des fonctions et des données membres.
- Les membres peuvent être privés ou publiques.
- Par défaut tout est publique dans une structure.
- Par défaut tout est privé dans une classe.

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"

using namespace std;

int main()
{
   Jour aujourdhui ;
   bool bonjour ;

   aujourdhui.initialise() ;

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


Type jour (Utilisation)

```
   if (!bonjour)
      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 Type_journee { aucun=-1, dimanche, lundi, mardi,
                       mercredi, jeudi, vendredi, samedi};
   Type_journee un_jour ;
   bool valide ;
```

Type jour (définition)

```
  public:
     void initialise() ;
     // Les fcts retournent faux si detectent une erreur

     // fct qui lit les 2 premiers carac. d'une journee
     bool lire_jour() ;

     //fct qui convertit une journee en chaine de car.
     string en_string() ;
};

#endif
```


Type jour (implémentation)

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

void jour::initialise()
{
        valide = false ;
}
```


Type jour (implémentation)

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

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

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

   valide = true ;
```


Type jour (implémentation)

```
switch (lettre1){
   case 'D' : un_jour = dimanche ;
              break ;
   case 'L' : un_jour = lundi ;
              break ;
   case 'M': switch (lettre2)
             {
               case 'A' : un_jour = mardi ;
                          break ;
               case 'E' : un_jour = mercredi ;
                          break ;
               default  : valide = false ;
                          un_jour = aucun ;
             }
             break ;
```

Type jour (implémentation)

```
        case 'J' : un_jour = jeudi ;
                   break ;
        case 'V' : un_jour = vendredi ;
                   break ;
        case 'S' : un_jour = samedi ;
                   break ;
        default  : valide = false ;
                   un_jour = aucun ;
     }
   return valide ;
}
```


Type jour (implémentation)

```
string Jour::en_string() {
   string jour;
   if (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;
}
```


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
#define MAX_INT 100
#define MIN_INT -100

class Compteur
{
        int valeur ;
    public:
        Compteur() ;
        void incremente() ;
        void decremente() ;
        int valeur_courante() ;
} ;
#endif
```

Classe Compteur : implémentation

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


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

etc..
```


Multiples constructeurs

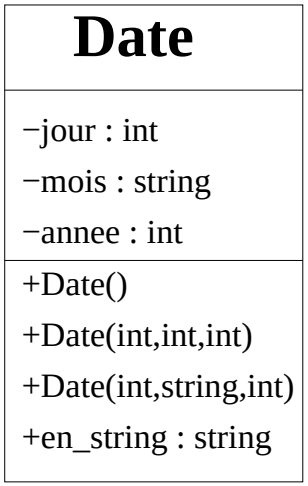
1. Il est possible qu'une classe ait plusieurs constructeurs.

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

```
void main()
{
        Date hier(12, 11, 2008) ;
        Date demain(14, "novembre", 2008) ;

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

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


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 : implémentation

```
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 : implémentation

```
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.
- Les constructeurs doivent avoir
  - 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.

```
Date() ;

Date::Date()
{
     jour = 1 ;
     mois =  "Janvier" ;
     annee = 1900 ;
}

Date siecle ;
```


Destruction automatique : destructeur

- Il est exécuté automatiquement lorsque l'on sort de la portée de l'objet.
- Utile principalement lorsque l'espace de l'objet est créé via l'allocation dynamique de mémoire.
- Donc peu utile pour nous (pour le moment).
- Nom de la classe précédé d'un « $\sim$ » .