Programmation en C/C++

Série 14
P.O.O.

Encapsulation / Héritage / Héritage multiple Polymorphisme / Masquage / Généricité

Important !

Difficile !

Objectifs
Comprendre les avantages de la programmation orientée objet.
Utiliser l'encapsulation de données, l'héritage, le masquage de méthode en C++.

1- Encapsulation de données en P.O.O

Nous avons vu au chapitre précédent que l'encapsulation de données en P.O.O est un terme qui signifie que les variables d'instance et méthodes sont bien encapsulées et donc protégées dans leur classe. Si on les déclare en "*private*" seules des méthodes de la classe pourront appeler les autres méthodes et lire ou modifier les variables d'instance de la dite classe. Il s'agit donc d'un système de protection et de contrôle d'accès aux données (variables d'instances) et fonctions (méthodes).

C'est l'un des grands avantages de la P.O.O. :

- Une variable ne peut pas être modifiée par n'importe qui ;
- Une méthode ne peut pas être appelée par n'importe qui ;
- Une méthode ne peut pas modifier n'importe quoi !

Il existe trois catégories d'accès aux variables d'instance et méthodes :

- *Public*,
- *Private*,
- *Protected*.

Private
Les méthodes et variables d'instance "*private*", ne sont accessibles qu'à partir de méthodes appartenant à la même classe. Même le *main()* ne peut y accéder !

Public
Les méthodes et variables d'instance "*public*", sont accessibles à partir de méthodes de la classe ou extérieures à la classe.

Protected
Les méthodes et variables d'instance "*protected*" sont accessibles par toute méthode de la classe ou d'une classe qui lui est dérivée (notion abordée lors du paragraphe sur l'héritage).

2-Héritage


Un objet est une instance d'une classe : il possède donc toutes les caractéristiques de sa classe d'appartenance avec ses variables d'instance et ses méthodes.

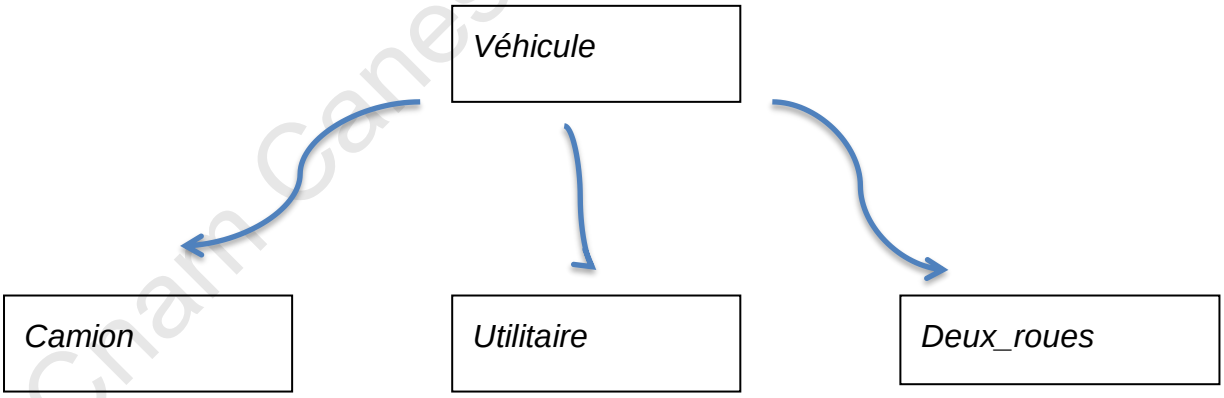

On peut créer une nouvelle **classe dérivée** de la classe de départ et cette classe **héritera** des caractéristiques de la **classe mère** et possèdera éventuellement des variables d'instance et/ou des méthodes supplémentaires.

Exemple
On créera ci-après une classe *Véhicule* pour caractériser des véhicules à moteur :

```
class Vehicule
{

/*Variables d'instance*/
string Immatriculation ;
string Marque ;
string Modele ;
string Couleur ;
int Puissance ;
string Carburant ;
int nombreKilomètres

/*méthodes*/

void Avancer ()
        {
…
        }

void Reculer ()
        {
…
```

```
        }

void Stationner ()
        {
…
        }

void Tomber_en_Panne ()
        {
…
        }

void Accidenter ()
        {
…
        }

void Reparer ()
        {
…
        }

} ; /*fin de la classe Vehicule*/
```

On peut maintenant réaliser un héritage en créant trois nouvelles classes dérivées de la classe de départ *Vehicule*.
On créera, par exemple, une classe *Camion*, une classe *Utilitaire* (fourgon, camionnette…), une classe *Deux_roues*.



Exemple d'héritage simple

La classe *Deux_roues* possèdera toutes les caractéristiques de la classe *Vehicule*.

Elle possèdera, par exemple, 3 variables d'instance supplémentaires :
bool Necessite_permis ;
int Nb_passagers ;
bool Bequille_laterale;

Pour tenir compte de la nécessité éventuelle de posséder le permis moto pour le conduire, le nombre de passagers transportables (1 ou 2) et la présence ou non d'une béquille latérale.

Elle possèdera, par exemple, également une méthode supplémentaire :
void Repare_Deux_roues()
{
…
}
qui prendra en charge des réparations spécifiques d'un 2 roues (changer la fourche, réparer la poignée d'accélération, changer la selle…)

On dira que :
- La **classe mère** est la classe *Vehicule*
- La classe **dérivée** *Deux_roues* est une **classe fille**
- La classe *Deux_roues* **hérite** des caractéristiques de la classe *Vehicule*
- La classe *Deux_roues* est une **spécialisation** de la classe *Vehicule*

Remarque
La classe dérivée aura un constructeur par défaut mais pourra également posséder ses propres constructeurs.

Création d'un objet Scooter125
Pour créer un objet nommé *Scooter 125*, instanciation de la classe *Deux_roues*, il suffit d'écrire la ligne suivante au niveau du programme principal :
Deux_roues Scooter125;

Quand on créera cet objet le constructeur de *Vehicule* sera appelé en premier suivi de l'appel au constructeur de la classe dérivée Deux_roues.

Scooter125 possèdera les variables d'instance suivantes :
string Immatriculation ;
string Marque ;
string Modele ;
string Couleur ;
int Puissance ;
string Carburant ;

bool Necessite_permis ;
int Nb_passagers ;
bool Bequille_laterale;

Scooter125 pourra bénéficier des méthodes suivantes :
void Avancer ()
void Reculer ()
void Stationner ()
void Tomber_en_Panne ()
void Accidenter ()
void Reparer ()

void Repare_Deux_roues()

Implémentation d'un héritage
Voici le code C++ correspondant à la classe *Deux_roues* dérivée de la classe *Vehicule* :

```
class Deux_roues : public Vehicule
{
bool Necessite_permis ;
int Nb_passagers ;
bool Bequille_laterale;

void Repare_Deux_roues()
{
//…
}

}; //fin de la classe Deux_roues
```

Remarque
La ligne "*class Deux_roues : public Vehicule*" signifie que l'on crée une nouvelle classe nommée *Deux_roues* qui héritera de la classe mère nommée *Vehicule*.

Création d'un objet Scooter125
Pour créer un objet nommé *Scooter 125*, instanciation de la classe Deux_roues, il suffit d'écrire la ligne suivante au niveau du programme principal :
Deux_roues Scooter125;

Quand on créera cet objet, le constructeur de *Vehicule* sera appelé en premier suivi de l'appel au constructeur de la classe dérivée Deux_roues.

Scooter125 possèdera les variables d'instance suivantes :
string Immatriculation ;
string Marque ;
string Modele ;
string Couleur ;
int Puissance ;
string Carburant ;

bool Necessite_permis ;
int Nb_passagers ;
bool Bequille_laterale;

Scooter125 pourra être manipulé par les méthodes suivantes :
void Avancer ()
void Reculer ()
void Stationner ()

void Tomber_en_Panne ()
void Accidenter ()
void Reparer ()

void Repare_Deux_roues()

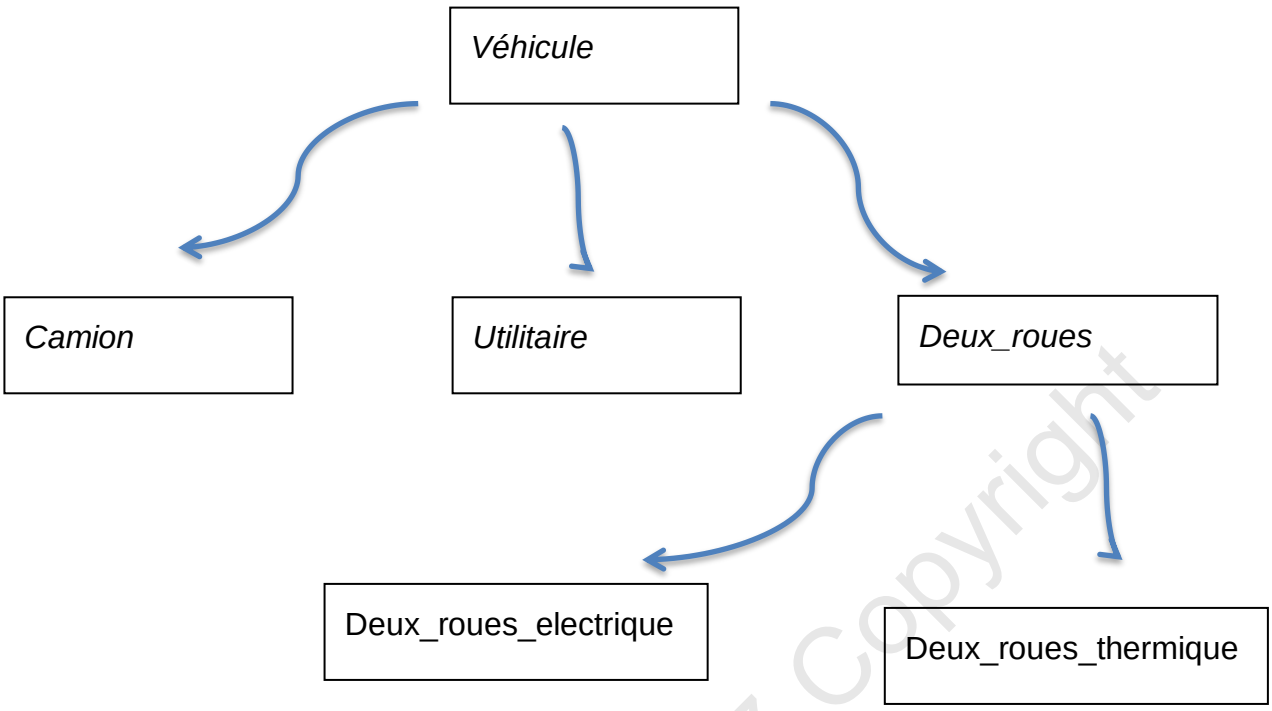
3-  Héritage multiple (hors programme)

Le langage C++ autorise l'héritage multiple. En effet, une classe dérivée hérite des caractéristiques de sa classe mère ; mais elle peut donner naissance à de nouvelles classes dérivées, à son tour. On parlera d'héritage multiple.
Dans notre exemple précédent nous avions une classe mère nommée *Vehicule* et une classe dérivée nommée *Deux_roues*. La classe *Deux_roues* peut donner naissance à deux classes filles nommées par exemple : *Deux_roues_electrique* et *Deux_roues_thermique* qui n'ont pas les mêmes caractéristiques techniques (variables d'instances) ou d'entretien (méthodes) du fait de la différence des types de moteurs électrique et à essence.

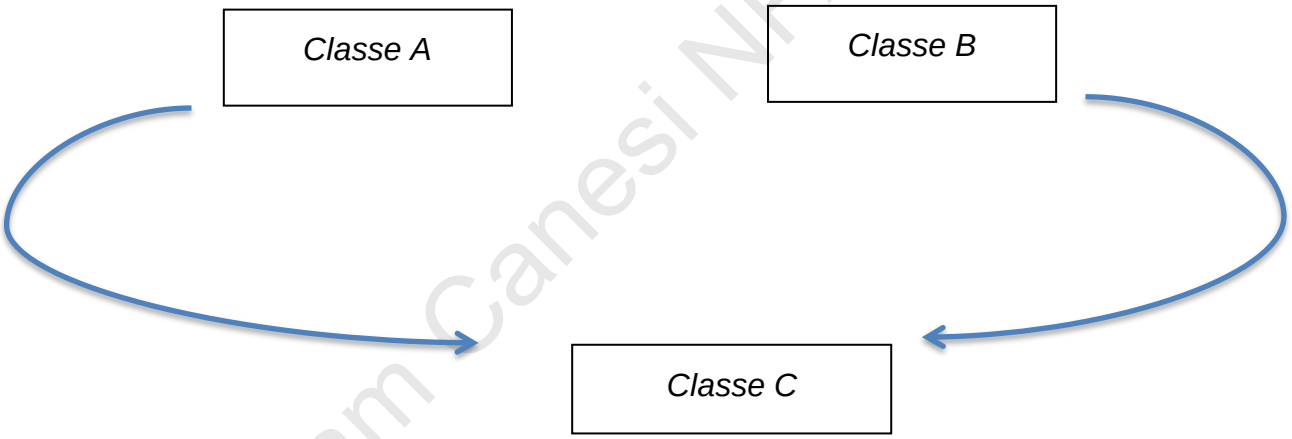
Une classe peut également hériter de deux classes mères (héritage multiple).

 **Remarque**

Dans le cadre d'un projet important il faudra avoir une visibilité, dès le départ, de la classe mère et de ses classes dérivées et sous-dérivées. On n'aura pas intérêt à trop spécialiser la classe mère afin qu'elle comporte uniquement des variables d'instance et méthodes très générales et communes à tous les futurs objets créés. Puis, on spécialisera en réalisant des héritages multiples en ajoutant, à chaque fois, un niveau supplémentaire de spécialisation.



Exemple d'héritage multiple



Autre exemple d'héritage multiple
La classe C hérite des classes A et B

Encapsulation des données et méthodes en mode *protected*

On a vu plus haut, que le mode *protected* pouvait également être utilisé pour protéger les variables d'instances et les méthodes d'une classe. C'est un mode à mi-chemin entre le mode *private* et le mode *public* : seules les méthodes des classes dérivées (et également sous-dérivées dans le cadre d'un héritage multiple) pourront accéder aux variables d'instance et/ou autres méthodes de la classe mère.

Problèmes posés par l'héritage multiple

Soit une classe C héritant de deux classes A et B.
Un problème se pose pour les cas suivants :

- Les classes A et B ont des variables d'instance de même nom
- Les classes A et B ont des méthodes de même nom

Il faudra alors préciser le nom de la classe mère en préfixe de l'attribut ou de la méthode héritée. On utilisera l'opérateur de portée "::" qui précisera la classe mère et évitera les problèmes et confusions. **Exemple** : *A :: Var1;* (*Var1* hérite de *A*).

4- Polymorphisme

Le polymorphisme c'est la possibilité pour certaines méthodes d'être utilisées d'une manière différente en fonction du contexte d'exécution.

Ce polymorphisme peut être mis en œuvre par :

- La redéfinition/masquage de méthodes lors de l'héritage,
- La réalisation de patrons de méthodes (templates) grâce à la généricité
- L'écriture de méthodes dites virtuelles.

4-1 Redéfinition et masquage de méthodes

On a vu dans le chapitre précédent qu'en C++, des fonctions ou méthodes peuvent être surchargées : elles possèdent le même nom mais des paramètres différents.

En C++, il est également possible de masquer des méthodes : on pourra écrire deux méthodes de même nom et avec les mêmes paramètres ! D'habitude cela n'est pas possible mais C++ l'autorise dans le contexte d'une classe et d'un héritage : La première méthode sera située dans la classe mère, la seconde dans la classe dérivée. Quand la méthode sera utilisée sur un objet, le compilateur saura utiliser la méthode idoine (adaptée à la situation) en fonction de l'instanciation : objet de la classe mère ou objet de la classe dérivée.
Dans notre exemple précédent, au lieu de créer une nouvelle méthode *reparer_Deux_roues()*, on aurait pu écrire une nouvelle méthode *reparer()* spécifique à la classe *Deux_roues* et masquant donc la méthode *reparer()* de la classe mère *Vehicule*.
Quand *reparer()* est exécutée sur un objet de type Deux_roues la méthode *reparer()* de la classe *Vehicule* est masquée : elle n'est pas utilisée.

4-2  Généricité / Templates

Le langage C++ autorise le **polymorphisme**. Un terme pour dire qu'une méthode peut prendre plusieurs formes : on parle de **méthode générique**… Par exemple, elle pourra travailler avec des entiers comme avec des réels. On n'écrira qu'une seule fois la méthode et on n'utilisera pas de surcharge de méthode : avec la surcharge on aurait écrit une première méthode qui travaille avec des entiers puis on aurait surchargé cette méthode par une seconde qui travaille avec des réels. Là, nous allons écrire une seule méthode générique qui n'est pas spécialisée. On parle également de "patron" de méthode en anglais désigné par le mot *template* ou *pattern*. C'est lors de l'instanciation de l'objet que l'on dévoilera son type et les méthodes de la classe s'y adapteront.

Exemple de cours

Deux associations (Assos1 et Assos2) bénéficient de subventions publiques et de subventions privées qui pourront être des réels comme des entiers.
On créera une classe nommée *Association* avec un constructeur recevant en paramètres les deux montants.
On écrira une méthode **générique** non statique *Somme()* qui retourne le montant total des subventions reçues par une association et une autre méthode **générique** non statique *Affiche()* qui affiche les montants des subventions privée et publique.
On utilisera donc le polymorphisme autorisé par C++.

```
Total des subventions de l'association 1 : 50000 euros
Total des subventions publiques de l'association  : 20000 euros
Total des subventions privees de l'association : 30000 euros

Total des subventions de l'association 2 : 9001.3 euros
Total des subventions publiques de l'association  : 1000.5 euros
Total des subventions privees de l'association : 8000.8 euros


Process returned 0 (0x0)   execution time : 0.011 s
Press ENTER to continue.
```

Exemple de résultat obtenu avec deux méthodes génériques

Implémentation complète

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

template <class T> class Association
{
public : T Publ ;
public : T Prive ;

public :  Association (T a, T b)
{
```

```
Publ = a;
Prive = b;
}

public :  void Affiche ()
{
cout << "Total des subventions publiques de l'association  : " <<this->Publ<<" euros"<<endl;
cout << "Total des subventions privees de l'association : " <<this->Prive<<" euros"<<endl;
}

public :  T Somme ()
{
return (this->Publ + this->Prive);
}

} ; // fin de classe


int main (void)
{
Association<int> Assos1(20000,30000);
Association <double> Assos2(1000.5 , 8000.80);

cout <<endl;
cout << "Total des subventions de l'association 1 : " << Assos1.Somme()<<" euros"<<endl;
Assos1.Affiche();
cout <<endl;

cout << "Total des subventions de l'association 2 : "<<Assos2.Somme()<<" euros"<<endl;
Assos2.Affiche();
cout <<endl;
}
```

4-3 Méthodes virtuelles (non traité)

Les classes abstraites ou virtuelles permettent de définir des méthodes (comportements) sans en écrire le corps ; elles seront implémentées seulement dans les classes dérivées.
Une classe est dite virtuelle si elle possède au moins une méthode virtuelle pure. Dans ce cas il est impossible d'instancier un objet de cette classe.
Une méthode est dite **méthode virtuelle pure** si elle commence par le mot clé "*virtual*" et se termine par "*=0*"

Exemple

```
virtual int Surface () = 0;
```

La méthode virtuelle *Surface* de la classe *Figure* sera bien évidemment implémentée différemment dans les classes héritées *Carré* et *Cercle*.

Exercices

Exercice 1 Héritage

Énoncé

Créer un programme en langage C++ qui comporte deux classes nommées *Parent* et *Enfant*. La classe *Parent* comportera deux variables d'instance *Age* et *Prenom* de types respectifs *int* et *string*. Elle ne possède pas de méthodes propres.

La classe *Enfant* sera dérivée de la classe *Parent*, sans variable ni méthode supplémentaire dans un premier temps.

Créer ensuite, au niveau du programme principal un objet, instance de *Parent* (Eric, 32 ans) et un objet, instance de *Enfant* (Fanny, 2 ans). Afficher leurs variables d'instances à l'écran.

Corrigé

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

class Parent {
public : int Age;
string Prenom;

Parent (int a, string b)
{
Age= a;
Prenom = b;
}

};   //fin classe Parent


class Enfant : public Parent
{
// Constructeur de la classe Enfant
public : Enfant (int a, string b) : Parent(a,b)
{
}

}; // fin classe Enfant

int main() {
Parent P1(32,"Eric");
Enfant E1(2, "Fanny");
cout<<" Parent P1 : "<<P1.Prenom <<"  "<< P1.Age<<" ans"<<endl;
cout<<" Enfant E1 : "<<E1.Prenom<<"  "<<E1.Age<<" ans"<<endl;
}
```

Remarque

On peut aussi écrire le constructeur de la classe *Enfant* en dehors de la classe *Enfant*. Le code devient :

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

class Parent {
public : int Age;
string Prenom;

// constructeur de Parent (:: car en dehors de la classe Parent)
Parent (int a, string b)
{
Age= a;
Prenom = b;
}

};   //fin classe Parent

class Enfant : public Parent
{
public : Enfant (int a, string b);

}; // fin classe Enfant

// Constructeur de la classe Enfant
Enfant::Enfant (int a, string b) : Parent(a,b) // :: car on est à l'extérieur de la classe Enfant
{
}


int main() {
Parent P1(32,"Eric");
Enfant E1(2, "Fanny");
cout<<" Parent P1 : "<<P1.Prenom <<"  "<< P1.Age<<" ans"<<endl;
cout<<" Enfant E1 : "<<E1.Prenom<<"  "<<E1.Age<<" ans"<<endl;
}
```


Exercice 2 Héritage et variable supplémentaire

Énoncé

On améliorera ici le programme précédent en ajoutant une variable d'instance *NivClasse* (*string*) à la classe *Enfant* pour mémoriser le niveau scolaire de l'enfant.

Modifier la classe *Enfant*, le constructeur, les données d'affichage et créer ensuite des instances de ces classes pour obtenir l'affiche suivant à l'écran :

```
Parent P1 : Eric  32 ans
Enfant E1 : Julie  10 ans classe de CM2
```

Corrigé

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

class Parent {
public :
int Age;
string Prenom;
Parent (int a, string b);
};

// constructeur de Parent ; :: car constructeur en dehors de la classe
Parent:: Parent(int a, string b) {
Age= a;
Prenom = b;
}

class Enfant : public Parent {
public:   string NivClasse;

//constructeur de la classe Enfant, classe héritée de Parent
Enfant (int a, string b, string Niv) :Parent(a,b)
{
NivClasse = Niv;
}

};    // fin de classe Enfant


int main() {
Parent P1(32,"Eric");
Enfant E1(10, "Julie","CM2");
cout<<" Parent P1 : "<<P1.Prenom <<"  "<< P1.Age<<" ans"<<endl;
cout<<" Enfant E1 : "<<E1.Prenom<<"    "<<E1.Age<<" ans"<<" classe de "<<E1.NivClasse<<endl;
}
```


Exercice 3 Héritage et méthode supplémentaire

Énoncé

Ajouter maintenant une méthode non-statique et publique nommée *Jouer()*, spécifique à la classe *Enfant*. Cette méthode se contente d'afficher un message à l'écran.

```
Parent P1 : Eric  32 ans
Enfant E1 : Julie  10 ans classe de CM2
Je joue !
```

Corrigé

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


class Parent {
public :
int Age;
string Prenom;
Parent (int a, string b);
};

// constructeur de Parent :: car en dehors de la classe
Parent:: Parent(int a, string b) {
Age= a;
Prenom = b;
}

class Enfant : public Parent {
public:   string NivClasse;

//constructeur de Enfant classe héritée de Parent
Enfant (int a, string b, string Niv) :Parent(a,b) {
NivClasse = Niv;
}

public : void Joue()
{
cout<<"Je joue !"<<endl;
}
};



int main() {
Parent P1(32,"Eric");
Enfant E1(9,"Marine","CM2");

cout<<" Parent P1 : "<<P1.Prenom <<"  "<< P1.Age<<" ans"<<endl;
cout<<" Enfant E1 : "<<E1.Prenom<<"  "<<E1.Age<<" ans"<<" classe de
"<<E1.NivClasse<<endl;
E1.Joue();
}
```


Exercice 4 Masquage de méthode

Énoncé

Reprendre le programme précédent et supprimer toutes les lignes de code, au niveau du programme principal, affichant des données. Puis créer une première méthode non

statique, sans argument, nommée *Affiche()* qui affichera les valeurs des variables d'instance d'un parent. Créer ensuite une seconde méthode *Affiche()* qui affichera les valeurs des variables d'instance d'un enfant. Utiliser ces deux méthodes au niveau du *main().* La seconde méthode *Affiche()* présente dans la classe héritée *Enfant* masquera celle présente dans la classe mère *Parent*.

Corrigé

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

class Parent {
public :
int Age;
string Prenom;
Parent (int a, string b);

public : void Affiche () {
cout<<"Parent P1 : "<<this->Prenom <<"  "<< this->Age<<" ans"<<endl;
}

};  // Fin classe Parent

// constructeur de Parent :: car en dehors de la classe
Parent:: Parent(int a, string b) {
Age= a;
Prenom = b;
}

// Classe Enfant
class Enfant : public Parent {
public: string NivClasse;

//constructeur de Enfant ; classe héritée de Parent
Enfant (int a, string b, string Niv) : Parent(a,b) {
NivClasse = Niv;
}

public : void Joue()
{
cout<<"Je joue !"<<endl;
}

public : void Affiche ()
{
cout<<"Enfant E1 : "<<this->Prenom <<"  "<< this->Age<<" ans ;  classe de "<<
this->NivClasse<<endl;
}

};      //Fin classe Enfant
```

```
int main() {
Parent P1(32,"Eric");
Enfant E1(9, "Marine","CM2");

P1.Affiche();
E1.Joue();
E1.Affiche();
cout<<endl;
}
```

Remarque
On constate que les deux méthodes *Affiche()* sont non-statiques car elles utilisent le mot clé *this*, pointeur sur un objet.
Ces deux méthodes sont quasiment identiques : elles portent le même nom, elles ne renvoient rien et n'ont pas d'argument. Quand on utilise *Affiche()* sur un objet de la classe *Enfant*, la méthode *Affiche()* de la classe *Enfant* est utilisée et masque donc celle de la classe *Parent*. La méthode *Affiche()* de la classe *Enfant* affiche juste le niveau scolaire de l'enfant, en plus.

Remarque 2
On peut écrire le constructeur de la classe *Enfant* à l'extérieur de cette classe. Il faudra bien écrire "*Enfant::Enfant*"
Le code devient :

```
class Enfant : public Parent {
public: string NivClasse;

//Proto constructeur
Enfant (int a, string b, string NivClasse);

public : void Joue()
{
cout<<"Je joue !"<<endl;
}

public : void Affiche ()
{
cout<<"Enfant E1 : "<<this->Prenom <<"  "<< this->Age<<" ans ;  classe de "<<
this->NivClasse<<endl;
}

};      //Fin classe Enfant

//constructeur de Enfant ; classe héritée de Parent
Enfant::Enfant (int a, string b, string Niv) : Parent(a,b)
{
NivClasse = Niv;
}
```

Exercice 5 Classe complète

Énoncé

On se placera dans le contexte d'une épicerie qui vend des produits divers et variés (stylo, crayon, balai, livre, glaces, poisson, viande, courgettes, oeufs…).

Pour cela on créera une classe *Produit*. Cette classe possèdera les variables d'instances suivantes :

- une référence (string),
- un libellé (string),
- un prix hors taxes (double),
- un stock minimal à posséder en rayon (int),
- le stock en cours (int).

Cette classe possèdera également les méthodes suivantes :

- *Affiche()* qui affiche les variables d'instance d'un objet de la classe
- *Stock_dispo (),* qui renvoie la quantité en stock
- *Vendre (int nombre),* qui modifie le stock en le décrémentant
- *Stocker (int nombre),* qui modifie le stock en l'incrémentant
- *Est_dispo (),* qui renvoie si le produit est en stock
- *Commander (),* qui gère le réapprovisionnement en renvoyant automatiquement le nombre de produits commandés si la quantité en stock est inférieure au stock minimal.

Écrire un programme qui affiche les résultats comme ci-après :

```
Ref : AE2456
Libelle : Cahier spirale 24
Prix H.T. : 89
Stock mini : 2
Stock actuel : 8

Stock disponible avant vente : 8
Stock disponible apres vente : 5
Quantite a commander : 0
```

Exemple d'exécution du programme

Corrigé

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

class Produit
{
```

```
string ref;
string lib;
double prix_H_T;
int stock_mini;
int en_stock;


/*constructeur*/
public : Produit (string ref, string lib, double prix_H_T, int stock_mini, int en_stock)
	{
		this->ref=ref;
		this->lib=lib;
		this->prix_H_T=prix_H_T;
		this->stock_mini=stock_mini;
		this->en_stock=en_stock;
	}


/*méthodes*/

public : void Affiche ()
{
cout<<"Ref : "<<this->ref<<endl;
cout<<"Libelle : "<<this->lib<<endl;
cout<<"Prix H.T. : "<<this->prix_H_T<<endl;
cout<<"Stock mini : "<<this->stock_mini<<endl;
cout<<"Stock actuel : "<<this->en_stock<<endl;
cout<<endl;
}

public : int Stock_dispo ()
{
	return this->en_stock;
}



public : void Vendre(int nombre)
{
	this->en_stock -= nombre;
}



public : void Stocker (int nombre)
{
	this->en_stock += nombre;
}
```

```
public : bool Est_dispo ()
{
return (this->en_stock > 0);
}


public : int Commander ()
 {
 if (this->en_stock >= this->stock_mini)
        return 0;
else
        return this->stock_mini - this->en_stock;
 }


};



int main() {
/* on déclare et crée un objet de type produit nommé P1*/
Produit P1("AE2456","Cahier spirale 24",89,2,8);


P1.Affiche();
cout<<"Stock disponible avant vente : "<<P1.Stock_dispo()<<endl;
P1.Vendre(3);
cout<<"Stock disponible apres vente : "<<P1.Stock_dispo()<<endl;


cout<<"Quantite a commander : "<<P1.Commander()<<endl;
cout<<endl;
}
```

Exercice 6 Héritage

Énoncé

On reprendra ici l'exercice précédent et on créera une classe *Aliment* qui hérite de la classe *Produit*. Un objet de la classe *Aliment* aura pour variables d'instances supplémentaires :

- un taux de TVA (double),
- une date de péremption (date)
- un secteur de vente (chaîne de caractères).

La classe *Aliment* n'aura pas de méthode propre.

Écrire un programme qui crée un objet de chaque type et les utilise.

Corrigé

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

class Produit
{
protected :
string ref;
string lib;
double prix_H_T;
int stock_mini;
int en_stock;


/*constructeur*/
public : Produit (string ref, string lib, double prix_H_T, int stock_mini, int en_stock)
	{
		this->ref=ref;
		this->lib=lib;
		this->prix_H_T=prix_H_T;
		this->stock_mini=stock_mini;
		this->en_stock=en_stock;
	}


/*méthodes*/

public : void Affiche ()
{
cout<<"Ref : "<<this->ref<<endl;
cout<<"Libelle : "<<this->lib<<endl;
cout<<"Prix H.T. : "<<this->prix_H_T<<endl;
cout<<"Stock mini : "<<this->stock_mini<<endl;
cout<<"Stock actuel : "<<this->en_stock<<endl;
cout<<endl;
}



public : int Stock_dispo ()
{
	return this->en_stock;
}



public : void Vendre(int nombre)
{
```

```
        this->en_stock -= nombre;
}



public : void Stocker (int nombre)
{
        this->en_stock += nombre;
}


public : bool Est_dispo ()
{
return (this->en_stock > 0);
}


public : int Commander ()
 {
 if (this->en_stock >= this->stock_mini)
        return 0;
else
        return this->stock_mini - this->en_stock;
 }


};// fin classe Produit


class Date
        {
public :
int jour;
int mois;
int annee;
};

class Aliment : public Produit {
public : string secteur; Date date_perempt ; double taux_tva;

//constructeur de la classe fille pour les objets nommés Aliment
Aliment    (string ref,string lib,double prix_H_T, int stock_mini,
                int en_stock, string secteur, Date date_perempt,
                double taux_tva)
       : Produit(ref,lib,prix_H_T,stock_mini,en_stock) // ne pas mettre les types des variables
{
this->secteur=secteur;
this->date_perempt= date_perempt;
this->taux_tva= taux_tva;
```

```
}

};   //fin classe Aliment


int main() {
int vente;
Produit P1("AE2456","Cahier spirale 24",89,2,8);

        Date date_perempt ;
        date_perempt.jour=12;
        date_perempt.mois = 6;
        date_perempt.annee = 2016;

Aliment A1("CR3640","Boisson sport",3.2,5,21,"boissons",date_perempt,5.5);

P1.Affiche();
A1.Affiche();
cout<<"Stock disponible avant vente : "<<A1.Stock_dispo()<<endl;
cout<<"Quantite vendue ?";
cin>>vente;
A1.Vendre(vente);
cout<<"Stock disponible apres vente : "<<A1.Stock_dispo()<<endl;
cout<<"quantite a commander : "<<A1.Commander()<<endl;
}
```

Remarque
Dans ce code source on remarque que la méthode *Affiche*() peut-être utilisée sur un objet instanciant la classe *Aliment*. Cependant, seules ses cinq variables d'instance (héritées de la classe *Produit*) seront affichées : la méthode *Affiche*() n'ayant pas été modifiée et étant initialement prévue pour des objets instanciant la classe *Produit*.

Exercice 7 Masquage de méthode

Énoncé

On créera maintenant une seconde méthode *Affiche*(), propre à la classe *Aliment* qui masquera donc la méthode *Affiche*() de la classe *Produit()*. Créer deux objets distincts, instances respectives des classes *Produit* et *Aliment*. Afficher ensuite toutes leurs variables d'instances à l'aide des deux méthodes *Affiche*() comme suit :

```
Ref : AE2456
Libelle : Cahier spirale 24
Prix H.T. : 89
Stock mini : 2
Stock actuel : 8

Ref : CR3640
Libelle : Boisson sport
Prix H.T. : 3.2
Stock mini : 5
Stock actuel : 21
Secteur : boissons
Date de peremption : 12/6/2016
Taux de TVA : 5.5

Stock disponible avant vente : 21
Quantite vendue ?20
Stock disponible apres vente : 1
quantite a commander : 4
```

Exemple d'affichage avec un masquage de méthode

Conseil
Attention à la bonne gestion de la date de péremption.

Corrigé

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

class Produit
{
protected :
string ref;
string lib;
double prix_H_T;
int stock_mini;
int en_stock;


/*constructeur*/
public : Produit (string ref, string lib, double prix_H_T, int stock_mini, int en_stock)
        {
                this->ref=ref;
                this->lib=lib;
                this->prix_H_T=prix_H_T;
                this->stock_mini=stock_mini;
                this->en_stock=en_stock;
        }


/*méthodes*/

public : void Affiche ()
{
cout<<"Ref : "<<this->ref<<endl;
cout<<"Libelle : "<<this->lib<<endl;
```

```
cout<<"Prix H.T. : "<<this->prix_H_T<<endl;
cout<<"Stock mini : "<<this->stock_mini<<endl;
cout<<"Stock actuel : "<<this->en_stock<<endl;
cout<<endl;
}


public : int Stock_dispo ()
{
        return this->en_stock;
}


public : void Vendre(int nombre)
{
        this->en_stock -= nombre;
}


public : void Stocker (int nombre)
{
        this->en_stock += nombre;
}


public : bool Est_dispo ()
{
return (this->en_stock > 0);
}


public : int Commander ()
 {
 if (this->en_stock >= this->stock_mini)
        return 0;
else
        return this->stock_mini - this->en_stock;
 }


};// fin classe Produit


class Date
        {
        public :
        int jour;
        int mois;
        int annee;
        };
```

```
class Aliment : public Produit {
public : string secteur; Date date_perempt ; double taux_tva;

//constructeur de la classe fille pour les objets nommés Aliment
Aliment    (string ref,string lib,double prix_H_T, int stock_mini,
                int en_stock, string secteur, Date date_perempt,
                double taux_tva)
      : Produit(ref,lib,prix_H_T,stock_mini,en_stock) // ne pas mettre les types des variables
{
this->secteur=secteur;
this->date_perempt= date_perempt;
this->taux_tva= taux_tva;
}


public : void Affiche () {
cout<<"Ref : "<<this->ref<<endl;
cout<<"Libelle : "<<this->lib<<endl;
cout<<"Prix H.T. : "<<this->prix_H_T<<endl;
cout<<"Stock mini : "<<this->stock_mini<<endl;
cout<<"Stock actuel : "<<this->en_stock<<endl;
cout<<"Secteur : "<<this->secteur<<endl;
cout<<"Date de peremption : "<<this->date_perempt.jour<<"/"
                            <<this->date_perempt.mois<<"/"
                            <<this->date_perempt.annee<<endl;
cout<<"Taux de TVA : "<<this->taux_tva<<endl;  cout<<endl;
}

};   //fin classe Aliment


int main() {
int vente;
Produit P1("AE2456","Cahier spirale 24",89,2,8);

      Date date_perempt ;
      date_perempt.jour=12;
      date_perempt.mois = 6;
      date_perempt.annee = 2016;

Aliment A1("CR3640","Boisson sport",3.2,5,21,"boissons",date_perempt,5.5);

P1.Affiche();
A1.Affiche();
cout<<"Stock disponible avant vente : "<<A1.Stock_dispo()<<endl;
cout<<"Quantite vendue ?";
cin>>vente;
A1.Vendre(vente);
```

```
cout<<"Stock disponible apres vente : "<<A1.Stock_dispo()<<endl;
cout<<"quantite a commander : "<<A1.Commander()<<endl;
}
```

Exercice 8 Héritage multiple à partir de deux classes/masquage

Énoncé

On caractérise la position d'une fusée par ses longitude, latitude, altitude (nombre réels). On peut créer des fusées classiques et des fusées chargées dont on connait la charge (en kilogrammes) à mettre en orbite.
Créer une classe *Fusee* avec une méthode *Affiche.*
Créer une classe *Charge* avec une méthode *Affiche.*
Créer une classe *FuseeChargee* qui dérivera des deux classes *Fusee* et *Charge* (héritage multiple). Cette classe *FuseeChargee* possède également sa propre méthode *Affiche* qui va appeler la méthode *Affiche* de la classe *Fusee* et affichera également la charge de la fusée.

Corrigé

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

// 3 classes

class Fusee {
protected: float x, y, z;
public: void Creer(float, float,float);
void Affiche(void);
};

class Charge {
protected: short Poids;
public: void Creer(short);
void Affiche(void);
};

class FuseeChargee: public Fusee, public Charge {
public: void Creer(float, float, float, short);
void Affiche(void);
};

// constructeur Creer de Fusee
void Fusee::Creer(float Longitude, float Latitude, float Altitude){
x = Longitude;
y = Latitude;
z = Altitude;
}

// méthode Affiche de Fusee
void Fusee::Affiche(void) {
cout <<"\n Longitude : "<< x <<"\n Latitude : "<< y  <<"\n Altitude : "<< z << "\n ";
```

```
}

// constructeur Creer de Charge
void Charge::Creer(short Charg) {
Poids = Charg;
}

// méthode Affiche de Charge
void Charge::Affiche(void) {
cout << Poids;
}

// méthode Creer héritée de FuseeChargee
void FuseeChargee:: Creer (float Longitude, float Latitude, float Altitude, short Charg)
{
Fusee:: Creer(Longitude, Latitude, Altitude);
Charge:: Creer(Charg);
}

// méthode Affiche de FuseeChargee
void FuseeChargee:: Affiche(void) {
Fusee:: Affiche();
cout << "Charge de la fusee : ";
Charge :: Affiche();
cout << " Kg\n ";

}

int main(void)
{
FuseeChargee FusC1;
FusC1.Creer(-1.2   ,   5.4,   1900.7,  3400);
cout << "\nFusee non chargee ";
FusC1.Fusee::Affiche();

cout << "\nFusee chargee ";
Fusc1.FuseeChargee::Affiche();
}
```

Remarque

Vous pouvez créer une fusée et une fusée chargée et observer les différences de comportement des méthodes *Affiche()* de la classe mère et de la classe fille.
Y-a-t-il bien eu masquage ?

Exercice 9 Héritage multiple (non corrigé)

Énoncé

On créera maintenant deux classes nommées *Viande* et *Légume* héritant de la classe *Aliment* elle-même dérivant de la classe *Produit*.

Viande aura la variable d'instance supplémentaire : *type_race* ;

Légume aura la variable d'instance supplémentaire : *type_agriculture* ; et *origine_pays*; Créer un légume et une viande, instances respectives de *Légume* et *Viande* et afficher leurs variables d'instance.

Corrigé
#

Exercice 10 Héritage multiple (non corrigé)

Énoncé

On reprendra les exercices avec les classes Enfant et Parent ; on créera une nouvelle classe *Petit_Enfant*, classe dérivée de la classe *Enfant*, elle même dérivée de la classe *Parent*. Cette classe ne comportera pas de variable d'instance et de méthode propre. Créer ensuite des instances de ces classes : un parent, un enfant, un petit-enfant puis afficher leurs variables d'instances respectives.

Corrigé
#

Exercice 11 Classes virtuelles (non corrigé)

Énoncé

On s'intéressera à des figures géométriques type Carré, Rectangle, Triangle, Cercle on créera donc quatre classes dérivées de la **classe virtuelle** *Figure*. L'objectif est de créer une de chacune des figures différentes à l'aide d'un constructeur et d'afficher ses caractéristiques, son périmètre et sa superficie. Il existera donc 2 méthodes non statiques : *Périmètre* et *Surface*.

Rappels
Carré : **variables d'instance** : Longueur, Largeur
Périmètre : 2 x (Longueur + Largeur)
Surface : Longueur x Largeur

Rectangle : **variables d'instance** : Longueur, Largeur
Périmètre : 2 x (Longueur + Largeur)
Surface : Longueur x Largeur

Triangle : **variables d'instance** : Base, Côté 2, Côté 3, Hauteur
Périmètre : Base + Côté 2 + Côté 3,
Surface : ½ x (Base x Hauteur)

Cercle : **variables d'instance** : Rayon
Périmètre : 2 x 3,14 x Rayon
Surface : 3,14 x Rayon x Rayon

Corrigé
#