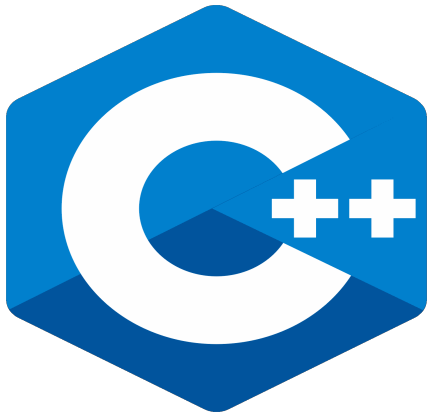
C++ : programmation orientée-objet

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Plan

Plan

Plan

C++

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

C++

Qu'est ce qu'une classe en POO ?

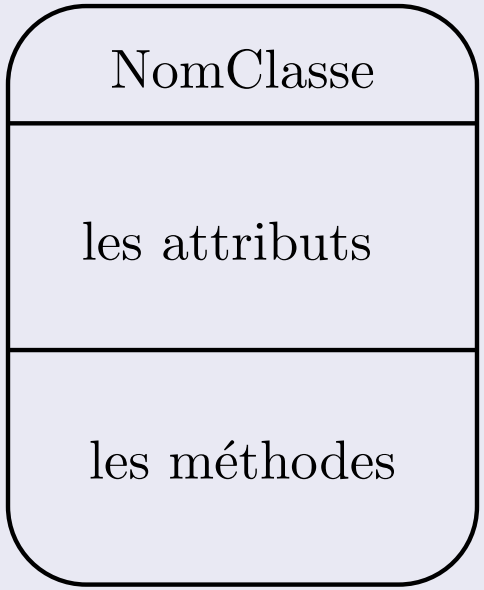
- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe
- instance $\equiv$ objet

C++

De quoi est composé une classe ?



- Attribut : [visibilité] + type + nom
- Méthode : [visibilité] + valeur de retour + nom + arguments ≡ signature : exactement comme les fonctions en procédurale

C++

Particularité du **C++** et bonnes pratiques

- Garder la fonction principale dans un fichier séparé
- Définir la structure de la classe dans un fichier `.h` portant le nom de la classe et l'implémentation dans un fichier `.cpp`
- Le mot-clé `this` permet de désigner l'objet courant.
- Pas de super-classe `Object` comme en **Java**, **C#**...

C++

Extension **VSC** pour la création de classes

C++ Class Creator

C++

Extension **VSC** pour la création de classes

C++ Class Creator

Comment créer une classe ?

- Appuyez sur `Alt` + `x`
- Saisir le nom de la classe (`Personne`) puis cliquez sur `Entrée`

C++

`Personne.h` **généré**

```
#ifndef PERSONNE_H
#define PERSONNE_H

#pragma once

class Personne
{
public:
    Personne();
    ~Personne();

private:

};

#endif
```

`Personne.cpp` **généré**

```
#include "Personne.h"

Personne::Personne()
{

}

Personne::~Personne()
{

}
```

C++

Ajoutons

- les attributs `num`, `nom`, `prénom` et `genre`
- une méthode : `afficherDetails()`

Remplaçons le contenu précédent de `Personne.h` par le suivant

```
#ifndef PERSONNE_H
#define PERSONNE_H

#include <iostream>

using namespace std;

class Personne {
    public: void afficherDetails() const;

    public: int num;
    public: string nom;
    public: string prenom;
    public: char genre;
};
#endif
```

Remplaçons le contenu précédent de `Personne.h` par le suivant

```
#ifndef PERSONNE_H
#define PERSONNE_H

#include <iostream>

using namespace std;

class Personne {
    public: void afficherDetails() const;

    public: int num;
    public: string nom;
    public: string prenom;
    public: char genre;
};
#endif
```


Remarques

- Par défaut, la visibilité des attributs et méthodes, en **C++**, est `private`.
- Les attributs ne seront donc pas accessibles depuis l'extérieur, il faut les déclarer `public`.
- Le mot clé `const` indique au compilateur que la méthode ne modifie pas l'objet.

C++

Le code précédent peut-être factorisé (en supprimant les `public` répétés)

```
#ifndef PERSONNE_H
#define PERSONNE_H

#include <iostream>

using namespace std;

class Personne
{
public:
    void afficherDetails() const;

public:
    int num;
    string nom;
    string prenom;
    char genre;
};

#endif
```

C++

commençons par déclarer `afficherDetails` **dans** `Personne.cpp`

```
#include "Personne.h"

void Personne::afficherDetails() const
{
}
```

C++

commençons par déclarer `afficherDetails` dans `Personne.cpp`

```
#include "Personne.h"

void Personne::afficherDetails() const
{
}
```

Ajoutons ensuite le contenu suivant

```
#include "Personne.h"

void Personne::afficherDetails() const
{
    cout << num << " " << prenom << " ";
    cout << nom << " " << genre  << endl;
}
```

C++

Pour instancier (créer un objet de) la classe : deux solutions

- sans pointeur :
  - déclarer une variable (comme toute autre variable) de type `Personne`
  - affecter des valeurs à ses attributs en utilisant l'opérateur `.`
- avec pointeur :
  - déclarer un pointeur sur `Personne`
  - utiliser l'opérateur `new` + nom de la classe pour le créer dynamiquement
  - affecter des valeurs à ses attributs en utilisant l'opérateur `->`

C++

Déclarons un objet de la classe `Personne`, comme une variable, dans le `main`

```
#include <iostream>
#include "Personne.h"
using namespace std;

int main()
{
    Personne personne;
}
```

C++

Déclarons un objet de la classe `Personne`, comme une variable, dans le `main`

```
#include <iostream>
#include "Personne.h"
using namespace std;

int main()
{
    Personne personne;
}
```

Pour affecter une valeur à un attribut

```
personne.nom = "wick";
```

C++

Déclarons un objet de la classe `Personne`, comme une variable, dans le `main`

```
#include <iostream>
#include "Personne.h"
using namespace std;

int main()
{
    Personne personne;
}
```

Pour affecter une valeur à un attribut

```
personne.nom = "wick";
```

Pour les autres attributs

```
personne.prenom = "john";
personne.genre = 'M';
personne.num = 100;
```

C++

Pour appeler une méthode

```
personne.afficherDetails();
// affiche 100 john wick M
```

C++

Deuxième solution : déclarons un pointeur de la classe `Personne`

```
Personne *personne;
```

C++

Deuxième solution : déclarons un pointeur de la classe `Personne`

```
Personne *personne;
```

Pour créer l'objet, il faut utiliser l'opérateur `new`

```
personne = new Personne();
```

C++

Deuxième solution : déclarons un pointeur de la classe `Personne`

```
Personne *personne;
```

Pour créer l'objet, il faut utiliser l'opérateur `new`

```
personne = new Personne();
```

Les deux étapes précédentes peuvent-être fusionnées

```
Personne *personne = new Personne();
```

Pour les attributs

```
personne->nom = "wick";
personne->prenom = "john";
personne->genre = 'M';
personne->num = 100;
```

C++

Pour les attributs

```
personne->nom = "wick";
personne->prenom = "john";
personne->genre = 'M';
personne->num = 100;
```


Pour appeler une méthode

```
personne->afficherDetails();
// affiche 100 john wick M
```

C++

Remarques

- Un objet non-constant a accès aux méthodes non-constantes et aux méthodes constantes.
- Un objet `constant` (par exemple `const Personne p` a accès uniquement aux méthodes `const` de l'objet.

C++

Hypothèse

Supposant que l'on n'accepte seulement les valeurs `M` ou `F` pour l'attribut `genre` de la classe `Personne`

C++

Hypothèse

Supposant que l'on n'accepte seulement les valeurs `M` ou `F` pour l'attribut `genre` de la classe `Personne`

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à `private`)
2. Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

Convention

- Mettre la visibilité `private` ou `protected` pour tous les attributs
- Mettre la visibilité `public` pour toutes les méthodes

C++

Mettons la visibilité `private` pour tous les attributs de la classe `Personne`

```
class Personne
{
public:
    void afficherDetails() const;

private:
    int num;
    string nom;
    string prenom;
    char genre;
};
```

C++

Mettons la visibilité `private` pour tous les attributs de la classe `Personne`

```
class Personne
{
public:
    void afficherDetails() const;

private:
    int num;
    string nom;
    string prenom;
    char genre;
};
```

Compilation ⇒ erreur

Attributs privés ⇒ inaccessibles.

C++

Déclarons les setters dans `Personne.h`

```
class Personne
{
public:
    void setNum(int num);
    void setNom(string nom);
    void setPrenom(string prenom);
    void setGenre(char genre);
    void afficherDetails() const;

private:
    int num;
    string nom;
    string prenom;
    char genre;
};
```

L'implémentation dans `Personne.cpp`

```
void Personne::setNum(int num)
{
    this->num = num;
}

void Personne::setNom(string nom)
{
    this->nom = nom;
}

void Personne::setPrenom(string prenom)
{
    this->prenom = prenom;
}

void Personne::setGenre(char genre)
{
   if(genre == 'F' || genre == 'M')
   {
        this->genre = genre;
   }
   else
   {
        this->genre = 'M';
   }
}
```

C++

Utilisons `const&` **dans** `Personne.h`

```
class Personne
{
public :
  void setNum(int const& num);
  void setNom(string const& nom);
  void setPrenom(string const& prenom);
  void setGenre(char const& genre);
  void afficherDetails() const;

private:
  int num;
  string nom;
  string prenom;
  char genre;
};
```

Adaptons `Personne.cpp`

```
void Personne::setNum(int const& num)
{
    this->num = num;
}

void Personne::setNom(string const& nom)
{
    this->nom = nom;
}

void Personne::setPrenom(string const& prenom)
{
    this->prenom = prenom;
}

void Personne::setGenre(char const& genre)
{
   if(genre == 'F' || genre == 'M')
        this->genre = genre;
   else
        this->genre = 'M';
}
```

C++

Remarque

`this` : pointeur sur l'objet courant

C++

Mettons à jour la fonction principale

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne;
    personne.setNom("wick");
    personne.setPrenom("john");
    personne.setGenre('M');
    personne.setNum(100);
    personne.afficherDetails();
    // affiche 100 john wick M
    return 0;
}
```

C++

Testons avec une valeur autre que `M` ou `F`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne;
    personne.setNom("wick");
    personne.setPrenom("john");
    personne.setGenre('X');
    personne.setNum(100);
    personne.afficherDetails();
    // affiche 100 john wick M
    return 0;
}
```

C++

Hypothèse

Si on voulait afficher les attributs (privés) de la classe `Personne`, un par un, dans la fonction principale.

C++

Hypothèse

Si on voulait afficher les attributs (privés) de la classe `Personne`, un par un, dans la fonction principale.

Démarche

Définir des méthodes qui retournent les valeurs des attributs (les `getter`)

C++

Déclarons les getters dans `Personne.h`

```
class Personne
{
public:
  void setNum(int const& num);
  void setNom(string const& nom);
  void setPrenom(string const& prenom);
  void setGenre(char const& genre);
  int getNum() const;
  string getNom() const;
  string getPrenom() const;
  char getGenre() const;
  void afficherDetails() const;

private:
  int num;
  string nom;
  string prenom;
  char genre;
};
```

C++

L'implémentation dans `Personne.cpp`

```
int Personne::getNum() const
{
    return num;
}

string Personne::getNom() const
{
    return nom;
}

string Personne::getPrenom() const
{
    return prenom;
}

char Personne::getGenre() const
{
    return genre;
}
```

C++

Mettons à jour la fonction principale

```
#include <iostream>
#include "Personne.h"
using namespace std;

int main()
{
    Personne personne;
    personne.setNom("wick");
    personne.setPrenom("john");
    personne.setGenre('X');
    personne.setNum(100);
    cout << personne.getNum() << " " ;
    cout << personne.getNom() << " " ;
    cout << personne.getPrenom() << " " ;
    cout << personne.getGenre() << endl;
    // affiche 100 wick john M
    return 0;
}
```

C++

Le constructeur

- Une méthode particulière portant le nom de la classe et ne retournant pas de valeur.
- Toute classe en **C++** a un constructeur par défaut sans paramètre.
- Ce constructeur sans paramètre n'a aucun code.
- On peut le définir explicitement si un traitement est nécessaire (ou si on veut vérifier l'appel).
- La déclaration d'un objet de la classe (par exemple `Personne personne`) fait appel à ce constructeur sans paramètre.
- Toutefois, et pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs.

Commençons par définir le prototype de ce nouveau constructeur dans `Personne.h`

```
class Personne
{
public:
    // constructeurs
    Personne(int num, string nom, string prenom, char genre);
    // setters et getters
    void setNum(int const &num);
    void setNom(string const &nom);
    void setPrenom(string const &prenom);
    void setGenre(char const &genre);
    int getNum() const;
    string getNom() const;
    string getPrenom() const;
    char getGenre() const;
    // autres méthodes
    void afficherDetails() const;

private:
    int num;
    string nom;
    string prenom;
    char genre;
};
```

C++

Implémentons le constructeur dans `Personne.cpp` : pour préserver la cohérence, il faut que le constructeur contrôle la valeur de l'attribut `genre`

```
Personne::Personne(int num, string nom, string prenom, char genre)
{
    this->num = num;
    this->nom = nom;
    this->prenom = prenom;
    if (genre == 'F' || genre == 'M')
    {
        this->genre = genre;
    }
    else
    {
        this->genre = 'M';
    }
}
```

C++

Remarque

L'instruction `Personne personne` est en rouge.

C++

Remarque

L'instruction `Personne personne` est en rouge.

Explication

Le constructeur par défaut n'existe plus.

C++

Remarque

L'instruction `Personne personne` est en rouge.

Explication

Le constructeur par défaut n'existe plus.

Solution

Recréer un constructeur sans paramètre

C++

Ajoutons le prototype du constructeur sans paramètre dans `Personne.h`

```
class Personne
{
public:
    // constructeurs
    Personne();
    Personne(int num, string nom, string prenom, char genre);
    // + les autres méthodes

private:
    int num;
    string nom;
    string prenom;
    char genre;
```

C++

Ajoutons le prototype du constructeur sans paramètre dans `Personne.h`

```
class Personne
{
public:
    // constructeurs
    Personne();
    Personne(int num, string nom, string prenom, char genre);
    // + les autres méthodes

private:
    int num;
    string nom;
    string prenom;
    char genre;
```

Implémentons ce nouveau constructeur dans `Personne.cpp`

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

C++

Mettons à jour la fonction principale pour tester les deux constructeurs

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne;
    personne.setNom("wick");
    personne.setPrenom("john");
    personne.setGenre('X');
    personne.setNum(100);
    personne.afficherDetails();
    // affiche 100 john wick M
    Personne personne2(101, "dalton", "jack", 'M');
    personne2.afficherDetails();
    // affiche 101 jack dalton M
    return 0;
}
```

C++

On peut aussi appelé le `setter` dans le constructeur

```
Personne::Personne(int num, string nom, string prenom, char genre)
{
    this->num = num;
    this->nom = nom;
    this->prenom = prenom;
    this->setGenre(genre);
}
```

C++

Relançons le main précédent

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne;
    personne.setNom("wick");
    personne.setPrenom("john");
    personne.setGenre('X');
    personne.setNum(100);
    personne.afficherDetails();
    // affiche 100 john wick M
    Personne personne2(101, "dalton", "jack", 'M');
    personne2.afficherDetails();
    // affiche 101 jack dalton M
    return 0;
}
```

C++

C++ nous offre la possibilité d'initialiser les attributs de la manière suivante

```
Personne::Personne(int num, string nom, string prenom, char genre) :
   num(num), nom(nom), prenom(prenom)
{
    this->setGenre(genre);
}
```

C++

C++ nous offre la possibilité d'initialiser les attributs de la manière suivante

```
Personne::Personne(int num, string nom, string prenom, char genre) :
   num(num), nom(nom), prenom(prenom)
{
    this->setGenre(genre);
}
```


On peut initialiser nos attributs avec des constantes.

C++

Relançons encore le `main` précédent

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne;
    personne.setNom("wick");
    personne.setPrenom("john");
    personne.setGenre('X');
    personne.setNum(100);
    personne.afficherDetails();
    // affiche 100 john wick M
    Personne personne2(101, "dalton", "jack", 'M');
    personne2.afficherDetails();
    // affiche 101 jack dalton M
    return 0;
}
```

C++

On peut aussi déclarer un pointeur et utiliser le constructeur à plusieurs paramètres

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne;
    personne.setNom("wick");
    personne.setPrenom("john");
    personne.setGenre('X');
    personne.setNum(100);
    personne.afficherDetails();
    // affiche 100 john wick M
    Personne *personne2 = new Personne(101, "dalton", "jack", 'M');
    personne2->afficherDetails();
    // affiche 101 jack dalton M
    return 0;
}
```

C++

Constructeur de conversion [**C++ 11**]

- Un constructeur de conversion permet à l'utilisateur d'affecter une valeur à un objet d'une classe comme toute autre variable
- Un constructeur de conversion a comme rôle la construction d'un objet à partir d'une affectation

Commençons par définir le prototype de ce constructeur dans `Personne.h`

```
Personne(int num);
```

Commençons par définir le prototype de ce constructeur dans `Personne.h`

```
Personne(int num);
```

Implémentons ce constructeur dans `Personne.cpp`

```
Personne::Personne(int num) : num(num), nom("doe"), prenom("john"),
   genre('M')
{
}
```

Commençons par définir le prototype de ce constructeur dans `Personne.h`

```
Personne(int num);
```

Implémentons ce constructeur dans `Personne.cpp`

```
Personne::Personne(int num) : num(num), nom("doe"), prenom("john"),
   genre('M')
{
}
```

Ainsi on peut écrire

```
Personne personne4(99);
personne4.afficherDetails();
// affiche 99 doe john M
```

Commençons par définir le prototype de ce constructeur dans `Personne.h`

```
Personne(int num);
```

Implémentons ce constructeur dans `Personne.cpp`

```
Personne::Personne(int num) : num(num), nom("doe"), prenom("john"),
   genre('M')
{
}
```

Ainsi on peut écrire

```
Personne personne4(99);
personne4.afficherDetails();
// affiche 99 doe john M
```

Et aussi

```
Personne personne4 = 99;
personne4.afficherDetails();
// affiche 99 doe john M
```

C++

Et si on considère la fonction suivante (à déclarer avant `main`)

```
void getNomComplet(Personne p)
{
    cout << p.getPrenom() << " " << p.getNom() << endl;
}
```

C++

Et si on considère la fonction suivante (à déclarer avant `main`)

```
void getNomComplet(Personne p)
{
    cout << p.getPrenom() << " " << p.getNom() << endl;
}
```

Alors le constructeur de conversion sera appelé

```
getNomComplet(100);
// affiche john doe
```

C++

Remarques

- Tout constructeur acceptant un seul paramètre est considéré en **C++** comme constructeur de conversion.
- Pour interdire les conversions via le constructeur mono-paramètre, on peut utiliser le mot-clé `explicit`.

C++

Remarques

- Tout constructeur acceptant un seul paramètre est considéré en **C++** comme constructeur de conversion.
- Pour interdire les conversions via le constructeur mono-paramètre, on peut utiliser le mot-clé `explicit`.

En ajoutant `explicit` au prototype du constructeur dans `Personne.h`, les deux dernières conversions seront interdites

```
explicit Personne(int num);
```

C++

Le destructeur

- Une méthode portant le nom de la classe et précédée par $\sim$
- Appelé lorsque l'objet n'est plus référencé ou si l'opérateur `delete` est utilisé.
- L'opérateur `delete` doit être utilisé pour détruire les objets dynamiques créés avec l'opérateur `new`

C++

Définissons le prototype du destructeur dans `Personne.h`

```
class Personne
{
public:
    // constructeurs
    Personne();
    Personne(int num);
    Personne(int num, string nom, string prenom, char genre);
    // desctructeur
    ~Personne();
    // + les autres méthodes

private:
    int num;
    string nom;
    string prenom;
    char genre;
```

C++

Définissons le prototype du destructeur dans `Personne.h`

```
class Personne
{
public:
    // constructeurs
    Personne();
    Personne(int num);
    Personne(int num, string nom, string prenom, char genre);
    // desctructeur
    ~Personne();
    // + les autres méthodes

private:
    int num;
    string nom;
    string prenom;
    char genre;
```

Implémentons le destructeur dans `Personne.cpp`

```
Personne::~Personne()
{
    cout << "objet personne num " << num << " détruit" << endl;
}
```

Considérons les deux objets suivants dans `main`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne(100, "wick", "john", 'M');
    personne.afficherDetails();
    Personne *personne2 = new Personne (101, "dalton", "jack", 'M');
    personne2->afficherDetails();
    return 0;
}
```

Considérons les deux objets suivants dans `main`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne(100, "wick", "john", 'M');
    personne.afficherDetails();
    Personne *personne2 = new Personne (101, "dalton", "jack", 'M');
    personne2->afficherDetails();
    return 0;
}
```

En exécutant, (Suprise) le résultat est

```
100 john wick M
101 jack dalton M
objet personne num 100 détruit
```

Considérons les deux objets suivants dans `main`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne(100, "wick", "john", 'M');
    personne.afficherDetails();
    Personne *personne2 = new Personne (101, "dalton", "jack", 'M');
    personne2->afficherDetails();
    return 0;
}
```

En exécutant, (Suprise) le résultat est

```
100 john wick M
101 jack dalton M
objet personne num 100 détruit
```

Pourquoi le deuxième objet n'a pas été détruit ?

- car il s'agit d'un pointeur
- il faut utiliser l'opérateur `delete()`

Utilisons `delete` pour détruire l'objet pointeur

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne(100, "wick", "john", 'M');
    personne.afficherDetails();
    Personne *personne2 = new Personne(101, "dalton", "jack", 'M');
    personne2->afficherDetails();
    delete (personne2);
    return 0;
}
```

Utilisons `delete` pour détruire l'objet pointeur

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne(100, "wick", "john", 'M');
    personne.afficherDetails();
    Personne *personne2 = new Personne(101, "dalton", "jack", 'M');
    personne2->afficherDetails();
    delete (personne2);
    return 0;
}
```

En exécutant, le résultat est

```
100 john wick M
101 jack dalton M
objet personne num 101 détruit
objet personne num 100 détruit
```

C++

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

C++

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe `Personne`)

C++

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe `Personne`)

Définition

Un attribut dont la valeur est partagée par toutes les instances de la classe est appelée : attribut statique ou attribut de classe

C++

Exemple

- Si on veut créer un attribut contenant le nombre d'objets créés à partir de la classe `Personne`
- Notre attribut doit être `static`, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut

C++

Ajoutons un attribut `static` appelé `nbrPersonnes` dans `Personne.h`

```
class Personne
{
public:
    // les méthodes
private:
    // les autres attributs
    static int nbrPersonnes;
```

Dans `Personne.cpp`, nous initialisons la valeur de cet attribut `static`

```
int Personne::nbrPersonnes = 0;
```

Dans `Personne.cpp`, nous initialisons la valeur de cet attribut `static`

```
int Personne::nbrPersonnes = 0;
```

Ensuite nous l'incrémentons dans les quatre constructeurs

```
Personne::Personne()
{
    nbrPersonnes++;
}

Personne::Personne(int num) : num(num), nom("doe"), prenom("john"),
  genre('M')
{
    nbrPersonnes++;
}

Personne::Personne(int num, string nom, string prenom, char genre) :
  num(num), nom(nom), prenom(prenom)
{
    this->setGenre(genre);
    nbrPersonnes++;
}
```

C++

Définissons un `getter` pour l'attribut static `nbrPersonnes` dans `Personne.h`

```
static int getNbrPersonnes();
```

C++

Définissons un `getter` pour l'attribut static `nbrPersonnes` dans `Personne.h`

```
static int getNbrPersonnes();
```

Implémentons ce `getter` dans `Personne.cpp`

```
int Personne::getNbrPersonnes()
{
    return nbrPersonnes;
}
```

Modifions le `main` et utilisons la classe `Personne` pour appeler `getNbrPersonnes()`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne(100, "wick", "john", 'M');
    cout << Personne::getNbrPersonnes() << endl;
    Personne *personne2 = new Personne (101, "dalton", "jack", 'M');
    cout << Personne::getNbrPersonnes() << endl;
    delete(personne2);
    return 0;
}
```

Modifions le `main` et utilisons la classe `Personne` pour appeler `getNbrPersonnes()`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Personne personne(100, "wick", "john", 'M');
    cout << Personne::getNbrPersonnes() << endl;
    Personne *personne2 = new Personne (101, "dalton", "jack", 'M');
    cout << Personne::getNbrPersonnes() << endl;
    delete(personne2);
    return 0;
}
```

Le résultat est

```
0
1
2
objet personne num 101 détruit
objet personne num 100 détruit
```

C++

Fonction amie (friend function)

- Fonction dont le prototype est déclaré dans une classe (comme toute méthode) mais avec le mot-clé `friend`
- L'implémentation de cette fonction ne doit pas être préfixée par `NomClasse::nomMethode()`
- Cette fonction prend généralement comme paramètre un objet de cette classe et elle a donc accès à tous ses attributs privés et protégés

C++

Définissons une fonction amie dans `Personne.h` (`&` pour pouvoir modifier l'objet)

```
// fonctions amies
friend void devenirNiHommeNiFemme(Personne &p);
```

C++

Définissons une fonction amie dans `Personne.h` (`&` pour pouvoir modifier l'objet)

```
// fonctions amies
friend void devenirNiHommeNiFemme(Personne &p);
```

Implémentons cette méthode dans `main.cpp` (par exemple)

```
void devenirNiHommeNiFemme(Personne & p)
{
    p.genre = 'D';
}
```

C++

Définissons une fonction amie dans `Personne.h` (`&` pour pouvoir modifier l'objet)

```
// fonctions amies
friend void devenirNiHommeNiFemme(Personne &p);
```

Implémentons cette méthode dans `main.cpp` (par exemple)

```
void devenirNiHommeNiFemme(Personne & p)
{
    p.genre = 'D';
}
```

Testons cette fonction amie dans le `main`

```
Personne personne(100, "wick", "john", 'M')
devenirNiHommeNiFemme(personne);
personne.afficherDetails();
// affiche john wick D
```

C++

Méthode `inline`

- Une méthode implémentée dans le fichier `.h` est implicitement `inline`.
- Une méthode préfixée par `inline` est explicitement `inline`.
- Le prototype d'une méthode explicitement `inline` doit être défini dans un fichier `.h`, son implémentation aussi (dans le `.h`).
- Les getter et setter est un bon exemple de méthode `inline`

Pour tester, modifions quelques déclarations dans `Personne.h`

```
inline void setNum(int const& num);
void setNom(string const& nom) { this->nom = nom;};
```

Pour tester, modifions quelques déclarations dans `Personne.h`

```
inline void setNum(int const& num);
void setNom(string const& nom) { this->nom = nom;};
```

la méthode `setNum()` **est explicitement déclarée** `inline`**, il faut donc l'implémenter dans** `Personne.h` **après la classe**

```
void Personne::setNum(int const& num)
{
    this->num = num;
}
```

Pour tester, modifions quelques déclarations dans `Personne.h`

```
inline void setNum(int const& num);
void setNom(string const& nom) { this->nom = nom;};
```

la méthode `setNum()` **est explicitement déclarée** `inline`**, il faut donc l'implémenter dans** `Personne.h` **après la classe**

```
void Personne::setNum(int const& num)
{
    this->num = num;
}
```

Testons les nouveaux setters dans `main`

```
Personne personne(100, "wick", "john", 'M');
personne.setNum(99);
personne.setNom("abruzzi");
personne.afficherDetails();
// affiche 99 john abruzzi M
// objet personne num 99 détruit
```

C++

Opérateur de conversion [**C++ 11**]

- Une classe peut avoir plusieurs opérateurs de conversion : un opérateur au plus par type
- L'opérateur de conversion permet de convertir un objet de cette classe type (objet ou primitif)

C++

Dans `Personne.h`**, commençons par définir un opérateur qui transforme un objet de type** `Personne` **en** `int`

```
operator int();
```

C++

Dans `Personne.h`**, commençons par définir un opérateur qui transforme un objet de type** `Personne` **en** `int`

```
operator int();
```

Implémentons cet opérateur dans `Personne.cpp`

```
Personne::operator int()
{
    return num;
}
```

C++

Dans `Personne.h`, commençons par définir un opérateur qui transforme un objet de type `Personne` en `int`

```
operator int();
```

Implémentons cet opérateur dans `Personne.cpp`

```
Personne::operator int()
{
    return num;
}
```

Testons cet opérateur dans le `main`

```
Personne personne(100, "wick", "john", 'M');
int x = personne;
cout << x << endl;
// affiche 100
// objet personne num 100 détruit
```

C++

Exercice

- Définir une classe `Adresse` avec trois attributs privés (`rue`, `codePostal` et `ville`) de type chaîne de caractère
- Définir un constructeur avec trois paramètres, les getters et les setters
- Dans la classe `Personne`, ajouter un attribut `adresse` de type `Adresse` et définir un nouveau constructeur à quatre paramètres et le getter et le setter de ce nouvel attribut
- Dans la fonction principale `main`, créer un objet de type `Adresse` et l'affecter à un de type `Personne` (qu'il faut créer aussi)
- Afficher tous les attributs de cet objet de la classe `Personne`

C++

Le fichier `Adresse.h`

```
#ifndef ADRESSE_H
#define ADRESSE_H

#include <string>

using namespace std;

class Adresse
{
public :
    Adresse();
    Adresse(string rue, string codePostal, string ville);
    void setRue(string const& rue);
    void setVille(string const& ville);
    void setCodePostal(string const& codePostal);
    string getRue() const;
    string getVille() const;
    string getCodePostal() const;

private:
    string rue;
    string codePostal;
    string ville;
};
#endif
```

C++

Le fichier `Adresse.cpp` (partie 1)

```
#include "Adresse.h"

Adresse::Adresse()
{
}

Adresse::Adresse(string rue, string codePostal, string ville) : rue(rue
   ), ville(ville), codePostal(codePostal)
{
}

void Adresse::setRue(string const &rue)
{
    this->rue = rue;
}

void Adresse::setVille(string const &ville)
{
    this->ville = ville;
}
```

C++

Le fichier `Adresse.cpp` (partie 2)

```
void Adresse::setCodePostal(string const &codePostal)
{
    this->codePostal = codePostal;
}

string Adresse::getRue() const
{
    return rue;
}

string Adresse::getVille() const
{
    return ville;
}

string Adresse::getCodePostal() const
{
    return codePostal;
}
```

C++

Importons la classe `Adresse` dans `Personne.h`

```
#include "Adresse.h"
```

C++

Importons la classe `Adresse` dans `Personne.h`

```
#include "Adresse.h"
```

Déclarons un attribut de type `Adresse` dans `Personne.h`

```
Adresse adresse;
```

C++

Importons la classe `Adresse` dans `Personne.h`

```
#include "Adresse.h"
```

Déclarons un attribut de type `Adresse` dans `Personne.h`

```
Adresse adresse;
```

Définissons un nouveau constructeur et les getter/setter

```
// le nouveau constructeur
Personne(int num, string nom, string prenom, char genre,
   Adresse adresse);

// le setter
void setAdresse(Adresse adresse);

// le getter
Adresse getAdresse() const;
```

C++

Implémentons le nouveau constructeur dans `Personne.cpp`

```
Personne::Personne(int num, string nom, string prenom, char genre,
   Adresse adresse): num(num), nom(nom), prenom(prenom), adresse(
   adresse)
{
    this->setGenre(genre);
    nbrPersonnes++;
}
```

C++

Implémentons le nouveau constructeur dans `Personne.cpp`

```
Personne::Personne(int num, string nom, string prenom, char genre,
   Adresse adresse): num(num), nom(nom), prenom(prenom), adresse(
   adresse)
{
    this->setGenre(genre);
    nbrPersonnes++;
}
```

Implémentons les getter/setter dans `Personne.cpp`

```
void Personne::setAdresse(Adresse adresse)
{
    this->adresse = adresse;
}

Adresse Personne::getAdresse() const
{
    return adresse;
}
```

Testons cela dans la fonction principale `main`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Adresse adresse("paradis", "13015", "Marseille");
    Personne personne3(102, "jordan", "mike", 'M', adresse);
    cout << personne3.getNom() << " " ;
    cout << personne3.getPrenom() << " " ;
    cout << personne3.getAdresse().getCodePostal() << endl;
    return 0;
}
```

Testons cela dans la fonction principale `main`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Adresse adresse("paradis", "13015", "Marseille");
    Personne personne3(102, "jordan", "mike", 'M', adresse);
    cout << personne3.getNom() << " " ;
    cout << personne3.getPrenom() << " " ;
    cout << personne3.getAdresse().getCodePostal() << endl;
    return 0;
}
```

Le résultat est

```
jordan mike 13015
```

C++

Essayons de modifier le code postal de cette personne

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Adresse adresse("paradis", "13015", "Marseille");

    Personne personne3(102, "jordan", "mike", 'M', adresse);
    cout << personne3.getAdresse().getCodePostal() << endl;
    // affiche 13015

    personne3.getAdresse().setCodePostal("69000");
    cout << personne3.getAdresse().getCodePostal() << endl;
    // affiche 13015
    return 0;
}
```

Question

Pourquoi le code postal n'a pas été changé ?

Question

Pourquoi le code postal n'a pas été changé ?

Réponse

Car le getter d'`adresse` dans `Personne` retourne une valeur (pas une référence)

Modifions `getAdresses()` **dans** `Personne.h`

```
Adresse & getAdresse();
```

C++

Modifions `getAdresses()` **dans** `Personne.h`

```
Adresse & getAdresse();
```

Modifions `getAdresses()` **dans** `Personne.cpp`

```
Adresse & Personne::getAdresse()
{
    return adresse;
}
```

C++

En testant maintenant, le code postal a bien été modifié

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Adresse adresse("paradis", "13015", "Marseille");

    Personne personne3(102, "jordan", "mike", 'M', adresse);
    cout << personne3.getAdresse().getCodePostal() << endl;
    // affiche 13015

    personne3.getAdresse().setCodePostal("69000");
    cout << personne3.getAdresse().getCodePostal() << endl;
    // affiche 69000
    return 0;
}
```

C++

Différence avec la solution précédente

- Dans la classe `Personne`, ajouter un pointeur vers la classe Adresse `*adresse`
- Modifier le constructeur, le getter et le setter de ce nouvel attribut
- Dans la fonction principale `main`, créer un pointeur sur `Adresse` et l'affecter à un de type `Personne`
- Afficher tous les attributs de cet objet de la classe `Personne`

C++

Déclarons un pointeur d'`Adresse` dans `Personne.h`

```
Adresse *adresse;
```

C++

Déclarons un pointeur d'`Adresse` dans `Personne.h`

```
Adresse *adresse;
```

Modifions aussi les constructeurs, getter et setter

```
// le nouveau constructeur
Personne(int num, string nom, string prenom, char
   genre, Adresse *adresse);

// le setter
void setAdresse(Adresse *adresse);

// le getter
Adresse * getAdresse() const;
```

Dans `Personne.cpp`, rien ne change à part la signature pour le constructeur

```
Personne::Personne(int num, string nom, string prenom, char genre,
   Adresse *adresse): num(num), nom(nom), prenom(prenom), adresse(
   adresse)
{
    this->setGenre(genre);
    nbrPersonnes++;
}
```

Dans `Personne.cpp`, rien ne change à part la signature pour le constructeur

```
Personne::Personne(int num, string nom, string prenom, char genre,
   Adresse *adresse): num(num), nom(nom), prenom(prenom), adresse(
   adresse)
{
    this->setGenre(genre);
    nbrPersonnes++;
}
```

De même pour les getter/setter

```
void Personne::setAdresse(Adresse *adresse)
{
    this->adresse = adresse;
}

Adresse * Personne::getAdresse() const
{
    return adresse;
}
```

Modifions l'instanciation d'`Adresse` et la récupération du `codePostal`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Adresse *adresse =  new Adresse("paradis", "13015", "Marseille");
    Personne personne3(102, "jordan", "mike", 'M', adresse);
    cout << personne3.getNom() << " " ;
    cout << personne3.getPrenom() << " " ;
    cout << personne3.getAdresse()->getCodePostal() << endl;
    return 0;
}
```

Modifions l'instanciation d'`Adresse` et la récupération du `codePostal`

```
#include <iostream>
#include "Personne.h"

using namespace std;

int main()
{
    Adresse *adresse =  new Adresse("paradis", "13015", "Marseille");
    Personne personne3(102, "jordan", "mike", 'M', adresse);
    cout << personne3.getNom() << " " ;
    cout << personne3.getPrenom() << " " ;
    cout << personne3.getAdresse()->getCodePostal() << endl;
    return 0;
}
```

Le résultat est

```
jordan mike 13015
```

Si on copie l'objet `personne` et on modifie l'`adresse`

```
Adresse *adresse =  new Adresse("paradis", "13015", "Marseille");
Personne personne3(102, "jordan", "mike", 'M', adresse);
cout << personne3.getNom() << " " ;
cout << personne3.getPrenom() << " " ;
cout << personne3.getAdresse()->getCodePostal() << endl;

Personne personne4(personne3);
personne3.getAdresse()->setCodePostal("13008");
cout << personne4.getNom() << " " ;
cout << personne4.getPrenom() << " " ;
cout << personne4.getAdresse()->getCodePostal() << endl;
```

Si on copie l'objet `personne` et on modifie l'`adresse`

```
Adresse *adresse =  new Adresse("paradis", "13015", "Marseille");
Personne personne3(102, "jordan", "mike", 'M', adresse);
cout << personne3.getNom() << " " ;
cout << personne3.getPrenom() << " " ;
cout << personne3.getAdresse()->getCodePostal() << endl;

Personne personne4(personne3);
personne3.getAdresse()->setCodePostal("13008");
cout << personne4.getNom() << " " ;
cout << personne4.getPrenom() << " " ;
cout << personne4.getAdresse()->getCodePostal() << endl;
```

Surprise, le résultat est

```
jordan mike 13015
jordan mike 13008
```

Si on copie l'objet `personne` et on modifie l'`adresse`

```
Adresse *adresse =  new Adresse("paradis", "13015", "Marseille");
Personne personne3(102, "jordan", "mike", 'M', adresse);
cout << personne3.getNom() << " " ;
cout << personne3.getPrenom() << " " ;
cout << personne3.getAdresse()->getCodePostal() << endl;

Personne personne4(personne3);
personne3.getAdresse()->setCodePostal("13008");
cout << personne4.getNom() << " " ;
cout << personne4.getPrenom() << " " ;
cout << personne4.getAdresse()->getCodePostal() << endl;
```

Surprise, le résultat est

```
jordan mike 13015
jordan mike 13008
```

`personne3` et `personne4` pointent sur le même objet `adresse`

Quelle solution ?

- Créer un constructeur de copie pour `Adresse`
- Utiliser ce dernier dans un constructeur de copie de `Personne`

Quelle solution ?

- Créer un constructeur de copie pour `Adresse`
- Utiliser ce dernier dans un constructeur de copie de `Personne`

Définissons le prototype de ce constructeur dans `Adresse.h`

```
Adresse(Adresse const& objetACopier);
```

Quelle solution ?

- Créer un constructeur de copie pour `Adresse`
- Utiliser ce dernier dans un constructeur de copie de `Personne`

Définissons le prototype de ce constructeur dans `Adresse.h`

```
Adresse(Adresse const& objetACopier);
```

Implémentons ce constructeur dans `Adresse.cpp`

```
Adresse::Adresse(Adresse const& objetACopier): rue(objetACopier
   .rue), ville(objetACopier.ville), codePostal(objetACopier.
   codePostal)
{
}
```

Déclarons un constructeur de copie dans `Personne.h`

```
Personne(Personne const& objetACopier);
```

C++

Déclarons un constructeur de copie dans `Personne.h`

```
Personne(Personne const& objetACopier);
```

Implémentons ce constructeur de copie dans `Personne.cpp`

```
Personne::Personne(Personne const& objetACopier): num(objetACopier.num)
   , nom(objetACopier.nom), prenom(objetACopier.prenom)
{
    this->setGenre(objetACopier.genre);
    this->adresse = new Adresse(*(objetACopier.adresse));
}
```

C++

Ne changeons rien dans `main`

```
Adresse *adresse =  new Adresse("paradis", "13015", "Marseille");
Personne personne3(102, "jordan", "mike", 'M', adresse);
cout << personne3.getNom() << " " ;
cout << personne3.getPrenom() << " " ;
cout << personne3.getAdresse()->getCodePostal() << endl;

Personne personne4(personne3);
personne3.getAdresse()->setCodePostal("13008");
cout << personne4.getNom() << " " ;
cout << personne4.getPrenom() << " " ;
cout << personne4.getAdresse()->getCodePostal() << endl;
```

C++

Ne changeons rien dans `main`

```
Adresse *adresse =  new Adresse("paradis", "13015", "Marseille");
Personne personne3(102, "jordan", "mike", 'M', adresse);
cout << personne3.getNom() << " " ;
cout << personne3.getPrenom() << " " ;
cout << personne3.getAdresse()->getCodePostal() << endl;

Personne personne4(personne3);
personne3.getAdresse()->setCodePostal("13008");
cout << personne4.getNom() << " " ;
cout << personne4.getPrenom() << " " ;
cout << personne4.getAdresse()->getCodePostal() << endl;
```

En exécutant, le résultat est cette fois-ci correct

```
jordan mike 13015
jordan mike 13015
```

C++

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est **une [sorte de]** `Classe2`

C++

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est **une [sorte de]** `Classe2`

Forme générale

```
class ClasseFille : modeDeVisivility ClasseMère
{
    // code
};
```

C++

Exemple

- Un enseignant a un numéro, un nom, un prénom, un genre et un salaire

C++

Exemple

- Un enseignant a un numéro, un nom, un prénom, un genre et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom, un genre et un niveau

C++

Exemple

- Un enseignant a un numéro, un nom, un prénom, un genre et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom, un genre et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne

C++

Exemple

- Un enseignant a un numéro, un nom, un prénom, un genre et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom, un genre et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom`, `prénom` et `genre`

C++

Exemple

- Un enseignant a un numéro, un nom, un prénom, un genre et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom, un genre et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom`, `prénom` et `genre`
- Donc, on peut peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom`, `prénom` et `genre`

C++

Exemple

- Un enseignant a un numéro, un nom, un prénom, un genre et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom, un genre et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom`, `prénom` et `genre`
- Donc, on peut peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom`, `prénom` et `genre`
- Les classes `Étudiant` et `Enseignant` hériteront donc de la classe `Personne`

C++

Contenu de `Etudiant.h`

```
#ifndef ETUDIANT_H_
#define ETUDIANT_H

#include "Personne.h"

class Etudiant : public Personne
{
public:
    void setNiveau(string niveau);
    string getNiveau() const;

private :
    string niveau;
};

#endif
```

Contenu de `Enseignant.h`

```
#ifndef ENSEIGNANT_H
#define ENSEIGNANT_H

#include "Personne.h"

class Enseignant : public Personne
{
public:
    void setSalaire(int num);
    int getSalaire() const;

private :
    int salaire;
};

#endif
```

C++

Contenu de `Etudiant.h`

```
#ifndef ETUDIANT_H_
#define ETUDIANT_H

#include "Personne.h"

class Etudiant : public Personne
{
public:
    void setNiveau(string niveau);
    string getNiveau() const;

private :
    string niveau;
};

#endif
```

Contenu de `Enseignant.h`

```
#ifndef ENSEIGNANT_H
#define ENSEIGNANT_H

#include "Personne.h"

class Enseignant : public Personne
{
public:
    void setSalaire(int num);
    int getSalaire() const;

private :
    int salaire;
};

#endif
```

Remarque

`class A : public B` permet d'indiquer que la classe `A` hérite de la classe `B`

Contenu de `Etudiant.cpp`

```
#include "Etudiant.h"

void Etudiant::setNiveau(string niveau)
{
    this->niveau = niveau;
}
string Etudiant::getNiveau() const
{
    return niveau;
}
```

Contenu de `Etudiant.cpp`

```
#include "Etudiant.h"

void Etudiant::setNiveau(string niveau)
{
    this->niveau = niveau;
}
string Etudiant::getNiveau() const
{
    return niveau;
}
```

Contenu de `Enseignant.cpp`

```
#include "Enseignant.h"

void Enseignant::setSalaire(int salaire)
{
    this->salaire = salaire;
}
int Enseignant::getSalaire() const
{
    return salaire;
}
```

Pour créer un objet de type `Enseignant` dans le `main`

```
Enseignant enseignant;
enseignant.setNom("wick");
enseignant.setSalaire(1700);
cout << enseignant.getNom() << " " << enseignant.getSalaire()
   << endl;
```

Pour créer un objet de type `Enseignant` dans le `main`

```
Enseignant enseignant;
enseignant.setNom("wick");
enseignant.setSalaire(1700);
cout << enseignant.getNom() << " " << enseignant.getSalaire()
   << endl;
```

Remarque

N'oublions pas `#include "Enseignant.h"`

Pour créer un objet de type `Enseignant` dans le `main`

```
Enseignant enseignant;
enseignant.setNom("wick");
enseignant.setSalaire(1700);
cout << enseignant.getNom() << " " << enseignant.getSalaire()
   << endl;
```

Remarque

N'oublions pas `#include "Enseignant.h"`

En exécutant, le résultat est :

```
wick 1700
```

Pour créer un objet de type `Enseignant` dans le `main`

```
Enseignant enseignant;
enseignant.setNom("wick");
enseignant.setSalaire(1700);
cout << enseignant.getNom() << " " << enseignant.getSalaire()
   << endl;
```

Remarque

N'oublions pas `#include "Enseignant.h"`

En exécutant, le résultat est :

```
wick 1700
```

Refaire la même chose pour `Etudiant`

C++

Pour utiliser un constructeur à plusieurs paramètres, il faut le définir dans `Etudiant.h`

```
Etudiant(int num, string nom, string prenom, char genre, string niveau)
   ;
```

C++

Pour utiliser un constructeur à plusieurs paramètres, il faut le définir dans `Etudiant.h`

```
Etudiant(int num, string nom, string prenom, char genre, string niveau)
   ;
```

Ensuite l'implémenter dans `Etudiant.cpp` **(on utilise le constructeur de la classe mère)**

```
Etudiant::Etudiant(int num, string nom, string prenom, char genre,
   string niveau) : Personne(num, nom, prenom, genre), niveau(niveau)
{
}
```

C++

Pour utiliser un constructeur à plusieurs paramètres, il faut le définir dans `Etudiant.h`

```
Etudiant(int num, string nom, string prenom, char genre, string niveau)
   ;
```

Ensuite l'implémenter dans `Etudiant.cpp` **(on utilise le constructeur de la classe mère)**

```
Etudiant::Etudiant(int num, string nom, string prenom, char genre,
   string niveau) : Personne(num, nom, prenom, genre), niveau(niveau)
{
}
```

Enfin, on peut l'appeler (dans `main` **par exemple)**

```
Etudiant etudiant(100, "green", "steven", 'M', "licence");
cout << etudiant.getNom() << " " << etudiant.getNiveau() << endl;
```

C++

À partir de la classe `Etudiant`

- On ne peut avoir accès direct à un attribut de la classe mère
- C'est-à-dire, on ne peut faire `this->num` car les attributs ont une visibilité `private`
- Pour modifier la valeur d'un attribut privé de la classe mère, il faut
  - soit utiliser les getters/setters
  - soit mettre la visibilité des attributs de la classe mère à `protected`

C++

Particularité du **C++**

- Héritage multiple : une classe peut hériter simultanément de plusieurs autres
- L'héritage multiple est autorisé par le langage C++

C++

Considérons la classe `Doctorant` qui hérite de `Enseignant` et `Etudiant`

```
#ifndef DOCTORANT_H
#define DOCTORANT_H

#include "Enseignant.h"
#include "Etudiant.h"

class Doctorant : public Etudiant, public Enseignant
{
public:
    Doctorant(int numero, string nom, string prenom, char genre
       , string niveau, int salaire, int annee);
    void setAnnee(int anne);
    int getAnnee() const;

private:
    int annee;
};
#endif
```

C++

Le fichier `Doctorant.cpp`

```
#include "Doctorant.h"

Doctorant::Doctorant(int numero, string nom, string prenom, char genre,
    string niveau, int salaire, int annee) :
    Enseignant(numero, nom, prenom, genre, salaire),
    Etudiant(numero, nom, prenom, genre, niveau),
    annee(annee)
{
}

void Doctorant::setAnnee(int annee)
{
    this->annee = annee;
}

int Doctorant::getAnnee() const
{
    return annee;
}
```

C++

Essayons de créer un objet de type `Doctorant` dans le `main`

```
int main()
{
    Doctorant doctorant(100, "wick", "john", 'M', "doctorat", 1700, 1);
    cout << doctorant.getNom() << endl;
    cout << doctorant.getNiveau() << endl;
    cout << doctorant.getSalaire() << endl;
    cout << doctorant.getAnnee() << endl;
}
```

C++

Essayons de créer un objet de type `Doctorant` dans le `main`

```
int main()
{
    Doctorant doctorant(100, "wick", "john", 'M', "doctorat", 1700, 1);
    cout << doctorant.getNom() << endl;
    cout << doctorant.getNiveau() << endl;
    cout << doctorant.getSalaire() << endl;
    cout << doctorant.getAnnee() << endl;
}
```


Erreur à l'exécution : problème d'ambiguïté

- la méthode `getNom()` est définie dans `Etudiant`
- et elle est aussi définie dans `Enseignant`

C++

Pour résoudre ce problème, on peut préciser une base valide (soit `Etudiant` soit `Enseignant`)

```
int main()
{
    Doctorant doctorant(100, "wick", "john", 'M', "doctorat",
       1700, 1);
    cout << doctorant.Etudiant::getNom() << endl;
    cout << doctorant.getNiveau() << endl;
    cout << doctorant.getSalaire() << endl;
    cout << doctorant.getAnnee() << endl;
}
```

C++

Pour résoudre ce problème, on peut préciser une base valide (soit `Etudiant` soit `Enseignant`)

```
int main()
{
    Doctorant doctorant(100, "wick", "john", 'M', "doctorat",
       1700, 1);
    cout << doctorant.Etudiant::getNom() << endl;
    cout << doctorant.getNiveau() << endl;
    cout << doctorant.getSalaire() << endl;
    cout << doctorant.getAnnee() << endl;
}
```


Le résultat est :

```
wick
doctorat
1700
1
```

C++

Revenant au constructeur de `Doctorant` qui est redondant : certaines données sont passées à la fois à `Etudiant()` et à `Enseignant()`

```
Doctorant::Doctorant(int numero, string nom, string prenom, char genre,
    string niveau, int salaire, int annee) :
    Enseignant(numero, nom, prenom, genre, salaire),
    Etudiant(numero, nom, prenom, genre, niveau),
    annee(annee)
{
}
```

C++

Une solution consiste à faire

```
Doctorant::Doctorant(int numero, string nom, string prenom, char genre,
    string niveau, int salaire, int annee) :
     Enseignant(numero, nom, prenom, genre, salaire),
     Etudiant(niveau),
     annee(annee)
{
}
```

C++

Une solution consiste à faire

```
Doctorant::Doctorant(int numero, string nom, string prenom, char genre,
    string niveau, int salaire, int annee) :
     Enseignant(numero, nom, prenom, genre, salaire),
     Etudiant(niveau),
     annee(annee)
{
}
```


N'oublions pas de déclarer et implémenter ce nouveau constructeur avec un paramètre dans `Etudiant`.

C++

Forme générale

```
class ClasseFille : modeDeVisivility ClasseMère
{
    // code
};
```

C++

Forme générale

```
class ClasseFille : modeDeVisivility ClasseMère
{
    // code
};
```

Trois modes de visibilité

- `public`
- `private` (par défaut)
- `protected`

C++

Exemple

```
class A
{
public: int x;
protected: int y;
private: int z;
};

class B : public A
{
    // x est public
    // y est protected
    // z n'est pas accessible de B
};

class C : protected A
{
    // x est protected
    // y est protected
    // z n'est pas accessible de C
};

class D : private A
{
    // x est private
    // y est private
    // z n'est pas accessible de D
};
```

C++

Surcharge (overload)

- Définir dans une classe plusieurs méthodes avec
  - même nom
  - signature différente
- Possible en **C++**
- Meilleur exemple : les constructeurs (qui sont des méthodes particulières) qu'on a définis avec et sans paramètre dans une même classe (évidemment avec le même nom)

C++

Redéfinition (override)

- Définir une méthode dans une classe qui est déjà définie dans la classe mère
- Possible aussi en **C++**
- Deux manières de redéfinir une méthode
  - Proposer une nouvelle implémentation dans la classe fille différente et indépendante de celle de la classe mère (*simple comme si on définit une nouvelle méthode*)
  - Proposer une nouvelle implémentation qui fait référence à celle de la classe mère

C++

Exemple

Redéfinir la méthode `afficherDetails()` dans la classe `Etudiant` : une fois sans faire référence à celle de la classe `Personne` et une autre en faisant référence.

C++

Exemple

Redéfinir la méthode `afficherDetails()` dans la classe `Etudiant` : une fois sans faire référence à celle de la classe `Personne` et une autre en faisant référence.

Il faut commencer par la définir dans `Etudiant.h`

```
void afficherDetails() const;
```

C++

Exemple

Redéfinir la méthode `afficherDetails()` dans la classe `Etudiant` : une fois sans faire référence à celle de la classe `Personne` et une autre en faisant référence.

Il faut commencer par la définir dans `Etudiant.h`

```
void afficherDetails() const;
```

Ensuite l'implémenter dans `Etudiant.cpp`

```
void Etudiant::afficherDetails() const
{
    cout << num << " "<< nom << " " ;
    cout << prenom << " "<< genre << endl;
    cout << niveau << endl;
}
```

C++

Enfin, on peut l'appeler (dans `main`)

```
Etudiant etudiant(104, "green" , "steven", 'M', "licence");
etudiant.afficherDetails();

/* affiche
104 steven green M
licence
*/
```

On peut aussi utiliser l'implémentation de la classe `Personne`

```
void Etudiant::afficherDetails() const
{
    Personne::afficherDetails();
    cout << " " << niveau << endl;
}
```

On peut aussi utiliser l'implémentation de la classe `Personne`

```
void Etudiant::afficherDetails() const
{
    Personne::afficherDetails();
    cout << " " << niveau << endl;
}
```

En testant, le résultat est le même

```
Etudiant etudiant(104, "green", "steven", 'M', "licence");
etudiant.afficherDetails();

/* affiche
104 steven green M
licence
*/
```

On peut aussi utiliser l'implémentation de la classe `Personne`

```
void Etudiant::afficherDetails() const
{
    Personne::afficherDetails();
    cout << " " << niveau << endl;
}
```

En testant, le résultat est le même

```
Etudiant etudiant(104, "green", "steven", 'M', "licence");
etudiant.afficherDetails();

/* affiche
104 steven green M
licence
*/
```

Refaire la même chose pour `Enseignant`

C++

Dans `main.cpp`, créons une fonction `afficher(Personne p)` qui prend en paramètre un objet `Personne` et ensuite appelle sa méthode `afficherDetails()`

```
void afficher(Personne const&  p)
{
    p.afficherDetails();
}
```

C++

Dans `main.cpp`, créons une fonction `afficher(Personne p)` qui prend en paramètre un objet `Personne` et ensuite appelle sa méthode `afficherDetails()`

```
void afficher(Personne const&  p)
{
    p.afficherDetails();
}
```

Créons ensuite trois objets : un premier de type `Personne`, un deuxième de type `Enseignant` et un dernier de type `Etudiant` et appelons chaque fois la fonction `afficher()`

```
Personne personne(101, "dalton", "jack", 'M');
afficher(personne);
Enseignant enseignant(100, "wick", "john", 'M', 1700);
afficher(enseignant);
Etudiant etudiant(104, "green", "steven", 'M', "licence");
afficher(etudiant);
```

C++

Résultat attendu

```
101 jack dalton M
100 john wick M
 1700
104 steven green M
 licence
```

C++

Résultat attendu

```
101 jack dalton M
100 john wick M
 1700
104 steven green M
 licence
```

Résultat obtenu

```
101 jack dalton M
100 john wick M
104 steven green M
```

C++

Résultat attendu

```
101 jack dalton M
100 john wick M
 1700
104 steven green M
 licence
```

Résultat obtenu

```
101 jack dalton M
100 john wick M
104 steven green M
```

C’est ce qu’on appelle les liens statiques.

C++

Solution : méthodes virtuelles

- Déclarer ces méthodes comme virtuelle, et
- Utiliser une référence dans la fonction `afficher()`

C++

Dans `Personne.h`, déclarer `afficherDetails()` comme méthode virtuelle

```
virtual void afficherDetails() const;
```

C++

Dans `Personne.h`, déclarer `afficherDetails()` comme méthode virtuelle

```
virtual void afficherDetails() const;
```

Remarques

- Par définition d'héritage, les méthodes `afficherDetails()` redéfinies dans `Etudiant` et `Enseignant` sont aussi virtuelles
- Il est possible de le préciser pour ne pas l'oublier

C++

En exécutant maintenant, le résultat est

```
101 jack dalton M
100 john wick M
 1700
104 steven green M
 licence
```

C++

Hypothèse et problématique

- On voudrait définir une méthode `getStatutAcademique()` qui retourne
  - le grade d'un enseignant en se basant sur son salaire
  - le niveau d'expérience d'un étudiant selon son niveau
- Pour un objet de type `Personne`, cette méthode ne fait rien

C++

Hypothèse et problématique

- On voudrait définir une méthode `getStatutAcademique()` qui retourne
  - le grade d'un enseignant en se basant sur son salaire
  - le niveau d'expérience d'un étudiant selon son niveau
- Pour un objet de type `Personne`, cette méthode ne fait rien

Solution : méthode virtuelle pure

- une méthode virtuelle sans code dans la classe mère
- elle est seulement initialisée à zéro

C++

Dans `Personne.h`, on déclare cette nouvelle méthode

```
virtual string getStatutAcademique() const = 0;
```

C++

Dans `Personne.h`, on déclare cette nouvelle méthode

```
virtual string getStatutAcademique() const = 0;
```

Aucune implémentation à faire dans `Personne.cpp`

C++

Dans `Etudiant.h` **et** `Enseignant.h`**, déclarer** `getStatutAcademique()`

```
virtual string getStatutAcademique() const;
```

C++

Dans `Etudiant.h` **et** `Enseignant.h`**, déclarer** `getStatutAcademique()`

```
virtual string getStatutAcademique() const;
```

Et voici l'implémentation à ajouter dans `Etudiant.cpp`

```
string Etudiant::getStatutAcademique() const
{
    return niveau == "licence" ? "débutant" : "junior";
}
```

C++

Dans `Etudiant.h` **et** `Enseignant.h`**, déclarer** `getStatutAcademique()`

```
virtual string getStatutAcademique() const;
```

Et voici l'implémentation à ajouter dans `Etudiant.cpp`

```
string Etudiant::getStatutAcademique() const
{
    return niveau == "licence" ? "débutant" : "junior";
}
```

Et voici l'implémentation à ajouter dans `Enseignant.cpp`

```
string Enseignant::getStatutAcademique() const
{
    return salaire > 3000 ? "MCF" : "PR";
}
```

C++

Commençons par tester pour `Etudiant`

```
Etudiant etudiant(104, "green", "steven", 'M', "licence");
cout << etudiant.getStatutAcademique() << endl;
// affiche débutant
```

C++

Commençons par tester pour `Etudiant`

```
Etudiant etudiant(104, "green", "steven", 'M', "licence");
cout << etudiant.getStatutAcademique() << endl;
// affiche débutant
```

Ensuite pour `Enseignant`

```
Enseignant enseignant(100, "wick", "john", 'M', 1700);
cout << enseignant.getStatutAcademique() << endl;
// affiche MCF
```

C++

Commençons par tester pour `Etudiant`

```
Etudiant etudiant(104, "green", "steven", 'M', "licence");
cout << etudiant.getStatutAcademique() << endl;
// affiche débutant
```

Ensuite pour `Enseignant`

```
Enseignant enseignant(100, "wick", "john", 'M', 1700);
cout << enseignant.getStatutAcademique() << endl;
// affiche MCF
```

Pour `Personne` **: Erreur (la classe** `Personne` **est maintenant abstraite.)**

```
Personne personne(100, "wick", "john", 'M');
cout << personne.getStatutAcademique() << endl;
```

C++

Classe abstraite

- C'est une classe contenant au moins une méthode virtuelle pure
- C'est une classe qu'on ne peut instancier
- Les classes filles d'une classe abstraite doivent implémenter ses méthodes virtuelles pures
- Les classes filles d'une classe abstraite peuvent implémenter ses méthodes virtuelles

C++

Classe abstraite

- C'est une classe contenant au moins une méthode virtuelle pure
- C'est une classe qu'on ne peut instancier
- Les classes filles d'une classe abstraite doivent implémenter ses méthodes virtuelles pures
- Les classes filles d'une classe abstraite peuvent implémenter ses méthodes virtuelles

Pas de mot-clé `abstract` en **C++** comme en **Java**, **C#**...

C++

Interface

- Comme une classe avec uniquement des méthodes virtuelles pures
- Pas de mot clé `interface` pour la création d'interface en **C++**

C++

Opérateur

- **C++** nous offre la possibilité de surcharger les opérateurs
  - arithmétiques : +, -, *, /, %, =, +=, ++...
  - ou de comparaison : ==, !=, >=, <=...
- Il est possible de définir comment utiliser ces opérateurs entre deux objets

C++

Considérons la classe `Note` suivante définie dans `Note.h`

```
#ifndef NOTE_H
#define NOTE_H

class Note {

public:
    Note(float valeur = 0, int coefficient = 0);
    float getValeur() const;
    int getCoefficient() const;
    void setValeur(float valeur);
    void setCoefficient(int coefficient);

private:
    float valeur;
    int coefficient;
};

#endif
```

L'implémentation dans `Note.cpp`

```
#include "Note.h"

Note::Note(float valeur, int coefficient): valeur(valeur), coefficient(
   coefficient)
{
}
float Note::getValeur() const
{
    return valeur;
}
int Note::getCoefficient() const
{
    return coefficient;
}
void Note::setValeur(float valeur)
{
    this->valeur =  valeur;
}
void Note::setCoefficient(int coefficient)
{
    this->coefficient = coefficient;
}
```

C++

Nous voudrons surcharger l'opérateur `==` pour pouvoir comparer deux coefficients de deux objets de type `Note` comme le montre l'exemple suivant

```
#include <iostream>

#include "Note.h"

using namespace std;

int main()
{
    Note note1(12, 2);
    Note note2(15, 3);
    Note note3(12, 3);
    string b1 = note1 == note2 ? "note1 == note2" : "note1 != note2";
    string b2 = note2 == note3 ? "note2 == note3" : "note2 != note3";
    cout << b1 << endl;
    cout << b2 << endl;
    return 0;
}
```

C++

Pour cela, commençons par définir le prototype de notre opérateur dans `Note.h`

```
#ifndef NOTE_H
#define NOTE_H

class Note {
public:
    Note(float valeur = 0, int coefficient = 0);
    float getValeur() const;
    int getCoefficient() const;
    void setValeur(float valeur);
    void setCoefficient(int coefficient);
    bool operator == (Note const& obj1);
private:
    float valeur;
    int coefficient;
};

#endif
```

C++

Pour cela, commençons par définir le prototype de notre opérateur dans `Note.h`

```
#ifndef NOTE_H
#define NOTE_H

class Note {
public:
    Note(float valeur = 0, int coefficient = 0);
    float getValeur() const;
    int getCoefficient() const;
    void setValeur(float valeur);
    void setCoefficient(int coefficient);
    bool operator == (Note const& obj1);
private:
    float valeur;
    int coefficient;
};

#endif
```

Implémentons le prototype précédent dans `Note.h`

```
bool Note::operator == (Note const& obj1)
{
    return coefficient == obj1.getCoefficient();
}
```

C++

En testant le code précédent, le résultat est

```
note1 != note2
note2 == note3
```

C++

Exercice

- Étant donnés deux objets `note1` et `note2` de type `Note`
- Implémenter l'opérateur `+` qui permettra de retourner un objet de type `Note` dont la valeur = `valeur1 + valeur2` et le coefficient = coefficient1 (ou coefficient12) si les deux notes sont égales, -1 pour la valeur et -1 pour le coefficient sinon.
- `valeur1` et `coefficient1` (respectivement `valeur2` et `coefficient2`) désignent les attributs de `note1` (resp. `note2`)

C++

Commençons par déclarer le nouvel opérateur dans `Note.h`

```
#ifndef NOTE_H
#define NOTE_H

class Note {
public:
    Note(float valeur = 0, int coefficient = 0);
    float getValeur() const;
    int getCoefficient() const;
    void setValeur(float valeur);
    void setCoefficient(int coefficient);
    bool operator == (Note const& obj1);
    Note operator + (Note const& obj1);
private:
    float valeur;
    int coefficient;
};

#endif
```

C++

Implémentons cet opérateur dans `Note.cpp`

```
Note Note::operator+(Note const &obj1)
{
    Note note(-1, -1);
    if (*this == obj1)
    {
        note.setValeur(obj1.getValeur() + valeur);
        note.setCoefficient(coefficient);
    }
    return note;
}
```

Pour tester

```
#include <iostream>
#include "Note.h"

using namespace std;

int main()
{
    Note note1(12, 2);
    Note note2(15, 3);
    Note note3(12, 3);
    Note note4 = note1 + note2;
    Note note5 = note2 + note3;
    cout << note4.getValeur() << " " << note4.getCoefficient() << endl;
    cout << note5.getValeur() << " " << note5.getCoefficient() << endl;
    return 0;
}
```

Pour tester

```
#include <iostream>
#include "Note.h"

using namespace std;

int main()
{
    Note note1(12, 2);
    Note note2(15, 3);
    Note note3(12, 3);
    Note note4 = note1 + note2;
    Note note5 = note2 + note3;
    cout << note4.getValeur() << " " << note4.getCoefficient() << endl;
    cout << note5.getValeur() << " " << note5.getCoefficient() << endl;
    return 0;
}
```

Le résultat est

```
-1 -1
27 3
```

C++

Définir l'opérateur `+=` dans `Personne` qui nous permettra d'exécuter le code suivant

```
int main()
{
    Personne personne4(102, "jordan", "mike", 'M');
    Adresse a1 ("paradis", "13015", "Marseille");
    personne4 += a1;
    Adresse a2 ("pasteur", "69200", "Lyon");
    personne4 += a2;
    for(int i =0; i < 2; i++)
    {
        cout << personne4.getAdresses()[i].getCodePostal() << endl;
    }
}
```

Résultat attendu

```
13015
69200
```

C++

Commençons par définir un attribut privé indice dans `Personne.h`

```
private:
    int indice = 0;
```

C++

Commençons par définir un attribut privé indice dans `Personne.h`

```
private:
    int indice = 0;
```

Définissons l'opérateur dans `Personne.h`

```
void operator += (Adresse const& adresse);
```

C++

Commençons par définir un attribut privé indice dans `Personne.h`

```
private:
    int indice = 0;
```

Définissons l'opérateur dans `Personne.h`

```
void operator += (Adresse const& adresse);
```

Implémentons cet opérateur dans `Personne.cpp`

```
void Personne::operator += (Adresse const& adresse)
{
    adresses[indice] = adresse;
    indice++;
}
```

C++

Énumération

- Ensemble de constante
- Commençant par défaut de 0

C++

Déclarons une énumération (dans `main.cpp` et avant la fonction `main()`)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN
   , JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE,
   DECEMBRE };
```

C++

Déclarons une énumération (dans `main.cpp` et avant la fonction `main()`)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN
   , JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE,
   DECEMBRE };
```

Pour récupérer un élément

```
cout << JANVIER << endl;
// affiche 0
```

C++

Déclarons une énumération (dans `main.cpp` et avant la fonction `main()`)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN
   , JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE,
   DECEMBRE };
```

Pour récupérer un élément

```
cout << JANVIER << endl;
// affiche 0
```

Pour éviter un éventuel conflit avec une autre énumération, on peut faire

```
cout << mois::JANVIER << endl;
```

C++

Pour modifier l'indice du premier élément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL, MAI,
   JUIN, JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE
   , DECEMBRE };
```

C++

Pour modifier l'indice du premier élément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL, MAI,
   JUIN, JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE
   , DECEMBRE };
```

En affichant maintenant, le résultat est

```
cout << mois::JANVIER << endl;
// affiche 1
```

C++

Pour modifier l'indice du premier élément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL, MAI,
   JUIN, JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE
   , DECEMBRE };
```

En affichant maintenant, le résultat est

```
cout << mois::JANVIER << endl;
// affiche 1
```

Ceci décalera les indices suivants aussi

```
cout << mois::MARS << endl;
// affiche 3
```

C++

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN, JUILLET,
    AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

C++

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN, JUILLET,
    AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

En affichant, le résultat est

```
cout << mois::MARS << endl;
// affiche 3

cout << mois::JUIN << endl;
// affiche 12

cout << mois::DECEMBRE << endl;
// affiche 3
```

C++

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN, JUILLET,
    AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

En affichant, le résultat est

```
cout << mois::MARS << endl;
// affiche 3

cout << mois::JUIN << endl;
// affiche 12

cout << mois::DECEMBRE << endl;
// affiche 3
```

Ceci est une erreur, on ne peut modifier une constante

```
mois::JANVIER = 3;
```

C++

struct

- Une classe dont la visibilité par défaut de ses membres (attributs et méthodes) est `public`
- Se déclare avec le mot-clé `struct`
- Le mode de visibilité par défaut de l'héritage d'une `struct` qui hérite d'une classe est `public`

C++

Déclarons une `struct` **appelée** `Structure` **dans** `Structure.h`

```
#ifndef STRUCTURE_H
#define STRUCTURE_H

#include <iostream>

using namespace std;

struct Structure
{
    int num;
    string nom;
    char genre;
    void afficherDetails();
};

#endif
```

C++

Implémentons la méthode `afficherDetails()` **dans** `Structure.cpp`

```
#include "Structure.h"

void Structure::afficherDetails()
{
    std::cout << num << " " << nom << " " << genre << std::endl;
}
```

C++

Implémentons la méthode `afficherDetails()` dans `Structure.cpp`

```
#include "Structure.h"

void Structure::afficherDetails()
{
    std::cout << num << " " << nom << " " << genre << std::endl;
}
```

Pour tester

```
int main()
{
    Structure first;
    first.num = 100;
    first.nom = "wick";
    first.genre = 'M';
    first.afficherDetails();
    // affiche 100 wick M
}
```

C++

Functor

- Une classe ou une `struct` qui définit la méthode `operator`
- Ses objets pouvant être utilisés comme des fonctions.
- Très utilisés dans **STL** (comme `transform`).

C++

Déclarons le `functor` suivant

```
#ifndef INCREMENTER_H
#define INCREMENTER_H

class Incrementer
{
public:
    Incrementer(int num = 0) : num(num) {}
    ~Incrementer() {}
    int operator()(int val) const
    {
        return num + val;
    }

private:
    int num;
};

#endif
```

C++

Pour l'utiliser

```
#include <iostream>

#include "Incrementer.h"

using namespace std;

int main()
{
    Incrementer i(10);
    cout << i(5) << endl;
    // affiche 15

    cout << i(3) << endl;
    // affiche 13
}
```

C++

Les functors pourront aussi être utilisées dans certaines fonctions de la librairie `Algorithms` de la STL

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "Incrementer.h"

using namespace std;

int main()
{
    vector<int> vecteur{2, 5, 8, 4};
    transform(vecteur.begin(), vecteur.end(), vecteur.begin(),
       Incrementer(3));

    for (auto elt : vecteur)
    {
        cout << elt << " ";
    }
    // affiche 5 8 11 7
}
```