



Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar El-Mehraz Fès
Département d'Informatique

Programmation Orientée objet en C++

Par Nouredine Chenfour
2012-2015

Chapitre 1

Bases de la Programmation Orientée Objet

1.1 Principe de base

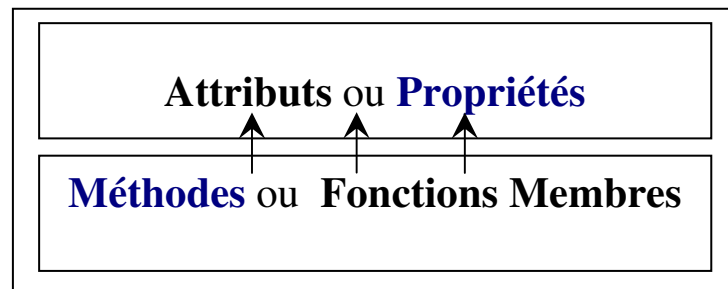
Une structuration classique d'un programme consiste en une structuration à deux niveaux : les données d'une part et le code d'une autre part. Ainsi les données qui décrivent ou caractérisent une même entité sont regroupées ensemble dans une même structure de donnée : un enregistrement ou un tableau. De la même manière, les instructions réalisant ensemble une tâche bien définie et complète sont regroupées dans une même procédure ou fonction.

La Programmation Orientée Objets (**POO**) consiste en une structuration de plus haut niveau. Il s'agit de regrouper ensemble les données et toutes les procédures et fonctions qui permettent la gestion de ces données. On obtient alors des entités comportant à la fois un ensemble de données et une liste de procédures et de fonctions pour manipuler ces données. La structure ainsi obtenue est appelée : **Objet**.

Un objet est alors une généralisation de la notion d'enregistrement. Il est composé de deux parties :

- Une partie statique composée de la liste des données de l'objet. On les appelle : **Attributs** ou **Propriétés**, ou encore : **Données Membres**.

- Une partie dynamique qui décrit le comportement ou les fonctionnalités de l'objet. Elle est constituée de l'ensemble des procédures et des fonctions qui permettent à l'utilisateur de configurer et de manipuler l'objet. Ainsi les données ne sont généralement pas accessibles directement mais à travers les procédures et les fonctions de l'objet. Celles-ci sont appelées : **Méthodes** ou **Fonctions Membres**.



1.2 Encapsulation

Un objet tel que défini plus haut est une variable caractérisée par des propriétés et des méthodes. La définition de ces propriétés et de ces méthodes devra être réalisée dans une structure de données appelée **classe**.

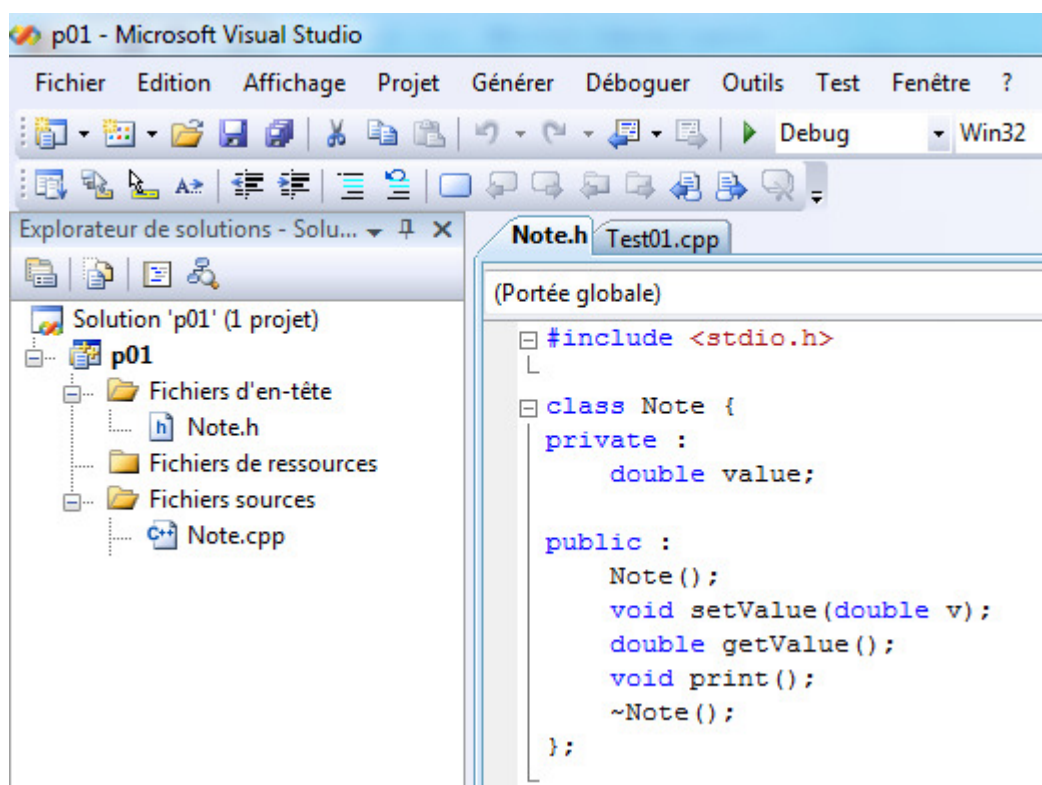
Une **classe** est donc le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Les données seront généralement appelées des propriétés, les fonctions qui opèrent sur ces propriétés sont appelées des méthodes.

Créer un objet depuis une classe est une opération qui s'appelle **l'instanciation**. L'objet ainsi créé pourra être appelé aussi : **instance**. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type de données et variable.

Pour utiliser une classe il faut créer une instance de cette classe (appelée aussi objet).

1.3 Syntaxe de définition d'une classe

En C++, on sépare d'habitude (ce n'est pas une obligation syntaxique) entre la déclaration d'une classe, qui contient les propriétés et les prototypes des méthodes, et la définition de celle-ci fournissant l'implémentation ou le corps des méthodes. La déclaration est faite dans un fichier d'entête avec l'extension « .h » (exemple Note.h), quand à la définition des méthodes elle est réalisée dans un fichier source qui porte le même nom avec l'extension .cpp (exemple Note.cpp) :



1.3.1 Déclaration d'une classe :

class NomDeClasse {
Propriétés de la classe
Méthodes de la classe
};

1.3.2 Visibilité des membres (Privatisation) :

3 mots clés sont utilisés pour définir un niveau de visibilité donné à un membre de la classe : `private`, `public` et `protected`.

- Le mot clé **`private`** permet de rendre un membre privé. Il s'agit du concept de privatisation. Les données membre sont généralement les cibles de **privatisation**. La manipulation d'un objet doit se faire en principe à l'aide de ces méthodes en ignorant complètement la structure interne de l'objet donc ces propriétés (l'abstraction de la composition interne de l'objet : Principe de l'encapsulation). Cependant, certaines méthodes peuvent aussi être privées ; des méthodes d'aide, ou des méthodes dont l'invocation externe peut créer des problèmes si non appelées dans un contexte particulier, ou encore des méthodes sans grande importance pour l'utilisateur final de la classe.
- Le mot clé **`public`** est appliqué à des membres (en principe seulement des méthodes) pour les rendre accessibles depuis l'extérieur de la classe.
- Le mot clé **`protected`** offre un niveau de visibilité situé entre les 2 niveaux précédant. Des membres « `protected` » sont des membres inaccessibles en dehors de leur classe sauf dans des classes filles (voir la rubrique de l'héritage).

Syntaxe :

```
class NomDeClasse {  
    private :  
        ...  
    public :  
        ...  
    protected :  
        ...  
};
```

Il est cependant possible d'alterner autant de fois que nécessaire entre ces différents mots clés.

Bonne pratique :

- Données privées
- Méthodes publiques

Exemple :

Note.h

```
class Note {  
  
private :  
    double value;  
  
public :  
    void setValue(double v);  
    double getValue();  
    void print();  
};
```

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode m1() puis la méthode m2(), m2() peut être appelée sans problème dans m1().

1.3.3 Définition des méthodes de la classe :

L'implémentation des méthodes préalablement déclarées dans un fichier d'entête est réalisée dans un fichier source .cpp selon la syntaxe suivante :

```
#include "Fichier d'entête.h"  
  
type NomDeClasse::nomDeMethode1(...) {  
    ...  
}  
  
type NomDeClasse::nomDeMethode2(...) {  
    ...  
}
```

Exemple :

Note.cpp

```
#include "Note.h"

void Note::setValue(double v) {
    if (v >= 0 && v <= 20) {
        value = v;
    }
}

double Note::getValue() {
    return value;
}

void Note::print() {
    printf("Note = %.2f\n", value);
}
```

1.4 Accesseurs

Parmi les méthodes pouvant assurer l'accès contrôlé aux propriétés qui sont habituellement privées, il est à définir des méthodes d'introduction et de récupération de données. Ces méthodes sont appelées des **Accesseurs**, ou encore **getters** et **setters** faisant appel à leur nomination.

Ainsi, pour une donnée membre nommée « x », il est à prévoir les deux méthodes suivantes :

```
void setX(type x) ;
```

Appelée setter et permettant d'affecter la valeur du paramètre à la propriété « x ».

Ainsi que la méthode getX :

```
type getX() ;
```

Appelée getter et permettant de retourner la valeur de « x ».

Chapitre 2

Instanciation

2.1 Instances de classe et Instanciation

Un **objet** d'une classe est appelé aussi « **instance** ». Il existe deux types d'objets en C++ : Les objets « **statiques** » et les objets « **dynamiques** ». L'opération de création d'une instance de classe est appelée « **Instanciation** ».

2.1.1 Objets statiques

La déclaration d'un objet statique est simplement réalisée de la manière suivante :

NomDeClasse nomD'objet ;

Suite à cette déclaration l'instance est automatiquement créée et est prête à usage. L'accès aux membres (publiques) est réalisé à l'aide de l'opérateur « . »

Exemple :

```
void exp02() {
    Note n1; // Déclaration d'un objet
    //n1.value = 25; Accès impossible
    n1.setValue(17);
    n1.print();

    n1.setValue(25);
    printf("La note après modification : %.2f\n",
           n1.getValue() );

    getchar();
}
```

2.1.2 Objets dynamiques

Dans ce cas, il est à séparer entre la déclaration et l'instanciation. La déclaration permet juste de définir un pointeur sur une instance non encore créée. La création ou l'instanciation doit être réalisée explicitement à l'aide de l'opérateur **new** qui se charge de créer une instance (ou un objet) de la classe et de l'associer à la variable (ou pointeur). Il s'agit d'allouer de l'espace mémoire à l'objet en question. Cet espace mémoire restera associé à l'objet jusqu'à ce qu'il y ait une désallocation explicitement réalisée à l'aide de l'opérateur inverse : **delete**.

Syntaxe :

<code>objet = new NomDeClasse();</code>
--

L'accès aux membres publics est dans ce cas réalisé à l'aide de l'opérateur « **->** ».

Exemple :

```
void exp03() {  
    // Déclaration :  
    Note *n1;  
  
    // Instanciation :  
    n1 = new Note();  
  
    n1->setValue(18);  
    n1->print();  
  
    delete n1;  
    getchar();  
}
```

2.1.3 L'objet « this »

Il existe un objet dynamique (pointeur) prédéfini « **this** » pouvant être utilisé à l'intérieur d'une classe, permettant de référencer à l'aide de la notation « **this->...** » les membres (propriétés ou méthodes) de l'objet courant. L'utilisation de « this » permettra par exemple de distinguer entre une propriété et une variable locale ou un paramètre. Exemple :

```
void Note::setValue(double value) {  
    if (value >= 0 && value <= 20) {  
        this->value = value;  
    }  
}
```

2.2 Constructeurs et Destructeurs

Un **Constructeur** est une méthode particulière invoquée automatiquement lors de l'instanciation d'un objet (déclaration pour un objet statique et instanciation à l'aide de `new` pour un objet dynamique). Elle contiendra l'ensemble des instructions permettant d'initialiser ou de créer le contenu de l'objet.

Le constructeur se distingue syntaxiquement des autres méthodes par les deux règles suivantes :

1. Il porte le nom de la classe et
2. Il ne dispose d'aucun type de retour.

Remarque :

Si la classe ne contient pas de constructeur, le compilateur se charge de créer un constructeur par défaut sans paramètres et sans contenu.

Le **Destructeur** est une méthode particulière invoquée automatiquement lors de la destruction d'un objet. Pour un objet statique, le destructeur est invoqué en fin du bloc où il a été déclaré. Pour un objet dynamique, le destructeur est invoqué lors de l'appel de l'opérateur **delete**.

Le destructeur sert à libérer, fermer, désallouer ou finaliser les opérations de création réalisée durant de l'existence de l'objet. Il est aussi sans type de retour et il porte le nom de la classe préfixé par le symbole tilde « `~` ».

Exemple : La classe Note avec constructeur et destructeur

Note.h

```
class Note {
private :
    double value;

public :
    Note();
    void setValue(double v);
    double getValue();
    void print();
    ~Note();
};
```

Implémentation :

```
Note::Note() {  
    value = 10;  
    printf("Constructeur Note()\n");  
}  
...  
Note::~~Note() {  
    printf("Destructeur Note()\n");  
}
```

Exemple d'utilisation :

1. Constructeur & Destructeur : cas d'un objet statique

```
void exp04() {  
    printf("Début\n");  
    {  
        Note n1; ← Invocation du Constructeur  
        n1.print();  
  
        printf("Fin du bloc\n");  
    } ← Invocation du Destructeur  
    printf("Après bloc\n");  
    getchar();  
}
```

Exécution :

```
Début  
Constructeur Note()  
Note = 10.00  
Fin du bloc  
Destructeur Note()  
Après bloc
```

2. Constructeur & Destructeur : cas d'un objet dynamique

```
void exp05() {  
    printf("Début\n");  
    printf("Déclaration\n");  
    Note *n1;  
    {  
        printf("Instanciation\n");  
        n1 = new Note();  
        n1->print();  
  
        printf("Fin du bloc\n");  
    }  
    printf("Après bloc\n");  
    delete n1;  
    getchar();  
}
```

Invocation du
Constructeur

Invocation du
Destructeur

Exécution :

```
Début  
Déclaration  
Instanciation  
Constructeur Note()  
Note = 10.00  
Fin du bloc  
Après bloc  
Destructeur Note()
```

Chapitre 3

Principaux concepts de la POO

3.1 Surcharge ou Surdéfinition

Il s'agit de définir une même méthode plusieurs fois dans la même classe. Dans ce cas il serait nécessaire que chaque définition soit faite avec une signature différente.

Exemple :

```
void setBirthday(Date birthday);  
void setBirthday(int day, int month, int year);
```

La surcharge peut aussi être appliquée au constructeur. Une classe peut ainsi disposer d'un ou plusieurs constructeurs.

Exemple :

Date.h

```
class Date {
private :
    int day, month, year;
public:
    Date();
    // Surcharge ou Surdéfinition du constructeur
    Date(int day, int month, int year);
    // Accesseurs : getters & setters
    void setDay(int day);
    void setMonth(int month);
    void setYear(int year);
    int getDay();
    int getMonth();
    int getYear();
    void print();
    ~Date(void);
};
```

Dans ce cas l'instanciation des objets de type « Date » pourra être effectuée de deux façons différentes :

Cas des objets statiques :

```
void exp01() {
    Date d1;
    d1.print();
}

void exp02() {
    Date d1(20, 11, 2015);
    d1.print();
}
```

Cas des Objets dynamiques :

```
void exp03() {
    Date *d1 = new Date();
    Date *d2 = new Date(14, 05, 2011);
    d1->print();
    d2->print();
}
```

Implémentation de la classe « Point » :

Date.cpp

```
#include "Date.h"

Date::Date(void) {
    day = 4;
    month = 11;
    year = 2014;
}

Date::Date(int day, int month, int year) {
    setDay(day);
    setMonth(month);
    setYear(year);
}

// Accesseurs : getters & setters
void Date::setDay(int day) {
    if (day >= 1 && day <= 31) {
        this->day = day;
    }
}

void Date::setMonth(int month) {
    if (month >= 1 && month <= 12) {
        this->month = month;
    }
}

void Date::setYear(int year) {
    if (year > 0) {
        this->year = year;
    }
}

int Date::getDay() {
    return day;
}

int Date::getMonth() {
    return month;
}

int Date::getYear() {
    return year;
}

void Date::print() {
    printf("%d/%d/%d\n", day, month, year);
}

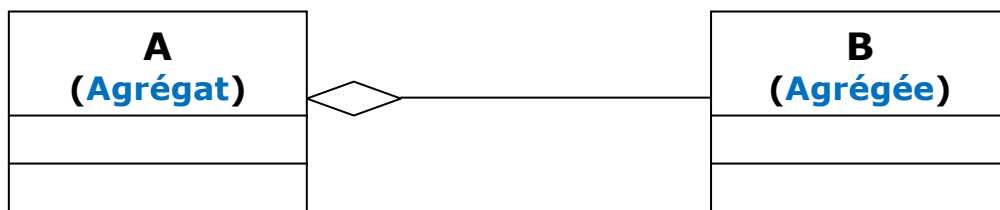
Date::~Date(void) {
}
```


3.2 Agrégation et Composition

Il s'agit de relations de dépendance entre les objets de classes différentes.

3.2.1 Agrégation

Permet d'exprimer qu'un objet d'une classe « **agrégat** » est relié à un objet d'une autre classe par une relation de dépendance. L'une des propriétés de l'objet « agrégat » de type « **A** » est ainsi de type une autre classe « **B** ». Cependant, l'absence d'information ou de valeur de « **B** » dans un objet de type « **A** » n'empêche pas l'objet d'exister. Il s'agit d'une relation de dépendance **faible** entre les deux classes « **A** » et « **B** ». Comme la relation qui pourrait exister entre un employé et une entreprise ou une personne est son téléphone.

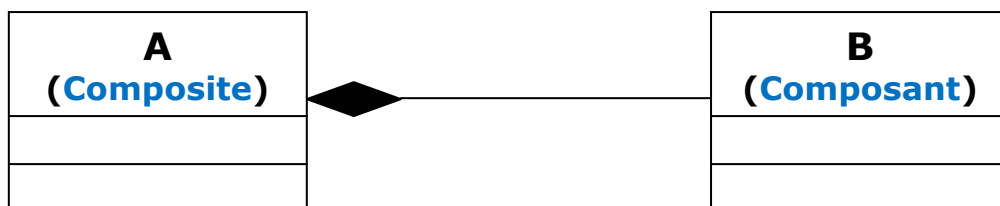


L'agrégation est exprimée généralement à l'aide de propriétés de type dynamique.

3.2.2 Composition

Forme particulière de l'agrégation : **agrégation ou dépendance forte**.

L'objet composant est « **physiquement** » contenu dans l'objet **agrégat** appelé aussi **composite** dans ce cas. On peut prendre comme exemple : un ordinateur et son écran ou un clavier et les touches qui le composent. On ne peut pas imaginer le « composite » sans le « composant ».



La composition est implémentée en C++ généralement à l'aide de propriétés de type statique.

Exemple :

Person.h

```
class Person {
private :
    int id;
    char *firstName;
    char *lastName;
    Date birthday; // Composition
public:
    Person(void);
    Person(int id, char *firstName, char *lastName,
            Date birthday);

    void setId(int id);
    void setFirstName(char *firstName);
    void setLastName(char *lastName);
    void setBirthday(Date birthday);
    // Surcharge
    void setBirthday(int day, int month, int year);
    void print();
    ~Person(void);
};
```

Exemples d'utilisation :

```
void exp04() {
    Person p1;
    p1.setId(1);
    p1.setFirstName("FN");
    p1.setLastName("LN");
    Date d1(10, 05, 1996);
    p1.setBirthday(d1);
    p1.print();
}

void exp05() {
    Person p1;
    p1.setId(1);
    p1.setFirstName("FN");
    p1.setLastName("LN");
    p1.setBirthday(10, 05, 199);
    p1.print();
}
```

```

void exp06() {
    Date d1(10, 05, 1996), d2(14, 10, 1993);
    Person *p1 = new Person(1, "FN1", "LN1", d1);
    Person *p2 = new Person(2, "FN2", "LN2", d2);
    p1->print();
    p2->print();
}

```

Implémentation de la classe « Person »

Person.cpp

```

#include "Person.h"
#include "Date.h"

Person::Person(void) {
}

Person::Person(int id, char *firstName, char *lastName,
               Date birthday) {
    setId(id);
    setFirstName(firstName);
    setLastName(lastName);
    setBirthday(birthday);
}

void Person::setId(int id) {
    this->id = id;
}

void Person::setFirstName(char *firstName) {
    this->firstName = firstName;
}

void Person::setLastName(char *lastName) {
    this->lastName = lastName;
}

void Person::setBirthday(Date birthday) {
    this->birthday = birthday;
}

void Person::setBirthday(int day, int month, int year) {
    Date d(day, month, year);
    this->birthday = d;
}

void Person::print() {
    printf("%d) %s %s : %d/%d/%d\n", id, firstName,
        lastName, birthday.getDay(), birthday.getMonth(),
        birthday.getYear());
}

Person::~Person(void) {
}

```

Exemple d'agrégation :

Produit.h

```
class Produit {
private:
    char *ref;
    char *desig;
    double pu;
    double qs;
    Date *date; // Agrégation
public:
    Produit(void);
    Produit(char *ref, char *desig, double pu, double qs);
    void setRef(char *ref);
    void setDesig(char *desig);
    void setDate(Date *date);
    void setDate(int day, int month, int year);
    void print();
    ~Produit(void);
};
```

Exemples d'utilisation :

```
void exp07() {
    Produit *p1 = new Produit("C01", "Clavier", 200, 10);
    p1->setDate(new Date(4, 11, 2014));
    p1->print();
}

void exp08() {
    Produit *p1 = new Produit("C01", "Clavier", 200, 10);
    p1->setDate(4, 11, 2014);
    p1->print();
    delete p1;
}
```

Produit.cpp

```
#include "Produit.h"
#include <malloc.h>
#include <string.h>

Produit::Produit(void) {
}

Produit::Produit(char *ref, char *desig, double pu, double qs)
{
    setRef(ref);
    setDesig(desig);
    this->pu = pu;
    this->qs = qs;
}

void Produit::setRef(char *ref) {
    this->ref = (char *)malloc(strlen(ref) + 1);
    strcpy(this->ref, ref);
}

void Produit::setDesig(char *desig) {
    this->desig = (char *)malloc(strlen(desig) + 1);
    strcpy(this->desig, desig);
}

void Produit::setDate(Date *date) {
    this->date = date;
}

void Produit::setDate(int day, int month, int year) {
    this->date = new Date(day, month, year);
}

void Produit::print() {
    printf("%s : %f, %d/%d/%d\n", desig, pu,
           date->getDay(), date->getMonth(), date->getYear()
    );
}

Produit::~Produit(void) {
    free(ref);
    free(desig);
    delete date;
    printf("Destructeur de Produit()\n");
}
```

3.3 Références dynamiques communes


Si deux objets ont une référence commune (le cas de la date dans l'exemple précédent), lors de la destruction d'un premier objet, l'objet agrégé et référencé par les deux objets sera lui-même détruit; on se trouvera alors avec le deuxième objet défectueux.

```
void exp09() {
    Date *d1 = new Date(4, 11, 2014);
    Produit *p1 = new Produit("C01", "Clavier", 200, 10);
    p1->setDate(d1);

    Produit *p2 = new Produit("C02", "Ecran", 1200, 5);
    p2->setDate(d1);

    p1->print();
    delete p1;

    p2->print();
    delete p2;
}
```



```
Produit::~Produit() {
    free(ref);
    free(desig);
    delete date;
}
```

Il s'agit donc d'une situation à éviter. Dans l'exemple suivant, chaque produit va référencer un objet « Date » différent :

```
void exp10() {
    Date *d1 = new Date(4, 11, 2014);
    Date *d2 = new Date(4, 11, 2014);

    Produit *p1 = new Produit("C01", "Clavier", 200, 10);
    p1->setDate(d1);

    Produit *p2 = new Produit("C02", "Ecran", 1200, 5);
    p2->setDate(d2);

    p1->print();
    delete p1;

    p2->print();
    delete p2;
}
```

Chapitre 4

Héritage

4.1 Principe et raison d'être

L'héritage consiste à développer de nouvelles classes issues de classes existantes. Il s'agit d'étendre les fonctionnalités de classes déjà définies en ajoutant de nouvelles propriétés et méthodes. Ainsi une classe existante et ayant déjà servi dans la réalisation d'un certain nombre d'applications n'a pas à être modifiée si on constate son insuffisance par rapport à des propriétés ou des services donnés. Plutôt que de reprendre son code, la solution étant de définir une nouvelle classe supportant les nouveaux besoins sans pour autant réécrire le code existant. Il s'agira donc d'une solution syntaxique permettant d'hériter d'un existant sans avoir à le reprendre :

```
class Fille : public Mère {  
  
};
```

Cette simple écriture permettra de créer une nouvelle classe nommée : Classe **Fille** ou classe **Dérivée**, depuis une classe existante nommée : classe **Mère**, **classe de base** ou encore **super classe**.

A travers un objet de la classe fille, on aura déjà accès à tous les services définis dans la classe mère. En plus, dans la fille, il est possible d'ajouter d'autres propriétés et services.

La classe fille sera alors une extension de la classe mère. En plus, tout objet de la classe fille pourra jouer le même rôle que celui des objets de la classe mère, avec en plus des possibilités supplémentaires.

Soit une classe de base « Point » caractérisée par ses coordonnées x et y :

```
class Point {
private:
    int x, y;
public:
    Point(void);
    Point(int x, int y);
    void setX(int x);
    void setY(int y);
    int getX();
    int getY();
    void print();
    ~Point(void);
};
```

On pourra ainsi manipuler un objet de la classe Point comme suit :

```
void expl1() {
    Point p1(20, 30);
    p1.print();
}
```

Ci-dessous une classe fille qui étendra les fonctionnalités de Point en ajoutant une nouvelle propriété « color » :

```
class ExtendedPoint : public Point {
private:
    int color;
public:
    ExtendedPoint(void);
    ExtendedPoint(int x, int y, int color);
    void setColor(int color);
    int getColor();
    ~ExtendedPoint(void);
};
```


Un objet de la nouvelle classe « ExtendedPoint » pourra bien accéder aux services définis dans la classe mère « Point » :

```
void exp12() {  
    ExtendedPoint p;  
    p.setX(20);  
    p.setY(30);  
    p.print();  
}
```

4.2 Constructeurs de la classe fille

Une classe fille garde son autonomie concernant les constructeurs. Ainsi, les constructeurs d'une classe mère ne peuvent pas servir dans la construction des objets de la classe fille. Une classe fille doit alors disposer de ses propres constructeurs.

Exemple :

```
class ExtendedPoint : public Point {  
private:  
    int color;  
public:  
    ExtendedPoint(void);  
    ExtendedPoint(int x, int y, int color);  
};
```

Cependant tous les constructeurs d'une classe fille appellent implicitement et automatiquement, par défaut, le constructeur sans paramètre de la classe mère qui est supposé existant ; sinon on aura un problème de compilation de la classe fille.

L'appel au constructeur sans paramètre de la classe mère sera réalisé comme première instruction implicite de tout constructeur de la classe fille.

Exemple :

Constructeur sans paramètre de la classe mère :

```
Point::Point(void) {  
    printf("Constructeur Point()\n");  
    this->x = 0;  
    this->y = 0;  
}
```

Constructeur sans paramètre de la classe fille :

```
ExtendedPoint::ExtendedPoint() {  
    printf("Constructeur ExtendedPoint()\n");  
    setX(0);  
    setY(0);  
    this->color = 0;  
}
```

Soit le code suivant :

```
ExtendedPoint p;
```

L'exécution donnera lieu au résultat suivant :

```
Constructeur Point()  
Constructeur ExtendedPoint()
```

Il est cependant à remarquer qu'on peut choisir explicitement quel constructeur invoquer depuis un constructeur de la classe fille. Il s'agira d'une syntaxe explicite utilisée lors de la définition d'un constructeur, permettant de dérouter l'appel à un constructeur au choix. La résolution du constructeur choisi est effectuée par l'intermédiaire de la signature de celui-ci :

```
Fille::Fille(...) : Mère(...) {  
    ...  
}
```

Exemple :

```
ExtendedPoint::ExtendedPoint(int x, int y, int color) : Point(x, y)
{
    this->color = color;
}
```

Ce constructeur invoquera alors automatiquement le constructeur avec deux paramètres « int » Point(int, int) de la classe mère au lieu du constructeur sans paramètres par défaut.

L'appel de ce constructeur sera fait comme suit (cas d'un objet statique) :

```
void exp13() {
    ExtendedPoint p(50, 60, 1);
    p.print();
}
```

4.3 Redéfinition de méthodes

Une méthode définie dans une classe mère peut ne plus être valable pour une classe fille.

Exemple :

```
void Point::print() {
    printf("Point(%d, %d)\n", x, y);
}
```

Cette méthode peut être invoquée sur des objets de la classe fille « ExtendedPoint » comme dans l'exemple (13) précédent. Mais le résultat sera comme suit :

```
Point(50, 60)
```

L'ExtendedPoint est affiché comme un Point.

Pour cela, il sera nécessaire de reprendre la définition de la méthode « print » dans la classe « ExtendedPoint ». Il s'agit d'un autre concept de la programmation orientée objet qu'on appelle « Redéfinition ».

Déclaration :

```
class ExtendedPoint : public Point {  
private:  
    int color;  
public:  
    ...  
    void print();  
    ...  
};
```

Redéfinition :

```
void ExtendedPoint::print() {  
    printf("ExtendedPoint(%d, %d, %d)\n", getX(), getY(), color);  
}
```

Remarque :

Si on veut appeler la version mère de la méthode depuis la classe fille, il suffit de la préfixer avec la notation : **Mère::**

Exemple :

```
void ExtendedPoint::methode() {  
    ...  
    Point::print();  
    ...  
}
```

4.4 Affectation entre objets

4.4.1 Objets de même type

Il est à rappeler qu'une affectation entre deux objets de même type (même classe) est bien évidemment possible :

Exemple 1 : Objets statiques :

```
void expl4() {  
    Point p1(20, 30);  
    Point p2 = p1;  
    p2.print();  
}
```

Remarque :

Il est aussi à rappeler ou à remarquer que le compilateur C++ génère automatiquement un constructeur sans paramètre par défaut qui ne fait aucun traitement si la classe ne contient aucun constructeur.

Nous signalons en plus dans cette rubrique que le compilateur génère aussi ce qu'on appelle un « **constructeur de copie** » qui est appelé à chaque initialisation d'un objet (à distinguer l'initialisation qui est une affectation d'objets réalisée lors de la déclaration par rapport à une affectation ordinaire) :

Classe objet2 = objet1 ;

Ce constructeur a comme fonction par défaut de copier toutes les propriétés de « objet1 » dans « objet2 ». Cette opération aura des résultats négatifs pour le cas des propriétés dynamiques. La copie sera ainsi une copie de référence et toute propriété dynamique sera en conséquence commune entre les deux objets. La destruction de l'un des deux objets entrainera alors la suppression de ces propriétés ; ce qui causera l'invalidité de l'autre objet.

Il sera alors nécessaire dans ce genre de situations de définir plus convenablement le constructeur de copie.

Le constructeur de copie pourra ainsi être défini explicitement comme suit :

```
class Classe {  
  
    Classe (Classe &p) ;  
  
};
```

Exemple :

```
class Point {  
private:  
    int x, y;  
  
public:  
    ...  
    Point (Point &p) ;  
    ...  
};
```

Implémentation :

```
Point::Point (Point &p) {  
    this->x = p.getX();  
    this->y = p.getY();  
}
```

Exemple 2 : Objets dynamiques :

```
void exp15() {  
    Point *p1 = new Point(20, 30);  
    Point *p2 = p1;  
    p2->print();  
}
```

Dans ce deuxième cas, il s'agira d'un seul objet créé et pointé à l'aide de deux variables pointeurs « p1 » et « p2 » ➔ toute modification du contenu de l'un affectera automatiquement l'autre.

```
p2->setX(45);  
p1->print();
```

→

Point(45, 30)

4.4.2 Affectation Fille → Mère

En dehors des objets de même type, seules des affectations entre objets de classes reliées par héritage sont possibles.

Cas d'objets statiques :

```
// Mère <- Fille : Objets Statiques  
void exp16() {  
    ExtendedPoint p1(20, 30, 2);  
    Point p2 = p1;  
    p2.print();  
}
```

Dans ce cas, l'objet « Fille » (p1) sera tronqué automatiquement vers l'objet « Mère » (p2). Les caractéristiques « filles » sont alors automatiquement perdues. La méthode « print() » invoquée dans ce cas est celle de la classe mère « Point ».

Point(20, 30)

Cas d'objets dynamiques :

```
// Mère <- Fille : Objets dynamiques  
void exp17() {  
    ExtendedPoint *p1 = new ExtendedPoint(20, 30, 2);  
    Point *p2;  
    p2 = p1;  
    p2->print();  
}
```

Dans ce cas, l'objet « Fille » pointé par « p1 » conserve son identité et son contenu et ne sera aucunement tronqué. Le pointeur de type « Mère » (p2) va tout simplement pointer sur cet objet. Les caractéristiques « filles » sont dans ce cas toutes conservées.

4.4.3 Méthodes virtuelles

Dans l'exemple précédant, l'appel de la méthode « `print()` » aura été résolu lors de la compilation comme étant le « `print()` » de la mère (Point). Il s'agit d'une « **ligature statique** » par défaut (faisant aussi défaut au langage C++). L'appel à la méthode « `print()` » ferait alors référence au « `print()` » de la mère ; ce qui n'est pas normal dans ce cas puisque l'objet conserve bien son identité de « fille » (ExtendedPoint). Le résultat sera similaire au cas des objets statique :

Point(20, 30)

Si l'on cible une « **ligature dynamique** », ce qui devrait en principe être le cas, il faudrait déclarer la méthode « `print()` » dans la classe mère comme étant une « **méthode virtuelle** » par l'intermédiaire du mot clé « **Virtual** » :

```
class Point {  
private:  
    int x, y;  
public:  
    ...  
    virtual void print();  
    ...  
};
```

Avec cette modification, le résultat sera comme suit:

ExtendedPoint(20, 30, 2)

4.4.4 Affectation Mère → Fille

Cas d'objets statiques :

Il n'est pas possible d'affecter un objet « Mère » à un objet « Fille », et ce même avec utilisation du casting :

```
// Fille <- Mère : Objet Statique ==> Impossible
void expl8() {
    Point p1(20, 30);
    ExtendedPoint p2;
    p2 = (ExtendedPoint)p1;
    p2.print();
}
```

==> Erreur :

```
1> test.cpp(136) : error C2440: 'cast de type' : impossible de
convertir de 'Point' en 'ExtendedPoint'
```

Remarque 1 :

Il y a une solution cependant pour pouvoir affecter un objet « Mère » à un objet « Fille » ; il s'agit de **surcharger** l'opérateur « = » dans la fille pour accepter un objet mère en paramètre (voir la rubrique « **Surcharge des opérateurs** ». Ci-dessous la solution :

```
class ExtendedPoint : public Point {
private:
    int color;
public:
    ...
    void operator =(Point p);
    ...
};
```

Avec la définition suivante :

```
void ExtendedPoint::operator =(Point p) {  
    this->setX(p.getX());  
    this->setY(p.getY());  
    this->color = 0;  
}
```

L'exemple 18 pourra alors être repris comme suit :

```
void exp18_2() {  
    Point p1(20, 30);  
    ExtendedPoint p2;  
    p2 = p1;  
    p2.print();  
}
```

Remarque 2 :

Une autre solution consiste à définir un constructeur de copie pour la classe fille acceptant en paramètre un objet de la classe mère :

```
class ExtendedPoint : public Point {  
private:  
    int color;  
public:  
    ...  
    ExtendedPoint (Point &p);  
    ...  
};
```

Avec la définition suivante :

```
void ExtendedPoint::ExtendedPoint (Point &p) {  
    this->setX(p.getX());  
    this->setY(p.getY());  
    this->color = 0;  
}
```

L'exemple 18 pourra en fin être repris comme suit :

```
void exp18_3() {  
    Point p1(20, 30);  
    ExtendedPoint p2 = p1;  
    p2.print();  
}
```

Cas d'objets dynamique :

Concernant le cas des objets dynamiques, l'affectation d'un objet « mère » à un objet « fille » n'est possible que si on utilise le casting.

```
void exp19() {  
    Point *p1 = new Point(20, 30);  
    ExtendedPoint *p2;  
    p2 = (ExtendedPoint *)p1;  
    p2.print();  
}
```

Cependant, c'est une opération très dangereuse car l'espace mémoire d'une « mère » (dans notre exemple « Point ») sera pointé par un pointeur de type « fille » (dans notre exemple « ExtendedPoint »), qui aura ainsi la possibilité d'accéder à des membres de la classe fille qui ne sont en principe pas existants :

```
p2->getColor();
```

4.5 Polymorphisme

Le terme « polymorphisme » est tout d'abord un mot grec qui signifie « plusieurs formes ». Il s'agit d'un même service ou méthode ayant la capacité d'avoir différents comportements (plusieurs formes) selon les situations ou plus exactement selon le type des objets sur lesquels il est invoqué.

Le concept du polymorphisme nous permet de définir des tableaux ou des collections d'objets de types différents mais ayant une classe mère commune. Lors de l'exécution, l'identité réelle des objets sera déterminée automatiquement ce qui leur permettra d'agir convenablement concernant des appels de méthodes qui sont définies différemment dans les différentes classes manipulées. Les appels aux méthodes seront ainsi résolus dynamiquement et chaque objet de la collection aura le comportement adéquat. Pour obtenir ce résultat, les méthodes ainsi redéfinies doivent être déclarées virtuelles dans la classe de base.

```
void exp20() {
    Point *t[] = {
        new Point(20, 30),
        new ExtendedPoint(10, 15, 3),
        new Point(10, 78)
    };
    printf("-----\n");
    for(int i=0; i<3; i++) {
        t[i]->print();
    }
    printf("-----\n");
}
```

Chapitre 5

Surcharge des opérateurs

A l'origine, il y a la surcharge de l'opérateur « = » permettant de fournir à cet opérateur un comportement plus adéquat. En effet, comme signalé précédemment, une affectation entre objets statiques mettra en commun les propriétés dynamiques. Ceci nécessite une nouvelle définition de l'opérateur « = » dans la classe concernée.

Exemple :

```
class Point {  
private:  
    int x, y;  
public:  
    ...  
    void operator =(Point p);  
    ...  
};
```

Un exemple d'Implémentation :

```
void Point::operator =(Point p) {  
    this->x = p.getX() * 2;  
    this->y = p.getY() * 2;  
}
```

Exemple de code :

```
void exp22() {  
    Point p1(20, 30);  
    Point p2;  
    p2 = p1;  
    p2.print();  
}
```

En aura à l'exécution le résultat suivant :

Point(40, 60)

Tous les autres opérateurs du langage acceptent ainsi d'être surchargés. Dans l'exemple suivant on surcharge l'opérateur « + » pour pouvoir faire la somme de deux points :

Déclaration :

```
class Point {  
private:  
    int x, y;  
public:  
    ...  
    Point operator +(Point p);  
    ...  
};
```

Implémentation :

```
Point Point::operator +(Point p) {  
    Point s(this->x + p.getX(), this->y + p.getY());  
    return s;  
}
```

Exemple d'utilisation :

```
void exp23() {  
    Point p1(20, 30);  
    Point p2(25, 14);  
    Point p3;  
    p3 = p1 + p2;  
    p3.print();  
}
```