Introduction au C/C++

Cours 6

Rémi Watrigant
(fortement inspiré du cours de M. Huchard de l'Université Montpellier 2)

Université de Nîmes

2013-2014

Présentation des cours qui restent :

Objectifs : Eclairage sur les aspects de la programmation par objets en C++

- introduction à l'orienté objets
- utilisation de C++
- classes, objets
- héritage, polymorphisme

Sommaire du cours 6

- introduction à la programmation par objets
- application sur C++

Programmation par objets

- But : construire un système informatique
- Méthode : modélisation des entités
  - aspects statiques
  - aspects dynamiques

  $\Rightarrow$ intéraction de ces entités

Par opposition à une approche de décomposition des fonctionnalités en procédures plus petites

Avantages :

- efficacité de conception
- efficacité de maintenance et évolution

Bref historique des langages à objets

- début dans les années 60, 70 : Smalltalk
- années 80 : Objective C, C++, Eiffel, CLOS, Ada
- années 90 : Python, Java
- années 2000 : C#, PHP 5...

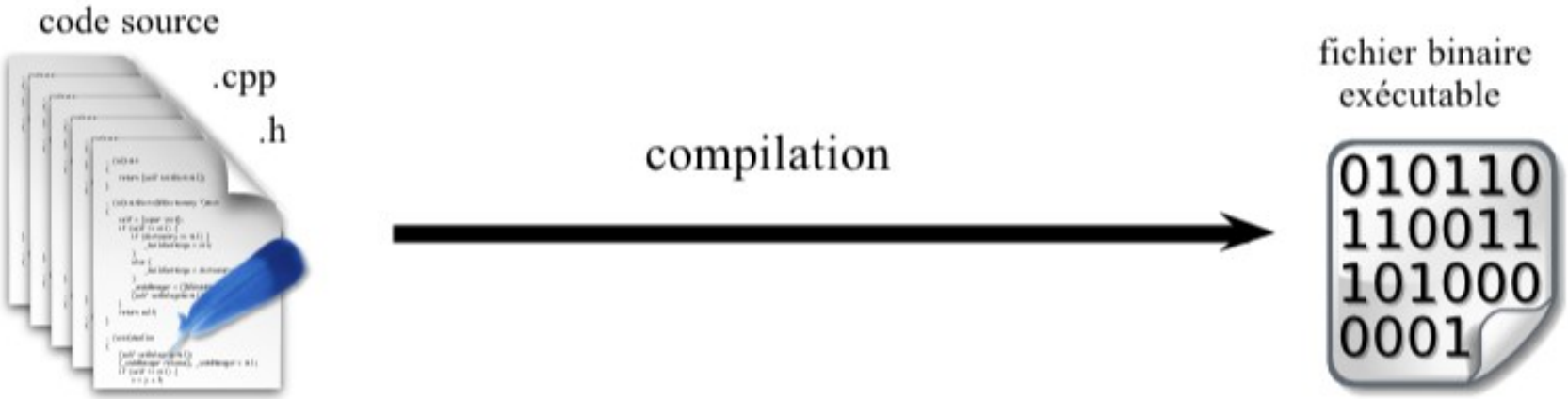
Différences : langages "tout objet" ou non, typage lourd ou faible, paradigme impératif ou fonctionnel, langage à classes ou prototypes...

Bref historique de C++

- auteur : Bjarne Stroustrup (né en 1950 au Danemark)
- inspiration du langage Simula 67
- *C with classes* (1979)
- C++ (1983)
- normalisation ANSI/ISO en 1998 mise à jour en 2003 **et 2011** : C++11

Le langage C++

Langage compilé :



Compilateur principalement utilisé : g++

Programmation par objets

Cinq concepts fondateurs

- objet
- message
- classe
- spécialisation/généralisation et héritage
- polymorphisme

Programmation par objets

Objet

Programmation par objets

Objet

Représentation informatique d'une entité du domaine sur lequel porte le système (ex. un compte bancaire particulier).
Caractérisé par :

- partie **statique** : informations descriptives, données, état. (ex. une voiture se décrit par sa marque, modèle, année, couleur...etc.)
- partie **dynamique** : ensemble d'opérations qui décrivent les actions de l'objet (ex. une voiture peut se démarrer, s'arrêter...etc).
- une identité, propre à chaque objet et permettant de le distinguer d'un autre sans ambiguïté.

Programmation par objets

Message

Programmation par objets

Message

Unité de communication entre les objets.
Version simple : un message envoyé à un objet correspond à une opération possible sur cet objet et qui sera invoquée.
Ex. invoquer l'opération d'avancer sur une voiture.

Programmation par objets

Classe

Programmation par objets

Classe

Abstraction (regroupement) d'un ensemble d'objets ayant une structure et un comportement commun.
$\Rightarrow$ correspond au "moule" d'un ensemble d'objets $\equiv$ concept du système.
Ex. la classe *Véhicule* représentera tous les véhicules du système

- Aspect extensionnel : l'ensemble des objets (ou instances) représentés par la classe
- Aspect intensionnel : description commune à tous les objets de la classe (partie statique et dynamique)

Programmation par objets

Spécialisation/généralisation et héritage

Programmation par objets

Spécialisation/généralisation et héritage

- point de vue extensionnel : sous ensemble d'objets
  Ex. parmi les véhicules, il y a des voitures, motos...etc.
- point de vue intensionnel : partage et réutilisation des attributs et opérations
  Ex. une *Voiture* contient tous les attributs et méthodes d'un *Véhicule*

Programmation par objets

Polymorphisme

Programmation par objets

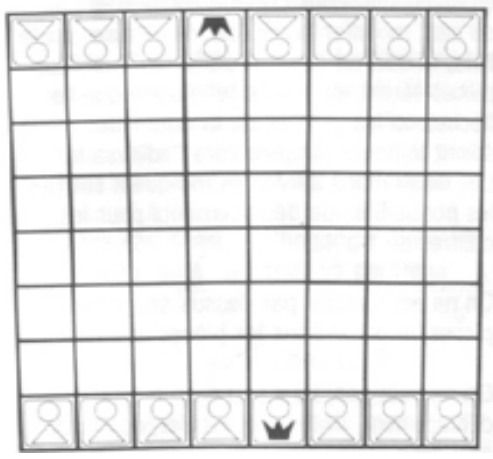
Polymorphisme

- Permet d'écrire des expressions valables pour des objets de différentes classes, y compris des classes qui ne sont pas encore créées... !
- Factorisation de code
- Extensibilité des programmes à objets

Ex. un véhicule a une marque, un modèle, et peut démarrer, s'arrêter, avancer... Ces opérations vont concerner les Voitures, Motos...

Exemple

Shogun



Jeu de pions (pièces)

- deux familles de (7 pions + 1 shogun)

- basé sur la capture de pion

- déplacement rectiligne et à angle droit

- valeur du déplacement suivant déterminée au hasard à chaque arrivée d'un pion sur une case

Exemple

Une **application = un ensemble d'objets** qui ont chacun leur rôle à jouer et qui interagissent

Objets informatique, ici des composants graphiques

- **fenêtre**
  - **barre de menu**
  - **bouton**
  - **zones de texte**
  - **panneau (sur lequel est affiché le jeu)**

Objets « métier »…

Exemple

Objets « métier » :

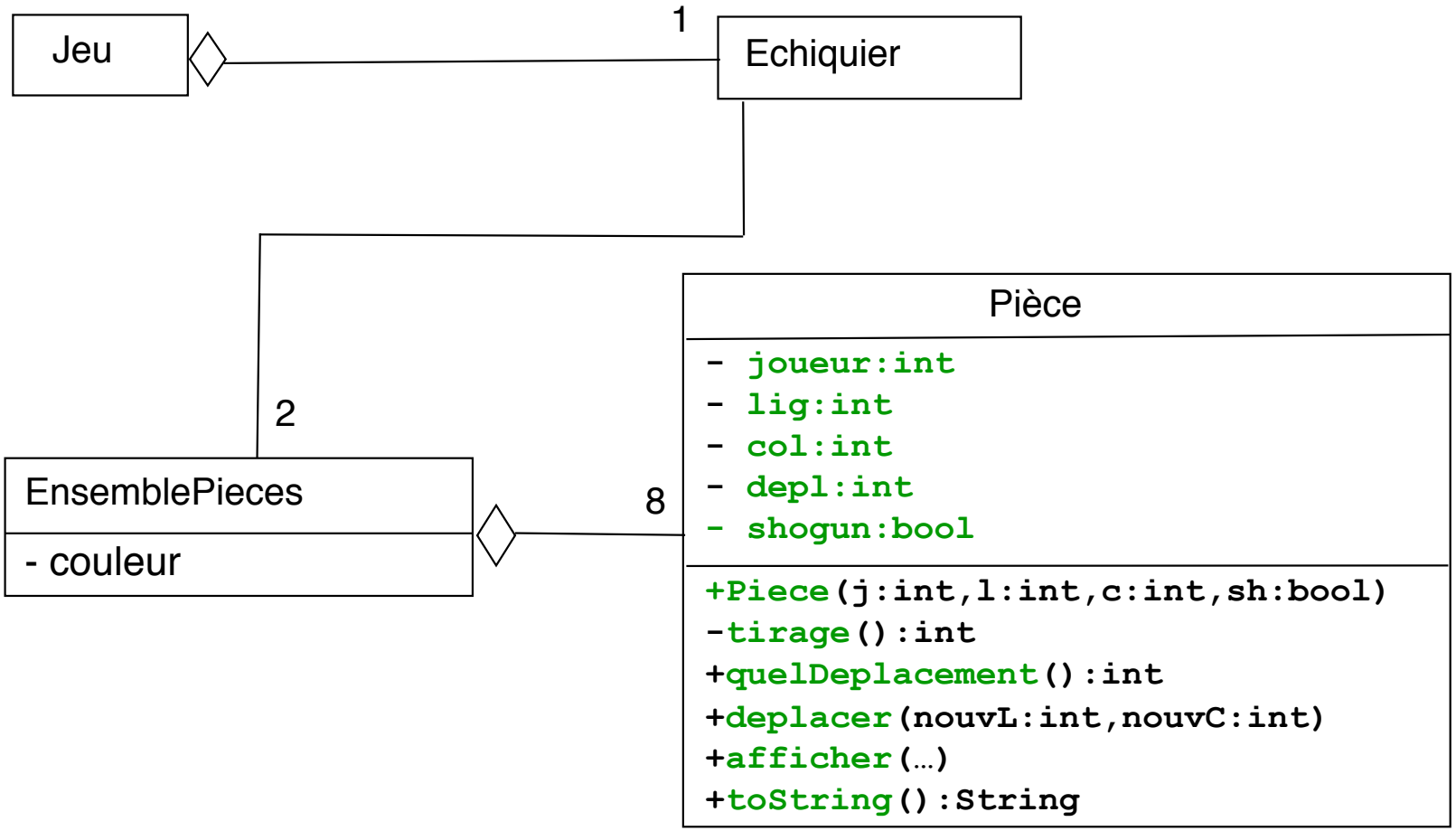
jeu *gère une partie*

- **échiquier** *gère la "grille" elle-même*
  - **ensemble de pièces 1**
    - **pièces**
  - **ensemble de pièces 2**
    - **pièces**

une pièce gère sa position sur l'échiquier
elle sait se déplacer et s'afficher

Exemple

Objets « métier » --> Représentés par des classes



Exemple

Les objets collaborent entre eux, chacun étant **responsable** des actions qu'il effectue

Ex : afficher le jeu

panneau jeu : « jeu, affiche-toi »

jeu : « échiquier, affiche-toi »

échiquier : « ens. pièces 1, affiche-toi »

ens. pièces 1 : à chaque pièce : « pièce, affiche-toi »

« ens. pièces 2, affiche-toi »

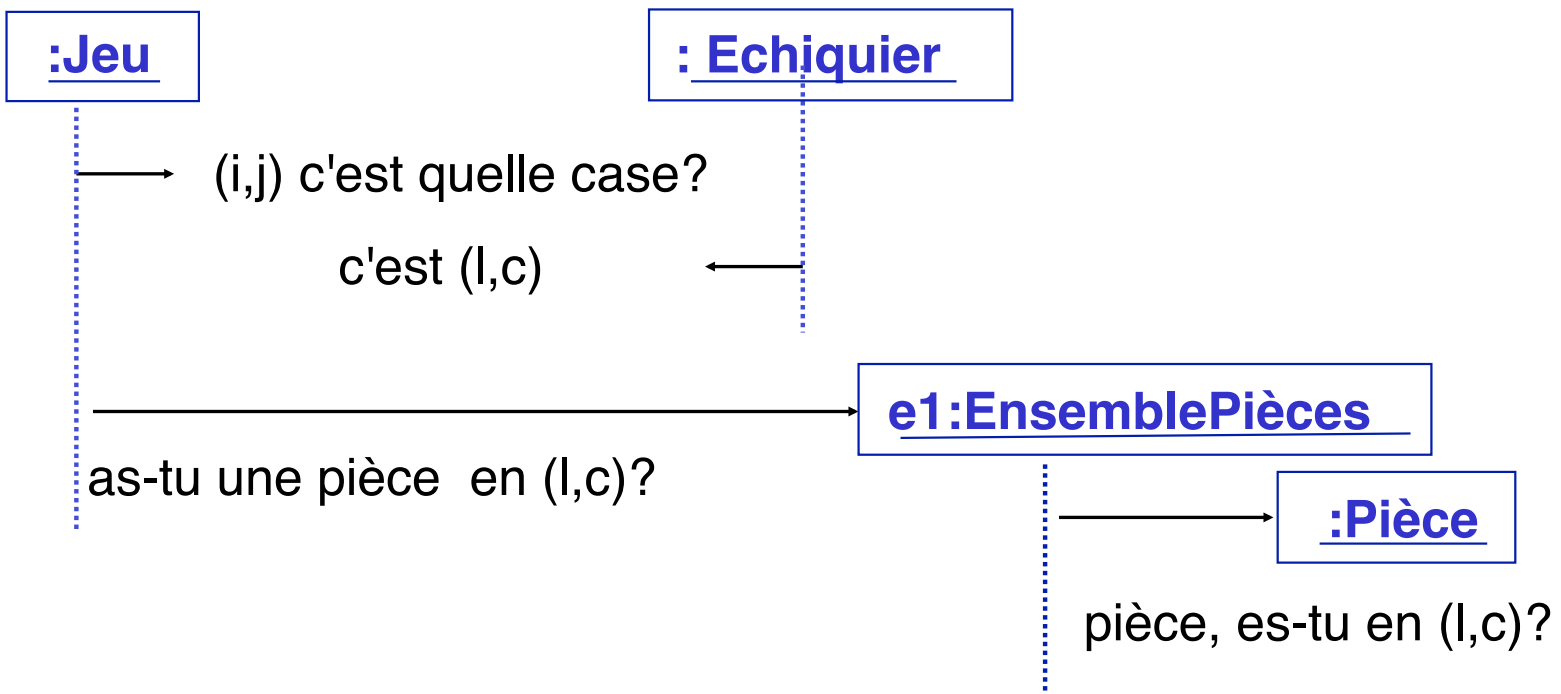
ens. pièces 2 : à chaque pièce : « pièce, affiche-toi »

Exemple

Ex: gérer le début de mouvement du joueur 1

Clic! sur panneau de jeu en (i,j)

jeu est averti



Exemple

Les objets sont des **instances** de **classes**

Une classe est un « modèle » qui définit :

- la structure d'un objet **(ses attributs)**

- et son comportement **(ses méthodes)**

La structure des objets est (+ ou -) **encapsulée**,
ainsi que certaines de leurs méthodes

Exemple

Une classe ordinaire

Partie privée **"encapsulée"**

attributs

+ des méthodes de « cuisine interne »

Partie publique

méthodes de « services offerts »

`private` : accessible seulement dans le corps des méthodes de la classe

`public` : accessible par toute méthode de toute classe

Exemple

Ex: l'objet *jeu* demande à la pièce *p* de se déplacer en case (4,3)

```
p.deplacer(4,3)
```

jeu n'a pas besoin de savoir comment p procède

D'ailleurs, *jeu* n'a pas le droit de modifier directement p :

```
p.lig = 4;
p.col = 3;
```

car col et lig sont encapsulés

C'est p qui modifie son propre état

Exemple

les objets communiquent par **envoi de message :**

```
p.deplacer(4,3)
```

jeu *envoie à* p le *message* "deplacer(4,3)"

jeu est l'expéditeur du message

(on est dans une méthode de jeu)

p est le receveur du message

Le langage C++

Un programme simple est généralement composé de :

- un fichier main.cpp qui contient le point d'entrée du programme : la fonction main :

```
int main(int argc, char *argv[])
{
cout « "hello world !" « endl ;
return 0 ;
}
```

- chaque classe = 2 parties (et 2 fichiers généralement) :
  - un fichier header (extension .h) qui contient l'ossature de la classe (déclaration des attributs et méthodes)
  - un fichier source (extension .cpp) qui contient les définitions des méthodes

L'interface (fichier .h)

Contenu :

- attributs
- déclarations (signatures ou entêtes) des méthodes

Macros classiques dans l'entête :

- `#define MaClasse_h`
- `#ifndef MaClasse_h` ... `#endif`

Permet d'éviter les inclusions récursives, les doubles analyses

L'interface de la classe Personne (fichier Personne.h)

```
#ifndef personne_h
#define personne_h

class Personne{
private:
  char prenom[256];
  char nom[256];
  int telephone;
public:
  Personne();
  Personne(char* prenom, char* nom, int telephone);
  virtual ~Personne();
  virtual char* getPrenom();
  virtual char* getNom();
  virtual int getTelephone();
  virtual void setPrenom(char* prenom);
  virtual void setNom(char* nom);
  virtual void setTelephone(int telephone)
  virtual void afficher();
```

Le fichier Personne.cpp

- contient les corps des méthodes les uns à la suite des autres

Le fichier Personne.cpp

```
#include "Personne.h"

Personne::afficher() {
   printf("Nom: %s, Prenom: %s, Telephone: %d", nom, prenom, telephone)
}

Personne::Personne(char* prenom, char* nom, int int telephone) {
    this->nom = nom;
    this->prenom = prenom;
    this->telephone = telephone;
}
```

This (l'objet receveur)

- Pseudo-variable `this` = désigner l'objet auquel on a envoyé un message pendant l'exécution de la méthode correspondante
- En C++ `this` est un pointeur constant sur l'objet
- Peut accéder aux propriétés (attributs et méthodes) de l'objet.

Exemple :

```
char* Personne::getNom() {
    return this->nom;    // equivaut a return (*this).nom;
}
```

Une classe ordinaire

Partie privée **"encapsulée"**

attributs

+ des méthodes de « cuisine interne »

Partie publique

méthodes de « services offerts »

`private` **: accessible seulement dans le corps des méthodes de la classe**

`public` **: accessible par toute méthode de toute classe**

Encapsulation - Objectifs principaux :

- cacher les détails d'implémentation (pour pouvoir en changer sans influer sur les utilisateurs de la classe)
- pour contrôler les accès à l'état (attributs) des objets et le garder cohérent

En C++ une première approximation :

- `public:` accès depuis n'importe quelle autre partie du code
- `private:` accès depuis des parties de code de la même classe
- `protected:` voir le cours sur l'héritage

classes versus *struct*

- dans les `class` tout est privé par défaut
- dans les `struct` tout est public par défaut

Création des objets

- créer un objet = instancier une classe
- demande de gérer la mémoire (pas de ramasse-miettes comme en Java)
  - avantageux pour les logiciels où la maîtrise de la gestion mémoire est nécessaire (embarqué, scientifique, temps réel, soit très coûteux en espace, soit dans des contextes où on a peu de mémoire)
  - désavantageux pour les logiciels complexes et en constante évolution, où le programmeur doit être déchargé des problèmes de bas niveau

Constructeur

Méthodes spéciales, appelées automatiquement lors de la création d'un objet

- même nom que la classe
- ne retourne rien (même pas *void*)
- contenu :
  - acquisition de ressource (mémoire, ouverture fichier, connexion, etc)
  - initialisation des attributs (jamais automatique en C++)
  - $\longrightarrow$ à vous de le faire !

Création des objets

Plusieurs modes d'allocation :

- automatique
- dynamique

L'allocation automatique

- l'objet est créé localement dans un bloc
- stocké dans la pile
- durée de vie : depuis l'instruction de création jusqu'à la fin du bloc

```
{ // début du bloc
. . .
Personne p("Jean", "Dupont");

p.afficher();

. . .
} // fin du bloc
```

L'allocation dynamique

- L'objet est référencé par un pointeur
- stocké dans le tas
- créé grâce à l'opérateur `new`
- détruit par l'usage de l'opérateur `delete`

```
Personne* p = new Personne("Jean" "Dupont");

p->afficher();

delete p;
```

Rappel - utilisation de pointeurs

`Personne* pp`
`pp` est l'adresse d'un emplacement mémoire contenant une (ou plusieurs) instance(s) de Personne.
`*pp` est :

- une instance de Personne, avec $*$ opérateur de déréférencement
- tableau de Personne d'une taille qui n'est pas encore connue et qui sera alloué dynamiquement, avec par exemple : `Personne* pp = new Personne[nbPersonnes]`

`Personne** pv` est un tableau (ou pointeur) de pointeurs vers des instances de Personne.

Passages de paramètres dans les fonctions

Trois méthodes possibles :

- passage par valeur
- passage par adresse
- passage par référence

Trois formes de passage de paramètres

```
void f(int fi, int* fpi, int &fri){
        fi++;
        (*fpi)++;
        fri++;
}

main(){
     int i = 4;
     int *pi = new int;
     *pi = 4;
     int j=4;

     // i=4  *pi=4  j=4
     f(i, pi, j);

     // i=4  *pi=5  j=5
}
```

Trois formes de passage de paramètres

- passage *par valeur* lors de l'appel `f(i, pi, j)`, la valeur de `i` est copiée dans `fi`, puis `fi` est incrémenté, ce qui est sans effet sur `i`
- passage *par adresse* lors de ce même appel, la valeur de `pi` (qui est l'adresse d'une zone de type entier) est copiée dans `fpi`, puis la valeur pointée par `fpi` est incrémentée, et cette valeur est toujours pointée par `pi` donc apparaît bien modifiée lorsqu'on termine `f`
- passage *par référence* après l'initialisation d'une référence telle que `fri` par une variable telle que `j`, il faut comprendre que `fri` est un alias pour `j`, quand on incrémente `fri` on incrémente donc `j` puisque c'est la même entité avec deux noms différents.

Destructeur

- unique
- même nom que la classe derrière le préfixe composé du caractère ~
- destruction des attributs (dont le type est une classe) qui composent l'objet