Programmation Orientée Objets 2
L3 MI
Cours 1

Rémi Watrigant[1]

[1] basé sur le cours de Marianne Huchard

Présentation du module

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

- utilisation de C++
- classes, objets
- héritage, polymorphisme
- généricité
- librairies standards
- exceptions
- design patterns

Sommaire du cours 1

- rappel sur la programmation par objets
- rappel sur C++

Rappel sur la 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

Rappel sur la programmation par objets

Cinq concepts fondateurs

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

Objet

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.

Message

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.

Classe

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)

Spécialisation/généralisation et héritage

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*

Polymorphisme

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...

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...

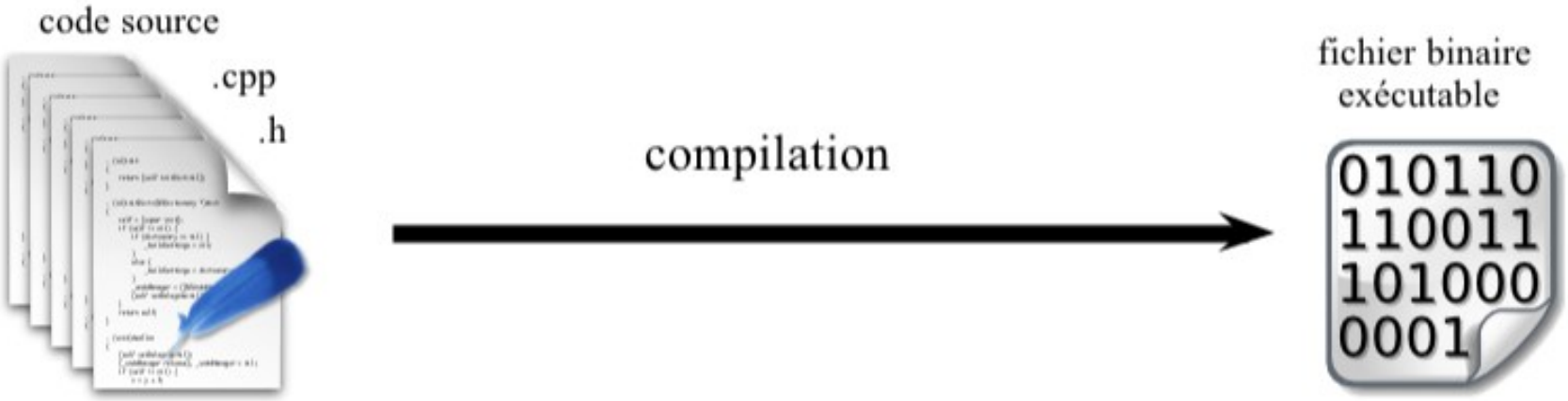
Différences : langages "tout objet" ou non, typage lourd ou faible, paradigme impératif ou fonctionnel, langage à classes ou prototypes...

Brève 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++

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();
 };

#endif
```

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

Création des objets

Trois modes d'allocation :

- statique
- automatique
- dynamique

L'allocation statique

- L'objet est créé lorsque le flot de contrôle atteint l'instruction de création et pour toute la durée du programme.
- exemple : attributs `static`.

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.

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).suiteMots;
}
```

Constructeur

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

- même nom que la classe
- jamais virtuelles
- acquisition de ressource (mémoire, ouverture fichier, connexion, etc.
- initialisation des attributs (jamais automatique en C++)
- constructeur par copie: recopie en profondeur des attributs

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

Programmer en C++

Petite introduction aux makefiles

Pour compiler un programme comportant :

- une classe Personne : Personne.h Personne.cpp
- un fichier main : main.cpp

Lignes de commandes nécessaires :

- création des fichiers .o :
  g++ -c Personne.cpp
  g++ -c main.cpp
- linkage et création de l'exécutable :
  g++ -o monProgramme Personne.o main.o

Modification de Personne.cpp $\Rightarrow$ retaper toutes les lignes de commandes
Programme avec 10 classes (...)

Petite introduction aux makefiles

Programme **make** : permet d'appeler les commandes nécessaires à la création de fichiers.
Fonctionne à l'aide d'un fichier de configuration : **Makefile**

```
   # Indiquer quel compilateur est a utiliser
CPP=g++

# Specifier les options du compilateur
CFLAGS=
LDFLAGS=

#Nom de l'executable
EXEC=nom

# Liste de fichiers objets necessaires pour le programme final
#SRC = fichier.cpp fichier1.cpp fichier2.cpp
OBJ = fichier.o fichier1.o fichier2.o

all: $(EXEC)

# Creation des .o a partir des .cc
%.o : %.c
        $(CPP) $<

# Generation du fichier executable
$(EXEC): $(OBJ)
#       $(CPP) -o $(EXEC) $(OBJ) $(LDFLAGS)

# Deuxieme possibilite ATTENTION : OBJ contenant plusieurs
# dependances, il faut utiliser $^ et non $<
        $(CPP) -o $@ $^
```

Petite introduction aux makefiles

Puis appel au programme make (avec fichier Makefile dans le même répertoire) :

make

Possibilité de définir plusieurs exécutables utilisant différents main, différentes classes...

Pour développer en C++

Plusieurs possibilités :

- votre éditeur de texte préféré (bloc note, vim, emacs...)
  puis compilation "à la main" (ou Makefile)
- utilisation d'un IDE (Environnement de Développement Intégré)
  - code blocks (Windows, Linux)
  - dev-c++ (Windows)
  - Xcode (MacOS)
  - Eclipse (Windows, Linux, MacOS)
  - ...

  Inclut le plus souvent :
  - éditeur de texte
  - gestion d'un workspace
  - génération automatique de Makefile/exécutables
  - auto-complétion
  - debugger
  - lien avec logiciels de versioning
  - ...

à vous !