Introduction à la programmation orientée objet

Yannick Pencolé

CNRS-LAAS, Vacataire à l'INSA de Toulouse, FRANCE

2023-2024

Chargé de TD (5 séances) : Yannick Pencolé
Chargés de TP (2 séances) : Yannick Pencolé, Aya Attia

Paradigmes de programmation

- **Programmation impérative** : programme = séquence d'ordres, d'instructions
- **Programmation procédurale** : programme = ensemble de procédures impératives
- **Programmation fonctionelle** : programme = ensemble de fonctions (pas de variables, pas d'itération)
- **Programmation logique** : programme = ensemble de formules logiques
- Programmation orientée objet : programme = ensemble d'objets qui intéragissent entre eux. Extension de la programmation procédurale.

Paradigmes $\neq$ Langages

- Un paradigme est un ensemble de concepts de programmation
- Un langage de programmation, c'est un moyen de programmer à partir d'une syntaxe
- Un langage de programmation peut exploiter un ou plusieurs paradigmes de programmation (langage multi-paradigmes)
- POO : un ensemble de concepts s'appliquant à de nombreux langages de programmation

Langages dits orientés-objets

- Un langage orienté objet est un langage qui met en oeuvre le paradigme POO
- *Simula 67* est le langage précurseur (1967)
- *Smalltalk* (1972, 1980)
- *C++* (1985)
- *Eiffel* (1986)
- *Python* (1991)
- *Ada95* (1995)
- *Java* (1995)
- *Ruby* (1995)
- *C#* (2001)
- + D, PHP4, ocaml, delphi....

Langage C++

- Le langage cible pour ce cours de POO
- C++ langage multi-paradigmes (procédurale, OO, générique, fonctionnelle)
- C++ = better C (Bjarne Stroustrup, son créateur)
- C++ est compatible avec C (voir la gentille initiation au C)
- Langage complexe, riche, puissant, performant et en constante évolution (C++98,11,14,17,20,23)
- Difficile à maitriser
- Ce cours n'est pas une introduction au C++
- Ce cours propose un moyen de mettre en œuvre les principes de POO en C++

Objectifs de ce cours

1. Introduction aux concepts fondamentaux de la POO
2. Application de ces concepts en C++
3. Modélisation orientée objet à l'aide du langage graphique UML
4. Traduction de cette modélisation UML en C++

Préambule : bibliothèque standard du C++

- Ensemble de fonctions, structures de données C++
- On l'appelle aussi STL : *Standard Template Library*
- Espace de nommage : `std`

Dans ce cours, nous utiliserons quelques fonctions et structures de données de la STL.

Préambule : chaînes de caractères en C++

- Utilisation de `string`
- Beaucoup plus flexible que les `char[]` du C
- `string` fait partie de la bibliothèque standard du C++
- Son vrai nom : `std::string`

Exemple :

```
#include<string>

int main() {
  std::string chat = "chat";
  std::string chou = chat;
  chou[2] = 'o';
  chou[3] = 'u';
  std::string chouchou = chou + chou;
  std::string chouchoute(chouchou);
  chouchoute += "te";
}
```

Préambule : entrée/sortie en C++ → cout

- Utilisation de l'opérateur de flux de sortie `cout` au lieu de `printf`.
- Son vrai nom → `std::cout`
- Retour chariot → `std::endl`

Exemple :

```
#include<iostream>

int main() {
  std::cout << "chat" << std::endl;
  std::cout << "Le langage C++ a " << 37 << " ans ,"
  std::cout << "il est ne en " << (2022-37) << '.' << '\n';
}
```

Préambule : entrée/sortie en C++ → cin

- Utilisation de l'opérateur de flux de d'entrée `cin` au lieu de `scanf`.
- Son vrai nom → `std::cin`

Exemple : si la chaîne suivante est sur l'entrée standard :

`Mon numéro est 06 24.`

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

int main() {
  std::string s1, s2, s3;
  char p;
  int n1, n2;
  std::cin >> s1 >> s2 >> s3 >> n1 >> n2 >> p;
}
```

Alors on obtient :

`s1=="Mon"`, `s2=="numéro"`, `s3=="est"`,
`n1==6`, `n2==24`, `p=='.'`.

Préambule : tableau en C++ → vector

- Tableau en C efficace mais très inflexible (taille fixe)
- En C++, il est préférable d'utiliser `std::vector<T>`
- `std::vector<T>` est un vecteur d'éléments de type `T`
- `T` est un type quelconque.

Exemple :

```
#include<vector>
#include<string>
using namespace std;
int main() {
  vector<int> entiers(5); // vecteur de 5 entiers
  vector<string> chaines(10);
  // vecteur de 10 chaines de caracteres
  vector< vector<int> > matrice;
  vector< int * > pointeurs(23);
  // vecteur de 23 pointeurs
}
```

Préambule : utilisations possibles de vector

```
#include<vector>

int main()
{
  //tableau de 4 entiers
  std::vector<int> entiers = { 1 , 2 , 3 , 4 };
  std::vector<int> entiers2;
  entier2 = entiers; // copie
  //affichage
  for(size_t i = 0; i < entiers.size(); ++i)
  {
    cout << entiers[i] << " ";
  }

  // modifications
  entiers[0] = 0; // 0 remplace 1
  entiers.push_back(5); // contenu 0, 2 , 3, 4, 5 size =5
  entiers.pop_front(); // contenu 2, 3, 4, 5 size = 4
  entiers.clear();
  if(entier.empty()) { cout << "vide\n"; }
}
```

Premiers pas vers la notion d’objet...

Premiers pas vers la notion d'objet...

Supposons que l'on désire écrire un programme qui gère des personnes.

Quelques attributs d'une personne :

- elle a un nom de famille
- elle a un prénom
- elle a un age

Comment représenter les attributs d'une personne dans un programme ?

- les attributs doivent pouvoir être consultés
- les attributs doivent pouvoir être modifiés

Première représentation : stockage brut

```
#include<vector>
#include<string>
using namespace std;
...
vector<string> noms;
vector<string> prenoms;
vector<unsigned> ages;
```

- Chaque attribut est stocké dans une structure de données à part.
- La personne `i` a le nom `noms[i]`, le prénom `prenoms[i]` et l'âge `ages[i]`.
- Avantages : type de données élémentaires, codage direct
- Inconvenients :
  - Gestion des personnes difficiles (ajout/suppression/modification)
  - Toujours garantir que `noms[i]`, `prenoms[i]` et `ages[i]` correspondent à la même personne.
  - Comment un développeur tiers peut-il être en mesure de comprendre dans ce programme que le triplet (`noms[i]`, `prenoms[i]` et `ages[i]`) représente une personne ?

Deuxième représentation : stockage structuré

Une personne est un concept que le programme doit savoir manipuler, création d'un type structuré `Personne`.

```
#include<vector>
#include<string>
...
struct Personne
{
    string nom;
    string prenom;
    int age;
};

...
vector<Personne *> personnes;
// vecteur de pointeurs sur des personnes
```

- Une personne est représentée par une seule variable de type `struct Personne`.
- Les attributs de la ième personne sont :
  - `personnes[i]->prenom`
    (note : `personnes[i]` est un pointeur : notation fléchée `->`)
  - `personnes[i]->nom`
  - `personnes[i]->age`

Type abstrait = encapsulation des données

- `struct Personne` est un type abstrait
- Un type abstrait encapsule d'autres types (abstraits ou non) pour former un nouveau concept.
- L'utilisation de type abstrait :
  - organise les données
  - plus compréhensible
  - plus modulaire, plus réutilisable

Manipulation de Personne : mode direct

```
#include<vector>
#include<string>
...
struct Personne
{
    string nom;
    string prenom;
    int age;
};
```

Le programmeur modifie les champs d'une personne directement :

```
Personne personne; // personne n'est pas un pointeur: notation pointée
personne.nom = "McFly";
personne.prenom = "Marty";
personne.age= 17;
cout << personne.prenom << " " << personne.nom
     << " a " << personne.age << " ans\n";
```

Pas gérable au long terme, sujet à des erreurs de logique :

```
personne.nom = "Brown"; personne.prenom = "Marty";
personne.age= 71;
```

Manipulation de Personne : mode structuré

```
#include<vector>
#include<string>
...
struct Personne
{
   string nom;
   string prenom;
   int age;
};
Personne * creerPersonne(string n,
                         string p,
                         int a);
void changerAge(int nouvelAge, Personne * personne);
....
```

Le programmeur utilise des fonctions liées au type `Personne`, moins source d'erreur :

```
Personne * marty = creerPersonne("McFly","Marty", 17);
Personne * george = creerPersonne("McFly","George", 47);
Personne * doc = creerPersonne("Brown","Emmett", 71);
// back to 1955
changerAge(george,17);
Personne * docJeune = creerPersonne("Brown","Emmett", 41);
```

Notion de classe

- Type abstrait : encapsulation des données
- Mais pourquoi ne pas aller plus loin et encapsuler les fonctions manipulant ces données ?
- Classe : un type abstrait associé à des fonctions de manipulation

```
class Personne
{
   string nom;
   string prenom;
   int age;

   Personne(string  n,
            string  p,
            int a);

   void changerAge(int nouvelAge);
};
...
...
Personne george("McFly" ,"George" , 47);
george.changerAge(17);
```

C'est bien un cours de POO ? Non ?

Oui. (ouf !)

- Classe : notion fondamentale en POO
  - Une classe est un type abstrait de données muni d'une implémentation éventuellement partielle. (B. Meyer)

- Objet : zone mémoire de l'ordinateur pour le stockage d'une variable dont le type est une classe (données + liens $\rightarrow$ fonctions encapsulées).
  - Tout objet est instance d'une certaine classe. (B. Meyer)

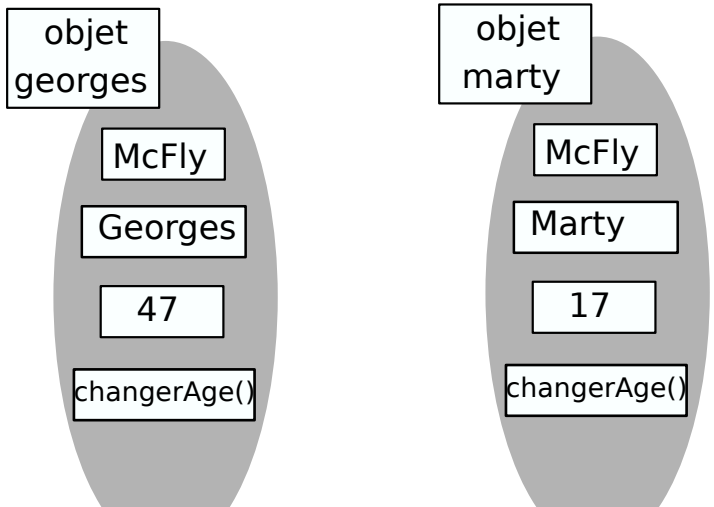
- Le programmeur programme des classes et leurs intéractions.
- L'ordinateur instancie des objets qui communiquent entre eux.

Classes versus Objets

Code du programme décrivant des classes.

```
class Personne
{
   string nom; string prenom; int age;
   Personne(string n, string p,int a);
   void changerAge(int nouvelAge);
};
int main()
{
  Personne george("McFly" ,"George" , 47); // instanciation george
  Personne marty("McFly" ,"Marty" , 17); // instanciation de marty
  ...
}
```

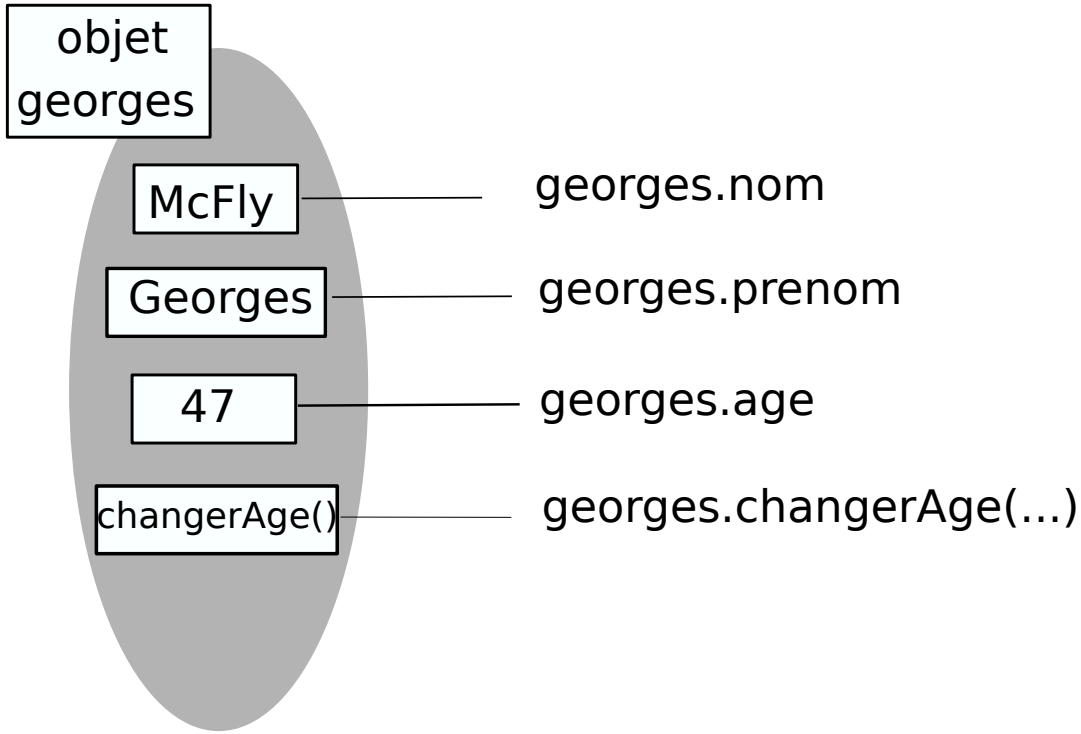
Objets du programme instancié (stocké) en mémoire :



Accès aux données/fonctions d'un objet

Notation pointée

```
cout << george.age << '\n'; // 47
george.changerAge(17);
cout << george.age << '\n'; // 17
```

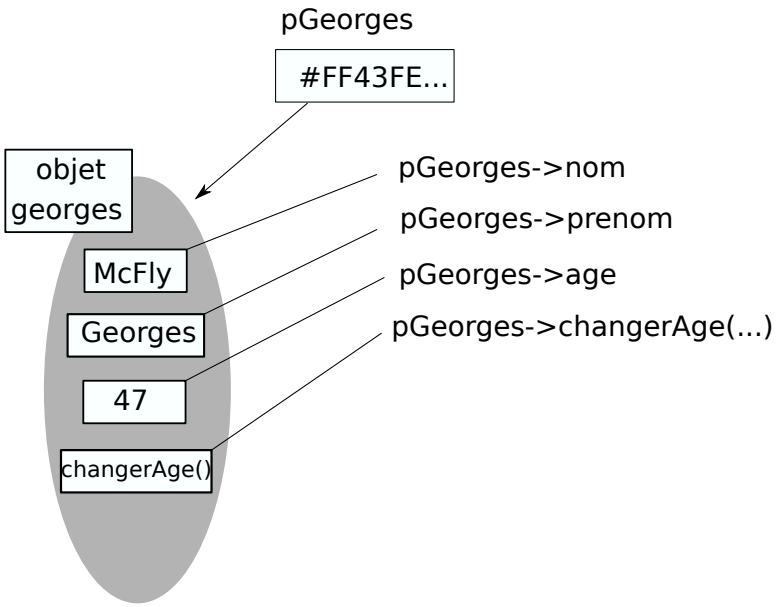


`george.age` est une variable entière.
`george.changerAge(..)` est une fonction.

Accès aux données/fonctions d'un objet

Notation flèchée (par pointeur)

```
Personne * pGeorge = &george;
cout << pGeorge->age << '\n'; // 47
pGeorge->changerAge(17);
cout << george.age << '\n'; // 17
```



`pGeorge->age` est une variable entière.
`pGeorge->changerAge(..)` est une fonction.
(autre notation : `(*pGeorge).age` et `(*pGeorge).changerAge(..)`)

Un peu de vocabulaire

```
class Personne
{
   string nom;
   string prenom;
   int age;

   Personne(string n,
            string p,
            int a);
   ~Personne();
   void changerAge(int nouvelAge);
};
```

- `nom`, `prenom`, `age` sont des attributs.
- `Personne(string n, string p,int a)` est un constructeur.
- `(string n, string p, int a)` sont les paramètres du constructeur.
- `~ Personne()` est l'unique destructeur.
- `void changerAge(int nouvelAge)` est une méthode.

Restriction d'accès aux attributs

À quoi bon coder la méthode

`void changerAge(int nouvelAge)` ?

si l'utilisateur a un droit d'accès à l'attribut `age` ? inutile et redondant ?

```
george.age = 17;
george.changerAge(17);
```

Règles de conception orientée objet :

1. Un attribut est interne à un objet, seule une méthode de l'objet a le droit d'accéder à un attribut de l'objet.
2. Interdire l'accès publique à un attribut de l'objet

Restriction d'accès aux attributs en C++

```
class Personne
{
 private: // acces prive
   string _nom; // astuce: nommer un attribut avec '_'
   string _prenom;
   int _age;

 public: // acces publique
   Personne(); // constructeur par défaut
   Personne(string nom, string prenom,
            int age);
   ~Personne();

   // mutateurs: modifient l'etat interne de l'objet
   void changerAge(int nouvelAge);
   void changerPrenom(string nouveauPrenom);
   void changerNom(string nouveauNom);

   // accesseurs: consultent l'etat interne de l'objet
   int age();
   string  prenom();
   string  nom();
};
```

Utilisation de la classe Personne (allocation statique).

```
Personne george("McFly","George",47); // constructeur paramétré
cout << george.prenom() << " "
     << george.nom() << " a "
     << george.age() << " ans en 1985.\n";

george.changeNom("Washington");
george.changeAge(67);
cout << george.prenom() << " "
     << george.nom() << " est décédé à l'âge de "
     << george.age() << " en 1799.\n";

Personne marty; // constructeur par défaut
marty.changeAge(17); marty.changeNom("McFly");
marty.changePrenom("Marty");
```

L'utilisation de `george._age`, `george._nom`, `marty._age` ... est interdite (ne compilera pas).

Même exemple en allocation dynamique

```
// allocation dynamique de l'objet george
Personne * george = new Personne("McFly","George",47);
cout << george->prenom() << " "
     << george->nom() << " a "
     << george->age() << "ans en 1985.\n";

george->changeNom("Washington");
george->changeAge(67);
cout << george->prenom() << " "
     << george->nom() << " est décédé à l'âge de "
     << george->age() << " en 1799.\n";

Personne * marty = new Personne(); // constructeur par défaut
marty->changeAge(17); marty->changeNom("McFly");
marty->changePrenom("Marty");

//destruction des objets en fin d'utilisation
delete george; george = nullptr;
delete marty; marty = nullptr;
```

Règles de l'allocation dynamique

1. Seulement l'opérateur new tu utiliseras (pas de malloc)

   Création d'un objet de classe C

   ```
   C * instance = new C(param1, ..., paramN);
   ```

   où `C(param1, ..., paramN);` est un constructeur.

2. Tout objet que tu créeras, le détruire tu devras avec l'opérateur delete (pas de free).

   Destruction d'un objet de classe C

   ```
   delete(instance); instance = nullptr;
   ```

Mise en œuvre de la classe Personne

- Pour le moment, on n'a fait que spécifier la classe Personne.
- La spécification s'écrit (en général) dans le fichier `Personne.h`
- L'utilisateur de la classe Personne n'utilise que ce fichier en utilisant `#include"Personne.h"`

```
#ifndef __PERSONNE_H
#define __PERSONNE_H
#include<string>

class Personne {
...
};

#endif
```

- La mise en œuvre des constructeurs, des méthodes et de destructeur est décrite en général dans le fichier `Personne.cpp`
- L'utilisateur de la classe Personne n'a pas à connaître cette mise en œuvre.

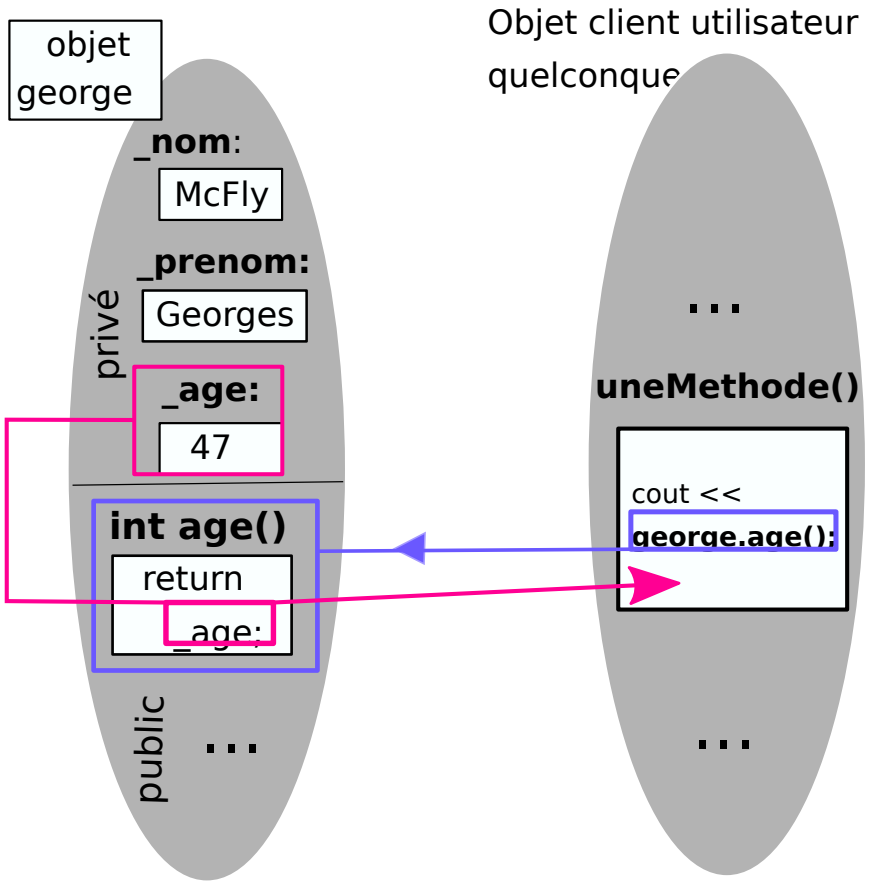
Forme générale du fichier Personne.cpp

```
#include"Personne.h"

// on inclut la spécification puis on met en oeuvre


// mise en oeuvre des constructeurs
Personne::Personne(): ...  // constructeur par défaut
{ ... }

Personne::Personne(string  nom , string prenom ,
                   int age ): ... // constructeur paramétré
{ ... }

// mise en oeuvre des accesseurs
int Personne::age()
{ ... }

...

// mise en oeuvre des modificateurs/mutateurs
void Personne::changerAge(int nouvelAge)
{ ... }

...
```

Mise en œuvre d'un accesseur

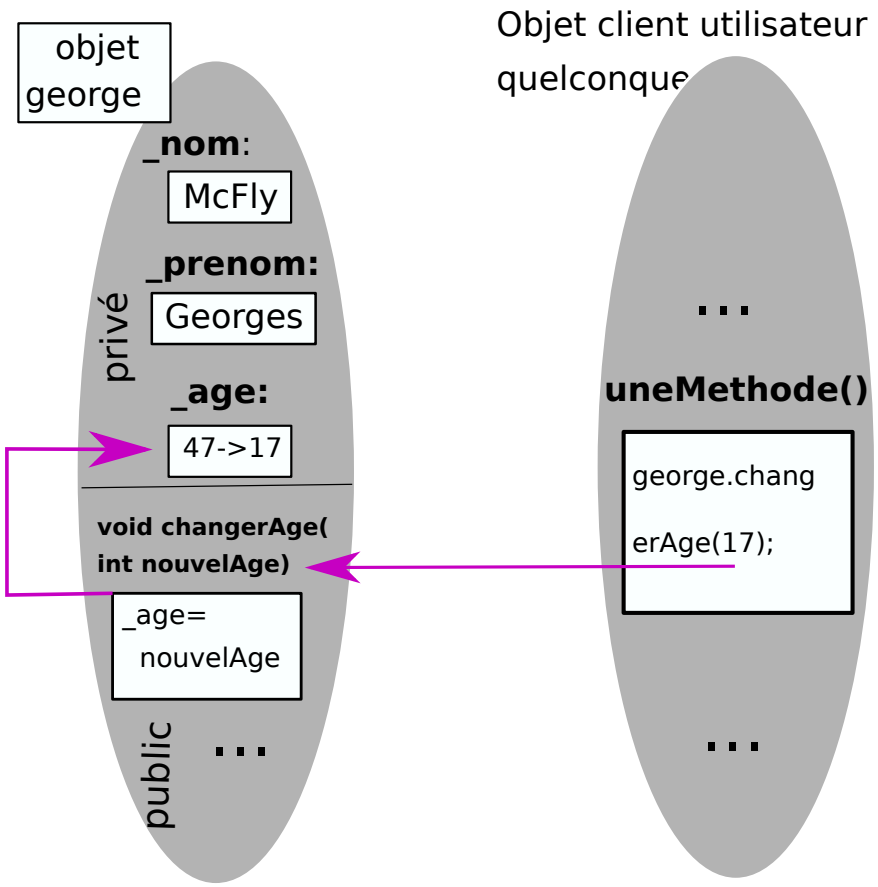
```
int Personne::age()
{
  return _age;
}
```

Point de vue objet en mémoire (pendant l'exécution du programme) :



Mise en œuvre d'un modificateur/mutateur

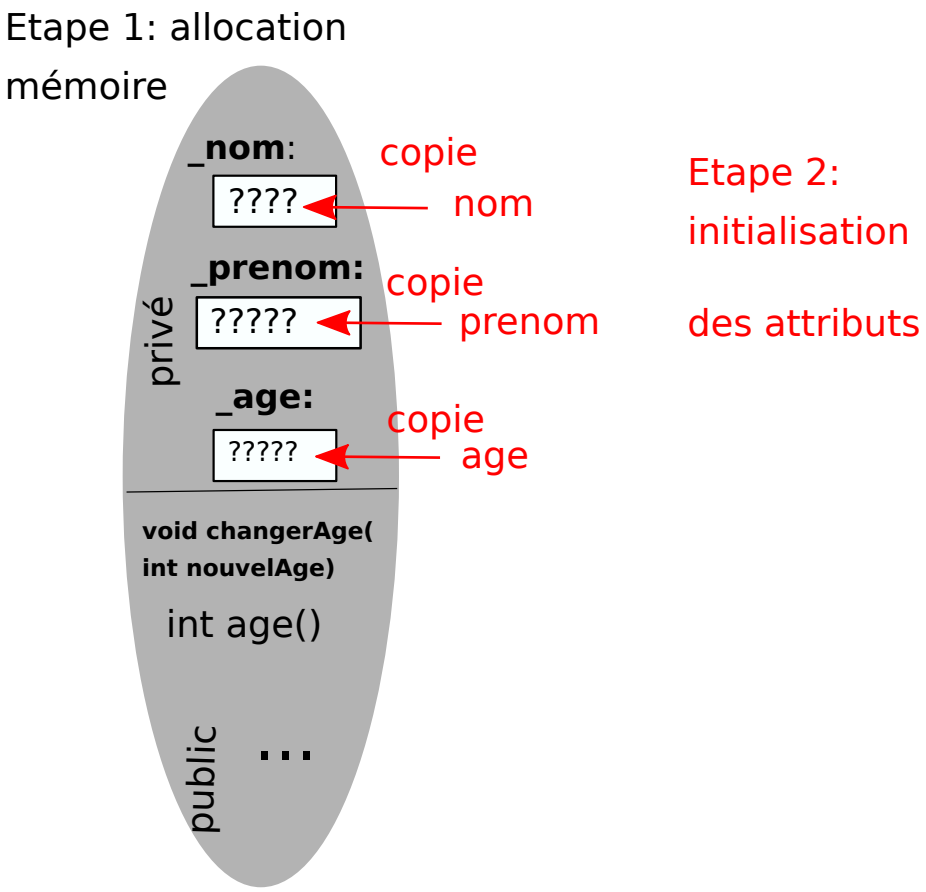
```
// mise en oeuvre des mutateurs
void Personne::changerAge(int nouvelAge)
{
    _age = nouvelAge;
}
```



Constructeurs

Constructeur paramétré : (respecter l'ordre des attributs)

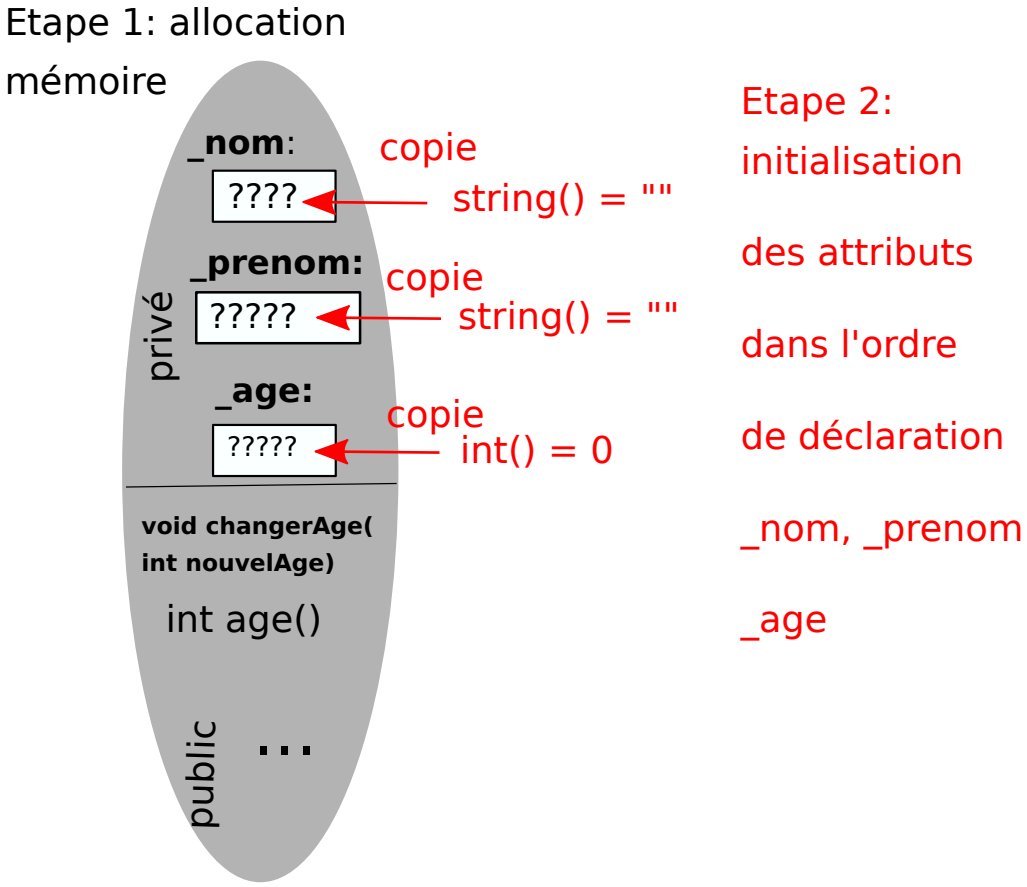
```
Personne::Personne(string nom , string  prenom,
                   int age ):_nom(nom),_prenom(prenom),_age(age)
{}
```



Constructeur par défaut

Constructeur sans paramètre :

```
Personne::Personne():_nom(),_prenom(),_age()
{}
```



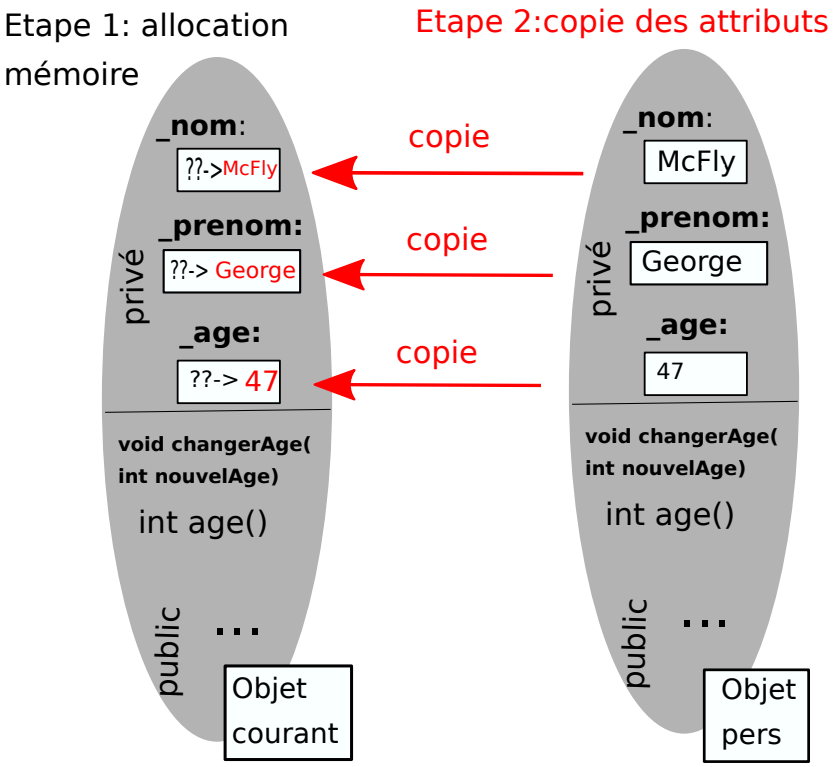
Utilisations du constructeur par défaut

```
int main()
{
    Personne unePersonne;
    cout << unePersonne.nom(); // n'affiche rien


    Personne * uneAutrePersonne   = new Personne();
    cout << uneAutrePersonne->age(); // affiche 0

    uneAutrePersonne->changerNom("McFly");
    uneAutrePersonne->changerPrenom("Marty");
    uneAutrePersonne->changerAge("17");


    Personne * marty = new Personne("McFly","Marty", 17);
    // marty et uneAutrePersonne sont deux personnes différentes
    // mais ont les memes nom, prenom, age
    delete marty; marty = nullptr;
    delete uneAutrePersonne; uneAutrePersonne = nullptr;
}
```

Constructeur par copie

Constructeur spécifique de duplication d'objet : paramètre référence constante (spécifique C++)

```
Personne::Personne(const Personne & pers):
    _nom(pers._nom),
    _prenom(pers._prenom),
    _age(pers._age)
{}
```



Au final : deux objets différents avec des attributs identiques : clones.

Utilisations du constructeur par copie

```
int main()
{
   // constructeur paramétré
   Personne * george1985 = new Personne("McFly","George", 47);

   // constructeur par copie
   Personne * george1955 = new Personne(*george1985);
   george1955->changerAge(george1985->age()-(1985-1955));


   delete george1985; george1985 = nullptr;
   delete george1955; george1955 = nullptr;

   // constructeur paramétré
   Personne marty1955("McFly","Marty", 17);

   // constructeur par copie
   Personne martyDeRetourEn1955(marty1955);

   // marty1955 et martyDeRetourEn1955 sont deux personnes
   // différentes (comme dans le film)
}
```

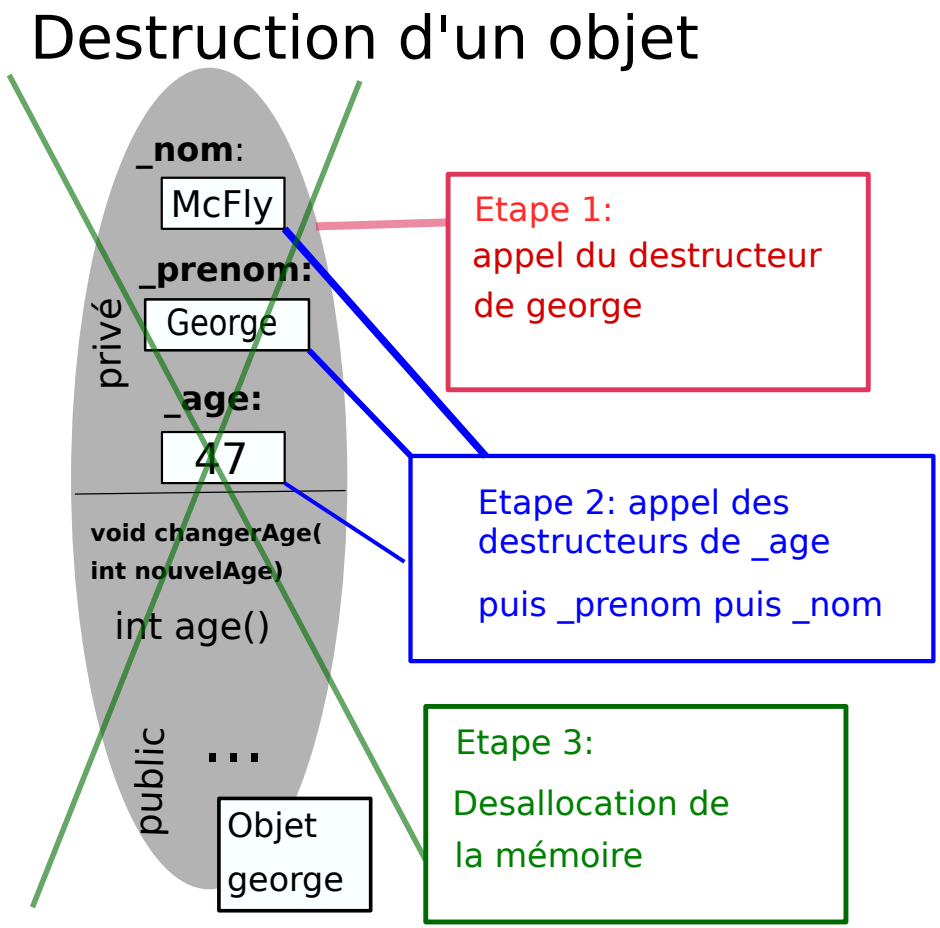
Mise en oeuvre de constructeurs implicites

- On peut définir autant de constructeurs que l'on veut (avec différents paramétrages).
- On peut aussi ne pas en définir.
- Si on ne définit pas de constructeur :
  - Le compilateur va définir implicitement un constructeur par défaut.
- Si on ne définit pas de constructeur par copie :
  - Le compilateur va définir implicitement un constructeur par copie.
- À moins que vous sachiez exactement ce que vous faites, prenez l'habitude de mettre en œuvre un constructeur par défaut et un constructeur par copie dans toutes vos classes.

Destructeur

Le destructeur est unique. Il est automatiquement appelé lorsque l'objet est détruit.

```
Personne::~Personne()
{}
```



Destructeur : règles

- Le destructeur d'une classe est unique. Il prépare la suppression de l'objet.
- Sauf si vous savez ce que vous faîtes, le destructeur est publique.
- Si le programmeur ne le définit pas, le compilateur va le définir automatiquement.
- Le destructeur est appelé automatiquement : ne jamais appeler un destructeur dans un programme, son usage est implicite.
- En règle générale, le destructeur est vide (attributs alloués statiquement), c'est le cas de `~Personne::Personne(){}`
- Si des attributs sont alloués dynamiquement, utiliser `delete` pour les détruire explicitement dans le destructeur.

Exemple : si l'attribut `_nom` était un `string *` dans `Personne`, alors

```
Personne::~Personne()
{
  if(_nom != nullptr) { delete _nom; }
}
```

- Le destructeur d'un pointeur ne détruit pas ce qu'il pointe.

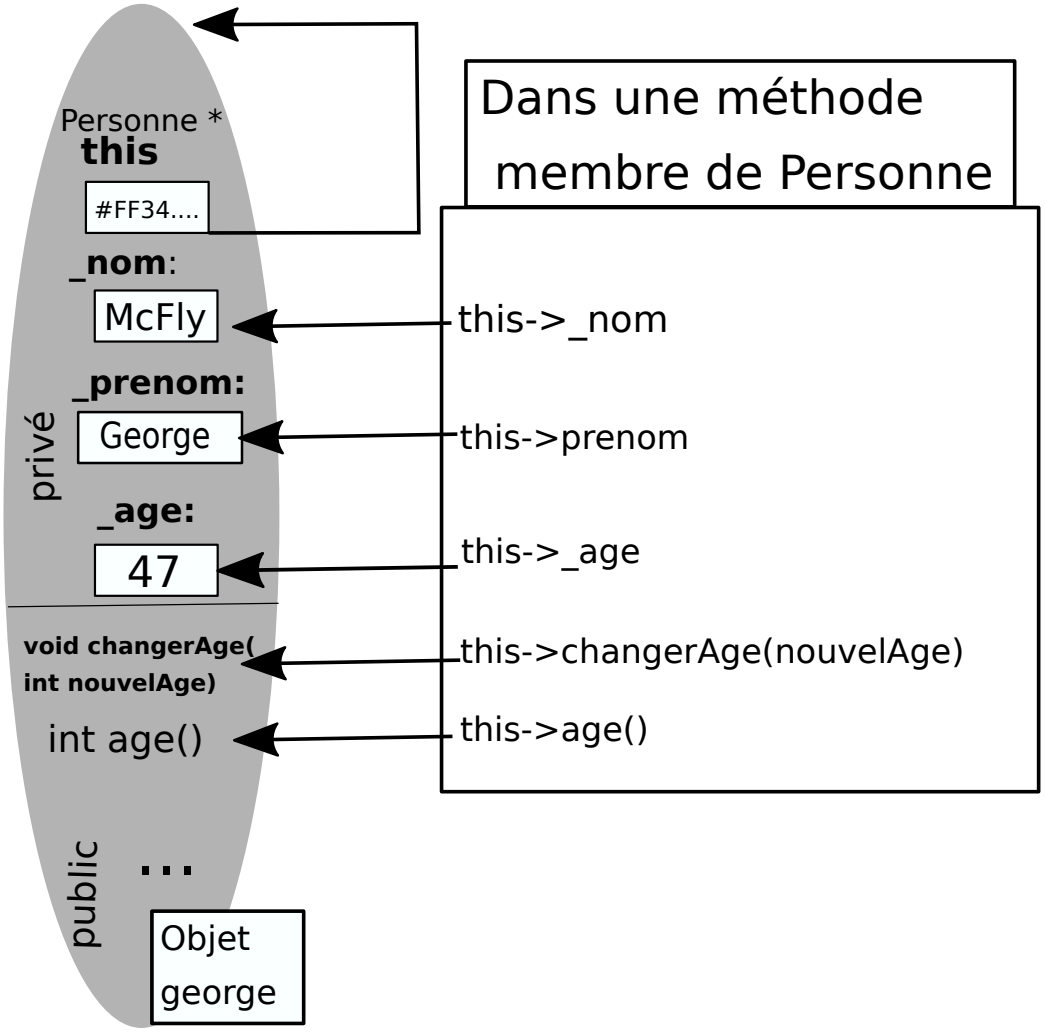
Définir un destructeur que l'on n'utilise pas ? ?

Son utilisation est automatique, décidée par le compilateur.

```
Personne * creerMarty(){
  return new Personne("McFly","Marty", 17);
}

void afficheAgeGeorges()
{
  Personne george("McFly","George", 17);
  cout << george.age();
}  // george est détruit ici (~Personne() est appelé)



int main()
{
  Personne * marty = creerMarty();
  cout << marty->age();
  delete marty; // marty est détruit ici (~Personne() est appelé)
  marty =  creerMarty();
  delete marty; // marty est détruit ici (~Personne() est appelé)
  afficheAgeGeorges();
}
```

Le pointeur `this`

- Chaque objet a la connaissance de sa propre adresse en mémoire.
- L'adresse de l'objet est stocké dans le pointeur `this`.
- Le pointeur `this` n'est accessible que dans une méthode de l'objet.



Utilisation du pointeur `this`

- Ne s'utilise donc que dans des méthodes d'une classe pour représenter l'objet courant.

```
void Personne::changerAge(int nouvelAge)
{
   this->_age = nouvelAge;
   // on affecte nouvelAge à l'attribut _age de l'objet courant
}
```

- Dans ce cas précis, c'est totalement équivalent à :

```
void Personne::changerAge(int nouvelAge)
{
   _age = nouvelAge;
   // on affecte nouvelAge à l'attribut _age de l'objet courant
}
```

- L'usage de `this` est implicite en général. Son usage est explicite pour désambiguiser entre des attributs et des variables locales.

```
void Personne::changerAge(int age)
{
   this->age = age; // this->age: attribut, age: parametre
}
```

Résumé d'étape : compétences à acquérir

- Comprendre le concept de classe, sa raison d'être
- Comprendre les notions suivantes :
  - Classe, Objet, Instance, Attribut, Méthode, Constructeur, Destructeur, Mutateur, Accesseur
- Savoir mettre en oeuvre une classe basique en C++ (constructeurs, méthodes, destructeur)
- Savoir gérer les contrôles d'accès (public/privé) sur les attributs
- Savoir allouer/desallouer dynamiquement un objet avec `new/delete`
- Connaître l'existence du pointeur spécifique `this`.
- Savoir instancier des objets dans un programme et utiliser leurs méthodes

Héritage simple...

Conception de la classe `Employe`

Supposons que l'on désire écrire un programme qui gère des employés :

Quelques attributs d'un employé :

- il a un nom de famille
- il a un prénom
- il a un âge
- il a un numéro d'identification au sein de son entreprise

Conception de la classe `Employe`

```
class Employe
{
 private: // acces prive
   string _nom; // astuce: nommer un attribut avec '_'
   string _prenom;
   int _age;
   int _id;

 public: // acces publique
   Employe();
   Employe(const Employe & employe);
   Employe(string nom, string prenom,
           int age, int id);
   ~Employe();

   void changerAge(int nouvelAge);
   void changerPrenom(string nouveauPrenom);
   void changerNom(string nouveauNom);
   void changerId(int nouvelId);
   int age();
   string prenom();
   string nom();
   int id();
};
```

C'est un peu redondant, non ?

- On retrouve dans la classe `Employe` beaucoup de choses qui étaient dans la classe `Personne`.
- Ne serait-il pas possible de réutiliser la classe `Personne` pour concevoir la classe `Employe` ?
- Conceptuellement,

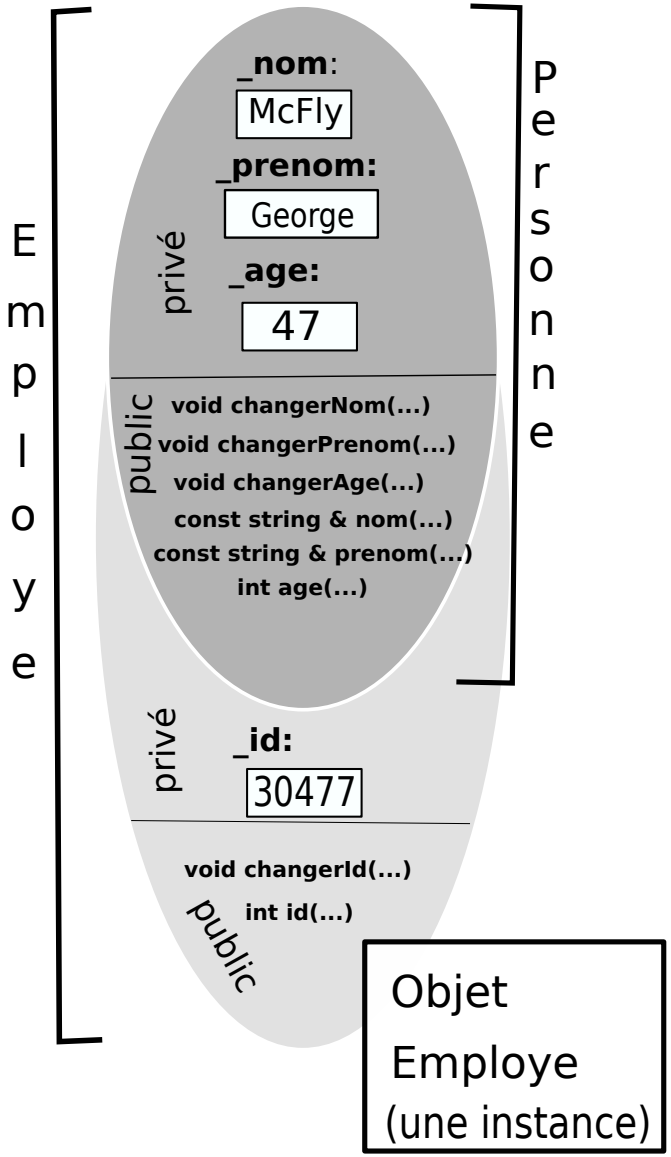
  Un employé **est** une personne.
- En POO, cette relation se répresente par un héritage
- On va concevoir la classe `Employe` en héritant de la classe `Personne`.

Héritage simple

```
class Employe : public Personne
{
   private:
      int _id;
   public:
      Employe();
      Employe(const Employe & employe);
      Employe(string nom, string prenom,
              int age, int id);
      ~Employe();
      void changerId(int nouvelId);
      int id();
};
```

- `Employe` hérite de la classe `Personne`.
- `Employe` dérive de la classe `Personne`.
- `Employe` est une classe dérivée.
- `Personne` est une classe parente.
- `public` ? Héritage publique (le plus courant, nous n'en verrons pas d'autres)

Héritage simple : en mémoire



Héritage simple : utilisation

```
Employe *  george = new Employe("McFly","George",47,30477);

cout << "L'identifiant de " << george->nom()
     << " " << george->prenom() << " est "
     << george->id() << ".\n";

// on change son age et son id
george->changeAge(48);
george->changeId(30478);

delete george;
george = nullptr;
```

- Les méthodes publiques de la classe parente sont des méthodes publiques de la classe fille.

Héritage simple : mise en œuvre constructeurs

```
Employe::Employe(): Personne(), _id(0) {}

Employe::Employe(string nom,
                 string prenom,
                 int age, int id):
                     Personne(nom,prenom,age),_id(id){}

Employe::Employe(const Employe & employe):
                     Personne(employe),_id(employe.id()){}
```

- Avant de construire l'objet `Employe`, on va d'abord demander la construction de sa partie `Personne`
- Une fois la partie `Personne` de l'objet construite, on initialise les attributs propres à `Employe`

Héritage simple : mise en œuvre destructeur

```
Employe::~Employe(){}
```

- Rien ne change
- L'appel des destructeurs est automatisé
- Appel des destructeurs en chaine :
  1. Appel du destructeur de la classe fille (`Employe`)
  2. Appel du destructeur de la classe mère (`Personne`)
  3. Désallocation de l'objet

Héritage simple : utilisation avancée 1

```
Employe *  georgeEmploye = new Employe("McFly","George",47,30477);

// affection de pointeur légale
// georgePersonne et georgeEmploye: meme objet
Personne * georgePersonne = georgeEmploye;

// ok
cout << georgePersonne->nom() << '\n';


// illegal: id() n'est pas une méthode de Personne
cout << georgePersonne->id() << '\n';

delete georgeEmploye;
georgeEmploye = nullptr;

// attention, ici georgePersonne pointe sur rien
// (objet détruit)
georgePersonne = nullptr;
```

Héritage simple : utilisation avancée 2

```
Employe *  georgeEmploye = new Employe("McFly","George",47,30477);

Personne * georgeAdolescent = new Personne(*georgeEmploye); //copie
georgeAdolescent->changeAge(17);
..
delete georgeAdolescent;
delete georgeEmploye;
...
```

- `georgeAdolescent` est une copie de la partie `Personne` de `georgeEmploye`
- La copie est légale car `*georgeEmploye` est une Personne (héritage)

Spécialisation de méthodes

Supposons que nous insérons une nouvelle méthode dans la classe `Personne` :

```
bool estProfessionnel();
```

et nous la mettons en œuvre comme suit :

```
bool Personne::estProfessionnel()
{
  return false;
}
```

- Que vaut alors `georgeEmploye->estProfessionnel()` ?
- Problème conceptuel : en général une Personne n'est pas un professionel mais dans le cadre d'un employé, on voudrait qu'il le soit !
- On voudrait spécialiser la méthode `estProfessionnel()` dans `Employe`.

Méthodes virtuelles

- Pour réaliser une spécialisation, il faut d'abord virtualiser la méthode `estProfessionnel()` dans la classe `Personne`

```
class Personne
{
...
virtual bool estProfessionnel();
...
virtual ~Personne();
};
```

- En ajoutant `virtual` à `estProfessionnel()` on autorise toute classe fille à redéfinir `estProfessionnel()`
- Si la classe fille ne redéfinit pas `estProfessionnel()` alors c'est la version de `Personne` qui sera utilisée.
- L'ajout d'au moins une méthode virtuelle nécessite un destructeur virtuel.
- Ne pas oublier d'ajouter `virtual`, en son absence, le compilateur ne se plaindra pas ! ! ..mais vous n'aurez pas le résultat escompté ! !

Méthodes spécialisées

Sachant que `estProfessionnel()` est virtuelle dans `Personne` on peut spécialiser la méthode dans `Employe`.

```
class Employe: public Personne
{
...
virtual bool estProfessionnel();
...
virtual ~Employe();
};
```

Mise en œuvre :

```
bool Employe::estProfessionnel()
{
  return true;
}
```

Toute classe fille de `Employe` hérite de la version `estProfessionnel` de `Employe` mais peut également la respécialiser.

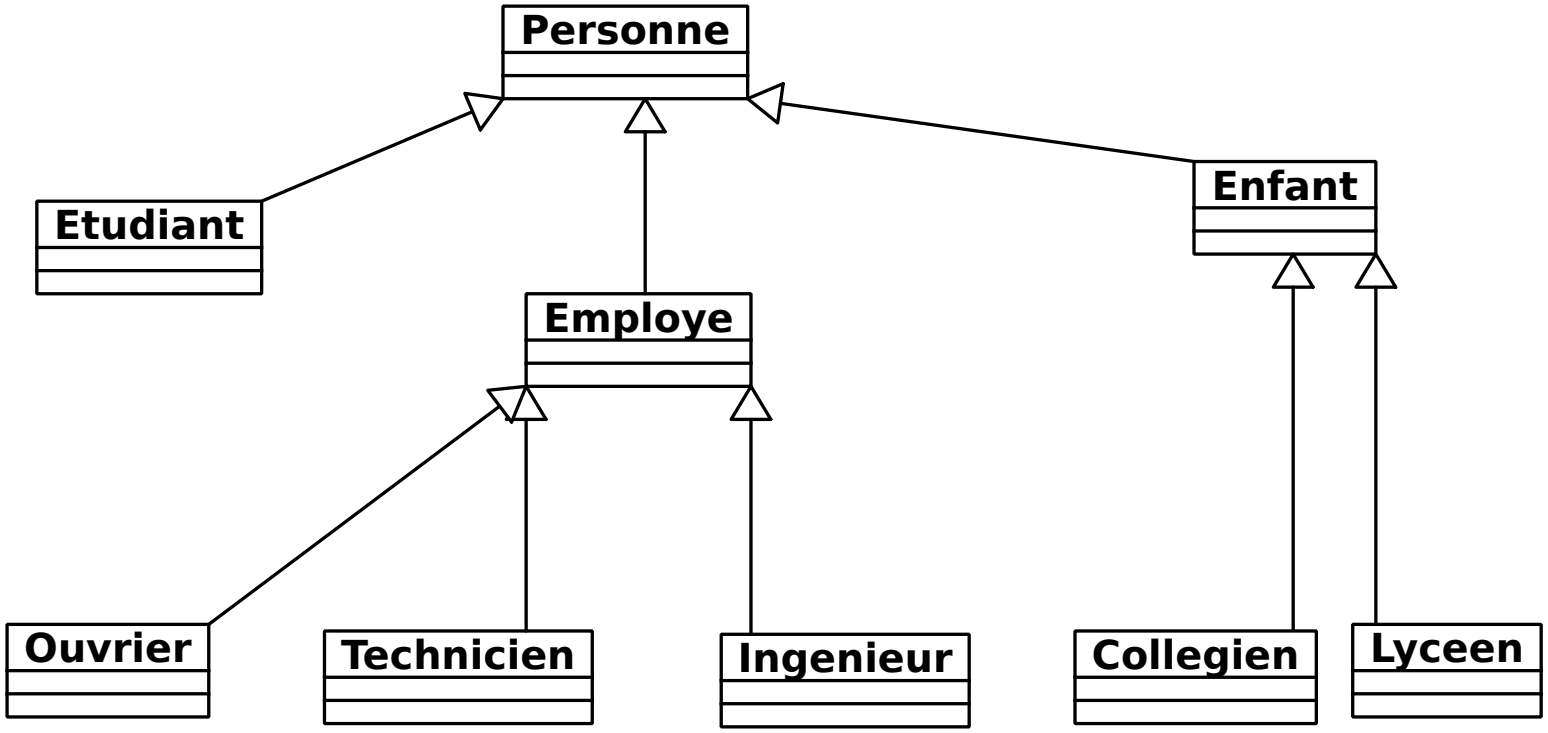
Exemples d'utilisation

```
Employe * george = new Employe("McFly","George",47,30477);
Personne * emmett = new Personne("Brown","Emmett",67);

cout << emmett->estProfessionnel() << '\n'; // non

cout << george->estProfessionnel() << '\n'; // oui

delete george;
delete emmett;
```

Exemples d'utilisation

```
vector<Personne *> personnes;
personnes.push_back(new Personne("Brown","Emmett",67));
personnes.push_back(new Employe("McFly","George",47,30477));

for(size_t i = 0; i < personnes.size(); ++i)
{
   cout << personnes[i]->nom()
        << " "
        << personnes[i]->prenom()
        << " est-il professionel ? "
        << personnes[i]->estProfessionnel()
        << '\n';
}

for(size_t i = 0; i < personnes.size(); ++i)
{
   delete personnes[i];
   personnes[i] = nullptr;
}
```

Hiérarchie de classes



Hiérarchie de classes

```
class Ingenieur : public Employe
{
....
};

class Etudiant : public Personne
{
....
};

class Enfant : public Personne
{
....
};

class Lyceen : public Enfant
{
....
};
```

Contrôle d'accès dans une hiérarchie

On ne considère ici que des héritages publiques (`class A: public B...`).

- Un attribut publique d'une classe mère A est publique dans B.
- Un attribut privé de A n'est pas accessible dans B :
  - `_age`, `_nom`,.... ne peuvent pas être utilisés directement dans une méthode propre à la classe `Employe`.
  - pour accéder à `_age`, `_nom` dans une méthode de `Employe` il faut utiliser les méthodes `age()`, `changerAge()`...
- Pour rendre accessible un attribut privé de A dans B, deux solutions :
  1. Le rendre publique (mais il le sera pour tous (B, C ...), dangereux)
  2. Le protéger (privé pour tous sauf les objets dérivants de A, en particulier les instances de B)

Attribut protégé :

```
class Personne
{
  protected:
     int _age; // age n'est plus privé mais protégé
};
```

Méthodes publiques/protégées/privées

Tout comme les attributs, on peut contrôler l'accès aux méthodes (dans des blocs : `public`,`protected`,`private`)

- méthodes publiques : méthodes pouvant être appelées sur un objet dans tout contexte
  - ex : `age()`, `changerAge()` sur des objets de type `Personne`
  - interface utilisateur d'une classe.
  - fournit les services
- méthodes privées : méthodes ne pouvant être appelées que par des méthodes de la même classe, classes filles exclues
  - Méthodes servant à mettre en œuvre des détails de mise en œuvre
  - Calculs intermédiaires sur les attributs
- méthodes protégées : méthodes ne pouvant être appelées que par des méthodes de la même classe, classes filles incluses.
  - idem méthodes privées à l'exception que des méthodes de classes filles peuvent également les utiliser

Accès publiques/protégées/privées

```
class Employe
{
  private:
    int _salaire; // attribut privé
  protected:
    int _rtt; // attribut protégé
  public:
    int id();  // methode publique
    int salaire(); // methode publique
    int rtt(); // methode publique
  private:
    void  calculSalaire(); // privé: utilisé dans salaire(),
                           // modifie _salaire
  protected:
    void calculRttBase();  // protégé -> utilisé dans rtt()
};                         // modifie _rtt

class Ingenieur : public Employe
{
  private:
    void calculRttInge();  // prive -> utilise calculRttBase()
                           // de Employe et accede/modifie
  ...                      // _rtt de Employe
};
```

Polymorphisme de la POO

- Polymorphisme : qui peut prendre plusieurs formes

- En POO : des fonctions/méthodes de mêmes noms peuvent avoir des comportements différents ou effectuer des opérations sur des données de types différents.

- Deux façons de faire du polymorphisme

  1. Redéfinition par dérivation :
     - `bool estProfessionnel()` n'agit pas de la même manière sur `Personne` (renvoie faux) et sur `Employe` (renvoie vrai).

  2. Surcharge : même nom de méthode mais avec des paramètres différents.
     - On a vu 4 constructeurs de Personne (même nom, paramètres différents)
     - On pourrait définir plusieurs méthodes avec le même nom et des paramètres différents

       ```
       Personne::change(int age)
       Personne::change(string nom, string prenom)
       ```

Compétences à acquérir

- Comprendre la notion d'héritage
- Savoir écrire une classe qui dérive d'une autre classe (spécification et mise en œuvre).
- Savoir spécialiser des méthodes (virtualisation)
- Comprendre la notion de polymorphisme
- Savoir concevoir et mettre en œuvre une hiérarchie de classes (héritage simple)
- Maîtriser le contrôle d'accès aux attributs/méthodes dans cette hiérarchie.
- Savoir surcharger des méthodes.

Conception orientée objet en UML

UML : Unified Modeling Language

- Langage graphique
- Visualisation, Specification, Construction, Docummentation de systèmes logiciels complexes
- Langage de description standardisé
- Permet de décrire différents niveaux d'un système
  - **Aspects business** (comment le système interagit avec le client)
  - **Aspects fonctionnels** (cahier des charges)
  - **Aspects comportementaux** (comment les composants du logiciel interagisent entre eux)
  - **Aspects structurels** (comment les composants logiciels sont conçus)

UML : sa philosophie

- Modélisation de systèmes : du concept à la mise en œuvre
- Approche multi-vues : une vue = un type de diagramme
- S'appuie sur des techniques de modélisation orientée objet
- Moyen d'aborder la conception d'un système très complexe
- Langage formel, lisible par l'humain, exploitable par la machine.
- Outils UML : certains permettent de générer automatiquement du code dans des langages cibles (des classes C++, Java,...)
- Outils de retroingénierie : analyse de code pour la synthèse automatique de diagrammes UML

UML : les types de diagrammes structurels

Conception, Modélisation des aspects statiques du système :

- Diagramme de classes
  Comment les classes sont-elles structurées entre elles ?

- Diagramme d'objets
  Scénarios d'interactions entre objets

- Diagramme de composants
  composant = ensemble de classes

- Diagramme de déploiement
  Comment les composants sont déployés, architecture

UML : les types de diagrammes comportementaux

Conception, Modélisation des aspects dynamiques du système :

- Diagramme de séquences/collaborations
  Scénarios de communications entre des objets

- Diagramme de cas utilisateurs
  Description de scénarios où l'on décrit ce que fait le système face à un acteur (client)

- Diagramme de Machine à états (StateChart)
  Description de comportements dynamiques

- Diagramme d'activités
  Description de comportements dynamiques entre activités

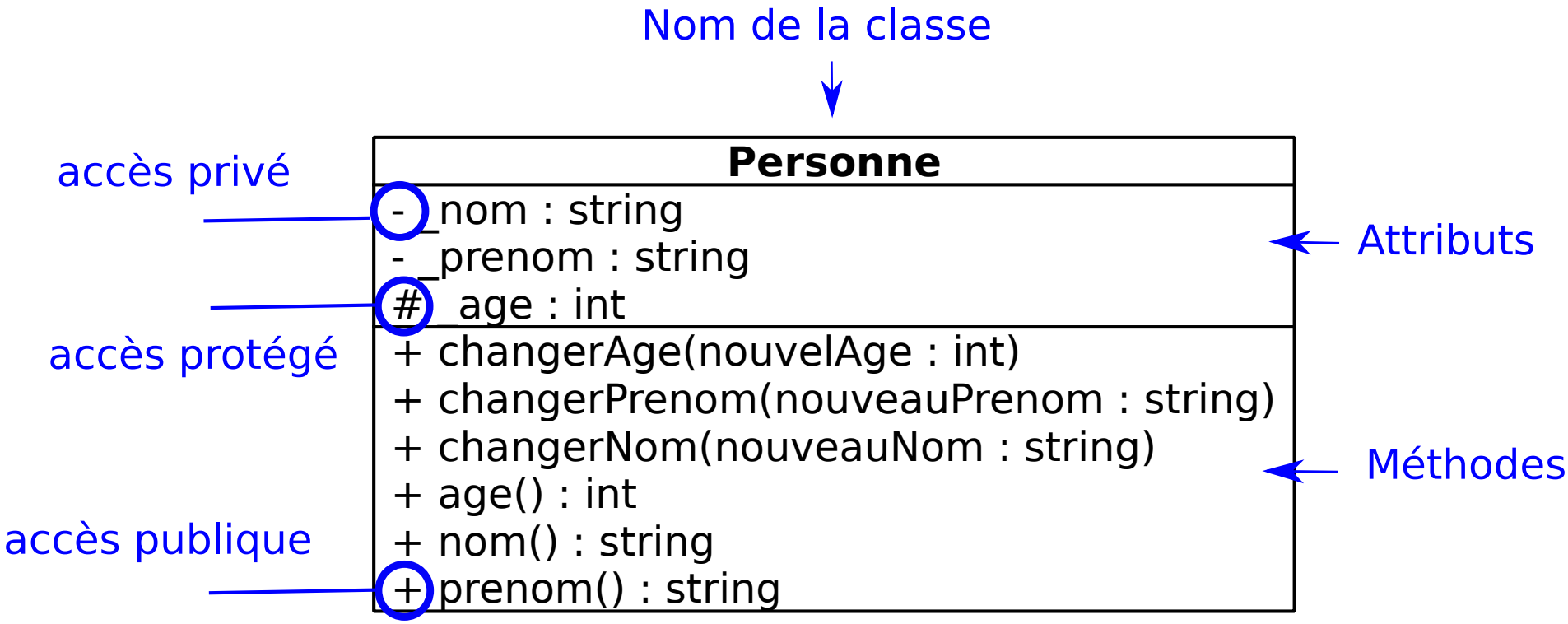
Objectifs dans ce cours

- Savoir lire un diagramme de classe UML
- Savoir lire un diagramme de séquence
- Savoir mettre en œuvre en C++ une spécification UML à l'aide d'un diagramme de classe.

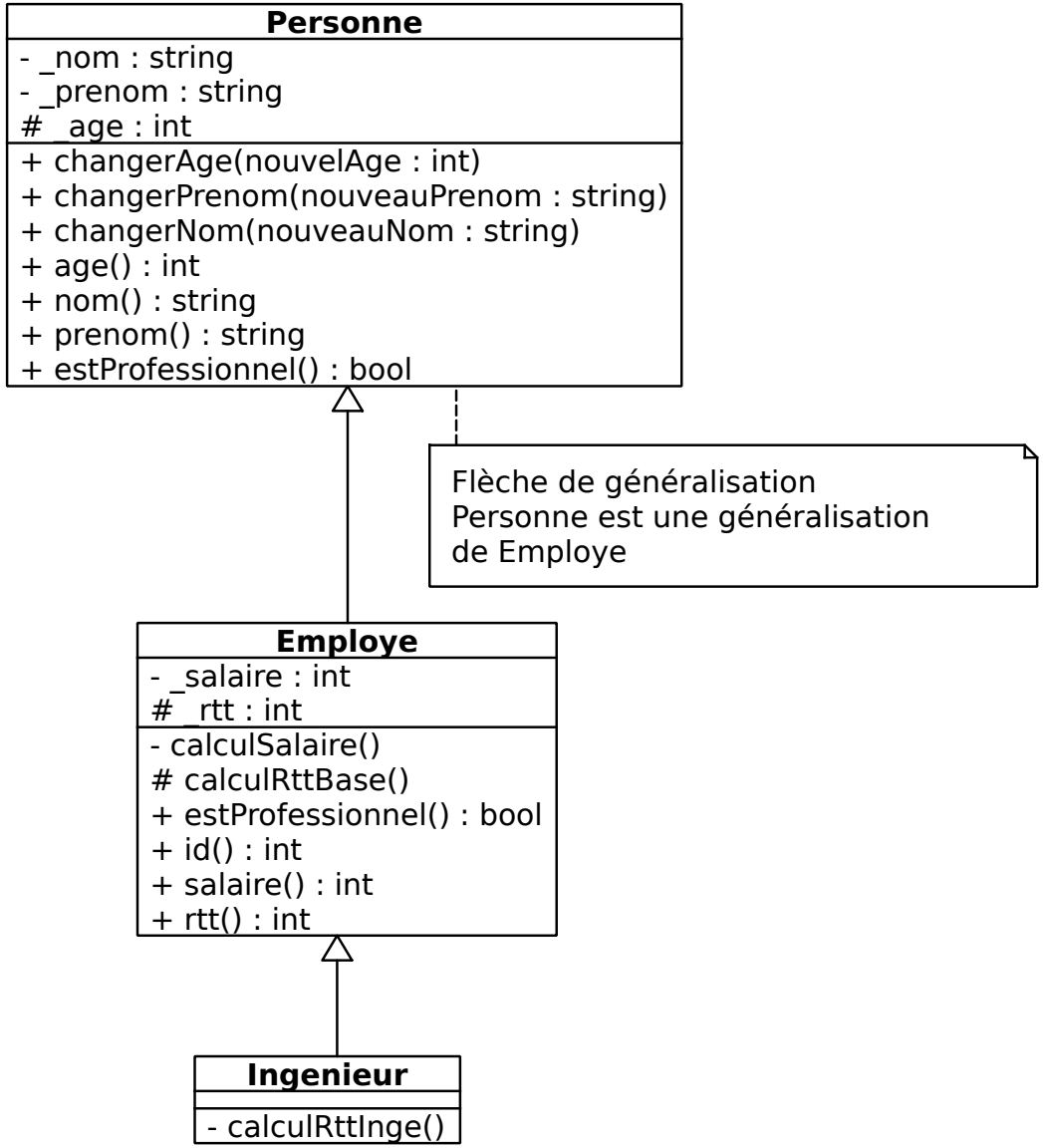
Diagramme de classes

- Le plus commun des diagrammes UML
- Décrit un ensemble de classes, leurs interfaces
- Décrit la collaboration entre ces classes
- Décrit leurs relations
  - Généralisation/Héritage
  - Association
  - Agrégation
  - Composition

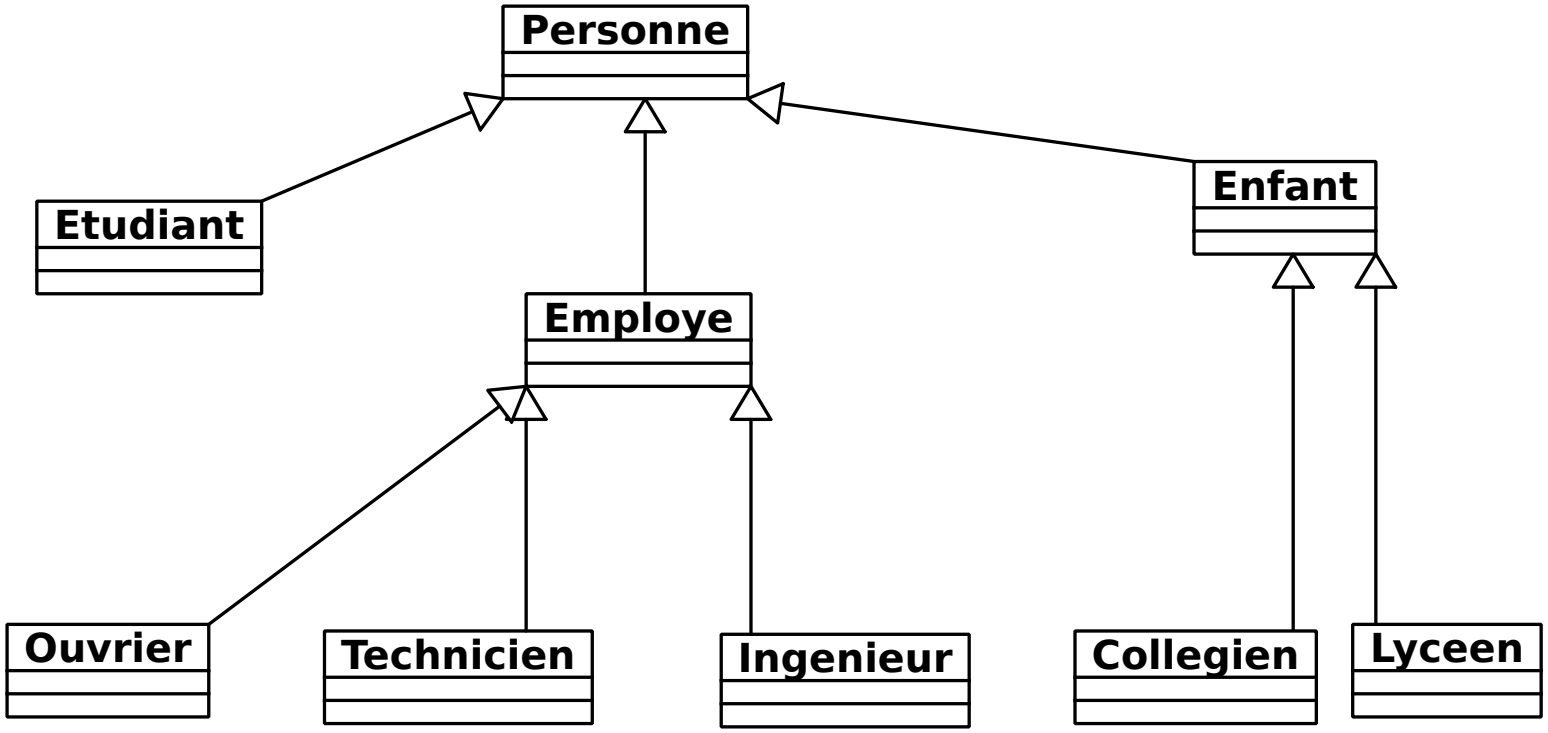
La classe Personne en UML



Héritage simple en UML



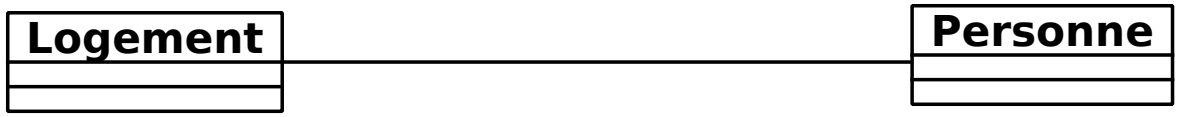
Hiérarchie de classes en UML



Association

- Deux objets de classe différente peuvent être amenés à collaborer
- Ils peuvent s'envoyer des messages
- Un objet de classe A peut appeler une méthode publique sur un objet de classe B
- Ce lien est appelé : association
- Il se décrit statiquement entre la classe A et la classe B.

Exemple :

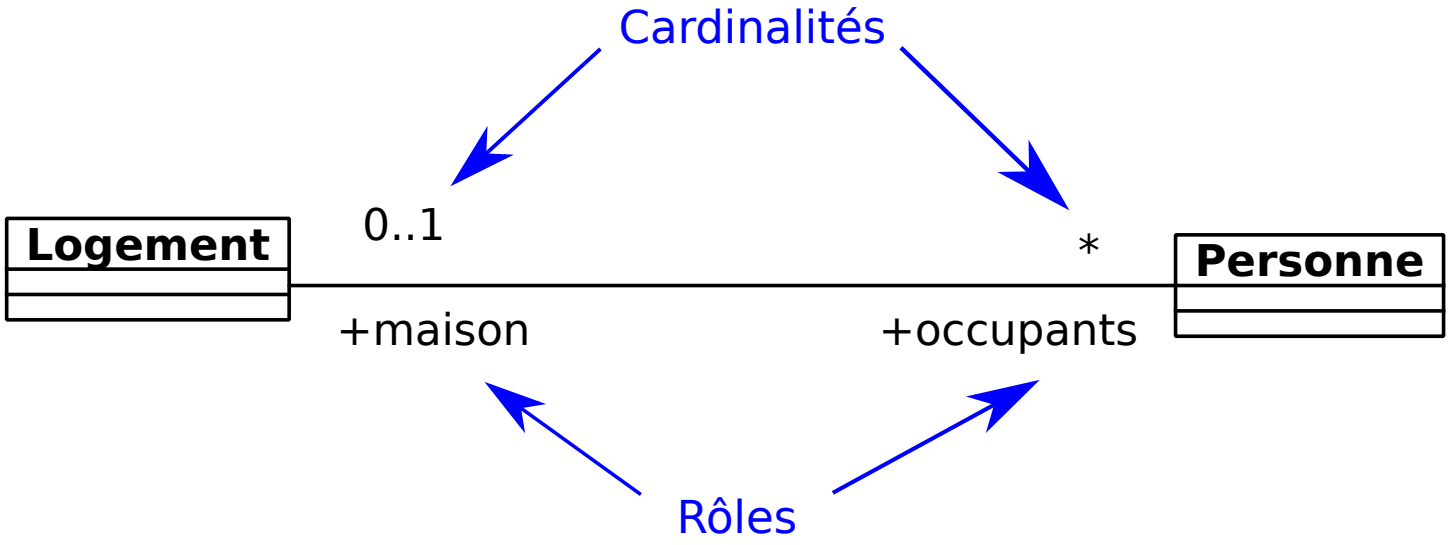


Association simple : un logement permet de loger des personnes. Une personne habite dans un logement.

Cardinalités et Rôles dans une association

- Cardinalités : nombre d'objets impliqués dans l'association
- Rôles : Description de l'association.

Exemple :



Le diagramme se lit de la façon suivante :

- Une Personne est dans 0 ou 1 Logement qui lui sert de maison.
- Un Logement est habité par un ensemble potentiellement vide de Personne qui sont des occupants.

Format des cardinalités dans une association

- `N` $\rightarrow$ `N` objets exactement (`N` entier positif)
- `N1..N2` $\rightarrow$ entre `N1` et `N2` objets
- $*$ $\rightarrow$ un ensemble d'objets quelconque voire vide
- `N..*` $\rightarrow$ au moins `N` objets
- Si pas de cardinalité indiquée alors cela signifie $*$

Mise en œuvre possible en C++ : attributs

Pour mettre en œuvre une association, il faut ajouter des attributs aux classes.

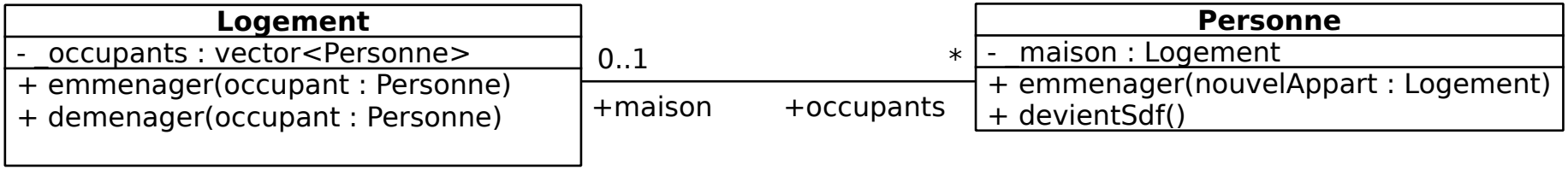
- `0..1` → 1 attribut pointeur sur un objet de la classe en association
- `N` → `N` attribut pointeurs ou un tableau de `N` pointeurs (ex : `vector<T *>`)
- `N1..N2` → un tableau contenant de `N1` pointeurs à `N2` pointeurs (ex : `vector<T *>`)
- `*` → un tableau de pointeurs potentiellement vide
- `N..*` → un tableau de pointeurs contenant au moins `N` pointeurs.

Mise en œuvre possible en C++ : méthodes

Pour mettre en œuvre une association, il faut également ajouter des méthodes pour initialiser les attributs représentant les associations. Plusieurs possibilités :

- Utilisation de constructeurs paramétrés
- Utilisation de modificateurs/mutateurs

Exemple de mise en œuvre d'une association




```
class Logement; // prédéclaration de Logement
// suffit pour la déclaration de Personne ci-dessous

class Personne {
private:
Logement * _maison; // nullptr si pas de logement
public:
void emmenager(Logement * nouvelAppart);
void devientSdf();
};
// pas besoin de prédéclaration de Personne (déclaration au dessus)
class Logement {
private:
std::vector<Personne *> _occupants;
public:
void emmenager(Personne * occupant);
void demenager(Personne * occupant);
};
```

Exemple d'instanciations d'une association

(allocation statique)

```
int main()
{
   Logement appart;
   Personne sheldon("Cooper","Sheldon",27); // en 2007
   Personne leonard("Hofstadter","Leonard",27);
   appart.emmenager(&leonard);
   leonard.emmenager(&appart);
   appart.emmenager(&sheldon);
   sheldon.emmenager(&appart);
   ...
}
```

Exemple d'instanciations d'une association

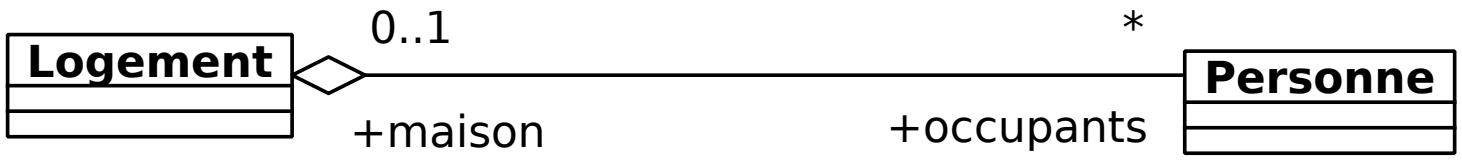
(allocation dynamique)

```
int main()
{
   Logement appart = new Logement();
   Personne *  sheldon = new Personne("Cooper","Sheldon",27);
   Personne *  leonard = new Personne("Hofstadter","Leonard",27);
   appart->emmenager(leonard);
   leonard->emmenager(appart);
   appart->emmenager(sheldon);
   sheldon->emmenager(appart);
   ...
   delete appart; delete leonard; delete sheldon;
}
```

Agrégation

- L'agrégation permet de modéliser une relation ensemble,partie
- Un objet de classe A forme un tout (ensemble) qui rassemble des objets de classes B (partie)
- Contrairement à l'association, les objets de classes A ne sont pas au même niveau que ceux de la classe B.

Exemple :



Le diagramme se lit de la façon suivante :

- Une Personne est dans 0 ou 1 Logement qui lui sert de maison.
- Un Logement **contient** un ensemble potentiellement vide d'occupants qui sont des Personne. (Logement est un contenant, Personne un contenu).
- La différence entre association et aggrégation est conceptuelle.

Exemple de mise en œuvre d'une agrégation

Identique à une association.


```
class Logement; // prédéclaration de Logement
class Personne {
private:
Logement * _maison; // nullptr si pas de logement
public:
void emmenager(Logement * nouvelAppart);
void devientSdf();
};

class Logement {
private:
std::vector<Personne *> _occupants;
public:
void ajouterLocataire(Personne * occupant);
void enleverLocataire(Personne * occupant);
vector<Personne *>  locataires();
}; // renforce l'idée de contenant
```

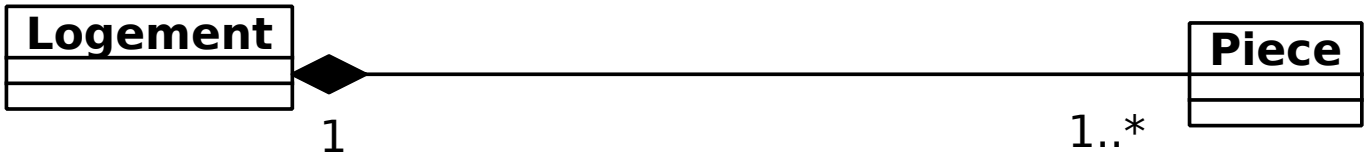
Exemple d'instanciations d'une agrégation

```
int main()
{
   Logement appart = new Logement();
   Personne *  sheldon = new Personne("Cooper","Sheldon",27);
   Personne *  leonard = new Personne("Hofstadter","Leonard",27);
   appart->ajouterLocataire(leonard);
   leonard->emmenager(appart);
   appart->ajouterLocataire(sheldon);
   sheldon->emmenager(appart);
   ...
   for(size_t i = 0; i < appart->locataires().size(); ++i)
   {
      cout << appart->locataires()->at(i)->prenom() << '\n';
   }

   ...
   delete appart; delete leonard; delete sheldon;

}
```

Composition

- La composition est une agrégation plus forte
- Un objet de classe A forme un tout (ensemble) qui rassemble des objets de classes B (partie)
- Contrairement à l'agrégation, les objets de classes B contenus dans A ont la même durée de vie que A. Les objets de classes B sont des briques de construction d'objet de classe A.

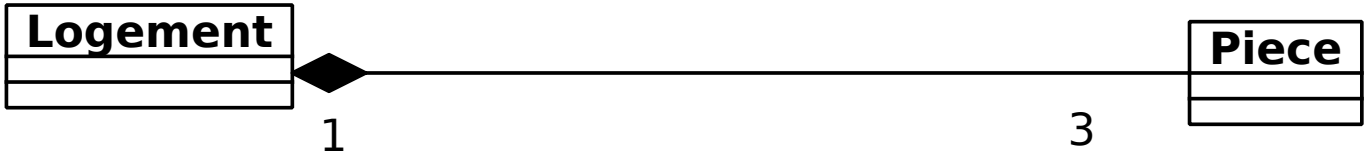
Exemple :



Le diagramme se lit de la façon suivante :

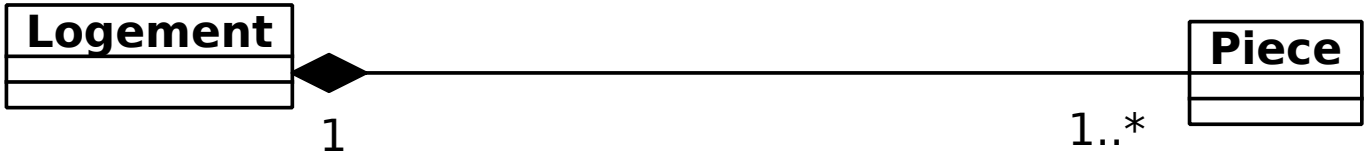
- Un Logement **contient** un ensemble de pieces. Si le logement est détruit les pièces le sont aussi
- La différence entre composition et aggrégation est liée à la durée de vie. (aggrégation : si le logement est détruit, les personnes s'en vont).

Exemple de mise en œuvre d'une composition (facon 1)



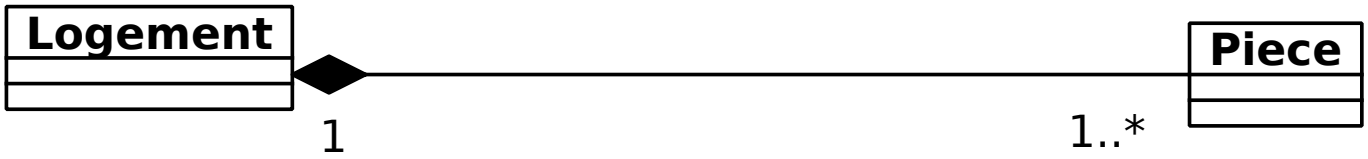

```
class Piece {
private:
Logement * _maison; // pointeur constant
...
};

class Logement {
private:
Piece _chambre; // on stocke des objets si pas trop nombreux
Piece _salon; // et leur nombre est connu à l'avance
Piece _toilettes; // avantage: pas de mémoire à gerer (alloc statique)
... // quand le logement est détruit les pieces aussi.
};
```

Exemple de mise en œuvre d'une composition (facon 2)




```
class Piece {
private:
Logement * _maison; // pointeur constant
public:
Piece(Logement * logement,TypePiece type); // Logement adossé à pièce
// dès la creation
...
};

class Logement {
private:
std::vector<Piece *> _pieces;
public:
Logement();
void creerPiece(TypePiece type);
~Logement();
};
```

Exemple de mise en œuvre d'une composition (facon 2)




```
Logement::Logement():_pieces(){}

void Logement::creerPiece(TypePiece type)
{
  _pieces.push_back(new Piece(this,type)); // this: Logement courant
}
```

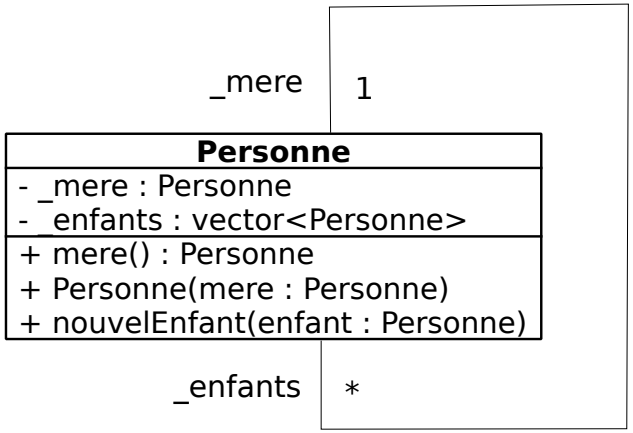
Attention au destructeur de `Logement`, il doit détruire les pièces allouées :

```
Logement::~Logement()
{
   for(size_t i = 0; i < _pieces.size(); ++i)
   {
        if(_pieces[i] != nullptr)
        {
           delete _pieces[i];
        }
   }
}
```

Exemple d'instanciations d'une composition

```
typedef
enum class { SALON, SDB, CHAMBRE, CUISINE, GARAGE } TypePiece;

int main()
{
   Logement * appart = new Logement();
   appart->creerPiece(SALON);
   appart->creerPiece(SDB);
   appart->creerPiece(CHAMBRE);
   appart->creerPiece(CHAMBRE);
   appart->creerPiece(CUISINE);

   delete appart; // les pieces sont détruites ici

}
```

Association en boucle



- Une personne a une mère qui est une personne.
- Une personne peut avoir des enfants qui sont des personnes.

```
class Personne
{
  private:
    Personne * _mere;
    vector<Personne *> _enfants;
  public:
    Personne(...Personne * mere);
    void nouvelEnfant(Personne * enfant);
    Personne * mere();
    ...
};
```

Exemple d'instanciations d'une association en boucle

```
...
// on suppose ici que stella (Baines) existe
Personne * lorraine = new Personne("Lorraine","Baines",47,stella);
stella->nouvelEnfant(lorraine);

Personne * dave = new Personne("Dave","McFly",22,lorraine);
lorraine->nouvelEnfant(dave);

Personne * linda = new Personne("Linda","McFly",20,lorraine);
lorraine->nouvelEnfant(linda);

Personne * marty = new Personne("Marty","McFly",17,lorraine);
lorraine->nouvelEnfant(marty);
...
```

- La classe Personne a un double rôle : mère et enfant.
- Mais une instance de Personne peut n'avoir qu'un rôle
  - `marty` est un enfant qui a une mère mais n'est la mère de personne.
  - `lorraine` est la mère de `marty` et l'enfant de Stella Baines

Diagramme de séquences

- Décrit le déroulement temporel d'un scénario
- Diagramme d'interactions entre instances
- Une instance : boite + une ligne de vie verticale

Diagramme de séquences

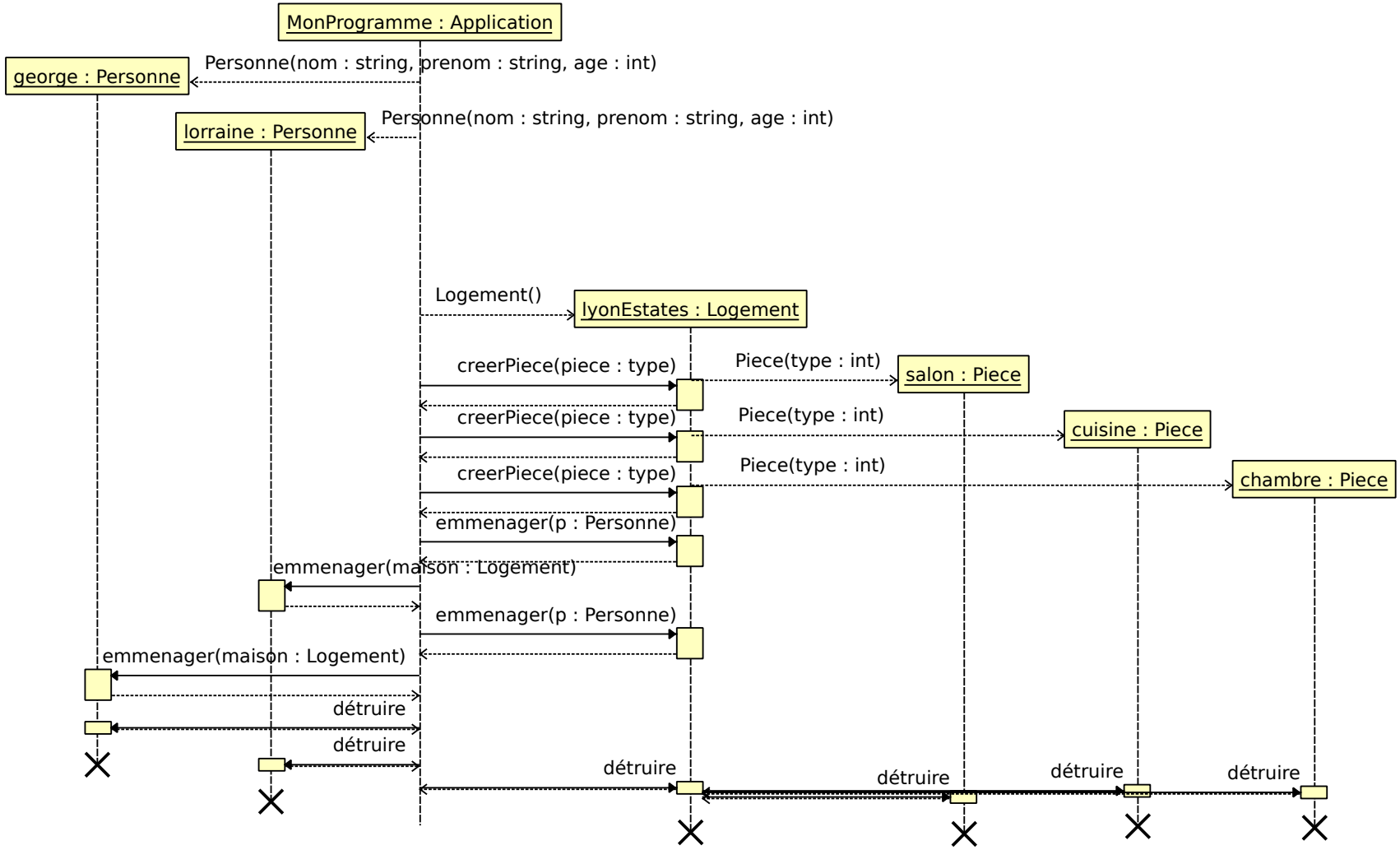
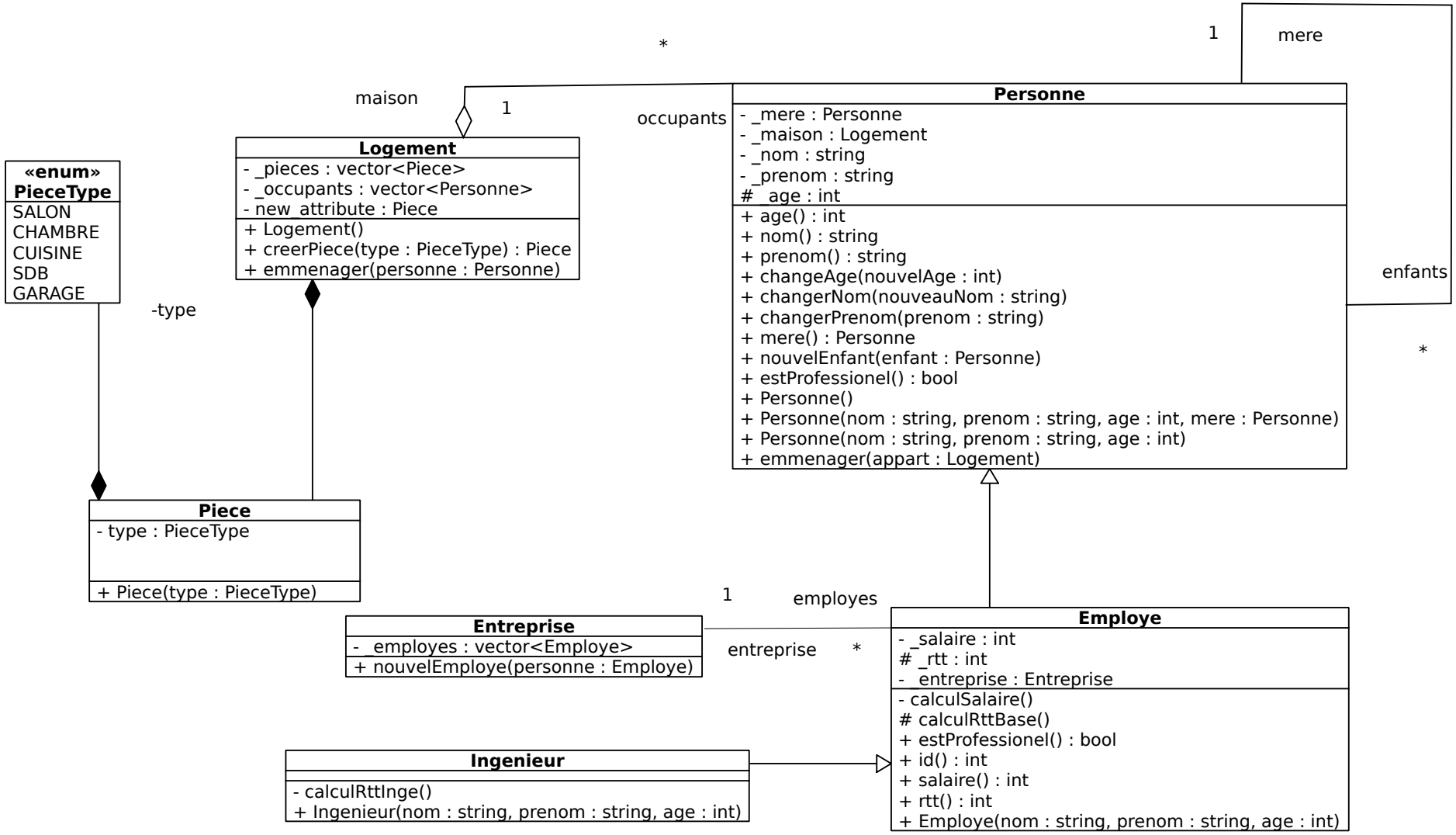


Diagramme de séquences : scénario en C++

```
int main()
{
  Personne * george = new Personne("McFly","George",47);
  Personne * lorraine = new Personne("Baines","Lorraine",47);
  Logement * lyonEstates= new Logement();
  lyonEstates->creerPiece(SALON);
  lyonEstates->creerPiece(CUISINE);
  lyonEstates->creerPiece(CHAMBRE);
  lyonEstates->emmenager(lorraine);
  lorraine->emmenager(lyonEstates);
  lyonEstates->emmenager(george);
  george->emmenager(lyonEstates);
  delete george;
  delete lorraine;
  delete lyonEstates;
};
```

Petit exemple UML complet



Compétences à acquérir

1. Savoir lire un diagramme de classe UML
2. Comprendre les relations :
   - Héritage simple, Association, Agrégation, Composition
3. Savoir proposer une mise en oeuvre en C++ d'un diagramme de classe.
4. Savoir instancier en C++ un diagramme de séquence.

Pour aller plus loin en C++

- Bibliothèque standard STL : `std::vector`, `std::set`, `std::list`, `std::map`
- Notion d'itérateurs `begin()`,`end()`
- Notion de programmation générique `template`
- Fonctions lambdas

Pour aller plus loin en POO

- Héritage multiple (C++ le fait, mais d'autres langages non)
- Classe abstraite (non instanciable)
- Méthodes virtuelles pures
- Notions d'interface (Très utilisée en Java)
- Notions d'invariants/préconditions/postconditions (programmation par contrat)
- Motifs de programmation orientés objets (Design Pattern)

Pour aller plus loin en UML

- Étude des autres diagrammes, plus abstrait
- Diagramme d'objets
- Diagramme de composants
- Diagramme de déploiement
- Diagramme de cas utilisateurs
- Diagramme de collaborations
- Diagramme de Machine à états (StateChart)
- Diagramme d'activités