Cours P.O.O par le C++

Proposé par Cyrille MBIA

Enseigné par Christian YESIBI

Introduction

Pour développer applications de traitement de données performantes (que ce soit des applications de base de données, des middleware, des applications de reporting, etc…), vous avez besoin d'une boîte à outils plus fournie que celle offerte par la programmation informatique classique ou la programmation fonctionnelle, car lorsqu'on veut développer des applications qui combinent les exigences d'un processus métier particulier aux contraintes technologiques du traitement de données, on a besoin de concepts, d'outils et de méthodologies qui n'existent que dans le paradigme orienté objet.

Définition et origine du paradigme orienté objet

La programmation orientée objet (**POO**) est un modèle de programmation informatique qui met en œuvre une conception basée sur les **objets**. Elle se différencie de la programmation **procédurale**, qui est basée sur l'utilisation de **procédures**, et de la **programmation fonctionnelle**, qui elle, repose entièrement sur le **concept de fonction**.

Les premiers concepts de la programmation orientée objet remontent aux années **1970** avec les les langages **Simula** et **Smalltalk**.

Définition et origine du paradigme orienté objet

De nos jours, de très nombreux langages permettent d'utiliser les principes de la POO dans des domaines variés : C++, Java et C# bien sûr, mais aussi PHP (à partir de la version 5), Python, etc… La Programmation orientée objet a entraînée un changement profond dans la manière de concevoir l'architecture des applications informatiques. Au cours de la modélisation d'une application on réfléchit désormais en termes d'objets : c'est la modélisation orientée objet (MOO), dont le principal support est le langage de modélisation UML.

Les langages orientés objet

Comme nous l'avons mentionné plus haut, l'effervescence des langages objet commence en 1980 et atteint son paroxysme dans les années 1990. Avant d'entrer dans les caractéristiques de ces langages, il est bon de rappeler les principaux avantages du paradigme orienté objet :

La modularité : les objets forment des modules compacts regroupant des données et un ensemble d'opérations.

Les langages orientés objet

L’abstraction : les objets de la POO sont proches de celles du monde réel. Les concepts utilisés sont donc proches des abstractions familières que nous exploitons.

Productivité et ré-utilisabilité : plus l’application est complexe et plus l’approche POO est intéressante. Le niveau de ré-utilisabilité est supérieur à la programmation procédurale.

Sûreté : l’encapsulation et le typage des classes offrent une certaine robustesse aux applications.

Les langages orientés objet

On distingue plusieurs catégories de langage orienté objet, tout dépend du degré d'utilisation des objets et du niveau d'intégration du langage au paradigme orienté objet :

Les langages dits de **POO** « **purs** », où tout est traité comme un objet, depuis les primitives telles que les caractères et la ponctuation, jusqu'aux classes, prototypes, blocs, modules. Ils ont été conçus spécifiquement pour faciliter, voire imposer, les méthodes orientés objet. Ex: Ruby, Scala, Smalltalk, Eiffel, Emerald, JADE, Self, Raku.

Les langages conçus principalement pour la programmation OO, mais avec quelques éléments procéduraux. Ex : Java, Python, **C++**, C#, Delphi/Object Pascal, VB.NET.

Les langages orientés objet

Les langages qui sont historiquement des langages procéduraux, mais qui ont été étendus avec certaines caractéristiques orientées objets. Ex : PHP, Perl, Visual Basic (dérivé de BASIC), MATLAB, C++, C#, COBOL 2002, Fortran 2003, ABAP, Ada 95, Pascal.

Les langages possédant la plupart des caractéristiques des objets (classes, méthodes, héritage), mais sous une forme nettement originale. Ex: Oberon (Oberon-1 ou Oberon-2).

Les langages avec support de types de données abstraits qui peuvent être utilisés pour ressembler à la programmation OO, mais sans toutes les caractéristiques de l'orienté objet Ex : JavaScript, Lua, Modula-2, CLU.

Les concepts fondamentaux de la programmation orientée objet

La programmation orientée objet repose sur 5 concepts fondamentaux à savoir :

La classe

L’objet

L’encapsulation

L’héritage

Le polymorphisme

Nous allons voir distinctement chacun de ces concepts.

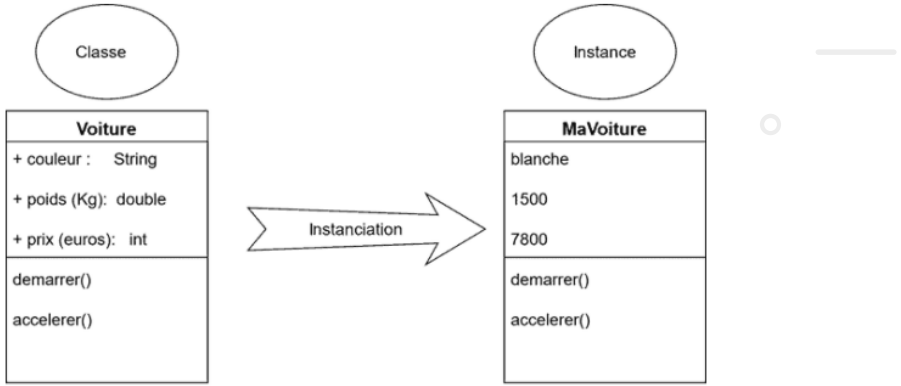
Le concept de Classe

Le premier concept fondamental de l'orienté objet est la classe. Une classe est une structure abstraite qui décrit des objets du monde réel sous deux angles : ses propriétés (ses caractéristiques) et ses méthodes (les actions qu'elle peut effectuer ou son comportement).

Par exemple, la Classe **Vehicule** représente un véhicule, **couleur** est l'une de ses propriétés et **accélérer/freiner** sont deux de ses méthodes. Un autre exemple : on peut représenter en programmation orientée-objet les employés sous forme de classe ; auquel cas, la classe Employés représente tous les employés qui peuvent avoir pour propriétés un nom, un prénom, une adresse et une date de naissance ; les opérations qui peuvent être effectuées sur les employés peuvent être le changement de salaire, la prise de congé, la prise de retraite, etc.

Le concept de Classe

La classe est finalement une sorte de moule, de modèle. Toutes les instances de classe s’appellent des objets. Les objets sont construits à partir de la classe, par un processus appelé instanciation. De ce fait, tout objet est une instance de classe.



Le concept de Classe

L’instanciation d’une classe fait appel à 3 méthodes spéciales dont la compréhension est très importante :

Le constructeur : on distingue trois types de constructeurs

Le constructeur par défaut appelé par défaut lors de la création d’un objet (offert par défaut lors de la compilation s’il n’y a pas de constructeur déclaré),

Le constructeur par recopie (ou constructeur de copie) a un unique argument du même type que l’objet à créer (généralement sous forme de référence constante) et il recopie les attributs depuis l’objet passé en argument sur l’objet à créer.

Le constructeur paramétrique appelé si la signature correspond à celle du constructeur.

Le concept de Classe

Les accesseurs (get) et les **mutateurs (set)** : ces méthodes spéciales permettent d'appeler les propriétés et modifier les propriétés d'une classe depuis l'extérieur, un peu comme une API. C'est grâce à elles que l'extérieur peut « appeler » les fonctionnalités de la classe. Les accesseurs permettent de récupérer la valeur des propriétés d'une instance de classe depuis l'extérieur sans y accéder directement. Ce faisant, ils sécurisent l'attribut en restreignant sa modification. Les mutateurs quant à eux permettent de modifier la valeur des propriétés tout en vérifiant que la valeur que l'on veut donner à l'attribut respecte les contraintes sémantiques qui ont été imposées à la classe.

Le destructeur : est une méthode qui met fin à la vie d'une instance de classe. Il peut être appelé à la suppression de l'objet, explicitement ou implicitement.

Le concept d’objet

Le second concept le plus important en programmation objet c’est justement, l’objet. Comme nous vous l’avons dit tout à l’heure, un objet est une instance de classe. Pour faire le parallèle avec le monde réel, l’objet c’est un peu comme une maison bâtit sur la base d’un plan particulier. Tant que les architectes se réfèrent à ce plan, ils produiront toujours les mêmes maisons.

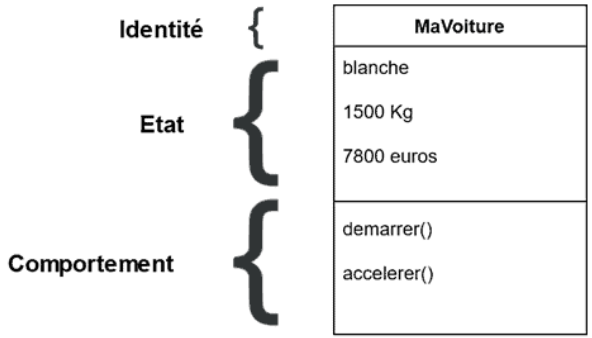
Techniquement, un objet est caractérisé par 3 choses :

une identité : l’identité doit permettre d’identifier sans ambiguïté l’objet (adresse/ référence ou nom)

Le concept d’objet

des états : chaque objet a une valeur par défaut (lorsqu’elle est indiqué à l’instanciation) pour chacune de ses propriétés. On appelle ces valeurs, des états de l’objet.

des méthodes : chaque objet est capable d’exécuter les actions ou le comportement défini dans la classe. Ces actions sont traduites en **POO** concrètement sous forme de méthodes. Les actions possibles sur un objet sont déclenchées par des appels de ces méthodes ou par des messages envoyées par d’autres objets.



Le concept d’encapsulation

Le troisième concept de la programmation orientée objet c’est **l’encapsulation**.

les propriétés des objets ne peuvent être accédées que par ses méthodes. Ainsi, la classe encapsule à la fois les attributs et les méthodes qui permettent de manipuler les objets indépendamment de leurs états.

L’encapsulation permet de restreindre l’accès direct aux états et empêche la modification de l’objet hors de ses méthodes. Par exemple, si vous avez une classe Voiture et que vous voulez définir la valeur de sa propriété couleur à bleu, il vous faut passer par une méthode par exemple **definirCouleur**, implémentée par le développeur de la classe. Cette méthode peut restreindre les différentes valeurs de couleur.

Le concept d’encapsulation

Ainsi, l’encapsulation est un mécanisme qui empêche donc de modifier ou d’accéder aux objets par un autre moyen que les méthodes proposées, et de ce fait, permet de garantir l’intégrité des objets.

Le concept d’héritage

L’héritage est le quatrième concept clé de la programmation objet. C’est un concept en POO qui désigne le fait qu’une classe peut hériter des caractéristiques (attributs et méthodes) d’une autre classe.

Les objets de classes peuvent hériter des propriétés d’une classe parent. Par exemple on peut définir une classe Employé et une classe Manager qui est une classe spécialisée de Employé, qui hérite de ses propriétés.

Soient deux classes A et B. Relation d’héritage : Classe B « étend la » Classe A

A est la super-classe ou classe mère/parent de B

B est la sous-classe ou classe fille de A

Le concept de polymorphisme

Le dernier concept clé de la programmation orientée objet c'est le **polymorphisme**. Un langage orienté objet est dit polymorphique, s'il offre la possibilité de percevoir un objet en tant qu'instance de différentes classes selon la situation. C++ par exemple, est un langage polymorphique. Nous allons expliquer ce concept avec des exemples concrets.

Chapitre 1

Les Classes & Objets

Classe C++

Une classe est un modèle pour l'objet.

Nous pouvons considérer une classe comme un croquis (prototype) d'une maison. Il contient tous les détails sur les sols, les portes, les fenêtres, etc. Sur la base de ces descriptions, nous construisons la maison. La maison est l'objet.

Créer une classe

Une classe est définie en C++ à l'aide du mot-clé **class** suivi du nom de la classe.

Le corps de la classe est défini à l'intérieur des accolades et terminé par un point-virgule à la fin.

```
class className {
   // some data
   // some functions
};
```

Classe C++

Par exemple,

Ici, nous avons défini une classe nommée **Room**.

```
class Room {
    public:
        double length;
        double breadth;
        double height;

        double calculateArea(){
            return length * breadth;
        }

        double calculateVolume(){
            return length * breadth * height;
        }

};
```

Les variables **length**, **breadth** et **height** déclarées à l'intérieur de la classe sont appelées membres de données.

Et, les fonctions **calculateArea()** et **calculateVolume()** sont appelées fonctions membres d'une classe ou méthodes.

Objets C++

Lorsqu'une classe est définie, seule la spécification de l'objet est définie ; aucune mémoire ou stockage n'est alloué.

Pour utiliser les données et les fonctions d'accès définies dans la classe, nous devons créer des objets.

Syntaxe pour définir un objet en C++

className objectVariableName;

Nous pouvons créer des objets de classe **Room** (définis dans l'exemple ci-dessus) comme suit :

```
// sample function
void sampleFunction() {
    // create objects
    Room room1, room2;
}

int main(){
    // create objects
    Room room3, room4;
}
```

Objets C++

Ici, deux objets **room1** et **room2** de la classe **Room** sont créés dans **sampleFunction()**. De même, les objets **room3** et **room4** sont créés dans **main()**.

Comme nous pouvons le voir, nous pouvons créer des objets d'une classe dans n'importe quelle fonction du programme. Nous pouvons également créer des objets d'une classe au sein de la classe elle-même, ou dans d'autres classes.

De plus, nous pouvons créer autant d'objets que nous le souhaitons à partir d'une même classe.

Accès aux données membres et fonctions membres

Nous pouvons accéder aux données membres et aux fonctions membres d'une classe en utilisant un opérateur (point). Par exemple, **room2.calculateArea();**

Objets C++

Cela appellera la fonction **calculateArea()** à l'intérieur de la classe **Room** pour l'objet **room2**.

De même, les membres de données sont accessibles en tant que : **room1.length = 5.5;**

Dans ce cas, il initialise la variable **length** de **room1** à **5.5.**

Exemple 1 : Objet et classe en programmation C++ (diapo suivante)

Objets C++

```cpp
// Program to illustrate the working of
// objects and class in C++ Programming

#include <iostream>
using namespace std;

// create a class
class Room {

   public:
    double length;
    double breadth;
    double height;

    double calculateArea() {
        return length * breadth;
    }

    double calculateVolume() {
        return length * breadth * height;
    }
};

int main() {

    // create object of Room class
    Room room1;

    // assign values to data members
    room1.length = 42.5;
    room1.breadth = 30.8;
    room1.height = 19.2;

    // calculate and display the area and volume of the room
    cout << "Area of Room =   " << room1.calculateArea() << endl;
    cout << "Volume of Room =   " << room1.calculateVolume() << endl;

    return 0;
}
```

Objets C++

Dans ce programme, nous avons utilisé la classe **Room** et son objet **room1** pour calculer la surface et le volume d'une pièce.

Dans **main()**, nous avons attribué les valeurs de **length**, de **breadth** et de **height** avec le code :

```
room1.length = 42.5;
room1.breadth = 30.8;
room1.height = 19.2;
```

Objets C++

Nous avons ensuite appelé les fonctions **calculateArea()** et **calculateVolume()** pour effectuer les calculs nécessaires.

Notez l'utilisation du mot-clé **public** dans le programme. Cela signifie que les membres sont **publics** et peuvent être consultés n'importe où à partir du programme.

Selon nos besoins, nous pouvons également créer des membres privés en utilisant le mot-clé **private**. Les membres privés d'une classe ne sont accessibles qu'à partir de la classe. Par exemple,

```
class Test {

private:
    int a;
    void function1() { }

public:
    int b;
    void function2() { }
}
```

Objets C++

Ici, **a** et **function1()** sont privés. Ils ne sont donc pas accessibles depuis l'extérieur de la classe.

En revanche, **b** et **function2()** sont accessibles de n'importe où dans le programme.

Exemple 2 : Utilisation de **public** et **private** dans la classe C++

```
// Program to illustrate the working of
// public and private in C++ Class

#include <iostream>
using namespace std;

class Room {

   private:
    double length;
    double breadth;
    double height;

   public:

    // function to initialize private variables
    void initData(double len, double brth, double hgt) {
        length = len;
        breadth = brth;
        height = hgt;
    }

    double calculateArea() {
        return length * breadth;
    }

    double calculateVolume() {
        return length * breadth * height;
    }
};

int main() {

    // create object of Room class
    Room room1;

    // pass the values of private variables as arguments
    room1.initData(42.5, 30.8, 19.2);

    cout << "Area of Room =   " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Objets C++

L'exemple ci-dessus est presque identique au premier exemple, sauf que les variables de classe sont maintenant privées.

Puisque les variables sont maintenant privées, nous ne pouvons pas y accéder directement depuis **main()**. Par conséquent, l'utilisation du code suivant serait invalide :

```
// invalid code
obj.length = 42.5;
obj.breadth = 30.8;
obj.height = 19.2;
```

Au lieu de cela, nous utilisons la fonction publique **initData()** pour initialiser les variables privées via les paramètres de fonction **double len**, **double brth** et **double hgt**.

Constructeurs

Un constructeur est un type spécial de fonction membre qui est appelé automatiquement lorsqu'un objet est créé.

En C++, un constructeur a le même nom que celui de la classe et il n'a pas de type de retour. Par exemple,

```
class  Wall {
  public:
    // create a constructor
    Wall() {
      // code
    }
};
```

Ici, la fonction **Wall()** est un constructeur de la classe **Wall**. Notez que le constructeur a le même nom que la classe, n'a pas de type de retour, et est public

Constructeurs

Constructeur par défaut

Un constructeur sans paramètres est appelé constructeur par défaut. Dans l'exemple ci-dessus, **Wall()** est un constructeur par défaut.

```cpp
// C++ program to demonstrate the use of default constructor

#include <iostream>
using namespace std;

// declare a class
class  Wall {
  private:
    double length;

  public:
    // default constructor to initialize variable
    Wall() {
      length = 5.5;
      cout << "Creating a wall." << endl;
      cout << "Length = " << length << endl;
    }
};

int main() {
  Wall wall1;
  return 0;
}
```

Constructeurs

Ici, lorsque l'objet **wall1** est créé, le constructeur **Wall()** est appelé. Cela définit la variable de **length** de l'objet à 5,5.

Remarque : si nous n'avons pas défini de constructeur dans notre classe, le compilateur C++ créera automatiquement un constructeur par défaut avec un code vide et aucun paramètre.

Constructeur paramétré

En C++, un constructeur avec des paramètres est appelé constructeur paramétré. Il s'agit de la méthode préférée pour initialiser les données des membres.

Constructeurs

Exemple 2 : constructeur paramétré C++

```cpp
// C++ program to calculate the area of a wall

#include <iostream>
using namespace std;

// declare a class
class Wall {
  private:
    double length;
    double height;

  public:
    // parameterized constructor to initialize variables
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }

    double calculateArea() {
      return length * height;
    }
};

int main() {
  // create object and initialize data members
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);

  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

Constructeurs

Ici, nous avons créé un constructeur paramétré **Wall()** qui a 2 paramètres : **double len** et **double hgt**. Les valeurs contenues dans ces paramètres sont utilisées pour initialiser les variables membres **length** et **height**.

Lorsque nous créons un objet de la classe **Wall**, nous passons les valeurs des variables membres en arguments. Le code pour cela est :

Wall wall1(10.5, 8.6);

Wall wall2(8.5, 6.3);

Avec les variables membres ainsi initialisées, nous pouvons maintenant calculer l'aire du mur avec la fonction **calculateArea().**

Constructeurs

Copier le constructeur

Le constructeur de copie en C++ est utilisé pour copier les données d'un objet vers un autre.

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

// declare a class
class Wall {
  private:
    double length;
    double height;

  public:

    // initialize variables with parameterized constructor
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }

    // copy constructor with a Wall object as parameter
    // copies data of the obj parameter
    Wall(Wall &obj) {
      length = obj.length;
      height = obj.height;
    }

    double calculateArea() {
      return length * height;
    }
};
```

```cpp
    double calculateArea() {
      return length * height;
    }
};

int main() {
  // create an object of Wall class
  Wall wall1(10.5, 8.6);

  // copy contents of wall1 to wall2
  Wall wall2 = wall1;

  // print areas of wall1 and wall2
  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

Constructeurs

Dans ce programme, nous avons utilisé un constructeur de copie pour copier le contenu d'un objet de la classe **Wall** dans un autre. Le code du constructeur de copie est :

```
Wall(Wall &obj) {

  length = obj.length;

  height = obj.height;

}
```

Notez que le paramètre de ce constructeur a l'adresse d'un objet de la classe **Wall**.

Nous affectons ensuite les valeurs des variables de l'objet **obj** aux variables correspondantes de l'objet appelant le constructeur de copie. C'est ainsi que le contenu de l'objet est copié.

Dans **main()**, nous créons ensuite deux objets **wall1** et **wall2** puis copions le contenu de **wall1** vers **wall2** :

Constructeurs

// copier le contenu de wall1 à wall2

Wall wall2 = wall1;

Ici, l'objet **wall2** appelle son constructeur de copie en passant l'adresse de l'objet **wall1** comme argument, c'est-à-dire **&obj = &wall1**.

Remarque : Un constructeur est principalement utilisé pour initialiser des objets. Ils sont également utilisés pour exécuter un code par défaut lors de la création d'un objet.

Destructeur C++

Un destructeur fonctionne à l'opposé du constructeur ; il détruit les objets des classes. Il ne peut être défini qu'une seule fois dans une classe. Comme les constructeurs, il est invoqué automatiquement.

Un destructeur se définit comme un constructeur. Il doit avoir le même nom que la classe. Mais il est préfixé d'un signe tilde (**~**).

Remarque : le destructeur C++ ne peut pas avoir de paramètres. De plus, les modificateurs ne peuvent pas être appliqués sur les destructeurs.

C++ pointeur this

En programmation C++, il s'agit d'un mot-clé qui fait référence à l'instance actuelle de la classe. Il peut y avoir 3 utilisations principales de ce mot-clé en C++.

✓ Il peut être utilisé pour passer l'objet courant en tant que paramètre à une autre méthode.

✓ Il peut être utilisé pour faire référence à la variable d'instance de classe actuelle.

✓ Il peut être utilisé pour déclarer des indexeurs.

C++ pointeur this

```cpp
#include <iostream>
using namespace std;
class Employee {
   public:
       int id; //data member (also instance variable)
       string name; //data member(also instance variable)
       float salary;
       Employee(int id, string name, float salary)
        {
            this->id = id;
           this->name = name;
           this->salary = salary;
        }
       void display()
        {
           cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
```

Comment passer et retourner un objet à partir des fonctions C++ ?

En programmation C++, nous pouvons passer des objets à une fonction de la même manière que passer des arguments normaux.

Exemple 1 : C++ passe des objets à la fonction (diapo suivante)

Comment passer et retourner un objet à partir des fonctions C++ ?

```cpp
// C++ program to calculate the average marks of two students

#include <iostream>
using namespace std;

class Student {

   public:
    double marks;

    // constructor to initialize marks
    Student(double m) {
        marks = m;
    }
};

// function that has objects as parameters
void calculateAverage(Student s1, Student s2) {

    // calculate the average of marks of s1 and s2
    double average = (s1.marks + s2.marks) / 2;

   cout << "Average Marks = " << average << endl;

}

int main() {
    Student student1(88.0), student2(56.0);

  // pass the objects as arguments
   calculateAverage(student1, student2);

    return 0;
}
```

Comment passer et retourner un objet à partir des fonctions C++ ?

Ici, nous avons passé deux objets **Student student1** et **student2** comme arguments à la fonction **calculateAverage().**

Surcharge d'opérateur C++

En C++, nous pouvons changer la façon dont les opérateurs fonctionnent pour les types définis par l'utilisateur comme les objets et les structures. C'est ce qu'on appelle la surcharge d'opérateur. Par exemple,

Supposons que nous ayons créé trois objets **c1**, **c2** et que nous résultions d'une classe nommée **Complex** qui représente des nombres complexes.

Étant donné que la surcharge d'opérateurs nous permet de modifier le fonctionnement des opérateurs, nous pouvons redéfinir le fonctionnement de l'opérateur **+** et l'utiliser pour ajouter les nombres complexes de **c1** et **c2** en écrivant le code suivant : result = c1 + c2;

au lieu de quelque chose comme **result = c1.addNumbers(c2);**

Cela rend notre code intuitif et facile à comprendre.

Surcharge d'opérateur C++

Remarque : Nous ne pouvons pas utiliser la surcharge d'opérateurs pour les types de données fondamentaux tels que int, float, char, etc.

Syntaxe pour la surcharge d'opérateur C++

Pour surcharger un opérateur, nous utilisons une fonction d'opérateur spéciale. Nous définissons la fonction à l'intérieur de la classe ou de la structure dont nous voulons que l'opérateur surchargé travaille avec les objets/variables.

```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

Surcharge d'opérateur C++

Ici, **returnType** est le type de retour de la fonction.

opérator est un mot-clé.

symbol est l'opérateur que nous voulons surcharger. Comme : +, <, -, ++, etc.

arguments sont les arguments passés à la fonction.

Surcharge d'opérateur C++

Surcharge d'opérateurs dans les opérateurs unaires

Les opérateurs unaires opèrent sur un seul opérande. L'opérateur d'incrémentation **++** et l'opérateur de décrémentation **--** sont des exemples d'opérateurs unaires.

Surcharge d'opérateur C++

Exemple 1 : Surcharge de l'opérateur ++ (opérateur unaire)

```
// Overload ++ when used as prefix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;
```

```
int main() {
    Count count1;

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Surcharge d'opérateur C++

Ici, lorsque nous utilisons **++count1;**, l'opérateur **void ++()** est appelé. Cela augmente l'attribut value pour l'objet **count1**de 1.

Remarque : lorsque nous surchargeons les opérateurs, nous pouvons l'utiliser pour travailler comme bon nous semble. Par exemple, nous aurions pu utiliser ++ pour augmenter la valeur de 100.

Cependant, cela rend notre code confus et difficile à comprendre. C'est notre travail en tant que programmeur d'utiliser la surcharge opérateur correctement et de manière cohérente et intuitive.

Surcharge d'opérateur C++

Ici, lorsque nous utilisons **++count1;**, l'opérateur **void ++()** est appelé. Cela augmente l'attribut value pour l'objet **count1**de 1.

Remarque : lorsque nous surchargeons les opérateurs, nous pouvons l'utiliser pour travailler comme bon nous semble. Par exemple, nous aurions pu utiliser ++ pour augmenter la valeur de 100.

Cependant, cela rend notre code confus et difficile à comprendre. C'est notre travail en tant que programmeur d'utiliser la surcharge opérateur correctement et de manière cohérente et intuitive.

Surcharge d'opérateur C++

L'exemple ci-dessus ne fonctionne que lorsque ++ est utilisé comme préfixe. Pour faire fonctionner ++ comme suffixe, nous utilisons cette syntaxe.

```
void operator ++ (int) {
    // code
}
```

Remarquez l'**int** à l'intérieur des parenthèses. C'est la syntaxe utilisée pour utiliser les opérateurs unaires comme suffixe ; ce n'est pas un paramètre de fonction.

Surcharge d'opérateur C++

Exemple 2 : surcharge de l'opérateur ++ (opérateur unaire)

```
// Overload ++ when used as prefix and postfix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}


    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }


    // Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }
    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Surcharge d'opérateur C++

L'exemple 2 fonctionne lorsque **++** est utilisé à la fois comme préfixe et comme suffixe.
Cependant, cela ne fonctionne pas si nous essayons de faire quelque chose comme ceci :

```
Count count1, result;

// Error
result = ++count1;
```

C'est parce que le type de retour de notre fonction d'opérateur est **void**. Nous pouvons résoudre ce problème en faisant de **Count** le type de retour de la fonction opérateur.

Surcharge d'opérateur C++

```
// return Count when ++ used as prefix

Count operator ++ () {
    // code
}

// return Count when ++ used as postfix

Count operator ++ (int) {
   // code
}
```

Les pointeurs

En C++, les pointeurs sont des variables qui stockent les adresses mémoire d'autres variables.

Adresse en C++

Si nous avons une variable **var** dans notre programme, **&var** nous donnera son adresse dans la mémoire.

Voici comment déclarer des pointeurs.

```
int *pointVar;
```

Les pointeurs

Ici, nous avons déclaré un pointeur **pointVar** de type **int**.

Nous pouvons également déclarer des pointeurs de la manière suivante.

```
int* pointVar; // preferred syntax
```

Pointeurs et tableaux C++

En C++, les pointeurs sont des variables qui contiennent les adresses d'autres variables. Non seulement un pointeur peut stocker l'adresse d'une seule variable, il peut également stocker l'adresse des cellules d'un tableau.

Considérez cet exemple :

```
int *ptr;
int arr[5];

// store the address of the first
// element of arr in ptr
ptr = arr;
```

Pointeurs et tableaux C++

Ici, **ptr** est une variable pointeur tandis que **arr** est un tableau **int**. Le code **ptr = arr;** stocke l'adresse du premier élément du tableau dans la variable **ptr**.

Notez que nous avons utilisé **arr** au lieu de **&arr[0]**. C'est parce que les deux sont identiques. Ainsi, le code ci-dessous est le même que le code ci-dessus.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

Les adresses des autres éléments du tableau sont données par **&arr[1], &arr[2], &arr[3]** et **&arr[4].**

Appel C++ par référence : utilisation de pointeurs

Nous avons appris à passer des arguments à une fonction. Cette méthode utilisée est appelée passage par valeur car la valeur réelle est passée.

Cependant, il existe une autre façon de passer des arguments à une fonction où les valeurs réelles des arguments ne sont pas transmises. Au lieu de cela, la référence aux valeurs est transmise.

Appel C++ par référence : utilisation de pointeurs

Exemple 1 : Passage par référence sans pointeurs

Exemple 2 : Passage par référence à l'aide de pointeurs

Héritage C++

L'héritage est l'une des fonctionnalités clés de la programmation orientée objet en C++. Il nous permet de créer une nouvelle classe (classe dérivée) à partir d'une classe existante (classe de base).

La classe dérivée hérite des fonctionnalités de la classe de base et peut avoir ses propres fonctionnalités supplémentaires. Par exemple,

```
class Animal {
    // eat() function
    // sleep() function
};

class Dog : public Animal {
    // bark() function
};
```

Héritage C++

L'héritage est l'une des fonctionnalités clés de la programmation orientée objet en C++. Il nous permet de créer une nouvelle classe (classe dérivée) à partir d'une classe existante (classe de base).

La classe dérivée hérite des fonctionnalités de la classe de base et peut avoir ses propres fonctionnalités supplémentaires. Par exemple,

Ici, la classe **Dog** est dérivée de la classe **Animal**. Puisque **Dog** est dérivé d'**Animal**, les membres d'**Animal** sont accessibles à **Dog**.

Notez l'utilisation du mot-clé **public** lors de l'héritage de **Dog** d'**Animal**.

```
class Animal {
    // eat() function
    // sleep() function
};

class Dog : public Animal {
    // bark() function
};
```

Héritage C++

On peut aussi utiliser les mots-clés **private** et **protected** au lieu de **public**. Nous découvrirons les différences entre l'utilisation de **private**, **public** et **protected** plus tard dans ce didacticiel.

Exemple 1 : exemple simple d'héritage C++

```cpp
// C++ program to demonstrate inheritance

#include <iostream>
using namespace std;

// base class
class Animal {

   public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }
};

// derived class
class Dog : public Animal {

   public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};
```

```cpp
int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();

    // Calling member of the derived class
    dog1.bark();

    return 0;
}
```

Héritage C++

Ici, **dog1** (l'objet de la classe dérivée Dog) peut accéder aux membres de la classe de base Animal. C'est parce que **Dog** est hérité d'**Animal**.

Membres protégés

Le modificateur d'accès **protected** est particulièrement pertinent lorsqu'il s'agit d'héritage C++.

Comme les membres privés, les membres protégés sont inaccessibles en dehors de la classe. Cependant, elles sont accessibles par les classes dérivées et les classes/fonctions amies.

Nous avons besoin de membres protégés si nous voulons masquer les données d'une classe, tout en souhaitant que ces données soient héritées par ses classes dérivées.

Héritage C++

Modificateurs d'accès C++

Les modificateurs d'accès de C++ sont **public**, **private** et **protected**.

L'une des principales caractéristiques des langages de programmation orientés objet tels que C++ est le masquage des données.

Le masquage des données fait référence à la restriction de l'accès aux données membres d'une classe. Cela permet d'éviter que d'autres fonctions et classes ne falsifient les données de la classe.

Cependant, il est également important de rendre accessibles certaines fonctions membres et données membres afin que les données cachées puissent être manipulées indirectement.

Les modificateurs d'accès de C++ nous permettent de déterminer quels membres de classe sont accessibles aux autres classes et fonctions, et lesquels ne le sont pas.

Héritage C++

Ici, les variables **patientNumber** et **diagnosis** de la classe **Patient** sont masquées à l'aide du mot-clé **private**, tandis que les fonctions membres sont rendues accessibles à l'aide du mot-clé **public**.

```
class Patient {

    private:
        int patientNumber;
        string diagnosis;

    public:

      void billing() {
          // code
      }

      void makeAppointment() {
          // code
      }
};
```

Modificateur d'accès public

Le mot-clé **public** est utilisé pour créer des membres publics (données et fonctions).

Les membres du **public** sont accessibles à partir de n'importe quelle partie du programme.

Héritage C++

Dans ce programme, nous avons créé une classe nommée Sample, qui contient une variable publique **age** et une fonction publique **displayAge().**

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

// define a class
class Sample {

    // public elements
   public:
    int age;

    void displayAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // declare a class object
    Sample obj1;

    cout << "Enter your age: ";

    // store input in age of the obj1 object
    cin >> obj1.age;

    // call class function
    obj1.displayAge();
```

Dans **main()**, nous avons créé un objet de la classe **Sample** nommé **obj1**. On accède alors directement aux éléments publics en utilisant les codes **obj1.age** et **obj1.displayAge()**.

Héritage C++

Notez que les éléments publics sont accessibles depuis **main()**. En effet, les éléments publics sont accessibles à partir de toutes les parties du programme.

Héritage C++

Modificateur d'accès privé

Le mot-clé **private** est utilisé pour créer des membres privés (données et fonctions).

Les membres privés ne sont accessibles qu'à partir de la classe.

Cependant, les classes d'amis et les fonctions d'amis peuvent accéder aux membres privés

Héritage C++

Exemple 2 : spécificateur d'accès privé C++

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

// define a class
class Sample {

    // private elements
   private:
    int age;

    // public elements
   public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }
};

int main() {

    int ageInput;

    // declare an object
    Sample obj1;

    cout << "Enter your age: ";
    cin >> ageInput;
```

```cpp
int main() {

    int ageInput;

    // declare an object
    Sample obj1;

    cout << "Enter your age: ";
    cin >> ageInput;

    // call function and pass ageInput as argument
    obj1.displayAge(ageInput);

    return 0;
}
```

Héritage C++

Dans **main()**, l'objet **obj1** ne peut pas accéder directement à la variable de classe **age**.

```
// error
cin >> obj1.age;
```

Nous ne pouvons manipuler l'âge qu'indirectement via la fonction publique **displayAge()**, puisque cette fonction initialise l'âge avec la valeur de l'argument qui lui est passé, c'est-à-dire le paramètre de fonction **int a**.

Modificateur d'accès protégé

Avant de découvrir le spécificateur d'accès protégé, assurez-vous de connaître l'héritage en C++.

Le mot-clé **protected** est utilisé pour créer des membres protégés (données et fonction).

Les membres protégés sont accessibles dans la classe et à partir de la classe dérivée.

Héritage C++

Exemple 3 : spécificateur d'accès protégé par C++

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

// declare parent class
class Sample {
    // protected elements
   protected:
    int age;
};

// declare child class
class SampleChild : public Sample {

   public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }

};

int main() {
    int ageInput;

    // declare object of child class
    SampleChild child;
```

```
int main() {
    int ageInput;

    // declare object of child class
    SampleChild child;

    cout << "Enter your age: ";
    cin >> ageInput;

    // call child class function
    // pass ageInput as argument
    child.displayAge(ageInput);

    return 0;
}
```

Héritage C++

Ici, **SampleChild** est une classe héritée qui est dérivée de **Sample**. La variable **age** est déclarée dans **Sample** avec le mot clé **protected**.

Cela signifie que **SampleChild** peut accéder à l'âge puisque **Sample** est sa classe parent.

Nous voyons cela car nous avons affecté la valeur de l'âge dans **SampleChild** même si l'âge est déclaré dans la classe **Sample**.

Héritage C++

Résumé : public, private et protected

les éléments **public** sont accessibles par toutes les autres classes et fonctions.

les éléments **private** ne sont pas accessibles en dehors de la classe dans laquelle ils sont déclarés, sauf par les classes et fonctions amies.

les éléments **protected** sont comme les **private**, sauf qu'ils sont accessibles par des classes dérivées.

Remarque : par défaut, les membres de la classe en C++ sont privés, sauf indication contraire.

Specifiers	Same Class	Derived Class	Outside Class
`public`	Yes	Yes	Yes
`private`	Yes	No	No
`protected`	Yes	Yes	No

Source : Javapoint.com

Héritage C++

Modes d'accès dans l'héritage C++

Jusqu'à présent, nous avons utilisé le mot-clé **public** afin d'hériter d'une classe d'une classe de base existante. Cependant, nous pouvons également utiliser les mots-clés **private** et **protected** pour hériter des classes. Par exemple,

```
class Animal {
    // code
};

class Dog : private Animal {
    // code
};
```

```
class Cat : protected Animal {
    // code
};
```

Héritage C++

Les différentes façons dont nous pouvons dériver des classes sont appelées **modes d'accès.** Ces modes d'accès ont l'effet suivant :

public : si une classe dérivée est déclarée en mode **public**, les membres de la classe de base sont hérités par la classe dérivée tels quels.

private : dans ce cas, tous les membres de la classe de base deviennent des membres privés de la classe dérivée.

protected : les membres publics de la classe de base deviennent des membres protégés dans la classe dérivée.

Les membres privés de la classe de base sont toujours privés dans la classe dérivée.

Héritage C++

Remplacement d'une fonction membre dans l'héritage

Supposons que la classe de base et la classe dérivée aient des fonctions membres avec le même nom et les mêmes arguments.

Si nous créons un objet de la classe dérivée et essayons d'accéder à cette fonction membre, la fonction membre de la classe dérivée est invoquée au lieu de celle de la classe de base.

La fonction membre de la classe dérivée remplace la fonction membre de la classe de base.

Héritage multiple en C++

L'héritage multiple est le concept de l'héritage en C++ qui permet à une classe enfant d'hériter des propriétés ou du comportement de plusieurs classes de base. Par conséquent, nous pouvons dire que c'est le processus qui permet à une classe dérivée d'acquérir des fonctions membres, des propriétés, des caractéristiques de plus d'une classe de base.

Syntaxe de l'héritage multiple

```
class A
{
// code of class A
}
class B
{
// code of class B
}
class C: public A, public B (access modifier class_name)
{
// code of the derived class
}
```

Problème ? Conflit lors de l'utilisation de deux méthodes de mêmes noms dans la classe fille. Solution ?

Héritage multiple en C++

Ce problème peut être résolu en utilisant la fonction de résolution de portée pour spécifier quelle fonction classer de **base1** ou de **base2**

```
class base1 {
  public:
      void someFunction( ) {....}
};
class base2 {
    void someFunction( ) {....}
};
class derived : public base1, public base2 {};

int main() {
    derived obj;
    obj.someFunction() // Error!
}
```

```
int main() {
    obj.base1::someFunction( );  // Function of base1 class is called
    obj.base2::someFunction();   // Function of base2 class is called.
}
```

Héritage hiérarchique C++

Si plusieurs classes sont héritées de la classe de base, on parle d'héritage hiérarchique. Dans l'héritage hiérarchique, toutes les fonctionnalités communes aux classes enfants sont incluses dans la classe de base.

Par exemple, la physique, la chimie, la biologie sont dérivées du cours de sciences. De même, Dog, Cat, Horse sont dérivés de la classe Animal.

Syntaxe de l'héritage hiérarchique

```
class base_class {
    ... .. ...
}

class first_derived_class: public base_class {
    ... .. ...
}

class second_derived_class: public base_class {
    ... .. ...
}

class third_derived_class: public base_class {
    ... .. ...
}
```

Héritage hiérarchique C++

```cpp
// C++ program to demonstrate hierarchical inheritance

#include <iostream>
using namespace std;

// base class
class Animal {
   public:
    void info() {
        cout << "I am an animal." << endl;
    }
};

// derived class 1
class Dog : public Animal {
   public:
    void bark() {
        cout << "I am a Dog. Woof woof." << endl;
    }
};

// derived class 2
class Cat : public Animal {
   public:
    void meow() {
        cout << "I am a Cat. Meow." << endl;
    }
};
```

Ici, les classes **Dog** et **Cat** sont toutes deux dérivées de la classe **Animal**. Ainsi, les deux classes dérivées peuvent accéder à la fonction **info()** appartenant à la classe **Animal**.

```cpp
int main() {
    // Create object of Dog class
    Dog dog1;
    cout << "Dog Class:" << endl;
    dog1.info();  // Parent Class function
    dog1.bark();

    // Create object of Cat class
    Cat cat1;
    cout << "\nCat Class:" << endl;
    cat1.info();  // Parent Class function
    cat1.meow();

    return 0;
}
```

Fonctions amies et Classes amies

Friend Fonction et friend Classes

Le masquage des données est un concept fondamental de la programmation orientée objet. Il restreint l'accès des membres privés de l'extérieur de la classe.

De même, les membres protégés ne sont accessibles que par les classes dérivées et sont inaccessibles de l'extérieur. Par exemple,

```
class MyClass {
    private:
        int member1;
}

int main() {
    MyClass obj;

    // Error! Cannot access private members from here.
    obj.member1 = 5;
}
```

Friend Function et friend Classes

Cependant, il existe une fonctionnalité en C++ appelée **friend functions** (fonctions amies) qui enfreint cette règle et nous permet d'accéder aux fonctions membres de l'extérieur de la classe.

De même, il existe également une **friend class** (classe d'amis), que nous apprendrons plus tard dans ce didacticiel.

```
class MyClass {
    private:
        int member1;
}

int main() {
    MyClass obj;

    // Error! Cannot access private members from here.
    obj.member1 = 5;
}
```

Friend Function en C++

Une fonction amie peut accéder aux données privées et protégées d'une classe. Nous déclarons une fonction **friend** en utilisant le mot-clé **friend** dans le corps de la classe.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

Friend Function en C++

Exemple :

```cpp
// C++ program to demonstrate the working of friend function

#include <iostream>
using namespace std;

class Distance {
    private:
        int meter;

        // friend function
        friend int addFive(Distance);

    public:
        Distance() : meter(0) {}

};

// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}
```

```cpp
int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Friend Function en C++

Ici, **addFive()** est une fonction d'amie qui peut accéder à la fois aux membres de données privées et publics.

Bien que cet exemple nous donne une idée du concept de fonction d'amie, il ne montre aucune utilisation significative.

Une utilisation plus significative serait d'opérer sur des objets de deux classes différentes. C'est alors que la fonction ami peut être très utile.

```cpp
// Add members of two different classes using friend functions

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {

    public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}

    private:
        int numA;

        // friend function declaration
        friend int add(ClassA, ClassB);
};
```

```cpp
class ClassB {

    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    private:
        int numB;

        // friend function declaration
        friend int add(ClassA, ClassB);
};

// access members of both classes
int add(ClassA objectA, ClassB objectB) {
    return (objectA.numA + objectB.numB);
}

int main() {
    ClassA objectA;
    ClassB objectB;
    cout << "Sum: " << add(objectA, objectB);
    return 0;
}
```

Friend Function en C++

Dans ce programme, **ClassA** et **ClassB** ont déclaré **add()** comme **fonction friend**. Ainsi, cette fonction peut accéder aux données privées des deux classes.

Une chose à noter ici est que la fonction d'ami à l'intérieur de **ClassA** utilise la **ClassB**. Cependant, nous n'avons pas défini **ClassB** à ce stade.

```
// inside classA
friend int add(ClassA, ClassB);
```

Pour que cela fonctionne, nous avons besoin d'une déclaration avancée de **ClassB** dans notre programme.

```
// forward declaration
class ClassB;
```

Classe amie en C++

Nous pouvons également utiliser une classe friend en C++ en utilisant le mot-clé friend. Par exemple,

```
class ClassB;

class ClassA {
   // ClassB is a friend class of ClassA
   friend class ClassB;
   ... .. ...
}

class ClassB {
   ... .. ...
}
```

Lorsqu'une classe est déclarée classe amie, toutes les fonctions membres de la classe amie deviennent des fonctions amies.

Puisque **ClassB** est une classe d'amis, nous pouvons accéder à tous les membres de **ClassA** depuis **ClassB**.

Cependant, nous ne pouvons pas accéder aux membres de **ClassB** à partir de **ClassA**. C'est parce que la relation d'ami en C++ est seulement accordée, pas prise.

Classe amie en C++

Nous pouvons également utiliser une classe friend en C++ en utilisant le mot-clé friend. Par exemple,

```cpp
// C++ program to demonstrate the working of friend class

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {
    private:
        int numA;

        // friend class declaration
        friend class ClassB;

    public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}
};
```

```cpp
class ClassB {
    private:
        int numB;

    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    // member function to add numA
    // from ClassA and numB from ClassB
    int add() {
        ClassA objectA;
        return objectA.numA + numB;
    }
};

int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

Classe amie en C++

Ici, **ClassB** est une classe amie de **ClassA**. Ainsi, **ClassB** a accès aux membres de **classA**.

En **ClassB**, nous avons créé une fonction **add()** qui renvoie la somme de **numA** et **numB**.

Puisque **ClassB** est une classe amie, nous pouvons créer des objets de **ClassA** à l'intérieur de **ClassB**.

Bonne assimilation

christianyesibi@gmail.com

Suivre les actualités sur les plateformes de l'IAI-Cameroun :