

# La programmation orientée objet et le langage C++

Pablo Rauzy

rauzy@enst.fr

`pablo.rauzy.name/teaching.html#epu-cpp`

Ce cours est en partie inspiré du polycopié d'Henri Garreta.

EISE4 @ Polytech'UPMC

15 septembre 2014 — Cours 1

Déroulement du cours

Préambule

C++ : approche superficielle

Premières différences avec le C

Programmation orientée objet

Les classes

Mini-TD

- ▶ Cours magistraux :  $6 \times 2h$ .
- ▶ Travaux pratiques :
  - ▶  $5 \times 2h$  (seul ou à deux),
  - ▶  $1 \times 4h$  (seul),
  - ▶  $1 \times 4h$  (démarrage projet).
- ▶ Projet libre (avec contraintes techniques).
  
- ▶ La notation prendra en compte :
  - ▶ le projet,
  - ▶ le TP seul,
  - ▶ l'examen final (QCM + questions),
  - ▶ les TP ;

pour chacun de ces éléments, la justesse mais aussi la qualité du code et de son organisation seront pris en compte.

- ▶ *Apparition* : le langage C++ a été créé en 1983 par Bjarne Stroustrup pendant qu'il travaillait dans le laboratoire de recherche Bell d'AT&T.
- ▶ *But* : améliorer le langage C, d'où le nom (*C with classes*, puis C++), tout en conservant au maximum sa rapidité.
- ▶ *Méthode* : extension du langage (ce n'est plus totalement vrai) pour conserver une compatibilité avec C.
- ▶ *Paradigme* : programmation orientée objet.
- ▶ Cependant, C++ reste un langage de bas niveau où on gère la mémoire à la main.

- ▶ Organisation modulaire des fichiers :
  - ▶ en-têtes en `.hh` ou `.hpp` ;
  - ▶ implémentations en `.cc` ou `.cpp`.
- ▶ Syntaxe : comme en C, Java, etc.
- ▶ Outils nécessaires :
  - ▶ éditeur de texte (e.g., Emacs, Vim) ;
  - ▶ compilateur (e.g., g++, clang) ;
  - ▶ moteur de production (e.g., make) ;
  - ▶ débogueur (e.g., gdb) ;
  - ▶ profileur (e.g., valgrind).

- ▶ Déclaration des variables : placement libre.
- ▶ Vrais Booléens.
- ▶ Références.
- ▶ Fonctions en ligne.
- ▶ Valeurs par défaut des arguments de fonctions.
- ▶ Surcharge des noms de fonctions.
- ▶ Appel de fonctions écrites en C.
- ▶ Entrées / sorties plus simples.
- ▶ Allocations dynamiques de mémoire.

- ▶ Les déclarations ne doivent pas forcément se trouver en début de bloc.
- ▶ En dehors de cas particuliers, il est tout de même préférable de garder cette habitude.

## C

```
void save_to_file (Document *doc)
{
    FILE *file = NULL;
    char *path = NULL;
    file_picker(path);
    file = fopen(path, "w");
    write_to_file(file, doc);
    fclose(file);
    free(path);
}

{
    int i;
    //...
    for (i = 0; i < 42; i++) {
        //...
    }
}
```

## C++

```
void save_to_file (Document *doc)
{
    char *path = NULL;
    file_picker(path);
    FILE *file = fopen(path, "w");
    write_to_file(file, doc);
    fclose(file);
    free(path);
}

{
    //...
    for (int i = 0; i < 42; i++) {
        //...
    }
}
```

- ▶ En C++ il existe un type *Booléen* `bool` dont les valeurs sont `false` et `true`.
- ▶ Le résultat des opérateurs logique `&&`, `||`, etc. est de type `bool`.
- ▶ Les conditions sont attendues de type `bool`.
- ▶ Du coup, il vaut mieux éviter d'utiliser des entiers comme Booléens (et vice-versa).

## C

```
int x;  
//...  
if (x) {  
    //...  
}
```

## C++

```
int x;  
//...  
if (x != 0) {  
    //...  
}
```



- ▶ En C++, il existe une notion de *référence* en plus des pointeurs.
- ▶ C'est aussi une manière de manipuler les adresses des objets placés dans la mémoire.
- ▶ Pour un type  $T$ , le type "référence sur  $T$ " se note  $T \ \&$ .

```
int n;  
int &r = n; // r est une référence sur n
```

- ▶ Une référence doit être initialisée à sa création, après toute opération réalisée affecte l'objet référencé.

```
int j;  
r = j; // r reste une référence sur n, ceci affecte la valeur de j à n
```

- ▶ Les références permettent de passer aux fonctions des paramètres modifiables sans utiliser les pointeurs.

```
void swap (int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int answer = 42;
int year = 2014;

swap(year, answer);
// maintenant, answer vaut 2014 et year vaut 42
```

- ▶ Attention, ce genre de facilité offert par C++ est à double tranchant !

```
void save_preferences (const Preferences &user_pref)
{
    // ici user_pref n'est pas une copie mais une référence (plus rapide !),
    // mais il ne peut pas être modifié par la fonction (plus sûr).
}
```

- ▶ Une fonction peut renvoyer une référence.
- ▶ Cela permet à l'appel de la fonction d'être utilisé comme membre gauche d'une affectation.

```
char *names[N];
int ages[N];

int &age(char *name) {
    for (int i = 0; i < N; i++) {
        if (strcmp(name, names[i]) == 0) {
            return ages[i];
        }
    }
}

age("Cunégonde") = 77;
age("Régimbald")++;
```

- ▶ Du coup, quand utiliser les références plutôt que les pointeurs ?
- ▶ Il n'y a pas d'arithmétique des références.
- ▶ Une référence ne peut pas être vide (NULL).
- ▶ Mais il n'y a pas vraiment de critères simples et universels.
- ▶ Toutefois, le code d'un même projet doit être cohérent.
- ▶ Attention au & :
  - ▶ Dans une déclaration de type, il signifie "référence sur".
  - ▶ En tant qu'opérateur unaire, il signifie "adresse de" (comme en C).
  - ▶ En tant qu'opérateur binaire, il signifie "et logique bit à bit" (idem).

- ▶ Dans un but d'efficacité, on peut éviter le coût de l'appel d'une fonction en la rendant *en ligne* avec le mot clef `inline`.
- ▶ Dans ce cas, le corps de la fonction remplacera son appel.
- ▶ Moins dangereux que les macros préprocesseur de C.

## C

```
#define max(a, b) \  
    ((x >= y) ? x : y)  
  
int a = 42;  
int p = 50;  
int m = max(a, ++p); // m vaut 52
```

## C++

```
inline int max (int x, int y) {  
    return x >= y ? x : y;  
}  
  
int a = 42;  
int p = 50;  
int m = max(a, ++p); // m vaut 51
```

- ▶ En C++, on peut donner une valeur par défaut aux arguments d'une fonction.

```
void connect (char *host = "localhost", int port = 80)
{
    //...
}

connect() // connexion au serveur HTTP local
connect("www.polytech.upmc.fr") // connexion au serveur HTTP de Polytech'UPMC
connect("www.polytech.upmc.fr", 25) // connexion au serveur SMTP de Polytech'UPMC
```

- ▶ En C++, la *signature* d'une fonction comprend les types de ses arguments.
- ▶ Du coup, des fonctions différentes peuvent avoir le même nom si leurs signatures sont différentes, c'est ce qu'on appelle la *surcharge*.

```
void connect (char *host = "localhost", int port = 80)
{
    // résolution du nom puis connexion
}

void connect (int ip = 0x7f000001, int port = 80)
{
    // connexion directe à l'adresse IP
}
```

- ▶ Pour que la surcharge fonctionne, le compilateur C++ fabrique en fait des noms plus longs comprenant les noms des types des arguments pour chaque fonction.

```
// les "vrais" noms des deux fonctions connect pourraient être ceux-ci :  
connect__char_ptr__int  
connect__int__int
```

- ▶ Afin de pouvoir interagir avec du code C il faut pouvoir sortir de ce mode de fonctionnement.
- ▶ La directive `extern "C"` le permet.

```
extern "C" {  
    // déclarations et définitions de fonctions  
    // dont le nom sera représenté à la manière de C  
  
    // pas de surcharge !  
}
```



- ▶ En C++, la surcharge peut aussi se faire sur les opérateurs (on verra cela plus tard).
- ▶ Grâce à ça, la gestion des entrées/sorties est grandement simplifiée.

## C

```
#include <stdio.h>

char name[256];
int age;

printf("Quel est ton nom ? ");
scanf("%s", &name);
printf("Quel âge as-tu ? ");
scanf("%d", &age);
printf("Bonjour %s, tu as %d ans\n",
       name, age);
```

## C++

```
#include <iostream>
using namespace std;

char name[256];
int age;

cout << "Quel est ton nom ? ";
cin >> name;
cout << "Quel âge as-tu ? ";
cin >> age;
cout << "Bonjour " << name
     << " tu as " << age << " ans"
     << endl;
```

- ▶ Les fonctions `malloc` et `free` de C sont disponibles, mais il vaut mieux leur préférer les opérateurs `new` et `delete`.
- ▶ Ces opérateurs fonctionnent de pair avec les objets C++, et font appel à leurs *constructeurs* et leurs *destructeurs* (on verra cela plus tard).

```
int *array = new int[10];  
// array est un pointeur sur un tableau de 10 int  
delete[] array;
```

```
Object *o = new Object;  
// o est pointeur sur un objet de type Object, qui a été instancié par son constructeur  
delete o;
```

- ▶ Paradigme de programmation inventé par Ole-Johan Dahl et Kristen Nygaard au début des années 1960 et poursuivi par les travaux d'Alan Kay dans les années 1970.
- ▶ L'idée est de définir des *briques logicielles* appelées *objets*.
- ▶ Un objet représente un concept, une idée, ou toute entité du monde physique.
- ▶ Il possède une structure interne (*implémentation*) et un comportement (*interface*).
- ▶ Le but de cette méthode est de permettre une meilleure *modélisation* des problèmes, et donc une meilleure programmation.
- ▶ Explosion de la POO dans les années 80 et 90.

- ▶ On peut programmer orienté objet dans n'importe quel langage.
- ▶ Certains langages offre un support natif de la POO :  
Smalltalk, Objective-C, Java, C++, Go, Python, Self, JavaScript, ...
- ▶ Mais pas tous de la même façon :
  - ▶ avec des classes qu'on instancie (C++, Java, Python);
  - ▶ avec des prototypes qu'on clone (Self, JavaScript);
  - ▶ d'autres choses moins clairement nommées (Go).
- ▶ Avec différentes façons de typer :
  - ▶ typage fort ou faible (Java vs. Python);
  - ▶ typage statique ou dynamique (C++ vs. Objective C).

- ▶ C++ est un langage objet à base de classes.
- ▶ Il utilise un typage statique et fort (mais non sûr).

- ▶ Un objet est constitué :
  - ▶ de données membres (*attributs, propriétés, ou variables d'instance*),
  - ▶ de fonctions membres (*méthodes*).
- ▶ La définition d'un type d'objet s'appelle une *classe*.
- ▶ On dit qu'on crée une *instance* d'une classe lors de la création d'un objet qui a cette classe pour type.

- ▶ Syntaxe similaire à `struct` de C (mais `class` remplace `struct`)
- ▶ Pas besoin de `typedef`, le nom est directement accessible.
- ▶ Les membres d'une classe peuvent être des fonctions.
- ▶ L'ordre des déclarations n'importe pas.

```
class Person
{
public:
    void set_name (string n)
    {
        // vérification de la validité de l'argument (e.g., pas de chiffre)
        // puis, si tout va bien :
        name = n;
    }

    void introduceSelf ()
    {
        cout << "Bonjour, je suis " << name << "." << endl;
    }

    string name;
};
```

- ▶ À l'intérieur d'une classe, on accède à ses membres simplement par leur nom.
- ▶ On peut accéder à soi-même avec le mot clef `this` qui est un pointeur sur l'instance de la classe (l'objet) qui a appelé la fonction.

```
class Person
{
public:
    void set_name (string name)
    {
        // vérification de la validité de l'argument (e.g., pas de chiffre)
        // puis, si tout va bien :
        this->name = name;
    }

    void introduceSelf ()
    {
        cout << "Bonjour, je suis " << name << '.' << endl;
    }

    string name;
};
```



► De l'extérieur : même syntaxe que pour les structures en C.

```
Person a;  
a.name = "Hildegarde";  
a.introduceSelf(); // affiche "Bonjour, je suis Hildegarde."  
  
Person *b = new Person;  
b->name = "Gontrand";  
b->introduceSelf(); // affiche "Bonjour, je suis Gontrand."  
delete b;
```



- ▶ Le code qu'on vient de voir n'est pas bon !
- ▶ Une des forces de la programmation orientée objet est l'*encapsulation*.
- ▶ Il s'agit de cacher les détails de l'implémentation d'une classe à ses utilisateurs.
- ▶ De cette façon,
  - ▶ l'implémentation peut changer sans que l'utilisateur ne s'en soucie tant que l'interface reste stable,
  - ▶ et les objets sont seuls responsables du maintien de leur cohérence interne.
- ▶ Cela évite le "code spaghetti".

- ▶ En C++, on ne peut pas vraiment cacher les détails de l'implémentation, on peut seulement en restreindre l'accès.
- ▶ Il existe trois niveaux de restriction :
  - ▶ publique : accès libre par tous,
  - ▶ privée : accès réservé aux fonctions membres de la classe propriétaire,
  - ▶ protégée : niveau intermédiaire que l'on verra plus tard en étudiant les notions d'*héritage* et de *hiérarchie* de classes.
- ▶ Par défaut dans une classe les membres sont privés.

```
class Person
{
    // ici les déclarations sont privées
public:
    // ici les déclarations sont publiques
    void set_name (string name);
    string name ();
    void introduceSelf ();
    void addFriend (Person p);
    bool isFriendWith (Person p);

private:
    // ici les déclarations sont privées
    string name_;
    string *friends_;
    int friends_count_;
};
```

## Exemple : encapsulation (1/2)

```
class Person
{
public:
    void set_name (string name)
    {
        // vérifications, puis
        name_ = name;
    }
    string name () { return name_; }
    void introduceSelf ()
    {
        cout << "Bonjour, je suis " << name_ << "." << endl;
    }
    void addFriend (Person p)
    {
        if (!isFriendWith(p)) {
            friends_[friends_count_++] = p.name_;
        }
    }
    bool isFriendWith (Person p)
    {
        for (int i = 0; i < friends_count_; i++) {
            if (friends_[i] == p.name_) return true;
        }
        return false;
    }

private:
    string name_, *friends_;
    int friends_count_;
};
```



## Exemple : encapsulation (2/2)

```
class Person
{
public:
    void set_name (string name)
    {
        // vérifications, puis
        name_ = name;
    }
    string name () { return name_; }
    void introduceSelf ()
    {
        cout << "Bonjour, je suis " << name_ << "." << endl;
    }
    void addFriend (const Person &p)
    {
        if (!isFriendWith(p)) {
            friends_[friends_count_++] = p.id_;
        }
    }
    bool isFriendWith (const Person &p)
    {
        for (int i = 0; i < friends_count_; i++) {
            if (friends_[i] == p.id_) return true;
        }
        return false;
    }

private:
    int id_;
    string name_;
    int *friends_, friends_count_;
};
```



- ▶ Le mot clef `struct` existe toujours en C++.
- ▶ Il fait la même chose que `class`, sauf que par défaut les déclarations sont publiques.

```
struct Foo
{
    // ici les déclarations sont publiques
private:
    // ici les déclarations sont privées
public:
    // ici les déclarations sont publiques
};
```

- ▶ Tous les membres d'une classe `Foo` doivent être déclarés dans `class Foo { ... }`.
- ▶ Pour les fonctions on peut n'écrire que leur prototype dans la classe et définir leur corps ailleurs.
- ▶ Dans ce cas, pour signifier qu'une fonction est membre d'une classe, on utilise l'*opérateur de résolution de portée*, noté "`::`".
- ▶ Cela permet de séparer l'interface de la classe (dans un fichier `.hpp`) de son implémentation (dans un fichier `.cpp`).

## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    void set_name (string name);
    string name ();

    void introduceSelf ();
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p);

private:
    int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## person.cpp

```

#include "person.hpp"
#include <iostream>
using namespace std;

//...

void Person::addFriend (const Person &p)
{
    if (!isFriendWith(p)) {
        friends_[friends_count_++] = p.id_;
    }
}

bool Person::isFriendWith (const Person &p)
{
    for (int i = 0; i < friends_count_; i++) {
        if (friends_[i] == p.id_) return true;
    }
    return false;
}

//...

```



- ▶ Un constructeur d'une classe est une fonction membre spéciale qui a le même nom que la classe et n'a pas de type de retour (ni d'instruction `return`).
- ▶ Le rôle d'un constructeur est d'initialiser un objet.
- ▶ Il est appelé immédiatement après que la mémoire nécessaire lui ait été allouée (de quelque façon que ce soit).



## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    Person ();
    Person (string name);

    void set_name (string name);
    string name ();

    void introduceSelf ();
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p);

private:
    int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## person.cpp

```

#include "person.hpp"
#include <iostream>
using namespace std;

//...

Person::Person ()
{
    id_ = getUniqueIdentifier(); // magie
    friends_count_ = 0;
    friends_ = new int[100];
}

Person::Person (string name)
{
    id_ = getUniqueIdentifier();
    friends_count_ = 0;
    friends_ = new int[100];

    set_name(name);
}

//...

```

- ▶ Un constructeur est toujours appelé lorsqu'un objet est instancié.

```
// appels explicites
Person a("Alphonsine");
Person f = Person("Dumontine");

// appel implicite (possible quand il existe un constructeur à un seul argument)
Person c = "Chrysogone";

// allocation dynamique
Person *m = new Person("Mélissandre");
delete m;

// objet anonyme
Person("Sigismond").introduceSelf();
c = Person("Philomène");
```

- ▶ Attention à ne pas confondre initialisation et affectation.

- ▶ Le constructeur par défaut est un constructeur sans argument (ou avec des valeurs par défaut pour tous ses arguments).
- ▶ Il est appelé chaque fois qu'un objet est créé sans appel explicite au constructeur.

```
Person p; // même chose que Person p = Person();  
Person q[5]; // 5 appels de Person();
```

```
Person *r = new Person; // même chose que Person *r = new Person();  
Person *s = new Person[5]; // 5 appels de Person();
```

```
delete r;  
delete[] s;
```

- ▶ Si aucun constructeur n'est défini, le compilateur en synthétise un.
- ▶ Par défaut il ne fait rien, mais si la classe a des objets membres, leur constructeur par défaut sera appelé.
- ▶ Si un constructeur est défini, même si il demande des arguments, alors aucun constructeur ne sera synthétisé.

- ▶ Le constructeur par copie est appelé lorsqu'un objet est initialisé en copiant un objet existant.
- ▶ Il doit prendre une référence ou une référence constante sur une instance de la classe comme premier argument, et avoir des valeurs par défaut pour ses autres arguments.
- ▶ S'il n'existe pas il est synthétisé par le compilateur ; il fera une copie de chaque membre (en appelant leur constructeur par copie si applicable).

```
Person a("Abélard");  
Person b = a; // construction de b par copie de a  
  
void foo (Person p); // fonction définie ailleurs  
foo(a); // passage par copie de a
```



## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    Person ();
    Person (string name);
    Person (const Person &p);

    void set_name (string name);
    string name ();

    void introduceSelf ();
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p);

private:
    int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## person.cpp

```

#include "person.hpp"
#include <iostream>
using namespace std;

//...

Person::Person (const Person &p)
{
    id = p.id_;

    name_ = p.name_;

    friends_count_ = p.friends_count_;

    friends_ = new int[100];
    for (int i = 0; i < friends_count_; i++) {
        friends_[i] = p.friends_[i];
    }
}

//...

```



- ▶ Quand une classe a des *objets membres*, ils doivent aussi être instanciés (et donc initialisés) lors de l'instanciation de cette classe.
- ▶ Il existe une syntaxe spéciale pour faire appel aux constructeurs des objets membres.
- ▶ Cette syntaxe marche en fait aussi avec les types de base.



## poem.hpp

```

#include "person.hpp"
#include <string>
using namespace std;

class Poem
{
public:
    Poem (string author, int verses_count);
    Poem (Person author, int verses_count);

    void set_author (string author);
    void set_author (Person author);
    Person author ();

    void set-verses (string verses[]);
    void display ();

private:
    Person author_;
    string *verses_;
};

```

## poem.cpp

```

#include "person.hpp"
#include "poem.hpp"
#include <iostream>
using namespace std;

Poem::Poem (string author = "Anonyme",
            int verses_count = 14)
    : author_(author)
{
    verses_ = new string[verses_count];
}

Poem::Poem (Person author,
            int verses_count = 14)
    : author_(author) // construction par copie
{
    verses_ = new string[verses_count];
}

void Poem::set-author (string author)
{
    author_.set_name(author);
}

void Poem::set-author (Person author)
{
    author_ = author;
}

//...

```



- ▶ Tout comme il y a besoin d'initialiser les objets lors de leur construction, il faut penser à nettoyer derrière eux lors de leur destruction.
- ▶ Un destructeur d'une classe est une fonction membre spéciale qui a le même nom que la classe avec un “~” devant, et n'a pas de paramètre ni de type de retour (ni d'instruction `return`).
- ▶ Il est appelé quand l'objet est détruit, soit par le système, soit explicitement avec `delete`.
- ▶ Par défaut, le compilateur synthétise un destructeur qui appelle récursivement les destructeurs des objets membres.



## Exemple : destructeur de Person

## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    Person ();
    Person (string name);
    Person (const Person &p);
    ~Person ();

    void set_name (string name);
    string name ();

    void introduceSelf ();
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p);

private:
    int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## person.cpp

```

#include "person.hpp"
#include <iostream>
using namespace std;

//...

Person::~Person ()
{
    delete[] friends_;
}

//...

```



- ▶ Une donnée membre d'une classe peut être qualifiée de `const`.
- ▶ Dans ce cas, il faut l'initialiser au moment de la construction de l'objet, et on a plus le droit de la modifier après.



## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    Person ();
    Person (string name);
    Person (const Person &p);
    ~Person ();

    void set_name (string name);
    string name ();

    void introduceSelf ();
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p);

private:
    const int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## person.cpp

```

#include "person.hpp"
#include <iostream>
using namespace std;

//...

Person::Person (string name)
    : id_(getUniqueIdentifier()) // magie
    , friends_count_(0)
{
    friends_ = new int[100];
    set_name(name);
}

Person::Person (const Person &p)
    : id_(p.id_)
    , friends_count_(p.friends_count_)
    , name_(p.name_)
{
    friends_ = new int[100];
    for (int i = 0; i < friends_count_; i++) {
        friends_[i] = p.friends_[i];
    }
}

//...

```

- ▶ Quand une fonction membre ne doit pas modifier l'objet, on peut la déclarer comme étant constante.
- ▶ On fait cela en ajoutant le mot clef `const` à la fin de sa déclaration.
- ▶ Il est conseillé de faire cela pour toutes les fonctions de consultation, qui pourront du coup s'appliquer aussi aux instances constantes.

## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    Person ();
    Person (string name);
    Person (const Person &p);
    ~Person ();

    void set_name (string name);
    string name () const;

    void introduceSelf () const;
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p) const;

private:
    const int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## person.cpp

```

#include "person.hpp"
#include <iostream>
using namespace std;

//...

string Person::name () const
{
    return name_;
}

//...

void Person::introduceSelf () const
{
    cout << "Bonjour, je suis "
         << name_ << "." << endl;
}

//...

```

## Exemple : fonctions membres constantes (2/2)

```
Person p("Phélipote");  
const Person h("Hardegin");  
  
p.introduceSelf(); // affiche "Bonjour, je suis Phélipote"  
p.set_name("Perronnele");  
p.introduceSelf(); // affiche "Bonjour, je suis Perronnele"  
  
h.introduceSelf(); // affiche "Bonjour, je suis Hardegin"  
h.set_name("Horménias"); // ERREUR : h est constante, et Person::set_name ne l'est pas
```



- ▶ Contrairement aux membres non statiques qu'on a vu jusqu'à présent, les membres statiques n'existent pas dans chaque objet mais une seule fois par classe.
- ▶ On peut déclarer des membres statiques (*variables de classe* et *méthodes de classe*) en précédant leur déclaration du mot clef `static`.
- ▶ Les fonctions membres statiques n'ont pas accès au pointeur `this` et ne peuvent faire appel qu'aux membres statiques de la classe.



## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    Person ();
    Person (string name);
    Person (const Person &p);
    ~Person ();

    void set_name (string name);
    string name () const;

    static int census ();

    void introduceSelf () const;
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p) const;

private:
    static int census_;
    const int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## person.cpp

```

#include "person.hpp"
#include <iostream>
using namespace std;

// initialisation d'une donnée
// membre statique
int Person::census_ = 0;

//...

Person::Person (string name)
    : id_(census_)
    , friends_count_(0)
{
    census_++;
    friends_ = new int[100];
    set_name(name);
}

//...

int Person::census () const
{
    return census_;
}

//...

```



- ▶ Une fonction amie est une fonction extérieure à une classe mais qui a les mêmes accès que les fonctions membres de la classe à ses éléments.
- ▶ La classe qui veut autoriser l'accès interne à une fonction amie doit la déclarer, en la précédant du mot clef `friend`.
- ▶ On peut aussi déclarer toute une classe comme étant amie.

## person.hpp

```

#include <string>
using namespace std;

class Person
{
public:
    Person ();
    Person (string name);
    Person (const Person &p);
    ~Person ();

    void set_name (string name);
    string name () const;

    static int census ();

    void introduceSelf () const;
    void addFriend (const Person &p);
    bool isFriendWith (const Person &p) const;

    friend bool Tests::testPerson ();

private:
    static int census_;
    const int id_;
    string name_;
    int *friends_, friends_count_;
};

```

## tests.cpp

```

//...

#include "person.hpp"

//...

bool Tests::testPerson ()
{
    bool pass = true;
    Person u("Ursénia"),
           m("Mézellinie"),
           o("Ouzélinie");

    pass = pass && u.name_ == "Ursénia";
    u.set_name("Lusithée");
    pass = pass && u.name_ == "Lusithée";
    pass = pass && u.name() == "Lusithée";
    pass = pass && u.friends_count_ == 0;
    pass = pass && u.census() == 0;
    u.addFriend(m);
    pass = pass && u.friends_count_ == 1;
    pass = pass && u.census() == 1;
    pass = pass && u.friends_[0] == m.id_;
    pass = pass && u.isFriendWith(m);
    pass = pass && !u.isFriendWith(o);

    return pass;
}

//...

```

```
#include <iostream>
using namespace std;

class A {
public:
    A () { cout << "Construction A" << endl; }
    A (const A &a) { cout << "Construction par copie A" << endl; }
    ~A () { cout << "Destruction A" << endl; }
};

class B {
public:
    B () { cout << "Construction B" << endl; }
    B (const B &b) : a_(b.a_) { cout << "Construction par copie B" << endl; }
    B (A a) : a_(a) { cout << "Construction B(a)" << endl; }
    B (int n, A &a) : a_(a) { cout << "Construction B(&a)" << endl; }
    ~B () { cout << "Destruction B" << endl; }
private:
    A a_;
};

class C {
public:
    C () { cout << "Construction C" << endl; }
    C (const C &c) : b_(c.b_) { cout << "Construction par copie C" << endl; }
    C (A a) : b_(a) { cout << "Construction C(a)" << endl; }
    C (int n, A &a) : b_(0, a) { cout << "Construction C(&a)" << endl; }
    ~C () { cout << "Destruction C" << endl; }
private:
    B b_;
};
```

- ▶ Quelles vont être les sorties des programmes suivants :
  - ▶ `A a;`
  - ▶ `B b;`
  - ▶ `C c;`
  - ▶ `A a; B b(a);`
  - ▶ `A a; A &a_ref = a; B b(0, a_ref);`
  - ▶ `A a; C c(a);`
  - ▶ `A a; A &a_ref = a; C c(0, a_ref);`

## Déroulement du cours

### Préambule

#### C++ : approche superficielle

#### Premières différences avec le C

- Déclaration des variables : placement libre
- Vrais Booléens
- Références
- Fonctions en ligne
- Valeurs par défaut des arguments des fonctions
- Surcharge des noms de fonctions
- Appel de fonctions écrites en C
- Entrées / sorties plus simples
- Allocations dynamiques de mémoire

#### Programmation orientée objet

- Différentes façons d'être orienté objet
- Le cas de C++

#### Les classes

- Syntaxe
- Accès aux membres
- Encapsulation
- Structures
- Définition des classes
- Constructeurs
- Destructeurs
- Membres constants
- Membres statiques

#### Amie

#### Mini-TD

- Code
- Questions