Introduction à la programmation objet

January 30, 2023

Outline

Table of Contents

Ce que vous savez faire . . .

- Vous savez modulariser
- Vous connaissez la notion de
  - variables/types de données
  - traitements de ces données

qui apparaissaient jusqu'à présent de manière séparées.

- Traitements apparaissent à travers des fonctions, lien avec les données se fait avec le passage des arguments.
- En POO, traitements et données sont réunis dans une même entité.

Exemple impératif non-objet

```
double surface (double largeur, double hauteur) {
   return (largeur*hauteur);
}

int main(){
   double largeur=3.0;
   double hauteur=2.0;
   printf("La␣surface␣est␣%lf\n", surface(largeur,hauteur);
   exit 0;
}
```

Critique :

- pas de lien sémantique exprimé entre largeur et hauteur : on pourrait inverser par erreur les paramètres lors d'un appel et ne pas s'en rendre compte.
- de même la sémantique du calcul de la surface d'un rectangle est bcp lié au nom de fonction.

Regroupement données et traitements

Idée

Regrouper données et traitements dans une même entité qui s'appelle *rectangle*.

De manière générale POO va donner:

- de la robustesse par rapport
  - au changement
  - aux erreurs de manipulations
- de la modularité
- de la lisibilité

=> Meilleure maintanibilité

Notions

On va retrouver en POO quatre notions fondamentales

- encapsulation
- abstraction
- héritage
- polymorphisme

Encapsulation

Définition

Mettre données et traitements dans une même entité (cf exemple Rectangle) qu'on appelle objet

- On appelle
  - *attributs* : les données incluses dans l'objet
  - *méthodes* : les traitements
  - *membres* : les attributs ou les méthodes

Abstraction

- Pour être vraiment intéressant, un objet doit permettre un certain niveau d'*abstraction*.
- Le processus d'abstraction consiste à identifier pour un ensemble d'éléments
  - des caractéristiques communes à tous les éléments
  - des mécanismes communs à tous les éléments

Abstraction

Description générique de l'ensemble considéré, cacher les détails.

Exemple rectangle

- commun à tous les rectangles : calcul de surface.
- Plutôt que d'appeler à chaque fois une fonction surface avec les bons paramètres, définir 2 rectangles `R1` et `R2`.
- On ne travaille plus qu'avec des rectangles, et on peut invoquer `R1.surface()`, `R2.surface()`, c-a-d utiliser l'interface offerte par l'objet.

Abstraction (analogie)

- Analogie avec la voiture: on a juste besoin de savoir comment la conduire, avec volant, pédales, levier de vitesse, mais pas de comprendre comment marche le moteur.
- Et l'interface ne change pas ... si le moteur change: on a toujours les mêmes fonctionnalités, on ne ré-apprend pas à conduire.
- Et même si on change de voiture, *abstraction* de la notion de voiture (en tant qu'objet à conduire).

Abstraction et Encapsulation

Deux niveaux de visibilité d'un objet:

- niveau externe : partie visible par les programmeurs-utilisateurs à travers les interfaces. Par exemple, le programmeur écrira:

```
Rectangle R1;
.... R1.surface(); ...
```

- niveau interne : détails d'implémentation, dans le corps de l'objet.

Abstraction et Encapsulation

Donc 2 facettes à l'encapsulation:

1. regroupement de tout ce qui caractérise l'objet: données (attributs) et traitements (méthodes)
2. isolement et dissimulation des détails d'implémentation.

Comparons les codes:

```
double largeur=3.0;
double hauteur=2.0;
printf("Surface est %lf\n", surface(largeur,hauteur));
```

```
Rectangle R(3.0,2.0);

cout << "Surface :"<< R.surface() << endl;
```

Règle : les attributs d'un objet ne doivent pas être accessibles depuis l'extérieur (permettant ainsi de changer l'implémentation).

A retenir

- Le résultat du processus d'abstraction et d'encapsulation s'appelle une classe (classe = catégorie d'objet)
- Une classe définit un type
- Une réalisation particulière d'une classe s'appelle une instance (instance = objet)
- Un objet est une variable

Table of Contents

Classe

- Une classe en C++ se déclare avec le mot-clé `class`.

```
class Rectangle { ... } ;
```

attention: ne pas oublier le ; à la fin.

- Définit un type qu'on peut utiliser pour créer des instances.

```
Rectangle r1,r2;
```

Attributs

- Les attributs apparaissent dans la classe

```
class Rectangle {
  double hauteur;
  double largeur;
```

- et leur accès est identique aux champs d'une structure C, par notation pointée : `nom_instance.nom_attribut`.

```
Rectangle r1;
 ... r1.hauteur ...
```

Méthodes

- Ce sont des fonctions classiques dans leur définition.
- Mais il est souvent inutile de passer des paramètres car on peut souvent utiliser les attributs.
- Portée de classe : Une méthode est une fonction propre à la classe qui a accès aux attributs de la classe.

```
class Rectangle {
  double hauteur;
  double largeur;

  double surface () {
    return hauteur*largeur;
  }
};
```

Définition externes des méthodes

- Quand on veut modulariser, on peut vouloir sortir la définition de la méthode de la classe.
- On relie les méthodes aux classes auxquelles elles se rapportent en faisant précéder la méthode du nom de la classe suivant de l'opérateur : :

```
typeRetour nomClasse::nommethode(type1 param1, type2 param2, ...)
{...}
```

Méthodes: actions et prédicats

- Les méthodes qui modifient les attributs d'un objet sont parfois qualifiées d'actions et celles qui ne changent rient à l'objet de prédicat.
- On peut ajouter le mot clé `const` après les paramètres de la méthodes prédicat.

```
class Rectangle {
   double hauteur; double largeur;

   double surface ();
 };
double Rectangle::surface() const
{
  return hauteur*largeur;
}
```

Le compilateur vérfiera que la méthode ne modifie pas l'objet, sinon : `assignement of data member in read-only structure`

Appel au méthodes

Comme pour les attributs, on utilise la notation pointée :

```
Rectangle r1;

r1.surface();
```

Encapsulation et interface

Pour mettre en oeuvre l'encapsulation, C++ définit des droits d'accès aux membres d'un objet (i.e attributs et méthodes).

- Les droits d'accès sont: `public` et `private`
- Tout ce qu'il n'est pas nécessaire de connaitre a l'extérieur est `private`.
- Par défaut, les droits d'accès sont `private`.

Droits d'accès: `public` et `private`

```
class Rectangle {
  public:
    double surface (return hauteur*largeur);

  private:
    double hauteur;
    double largeur;
};
```

- Le droit d'accès `private` s'applique à tous les attributs suivants le mot-clé.
- Le compilateur vérifie, provoque une erreur par exemple sur un accès comme `rect1.hauteur` à l'extérieur de la classe.
- A l'inverse, le mot-clé `public` rend accessible de l'extérieur méthode ou attribut.

Bonne pratique

La bonne pratique est

- mettre en privé tous les attributs
- mettre en privé la plupart des méthodes
- mettre en public quelques méthodes qui représenteront l'interface

Acesseurs et manipulateurs

Si les attributs sont tous privés, comment y accéder de l'extérieur ?
Par exemple:

```
cout << "Hauteur du rectangle :" << r1.hauteur << endl;
```

Acesseurs et manipulateurs

Si les attributs sont tous privés, comment y accéder de l'extérieur ?
Par exemple:

```
cout << "Hauteur du rectangle :" << r1.hauteur << endl;
```

On définit des accesseurs (ou getters) et manipulateurs (ou setters) qui sont des métodes publiques permettant de respectivement lire et modifier les attributs.

```
r1.setHauteur(3.5);
cout << "Hauteur du rectangle :"<< r1.getHauteur() << endl;

// ... avec :
void setHauteur (double h) {
      hauteur = h;
}
double getHauteur () const { return hauteur; }
```

Masquage (shadowing) et `this`

- Rien n'empêche d'utiliser les mêmes noms pour les paramètres et les attributs

```
class Rectangle {
public:
  double setHauteur (double hauteur) {
    hauteur = hauteur; // Hmmm ... ambigu, probleme
  }
private:
  double hauteur;
  double largeur;
};
```

- Un pointeur prédéfini, appelé `this` pointe toujours vers l'objet en cours d'exécution

Masquage (shadowing) et `this`

On peut lever l'ambiguité en utilisant le pointeur prédéfini `this` qui pointe vers l'objet.

```
class Rectangle {
public:
  double setHauteur (double hauteur) {
    this->hauteur = hauteur; // ok
  }
private:
  double hauteur;
  double largeur;
};
```

Table of Contents

Initialiser les attributs

- on pourrait utiliser les setters pour initialiser individuellement chaque attribut mais pose problème car
  - l'utilisateur serait obligé de penser à initialiser les atrtibuts
  - tous les attributs devraient être exposés par des méthodes setters
- l'idée est d'avoir une méthode d'initialisation
- Les langages objets possèdent un mécanisme qui automatise cette initialisation

Constructeur

Un constructeur est une méthode:

- appelée automatiquement à la déclaration d'un objet

```
class Rectangle {
 public:
  Rectangle (double h, double l) { // constructeur
    hauteur=h; largeur=l;
  }
 };
 Rectangle r1(3.0, 2.5); // appel constructeur
```

Constructeur

Un constructeur est une méthode:

- appelée automatiquement à la déclaration d'un objet
- doit avoir une visibilité `public`

```
class Rectangle {
 public:
  Rectangle (double h, double l) { // constructeur
    hauteur=h; largeur=l;
  }
 };
 Rectangle r1(3.0, 2.5); // appel constructeur
```

Constructeur

Un constructeur est une méthode:

- appelée automatiquement à la déclaration d'un objet
- doit avoir une visibilité `public`
- chargée d'effectuer toutes les opérations requises en début de vie de l'objet (dont initialisations)

```
class Rectangle {
 public:
  Rectangle (double h, double l) { // constructeur
    hauteur=h; largeur=l;
  }
 };
 Rectangle r1(3.0, 2.5); // appel constructeur
```

Constructeur

Un constructeur est une méthode:

- appelée automatiquement à la déclaration d'un objet
- doit avoir une visibilité `public`
- chargée d'effectuer toutes les opérations requises en début de vie de l'objet (dont initialisations)
- le nom de la méthode porte le nom de la classe

```
class Rectangle {
 public:
  Rectangle (double h, double l) { // constructeur
    hauteur=h; largeur=l;
  }
 };
 Rectangle r1(3.0, 2.5); // appel constructeur
```

Constructeur

Un constructeur est une méthode:

- appelée automatiquement à la déclaration d'un objet
- doit avoir une visibilité `public`
- chargée d'effectuer toutes les opérations requises en début de vie de l'objet (dont initialisations)
- le nom de la méthode porte le nom de la classe
- pas de type de retour

```
class Rectangle {
 public:
  Rectangle (double h, double l) { // constructeur
    hauteur=h; largeur=l;
  }
 };
 Rectangle r1(3.0, 2.5); // appel constructeur
```

Constructeurs

Comme les autres méthodes, les constructeurs peuvent:

- être surchargés
- on peut donner des valeurs par défaut à leurs paramètres

Donc, une classe peut avoir plusieurs constructeurs avec des paramètres différents.

Constructeurs: exemple

```
class Rectangle {

public:
  // constructeur
  Rectangle (double h, double l) {
    hauteur=h; largeur=l;
  }

  double surface () const {
    return largeur*hauteur;
  }

private:
  double hauteur;
  double largeur;
 };
```

Construction des attributs

Que se passe t-il si les attributs sont eux même des objets ?

```
class RectangleColore {

private:
  Rectangle rectangle;
  Couleur couleur;
  // .....
 };
```

- mauvaise solution

```
RectangleColore (double h, double l, Couleur c) {
  rectangle = Rectangle(h,l);
  couleur = c;
 }
```

Car l'instruction `Rectangle(h,l);`, bien que licite, créé une instance anonyme de Rectangle, qui est ensuite copiée dans l'attribut rectangle. Donc existence de 2 rectangles dont un seul sert.

- Il faut initialiser directement les attributs !

Liste d'initialisations

Un constructeur devrait normalement contenir une section d'appel aux constructeurs des attributs ... ainsi que l'initialisation des types de base. C'est la liste d'initialisation du constructeur

- Syntaxe générale

```
NomClasse (liste parametres)
  : attribut1 (...), // appel au constructeur attribut1
  // ...
    attributN (...) // appel au constructeur attributN
 { // autres operations }
```

Notez le : pour introduire la section

Liste d'initialisations

```
class Rectange {
  Rectangle (double h, double l);
  // ....
};

class RectangleColore (double h, double l, Couleur c) {
  RectangleColore (double h, double l, Couleur c)
      : rectangle (h,l), couleur (c)
     { }

 private:
  Rectangle rectangle;
  Couleur couleur;
};
```

La liste d'initialisation utilise les mêmes noms que les attributs de la classe RectangleColore.
Exemple d'initialisation `RectangleColore r(3.0,4.5,rouge);`

Liste d'initialisations

- Les initialisations sont appelées tout de suite, avant même de rentrer dans le corps du constructeur.
- Cette liste d'initialisation est optionnelle mais recommandée.
- Les attributs non-initialisés ici
  - prennent une valeur par défaut si ce sont des objets
  - restent indéfinis si ce sont des types de base
- Les attributs initialisés ici peuvent bien sûr être modifiés par la suite.

Liste d'initialisations

Au final, notre programme ressemble à :

```
class Rectange {
public:
  Rectangle (double h, double l):
    hauteur(h), largeur(l)
  { }

  double surface() {
    return (hauteur*largeur);
  }

private:
   double hauteur;
   double largeur;
};

int main {
  Rectangle rect(3.0,4.5);
  // ...
```

On note que le constructeur n'a plus aucune instruction, assez fréquent en C++.

Constructeur par défaut

Un constructeur par défaut est un constructeur qui n'a pas de paramètre, ou dont tous les paramètres ont des valeurs par défaut.

- Exemple : ici 2 constructeurs dans la classe

```
// Le constructeur par defaut
Rectangle (): hauteur(1.0), largeur(2.0)
{ }
// Un autre constructeur
Rectangle (double h, double l): hauteur(h), largeur(l)
{}
// ...
Rectangle r; // le constructeur par defaut est appele,
Rectangle r(3.0,4.5); // L'autre constructeur est appele
```

Attention : Une erreur fréquente est de déclarer `Rectangle r();` au lieu de `Rectangle r;`. Le compilateur ne signalera pas d'erreur, il interprete cela comme le prototype d'une fonction `r` retournant un `Rectangle`.

Constructeur par défaut

L'autre cas de constructeur par défaut est celui dans lequel tous les paramètres ont des valeurs par défaut.

```
    // Le constructeur par defaut
    Rectangle (double h=1.0, double l=2.0)
      : hauteur(h), largeur(l)
    { }

// ...
```

- Si les paramètres ne sont pas passés, leur valeur par défaut est prise. `Rectangle r;` initialise un rectangle de hauteur 1 et de largeur 2
- Sinon, c'est la valeur précisée qui supplante la valeur par défaut. `Rectangle r(3.0,4.0);` utilise le même constructeur par défaut mais avec les valeurs spécifiées.

Constructeur par défaut, par défaut

- Que se passe t-il quand aucun constructeur par défaut n'a été écrit ?
- Le compilateur génère automatiquement un constructeur par défaut minimaliste qui
  - appelle le constructeur par défaut des attributs objets
  - laisse non initialisés les attributs de type de base

Attention : si au moins un constructeur existant dans la classe, le constructeur par défaut par défaut n'est plus fourni.

Constructeur de copie

Regardons ce qu'il se passe quand un objet est initialisé avec un autre objet de la même classe. C++ offre un moyen de créer la copie d'une instance : constructeur de copie.

Exemple

```
Rectangle r1( 2.5, 4.5);
Rectangle r2(r1);
```

- `r1` et `r2` sont 2 instances distinctes, mais ayant les mêmes valeurs d'attributs après la création (`r2` copie de `r1`).
- *Remarque* : quand on appelle une fonction `f(r)`, avec `Rectangle r;` et `void f(Rectangle r1)` , le passage de paramètre est par valeur, et on doit donc faire une copie de l'objet. Le constructeur de copie de la classe est invoqué.

Constructeur de copie : définition

Le constructeur de copie permet d'initialiser une instance en copiant les attributs d'une autre instance de même type.
Syntaxe:

```
NomClasse (NomClasse const & orig) { ... }
```

Exemple

```
Rectangle (Rectangle const & rorig )
: hauteur(rorig.hauteur), largeur(rorig.largeur)
  {}
```

On passe l'adresse, passage par référence (sans quoi on invoquerait récursivement la copie) et comme la copie ne modifie pas l'instance originale, on met `const`.

Quand est invoqué un constructeur de copie ?

Le constructeur de copie est invoqué dans les cas suivants:

1. Quand l'objet est retourné par valeur
2. Quand l'objet est passé en argument d'une fonction par valeur
3. Quand un objet est initialisé à partir d'un objet de la même classe
4. Quand le compilateur génère un objet temporaire

Constructeur de copie par défaut

- Un constructeur de copie est automatiquement généré par le compilateur si il n'est pas explicitement défini.
- Ce constructeur fait une copie de surface (pas copie profonde).

Destructeurs

- Motivation : action nécessaire pour libérer les ressources occupées par les instances

d'objet.

Exemple Destructeur

Exemple: soit un constructeur qui alloue de la mémoire pour ses attributs:

```
class Rectangle {
public:
  Rectangle ()
  : hauteur(new double(2.0)), largeur( new double(1.5))
    {}
private:
double *hauteur;
double *largeur;
```

Si on déclare une variable locale `Rectangle r;`, que cette variable disparaît en sortant du bloc, les zones mémoires réservées pour hauteur et largeur survivent, mais ne sont plus accessibles à travers `r`.

Destructeur nécessaire

- Tout espace alloué par `new()` doit être désalloué par le programmeur par `delete()`.
- On ne veut pas faire de désallocation des attributs en-dehors de l'objet car on casse l'encapsulation.
- C++ offre un mécanisme automatique de nettoyage

Destructeur : syntaxe

Syntaxe:

```
~NomClasse () { // operations de nettoyage... }
```

- méthode sans paramètres
- nom de la classe précédée de ~
- si non écrit, C++ génère un destructeur minimaliste

Exemple:

```
~Rectangle () {
  delete hauteur;delete largeur;
}
```

Bonne pratique

Lorsqu'on déclare explicitement un des constructeurs (intitialisation, copie) ou destructeur, on doit au moins se poser la question si il n'est pas nécessaire de définir explicitement aussi les 2 autres.

Exemple nécessité ou pas de libération mémoire

- Le cas : un objet de classe `Entreprise` mémorise son dirigeant sous la forme d'un objet de classe `Personne`

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

class Personne {
  private:
    string nom;
  public:
    Personne(string nom) : nom(nom) {};
    string get_nom() { return(nom); }

};
```

- cas 1 : le dirigeant est un pointeur sur une `Personne` existante
- cas 2 : le dirigeant est un pointeur sur une `Personne` allouée sur le tas

Cas 1 avec référence

- On utilise l'adresse de la variable initiale, aucune libération mémoire à gérer.

```
class Entreprise {
  private:
    const Personne & dirigeant; // reference
  public:
    Entreprise(const Personne & chef) : dirigeant(chef) {
      cout<<"Entreprise cree" <<endl;
  }
};

int main() {
  Personne bigboss("Pierre");
  Entreprise e(bigboss);
}
```

Cas 1 avec pointeur

- Pointeur : il faut modifier l'appel du constructeur en passant l'adresse &bigboss. On utilise l'adresse de la variable initiale, pas de libération à gérer.

```
class Entreprise {
 private:
   const Personne * dirigeant; // pointeur
 public:
   Entreprise(const Personne * chef): dirigeant(chef){
       cout<<"Entreprise␣cree" <<endl;
   }
};

int main() {
  Personne bigboss("Pierre");
  Entreprise e(&bigboss);
}
```

Cas 2 : avec allocation mémoire sur le tas

- Variante pointeur : on passe la personne par valeur, mais on veut conserver cette information dans la mémoire tas (heap). (Il y a duplication). Celui qui alloue doit libérer.

```
class Entreprise {

  private:
   const Personne * dirigeant; // pointeur
 public:
   Entreprise(const Personne chef) {
     dirigeant = new Personne(chef); // alloc avec copie
     cout<<"Entreprise cree" <<endl;
   }
   ~Entreprise() {
     delete dirigeant; // liberation obligatoire
   }
};

int main() {
  Personne bigboss("Pierre");
  Entreprise e(bigboss);
}
```

Cas 2 : variante avec liste d'initialisation

- Variante pointeur idem que précédente mais meilleure pratique : en liste d'initialisation.

```
class Entreprise {

  private:
    const Personne * dirigeant; // pointeur
  public:
    Entreprise(const Personne chef) : dirigeant(new Personne(chef)) {
        cout<<"Entreprise creee" <<endl;
    }
    ~Entreprise() {
        delete dirigeant; // liberation obligatoire
    }
};

int main() {
   Personne bigboss("Pierre");
   Entreprise e(bigboss);
 }
```

Table of Contents

Attributs de classe

- Comment peut on accéder à une zone mémoire commune à tous les objets d'une classe ?
- Par exemple, Classe `Employé` avec un age de départ à la retraite, le même pour tout le monde
- Dupliquer cette valeur en attribut constant dans chaque objet crée gaspille de l'espace
- La variable globale est possible mais brise l'encapsulation
- Solution : attribut de classe

Attribut de classe (static)

```
class Rectangle {
  // ...
 private:
  double hauteur, largeur;
  static int compteur;

 };
```

- mot clé `static`
- un attribut de classe est partagé par toutes les instances
- il existe même quand aucune instance n'est créé. On le référence par `Rectangle::compteur`
- il peut être privé ou public

Initialisation des attributs de classes

On a vu qu'un attribut peut exister avant même la création d'un classe. Comment l'initialiser ?

```
/* Initialisation de l'attribut de classe dans le fichier
   de la classe mais HORS la classe */
 int Rectangle::compteur(0);
```

Méthode de Classe

- On peut faire la même chose avec les méthodes

```
class A {

public:
  // constructeur
  static methode1 () { ...}
         methode2 () { ... }
  }
};

int main() {
  A::methode1(); // OK
  A::methode2(); // NOK
  A a;
  a.methode1(); // OK
  a.methode2(); // OK
 }
```

Restrictions des methodes de classe

Puisqu'une methode de classe peut être appelée sans objet:

- elles ne peuvent pas utiliser de méthodes d'instance ou d'attributs d'instance (y compris `this`)
- elles ne peuvent accéder qu'à d'autres méthodes de classe ou attribut de classe

Le recours à ces méthodes statiques ne se justifient que dans des cas très particuliers.

Méthode de Classe (exemple)

Timer.hpp

```
class Timer {
 private:
   static double timer;
 public:
  static void inc_timer (double duration) {
     timer += duration;
  }
  static void dec_timer (double duration) {
     timer -= duration;
  }
  static double get_timer () {
     return timer;
  }
};
// Tout attrib. de classe doit etre
// initialise hors la classe
double Timer::timer = 0.;
```

```
#include <iostream>
#include "timer.hpp"

class Personnage {
   std::string name;
   double age;
   public:
    Personnage(std::string name) : name(name), age(0) {}
    void tempo (Timer timer, double delay) {
      age += timer.get_timer() + delay;
      std::cout << name << "␣age:␣" << age << std::endl;
      timer.inc_timer(delay);
    }
  };

int main() {
  Timer timer; // seule instance necessaire
  Personnage p1("Gandalf");
  Personnage p2("Frodo");
  p1.tempo(timer, 2); // Gandalf age: 2
  p2.tempo(timer, 3); // Frodo age: 5
  return 0;
}
```

Méthode de Classe (exemple)

Démonstration que plusieurs instances impliquant l'attribut static donnent le même résultat.

Timer.hpp

```
class Timer {
 private:
   static double timer;
 public:
  static void inc_timer (double duration) {
     timer += duration;
  }
  static void dec_timer (double duration) {
     timer -= duration;
  }
  static double get_timer () {
     return timer;
  }
};
// Tout attrib. de classe doit etre
// initialise hors la classe
double Timer::timer = 0.;
```

```
#include <iostream>
#include "timer.hpp"

class Personnage {
   std::string name;
   double age;
   public:
    Personnage(std::string name) : name(name), age(0) {}
    void tempo (Timer timer, double delay) {
      age += timer.get_timer() + delay;
      std::cout << name << "␣age:␣" << age << std::endl;
      timer.inc_timer(delay);
    }
  };

int main() {
  Timer timer1; // Que se passe t-il avec
  Timer timer2; // 2 instances de timer ?
  Personnage p1("Gandalf");
  Personnage p2("Frodo");
  p1.tempo(timer1, 2); // Gandalf age: 2
  p2.tempo(timer2, 3); // Frodo age: 5
  return 0;
}
```

Table of Contents

Intérêt ?

Exemple avec les nombres complexes

```
class Complexe {
   // ....
};
Complexe z1,z2,z3;
```

- On voudrait écrire `z3 = z1 + z2;` mais + pas défini.
- On pourrait définir une fonction `add()`
- Beaucoup plus clair de pouvoir utiliser le + : il faut le surcharger

De même on aimerait surcharger l'opérateur « pour pouvoir écrire

```
cout "z3=" << z3 << endl;
```

Opérateurs

Rappel

Un opérateur est une opération sur une ou entre deux opérandes (expressions).

- opérateurs arithmétiques : `+ - * /`, ...
- opérateurs logiques : `and` , `or`, `not`
- opérateurs de comparaison: = >= <= > <=
- opérateurs d'incrément ++ --
- opérateur d'affectation =

Appel à un opérateur

Un appel à un opérateur est un appel à une fonction ou à une méthode spécifique:

```
a Op b ->  operatorOp(a,b)  ou a.operatorOp(b)
Op a   ->  operatorOp(a)    ou a.operatorOp()
```

Exemple:

```
a + b      -> operator+(a,b)       ou a.operator+(b)
cout << a -> operator<<(cout,a)  ou cout.operator<<(a)
```

Pour = c'est la méthode:

```
a = b      -> a.operator=(b)
```

Surcharge ?

- La surcharge de fonctions : des fonctions ayant le même nom mais pas les mêmes paramètres.
- Exemple:

```
int max(int,int);
double max(double,double);
```

- De la même façon, on peut écrire plusieurs fonctions pour les opérateurs:

```
Complexe operator+(Complexe, Complex);
Matrice operator+(Matrice, Matrice);
```

- Que pensez vous de :

```
int age(10);
string nom("Mathieu");
cout << "age␣de␣" << nom << ":" << age << "␣ans\n";
```

Surcharge interne et externe

La surcharge des opérateurs peut se faire soit à l'extérieur, soit à l'intérieur de la classe à laquelle ils s'appliquent.

À l'extérieur

```
Complexe operator+(Complexe, Complexe);
```

À l'intérieur

```
class Complexe {
 public:
 Complexe operator+(Complexe) const;
```

- Les opérateurs externes sont des *fonctions*
- Les opérateurs internes sont des *méthodes*

Exemple surcharge externe

- Supposons définie une classe `Complexe`

```
Complexe z1,z2,z3;
// ...
z3 = z1 + z2;
```

La surcharge externe est donc une fonction :

```
z3 = operator+(z1,z2);
```

Le prototype adéquat pour cette fonction est

```
Complexe operator+(Complexe const &, Complexe const &)
```

afin d'éviter les copies des objets.

Exemple complet Complexe

Au final, notre classe ressemble à :

```
  class Complexe {
  public:
    Complexe (double x, double y) : x(x), y(y)
    { }

    double get_x() const { return x; }
    double get_y() const { return y; }

  private:
     double x;
     double y;
  };

  const Complexe operator+(Complexe const & z1, Complexe const & z2) {
    Complexe z3(z1.get_x() + z2.get_x(),
                z1.get_y() + z2.get_y());
    return z3;
}
```

Nécessité de la surcharge externe

- La surcharge externe est nécessaire pour des opérateurs concernés par une classe mais pour lesquels la classe en question n'est pas l'opérande de gauche.
- Exemple :

```
Complexe z1, z2;
double x;

z2 = x * z1;
// Avec surcharge definie comme :
// interne -> z2 = x.operator*(z1) -> NON, x est un type simple
// externe -> z2 = operator*(x,z1) -> OK
```

Nécessité de la surcharge externe (2)

- Exemple 2:

```
 Complexe z1,z2;
 double x;

 cout << z1;
// Avec surcharge definie comme :
// interne -> cout.operator<<(z1)
// -> NON, car on surchargerait l'operateur << de la classe ostream
// et non celui de la classe Complexe
//
// externe -> operator<<(cout,z1) -> OK
// de prototype:
ostream & operator<<(ostream &,Complexe const &);
```

Définition de l'opérateur d'affichage

```
ostream & operator<<(ostream &sortie,Complexe const &z) {
  sortie << "(" << z.get_x() << "," << z.get_y() << ")";
  return sortie;
}
```

- Acceptons pour l'instant le retour en tant que référence vers la sortie ostream (il y a modification de l'affichage).

Remarque friendship

- Pas conseillé : mais si la fonction voulait accéder directement aux attributs privés `x` et `y` de la classe, elle pourrait le faire. Il faudrait alors annoncer dans la classe que cette fonction extérieure est `friend`

```
 ostream & operator<<(ostream &sortie,Complexe const &z) {
   sortie << "(" << z.x << "," << zy << ")";
   return sortie;
 }

class Complexe() {
 friend ostream & operator<<(ostream &sortie,Complexe const &z);
 // ...
private: double x,y;
};
```

Surcharge interne

Rappel

- Une surcharge interne implique une *méthode*.
- Si on écrit `z1+z2`, cela implique que des objets `z1` et `z2` ont été créés,
- et qu'on applique : `z1.operator+(z2)` : on ne donne pas l'instance courante en paramètre.

Exemple surcharge +=

```
class Complexe() {
  operator+=(Complexe const &z2) {
    x += z2.x;
    y += z2.y;
  }
 // ...
Private: double x,y;
};
```

Table of Contents

Exemple Héritage

- Imaginons qu'on doive modéliser les personnages d'un jeu ...

Valkyrie
string nom size_t vitesse size_t vie Hache hache
deplacer() rencontrer(Personnage&)

Sorcier
string nom size_t vitesse size_t vie Baguette baguette
deplacer() rencontrer(Personnage&)

P.E.K.K.A
string nom size_t vitesse size_t vie Sabre sabre
deplacer() recontrer(Personnage&)

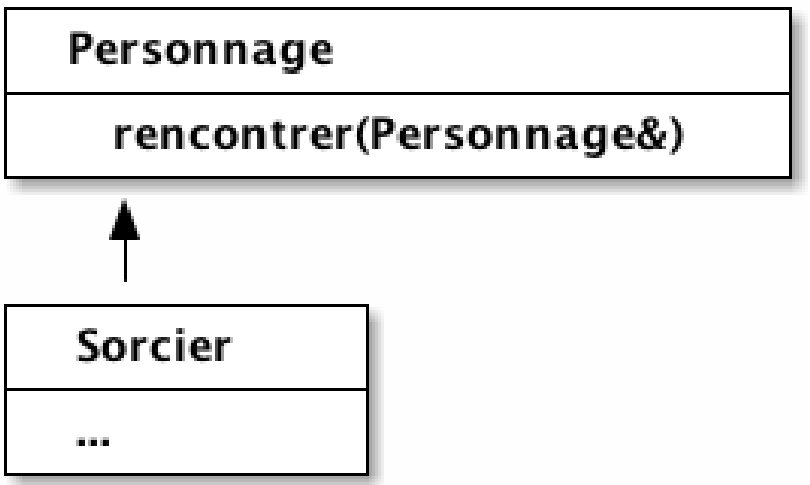
- Ces trois classes sont des déclinaisons du concept plus général de *Personnage*, capable de factoriser certains attributs et méthodes.

Héritage

- Après les notions d'encapsulation et d'abstraction, le troisième aspect essentiel de la POO est la notion d'héritage.
- L'héritage représente la relation "est-un".
- Il permet de créer des classes plus spécialisées, appelées sous-classes, à partir de classes plus générales déjà existantes, appelées super-classes.

Héritage

On peut créer ici une super-classe `Personnage` qui regroupe les notions communes à toutes sortes de personnages.



Héritage (2)

- Une sous-classe d'une super-classe C hérite de l'ensemble:
  - des attributs de C (sans avoir à les re-définir)
  - des méthodes de C (sauf les constructeurs et destructeur)
- Par ailleurs:
  - des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe : enrichissement
  - des méthodes héritées de C peuvent etre redéfinies dans la sous-classe : spécialisation

Héritage: Exemple

Une `Valkyrie` *est un* `Personnage` donc:

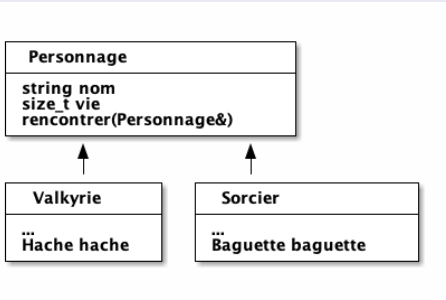
```
Personnage p;
Valkyrie v;

// ... on a le droit d'ecrire :
p = v;
// .. car un valkyrie est un personnage
// seul les attributs de p existants sont copies de v
// .. et on ne peut pas ecrire v=p

// si on considere une methode d'affichage de Personnage
void affiche(Personnage const &p);
// on peut lui passer une valkyrie
affiche(v);
// la Valkyrie etant caste en Personnage lors de l'appel
```

Héritage: Exemple

Lorsqu'une sous-classe C_1 (e.g `Valkyrie` ou `Sorcier`) est créée à partir d'une super-classe C (`Personnage`), `Valkyrie` va hériter des attributs et méthodes de `Personnage` (sauf constructeurs et destructeurs):




```
Sorcier s;
Valkyrie v;

// ... on a le droit d'ecrire :
s.rencontrer(v);

// .. de meme dans une methode de Sorcier, on
// pourrait acceder a l'attribut vie si il
// n'est pas prive

Sorcier::dec_vie() (
  vie -= ....
```

Héritage: Syntaxe

On note l'héritage public, pour une classe C1 héritant de C

```
class C { /* ... */ };

class C1 : public C { /* ... */ };
```

Ainsi :

```
class Personnage {
  // ...
 };

class Valkyrie : public Personnage {
   public:
    // constructeurs, etc.
   private:
     Hache hache;
};
```

Héritage privé

Attention : par défaut l'héritage est privé, c'est-à-dire que cette notation omettant `public` implique un héritage privé :

```
class C { /* ... */ };

class C1 : C { /* ... */ };
```

- Dans un héritage privé, même si `C1` hérite des attributs et méthodes de `C` ...
- ... on n'a plus la relation "est-un" car on ne peut plus faire de *cast* de `C1` en `C`.
- ... de plus, tous les membres protégés et publics dans `C` deviennent privés dans `C1`
- La relation doit être comprise comme "C peut être implémenté comme C1". Permet d'obtenir les attributs et méthodes interessantes mais pas forcément de relation conceptuelle entre C1 et C.

Transitivité de l'héritage

Transitivité des membres

Par transitivité, les instances d'une sous-classe possèdent :

- les attributs
- et les méthodes (hors constructeurs/destructeur)

de l'ensemble des classes parentes (super-classe, super-super-classe, etc.)

Transivité affectations

Considérons

```
class SorcierDeGlace : public Sorcier {
  ...
};
```

```
Personnage p;
Sorcier s;
p = s; // OK, un Sorcier est un Personnage

SorcierDeGlace sg;
p = sg; // OK, car par transitivite, un SorcierDeGlace est un Personnage
```

Héritage : droit d'accès `protected`

On a vu jusqu'ici que l'accès aux membres, méthodes ou attributs étaient:

- *public* : visibilité totale dans et hors la classe
- *private*: visibilité uniquement dans la classe

Héritage : droit d'accès protected

Que se passe t-il dans la situation suivante:

```
class A {
  private int a;
}

class B : public A {

  void modifie() {
      a = ...
  }
}
```

$\Leftrightarrow$ Par héritage, B hérite de l'attribut a de A mais ne peut y accéder. (!!)

Protected

- Il existe un troisième type de droit d'accès : `protected`
- Il assure la visibilité des membres d'une classe dans les classes de sa descendance.

```
class Personnage {
  protected:
    int vie;
  // ....
 };

class Valkyrie : public Personnage {
   public:
    void recoit_coup(Personnage &adversaire) {
        if (vie <= 0)
        // ...
    }
};
```

Portée de protected

- Attention : l'accès à un membre protected dans une sous-classe se fait à travers la porté de cette sous-classe.

```
class A {
  protected: int a;
};

class B : public A {

  void f( B autreB, A autreA) {
     a = 2; // OK, A::a est accessible grace a protected, on manipule B::a
     a += autreB.a; // OK, dans la meme portee (B::)
     a += autreA.a; // NOK, interdit car this->a (portee B::) n'est pas dans la meme
                    // portee que autreA.a (portee A::)
  }
};
```

Masquage : Problématique

Que se passe t-il si la méthode `rencontrer()` de la classe `Personnage` ne convient pas à toutes les sous-classes ?

- On *ajoute* une méthode spécifique `rencontrer()` pour la sous-classe qui en a besoin $\Rightarrow$ spécialisation.
- On parle alors de masquage car c'est la méthode spécialisée qui sera appelée quand on l'invoquera sur la sous-classe.

Définition

- Le masquage dans une hiérarchie désigne le fait qu'un identificateur en cache un autre.
- Cela peut concerner des attributs (cas peu fréquent) mais surtout des méthodes, ce qui est très courant et pratique.

Accès à une méthode masquée

Il est possible d'accéder explicitement à une méthode masquée en utilisant l'opérateur de résolution de portée.
Par exemple:

```
  class Sorcier : public Personnage {

    void rencontrer(Personnage &p) {
        cout << "Je␣suis␣un␣sorcier\n";
        Personnage::rencontrer(p);
    }
}
```

Constructeur et héritage

- Lors de l'instanciation d'une sous-classe, il faut initialiser:
- les attributs propres à la sous-classe (e.g `Rectangle r(3.5,6.5)`)
- les attributs hérités des super-classes

MAIS

... il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui-même l'initialisation des attributs hérités.

- L'accès à ces attributs pourrait être notamment interdit ! (`private`)
- L'initialisation des attributs hérités doit donc se faire au niveau des classes où ils sont explicitement définis.
- Solution: l'initialisation des attributs hérités doit se faire en invoquant les constructeurs des super-classes.

Constructeur et héritage : appel explicite

- L'invocation du constructeur de la super-classe se fait au début de la section d'appel aux constructeurs des attributs (liste d'initialisation).
- Syntaxe :

```
SousClasse(liste de parametres)
  : SuperClasse(Arguments), // ici initialisation par appel constructeur SuperClasse
    attribut1(valeur1), ..., attributN(valeurN)
{
    // corps du constructeur
}
```

- Lorsque la super-classe admet un constructeur par défaut, l'invocation explicite de ce constructeur dans la sous-classe n'est pas obligatoire.
- (Le compilateur se charge de réaliser l'invocation du constructeur par défaut).

Constructeurs et heritage : exemple 1

- Si la classe parente n'admet pas de constructeur par défaut, l'invocation explicite d'un de ses constructeurs est obligatoire (sinon le compilateur ne sait pas quoi faire) dans les constructeurs de la sous-classe.
- La sous-classe doit admettre au moins un constructeur explicite.
- Exemple :

```
class FigureGeometrique {
protected: Position positon;
public:
  FigureGeometrique(double x, double y) : position(x,y) {}
  // ...
};

class Rectangle : public FigureGeometrique {
  protected: double largeur; double hauteur;
  public:
      Rectangle(double x, double y, double l, double h)
       : FigureGeometrique(x,y), largeur(l), hauteur(h)
       {}
// ...
};
```

Table of Contents

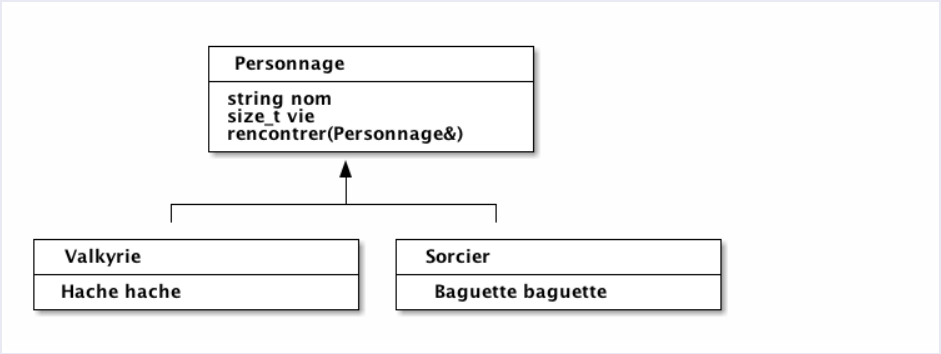
Polymorphisme

Idée générale

- De manière générale, le polymorphisme désigne la capacité d'un même code à s'adapter aux données qu'il traite.
- Important pour écrire du code générique, unifié, qui fonctionne avec différents types de données.

Exemple (Motivation)

On veut provoquer la rencontre d'un personnage `joueur` avec une série de personnages stockés dans un vecteur.



exemple d'utilisation

```
Personnage joueur;
Vector <Personnage> autres;
Valkyrie v(..);
Sorcier s(...);
autres[0] = v;
autres[1] = s;
for (auto perso : autres) {
  perso.rencontrer(joueur); //code generique, s'adapte au type de personnage

}
```

Héritage au niveau des paramètres

On a vu que l'affectation d'une instance de sous-classe dans sa super-classe était permis. Ceci vaut aussi pour les paramètres. On peut écrire par exemple:

```
class Personnage {
  public:
  //...
  void rencontrer(Personnage &p) const
  { // ... }
};

class Sorcier : public Personnage {
  // ...
};
```

```
void faire_recontrer( Personnage const & p1,
                      Personnage const & p2) {
   cout << p1.get_nom() << " rencontre " <<
   p2.get_nom() << " : " << p1.rencontrer(p2);
}

int main() {
   Valkyrie v;
   Sorcier s;
   faire_rencontrer(v,s);
}
```

Polymorphisme et résolution des liens

- Il est possible d'avoir une méthode de même nom dans une classe et ses sous-classes. Laquelle sera invoquée ?

```
class Personnage {
  public:
  //...
  void rencontrer(Personnage &p) const
  { cout << "␣Bonjour␣!\n"; }
};

class Sorcier : public Personnage {
  public:
  void rencontrer(Personnage &p) const
  { jette_boule_feu(p); }
};
```

- On appelle ce mécanisme la résolution des liens.

Polymorphisme et résolution des liens

```
class Personnage {
  public:
  //...
  void rencontrer(Personnage &p) const
  { cout << "␣Bonjour␣!\n"; }
};

class Sorcier : public Personnage {
  public:
  void rencontrer(Personnage &p) const
  { jette_boule_feu(p); }
};
```

```
void faire_recontrer( Personnage const p1,
                      Personnage const p2) {
   cout << p1.get_nom() << "␣rencontre␣" <<
   p2.get_nom() << "␣:␣" << p1.rencontrer(p2);
}


int main() {
   Sorcier s;
   Pekka p;
   faire_rencontrer(s,p); // passage par valeurs
}
```

- Quelle méthode `rencontrer()` va être appelée ? soit:
  - `faire_rencontrer()` prend en paramètre un `Personnage` et on pourrait penser que `Personnage::rencontrer()` est invoquée (le type de la variable qui prime).
  - on a substitué à `p1` un `Sorcier` et donc `Sorcier::rencontrer()` est invoquée (le type de l'objet effectivement contenu dans la variable qui prime).

Résolutions des liens

Résolution statique par défaut

- En C++, si on ne fait rien, c'est le type de la variable qui prime.
- Dans l'exemple, c'est `Personnage::rencontrer()` qui est invoquée.
- On parle de résolution statique car le compilateur peut déterminer la méthode à appeler à la compilation.

Quizz résolution

```
class Animal {
public:
   string to_string() { return "Animal"; }
};

class Dragon : public Animal {
public:
  string to_string() { return "Dragon"; }
};

int main() {
  Dragon d;
  Animal a(d);
  cout << a.to_string() << endl;
}
```

1. Il affiche `Animal`
2. Il affiche `Dragon`
3. Il ne compile pas car on ne peut pas mettre un Dragon dans un Animal

Quizz résolution

```
class Animal {
public:
   string to_string() { return "Animal"; }
};

class Dragon : public Animal {
public:
  string to_string() { return "Dragon"; }
};

int main() {
  Dragon d;
  Animal a(d);
  cout << a.to_string() << endl;
}
```

1. Il affiche `Animal`
2. Il affiche `Dragon`
3. Il ne compile pas car on ne peut pas mettre un Dragon dans un Animal

Bonne Réponse: 1. C'est le type de la variable qui prime, peu importe si on a mis un objet de type Dragon dans Animal.

Résolution dynamique des liens

- On peut vouloir choisir la méthode correspondante à la *nature réelle* de l'instance.
- Dans ce cas, il faut permettre la résolution dynamique des liens, c'est-à-dire différer le choix de la méthode lors de l'exécution.
- Pour la mettre en oeuvre, il faut
  - des méthodes virtuelles
  - appliquer ces méthodes à des pointeurs ou des références (des adresses)

Déclarations des méthodes virtuelles

- En C++ on doit indiquer au compilateur qu'une méthode peut faire l'objet d'une résolution dynamique en la déclarant comme `virtuelle` (mot clé `virtual`).
- Cette déclaration doit se faire dans la méthode la plus générale
- Toute redéfinition de la méthode dans les sous-classes héritera du caractère virtuel par transitivité.

Sur notre exemple

```
class Personnage {
  public:
  //...
  virtual void rencontrer(Personnage &p) const
  { cout << "␣Bonjour␣!\n"; }
};
```

Résolution virtuelle nécessite des adresses

Au final, la résolution dynamique de liens :

```
class Personnage {
  public:
  //...
  virtual void rencontrer(Personnage &p) const
  { cout << "␣Bonjour␣!\n"; }
};

class Sorcier : public Personnage {
  public:
  void rencontrer(Personnage &p) const
  { jette_boule_feu(p); }
};
```

```
 void faire_recontrer( Personnage const & p1,
                       Personnage const & p2) {
   cout << p1.get_nom() << "␣rencontre␣" <<
   p2.get_nom() << "␣:␣" << p1.rencontrer(p2);
}


int main() {
   Sorcier s;
   Pekka p;
   faire_rencontrer(s,p); // passage par reference
}
```

Cette fois, c'est `Sorcier::rencontrer()` qui sera appelée.

Autre exemple virtuelle/non-virtuelle

```
#include <iostream>
 using namespace std;
class Mammifere {
  public:
          Mammifere() { cout << "Nouveau␣mammifere␣!" << endl; }
    virtual ~Mammifere() { cout << "Mort␣mammifere␣!" << endl; }
          void manger() const { cout << "Mammifere␣mange." << endl; }
    virtual void avancer() const { cout << "Mammifere␣avance." << endl; }
};


class Dauphin : public Mammifere {
  public:
    Dauphin() { cout << "Nouveau␣dauphin␣!" << endl; }
   ~Dauphin() { cout << "Mort␣dauphin␣!" << endl; }
    void manger() const { cout << "Dauphin␣mange." << endl; }
    void avancer() const { cout << "Dauphin␣nage." << endl; }
};
```

Autre exemple virtuelle/non-virtuelle

Que produit le code suivant ?

```
#include "mammifere.h"
int main() {
   Mammifere* lui(new Dauphin());
   lui->avancer();
   lui->manger();
   delete lui;
   return 0;
}
```

- La ligne 1 déclare une variable `lui` de type pointeur sur `Mammifere`

Autre exemple virtuelle/non-virtuelle

Que produit le code suivant ?

```
#include "mammifere.h"
int main() {
   Mammifere* lui(new Dauphin());
   lui->avancer();
   lui->manger();
   delete lui;
   return 0;
}
```

- La ligne 1 déclare une variable `lui` de type pointeur sur `Mammifere`
- Cette variable reçoit l'adresse d'un objet de type `Dauphin` alloué dynamiquement

Autre exemple virtuelle/non-virtuelle

Que produit le code suivant ?

```
#include "mammifere.h"
int main() {
   Mammifere* lui(new Dauphin());
   lui->avancer();
   lui->manger();
   delete lui;
   return 0;
}
```

- La ligne 1 déclare une variable `lui` de type pointeur sur `Mammifere`
- Cette variable reçoit l'adresse d'un objet de type `Dauphin` alloué dynamiquement
- l'objet de type `Dauphin` et alloué par le constructeur par défaut de la classe `Dauphin`

Autre exemple virtuelle/non-virtuelle

Que produit le code suivant ?

```
#include "mammifere.h"
int main() {
   Mammifere* lui(new Dauphin());
   lui->avancer();
   lui->manger();
   delete lui;
   return 0;
}
```

- La ligne 1 déclare une variable `lui` de type pointeur sur `Mammifere`
- Cette variable reçoit l'adresse d'un objet de type `Dauphin` alloué dynamiquement
- l'objet de type `Dauphin` et alloué par le constructeur par défaut de la classe `Dauphin`
- le constructeur d'une sous-classe invoque toujours le constructeur de la super-classe

Autre exemple virtuelle/non-virtuelle

```
int main() {
  Mammifere* lui(new Dauphin());
  lui->avancer();
  lui->manger();
  delete lui;
  return 0;
}
```

- Nouveau `mammifere` ! car appel avant tout du constructeur super-classe

Autre exemple virtuelle/non-virtuelle

```
int main() {
  Mammifere* lui(new Dauphin());
  lui->avancer();
  lui->manger();
  delete lui;
  return 0;
}
```

- Nouveau mammifere ! car appel avant tout du constructeur super-classe
- Nouveau dauphin ! car appelle constructeur dauphin

Autre exemple virtuelle/non-virtuelle

```
int main() {
  Mammifere* lui(new Dauphin());
  lui->avancer();
  lui->manger();
  delete lui;
  return 0;
}
```

- Nouveau `mammifere !` car appel avant tout du constructeur super-classe
- Nouveau `dauphin !` car appelle constructeur dauphin
- `Dauphin nage.` car `avancer()` virtuelle sur adresse d'un dauphin: résolution dynamique : `Dauphin::avancer()`

Autre exemple virtuelle/non-virtuelle

```
int main() {
  Mammifere* lui(new Dauphin());
  lui->avancer();
  lui->manger();
  delete lui;
  return 0;
}
```

- Nouveau mammifere ! car appel avant tout du constructeur super-classe
- Nouveau dauphin ! car appelle constructeur dauphin
- Dauphin nage. car avancer() virtuelle sur adresse d'un dauphin: résolution dynamique : Dauphin::avancer()
- Mammifere mange. car manger() non virtuelle : résolution statique Mammifere::manger()

Autre exemple virtuelle/non-virtuelle

```
int main() {
  Mammifere* lui(new Dauphin());
  lui->avancer();
  lui->manger();
  delete lui;
  return 0;
}
```

- Nouveau mammifere ! car appel avant tout du constructeur super-classe
- Nouveau dauphin ! car appelle constructeur dauphin
- Dauphin nage. car avancer() virtuelle sur adresse d'un dauphin: résolution dynamique : Dauphin::avancer()
- Mammifere mange. car manger() non virtuelle : résolution statique Mammifere::manger()
- Mort Dauphin ! car destructeur virtuel Dauphin::~Dauphin()

Autre exemple virtuelle/non-virtuelle

```
int main() {
  Mammifere* lui(new Dauphin());
  lui->avancer();
  lui->manger();
  delete lui;
  return 0;
}
```

- Nouveau mammifere ! car appel avant tout du constructeur super-classe
- Nouveau dauphin ! car appelle constructeur dauphin
- Dauphin nage. car avancer() virtuelle sur adresse d'un dauphin: résolution dynamique : Dauphin::avancer()
- Mammifere mange. car manger() non virtuelle : résolution statique Mammifere::manger()
- Mort Dauphin ! car destructeur virtuel Dauphin::~Dauphin()
- Mort mammifere ! car ordre inverse des constructeurs Mammifere::~Mammifere()

Destructeurs virtuels 1/3

Il faut déclarer virtuel un destructeur d'une classe qui est dérivée (indice: quand une classe possède au moins une méthode virtuelle, c'est qu'elle est dérivée).

- Si le destructeur `~Mammifere()` n'était pas virtuel, `delete(lui)` n'appelerait pas le destructeur de dauphin et on aurait une fuite mémoire. Voir autre exemple ci-après.
- Par contre, comme un constructeur a toujours pour vocation d'initialiser l'instance courante, il ne peut pas être virtuel.

Destructeurs virtuels 2/3

- Autre exemple schématisé du problème quand on détruit un objet d'une sous-classe à travers un pointeur vers un objet de la classe parente.
- Considérons le code suivant:

```
class A {
   // ...
};

class B : public A
{
   ~B()
   {
      // liberation memoire
   }
};
```

- Notez que le destructeur par défaut de `A` n'est pas virtuel.

Destructeurs virtuels 3/3

- Et imaginons maintenant le code suivant

```
A *monObj = new B;
// ... utilisation de monObj ...
delete monObj; // probleme!
```

- Le type statique de `monObj` est `A` tandis que son type dynamique est `B`.
- L'invocation du destructeur, si il n'est pas virtuel, va exécuter le destructeur de `A` et non celui de l'objet réel de type `B`

Table of Contents

Trois idées différentes

Nous avons vu:

- la surcharge (*overloading*) de méthodes ou d'opérateur
- le masquage (*shadowing*) de méthodes
- les méthodes virtuelles, qu'on appelle aussi re-définition (*overriding*) ou substitution

Masquage, redéfinition et surcharge : définitions

Rappel

- surcharge : même nom, mais paramètres différents, dans la même portée
- masquage : entités de même nom mais de portées différentes, masquées par les règles de résolution de portée (la portée la plus proche masque la portée plus lointaine).
- redéfinition des méthodes virtuelles : redéfinir dans une sous-classe une méthode héritée d'une super-classe.

Masquage, redéfinition et surcharge : Exemple (1/3)

```
class A {
  public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};
  class B : public A {
  public:
    // substitution de l'une des deux, l'autre devient hors de porteee (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
  };
  class C : public A {
  public:
   // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
  };
```

Masquage, redéfinition et surcharge : Exemple (2/3)

```
int main() {
    B b;
  //b.m1(2); // NON : compilateur: no matching function for call to 'B::m1(int)'
    b.A::m1(2); // ... mais elle est bien laa
    b.m1("2");

    C c;
    c.m1(2); // Attention ici : c'est celle avec double (celle avec entier est masquee) !!
  //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2); // OK, et la c'est celle avec int

    return 0;
}
```

Masquage, redéfinition et surcharge : Exemple (3/3)

- Pour voir les effets de la redéfintion des méthodes virtuelles, il faut utiliser des pointeurs, que nous voyons sur ce nouvel exemple.

```
int main() {
  B b;
  C c;
  A* pa(nullptr);
  pa = &b;
  pa->m1("2");
  pa->m1(2); // OK (nous sommes dans A::)
  pa = &c;
  pa->m1(2.1); // Attention ici : c'est celle avec int !! il y a conversion double->int
               // Nous sommes dans A::
  // pa->C::m1(2.1); // Impossible ! ne compile pas car A n'herite pas de C !!
  return 0;
}
```

Mots clé `final` et `override`

- A partir de C++11, des mots clés permettent au programmeur d'indiquer au compilateur ses intentions pour se prémunir d'erreurs.
  - avec le qualificatif `override` pour dire qu'il pense substituer/redéfinir une méthode virtuelle
  - avec le qualificatif `final`, empêcher la substitution/redéfinition future d'une méthode virtuelle.

Mots clé final et override

```
class A {
  // ...
  virtual void f1();
  virtual void f2() const;
        void f3(); // non virtuelle (oubli?)
  virtual void f4() final; // pas de redefinition
};
class B : public A {
  // ...
  virtual void f1() override; // OK
  virtual void fl() override; // Erreur faute de frappe : 1 <-> l
  virtual void f2() override; // Erreur: a oublie le const
        void f3() override; // Erreur: non virtuelle
  virtual void f4(); // Erreur : f4 etait final
};
```

Table of Contents

Méthodes virtuelles pures

Nous allons voir que le polymorphisme aide à l'abstraction

- Au sommet d'une hiérarchie de classes
  - il n'est pas toujours possible de donner une défintion de méthodes qui conviennent à toutes les sous-classes
  - . . . même si toutes les sous-classes vont effectivement implanter ces méthodes.

Exemple : besoin de méthode virtuelle pure

```
class FigureFermee {
  // ...
  // difficile a definir a ce niveau !..
  virtual double surface(...) const { ??? }
  virtual double perimetre(..) const {???}

  // ...pourtant la methode suivante en aurait besoin !
  double volume(double hauteur) const {
     return hauteur * surface();
  }
};
```

- Ici, difficile de calculer la surface pour un objet aussi général.
- Même si toutes les figures fermées concrètes (e.g un cercle) vont avoir une méthode `surface()`.
- Cette méthode pourrait même etre utilisée: par exemple pour `volume()`.
- On la déclare virtuelle pure

Méthode virtuelle pure: définition et syntaxe

Défintion

Une méthode virtuelle pure (on dit aussi abstraite) est une méthode destinée à être redéfinie par ses sous-classes.

- Les sous-classes de la classe définissant une méthode abstraite doivent redéfinir cette méthode
- Syntaxiquement cette méthode comporte =0 en fin de prototype
- Cette méthode contient généralement un corps vide.

```
class FigureFermee {
   virtual double surface(...) const = 0;
   virtual double perimetre(..) const = 0;
// ...
};
```

Classe abstraite

Définition

Une classe abstraite est une classe qui contient au moins une méthode abstraite.

- Elle *ne peut pas être instanciée*
- Ses sous-classes restent abstraites tant qu'elles n'ont pas redéfini toutes ses méthodes abstraites.

Classe abstraite: exemple

```
class Cercle: public FigureFermee {
public:
  double surface() const override {
    return M_PI * rayon * rayon;
  }
  double perimetre() const override {
   return 2.0 * M_PI * rayon;
 }

protected:
   double rayon;
};
```

La classe `Cercle` n'est pas une classe abstraite car elle redéfinit et implémente la méthode abstraite `surface()`.

Classe abstraite: exemple

```
class Polygone: public FigureFermee {
public:
  double perimetre() const override {
    double p(0.0);
    for (auto cote : cotes) { p += cote; }
    return p;
  }

protected:
   double rayon;
}
```

La classe `Polygone` reste abstraite car `surface()` n'a pas été définie.

Table of Contents

Introduction

- On rappelle qu'un intérêt majeur du ploymorphisme est d'appliquer un traitement syntaxiquement concis à des objets de classes différentes.

```
vector<Personnages *> lpersos;
Sorcier s1(10);
Valkyrie v1(20);
Sorcier s2(9);
lpersos.push_back(&s1);
lpersos.push_back(&v1);
lpersos.push_back(&s2);

for (auto p : lpersos)
  p->deplace();
```

- Ici, `lpersos` est hétérogène si elle stocke des objets de sous-classes différentes de `Personnage`.
- Le polymorphisme permet d'appeler la méthode de l'objet réellement stocké dans cette liste.

Collection hétérogène

- Parfois, on a besoin de déterminer le type réel de l'objet, qui ne peut être connu qu'à l'exécution.
- Supposons qu'on veuille retirer de cette collection, uniquement les objet de type `Sorcier`. Comment faire ?

Collection hétérogène

- Parfois, on a besoin de déterminer le type réel de l'objet, qui ne peut être connu qu'à l'exécution.
- Supposons qu'on veuille retirer de cette collection, uniquement les objet de type `Sorcier`. Comment faire ?

```
vector<Sorcier *> lsorciers;
// on ne selectionne que les sorciers
 for (auto p : lpersos) {
    Sorcier *s = dynamic_cast<Sorcier*>(p);
    if (s) // s==nullptr if dynamic_cast failed
       lsorciers.push_back(s);
 }
```

- `dynamic_cast` permet, à l'exécution, une conversion de type vérifiée inoffensive.
- Rend `nullptr` si la conversion n'est pas possible (par ex: une `Valkyrie`, ou une classe hors de la hierarchie).

Collection hétérogène

- Autre moyen d'inspecter le type réel d'un objet à l'exécution : `typeid`
- Donne le type d'une variable ou d'une expression
- On pourrait l'utiliser comme alternative dans l'exemple précédent

```
vector<Sorcier *> lsorciers;
// on ne selectionne que les sorciers
for (auto p : lpersos) {
  cout << "element␣de␣type␣:" << typeid(*p).name() << endl;
  if (typeid(*p)==typeid(Sorcier))
    lsorciers.push_back(dynamic_cast<Sorcier *>(p));
 }
```

Collection de pointeurs

- La solution consiste à utiliser un vecteur de pointeurs.

```
class Jeu {
  // ...
private:
   vector<Personnage *> personnages;
   //...
};
```

ou, alternative C++11 avec smartpointers:

```
#include <memory>
class Jeu {
  //...
private:
   vector<unique_ptr<Personnage>> personnages;
   //...
};
```

Exemple méthodes de Jeu

Comment continuer la classe Jeu ?

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier (...));
```

Donc:

```
class Jeu {
  public:
    void ajouter_personnage(Personnage *nouveau) {
        if (nouveau != 0)
          personnages.push_back( nouveau );
    }

    void afficher() const {
        for (auto p : personnages)
           p->afficher();
    }

  // ...
  private:
    vector<unique_ptr<Personnage>> personnages;}
;
```

Exemple méthodes de Jeu (c++11)

Comment continuer la classe Jeu ?

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier (...));
```

Donc:

```
class Jeu {
public:
  void ajouter_personnage(Personnage *nouveau) {
    if (nouveau != nullptr)
      personnages.push_back(unique_ptr<Personnage>(nouveau));
  }
  void afficher() const {
    for (auto &p const : personnages) // subtilite des unique_prt :
      p->afficher(); // sans reference &p, p est un
  } // 2e pointeur pointant vers la
                                // meme case memoire : interdit

  // ...
private:
  vector<unique_ptr<Personnage>> personnages;}
  ;
```

Risque d'une collection de pointeur

Il faut être certain que tous les éléments de la collection existent vraiment.
Exemple:

```
void creer_magicien(Jeu &jeu) {
  Magicien m(..);
  jeu.ajouter_personnage(m);
}

int main() {
   Jeu mon_jeu;
   creer_magicien( mon_jeu);
   mon_jeu.afficher(); // aie, probleme
   return 0;
}
```

Table of Contents

Qu'est ce que l'héritage multiple ?

Héritage multiple

En C++ une sous-classe peut hériter de plusieurs super-classes

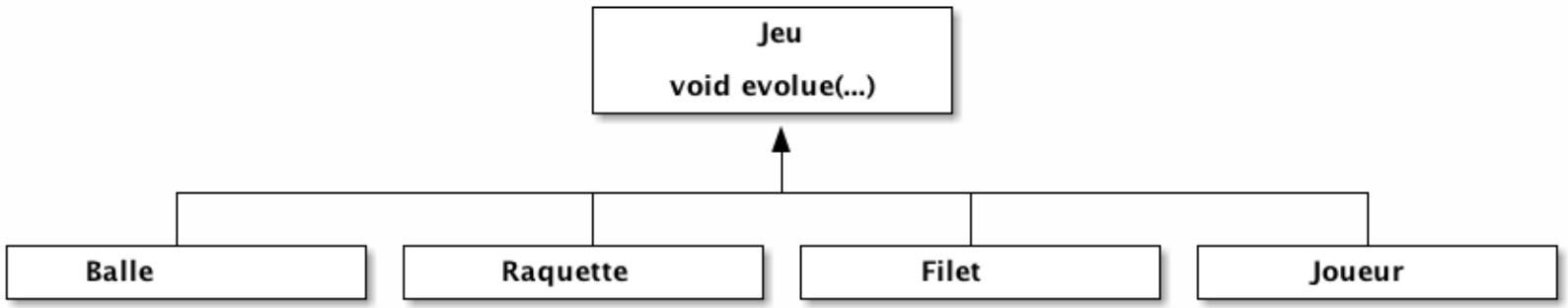
- Comme pour l'héritage simple, la sous-classe hérite de tous
  - les attributs
  - les méthodes sauf constructeurs/destructeurs
  - les types
- Tous les langages orientés objets ne permettent pas l'héritage multiple

Syntaxe

```
class Sousclass : public Superclass1, public SuperclassN {
    // ....
};
```

Exemple jeu Tennis

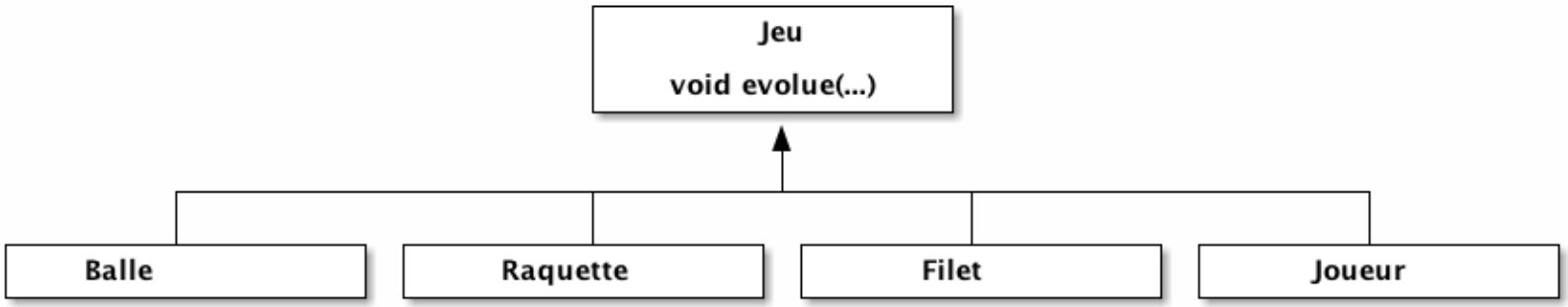
- Un jeu met en oeuvre les entités:
  - Balle
  - Raquette
  - Filet
  - Jouer
- Chaque entité sera doté d'une méthode `evolue()` gérant l'évolution de l'entité dans le jeu.



Exemple jeu Tennis

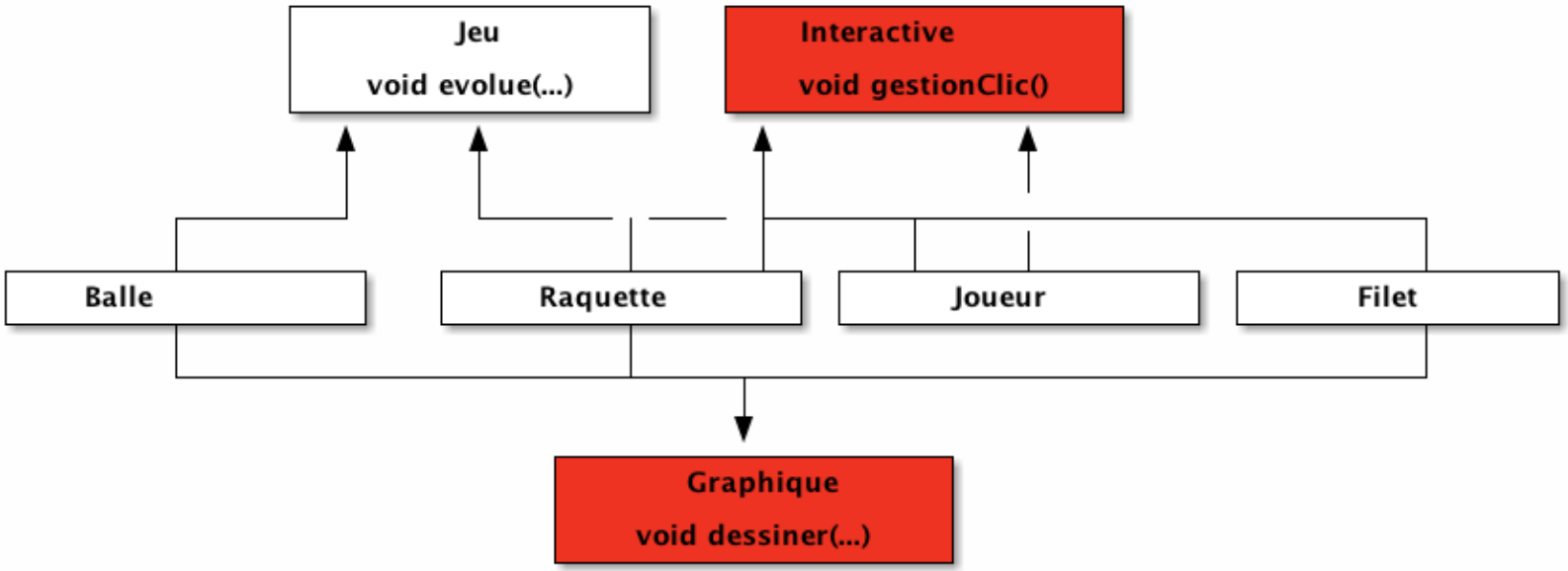
Imaginons maintenant les besoins suivants:

- gérer le contrôle par clics de souris du joueur et de la raquette
- gérer l'affichage graphique: tout sauf le joueur (RPG)



Exemple jeu Tennis

- Faire hériter des classes supplémentaires suivantes



Exemple

```
class Ovovivipare : public Ovipare, public Vivipare {
public:
  Ovovivipare (unsigned int, unsigned int) ;
  vitual ~Ovivipare();
protected:
  bool espece_rare;
};
```

- L'ordre de déclaration des super-classes est pris en compte lors de l'invocation des constructeurs et destructeurs

Constructeurs

- Comme pour l'héritage simple, l'initialisation des attributs hérités doit être faite par l'invocation des constructions des super-classes:

```
class Sousclass : public Superclass1,
                  public SuperclassN {

                  // ....
}
: Superclass1(arguments1),
  SuperclassN(argumentsN),
  attribut1(valeur1),
  attributM(valeurM)
  {}
```

- Si une superclasse admet un constructeur par défaut, il n'est pas nécessaire de l'invoquer explicitement.

Crédits

- Ce document est construit à partir du cours *Initiation à la programmation en C++* de Jean-Cédric Chappelier, Jamila Sam, Vincent Lepetit - EPFL
- Les illustrations de personnages sont celles du jeu Clash of Clans (SuperCell)