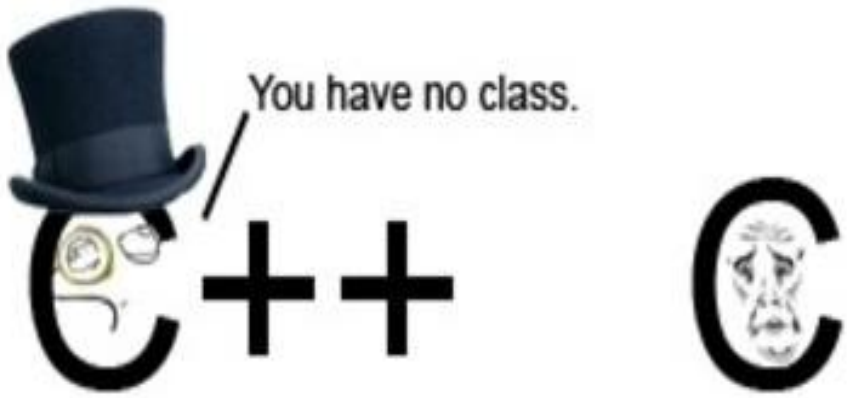
# Programmation Orientée Objet - C++

Bastien DI PIERRO

Université Claude Bernard Lyon 1

25 Mars 2016



1. Historique et philosophie
2. Classe et Objet
3. La surcharge
4. Héritage
5. Objets et pointeurs
6. Polymorphisme
7. Classe abstraite

# Histoire de la POO

C++ :

- 1983
- Bjorne Stroustrup
- Extension du langage C
- But : faire face à la complexité grandissante des logiciels

Depuis :

- Java (Just Another Vague Acronym)
- C# (propriété Microsoft)
- Javascript (Netscape)
- Python (libre, facile, rapide ...)

# Un exemple simple

```
#include <iostream>

using namespace std;

int main()
{
  int i=0;
  double x=1.0;

  for(i=0;i<100;i++)
  {
    x += x/2.0;
    cout<<" iteration "<<i<<" , x vaut : "<<x<<endl;
  }

  return 0;
}
```

# Philosophie

- Le programme est découpé en “brique” élémentaire.
- 1 brique élémentaire = 1 unité logique = 1 objet

# Philosophie

- Le programme est découpé en “brique” élémentaire.
- 1 brique élémentaire = 1 unité logique = 1 objet
- Objet : une idée, un concept, une entité du monde physique (ou imaginaire)
- Exemple : un humain, un animal, un outil, une équation, une statistique, un bouton (GUI), une licorne ...

# Philosophie

- Le programme est découpé en “brique” élémentaire.
- 1 brique élémentaire = 1 unité logique = 1 objet
- Objet : une idée, un concept, une entité du monde physique (ou imaginaire)
- Exemple : un humain, un animal, un outil, une équation, une statistique, un bouton (GUI), une licorne ...
- 1 objet =

# Philosophie

- Le programme est découpé en “brique” élémentaire.
- 1 brique élémentaire = 1 unité logique = 1 objet
- Objet : une idée, un concept, une entité du monde physique (ou imaginaire)
- Exemple : un humain, un animal, un outil, une équation, une statistique, un bouton (GUI), une licorne ...
- 1 objet =
  - une structure propre
    humain $\neq$ équation $\Rightarrow$ pas la même construction
  - un comportement propre
    un humain marche et parle, une voiture roule et pollue

# Philosophie

- Le programme est découpé en “brique” élémentaire.
- 1 brique élémentaire = 1 unité logique = 1 objet
- Objet : une idée, un concept, une entité du monde physique (ou imaginaire)
- Exemple : un humain, un animal, un outil, une équation, une statistique, un bouton (GUI), une licorne ...
- 1 objet =
  - une structure propre
    humain $\neq$ équation $\Rightarrow$ pas la même construction
  - un comportement propre
    un humain marche et parle, une voiture roule et pollue
- Les objets peuvent intéragir entre eux
  ex : un humain conduit une voiture
  une clef anglaise serre un boulon

# Philosophie

## Interacttion entre objets

Permet de concevoir, imaginer et réaliser l'ensemble des tâches du programme

# Philosophie

## Interacttion entre objets

Permet de concevoir, imaginer et réaliser l'ensemble des tâches du programme

$\Rightarrow$ **Algorithme**

# Prog procédurale Vs Prog. Orientée Objet

- Unité logique : module (fonction, routine, ...)
- 1 fonction = 1 problème
- Structuration déscendante

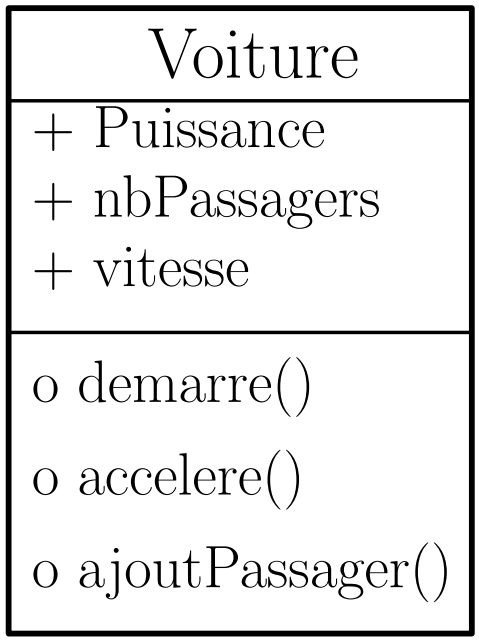
```
double fct1(int x){
   ...
   x = fct2(y);
   ...
}
int fct2(int z){
   fct3();
   ...
}
void fct3(){
   ...
}
```

# Prog procédurale Vs Prog. Orientée Objet

- Unité logique : module (fonction, routine, ...)
- 1 fonction = 1 problème
- Structuration déscendante

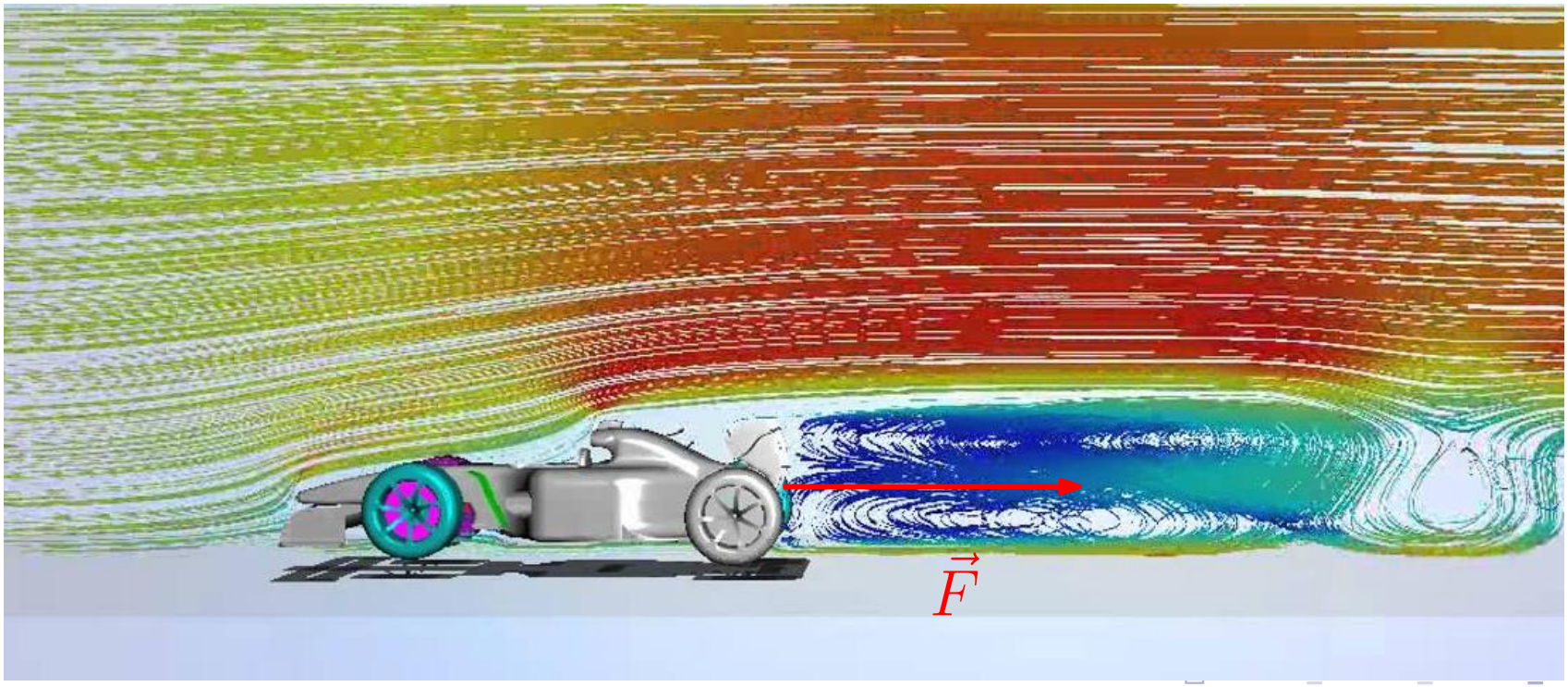
```
double fct1(int x){
   ...
   x = fct2(y);
   ...
}
int fct2(int z){
   fct3();
   ...
}
void fct3(){
   ...
}
```

- Unité logique : Objet
- 1 Objet = 1 concept
- Structuration : Encapsulation



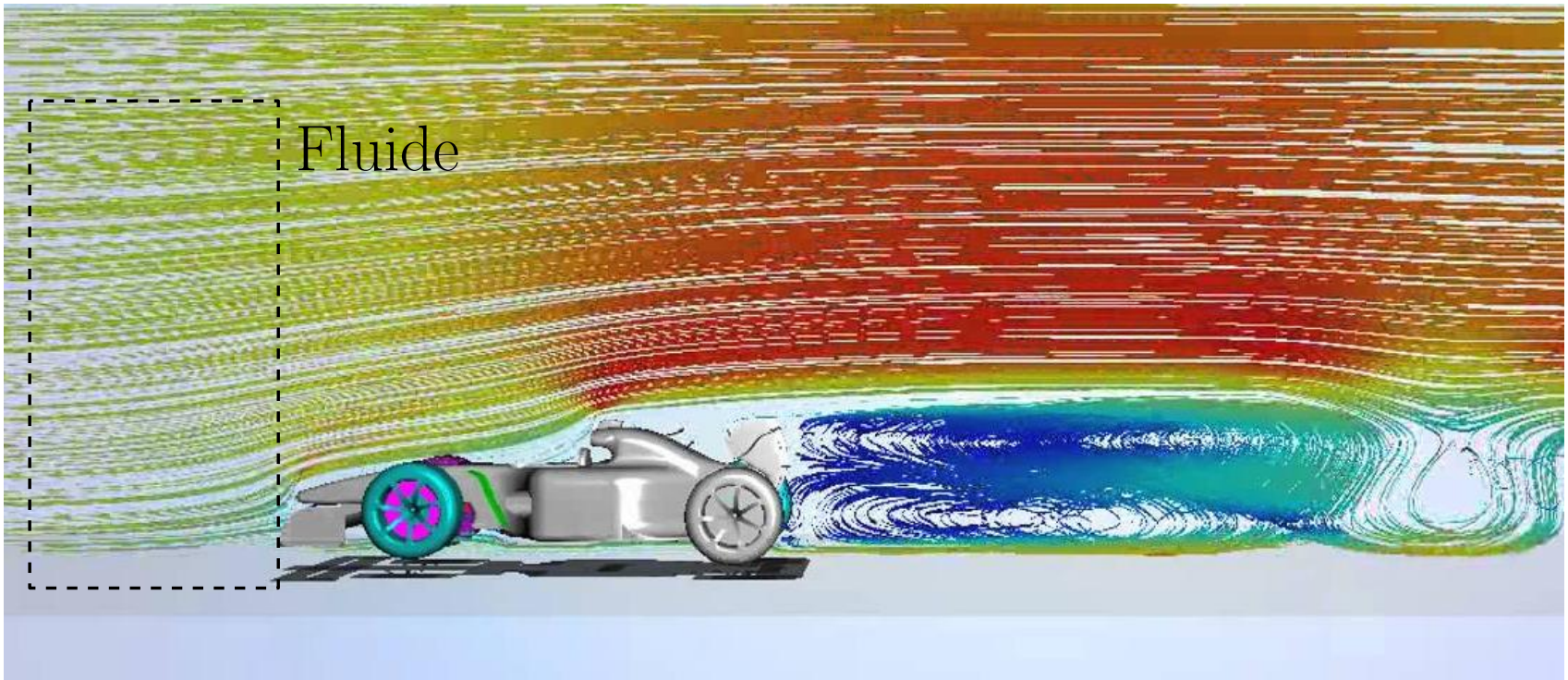
# Fil rouge

- Trainée d'une voiture ?



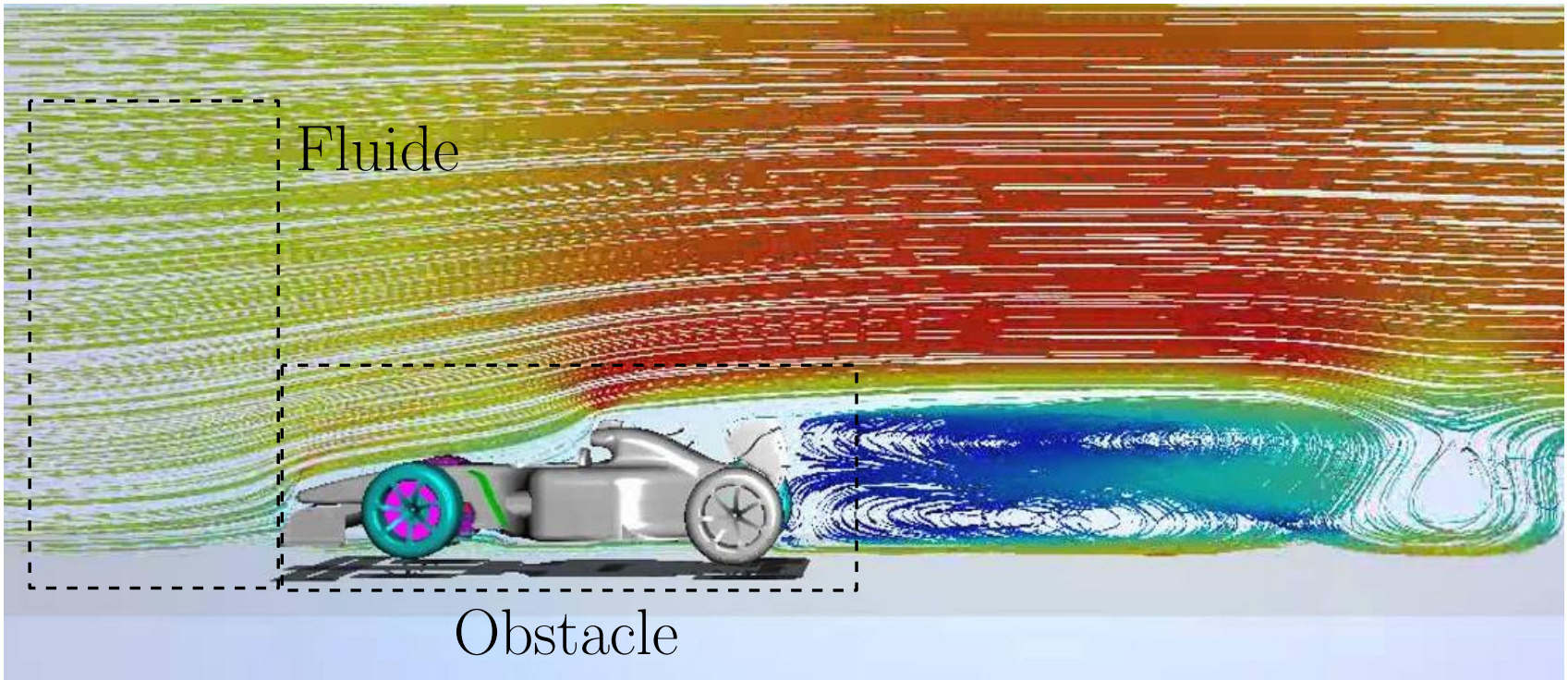
# Fil rouge

- Trainée d'une voiture ?
- Fluide : viscosite $\nu$, densité $\rho$



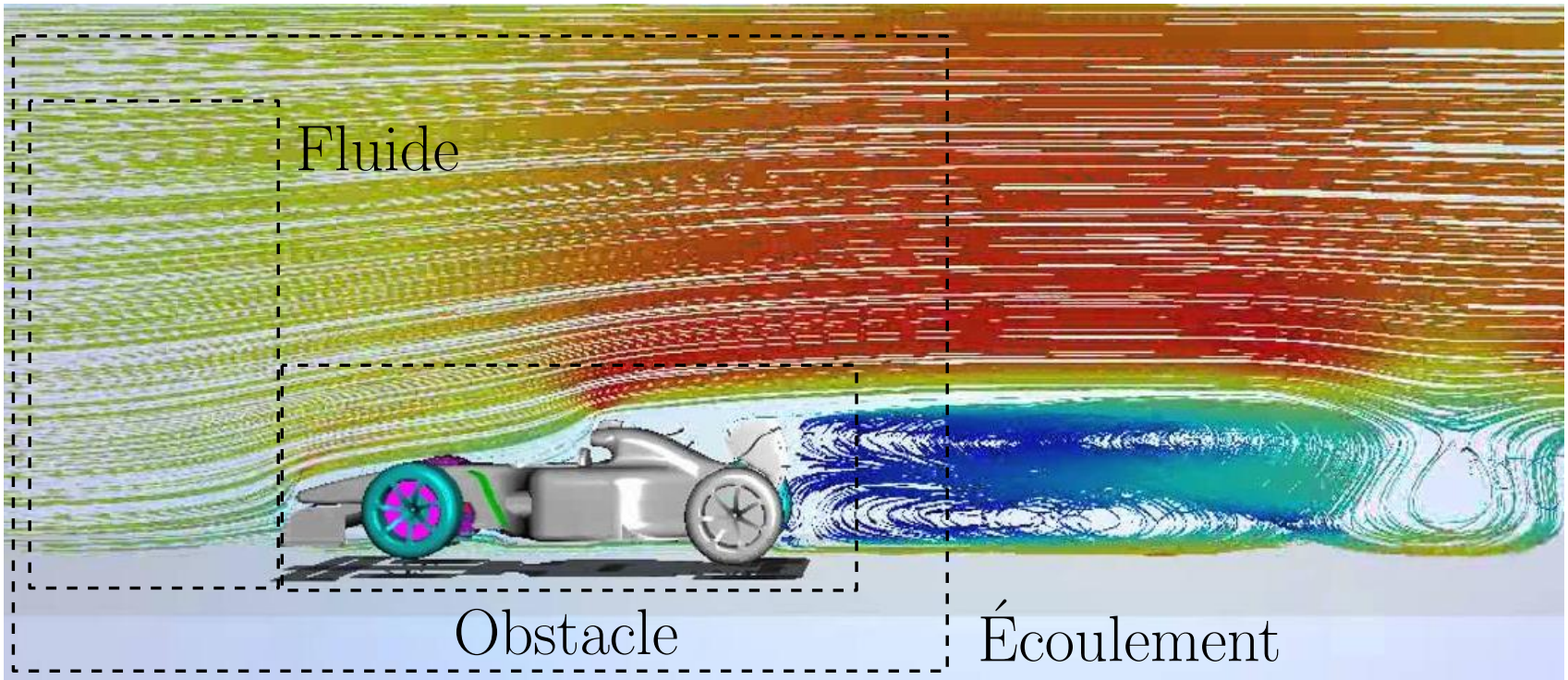
# Fil rouge

- Trainée d'une voiture ?
- Fluide : viscosite $\nu$, densité $\rho$
- Obstacle : vitesse $V$, longueur $L$, surface $S$



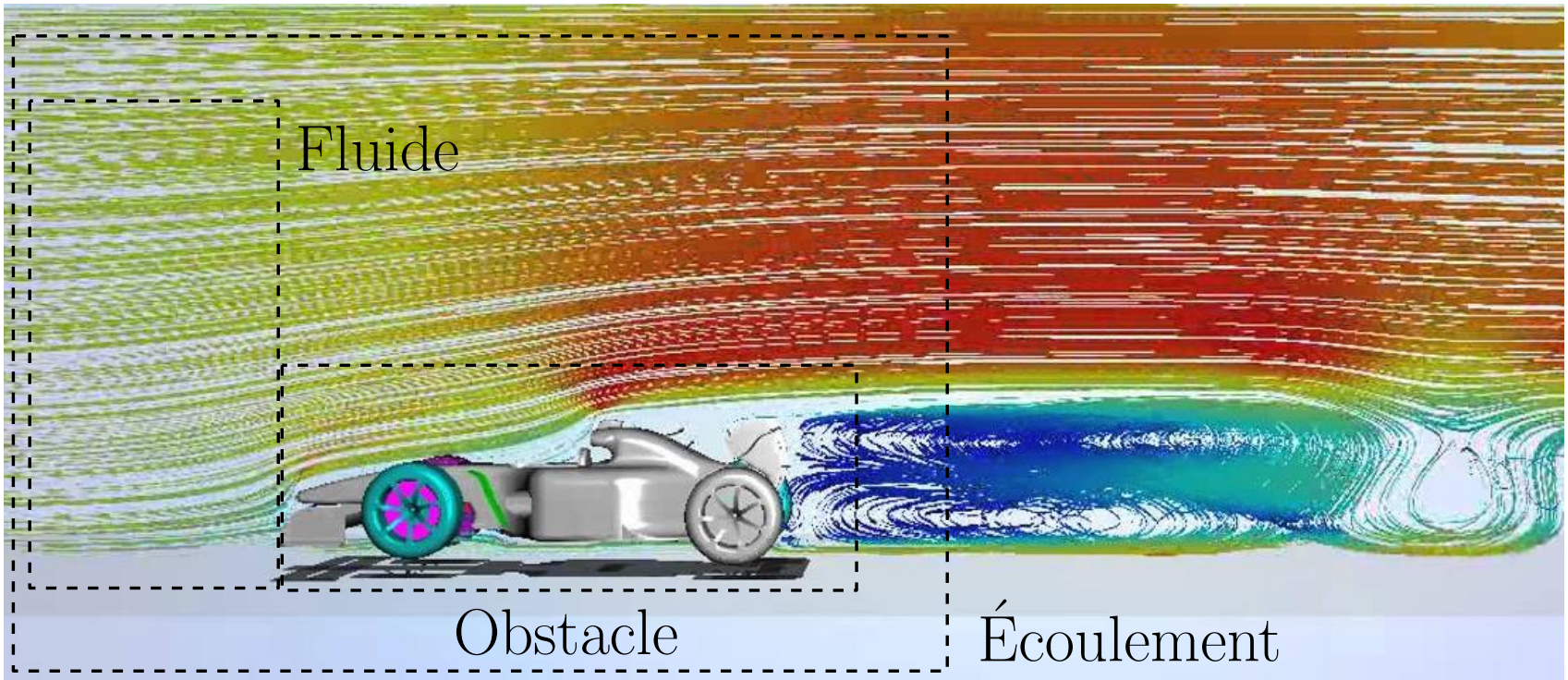
# Fil rouge

- Trainée d'une voiture ?
- Fluide : viscosite $\nu$, densité $\rho$
- Obstacle : vitesse $V$, longueur $L$, surface $S$
- Écoulement = Fluide + Obstacle + ...



# Fil rouge

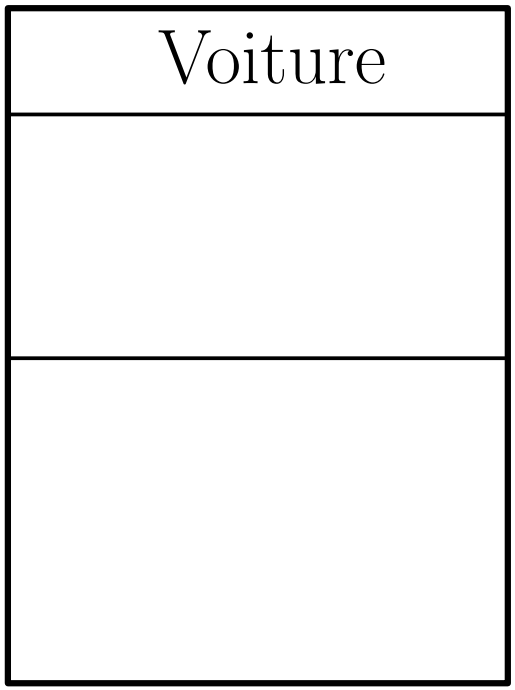
- Trainée d'une voiture ?
- Fluide : viscosite $\nu$, densité $\rho$
- Obstacle : vitesse $V$, longueur $L$, surface $S$
- Écoulement = Fluide + Obstacle + ...
- $F = 0.5 C_x \rho S V^2$, $C = 24/Re$, $Re = \rho V L/\nu$



1. Historique et philosophie
2. Classe et Objet
3. La surcharge
4. Héritage
5. Objets et pointeurs
6. Polymorphisme
7. Classe abstraite

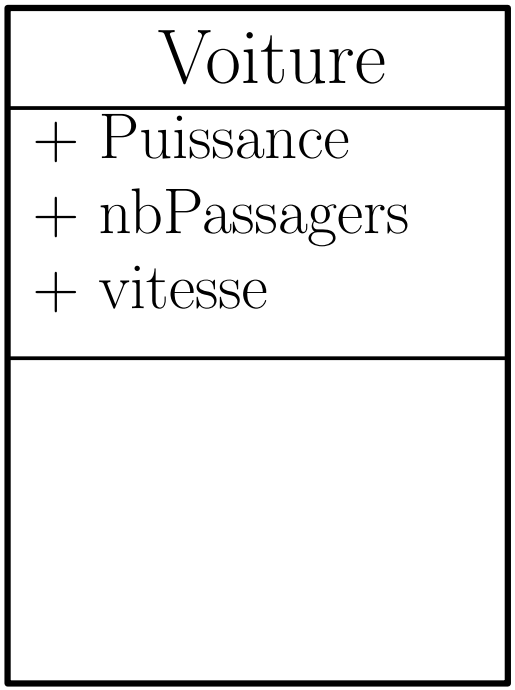
# Objet

- Définition : Structure de donnée valuée qui envoie et répond à des messages d'autres objets



# Objet

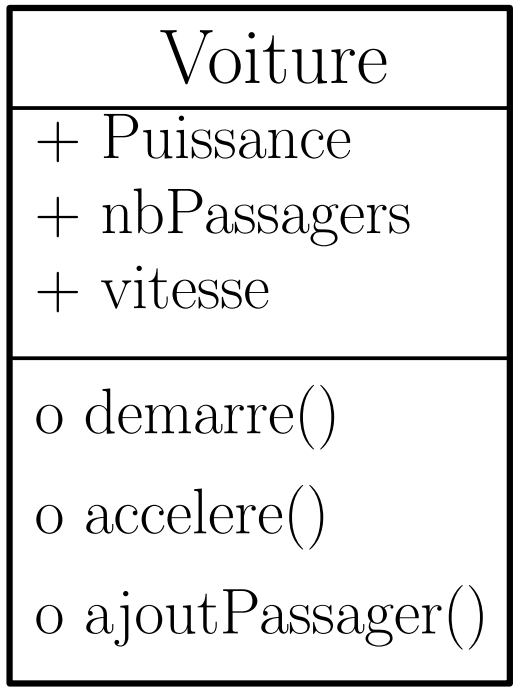
- Définition : Structure de donnée valuée qui envoie et répond à des messages d'autres objets
- Données : décrivent l'état de l'objet (variables)
  Sont appelées Attributs.



# Objet

- Définition : Structure de donnée valuée qui envoie et répond à des messages d'autres objets
- Données : décrivent l'état de l'objet (variables)
  Sont appelées Attributs.
- Messages :
  - décrivent le comportement de l'objet (modification de l'état)
  - comment les objets communiquent (interagissent) entre eux

  Sont appelés Méthodes
  $\Rightarrow$ fonctions.



# Classe

Définition : classe = un moule pour créer un objet
Définit les méthodes et les attributs (sans les initialiser)
Exemple :

```
class Fluide{
  // attributs
  double rho; // masse volumique
  double nu;  // viscosite

  // methodes
  double reynolds(double V, double L)
  {
    return rho*V*L/nu;
  }

  double cx(double V, double L)
  {
    return 24.0/reynolds(V,L);
  }

}
```

# Créer un objet

- On déclare un objet instance de classe “Fluide”
- On fait appel aux attributs et aux méthodes par “.”

```
int main()
{
  // air est une instance de classe Fluide
  Fluide air;

  // modification des attributs
  air.rho = 1.12;
  air.nu = 0.00001;

  // appel au m\'ethodes
  double nbReynolds = air.reynolds(10.0,1.0);
  double coeff = air.cx(40.0,4.0);

  return 0;
}
```

# Conventions

- On sépare les “mots” par des majuscules
- Une classe commence par une majuscule :
  class CeciEstUneClasse
- Méthodes et attributs commencent par une minuscule :
  double ceciEstUnAttribut;
  void ceciEstUneMethode();
- Les constantes sont entièrement en majuscule (séparateur : underscore)
  int CECI_EST_UNE_CONSTANTE;

# Droits et accès (Encapsulation)

- Chaque attribut et méthode possèdent des droits d'accès
- Sécurise les données et verrouille certaines méthodes

# Droits et accès (Encapsulation)

- Chaque attribut et méthode possèdent des droits d'accès
- Sécurise les données et verrouille certaines méthodes
- 3 type de droits :
  - public : méthodes et attributs accessibles partout (depuis n'importe quelle classe/fonction)
  - private : méthodes et attributs accessible uniquement depuis la classe dans laquelle ils ont été définis
  - protected : voir plus tard ...

# Droits et accès (Encapsulation)

- Chaque attribut et méthode possèdent des droits d'accès
- Sécurise les données et verrouille certaines méthodes
- 3 type de droits :
  - public : méthodes et attributs accessibles partout (depuis n'importe quelle classe/fonction)
  - private : méthodes et attributs accessible uniquement depuis la classe dans laquelle ils ont été définis
  - protected : voir plus tard ...

**Règle en POO**

les attributs doivent Ã^atre private !
Un objet ne doit pas pouvoir être modifié par un autre objet autrement qu'au travers d'un message (méthode)

# Retour à notre classe

```
class Fluide{
  // attributs inaccessibles depuis l'exterieur
  private :
  double rho; // masse volumique
  double nu;  // viscosite

  // methodes
  // nombre de Reynolds, private aussi
  double reynolds(double V, double L)
  {
    return rho*V*L/nu;
  }

  public :
  // coefficient de frottement, seule methode utile
  // en dehors de la classe
  double cx(double V, double L)
  {
    return 24.0/reynolds(V,L);
  }

}
```

# Méthodes particulières

- Constructeur : initialise l'état de l'objet
  Doit porter le même nom que la classe et n'est pas typé.

```
class Fluide{
  // constructeur de la classe
  Fluide(double r, double n)
  {
    rho = r;
    nu = n;
  }
  ...
}

int main()
{
  Fluide air(1.2,0.00001);
  double coeff = air.cx(10.0,1.0);
  return 0;
}
```

# Méthodes particulières

- Destructeur : désalloue la mémoire d'un objet
  Porte le même nom que la classe précédé de ~

```
class Fluide{
  ...
  ~Fluide()
  {
  }
  ...
}
```

- Est appelé par le compilateur lorsque l'objet est détruit.
- Doit toujours être défini même s'il est vide !

# Méthodes particulières

- Accesseur : permet d'accéder à la valeur d'un attribut (private)
  S'appelle généralement getLeNomDeLAttribut()

```
class Fluide{
  ...
  // attributs
  private :
  double rho;
  double nu;
  //accesseurs
  public :
  double getDensite(){
    return rho;
  }
  double getVisco(){
    return nu;
  }
  ...
}
```

# Méthodes particulières

- Modificateur : permet de modifier la valeur d'un attribut
  S'appelle généralement setLeNomDeLAttribut() et est de type void

```
class Fluide{
  ...
  // attributs
  private :
  double rho;
  double nu;
  //accesseurs
  public :
  void setDensite(double r){
    rho = r;
  }
  void setVisco(double n){
    nu = n;
  }
  ...
}
```

# Séparer prototype et définitions

- Idée : séparer la classe + les prototypes de méthodes du contenu (définition) des méthodes.
- Convention en C++ (pas dans tous les langages de POO)

# Séparer prototype et définitions

- Idée : séparer la classe + les prototypes de méthodes du contenu (définition) des méthodes.
- Convention en C++ (pas dans tous les langages de POO)
- Pour chaque classe on créé :
  - Un fichier .h (header) contenant la classe, les attributs et les prototypes : Fluide.h

```
class Fluide{
  private :
  double rho;
  double nu;

  // seulement le prototype
  double reynolds(double V, double L);
  public :
  Fluide(double r, double n);
  double cx(double V, double L);
}
```

# Séparer prototype et définitions

- Idée : séparer la classe + les prototypes de méthodes du contenu (définition) des méthodes.
- Convention en C++ (pas dans tout les langages de POO)
- Pour chaque classe on créé :
  - Un fichier .h (header) contenant la classe, les attributs et les prototypes : Fluide.h
  - Un fichier .cpp (source) contenant la définition des méthodes : Fluide.cpp

```
//definition des methodes
double Fluide::reynolds(double V, double L){
  return rho*V*L/nu;
}
double Fluide::cx(double V, double L){
  return 24.0/reynolds(V,L);
}
```

  - on doit alors spécifier la classe suivi de "::"

# ... et la classe obstacle (Obstacle.h)

```
class Obstacle{
  private : //Atributs
  double longueur;
  double vitesse;
  double surface;

  public : // Methodes (toutes publiques)
  Obstacle(double l, double v, double s);  // constructeur
  double getLongueur();    // accesseurs
  double getVitesse();
  double getSurface();
  void accelere(double v);  // modificateur
}
```

# ... et la classe obstacle (Obstacle.cpp)

```
// constructeur
Obstacle::Obstacle(double l, double v, double s){
  longueur = l;
  surface = s;
  vitesse = v
}

//accesseurs
double Obstacle::getLongueur(){
  return longueur;
}
double Obstacle::getVitesse(){
  return vitesse;
}
double Obstacle::getSurface(){
  return surface;
}

//modificateur (uniquement la vitesse)
void accelere(double v){
  vitesse = vitesse + v;
}
```

# ... et dans le main.cpp

```
#include "Fluide.h"
#include "Obstacle.h"

int main()
{
  // Declaration des objets
  Fluide air(1.2, 0.00001);
  Obstacle voiture(4.0, 0.0, 0.5);
  ...
  // modification d'un objet
  voiture.accelere(10.0);
  ...
  // calcul des coefficients
  double coeff = air.cx(voiture.getLongueur(),
                                    voiture.getVistesse());
  double force = 0.5*coeff*...
  ...
  return 0;
}
```

# Associer les classes entre elles

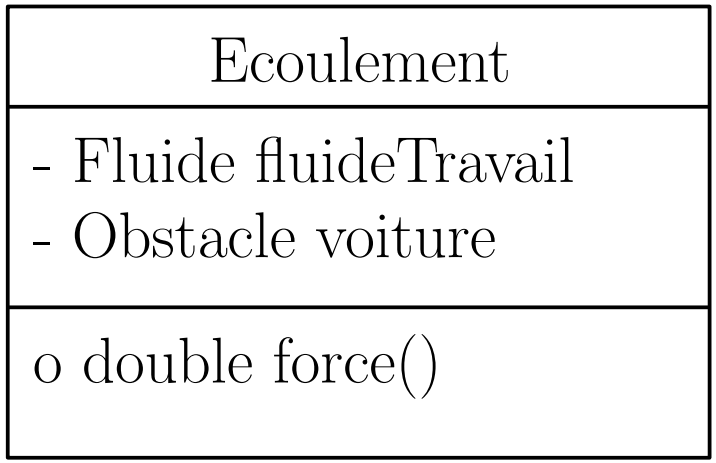
- But de la POO : faire interagir les objets entre eux ( = algorithme)

# Associer les classes entre elles

- But de la POO : faire interagir les objets entre eux ( = algorithme)
- Les objets peuvent communiquer (messages)

# Associer les classes entre elles

- But de la POO : faire interagir les objets entre eux ( = algorithme)
- Les objets peuvent communiquer (messages)
- Les objets peuvent contenir d'autres objets !



# Associer les classes entre elles

```
// inclusion des headers contenant les classes
// necessaire pour creer un Ecoulement !
#include "Fluide.h"
#include "Obstacle.h"

 class Ecoulement{
   private :
   // 2 Attributs : 1 Fluide + 1 Obstacle
   Fluide fluideTravail;
   Obstacle voiture;

   public :
   // constructeur a partir d'un fluide et d'un obstacle
   Ecoulement(Fluide f, Obstacle o);
   // calcul de la force
   double force();
}
```

# Associer les classes entre elles

```
#include "Ecoulement.h"

Ecoulement::Ecoulement(Fluide f,Obstacle o){
  fluideTravail = f;
  voiture = o;
}

double Ecoulement::force(){
  // recuparation de toutes les donnees utiles
  double v = voiture.getVitesse();
  double l = voiture.getLongueur();
  double s = voiture.getSurface();
  double rho = fluideTravail.getDensite();
  double coeff = fluideTravail.cx(v,l);
  // calcul final de la force
  return 0.5*rho*coeff*s*v*v;
}
```

# Cas des inclusions multiples

- Dans mon fichier main.cpp

```
// on inclut tout les headers dont on a besoin !
#include "Fluide.h"
#include "Obstacle.h"
#include "Ecoulement.h"

int main()
{
  Fluide air(1.2, 0.00001);
  Obstacle lada(4.0, 0.5, 3.0);
  Ecoulement cas1(air, lada);

  double resultat = cas1.force();
}
```

# Cas des inclusions multiples

- Dans mon fichier main.cpp

```
// on inclut tout les headers dont on a besoin !
#include "Fluide.h"
#include "Obstacle.h"
#include "Ecoulement.h"

int main()
{
  Fluide air(1.2, 0.00001);
  Obstacle lada(4.0, 0.5, 3.0);
  Ecoulement cas1(air, lada);

  double resultat = cas1.force();
}
```

- Fluide.h et Obstacle.h sont inclus, MAIS ils sont aussi inclus dans Ecoulement.h !
  Compilateur $\Rightarrow$ Erreur : Multiple définitions

# Cas des inclusions multiples : solution(s)

# Cas des inclusions multiples : solution(s)

- Réfléchir à tous les cas possibles,
  exclure systématiquement et manuellement toutes les inclusions multiples
  $\Rightarrow$ long et fastidieux
  $\Rightarrow$ à refaire à chaque modification !

# Cas des inclusions multiples : solution(s)

- Réfléchir à tous les cas possibles,
  exclure systématiquement et manuellement toutes les inclusions multiples
  $\Rightarrow$ long et fastidieux
  $\Rightarrow$ à refaire à chaque modification !
- Empêcher le compilateur de ré-inclure une classe déjà définie : directive de préprocesseur (dans le header)

# Cas des inclusions multiples : solution(s)

- Réfléchir à tous les cas possibles,
  exclure systématiquement et manuellement toutes les inclusions multiples
  ⇒ long et fastidieux
  ⇒ à refaire à chaque modification !
- Empêcher le compilateur de ré-inclure une classe déjà définie : directive de préprocesseur (dans le header)
- Exemple du fichier Fluide.h

```
#ifndef DEF_FLUIDE
#define DEF_FLUIDE

class Fluide{
  public :
  ...
  ...
}
#endif
```

# Cas des inclusions multiples : solution(s)

- Réfléchir à tous les cas possibles,
  exclure systématiquement et manuellement toutes les inclusions multiples
  ⇒ long et fastidieux
  ⇒ à refaire à chaque modification !
- Empêcher le compilateur de ré-inclure une classe déjà définie : directive de préprocesseur (dans le header)
- Exemple du fichier Fluide.h

```
#ifndef DEF_FLUIDE
#define DEF_FLUIDE

class Fluide{
  public :
  ...
  ...
}
#endif
```

- Idem pour chaque classe ...

# Oui, mais ...

- $\approx$ 100 lignes de codes ...

# Oui, mais ...

- $\approx$ 100 lignes de codes ...
- ... alors que :

```
program ecoulement
implicit none
integer, parameter :: dp = kind(1.D0)
real(kind=dp) :: vitesse=0.5d0, longueur=4.0d0, surface=2.d0
real(kind=dp) :: densite=1.2d0, viscosite=1.0d-5, reynolds
real(kind=dp) :: force, cx

reynolds = densite*vitesse*longueur/viscosite
cx = 24.d0/reynolds
force = 0.5*cx*densite*vitesse*vitesse*surface
print*, "f=",force

end program ecoulement
```

# Oui, mais ...

- $\approx$ 100 lignes de codes ...
- ... alors que :

```
program ecoulement
implicit none
integer, parameter :: dp = kind(1.D0)
real(kind=dp) :: vitesse=0.5d0, longueur=4.0d0, surface=2.d0
real(kind=dp) :: densite=1.2d0, viscosite=1.0d-5, reynolds
real(kind=dp) :: force, cx

reynolds = densite*vitesse*longueur/viscosite
cx = 24.d0/reynolds
force = 0.5*cx*densite*vitesse*vitesse*surface
print*, "f=",force

end program ecoulement
```

- Et pourtant :
  - + Portable : création de bibliothèques
  - + Simple : pas besoin de connaître le fonctionnement interne !
  - Et + encore !

1. Historique et philosophie
2. Classe et Objet
3. La surcharge
4. Héritage
5. Objets et pointeurs
6. Polymorphisme
7. Classe abstraite

# La surcharge

- Idée : modifier le comportement d'une méthode (n'importe laquelle !) en changeant les arguments (nombre ou type !)

# La surcharge

- Idée : modifier le comportement d'une méthode (n'importe laquelle !) en changeant les arguments (nombre ou type !)
- Exemple : Guitare

| Guitare |
| --- |
| - Corde [x6] |
| o Guitare(int nbFretes)<br>o accorde()<br>o void accorde(bool grave)<br>o void accorde(int capo)<br>o void accorde(bool grave, int capo) |

# Guitare : le header

```
#ifndef DEF_GUITARE
#define DEF_GUITARE

class Guitare{
  private :
  // les 6 cordes
  int corde1, corde2, corde3;
  int corde4, corde5, corde6;
  public :
  // constructeur
  Guitare(int nbFrete);
  // methode pour accorder la guitare
  void accorde();
  // surcharge : methode d'accordage en Re
  void accorde(bool grave);
  // Surcharge : ajout d'un capodastre
  void accorde(int capo);
  // Surcharge : accordage en Re et ajout d'un capodastre
  void accorde(bool grave, int capo);
  }

#endif
```

# La surcharge

```
// methode d'accordage
void Guitare::accorde(){
  corde1 = Mi;
  corde2 = La;
  corde3 = Re;
  corde4 = Sol;
  corde5 = Si;
  corde6 = Mi;
}
// methode d'accordage surchargee
void Guitare::accorde(bool grave){
  if(grave){
    corde1 = Re;
    corde2 = Sol;
    corde3 = Do;
    corde4 = Fa;
    corde5 = La;
    corde6 = Re;
  }
  else{
    accorde();
  }
}
```

# La surcharge

```
// avec un capodastre
void accorde(int capo){
  // on commence par accorder normalement
  accorde();
  // et on ajoute le capodastre
  corde1 = corde1 + capo;
  corde2 = corde2 + capo;
  corde3 = corde3 + capo;
  corde4 = corde4 + capo;
  corde5 = corde5 + capo;
  corde6 = corde6 + capo;
}

// et fianelement un accorage en re avec capodastre
void accorde(bool grave, int capo){
  // on accorde en Re
  accorde(grave);
  // et on ajoute le capodastre
  corde1 = corde1 + capo;
  ...
}
```

# Méthode du fénéant

```
class Guitare{
  ...
  public :
  // La surcharge se fait automatiquement !
  void accorde(bool grave = False, int capo = 0);
}

void Guitare::accorde(bool grave, int capo)
{
  if(grave){
    corde1 = Re;
    ...
  }
  else{
    corde1 = Mi;
    ...
  }
  corde1 = corde1 + capo;
  ...
}
```

# ... et dans notre cas ?

- méthode vitesse :

```
void Obstacle::accelere(double v){
  vitesse = vitesse + v;
}
```

- Surcharge sans argument

```
void Obstacle::accelere(){
  vitesse = 100.0;
}
```

- Surcharge avec 2 arguments

```
void Obstacle::accelere(double v, double vmax){
  accelere(v);
  if(vitesse>vmax){
    vitesse = vmax;
  }
}
```

# ... et dans notre cas ? Surcharge du constructeur !

- La classe

```
class Fluide{
  private :
    double rho,nu;
  public :
    Fluide(double r, double n);
    Fluide();
```

- Constructeur par défaut

```
Fluide::Fluide(double r, double n){
  rho = r;
  nu = n;
}
```

- Constructeur sans argument

```
// Si le fluide n'est pas defini
// on utilise de l'air automatiquement
Fluide::Fluide(){
  rho = 1.2;
  nu = 0.00001;
}
```

# Surcharge des opérateurs

- Cas de variables classiques

```
double x, y, z;
...
x = y + z;
if(y>z){
```

⇒ Aucun problème !

# Surcharge des opérateurs

- Cas de variables classiques

```
double x, y, z;
...
x = y + z;
if(y>z){
```

⇒ Aucun problème !

- Avec des objets ?

```
Fluide air;
Fluide helium(0.14, 0.00012);
Obstacle voiture(4.0,30.0,0.5);
Obstacle remorque(2.0,0.0,0.25);
Obstacle vehicule;
...
vehicule = voiture + remorque;
if(air>helium){
...
```

⇒ insultes du compilateur !

# Surcharge des opérateurs

- Les opérateurs agissant sur les objets ne sont pas définis
- C'est au programmeur (vous) de définir tous les opérateurs nécessaires

# Surcharge des opérateurs

- Les opérateurs agissant sur les objets ne sont pas définis
- C'est au programmeur (vous) de définir tous les opérateurs nécessaires
- Les opérateurs de comparaison ou d'opérations arithmétiques n'agissent pas sur l'objet lui-même
  $\Rightarrow$ les protopypes doivent être définis en dehors de la classe !

# Surcharge des opérateurs

- Les opérateurs agissant sur les objets ne sont pas définis
- C'est au programmeur (vous) de définir tous les opérateurs nécessaires
- Les opérateurs de comparaison ou d'opérations arithmétiques n'agissent pas sur l'objet lui-même
  $\Rightarrow$ les protopypes doivent être définis en dehors de la classe !
- Exemple sur la classe Fluide

```
class Fluide{
  private :
    double rho;
    double nu;
  public :
    ...
}
// surcharge de mes operateurs utiles
bool operator>(Fluide const &f1, Fluide const &f2);
Fluide operator+(Fluide const &f1, Fluide const &f2);
```

# Surcharge des opérateurs : la définition

```
Fluide::Fluide(double r, double n){
  rho = r ; nu = n;
}
 ...
bool operator>(Fluide const &f1, Fluide const &f2){
  bool plusLourd = False
  // un fluide est > a un autre s'il est plus lourd !
  if(f1.getDensite() > f2.getDensite()){
     plusLourd = True
  }
  return plusLourd;
}
Fluide operator+(Fluide const &f1, Fluide const &f2){
  // le fluide issue du melange a une densite moyenne
  double densiteMoy = 0.5*(f1.getDensite() + f2.getDensite());
  // et la masse volumique est la moyenne ponderee
  double viscoMoy =  f1.getVisco()*f1.getDensite();
  viscoMoy +=  f2.getVisco()*f2.getDensite() ;
  viscoMoy /= densiteMoy;
  Fluide melange(densiteMoy, viscoMoy);
  return melange;
}
```

# Surcharge des opérateurs : la définition

```
Obstacle operator+(Obstacle const& voiture,
                   Obstacle const& remorque){
  double v = voiture.getVitesse();
  double l = voiture.getLongueur() + remorque.getLongueur();
  double s = maxi(voiture.getSurface(),remorque.getSurface());
  Obstacle vehicule(l,v,s);
  return vehicule;
}
```

```
Obstacle operator*(double x, Obstacle const& voiture){
  double v = x*voiture.getVitesse();
  double l = voiture.getLongueur();
  double s = x*x*voiture.getSurface();
  Obstacle vehicule(l,v,s);
  return vehicule;
}
```

1. Historique et philosophie
2. Classe et Objet
3. La surcharge
4. Héritage
5. Objets et pointeurs
6. Polymorphisme
7. Classe abstraite

# L'héritage

- Technique permettant de construire une classe à partir d'une autre

# L'héritage

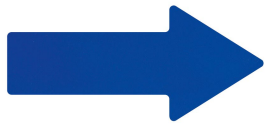
- Technique permettant de construire une classe à partir d'une autre
- Intérêt :
  - Spécialisation : réutilise tous les attributs/méthodes de la classe héritée en y **ajoutant d'autres attributs/méthodes**
  - Redéfinition : **modifie l'état et le comportement** (attributs/méthodes) de la classe dont on hérite

# L'héritage

- Technique permettant de construire une classe à partir d'une autre
- Intérêt :
  - Spécialisation : réutilise tous les attributs/méthodes de la classe héritée en y **ajoutant d'autres attributs/méthodes**
  - Redéfinition : **modifie l'état et le comportement** (attributs/méthodes) de la classe dont on hérite
- Exemple d'un moto

# L'héritage

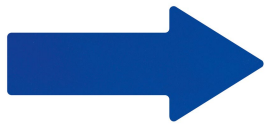
- Technique permettant de construire une classe à partir d'une autre
- Intérêt :
  - Spécialisation : réutilise tous les attributs/méthodes de la classe héritée en y **ajoutant d'autres attributs/méthodes**
  - Redéfinition : **modifie l'état et le comportement** (attributs/méthodes) de la classe dont on hérite
- Exemple d'un moto



Ajout : gyrophare (attribut) + allumeGyrophare() (méthode)

# L'héritage

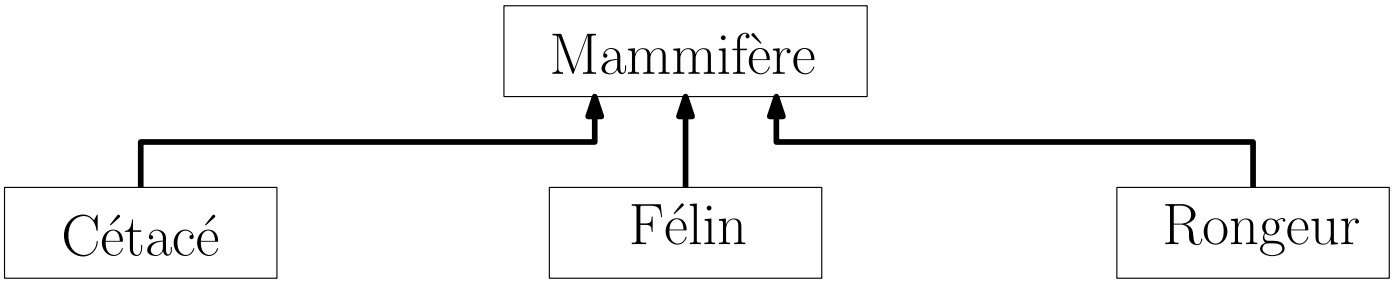
- Technique permettant de construire une classe à partir d'une autre
- Intérêt :
  - Spécialisation : réutilise tous les attributs/méthodes de la classe héritée en y **ajoutant d'autres attributs/méthodes**
  - Redéfinition : **modifie l'état et le comportement** (attributs/méthodes) de la classe dont on hérite
- Exemple d'un moto



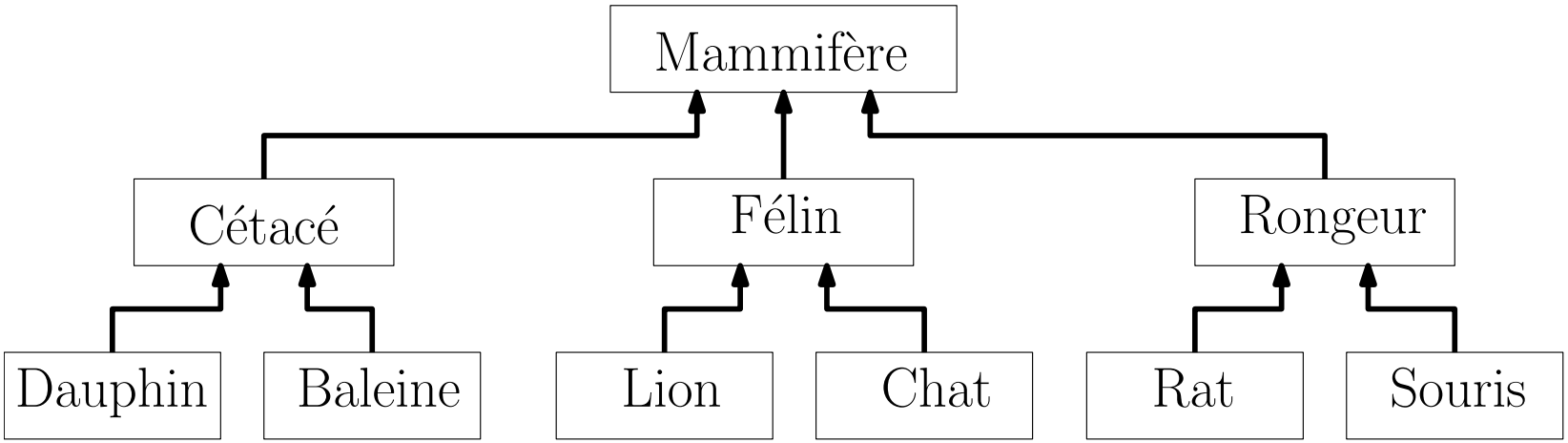
Modification : moteur (attribut) + demarre() (méthode)

# Héritage

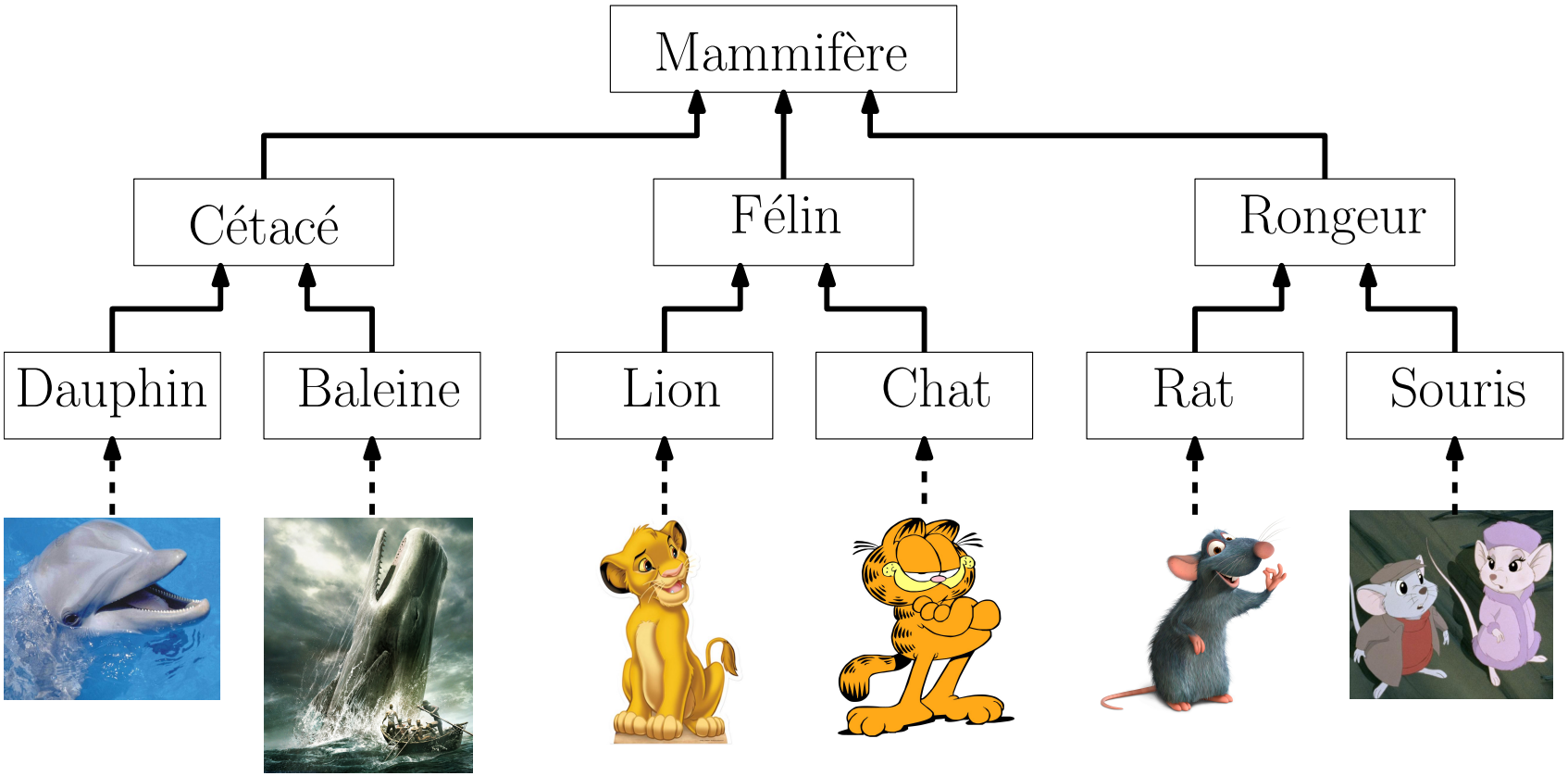
# Héritage



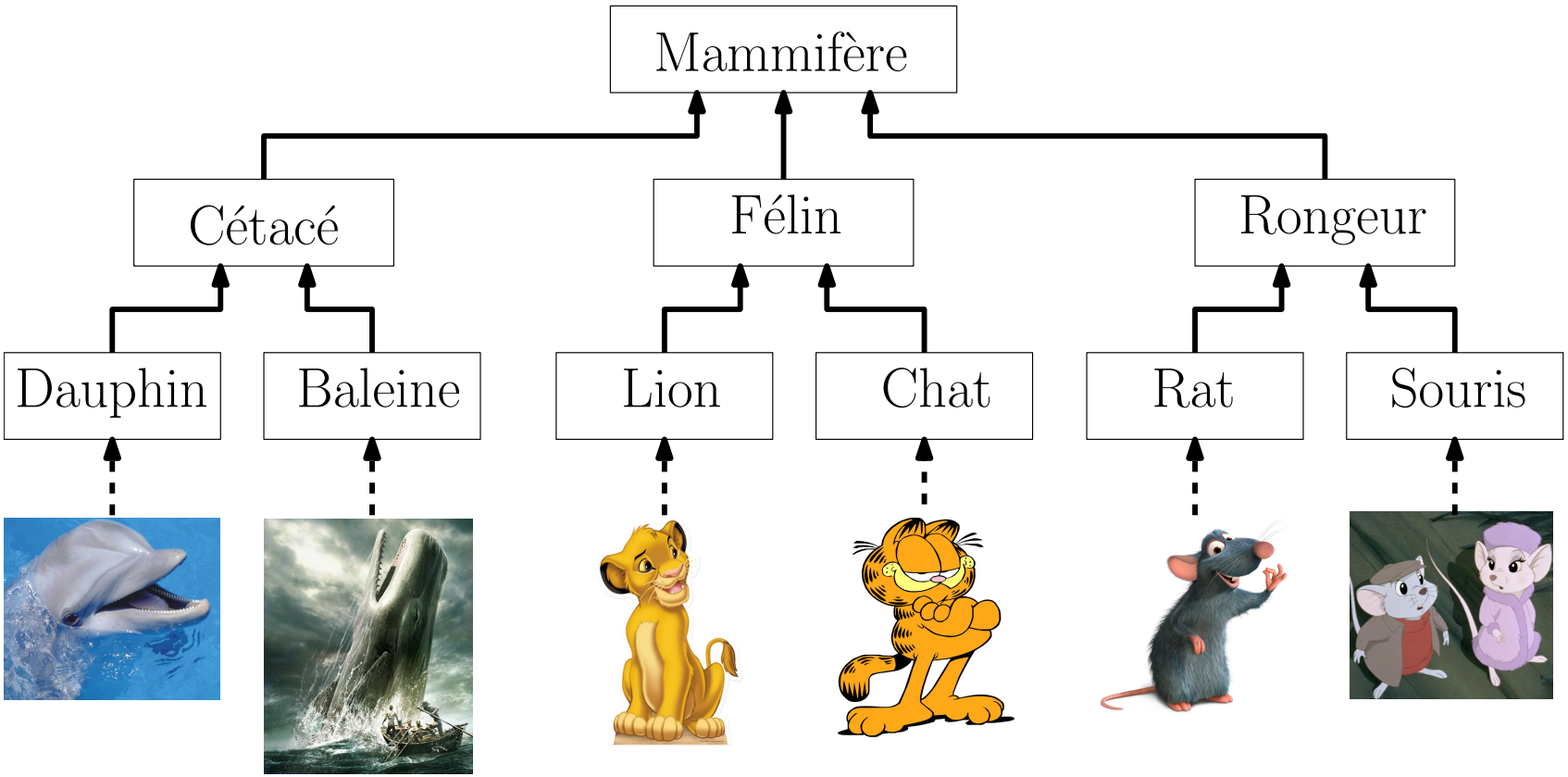
# Héritage


Mammifère
Cétacé
Félin
Rongeur
Dauphin
Baleine
Lion
Chat
Rat
Souris

# Héritage


Mammifère
Cétacé
Félin
Rongeur
Dauphin
Baleine
Lion
Chat
Rat
Souris

# Héritage




Garfield et Flipper sont avant tout des mammifères !

# Vocabulaire et accès

- rongeur hérite de mammifère, rat hérite de rongeur
  $\rightarrow$ rongeur hérite de mammifère
- mammifère est la classe mère, rongeur est la classe fille
  $\rightarrow$ rat est la classe petite fille !
  $\rightarrow$ mammifère est la classe grand-mère de la classe rat

# Vocabulaire et accès

- rongeur hérite de mammifère, rat hérite de rongeur
  $\rightarrow$ rongeur hérite de mammifère
- mammifère est la classe mère, rongeur est la classe fille
  $\rightarrow$ rat est la classe petite fille !
  $\rightarrow$ mammifère est la classe grand-mère de la classe rat
- La classe rat a accès à toutes les méthodes et tous les attributs (ou presque) de la classe rongeur
- rongeur n'a accès qu'aux méthodes et attributs de la classe mammifère, pas ceux de la classe rat.

# Vocabulaire et accès

- rongeur hérite de mammifère, rat hérite de rongeur
  $\rightarrow$ rongeur hérite de mammifère
- mammifère est la classe mère, rongeur est la classe fille
  $\rightarrow$ rat est la classe petite fille !
  $\rightarrow$ mammifère est la classe grand-mère de la classe rat
- La classe rat a accès à toutes les méthodes et tous les attributs (ou presque) de la classe rongeur
- rongeur n'a accès qu'aux méthodes et attributs de la classe mammifère, pas ceux de la classe rat.
- $3^{eme}$ niveau de protection : protected
  $\Rightarrow$ Accès restreint aux classes filles, inaccessible en dehors de la classe et des classes dont elles hérite.

# Les classe Fluide et Turbulence

- La classe Fluide (Fluide.h)

```
class Fluide{
  protected :     // protected car les attributs
    double rho;   // doivent etre visibles aux
    double nu;    // classes filles
    double reynolds(double v, double l);
  public :
    Fluide(double r=1.2, double nu=0.00001);
    double cx(double l, double v);
}
```

# Les classe Fluide et Turbulence

- La classe Fluide (Fluide.h)

```
class Fluide{
  protected :     // protected car les attributs
    double rho;   // doivent etre visibles aux
    double nu;    // classes filles
    double reynolds(double v, double l);
  public :
    Fluide(double r=1.2, double nu=0.00001);
    double cx(double l, double v);
}
```

- La classe Turbulent (Turbulent.h)

```
class Turbulent:public Fluide{
  protected : // je rajoute un attribut
    bool turbu;
  public :
    Turbulent(double r=1.2, double nu = 0.00001,
                                    bool t = False);
    double cx(double l, double v); // ???
}
```

# Définition des méthodes

```
// constructeur de la classe Turbulent
// fait appel au constructeur de Fluide
// directement !
Turbulent::Turbulent(double r, double n, bool t):fluide(r, n){
  turbu = t;
}
// on redefini le comportement de la classe cx()
double cx(double l, double v)
{
  double coeff;
  if(turbu){
    // si le fluide est turbulent, on change la formule
    coeff = 18.5/sqrt(reynolds(l,v));
  }
  else{
    // sinon on fait appel a la formule classique
    coeff = Fluide::cx(l,v);
  }
}
```

La méthode cx() a été redéfinie !

(A ne pas confondre avec la surcharge)

# Autre exemple : classe compressible

```
class Compressible:public Fluide{
 protected : // on ajoute une methode + un attribut
   double vitSon;
   double Mach(double v);
 public :
   Compressible(double r = 1.2,double n = 0.00001,double c = 320.
   double cx(double l, double v); // redefinition
}

Compressible::Compressible(double r, double n,
                                    double c):Fluide(r,n){
  vitSon = c;
}
double Compressible::Mach(double v){
  return v/vitSon; // calcul du nombre de Mach
}
double cx(double l, double v){
  double ma = Mach(l,v);
  double coeff = Fluide::cx(l,v)
  return (1.-Ma*Ma)*coeff // calcul du nouveau coefficient
}
```

1. Historique et philosophie
2. Classe et Objet
3. La surcharge
4. Héritage
5. Objets et pointeurs
6. Polymorphisme
7. Classe abstraite

# Fonctionnement d'un pointeur - C Vs Fortran

- Programme simple en C

```c
void mafonction(int a)
{
    a = a + 2;
    printf("a=%d",a);
}

int main()
{
    int x = 2;
    mafonction(x);
    printf("x=%d",x);
    return 0;
}
```

- Même programme Fortran

```fortran
subroutine maroutine(a)
  implicit none
  integer :: a
  a = a + 2;
  print*, "a=",a
end subroutine maroutine

program main
  implicit none
  integer :: x = 2

  call maroutine(x)
  print*, "x=",x
end program main
```

# Fonctionnement d'un pointeur - C Vs Fortran

- Programme simple en C

```c
void mafonction(int a)
{
    a = a + 2;
    printf("a=%d",a);
}

int main()
{
    int x = 2;
    mafonction(x);
    printf("x=%d",x);
    return 0;
}
```

- Résultat :

  a=4

  x=2

- Même programme Fortran

```fortran
subroutine maroutine(a)
  implicit none
  integer :: a
  a = a + 2;
  print*, "a=",a
end subroutine maroutine

program main
  implicit none
  integer :: x = 2

  call maroutine(x)
  print*, "x=",x
end program main
```

- Résultat :

  a=4

  x=4

# Fonctionnement d'un pointeur

## Pourquoi ?

# Fonctionnement d'un pointeur

## Pourquoi ?

- En C/C++, les arguments sont passés par valeurs.
  $\Rightarrow$ recopie des valeurs des arguments
  $\Rightarrow$ pas de modifications possibles des données

# Fonctionnement d'un pointeur

## Pourquoi ?

- En C/C++, les arguments sont passés par valeurs.
  $\Rightarrow$ recopie des valeurs des arguments
  $\Rightarrow$ pas de modifications possibles des données
- En Fortran, les arguments sont passés par pointeurs !
  $\Rightarrow$ Pas de recopie des données : économie de mémoire
  $\Rightarrow$ Attention au recyclage des variables !

# Fonctionnement d'un pointeur (C)

- Un petit programme ...

```c
int main()
{
    int a;
    int b;

    a = 4;
    b = -21;

    return 0;
}
```

- ... et dans la mémoire

| | Adresse | Valeur |
|---|---|---|
| | | |
| a | 192653 | 4 |
| | ⋮ | ⋮ |
| b | 221758 | -21 |
| | ⋮ | ⋮ |

# Fonctionnement d'un pointeur (C)

- Un petit programme ...

```c
int main()
{
    int a;
    int b;
    // on declare un pointeur
    int *p;

    a = 4;
    b = -21;
    // on fait pointer p sur a
    p = &a;
    // & represente la reference
    // d'une variable

    return 0;
}
```

- ... et dans la mémoire

| | Adresse | Valeur |
|---|---|---|
| | | |
| a | 192653 | 4 |
| | ⋮ | ⋮ |
| b | 221758 | -21 |
| | ⋮ | ⋮ |
| *p | 398561 | 192653 |
| | | |

# Vocabulaire et syntaxe

- Def : Un pointeur est une variable dont la valeur est l'adresse d'une autre variable

# Vocabulaire et syntaxe

- Def : Un pointeur est une variable dont la valeur est l'adresse d'une autre variable
- &a désigne l'adresse de la variable a = référence

# Vocabulaire et syntaxe

- Def : Un pointeur est une variable dont la valeur est l'adresse d'une autre variable
- &a désigne l'adresse de la variable a = référence
- On déclare un pointeur avec *
  Le type de la variable pointée doit être explicitement donné.
  ex :

```
int *x // pointeur sur un entier
double *y // pointeur sur un double
char *c // pointeur sur un caractere
```

# Vocabulaire et syntaxe

- Def : Un pointeur est une variable dont la valeur est l'adresse d'une autre variable
- &a désigne l'adresse de la variable a = référence
- On déclare un pointeur avec *
  Le type de la variable pointée doit être explicitement donné.
  ex :

```
int *x // pointeur sur un entier
double *y // pointeur sur un double
char *c // pointeur sur un caractere
```

- On fait pointer en utilisant l'adresse de la variable associée
  ex :

```
int variable_int;
int *pointeur_sur_int;
pointeur_sur_int = &variable_int;

double toto;
double *pointeur_toto = &toto;
```

# Intérêt

- *pointeur_toto permet d'accéder à la valeur de la variable pointée (toto)

```
double toto=23;
double *pointeur_toto = &toto;

*pointeur_toto = 42;
```

# Intérêt

- *pointeur_toto permet d'accéder à la valeur de la variable pointée (toto)

```
double toto=23;
double *pointeur_toto = &toto;

*pointeur_toto = 42;
```

- Ici on a directement modifié la valeur de toto au travers de son pointeur !

# Intérêt

- *pointeur_toto permet d'accéder à la valeur de la variable pointée (toto)

```
double toto=23;
double *pointeur_toto = &toto;

*pointeur_toto = 42;
```

- Ici on a directement modifié la valeur de toto au travers de son pointeur !
- Preuve :

```
printf("toto=%lf");
```

toto = 42

# Intérêt ? Retour sur l'exemple

```
// la fonction recoit un pointeur en argument
void mafonction(int *p)
{
    *p = *p + 2;
}

int main()
{
    int a = 2;
    // on passe donc la reference d'une variable
    // comme argument de la fonction
    mafonction(&a);
    // modification ?
    printf("a=%d",a);
    return 0;
}
```

Résultat : a = 4 !

La valeur de a a été modifiée dans la fonction

# Subtilité du C++

- Le C++ permet de traiter directement d'une référence sans passer explicitement par un pointeur

```
// on utilise directement la reference de la variable
void fonctionReference(int &y){
  y = y + 2;
}

int main()
{
  int a = 2;
  // on ne passe plus la reference,
  // mais directement la variable
  fonctionReference(a);
  // verification du resultat
  cout<<"a="<<a<<endl;
  return 0;
}
```

# Pointeurs et Objets

| Guitariste |
| --- |
| Guitare<br>Onglet |
| joue()<br>improvise()<br>~~changeGuitare(Guitare)~~ |

# Pointeurs et Objets

| Guitariste |
| --- |
| Guitare<br>Onglet |
| joue()<br>improvise()<br>~~changeGuitare(Guitare)~~ |

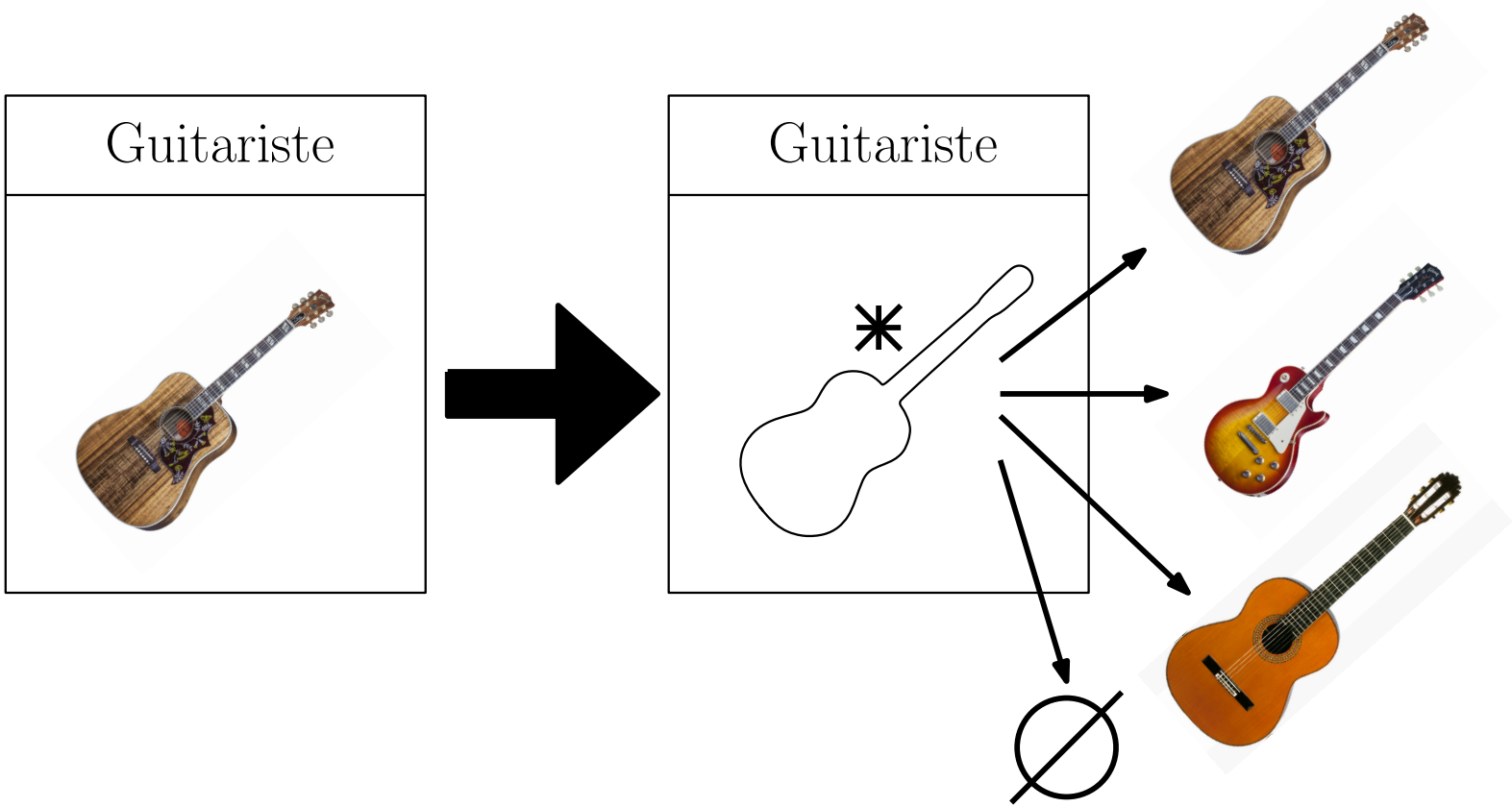
Cela n'a pas de sens de modifier tous les attributs de la Guitare pour que le Guitariste en change !

# Pointeurs sur Objets

- Solution : ne pas utiliser d'attribut Guitare mais un **pointeur sur un objet Guitare**

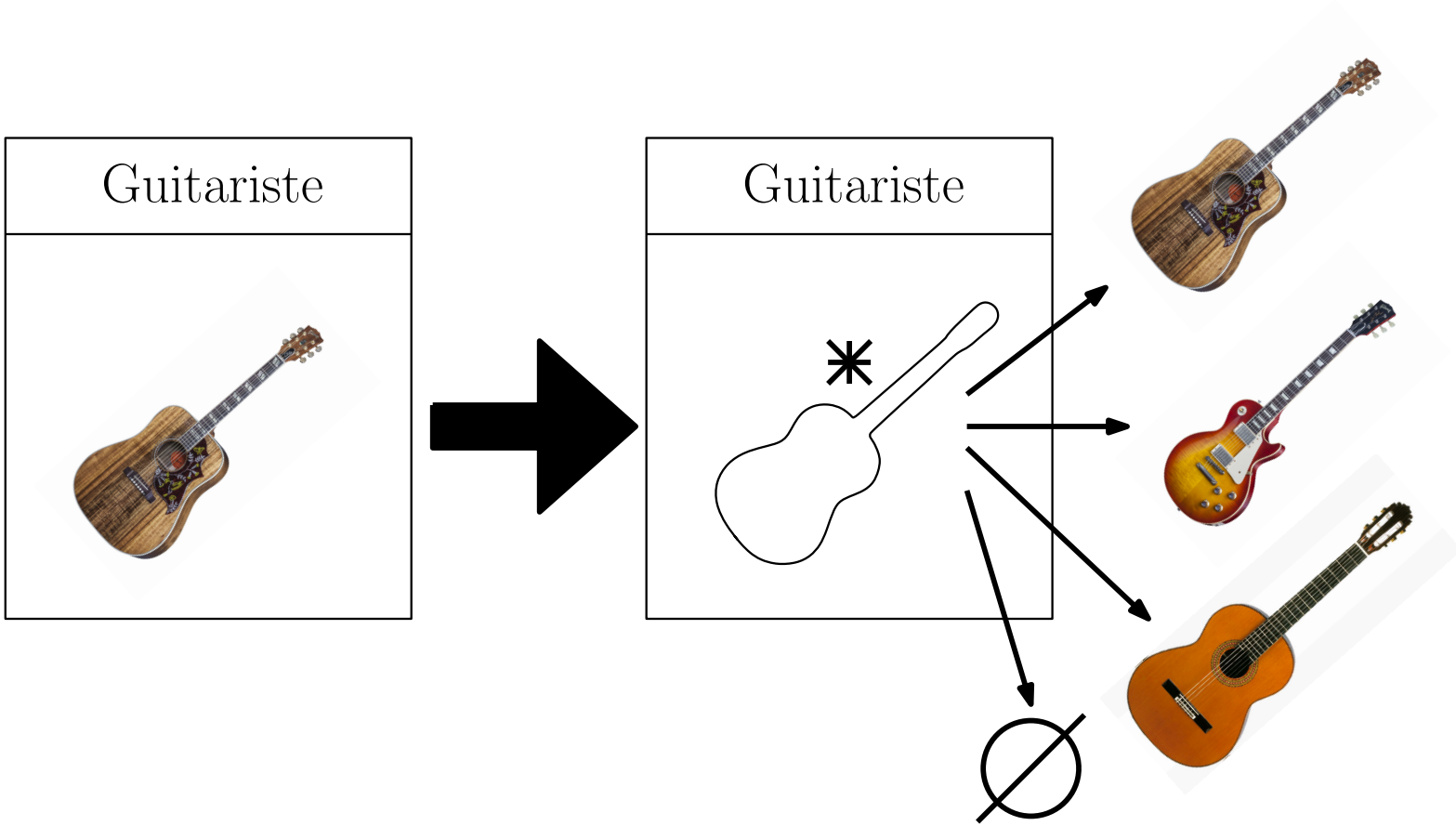
# Pointeurs sur Objets

- Solution : ne pas utiliser d'attribut Guitare mais un **pointeur sur un objet Guitare**



# Pointeurs sur Objets

- Solution : ne pas utiliser d'attribut Guitare mais un **pointeur sur un objet Guitare**




- La Guitare est en dehors du Guitariste !

# Pointeurs sur Objets

```
class Guitariste{
  private :
    // On cree un pointeur
    // sur un objet
    // Guitare
    Guitare *maGuitare;
  public :
    Guitariste();
    ~Guitariste();
    void joue();
    // je vais pouvoir
    // changer de Guitare
    // avec une reference
  void changeGuitare(Guitare &g);
};
```

```
class Guitare{
  private :
    int nbFretes;
    bool electric;
    //contient 6 cordes
    Cordes lesCordes;
  public :
    Guitare(int n,Cordes c,bool e);
    // on surcharge le
    // constructeur pour
    // copier une autre
    // Guitare
    Guitare(Guitare g);
    accorde();
};
```

# Pointeurs sur Objets

- Définition des méthodes

```
Guitariste::Guitariste(){
  // initialement le guitariste n'a pas de guitare !
  maGuitare = NULL;
}
void Guitariste::changeGuitare(Guitare &g){
  // maGuitare pointe vers une nouvelle Guitare !
  maGuitare = &g;
  // cas d'un pointeur, on utilise -> au lieu de .
  maGuitare->accorde();
  maGuitare->joue();
}
```

# Pointeurs sur Objets

- ... et le programme principal :

```
int main()
{
  Cordes cordeMetal;
  Guitare lesPauls(22, corde, true);
  Guitare takamine(24, corde, false);

  Guitariste yusufIslam; // Yusuf n'a pas de guitare
  Guitariste ericClapton; // eric non plus !

  // Yusuf utilise la lesPauls
  yusufIslam.changeGuitare(lesPauls);
  // et Eric la takamine
  ericClapton.changeGuitare(takamine);
  // puis ils echangent !
  yusufIslam.changeGuitare(takamine); // Attention ...
  ericClapton.changeGuitare(lesPauls);
  return 0;
}
```

# Pointeurs sur Objets - Allocation dynamique

- Cas du Luthier (fabricant de Guitare)

```
class Luthier{
  private :
    // On cree un pointeur
    // sur un objet
    // Guitare
    Guitare *maGuitare;
  public :
    Luthier();
    ~Luthier();
    void joue();
    // je vais pouvoir
    // changer de Guitare
    // avec une reference
    void fabrique(int n, Cordes c, bool e );
};
```

# Pointeurs sur Objets - Allocation dynamique

- Définitions

```
Luthier(){
  // j'alloue dynamiquement une guitare
  // a l'aide de new
  Cordes c;
  // creation d'une nouvelle Guitare
  maGuitare = new Guitare(20, c, false);
}
~Luthier(){
  // je dois desallouer la memoire
  // lorsque le luthier est detruit !
  delete maGuitare;
}
void Luthier::fabrique(int n, Cordes c, bool e){
  // je dois d'abord detruire (ou donner)
  // l'ancienne Guitare !
  delete maGuitare;
  // je re-alloue une nouvelle Guitare
  maGuitare = new Guitare(n, c, e);
}
```

# ... et notre fil rouge ?

```
class Ecoulement{
private :
  // le Fluide et l'Obstacle sont des pointeurs !
  Fluide *fluidetravail;
  Obstacle *vehicule;
public :
  Ecoulement();
  ~Ecoulement();
  double force();
  // je change d'obstacle au travers d'une reference !
  void changeObstacle(Obstacle &o);
  // je change de fluide avec une allocation dynamique !
  void changeFluide(double r, double n);
};
```

# ... et notre fil rouge ?

```
// je change d'obstacle au travers d'une reference !
void Ecoulement::changeObstacle(Obstacle &o){
  vehicule = &o;
}
// je change de fluide avec une allocation dynamique !
void Ecoulement::changeFluide(double r, double n){
  delete fluideTravail; // on detruit l'ancien Fluide
  // et on en alloue un nouveau
  fluideTravail = new Fluide(r,n);
}
double Ecoulement::force(){
  double v = vehicule->vitesse();
  double l = vehicule->longueur();
  double rho = fluideTravail->Densite();
  double c = fluideTravail->cx(v,l);
  //calcul de la force
  return 0.5*rho*c*s*v*v;
}
```

# ... et notre fil rouge ?

```
int main()
{
  double f;
  Ecoulement cas1; // Je cree un ecoulement VIERGE
  Obstacle lada(4.0,10.0,0.5);  // je cree 2 vehicules
  Obstacle peugeot104(2.5, 5.0, 0.45);

  // je mets de l'air dans mon banc d'essai
  cas1.changeFluide(1.2, 0.00001);
  // et une lada
  cas1.changeVehicule(lada);
  f = cas1.force();

  // je change de vehicule !
  cas1.changeVehicule(peugeot104);
  f = cas1.force();

  // et je change de fluide : Helium
  cas1.changeFluide(0.17, 0.00002);
  cas1.force();

  return 0;
}
```

# Bilan

- Avantages :
  - Plus grande souplesse dans la gestion des objets
  - Ex : changer d'objet, échanger des objets, de manière "transparente"
- Inconvénients :
  - Gestion explicite de la mémoire (Allocation/désallocation)
  - Deux pointeurs peuvent pointer vers le même objet ...

1. Historique et philosophie
2. Classe et Objet
3. La surcharge
4. Héritage
5. Objets et pointeurs
6. Polymorphisme
7. Classe abstraite

# Définition

- Ethymologie : Grec
  - Poly : plusieurs
  - Morphe : formes

# Définition

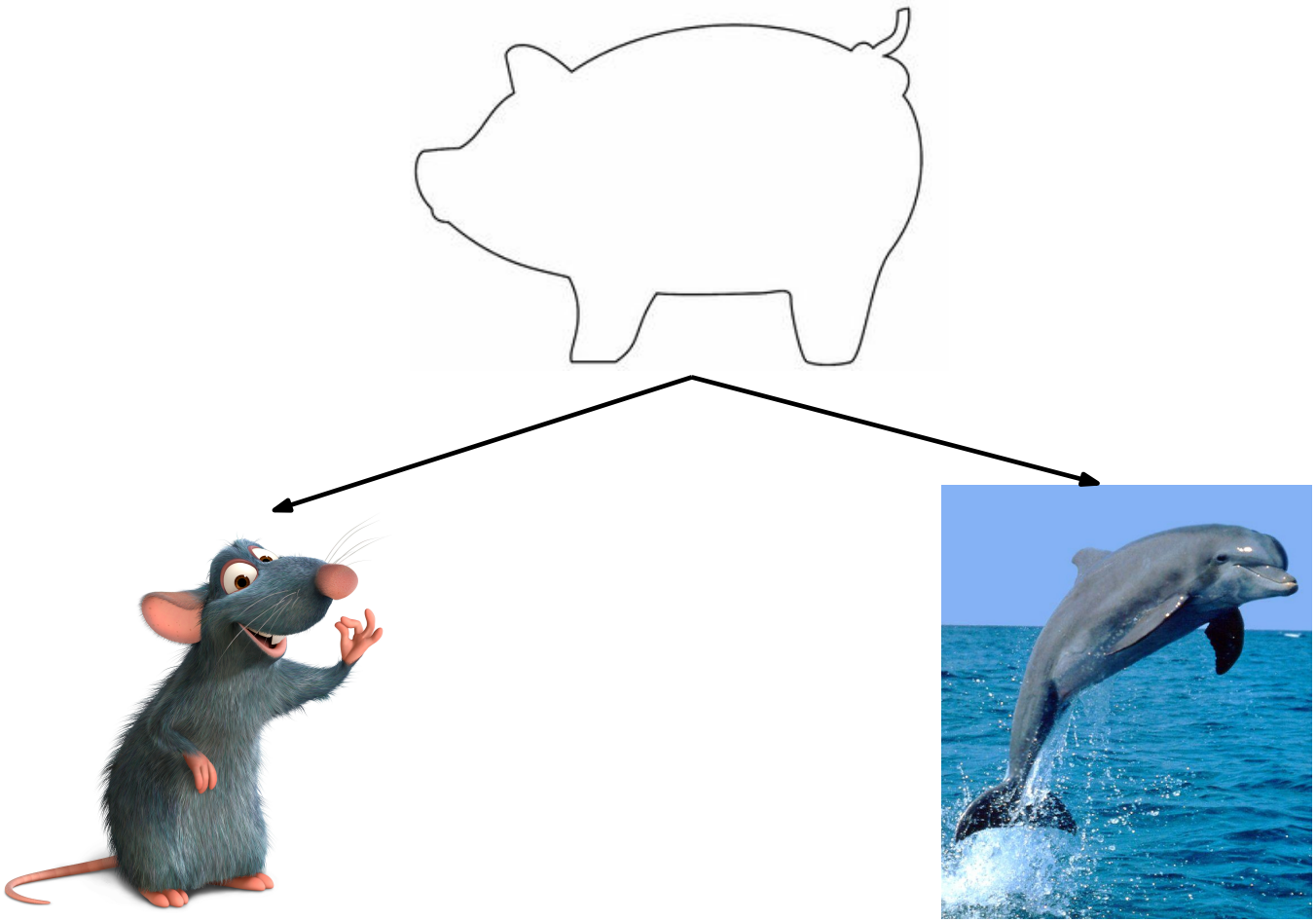
- Ethymologie : Grec
  - Poly : plusieurs
  - Morphe : formes
- Polymorphisme : possibilité de traiter les objets ayant plusieurs formes
- Concept fortement lié à la notion d'héritage

# Définition

- Ethymologie : Grec
  - Poly : plusieurs
  - Morphe : formes
- Polymorphisme : possibilité de traiter les objets ayant plusieurs formes
- Concept fortement lié à la notion d'héritage
- Rappel de l'héritage
  - Garfield est un chat, mais c'est avant tout un mammifère
  - Flipper est un dauphin, mais c'est avant tout un mammifère

Ils ont donc au moins 2 formes possibles ! Comment les gérer ...?

# Simplifions le problème



# Un petit exemple

```
class Animal{
  protected :
  int nbPattes;
  public :
  Animal(int n);
  void description();
};

Animal::Animal(int n){
  nbPattes = n;
}

void Animal::description(){
 cout<<"je suis un animal"<<endl;
 cout<<"j ai "<<nbPattes<<" pattes"<<endl;
}
```

# Petit exemple

```
class Rat:public Animal{
  protected :
  bool poil;
  public :
  Rat(int n, bool q);
  void description();
};

Rat::Rat(int n, bool p):Animal(n){
  poil = p;
}
void Rat::description(){
  cout<<"je suis un rat avec "<<nbPattes<<endl;
  if(poil){
    cout<<" et je suis poilu"<<endl;
  }
  else{
    cout<<" et je n ai pas de poil"<<endl;
  }
}
```

# Un petit exemple

```
class Dauphin:public Animal{
  protected :
  int nbNageoire;
  public :
  Dauphin(int n, int m);
  void description();
};

Dauphin::Dauphin(int n, int m):Animal(n){
  nbNageoire = m;
}

void Dauphin::description(){
  cout<<"je suis un dauphin"<<endl;
  cout<<" je n ai aucune pattes"<<endl;
  cout<<" mais j ai "<<nbNageoire<<" Nageoires"<<endl;
  cout<<" pour nager sous l'eau"<<endl;
}
```

# Un petit exemple

```
int main()
{
  Animal gafield(2);
  Rat ratatouille(4,true);
  Dauphin flipper(0,4);

  garfield.description();
  ratatouille.description();
  flipper.description();
  return 0;
}
```

```
Je suis un animal
 j ai 2 pattes

Je suis un rat avec 4 pattes
 et je suis poilu

Je suis un dauphin
 je n'ai aucune pattes
 mais j'ai 4 nageoires
 pour nager sous l'eau
```

# Un petit exemple

```
void affiche(Animal a){
  a.description();
}

int main()
{
  Animal gafield(2);
  Rat ratatouille(4,true);
  Dauphin flipper(0,4);

  affiche(garfield);
  affiche(ratatouille);
  affiche(flipper);
  return 0;
}
```

- Le compilateur accepte : ratatouille et flipper sont des animaux !

# Un petit exemple

```
void affiche(Animal a){
  a.description();
}

int main()
{
  Animal gafield(2);
  Rat ratatouille(4,true);
  Dauphin flipper(0,4);

  affiche(garfield);
  affiche(ratatouille);
  affiche(flipper);
  return 0;
}
```

- Le compilateur accepte : ratatouille et flipper sont des animaux !
- Résultat :

```
Je suis un animal
 j ai 2 pattes

Je suis un animal
 j ai 4 pattes

Je suis un animal
 j ai 0 pattes
```

# Un petit exemple

```
void affiche(Animal a){
  a.description();
}

int main()
{
  Animal gafield(2);
  Rat ratatouille(4,true);
  Dauphin flipper(0,4);

  affiche(garfield);
  affiche(ratatouille);
  affiche(flipper);
  return 0;
}
```

- Le compilateur accepte : ratatouille et flipper sont des animaux !
- Résultat :

```
Je suis un animal
 j ai 2 pattes

Je suis un animal
 j ai 4 pattes

Je suis un animal
 j ai 0 pattes
```

- ratatouille et flipper sont traités comme des animaux !

# Un petit exemple

## Passage à la fonction

- ratatouille et flipper ont perdu leur vraie nature lors du passage à la fonction

# Un petit exemple

## Passage à la fonction

- ratatouille et flipper ont perdu leur vraie nature lors du passage à la fonction
- La fonction ne les reconnaît **que comme des objets "Animal"**

# Un petit exemple

## Passage à la fonction

- ratatouille et flipper ont perdu leur vraie nature lors du passage à la fonction
- La fonction ne les reconnaît **que comme des objets "Animal"**
- **C'est le type de l'argument dans la fonction qui détermine quelle méthode appeler et non la vraie nature de l'objet !** $\Rightarrow$ Résolution **statique** des liens
  $\Rightarrow$ La méthode appelée est fixée à la compilation

# Résolution dynamique des liens

- Solution : utiliser une résolution dynamique des liens
- On va chercher **à l'exécution** la méthode correspondant à la vraie nature de l'objet.

# Résolution dynamique des liens

- Solution : utiliser une résolution dynamique des liens
- On va chercher **à l'exécution** la méthode correspondant à la vraie nature de l'objet.
- 2 ingrédients :
  1. Les méthodes doivent être virtualisées
     $\Rightarrow$ faire comprendre au compilateur que la méthode peut varier d'un objet à l'autre
  2. Traiter une référence (et non pas une copie)
     $\Rightarrow$ à l'exécution, récupérer la vraie nature de l'objet

# Virtualisation des méthodes

- Mot-clef virtual
- Ne s'utilise que dans les prototypes (classes, ficher header), pas dans la définition

# Virtualisation des méthodes

- Mot-clef virtual
- Ne s'utilise que dans les prototypes (classes, ficher header), pas dans la définition
- Obligatoire pour la classe mère, mais pas indispensable pour les classes filles

# Virtualisation des méthodes

- Mot-clef virtual
- Ne s'utilise que dans les prototypes (classes, ficher header), pas dans la définition
- Obligatoire pour la classe mère, mais pas indispensable pour les classes filles
  - raison : si une méthode est virtuelle dans la classe mère, elle l'est forcément par héritage dans les classes filles
  - conseil : le spécifier aussi dans la classe fille, pour ne pas oublier en cours de developpement !

# Virtualisons nos animaux !

```
class Animal{
  protected :
  int nbPattes;
  public :
  Animal(int n);
  // la methode est virtualisee en vue du polymorphisme
  virtual void description();
};

Animal::Animal(int n){
  nbPattes = n;
}

void Animal::description(){
 cout<<"je suis un animal"<<endl;
 cout<<"j ai "<<nbPattes<<" pattes"<<endl;
}
```

# Virtualisons nos animaux !

```
class Rat:public Animal{
  protected :
  bool poil;
  public :
  Rat(int n, bool q);
  // la methode est virtualisee en vue du polymorphisme
  virtual void description();
};

Rat::Rat(int n, bool p):Animal(n){
  poil = p;
}
void Rat::description(){
  cout<<"je suis un rat avec "<<nbPattes<<endl;
  if(poil){
    cout<<" et je suis poilu"<<endl;
  }
  else{
    cout<<" et je n ai pas de poil"<<endl;
  }
}
```

# Virtualisons nos animaux !

```
class Dauphin:public Animal{
  protected :
  int nbNageoire;
  public :
  Dauphin(int n, int m);
  // la methode est virtualisee en vue du polymorphisme
  virtual void description();
};

Dauphin::Dauphin(int n, int m):Animal(n){
  nbNageoire = m;
}

void Dauphin::description(){
  cout<<"je suis un dauphin"<<endl;
  cout<<" je n ai aucune pattes"<<endl;
  cout<<" mais j ai "<<nbNageoire<<" Nageoires"<<endl;
  cout<<" pour nager sous l'eau"<<endl;
}
```

# Deuxième ingrédient : la référence

```
void affiche(Animal &a){
  a.description();
}

int main()
{
  Animal gafield(2);
  Rat ratatouille(4,true);
  Dauphin flipper(0,4);

  affiche(garfield);
  affiche(ratatouille);
  affiche(flipper);
  return 0;
}
```

# Deuxième ingrédient : la référence

```
void affiche(Animal &a){
  a.description();
}

int main()
{
  Animal gafield(2);
  Rat ratatouille(4,true);
  Dauphin flipper(0,4);

  affiche(garfield);
  affiche(ratatouille);
  affiche(flipper);
  return 0;
}
```

- Résultat :

```
Je suis un animal
 j ai 2 pattes

Je suis un rat avec 4 pattes
 et je suis poilu

Je suis un dauphin
 je n'ai aucune pattes
 mais j'ai 4 nageoires
 pour nager sous l'eau
```

# Virtualisons les Fluides !

```
class Fluide{
  protected :     // protected car les attributs
    double rho;   // doivent etre visibles aux
    double nu;    // classes filles
    double reynolds(double v, double l);
  public :
    Fluide(double r=1.2, double nu=0.00001);
    // la methode est virtualisee pour le polymorphisme !
    virtual double cx(double l, double v);
};
```

# Virtualisons les Fluides !

```
class Turbulent:public Fluide{
  protected : // je rajoute un attribut
    bool turbu;
  public :
    Turbulent(double r=1.2, double nu = 0.00001,
                                          bool t = False);
   // virtualisation
    virtual double cx(double l, double v);
};

class Compressible:public Fluide{
 protected : // on ajoute une methode + un attribut
   double vitSon;
   double Mach(double v);
 public :
   Compressible(double r = 1.2,double n = 0.00001,
                                          double c = 320.0);
   // virtualisation
   virtual double cx(double l, double v);
};
```

# Jonglons avec les Fluides !

```
class Ecoulement{
  private :
    Obstacle vehicule;
  public :
    Ecoulement(Obstacle o);
    double force(Fluide &f);
};

double Ecoulement::force(Fluide &f){
  double v = vehicule.vitesse();
  double l = vehicule.longueur();
  double rho = f.Densite();
  double coeff = f.cx(v,l);
  //calcul de la force
  return 0.5*rho*coeff*s*v*v;
}
```

# Jonglons avec les Fluides !

```
int main()
{
  double trainee=0.0;
  Fluide air(1.2,0.00001);
  Compressible helium(0.14,0.00002);
  Turbulent SF6(6.5,0.000001);
  Obstacle lada(4.0,10.0,0.5);
  Ecoulement cas1(lada);

  trainee = Ecoulement.force(air);
  ...
  trainee = Ecoulement.force(helium);
  ...
  trainee = Ecoulement.force(SF6);
}
```

# Cas très pratique : la surcharge !

- Un fluide est "supérieur" à un autre si son $C_x$ est supérieur pour un obstacle donné

```
bool operator>(Fluide &f1, Fluide &f2){
  bool superieur = false;
  if( f1.cx(1.0,1.0) > f2.cx(1.0,1.0) ){
    superieur = true;
  }
  return superieur;
}
```

# Cas très pratique : la surcharge !

- Un fluide est "supérieur" à un autre si son $C_x$ est supérieur pour un obstacle donné

```
bool operator>(Fluide &f1, Fluide &f2){
  bool superieur = false;
  if( f1.cx(1.0,1.0) > f2.cx(1.0,1.0) ){
    superieur = true;
  }
  return superieur;
}
```

- Je peux comparer un Fluide et un Turbulent !

```
int main(){
 Fluide air(1.2,0.00001);
 Compressible helium(0.14,0.00002);
 Turbulent SF6(6.5,0.000001);
   ...
 if( air>helium ){
   ...
 if( helium> SF6 ){
   ...
}
```

1. Historique et philosophie
2. Classe et Objet
3. La surcharge
4. Héritage
5. Objets et pointeurs
6. Polymorphisme
7. Classe abstraite

# Règles et définitions

- Définition : Un classe est abstraite si **au moins une de ses méthodes est virtuelle pure**
- Une méthode virtuelle pure est une méthode sans définition. Son prototype est alors suivi de "=0"

# Règles et définitions

- Définition : Un classe est abstraite si **au moins une de ses méthodes est virtuelle pure**
- Une méthode virtuelle pure est une méthode sans définition. Son prototype est alors suivi de "=0"
- Règles :
  - On ne peut pas instancier un objet à partir d'une classe abstraite pure
  - Lors de l'héritage d'une classe abstraite, **toutes** les méthodes virtuelles pures doivent être redéfinies, sinon la classe fille est **aussi virtuelle pure** !

# Règles et définitions

- Définition : Un classe est abstraite si **au moins une de ses méthodes est virtuelle pure**
- Une méthode virtuelle pure est une méthode sans définition. Son prototype est alors suivi de "=0"
- Règles :
  - On ne peut pas instancier un objet à partir d'une classe abstraite pure
  - Lors de l'héritage d'une classe abstraite, **toutes** les méthodes virtuelles pures doivent être redéfinies, sinon la classe fille est **aussi virtuelle pure** !

**Intérêt**

Permet de créer des classes généralistes qui doivent **obligatoirement** être rafinées dans leurs comportements (Héritage)

# Exemple : moto abstraite

| Moto |
| --- |
| - Roues (x2)<br>- Guidon<br>- Cadre |
| o virtual void demarre() |

La moto n'a pas de moteur ! La méthode demarre() n'a donc pas de sens ! Cette méthode doit être virtuelle pure !

# La moto abstraite

```
class Moto{
  protected :
    Roue roueAV;
    Roue roueAR;
    Cadre le Cadre;
    Guidon guidonDroit;
  public :
    // On peut tout de meme definir un constructeur !
    Moto(Roue r1, Roue r2, Cadre c, Guidon g);
    // ma methode est virtuelle pure !
    virtual void demarre()=0;
};

Moto::Moto(Roue r1, Roue r2, Cadre c, Guidon g);
{
  roueAV = r1;
  ....
}
```

- La méthode demarre() **ne doit pas être définie**

# Héritage de la Moto abstraite

# Héritage de la Moto abstraite : MotoThermique

```
class MotoThermique:public Moto{
  protected :
   // J'ajoute un moteur thermique
   MoteurThermique leMoteur;
  public :
   MotoThermique(Roue r1, Roue r2, Cadre c, Guidon g,
                                          MoteurThermique m);
   virtual void demarre();
};
// je redefinis la methode virtuelle
// On demarre par admission d'essence,
// d'air et une etincelle
void MotoThermique::demarre(){
  leMoteur.admissionEssence();
  leMoteur.appelDAir();
  leMoteur.etincelleBougie();
}
```

# Héritage de la Moto abstraite : MotoElectrique

```
class MotoElectrique:public Moto{
  protected :
   // J'ajoute un moteur electrique
   MoteurElectrique leMoteur;
  public :
   MotoElectrique(Roue r1, Roue r2, Cadre c, Guidon g,
                                          MoteurElectrique m);
   virtual void demarre();
};
// je redefinis la methode virtuelle
// on demarre en allumant la Batterie
void MotoThermique::demarre(){
  leMoteur.brancheBatterie();
}
```

# Il n'y a plus qu'à démarrer !

```
int main()
{
  Roue roueAV, roueAR;
  Cadre leCadre;
  Guidon leGuidon;
  // ici le compilateur renverra une erreur
  Moto motoSansMoteur(roueAV, roueAR, leCadre, leGuidon);

  MoteurThermique moteur1;
  MoteurElectrique moteur2;

  MotoThermique bandit(roueAV,roueAR, leCadre, leGuidon, moteur1);
  MotoElectrique zeroS(roueAV,roueAR, leCadre, leGuidon, moteur2);

  bandit.demarre();
  zeroS.demarre();

  return 0;
}
```

- Vous connaissez toutes les subtilités de la programmation orientée objet
  - Classes et objets
  - Surcharge (méthode et opérateur)
  - Héritage et Polymorphisme
  - Gestion des pointeurs
  - Classes abstraites
- Il ne reste que l'entraînement !

- Vous connaissez toutes les subtilités de la programmation orientée objet
  - Classes et objets
  - Surcharge (méthode et opérateur)
  - Héritage et Polymorphisme
  - Gestion des pointeurs
  - Classes abstraites
- Il ne reste que l'entraînement !

- Le C++ est assez complexe, souvent repoussant ...
- ... mais c'est le langage le plus complet, le plus puissant, le plus proche de la mémoire, le plus rapide des laguages de POO