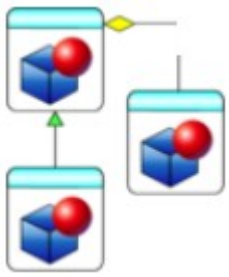
Programmation Orienté Objet

Table des matières

La programmation orientée objet (POO) est un paradigme de programmation inventé au début des années 1960. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique. Il possède une structure interne et un comportement. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues.



1. Introduction

À la différence de la programmation impérative, un programme écrit dans un langage objet répartit l'effort de résolution de problèmes sur un ensemble d'objets collaborant par envoi de messages. Chaque objet se décrit par un ensemble d'attributs (caractéristiques de l'objet) et un ensemble de méthodes portant sur ces attributs (fonctionnalités de l'objet).

La POO résulte de la prise de conscience des problèmes posés par l'industrie du logiciel ces dernières années en termes de complexité accrue et de stabilité dégradée. L'objet solutionne certains de ces problèmes à travers :

- l'encapsulation des attributs qui empêche toute modification externe accidentelle
- l'héritage qui permet la ré utilisabilité du code

Il est inconséquent d'opposer la programmation impérative à l'OO car, in fine, toute programmation des méthodes reste tributaire des mécanismes de programmation procédurale et structurée. On y rencontre des variables, des arguments, des boucles, des arguments de fonction, des instructions conditionnelles, tout ce que l'on trouve classiquement dans les boîtes à outils impératives.

2. Définitions

2.1. Classe

Une classe est la modélisation informatique d'un concept. Si dans la vie courante vous vous dite : ceci est un concept, une notion bien précise possédant plusieurs aspects, alors la représentation informatique de cette notion, le sera sous forme de classe. Une classe est l'équivalent d'un type de donnée (abstrait).

Une classe permet de définir les données relatives à une notion, ainsi que les actions qu'y s'y rapportent. Il sera possible, au sein d'une seule et même structure informatique, de regrouper le pendant informatique des données - des variables - et le pendant informatique des actions, à savoir des procédures et des fonctions. Les variables à l'intérieur d'une classe seront appelées *attributs*, et les procédures et les fonctions seront appelées *méthodes*. Les attributs et les méthodes devront posséder des noms utilisables en temps normal pour des variables ou des procédures/fonctions.

Prenons l'exemple de la télévision. Le concept général de télévision se rattache, comme toute notion, à des données ainsi que des actions. Les données sont par exemple le poids, la longueur de la diagonale de l'écran, les chaînes mémorisées ainsi que l'état allumé/veille/éteint de la télévision. Les actions peuvent consister à changer de chaîne, à allumer, éteindre ou mettre en veille la télévision. D'autres actions peuvent consister en un réglage des chaînes. Voici une représentation possible du concept de télévision :

Télévision
Poids longueur diagonale allumée ? Chaînes mémorisées
Éteindre

```
class television {
    float   poids;          //---- attributs
    int     diagonale;
    bool    estAllume;
    int     chaines[22];

    void eteindre() {       //---- méthodes
        estAllume = false;
```

allumer
mémoriser une chaîne

```
    }
void allumer() {
    estAllume = true;
    }
void memoriser(const int index, const int c) {
    chaines[index] = c;
    }
};
```

2.1. Objet

Nous avons vu précédemment que les classes étaient le pendant informatique des concepts. Les objets sont pour leur part le pendant informatique de la représentation réelle des concepts, à savoir des occurrences bien réelles de ce qui ne serait sans cela que des notions sans application possible. Une classe n'étant que la modélisation informatique d'un concept, elle ne se suffit pas à elle-même et ne permet pas de manipuler les occurrences de ce concept. Pour cela, on utilise les objets.

Pour manipuler une occurrence du concept modélisé par une classe, on utilise un *objet*. Un objet est dit d'*une classe* particulière, ou *instance* d'une classe donnée. En programmation objet, un objet ressemble à une variable et on pourrait dire en abusant un petit peu que son type serait une classe définie auparavant.

L'instanciation est l'action d'instancier, de créer un objet à partir d'un modèle. Elle est réalisée par la composition de deux opérations : l'allocation et l'initialisation. L'allocation consiste à réserver un espace mémoire au nouvel objet. L'initialisation consiste à fixer l'état du nouvel objet. Cette opération fait par exemple appel à l'un des constructeurs de la classe de l'objet à créer.

```
television t;// instanciation de la classe
```

Un objet est un élément variable possédant un exemplaire personnel de chaque attribut défini par sa classe. Il est impossible de lire ou d'écrire la valeur d'un attribut d'une classe, mais il devient possible de lire ou d'écrire la valeur d'un attribut d'un objet. Le fait de modifier la valeur d'un attribut pour un objet ne modifie pas la valeur du même attribut d'un autre objet de même classe.

Les méthodes sont partagées par tous les objets d'une même classe, mais une méthode peut être appliquée non pas à la classe qui la définit, mais à un objet de cette classe. Les attributs manipulés par la méthode sont ceux de l'objet qui a reçu l'appel de méthode. Ceci permet d'avoir pour chaque objet une copie des attributs avec des valeurs personnalisées, et des méthodes s'appliquant à ces attributs uniquement. C'est pour cela qu'on introduit souvent la programmation objet en présentant le besoin de regrouper données et instructions dans une même structure, à savoir l'objet.

Il est possible de manipuler un nombre quelconque d'objets d'une classe particulière, mais un objet est toujours d'une seule et unique classe. Les attributs et les méthodes d'un objet sont ses membres et on notera les membres d'un objet Objet.Membre de façon à lever toute ambiguïté quant au propriétaire du membre considéré.

Voici un tableau vous montrant les 3 facettes d'un même élément dans les 3 mondes que vous connaissez désormais : le vôtre, celui de la programmation objet, et celui de la programmation impérative (sans objets). Vous pouvez y voir la plus forte lacune de la programmation impérative, à savoir qu'un concept ne peut pas y être modélisé directement.

Monde réel	Programmation objet	Programmation impérative
Concept	Classe	(rien)
Donnée	attributs	Variable
Action	Méthode	Procédure Fonction

3. Les trois fondamentaux

La POO est dirigée par trois fondamentaux qu'il convient de toujours garder à l'esprit : encapsulation, héritage et polymorphisme.

3.1. Encapsulation

Derrière ce terme se cache le concept même de l'objet : réunir sous la même entité les données et les moyens de les gérer, à savoir les attributs et les méthodes.

L'encapsulation introduit donc une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.

Sous ce nouveau concept se cache également un autre élément à prendre en compte : pouvoir masquer aux yeux d'un programmeur extérieur tous les rouages d'un objet et donc l'ensemble des procédures et fonctions destinées à la gestion interne de l'objet, auxquelles le programmeur final n'aura pas à avoir accès. L'encapsulation permet donc de masquer un certain nombre de attributs et méthodes tout en laissant visibles d'autres attributs et méthodes.

L'encapsulation permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

3.1.1. Attributs et méthodes publics

Comme leur nom l'indique, les attributs et méthodes dits publics sont accessibles depuis tous les descendants et dans tous les modules.

On peut considérer que les éléments publics n'ont pas de restriction particulière.

```
class appareilEcran {
    public:  // attributs accessibles depuis l'extérieur
    string  nom;
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:
    appareilEcran(const float p, const int d) {
        poids = p;
        diagonale = d;
        estAllume = false;
        }
    };
```

```
class television : public appareilEcran {
    int     chaines[22];

    public:
    television(const float poids = 3.1, const int diagonale = 75) :
       appareilEcran(poids, diagonale) {}
    };

int main()
{
    television t;          // instanciation de la classe
    t.diagonale = 60;      // la diagonale de l'écran est de 60 cm
}
```

Les méthodes publiques sont communément appelées accesseurs : elles permettent d'accéder aux attributs d'ordre privé.

Il existe des accesseurs en lecture, destinés à récupérer la valeur d'un attribut, et des accesseurs en écriture destinés pour leur part à la modification d'un attribut.

Il n'est pas nécessaire d'avoir un accesseur par attribut privé, car ceux-ci peuvent n'être utilisés qu'à des fins internes.

Très souvent, les accesseurs en lecture verront leur nom commencer par get quand leurs homologues en écriture verront le leur commencer par set.

Les constructeurs et les destructeurs éventuels d'un objet devront bénéficier de la visibilité publique, sans quoi un programme externe ne pourrait pas les appeler !

Attention !

Un attribut ne devra être public que si sa modification n'entraîne pas de changement dans le comportement de l'objet. Dans le cas contraire, il faut passer par une méthode. Modifier un attribut "manuellement" et ensuite appeler une méthode pour informer de cette modification est une violation du principe d'encapsulation.

3.1.2. Attributs et méthodes privés

La visibilité privée restreint la portée d'un attribut ou d'une méthode au module où il ou elle est déclaré(e). Ainsi, si un objet est déclaré dans une unité avec un attribut privé, alors cet attribut ne pourra être accédé qu'à l'intérieur même de l'unité.

Cette visibilité est à bien considérer. En effet, si un descendant doit pouvoir accéder à un attribut ou une méthode privé(e), alors ce descendant doit nécessairement être déclaré dans le même module que son ancêtre.

Généralement, les accesseurs, autrement dit les méthodes destinées à modifier les attributs, sont déclarés comme privés.

```
class appareilEcran {
    private:   // par défaut les attributs et méthodes sont déclarés « privé »
    string  nom;
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:
    appareilEcran(const float p, const int d) {
```

```
        poids = p;
        diagonale = d;
        estAllume = false;
        }
    const int setDiagonale(const int d) {
        if ( d < 25 || d > 180 )
            return 1;   // erreur de taille écran
        else
            diagonale = d;
        return 0;   // c'est OK
        }
    };

class television : public appareilEcran {
    int     chaines[22];

    public:
    television(const float poids = 3.1, const int diagonale = 75) :
      appareilEcran(poids, diagonale) {}
    };

int main()
{
    television t;         // instanciation de la classe
    t.setDiagonale(36);
}
```

3.1.3. Attributs et méthodes protégés

La visibilité protégé correspond à la visibilité privé excepté que tout attribut ou méthode protégé(e) est accessible dans tous les descendants, quel que soit le module où ils se situent.

Cette visibilité est souvent à préférer à la visibilité privée.

```
class appareilEcran {
    protected:      // attributs uniquement accessibles aux classes filles
    string  nom;
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:
    appareilEcran(const float p, const int d) {
        poids = p;
        diagonale = d;
        estAllume = false;
        }
    const int getDiagonale() const { return diagonale; }   //---- accesseur lecture
    };

class television : public appareilEcran {
    int     chaines[22];

    public:
    television(const float poids = 3.1, const int diagonale = 75) :
      appareilEcran(poids, diagonale) {}
```

```
    const int setDiagonale(const int d) {                  //---- accesseur écriture
        if ( d < 45 || d > 180 )
            return 1;   // erreur de taille écran
        else
            diagonale = d;   // attribut accessible depuis la classe fille uniquement
        return 0;   // c'est OK
        }
    };

int main()
{
    television t;          // instanciation de la classe
    if ( t.setDiagonale(6) )     // cette modification ne sera pas prise en compte...
        cout << "Erreur taille écran : " << t.getDiagonale() << " cm" << endl;
}
```

3.2. Héritage

Cette notion est celle qui s'explique le mieux au travers d'un exemple. Considérons des écrans d'ordinateur : ils sont également caractérisés par leur poids, leur diagonale, mais là où la télévision s'intéresse aux chaînes, l'écran d'ordinateur possède plutôt une liste de résolutions d'écran possibles, ainsi qu'une résolution actuelle. Un tel écran peut en outre être allumé, éteint ou en veille et doit pouvoir passer d'un de ces modes aux autres par des actions. En informatique, le concept d'écran d'ordinateur pourra être représenté par une classe définissant par exemple un réel destiné à enregistrer le poids, un entier destiné à mémoriser la diagonale en centimètres. Une méthode (donc une action transcrite au niveau informatique) permettra de l'allumer et une autre de l'éteindre, par exemple.

Si l'on s'en tient à ce qui vient d'être dit, et en adoptant la même représentation simple et classique que précédemment pour les concepts, nous aurons :

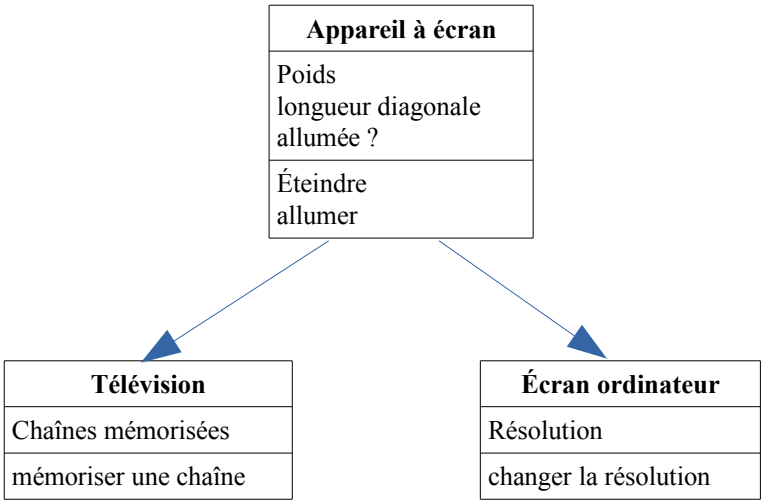
Télévision
Poids longueur diagonale allumée ? Chaînes mémorisées
Éteindre allumer mémoriser une chaîne

Écran ordinateur
Poids longueur diagonale allumée ? Résolution
Éteindre allumer changer la résolution

Un appareil à écran possède un poids, une longueur de diagonale et il est possible de l'éteindre et de l'allumer. Une télévision est un tel appareil, possédant également une liste de chaînes mémorisées, une action permettant de mémoriser une chaîne, ainsi que d'autres non précisées ici qui font d'un appareil à écran... une télévision. Un écran d'ordinateur est également un appareil à écran, mais possédant une résolution actuelle, une liste de résolutions possibles, et une action permettant de changer la résolution actuelle. La télévision d'un coté, et l'écran d'ordinateur de l'autre sont deux concepts basés sur celui de l'appareil à écran, et lui ajoutant divers éléments qui le rendent particulier.

Cette manière de voir les choses est l'un des fondements de la programmation objet qui permet (et se base) sur ce genre de regroupements permettant une réutilisation d'éléments déjà programmés dans un cadre général. Voici le diagramme précédent, modifié pour faire apparaître ce que l'on obtient par regroupement des informations et actions re-

dondantes des deux concepts (un nouveau concept intermédiaire, appelé "appareil à écran" a été introduit à cette fin) :



La programmation objet accorde une grande importance à ce genre de schémas : lorsqu'il sera question d'écrire un logiciel utilisant les capacités de la programmation objet, il faudra d'abord *modéliser* les concepts à manipuler sous forme de diagramme de concepts, puis traduire les diagrammes de concepts en diagrammes de classes.

L'intérêt de ce genre de représentation est qu'on a regroupé dans un seul concept plus général ce qui était commun à deux concepts proches l'un de l'autre. On a ainsi un concept de base très général, et deux concepts un peu plus particuliers s'appuyant sur ce concept plus général mais partant chacun d'un coté en le spécialisant. Lorsqu'il s'agira d'écrire du code source à partir des concepts, il ne faudra écrire les éléments communs qu'une seule fois au lieu de deux (ou plus), ce qui facilitera leur mise à jour : Il n'y aura plus de risque de modifier par exemple le code source réalisant l'allumage d'une télévision sans modifier également celui d'un écran d'ordinateur puisque les deux codes source seront en fait un seul et même morceau de code général applicable à n'importe quel appareil à écran.

```
// déclaration de la classe mère appareilEcran
class appareilEcran {
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:
    const void eteindre() {
        estAllume = false;
        }
    const void allumer() {
        estAllume = true;
        }
    };

// déclaration de la classe fille television qui hérite de appareilEcran
class television : public appareilEcran {
    int     chaines[22];

    public:
```

```
    const void memoriserChaine(const int index, const int c) {
        chaines[index] = c;
        }
    };

// déclaration de la classe fille ecranOrdinateur qui hérite de appareilEcran
class ecranOrdinateur : public  appareilEcran {
    int     resolution;

    public:
    const void changerResolution(const int r) {
        resolution = r;
        }
    };
```

3.3. Polymorphisme

Afin de mieux cerner cette notion, il suffit d'analyser la structure du mot : poly comme plusieurs et morphisme comme forme. Le polymorphisme traite de la capacité de l'objet à posséder plusieurs formes.

Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, un objet va hériter des attributs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire, ou de la compléter.

On voit donc apparaître ici ce concept de polymorphisme : choisir en fonction des besoins quelle méthode ancêtre appeler, et ce au cours même de l'exécution. Le comportement de l'objet devient donc modifiable à volonté.

Le concept de polymorphisme ne doit pas être confondu avec celui d'héritage multiple. En effet, l'héritage multiple permet à un objet d'hériter des membres (attributs et méthodes) de plusieurs objets à la fois, alors que le polymorphisme réside dans la capacité d'un objet à modifier son comportement propre et celui de ses descendants au cours de l'exécution.

Attention à ne pas confondre surcharge et polymorphisme. La surcharge consiste à définir des méthodes qui portent des noms identiques au sein d'une classe : c'est un polymorphisme statique. Alors que le polymorphisme dynamique porte sur les objets eux mêmes et s'effectue à l'exécution.

3.3.1. Polymorphisme statique : surcharge

3.3.1.1. Surcharge de méthodes

La plupart du temps, lorsque l'on surcharge une méthode, le but n'est pas d'écraser l'ancienne, mais de la compléter de façon à apporter de nouvelles fonctionnalités. Il est donc nécessaire de pouvoir appeler la méthode ancêtre.

```
class appareilEcran {
    string  nom;
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:  // surcharge du constructeur
    appareilEcran(const float p, const int d) {
        poids = p;
```

```
        diagonale = d;
        estAllume = false;
        }
    appareilEcran(const char *n) {
        nom = n;
        estAllume = false;
        }
    };

class television : public appareilEcran {
    int     chaines[22];

    public:  // appel au 1er constructeur avec 2 paramètres par défaut
    television(const float poids = 3.1, const int diagonale = 75) :
       appareilEcran(poids, diagonale) {}
    };

class ecranOrdinateur : public  appareilEcran {
    int     resolution;

    public:  // appel aux 2 constructeurs surchargés de la classe mère
    ecranOrdinateur(const float poids = 1.4, const int diagonale = 45) :
       appareilEcran(poids, diagonale) {}
    ecranOrdinateur(const char *nom) :
       appareilEcran(nom) {}
    };

int main()
{
    television      t;          // instanciation de la classe
    television      tv(2.9);    // instanciation de la classe
    ecranOrdinateur o("ACER");  // instanciation de la classe
}
```

Il n'est pas nécessaire de surcharger ou de redéfinir une méthode ! Ainsi, si un objet ne surcharge pas une méthode, c'est celle du premier ancêtre la définissant ou la surchargeant qui sera appelée.

De fait, il n'est pas nécessaire pour un objet de réécrire un constructeur (ou un destructeur) si celui de son ancêtre suffit à son initialisation.

3.3.1.2. Surcharge d'opérateurs

La surcharge d'opérateur permet aux opérateurs du C++ d'avoir une signification spécifique quand ils sont appliqués à des types spécifiques. Surcharger les opérateurs standards permet de tirer parti de l'intuition des utilisateurs de la classe. L'utilisateur va en effet pouvoir écrire son code en s'exprimant dans le langage du domaine plutôt que dans celui de la machine.

```
class complex {
    float   re;
    float   im;

    public:
    complex(const float r, const float i) {
        re = r;
        im = i;
```

```
        }
    ~complex() {}
    const float real() const { return re; }
    const float imag() const { return im; }
    complex& operator = (const complex& c) {        // opérateur d'affectation
        re = c.real();
        im = c.imag();
        return *this;
        }
    complex& operator += (const complex& c) {
        re += c.real();
        im += c.imag();
        return *this;
        }
    complex operator + (const complex& x) const {    // opérateur d'addition
        complex c = *this;
        c += x;
        return c;
        }
    };

int main()
{
    complex a(1, 2);
    complex b(3.7, 4);
    complex c = a + b;    // addition et affectation

    return 0;
}
```

3.3.2. Polymorphisme dynamique

Le code ci-dessous donne un exemple de résolution dynamique des liens au moment de l'exécution à l'aide de méthodes virtuelles (voir § 5) et de références.

```
class appareilEcran {
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:
    appareilEcran(const float p, const int d) {
        poids = p;
        diagonale = d;
        estAllume = false;
        }
    virtual const void affiche() const = 0;
    };

class television : public appareilEcran {
    int     chaines[22];

    public:
    television(const float poids = 3.1, const int diagonale = 75) :
        appareilEcran(poids, diagonale) {}
    virtual const void affiche() const {
        cout << "ceci est une tv" << endl;
```

```
        }
    };

class ecranOrdinateur : public  appareilEcran {
    int     resolution;

    public:
    ecranOrdinateur(const float poids = 1.4, const int diagonale = 45) :
        appareilEcran(poids, diagonale) {}
    virtual const void affiche() const {
        cout << "ceci est un ordinateur" << endl;
        }
    };

int main()
{
    appareilEcran   *t, *o;             // instanciation de la classe
    t = new television;
    o = new ecranOrdinateur;

    t->affiche();   // affiche "ceci est un ordinateur"

    delete t;
    delete o;

    return 0;
}
```

4. Constructeurs et destructeurs

4.1. Constructeurs

Comme leur nom l'indique, les constructeurs servent à construire l'objet en mémoire. Un constructeur va donc se charger de mettre en place les données, d'associer les méthodes avec les attributs et de créer le diagramme d'héritage de l'objet, autrement dit de mettre en place toutes les liaisons entre les ancêtres et les descendants.

S'il peut exister en mémoire plusieurs instances d'un même type objet, autrement dit plusieurs variables du même type, seule une copie des méthodes est conservée en mémoire, de sorte que chaque instance se réfère à la même zone mémoire en ce qui concerne les méthodes. Bien entendu, les attributs sont distincts d'un objet à un autre. De fait, seules les données diffèrent d'une instance à une autre.

```
class appareilEcran {
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:
    appareilEcran(const float p = 1.2, const int d = 35) {
        poids = p;
        diagonale = d;
        estAllume = false;
        }
    };
```

Certaines remarques sont à prendre en considération concernant les constructeurs :

- Un objet peut ne pas avoir de constructeur explicite. Dans ce cas, c'est le compilateur qui se charge de créer de manière statique les liens entre attributs et méthodes.
- Un objet peut avoir plusieurs constructeurs : c'est l'utilisateur qui décidera du constructeur à appeler. La présence de constructeurs multiples peut sembler saugrenue de prime abord, leur rôle étant identique. Cependant, comme pour toute méthode, un constructeur peut être surchargé, et donc effectuer diverses actions en plus de la construction même de l'objet. On utilise ainsi généralement les constructeurs pour initialiser les attributs de l'objet. À différentes initialisations peuvent donc correspondre différents constructeurs.
- S'il n'est pas nécessaire de fournir un constructeur pour un objet statique, il devient obligatoire en cas de gestion dynamique, car le diagramme d'héritage ne peut être construit de manière correcte que lors de l'exécution, et non lors de la compilation.

4.2. Destructeurs

Le destructeur est le pendant du constructeur : il se charge de détruire l'instance de l'objet. La mémoire allouée pour le diagramme d'héritage est libérée. Certains compilateurs peuvent également se servir des destructeurs pour éliminer de la mémoire le code correspondant aux méthodes d'un type d'objet si plus aucune instance de cet objet ne réside en mémoire.

```
class television : public appareilEcran {
    int     *chaines;

    public:
    television(const float poids = 3.1, const int diagonale = 75) :
       appareilEcran(poids, diagonale) {
         chaines = NULL;
         }
    ~television() {
        // libération de la mémoire si nécessaire à la destruction de l'objet
        if ( chaines )
            delete [] chaines;
        }
    const void setChaines(const int size) {
        if ( chaines == NULL )
            chaines = new int(size);
        }
    };
```

Là encore, différentes remarques doivent être gardées à l'esprit :

- Tout comme pour les constructeurs, un objet peut ne pas avoir de destructeur. Une fois encore, c'est le compilateur qui se chargera de la destruction statique de l'objet.
- Un objet peut posséder plusieurs destructeurs. Leur rôle commun reste identique, mais peut s'y ajouter la destruction de certaines variables internes pouvant différer d'un destructeur à l'autre. La plupart du temps, à un constructeur distinct est associé un destructeur distinct.

- En cas d'utilisation dynamique, un destructeur s'impose pour détruire le diagramme créé par le constructeur.

5. Méthodes virtuelles et abstraites

5.1. Méthodes virtuelles

Une méthode dite virtuelle est une méthode dont la résolution des liens est effectuée dynamiquement.

Toute méthode est susceptible d'être surchargée dans un descendant, de manière à être écrasée ou complétée. Par conséquent, toute méthode surchargée donne lieu à création d'une nouvelle section de code, et donc à une nouvelle adresse en mémoire.

De plus, tout objet possède un lien vers la table des méthodes de ses ancêtres : le diagramme d'héritage. De fait, tout type objet est directement lié à ses types ancêtres. Autrement dit, si nous reprenons l'exemple du début, l'objet Maison peut être assimilé à un Bâtiment.

Considérons à présent la méthode Ouvrir d'un Bâtiment. Celle-ci consiste à ouvrir la porte principale.

À présent, surchargeons cette méthode pour l'objet Maison, de sorte que la méthode Ouvrir non seulement ouvre la porte principale, mais également les volets de notre Maison.

Déclarons maintenant une instance statique de Bâtiment, et appelons cette méthode Ouvrir. Lors de la création de l'exécutable, le compilateur va vérifier le type d'instance créé. Le compilateur lie alors notre appel à celui de Bâtiment.Ouvrir (la méthode Ouvrir de l'objet Bâtiment), en toute logique. Il ne se pose aucun problème.

Considérons à présent un autre exemple : déclarons une variable dynamique destinée, en principe, à recevoir un objet Bâtiment. Comme nous l'avons vu juste avant, l'objet Maison est compatible avec l'objet Bâtiment. Comme nous travaillons en dynamique, nous nous servons de pointeurs. De fait, on peut très bien décider, avec cette variable pointant vers un objet Bâtiment, de déclarer une instance de type Maison : le compilateur ne montrera aucune réticence.

Si nous résumons, nous avons donc une variable de type officiel pointeur vers Bâtiment et contenant en réalité une Maison.

Appelons alors notre méthode Ouvrir. Comme nous avons une Maison, il faut que l'on ouvre les volets. Or, si nous exécutons notre programme, les volets resteront clos. Que s'est-il passé ?

Lors de la création du programme, le compilateur s'est arrêté sur notre appel à Ouvrir. Ayant déclaré un objet Bâtiment, le compilateur ignore tout du comportement du programme lors de son exécution, et par conséquent ignore que la variable de type pointeur vers Bâtiment contiendra à l'exécution un objet Maison. De fait, il effectue une liaison vers Bâtiment.Ouvrir alors que nous utilisons une Maison !

La solution, réside dans l'utilisation des méthodes virtuelles. Grâce à celles-ci, la résolution des liens est effectuée dynamiquement, autrement dit lors de l'exécution. Ainsi, si nous déclarons notre méthode Ouvrir comme virtuelle, lors de la création du programme, le compilateur n'effectuera aucune liaison statique avant notre appel. Ce n'est que lors de

l'exécution, au moment de l'appel, que la liaison va s'effectuer. Ainsi, au moment où l'on désirera appeler Ouvrir, notre programme va interroger son pointeur interne pour déterminer son type. Bien évidemment, cette fois-ci, il va détecter une instance de Maison, et l'appel se fera donc en direction de Maison.Ouvrir. Les volets s'ouvrent...

Attention !

Les constructeurs des objets ne seront jamais déclarés comme virtuels, car c'est toujours le bon constructeur qui est appelé. Le caractère virtuel est donc inutile et sera même signalé comme une erreur par le compilateur.

Par contre, les destructeurs seront toujours déclarés comme virtuels car souvent surchargés.

Il n'en est pas de même pour les classes qui elles peuvent s'appuyer sur le principe de constructeur virtuel.

5.2. Méthodes abstraites

Une méthode abstraite est une méthode qu'il est nécessaire de surcharger. Elle ne possède donc pas d'implémentation. Ainsi, si on tente d'appeler une méthode abstraite, alors une erreur est déclenchée.

Bien entendu, il convient lors de la surcharge d'une telle méthode de ne pas faire appel à la méthode de l'ancêtre...

L'utilité de ce genre de procédé ne saute pas immédiatement aux yeux, mais imaginez que vous deviez concevoir une classe qui serve uniquement de base à ses classes dérivées, sans avoir un quelconque intérêt en tant que classe isolée. Dans ce cas, vous devriez déclarer certaines méthodes comme abstraites et chaque classe dérivant directement de cette classe se devra d'implémenter la méthode afin de pouvoir être instanciée. C'est un bon moyen, en fait, pour que chaque classe descendante d'une même classe de base choisisse son implémentation indépendante des autres. Cette implémentation n'est pas une option, c'est une obligation pour que la classe dérivée puisse être instanciée.

Les méthodes abstraites sont généralement utilisées lorsque l'on bâtit un squelette d'objet devant donner lieu à de multiples descendants devant tous posséder un comportement analogue.

```
class appareilEcran {
    protected:
    float   poids;
    int     diagonale;
    bool    estAllume;

    public:
    appareilEcran(const float p, const int d) {
        poids = p;
        diagonale = d;
        estAllume = false;
        }
    virtual const int setDiagonale(const int) = 0;  //---- méthode abstraite
    };

class television : public appareilEcran {
    int     chaines[22];
```

```
    public:  // appel au 1er constructeur avec 2 paramètres par défaut
    television(const float poids = 3.1, const int diagonale = 75) :
        appareilEcran(poids, diagonale) {}
    const int setDiagonale(const int d) {   //---- taille écran [45 ; 180] cm
        if ( d < 45 || d > 180 )
            return 1;
        else
            diagonale = d;
        return 0;   // c'est OK
        }
    };

class ecranOrdinateur : public  appareilEcran {
    int     resolution;

    public:  // appel aux 2 constructeurs surchargés de la classe mère
    ecranOrdinateur(const float poids = 1.4, const int diagonale = 45) :
        appareilEcran(poids, diagonale) {}
    ecranOrdinateur(const char *nom) :
        appareilEcran(nom) {}
    const int setDiagonale(const int d) {   //---- taille écran [25 ; 70] cm
        if ( d < 25 || d > 70 )
            return 1;
        else
            diagonale = d;
        return 0;   // c'est OK
        }
    };
```

6. Du langage C au langage C++

Créer de nouveaux types de variables devient indispensable quand on cherche à faire des programmes plus complexes. Le langage C permet de créer nos propres types de variables mais n'offre pas nativement des fonctionnalités comme l'encapsulation ou l'héritage.

6.1. Définir une structure

Une structure est un assemblage de variables qui peuvent avoir différents types. Contrairement aux tableaux qui obligent à utiliser le même type dans tout le tableau, on peut créer une structure comportant des variables de types long, char, int et double à la fois.

Une définition de structure commence par le mot-clé struct, suivi du nom de la structure.

```
struct complex {
    float   re;
    float   im;
    };
```

6.2. Accéder aux variables de la structure

Pour accéder donc à chaque membre de la structure, il faut écrire :

variable.nomDeLaStructure

Le point fait la séparation entre la variable et la structure.

```c
int main()
{
	struct complex c;   // déclaration de la variable complex
	c.re = 1;           // initialisation des membres de la structure
	c.im = 2;

	printf("c = %.2lf + %.2lfj\n", c.re, c.im);

	return 0;
}
```

Affichera :

```
c = 1.00 + 2.00j
```

6.3. Initialiser une structure

L'initialisation d'une structure ressemble à celle d'un tableau. En effet, on peut le faire à la déclaration de la variable :

```c
struct complex c = {1, 2};
```

Cela définira, dans l'ordre, c.re = 1 et c.im = 2.

6.4. Pointeur de structure

Un pointeur de structure se crée de la même manière qu'un pointeur de n'importe quelle autre type de base :

```c
struct complex *ptr = NULL;
```

On a ainsi un pointeur de complex appelé ptr.

On initialise le pointeur en donnant l'adresse d'une structure. On accède aux membres du pointeur par variable->nomDeLaStructure, équivalent à écrire (*variable).nomDeLaStructure.

La flèche est réservée aux pointeurs, le « point » est réservé aux variables.

```c
int main()
{
	struct complex c = {1, 2};
	struct complex *ptr = &c; // initialisation du pointeur par adresse

	printf("c = %.2lf + %.2lfj\n", ptr->re, ptr->im);

	return 0;
}
```

A travers ces exemples, on peut appréhender la similitude d'écriture entre les structures en langage C et les classes en C++.

Bjarne Stroustrup a développé C++ au cours des années 1980, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T. Il s'agissait d'améliorer le langage C pour en faire un langage orienté objet. Il l'avait d'ailleurs nommé C with classes (« C avec des classes »). En 1983, le nom du langage passa de C avec classes à celui de « C++ ».