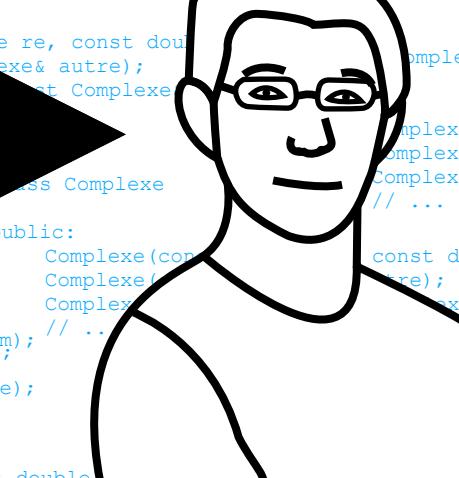




INTRODUCTION À LA PROGRAMMATION ORIENTÉE OBJET (EN C++)

A cartoon illustration of Steve Jobs, wearing his signature glasses and a white t-shirt, looking slightly to the right. A large black play button icon is positioned to the left of his head.

```
Complexe(const double re, const double im);
Complexe(const Complexe& autre);
Complexe operator+=(const Complexe& autre);
// ...

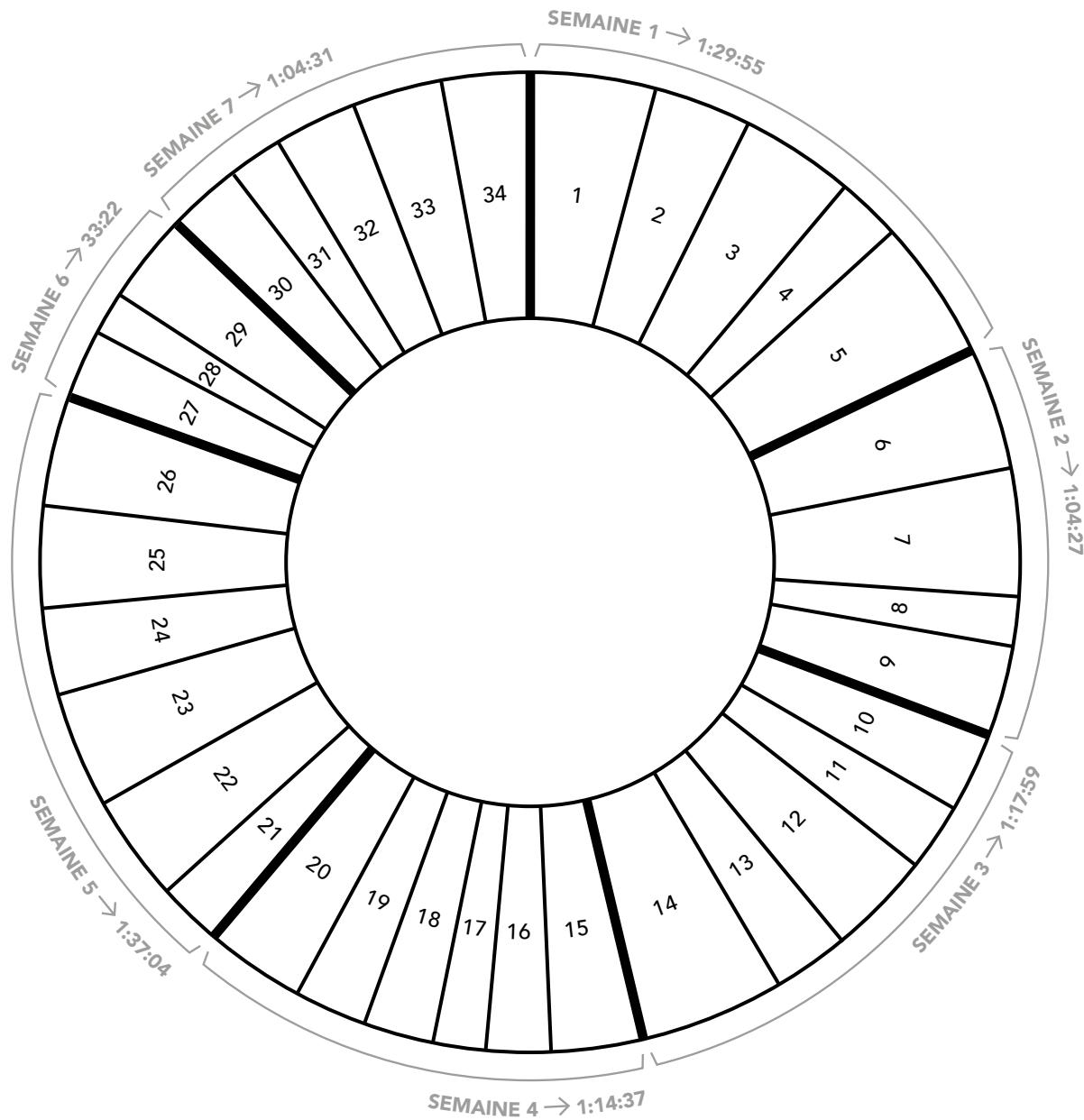
double re, const dou
Complexe& autre);
r+= const Complexe
Complexe(cons
Complexe(cons
Complexe ope
// ...

class Complexe
{
public:
    Complexe(con
    Complexe(
    Complexe(
        le im); // ...
    );
    autre);

    const double
    autre);
    Complexe& aut
```

A black and white line drawing of a man and a woman. The man is on the left, facing forward, wearing a light-colored button-down shirt. The woman is on the right, slightly behind him, facing forward, wearing a dark top and a necklace, with her hair styled in two pigtails. They are positioned in front of a large block of blue text representing C++ code.

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit





CONTENU

SEMAINE 1: BASES DE POO

1. Introduction	4
2. Classes, objets, attributs et méthodes en C++	6
3. Public: et private:	8
4. Encapsulation et abstraction: résumé	10
5. Encapsulation et abstraction: étude de cas	13

SEMAINE 2: CONSTRUCTEURS ET DESTRUCTEURS

6. Constructeurs (introduction)	16
7. Constructeurs par défaut en C++	18
8. Constructeur de copie	21
9. Destructeurs	22

SEMAINE 3: SURCHARGE DES OPÉRATEURS

10. Variables et méthodes de classe	24
11. Surcharge d'opérateurs: introduction	26
12. Surcharge d'opérateurs: surcharge externe	28
13. Surcharge d'opérateurs: surcharge interne	31
14. Surcharge d'opérateurs: compléments	33

SEMAINE 4: HÉRITAGE

15. Héritage: concepts	37
16. Héritage: droit protégé	39
17. Héritage: masquage	40
18. Héritage: constructeurs (1/2)	41
19. Héritage: constructeurs (2/2)	43
20. Copie profonde	45

SEMAINE 5: POLYMORPHISME

21. Polymorphisme et résolution dynamique des liens	47
22. Polymorphisme: méthodes virtuelles	49
23. Masquage, substitution et surcharge	52
24. Classes abstraites	55
25. Collections hétérogènes	57
26. Collections hétérogènes: compléments avancés	59

SEMAINE 6: HÉRITAGE MULTIPLE

27. Héritage multiple: concept et constructeurs	62
28. Héritage multiple: masquage	65
29. Classes virtuelles	67

SEMAINE 7: ÉTUDE DE CAS

30. Étude de cas: présentation et modélisation du problème	69
31. Étude de cas: affichage polymorphe	72
32. Étude de cas: surcharge d'opérateur et première version	74
33. Étude de cas: modélisation des mécanismes	78
34. Étude de cas: copie profonde	82



1. INTRODUCTION

PROGRAMMATION ORIENTÉE OBJET

Dans la programmation dite procédurale ou impérative, les données et les traitements apparaissent comme des entités séparées dans un programme. Par exemple, dans un programme qui permet de calculer la surface d'un rectangle, un style procédural déclarerait deux variables `largeur` et `hauteur`, pour ensuite créer une fonction `surface()` prenant ces variables comme arguments (fig. 1). Il n'existe donc pas de lien direct entre les données, stockées ici dans les variables, et le traitement, réalisé par une fonction. De plus, l'absence de lien sémantique entre les entités de ce genre de programme pose problème. Le lien qui devrait notamment unir la largeur et la hauteur, à savoir qu'il s'agit de paramètres d'un même rectangle, est difficile à établir. De même, en dehors du nom des entités, il n'est pas évident que notre fonction `surface()` réalise un calcul de surface de rectangle et non un produit quelconque.

```
double surface(double largeur, double hauteur);

int main()
{
    double largeur(3.0);
    double hauteur(4.0);

    cout << "La surface du rectangle est : "
        << surface(largeur, hauteur) << endl;

    return 0;
}

double surface(double largeur, double hauteur)
{
    return largeur * hauteur;
}
```

FIGURE 1

1:50

20:48

Exemple de programmation procédurale.

La **programmation orientée objet** permet le regroupement des traitements et des données en une seule et même entité. Dans l'exemple précédent, on préférera regrouper en une seule et même entité la notion de rectangle avec ses données caractéristiques ainsi que les traitements qui lui sont spécifiques, comme dans le deuxième code de la figure 2. Un intérêt fondamental de l'orienté objet est donc d'assurer une meilleure lisibilité et une meilleure cohérence au programme, puisque l'on crée un lien explicite entre données et traitements. Ce style, qui n'est pas spécifique au langage C++, donne davantage de robustesse, de modularité et de lisibilité aux programmes. Il repose sur quatre concepts centraux: l'encapsulation, l'abstraction, l'héritage et le polymorphisme.

```
double largeur(3.0);
double hauteur(4.0);

cout << "Surface : "
    << surface(largeur, hauteur)
    << endl;
```

```
Rectangle rect(3.0, 4.0);
cout << "Surface : "
    << rect.surface()
    << endl;
```

FIGURE 2

15:30

20:48

Comparaison entre programmation procédurale et orientée objet, plus lisible.

ENCAPSULATION ET ABSTRACTION

L'encapsulation consiste à regrouper dans une seule et même entité des données et des traitements qui agissent sur celles-ci. On désignera ces données par des **attributs** et les traitements par des **méthodes**. Le tout sera défini au sein d'une **classe**, qui constituera un nouveau type de données, une catégorie d'**objets**. Une variable de ce type sera aussi désignée par le terme d'**instance**.



Le processus d'abstraction identifie des caractéristiques et des mécanismes communs aux entités utilisées, permettant une description générique, abstraite de l'ensemble. Dans notre exemple, si nous désirons utiliser plusieurs rectangles, tous seraient caractérisés par une largeur et une hauteur auxquelles on pourrait attacher un même calcul de surface. Dans une approche procédurale, on aurait plutôt tendance à déclarer séparément chacun d'entre eux, ce qui serait fastidieux et source d'erreur.

L'encapsulation et l'abstraction consistent également à dissimuler des détails d'implémentation à l'utilisateur, qui n'exploitera plus que l'**interface d'utilisation** de l'objet, c'est-à-dire les diverses fonctionnalités qui lui sont accessibles pour sa manipulation. Il y a, en effet, deux niveaux de perception d'un objet: le niveau externe, visible, du programmeur utilisateur, celui qui utilise l'objet. Il s'agirait par exemple de la personne qui veut connaître la surface d'un rectangle. Il n'a pas besoin de connaître les détails techniques des calculs, tout comme nous n'avons pas besoin de savoir comment un moteur est construit pour pouvoir utiliser une voiture. Le second niveau est le niveau interne, du programmeur qui se charge des détails d'implémentation, comme définir comment se fait le calcul de surface.

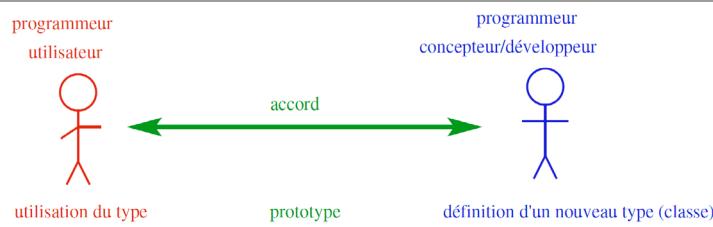


FIGURE 3

14:40

20:48

Les différentes facettes d'une classe.

Le programmeur concepteur définit différents **niveaux de visibilité** et d'accèsibilité de la classe, ce qui donne une grande robustesse au programme face aux changements et aux erreurs de manipulation. Par exemple, même si le moteur change selon le modèle des voitures, l'interface reste la même: l'action de conduire n'en est pas affectée. En orientée objet, toute modification de la structure interne d'un objet reste invisible de l'extérieur. De plus, l'accès limité de l'utilisateur à l'objet définit un cadre d'utilisation plus rigoureux, qui protège le programme des erreurs. On considère donc toujours les attributs comme des détails d'implémentation inaccessibles puisqu'ils reposent sur des choix techniques de modélisation et doivent être manipulés avec attention: l'interface se limite à quelques méthodes bien choisies.

Pour résumer, lorsque l'on définit un nouveau type d'objet au travers d'une classe, on définit des attributs caractéristiques de la classe ainsi que les méthodes qui lui sont spécifiques, et l'on détermine ce qui est visible, l'interface d'utilisation; et ce qui ne l'est pas, les détails d'implémentation.

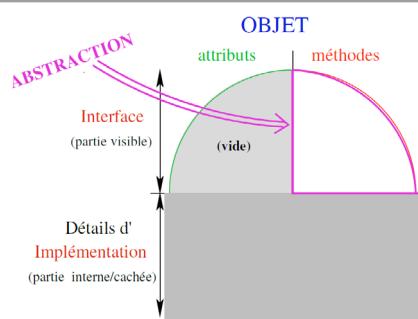


FIGURE 4

19:00

20:48

Résultat du processus d'encapsulation et d'abstraction: l'interface d'utilisation se limite à quelques méthodes bien choisies tandis que les détails d'implémentation restent cachés.



2. CLASSES, OBJETS, ATTRIBUTS ET MÉTHODES EN C++

```
class Rectangle {  
};  
  
int main()  
{  
    Rectangle rect1;  
  
    return 0;  
}
```

FIGURE 1

1:45

16:07

Exemple de déclaration d'une classe et d'un objet de ce type.

CLASSE

En C++, une classe se déclare avec le mot-clé `class`, qui définit un nouveau **type**. On utilise la syntaxe suivante :

```
class NomClasse {...};
```

Il faut être attentif à ne pas oublier le point-virgule à la fin de la déclaration.

Une fois le nouveau type défini, on peut l'utiliser pour déclarer des variables, **instances** de cette classe. La syntaxe sera celle d'une déclaration de variable quelconque, à savoir :

```
NomClasse NomVariable;
```

La figure 1 est un exemple de déclaration d'une classe `Rectangle`, et d'une variable `rect1` de ce type.

```
#include <iostream>  
using namespace std;  
  
class Rectangle {  
    double hauteur;  
    double largeur;  
};  
  
int main()  
{  
    Rectangle rect1;  
  
    rect1.hauteur = 3.0;  
    rect1.largeur = 4.0;  
  
    cout << "hauteur : " << rect1.hauteur  
        << endl;  
  
    return 0;  
}
```

FIGURE 2

3:00

16:07

Exemple de manipulation des attributs d'un objet.

ATTRIBUTS

Le processus de déclaration des attributs, au sein d'une classe, est similaire à celui des champs d'une structure : on écrit le type puis le nom de chaque attribut, le tout suivi d'un point-virgule. Pour utiliser ces attributs, on emploie également la même syntaxe que pour manipuler les champs d'une structure : le nom de l'instance, suivi d'un point et du nom de l'attribut permettent de le désigner.

Le code de la figure 2, dans lequel la classe `Rectangle` admet deux attributs, `hauteur` et `largeur`, donne un exemple d'utilisation d'attributs en C++. On déclare une instance `rect1` dans le `main`, ce qui permettra de manipuler ses attributs. Notons que le code donné ici ne compile pas en l'état, pour une raison qui sera détaillée dans la leçon 3.

MÉTHODES

Les méthodes sont des fonctions qui sont déclarées au sein de la classe. Le prototype d'une méthode indique donc le type de retour, le nom de la méthode puis entre parenthèses rondes, la liste des éventuels paramètres. L'entête de la méthode est complété par son corps, ou sa définition, entre accolades. La syntaxe sera donc la suivante :

```
type_retour NomMethode (type1 nom_parametre1,...) {  
    // Corps de la méthode ...  
}
```

Les paramètres d'une méthode sont des variables externes à la classe : chaque méthode d'une classe a accès aux attributs de celle-ci ; ils ne doivent donc pas être passés en arguments. On parle de **portée de classe** : les attributs sont des variables globales à la classe.



Dans la classe `Rectangle`, on ajoute par exemple la méthode `surface()` qui retourne un double qui ne prend pas de paramètres. En effet, elle n'exploite que les attributs de sa classe, à savoir la hauteur et la largeur, auxquels elle a directement accès. Dans d'autres situations, les méthodes peuvent cependant requérir des paramètres externes à la classe s'ils sont nécessaires pour leur fonctionnement.

Pour rendre le code plus lisible et pour favoriser la modularisation, c'est-à-dire une meilleure séparation entre l'interface et l'implémentation, on peut choisir de définir les méthodes à l'extérieur de la classe. Le prototype de la méthode devra cependant rester à l'intérieur de la déclaration de la classe. En dehors de la classe, pour indiquer au compilateur que l'on définit une méthode et non pas une fonction, il faut souligner à quelle classe elle appartient grâce à l'**opérateur de résolution de portée ::**, de la forme suivante :

```
type_retour NomClasse::NomMethode (type1 nom_parametre1,...) {  
    // Corps de la méthode ...  
}
```

Cet opérateur lie un nom local, comme le nom d'une méthode, avec un nom plus global, par exemple celui de la classe à laquelle cette méthode appartient.

ACTIONS ET PRÉDICATS

Pour une meilleure modélisation et plus de robustesse, il sera utile de distinguer les méthodes qui modifient les attributs de l'instance manipulée de celles qui ne le font pas. On désignera ces méthodes comme, respectivement, des **actions** et des **prédictats**. Un exemple de prédictat est la méthode `surface()` de la classe `Rectangle`. Il sera bon d'indiquer quand une méthode est un prédictat avec le mot `const` après la liste des paramètres pour dire qu'elle ne modifie pas l'instance à laquelle elle s'applique, comme dans l'entête de la méthode `surface()` dans la figure 3. La déclaration d'une action avec `const` dans son prototype résultera en un message d'erreur: « `assignment of data-member [...] in read-only structure` ».

Enfin, pour faire appel à ces méthodes, la syntaxe sera similaire à celle utilisée par exemple pour les `vector`:
`nom_instance.nom_methode(valeur_arg1,...)`

En effet, `vector` est une classe dont on peut utiliser la méthode `size()`, par exemple, en appelant `tableau.size()` où `tableau` est une instance de `vector`. L'appel à la méthode `surface()` dans le `main` du programme de la figure 3 est réalisé selon cette même forme: on notera qu'il ne faut pas oublier les parenthèses lors de l'appel d'une méthode qui ne prend pas d'argument, comme pour une fonction. Cette syntaxe limite le risque de confusion entre les attributs de différentes instances puisque l'appel se fait toujours sur une instance précise, gardant de ce fait la cohérence de ses attributs.

```
// ...  
class Rectangle {  
    double hauteur;  
    double largeur;  
    double surface() const  
    { return hauteur * largeur; }  
};  
  
int main()  
{  
    Rectangle rect1;  
  
    rect1.hauteur = 3.0;  
    rect1.largeur = 4.0;  
  
    cout << "surface : " << rect1.surface()  
        << endl;  
// ...
```

FIGURE 3



3. PUBLIC: ET PRIVATE:

En programmation orientée objet, il est important de séparer l'interface d'utilisation des détails d'implémentation, pour notamment limiter la partie accessible d'une classe à quelques méthodes bien choisies. En C++, les mots-clés `public` et `private` permettent d'établir cette distinction.

PUBLIC

Le mot-clé `private` est employé pour signaler quelle partie de la classe restera inaccessible à un utilisateur externe de la classe. Tout élément de la classe qui suivra le mot `private` suivi de deux points est donc un détail d'implémentation, inaccessible depuis l'extérieur de la classe. C'est dans cette catégorie que l'on déclare toujours les attributs, comme dans la classe `Rectangle` de la figure 1, mais également certaines méthodes internes qui ne peuvent être appelées qu'au sein d'une même classe, et non dans le `main` ou ailleurs à l'extérieur de la classe. Une tentative d'accéder à un élément privé d'une classe produira un message d'erreur du compilateur.

PUBLIC

Avec le mot-clé `public`, suivi de deux points, on indique quels membres d'une classe sont accessibles, visibles et utilisables. Avec la même syntaxe que précédemment, tout élément placé après cette ligne de signalement appartient à l'interface d'utilisation, qui doit se limiter à quelques méthodes. Dans l'exemple de la classe `Rectangle`, après avoir déclaré la méthode `surface()` dans la partie publique, comme cela a été fait dans la figure 1, on pourrait faire appel à celle-ci depuis le `main`. Notons que par défaut, sans spécification, tout élément d'une classe est `private`. Il est néanmoins recommandé de systématiquement expliciter les droits d'accès avec `public` et `private`. Dans la leçon 2, toute la déclaration de la classe `Rectangle` était privée puisqu'aucun droit d'accès n'avait été précisé: pour cette raison, un appel à ses éléments depuis le `main` aurait provoqué une erreur de compilation.

ACCESEURS ET MANIPULATEURS

Alors que tous les attributs sont en règle générale privés, inaccessibles, il peut être nécessaire de les utiliser depuis l'extérieur de la classe, notamment pour modifier ou connaître leur valeur. Pour cela, on peut proposer des méthodes publiques pour manipuler les attributs, en modification ou en consultation. Cette partie de la conception est extrêmement importante: il faut bien sélectionner quels attributs seront offerts au travers d'une méthode.

Des méthodes qui retournent la valeur d'un attribut sont des **accesseurs**, «méthodes get» ou «getters». Il s'agit de prédictats: le mot-clé `const` suivra leur entête. Des exemples d'accesseurs sont les méthodes `getHauteur()` et `getLargeur()` dans le code de la figure 1, qui permettent effectivement de connaître la valeur des attributs privés.



On peut aussi avoir besoin de **manipulateurs**, appelés aussi «méthodes set» ou «setters»: ceux-ci permettent de modifier la valeur d'un attribut. Ce sont donc des actions (au sens de la leçon 2). Une telle méthode prend en paramètre la valeur à affecter et ne retourne rien. Les méthodes `setHauteur()` et `setLargeur()` de la figure 1 appartiennent à cette catégorie.

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

class Rectangle {
public:
    double surface() const
    { return hauteur * largeur; }

    double getHauteur() const
    { return hauteur; }
    double getLargeur() const
    { return largeur; }

    void setHauteur(double h)
    { hauteur = h; }
    void setLargeur(double l)
    { largeur = l; }
};

private:
    double hauteur;
    double largeur;
};

int main()
{
    Rectangle rect1;

    rect1.setHauteur(3.0);
    rect1.setLargeur(4.0);

    cout << "hauteur : "
        << rect1.getHauteur()
        << endl;

    return 0;
}
```

FIGURE 1

8:10

18:59

Exemple d'une classe avec manipulateurs et accesseurs.

Le fait de garder les attributs privés et de ne les manipuler qu'à travers des méthodes permet de limiter des erreurs de l'utilisateur: le concepteur peut par exemple imposer qu'une hauteur de rectangle soit positive, garantissant ainsi l'intégrité des données. De plus, l'interface permet au concepteur de librement modifier sa représentation de la classe sans que l'utilisateur n'en soit affecté. Ces contraintes de la programmation orientée objet prennent vraiment du sens dans le contexte de gros programmes ou de partage du code.

```
class MaClasse {
private:
    int x;
    int y;
public:
    void une_methode( int x ) {
        ... y ...
        ... x ...
        ... this->x ...
    }
};
```

FIGURE 2

17:20 18:59

Exemple de masquage d'un attribut:
dans `une_methode()`, `x` désigne le paramètre
tandis que `this->x` désigne l'attribut
de l'instance courante.

MASQUAGE

On parle de masquage lorsqu'un identificateur en cache un autre, plus particulièrement quand une variable locale prend le pas sur une variable plus globale. Souvent, lorsque l'on déclare des manipulateurs, on choisit le même nom que l'attribut à modifier comme nom de paramètre: il y a donc ambiguïté derrière ce nom qui désigne deux entités, le paramètre et l'attribut. Pour ne pas recevoir de message d'erreur du compilateur, il est recommandé de choisir un nom différent pour le paramètre, mais on peut aussi désambiguïser le nom en indiquant quand il désigne l'attribut de classe. On utilise pour cela le pointeur `this`, pointeur sur l'instance courante. En tant que pointeur, il est utilisable sous la forme suivante: `this->attribut`, qui revient à écrire `(*this).attribut`. Cette syntaxe désigne donc le champ `attribut` de l'instance courante et de distinguer l'attribut d'une variable qui le masquerait. La figure 2 est un exemple d'utilisation de ce pointeur. Il reste cependant conseillé de limiter les situations de masquage.

4. ENCAPSULATION ET ABSTRACTION : RÉSUMÉ

Une classe regroupe des données, stockées dans ses attributs, et des traitements, décrits dans ses méthodes. De ce point de vue-là, on pourrait la comparer à une structure améliorée. De plus, le programmeur qui la conçoit peut déterminer lesquels de ses attributs ou méthodes seront accessibles ou pas de l'extérieur de la classe.

EXEMPLE DE CLASSE

La figure 1 est un exemple d'implémentation de la classe `Rectangle`, qui utilise la syntaxe de définition d'une classe. Le nom de la classe désignera ensuite un type dans le programme: pour cela, on choisit habituellement de le faire commencer par une majuscule. Une fois établis les attributs et les méthodes, il faut distinguer, parmi ces éléments, lesquels sont des aspects d'implémentation interne privés de ceux qui constituent l'interface d'utilisation publique. Dans la classe `Rectangle`, le concepteur offre des méthodes publiques de consultation et de manipulation des attributs ainsi que la méthode de calcul de surface, mais garde les attributs privés, comme souvent en programmation orientée objet. Cela permet notamment de protéger le code contre des erreurs, en contrôlant si les données fournies sont valables.

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

// définition de la classe
class Rectangle {
public:
    // définition des méthodes
    double surface() const
    { return hauteur * largeur; }
    double getHauteur() const { return hauteur; }
    double getLargeur() const { return largeur; }
    void setHauteur(double h) { hauteur = h; }
    void setLargeur(double l) { largeur = l; }

private:
    // déclaration des attributs
    double hauteur;
    double largeur;
};
```

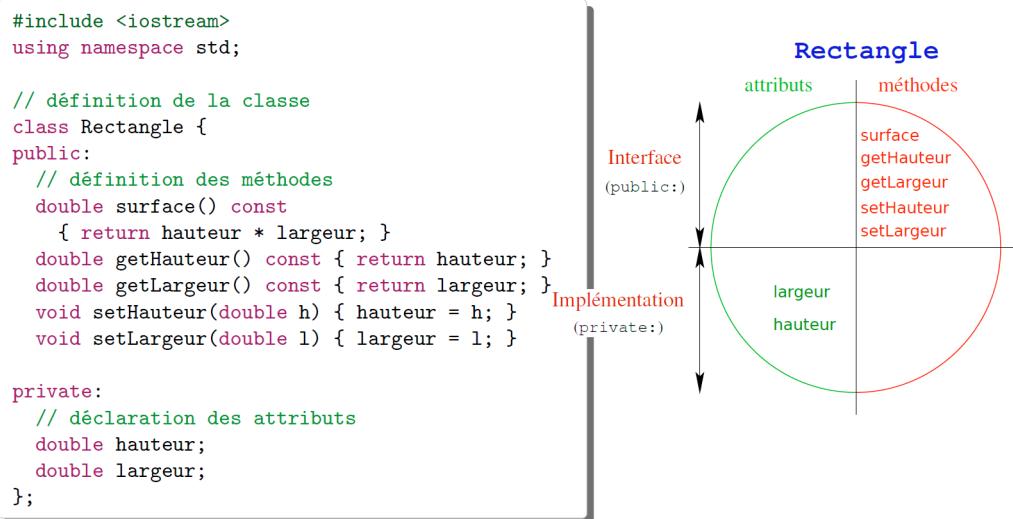


FIGURE 1

1:00

10:28

Un exemple concret de classe.

L'exemple de la classe `Rectangle` dans la figure 1 reste très basique: de manière générale, il n'est pas recommandé de systématiquement définir des accesseurs et des manipulateurs pour tous les attributs de la classe. Pour rappel, il est conseillé de marquer les prédictats, méthodes qui ne modifient pas les attributs, avec le mot `const`.

Une bonne encapsulation, une bonne séparation entre détails privés et interface publique, permet au programmeur concepteur de modifier plus librement son implémentation sans impact pour l'utilisateur. Par exemple, on pourrait choisir un tableau à deux éléments, au lieu de deux réels, pour représenter la largeur et la hauteur d'un `Rectangle`, sans aucun effet sur le programme qui utilise la classe, puisqu'il n'interagit jamais directement avec les attributs de cette classe.



Du côté du programmeur utilisateur, dont on a un aperçu avec la figure 2, `Rectangle` définit un nouveau type dont on peut déclarer une instance, un objet. Ensuite, il faut initialiser ses attributs par exemple grâce à des manipulateurs ou d'autres outils qui seront explorés plus loin. On peut alors invoquer d'autres fonctionnalités sur le rectangle comme un calcul de sa surface.

```
//utilisation de la classe

int main()
{
    Rectangle rect;
    double lu;
    cout << "Quelle hauteur ? "; cin >> lu;
    rect.setHauteur(lu);
    cout << "Quelle largeur ? "; cin >> lu;
    rect.setLargeur(lu);

    cout << "surface = " << rect.surface()
        << endl;

    return 0;
}
```

FIGURE 2

4:40

10:28

Une utilisation possible de la classe `Rectangle`.

DÉFINITION EXTERNE

L'écriture d'une classe est rendue plus lisible quand ses méthodes sont uniquement prototypées à l'intérieur de celle-ci, comme on le constate dans la figure 3. Dans ce cas, leur définition reste à l'extérieur, rattachée à la classe par l'opérateur de résolution de portée `::`. On appelle prototype de la classe les éléments qui sont écrits au sein de celle-ci, c'est-à-dire le prototype de ses méthodes, et ses attributs. Les définitions des méthodes constituent la définition de la classe.

```
class Rectangle
{
public:
    // prototypes des méthodes
    double surface() const;

    // accesseurs
    double hauteur() const;
    double largeur() const;

    // manipulateurs
    void hauteur(double);
    void largeur(double);

private:
    // déclaration des attributs
    double hauteur_;
    double largeur_;
};
```

```
double Rectangle::surface() const
{
    return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}

... // idem pour largeur
```

FIGURE 3

6:50

10:28

Externalisation des méthodes de la classe `Rectangle`, pour davantage de clarté.



Comme pour des fonctions, le prototype représente donc l'accord entre le programmeur utilisateur et le programmeur concepteur de la classe. Les définitions ne sont connues que par ce dernier, l'utilisateur n'ayant pas besoin de les connaître pour les utiliser.

Certains programmeurs adoptent une convention de nommage des attributs en les terminant par le caractère souligné, dans le but d'éviter des situations de masquage avec des noms d'arguments de méthode. Un autre choix particulier de noms est celui des accesseurs et manipulateur : grâce au mécanisme de surcharge des fonctions, on peut donner le même nom à l'accesseur et au manipulateur d'un attribut, comme il a été fait dans le programme de la figure 3. Ces méthodes se distinguent par leurs arguments et par la présence du modificateur `const`. Certains trouvent cette dénomination plus claire au niveau de l'utilisation.



5. ENCAPSULATION ET ABSTRACTION: ÉTUDE DE CAS

Pour une bonne conception, l'interface d'utilisation doit limiter l'accès de l'utilisateur aux détails d'implémentation, tout en lui permettant de travailler facilement avec la classe.

On cherche à programmer un jeu du Morpion, dans lequel deux joueurs s'affrontent sur une grille 3x3. À tour de rôle, le premier joueur pose des ronds et le deuxième des croix, par exemple, et le but de chaque joueur est d'aligner trois de ses pièces.

PREMIÈRE VERSION

Dans une première version de la classe `JeuMorpion`, présentée en figure 1, les données sont modélisées par un type `Grille`, un tableau de taille fixe pointé par l'attribut de la classe. Les fonctionnalités publiques sont une méthode `initialise()`, qui alloue un espace mémoire à la grille de jeu, et un accesseur `get_grille()` qui renvoie un pointeur vers celle-ci, permettant à un utilisateur externe d'y placer librement des jetons.

```
typedef array<int, 9> Grille;

class JeuMorpion {
public:
    void initialise() { grille = new Grille; }
    Grille* get_grille() { return grille; }

private:
    Grille* grille;
};
```

FIGURE 1

1:30

23:33

Code de la classe `JeuMorpion`.

Dès lors, l'utilisateur fait face à un certain nombre de problèmes. Tout d'abord, il doit adopter des conventions arbitraires liées aux choix de représentation de la grille, un tableau d'entiers à une dimension. Il faudra définir, par exemple, qu'un rond sera représenté par l'entier `1`, une croix par un `2` et une case vide par un `0`, représentation difficile à comprendre pour un lecteur externe. De plus, l'initialisation de la grille est incomplète puisque son contenu reste indéterminé après l'allocation mémoire : cette méthode devrait se charger du remplissage de la grille par des cases vides, des `0`. Un autre défaut de ce programme est le fait que l'utilisateur doive connaître les détails d'implémentation de la classe pour l'utiliser, ce qui rend le code cryptique pour tout observateur. On doit en effet savoir que placer un rond en haut à gauche revient à placer un `1` dans l'élément d'indice `0` de la grille de jeu, c'est-à-dire à écrire :

`(*jeu.get_grille())[0] = 1;`

La classe propose, à travers la méthode `get_grille()`, un accès libre à la grille de jeu, qui permet donc des manipulations dangereuses des données. L'utilisateur pourrait essayer d'accéder à une case d'indice invalide, ce qui pourrait provoquer des erreurs ; il pourrait également remplir sa grille par des valeurs autres que `0`, `1` ou `2` ; ou remplacer un rond déjà placé par une croix. L'accès aux données est incontrôlé et le fait d'avoir protégé l'attribut `grille` en le déclarant privé perd tout intérêt. Dans un cas général, lorsqu'un attribut est un pointeur vers un objet, offrir un accès extérieur à sa valeur nuit à l'encapsulation.

Enfin, la classe `JeuMorpion` n'offre aucune robustesse face aux changements des détails d'implémentation : si le programmeur concepteur de la classe décide de stocker la grille sous la forme d'un tableau à deux dimensions, l'utilisateur devrait complètement revoir son code afin de l'adapter à ce changement.



DEUXIÈME VERSION

La seconde version de la classe `JeuMorpion`, présentée en figures 2, 3 et 4, prend en charge la modélisation de la grille mais aussi des pions. On y introduit donc un type énuméré avec un nom parlant, et une modélisation plus naturelle de la grille par un tableau à trois lignes et trois colonnes qui ne sera plus pointé mais contenu par l'attribut `grille`. Son initialisation est complète, elle consiste à déclarer chaque case comme vide.

```
enum CouleurCase { VIDE, ROND, CROIX };
typedef array<array<CouleurCase, 3>, 3> Grille;

class JeuMorpion {
public:
    void initialise() {
        for (auto& ligne : grille) {
            for (auto& kase : ligne) {
                kase = VIDE;
            }
        }
    }
private:
    Grille grille;
    //...
}
```

FIGURE 2

14:30

23:33

Code de la nouvelle classe `JeuMorpion`.

```
public:
bool placer_rond( size_t ligne, size_t colonne) {
    return placer_coup(ligne, colonne, ROND);
}
bool placer_croix(size_t ligne, size_t colonne) {
    return placer_coup(ligne, colonne, CROIX);
}

// ici on peut rajouter une méthode get_joueur_gagnant() const
};

// fin de la classe JeuMorpion
```

FIGURE 3

16:30

23:33

Code de la nouvelle classe `JeuMorpion` (suite).

```
private:
/**
 * Place un coup sur le plateau en position (ligne, colonne).
 * Ligne et colonne : 0, 1 ou 2.
 */
bool placer_coup(size_t ligne, size_t colonne, CouleurCase coup) {
    if ( ligne < 0 or ligne >= grille.size()
        or colonne < 0 or colonne >= grille[ligne].size() ) {
        // traitement de l'erreur ici
    }
    if (grille[ligne][colonne] == VIDE) {
        // case vide, on peut placer le coup
        grille[ligne][colonne] = coup;
        return true;
    } else {
        // case déjà prise, on signale une erreur.
        // ...
        return false;
    } // suite
}
```

FIGURE 4

17:30

23:33

Code de la nouvelle classe `JeuMorpion` (suite).



L'interface d'utilisation de la classe permet de manipuler la grille par le biais des méthodes `placer_rond()` et `placer_croix()` de la figure 3 qui prennent en argument la position du jeton à placer et retournent un booléen qui indique si le placement a pu être réalisé. On peut regrouper les traitements en une méthode `placer_coup()`, vue en figure 4, que l'on appellera dans `placer_rond()` et `placer_croix()`. On la maintiendra privée puisqu'elle ne sert d'outil qu'à ces méthodes, jamais directement nécessaires à l'utilisateur externe.

Son corps contient des étapes de vérification de l'intégrité des coordonnées fournies par l'utilisateur. De plus, la méthode vérifie si la position donnée est bien celle d'une case vide. La fonction est uniquement invoquée par les méthodes de la classe, il n'est plus nécessaire de vérifier si le jeton à placer est bien un rond ou une croix. Si les données sont validées, on place le coup, et le booléen `true` est retourné; dans le cas contraire, on traite l'erreur et l'on retourne `false`.

Pour l'utilisateur, après avoir déclaré et initialisé une variable `jeu` de la classe `JeuMorpion`, il devient intuitif de placer un rond en haut à gauche, en écrivant :

```
valide = jeu.placer_rond(0,0);
```

où le booléen `valide` aurait déjà été déclaré plus haut, et permettrait de savoir si le placement a été effectué. La seule convention à connaître pour l'utilisateur est la numérotation des lignes et colonnes de `0` à `2` (ce qui doit être inclus dans la documentation de classe `JeuMorpion`). Ce code offre donc ce que l'on appelle la **séparation des soucis**, atout pour la robustesse du code: le programmeur utilisateur est responsable de la logique du jeu mais n'a pas besoin de connaître ou de comprendre les détails d'implémentations, les choix de représentation du programmeur concepteur. De plus, ce deuxième code devient facilement compréhensible grâce au nom explicite des méthodes. Enfin, la méthode privée `placer_coup()` se charge d'empêcher les erreurs de manipulation, tout en donnant un message compréhensible quand elles surviennent.

Une bonne encapsulation repose sur des détails d'implémentation invisibles à l'extérieur et sur une interface d'utilisation limitée et contrôlée.



6. CONSTRUCTEURS (INTRODUCTION)

```
class Rectangle {
public:
    void init(double h, double L)
    {
        hauteur = h;
        largeur = L;
    }
    ...
private:
    double hauteur;
    double largeur;
};
```

FIGURE 1

2:40 20:04

Figure 1 Exemple de méthode qui initialise les attributs de la classe Rectangle.

INITIALISATION DES ATTRIBUTS

Lors de la déclaration d'un nouvel objet, il existe différentes façons de donner une valeur à ses attributs. On peut opter pour des manipulateurs qui modifient individuellement chaque valeur, mais il s'agirait d'une mauvaise solution: avant tout, elle implique que tous les attributs soient assortis d'un manipulateur, ce qui ferait dépendre l'interface de l'implémentation et nuirait donc à l'encapsulation. De plus, cette solution oblige l'utilisateur à initialiser explicitement tous les attributs, au risque que certains soient oubliés.

Une meilleure solution, illustrée par la figure 1, consiste à définir une méthode dédiée à l'initialisation des attributs. Elle prend en paramètres les valeurs que prendront les attributs et permet, par exemple, de vérifier la validité des données avant l'affectation. Une telle méthode est appelée un **constructeur**.

```
// ...
class Rectangle {
public:
    Rectangle(double h, double L)
    {
        hauteur = h;
        largeur = L;
    }
    ...
private:
    double hauteur;
    double largeur;
};

int main()
{
    Rectangle rect1(3.0, 4.0);
    // ...
```

FIGURE 2

6:00 20:04

Exemple de constructeur pour la classe Rectangle , puis d'appel à celui-ci.

CONSTRUCTEURS

Un constructeur est une méthode invoquée systématiquement lors de la création d'un objet et chargée d'effectuer les opérations d'initialisation de celui-ci, notamment de ses attributs. La syntaxe de déclaration d'un constructeur est la suivante:

```
NomClasse (liste_paramètres)
{
    // Initialisation des attributs en utilisant
    liste_paramètres
}
```

Un constructeur est une méthode qui porte le même nom que la classe et n'a pas de type de retour. Un exemple en est fourni en figure 2. On peut également faire des surcharges de constructeurs et donner des valeurs par défaut à leurs paramètres.

Une fois le constructeur défini, il n'est donc pas toujours nécessaire de proposer des accesseurs pour chaque attribut: certains peuvent rester constants pour toute la durée de vie de l'objet, par exemple.

La syntaxe de **déclaration avec initialisation** d'un objet est la suivante:

```
NomClasse instance (valeur_arg1, ..., valeur_argN);
```

où `valeur_arg1, ..., valeur_argN` sont les valeurs passées aux paramètres du constructeur, comme dans la figure 2. Si la classe possède plusieurs constructeurs surchargés, on appellera celui dont les paramètres correspondent aux arguments donnés.



CONSTRUCTION DES ATTRIBUTS

Pour appeler directement le constructeur d'un attribut-objet, mais aussi pour initialiser des attributs de types de base, on utilise la **liste d'initialisation** en dehors du corps du constructeur. Son usage est fortement recommandé, afin d'éviter des copies superflues et par souci de lisibilité. On l'emploie par la syntaxe suivante, dont un exemple est donné en figure 3 :

```
NomClasse (liste_paramètres)
    : attribut1(...), //appel au constructeur de attribut1
    ...
    attributN(...)//appel au constructeur de attributN
{ // Autres opérations }
```

Pour les attributs de type de base, il suffit de spécifier la valeur entre parenthèses, après le nom de l'attribut, pour l'initialiser. De plus, les problèmes de masquage évoqués à la leçon 3 ne s'appliquent pas à la liste d'initialisation : celle-ci distingue attributs et paramètres, qui peuvent donc porter le même nom.

```
class Rectangle {
    Rectangle(double h, double L);
    // ...
};

class RectangleColore {

    RectangleColore(double h, double L, Couleur c)
        : rectangle(h, L), couleur(c)
    {}

    private:
        Rectangle rectangle;
        Couleur couleur;
};
```

FIGURE 3

12:00

20:04

Exemple de constructeur utilisant la liste d'initialisation.

`rectangle(h, L)` fait appel au constructeur de l'attribut de classe `Rectangle`.

Les initialisations par cette liste peuvent ensuite être modifiées dans le corps du constructeur. La figure 4 illustre cette situation : en entrant dans le corps de la méthode, la valeur de l'attribut `largeur` est encore indéterminée, puis elle est modifiée par une affectation. Un avantage de la liste d'initialisation est donc de donner une valeur initiale aux attributs avant même d'entrer dans le corps du constructeur, qui reste souvent vide.

```
Rectangle(double h, double L)
    : hauteur(h) //initialisation
{
    // largeur a une valeur indéfinie jusqu'ici
    largeur = 2.0 * L + h; // par exemple...
    // la valeur de largeur est définie à partir d'ici
}
```

FIGURE 4

16:30

20:04

Exemple de constructeur dont le corps modifie une valeur déterminée par la liste d'initialisation.



7. CONSTRUCTEURS PAR DÉFAUT EN C++

Le **constructeur par défaut** est appelé lors de la déclaration d'un objet sans valeurs d'initialisation. Il s'agit d'un constructeur qui ne prend pas de paramètre ou dont tous les paramètres ont des valeurs par défaut. Dans la figure 1, ce constructeur est le premier proposé, et initialise les attributs avec des valeurs par défaut.

```
// Le constructeur par defaut
Rectangle() : hauteur(1.0), largeur(2.0)
{}

// 2ème constructeur
Rectangle(double c) : hauteur(c), largeur(2.0*c)
{}

// 3ème constructeur
Rectangle(double h, double L) : hauteur(h), largeur(L)
{}
```

FIGURE 1

0:30

21:34

Exemples de constructeurs.

L'appel à un constructeur par défaut se fait en déclarant une instance par la syntaxe :

NomClasse NomInstance;

L'ajout de parenthèses vides après le nom de l'instance est une erreur à éviter, interprétée par le compilateur comme le prototype d'une fonction.

CONSTRUCTEUR PAR DÉFAUT PAR DÉFAUT

L'initialisation d'un objet étant considérée comme fondamentale, si aucun constructeur n'est explicité, le compilateur génère automatiquement une version minimale du constructeur par défaut. On parle alors du **constructeur par défaut par défaut**. Il est responsable d'appeler le constructeur par défaut des attributs lorsqu'il s'agit d'objets, et laisse les attributs de types de base (`int`, `double`, `char` ou `bool`) non initialisés. Notons que dès qu'un constructeur, par défaut ou non, est spécifié, le constructeur par défaut par défaut n'est plus fourni. Ceci est toutefois considéré comme une bonne chose, puisque si le concepteur de la classe définit un constructeur sans spécifier de constructeur par défaut, l'utilisateur se voit obligé d'initialiser explicitement les objets.

Depuis C++ 2011, il est cependant possible de réactiver le constructeur par défaut par défaut; ce choix est surtout pertinent lorsque les attributs sont des objets. On écrit alors la ligne suivante dans la classe :

NomClasse() = default;

Réactiver une variante par défaut est généralisable à d'autres méthodes. Il est aussi possible de supprimer une méthode par l'ajout de «`= delete;`» après le prototype de la méthode à supprimer. Un exemple en sera présenté à la leçon 8.

EXEMPLES DE CONSTRUCTEURS

La figure 2 propose quatre variantes de constructeurs pour une classe `Rectangle`, la A utilisant un constructeur par défaut par défaut, la B un constructeur par défaut, la C un constructeur à deux paramètres avec valeurs par défaut et la D un constructeur à deux paramètres.

A :

```
class Rectangle {  
private:  
    double h; double L;  
    // suite ...  
};
```

B :

```
class Rectangle {  
private:  
    double h; double L;  
public:  
    Rectangle()  
        : h(0.0), L(0.0)  
    {}  
    // suite ...  
};
```

C :

```
class Rectangle {  
private:  
    double h; double L;  
public:  
    Rectangle(double h=0.0,  
              double L=0.0)  
        : h(h), L(L)  
    {}  
    // suite ...  
};
```

D :

```
class Rectangle {  
private:  
    double h; double L;  
public:  
    Rectangle(double h,  
              double L)  
        : h(h), L(L)  
    {}  
    // suite ...  
};
```

FIGURE 2

11:20

21:34

Exemples de constructeurs pour la classe `Rectangle`.

Ces exemples n'admettent pas tous, ni ne traitent de la même manière, les tournures suivantes : `Rectangle r1;` et `Rectangle r2(1.0, 2.0);`.

A: Le constructeur par défaut par défaut étant un constructeur par défaut, la première variante autorise la première tournure et laisse les attributs non initialisés puisqu'ils sont de types de base. Ce constructeur ne permet pas la seconde tournure, puisqu'aucun constructeur à deux paramètres n'est spécifié.

B: La classe `Rectangle` B dispose d'un constructeur par défaut explicitement déclaré qui initialise les attributs à 0.0 avec la première tournure. La seconde est illicite pour les mêmes raisons que précédemment.

C: Le constructeur de la variante C peut être invoqué avec zéro, un ou deux paramètres : la première tournure donne aux attributs les valeurs par défaut 0.0 et la seconde, les valeurs passées en argument.

D: Enfin, la dernière variante propose un constructeur prenant deux paramètres, mais pas de constructeur par défaut puisqu'un constructeur est déjà explicitement défini. La première tournure est donc illicite, la seconde autorisée.



APPEL AUX AUTRES CONSTRUCTEURS

Depuis C++ 2011, il est autorisé qu'un constructeur en appelle un autre, en utilisant la liste d'initialisation et les arguments correspondants. La syntaxe est donnée en figure 3. Plutôt que de réactiver le constructeur par défaut par défaut qui n'initialise pas les types de base, il est conseillé d'employer cette tournure.

```
class Rectangle {
private:
    double hauteur; double largeur;
public:
    Rectangle(double h, double L) : hauteur(h), largeur(L) {}

    Rectangle() : Rectangle(0.0, 0.0) {}
    // bien mieux que le =default précédent

    // suite ...
};
```

FIGURE 3

19:00 21:34

Appel à un autre constructeur.

INITIALISATION PAR DÉFAUT DES ATTRIBUTS

Il est possible de donner directement une valeur par défaut aux attributs, comme illustré en figure 4. Lorsqu'une instance de cette classe est initialisée, si le constructeur appelé ne modifie pas la valeur de cet attribut, ce dernier prend alors la valeur par défaut. Il est toutefois conseillé d'affecter des valeurs aux attributs à l'intérieur des constructeurs plutôt que d'utiliser les valeurs par défaut, pour avoir une description regroupée et complète du travail d'initialisation d'objets.

```
class Rectangle {
    // ...
private:
    double hauteur = 0.0;
    double largeur = 0.0;
    // ...
};
```

FIGURE 4

19:45 21:34

Initialisation par défaut des attributs.



8. CONSTRUCTEUR DE COPIE

En C++, il est possible de créer une copie d'une instance au moyen de ce que l'on appelle le **constructeur de copie**. Ce constructeur s'utilise comme les autres et prend pour seul argument l'instance à copier:

```
NomClasse Instance1(liste_arguments);
NomClasse Instance2(Instance1);
```

Après l'exécution de ces instructions, `Instance1` et `Instance2` sont deux instances distinctes mais ayant les mêmes valeurs pour leurs attributs. Le constructeur de copie est notamment utilisé lorsque l'argument d'une fonction passé par valeur et ainsi copié dans une variable locale.

S'agissant d'un constructeur dont le paramètre est une instance de la même classe, le prototype du constructeur de copie est le suivant:

```
NomClasse(NomClasse const& autre) { ... }
```

L'usage de la référence (constante puisque l'instance copiée n'est pas modifiée) est important pour ne pas avoir de copie qui induirait un appel récursif (fonction s'appelant elle-même) de ce constructeur. Sa définition ne peut suivre que le même schéma que dans la figure 1.

```
Rectangle(Rectangle const& autre)
: hauteur(autre.hauteur), largeur(autre.largeur)
{}
```

FIGURE 1

2:20

7:54

Exemple de constructeur de copie.

Il n'est souvent pas nécessaire de définir explicitement le constructeur de copie : le compilateur C++ en fournit une version par défaut souvent suffisante. Cet outil réalise alors une **copie de surface**, qui recopie la valeur des attributs, comme dans la figure 1. La redéfinition de ce constructeur n'est nécessaire que dans des situations particulières abordées en leçon 20. Une règle finale à mémoriser est que si l'on redéfinit une des méthodes que sont le constructeur de copie, le destructeur et l'opérateur d'affectation (présentés respectivement en leçons 9 et 14), alors on doit s'interroger sur le statut des deux autres : doivent-ils aussi être redéfinis ?

SUPPRESSION DE LA COPIE

Lorsqu'une instance de classe occupe trop de mémoire pour être dupliquée, par exemple, C++ 2011 permet d'en empêcher la copie. Pour cela, on place «= `delete`» à la fin du prototype du constructeur de copie, syntaxe utilisée en figure 2.

```
class PasCopiable {
/* ... */
PasCopiable(PasCopiable const&) = delete;
};
```

FIGURE 2

6:30

7:54

Exemple de suppression du constructeur de copie.



9. DESTRUCTEURS

GESTION DE LA FIN DE VIE

La gestion de la fin de vie d'un objet devient nécessaire lorsque l'initialisation de ce dernier mobilise des ressources comme des fichiers, des périphériques, des portions de mémoire... Il est alors important de libérer ces ressources après usage.

Plaçons-nous dans le cas d'une classe où les attributs sont des pointeurs et où le constructeur alloue dynamiquement de la mémoire pour ces attributs lors de la création d'instances. Au terme du bloc dans lequel on déclare un objet de cette nature, celui-ci est en fin de vie et ne devient plus utilisable; pourtant, les zones mémoire mobilisées ne sont pas pour autant libérées. Une solution possible consisterait à permettre à l'utilisateur d'invoquer `delete` sur chacun des attributs avant la fin de vie de l'instance: cette idée brise l'encapsulation puisqu'elle implique de rendre les attributs publics ou de livrer un pointeur par le biais d'un accesseur, pratique peu recommandée (leçon 5). De plus, cette libération n'est pas systématisée et pourrait donc être source d'erreurs ou d'oublis.

```
int main()
{
    // compteur = 0
    Rectangle r1;
    // compteur = 1
    {
        Rectangle r2;
        // compteur = 2
        // ...
    }
    // compteur = 1
    return 0;
} // compteur = 0
```

FIGURE 1

8:10

DESTRUCTEUR

C++ propose, dans cette optique, des méthodes appelées **destructeurs**, dont la particularité est d'être invoquées automatiquement en fin de vie de l'instance. Dans l'exemple précédent, le destructeur serait chargé de libérer automatiquement les zones mémoire sans les inconvénients d'une libération explicite par l'utilisateur.

Le nom du destructeur est celui de la classe, précédé du symbole `~`. Il s'agit d'une méthode sans paramètres, qui ne peut donc pas être surchargée. La syntaxe de déclaration du destructeur est la suivante:

```
~NomClasse() { // opérations (de libération) }
```

Si le destructeur n'est pas défini explicitement, le compilateur en génère automatiquement une version minimale par défaut, dont le corps est vide.

Cette version minimale du destructeur n'est pas toujours suffisante, même si sa classe n'effectue pas d'allocation de ressources: l'exemple des figures 1 et 2 le montre. Dans ce programme, on souhaite compter le nombre d'instances actives de `Rectangle` à chaque instant: on déclare pour cela une variable globale `compteur`, initialisée à 0, incrémentée à la construction de chaque nouvelle instance. Pour que le comptage se mette à jour à la fin de vie des objets, il faut aussi définir explicitement le destructeur pour décrémenter le compteur lorsqu'une instance disparaît.

```
long compteur(0); /* Hmm.... On y reviendra
                     dans une autre séquence vidéo ! */

class Rectangle {
//...
    Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur
        ++compteur;
    }
    ~Rectangle() { --compteur; } // destructeur
// ...
```

FIGURE 2

10:35

14:56



Dans cet exemple, il se révèle aussi nécessaire de définir le constructeur de copie: sans cela, les nouvelles instances copiées d'une autre ne sont pas comptabilisées par le compteur, alors que leur destruction affectera ce dernier qui atteindra une valeur négative inattendue. On ajoute donc au code précédent la redéfinition du constructeur de copie défini de sorte à ce qu'il incrémente aussi le compteur. Une règle à mémoriser est que si l'on redéfinit une des méthodes que sont le destructeur, le constructeur de copie et l'opérateur d'affectation, alors on doit s'interroger sur le statut et la possible redéfinition des deux autres. On pourrait également ajouter le constructeur de déplacement et l'opérateur de déplacement à ce groupe de méthodes liées, outils trop avancés pour être décrits dans ce cours.



10. VARIABLES ET MÉTHODES DE CLASSE

```
class Rectangle {
private:
    double hauteur, largeur;
    static int compteur;
//...
};
```

FIGURE 1

4:20

13:10

Déclaration d'un attribut de classe.

ATTRIBUTS DE CLASSE

Dans les leçons précédentes, les attributs représentaient des informations spécifiques à une instance de la classe. Il existe cependant des situations dans lesquelles plusieurs instances d'une même classe doivent accéder à une information commune.

Dans la leçon 9, par exemple, le programme de la figure 2 comptabilise l'ensemble des instances existant à un moment donné. Pour cela, le programmeur a défini son compteur comme une variable globale, accessible par toutes les instances, incrémentée à chaque création d'objet et décrémentée à chaque destruction. Il s'agit d'une mauvaise solution: une variable globale brise le principe d'encapsulation et de modularisation, et peut induire des effets de bord néfastes, étant accessible de façon incontrôlée à tout endroit du programme.

La solution consiste à définir comme **attribut de classe** l'information commune à toutes les instances, et non plus relative à celles-ci. Sa déclaration suit le même modèle qu'une déclaration d'un attribut d'instance, à l'intérieur de la classe, mais est précédée du mot réservé `static`, comme on le voit dans la figure 1. Un attribut de classe peut également être privé ou public.

Dans un programme où coexistent plusieurs instances d'une même classe, chaque objet a des valeurs spécifiques pour ses **attributs d'instance**, comme la hauteur et la largeur dans la classe `Rectangle`. En revanche, l'attribut de classe, comme le compteur dans la figure 1, est une zone mémoire unique qui est accessible par toutes les instances. Notons qu'elle est accessible sans qu'aucune instance ne soit encore créée, au travers du nom de la classe et de l'opérateur de résolution de portée. La syntaxe suivante permet donc d'accéder à l'attribut de classe en dehors de celle-ci:

`NomClasse::AttributClasse`

Tandis qu'un attribut d'instance est généralement initialisé lors de la construction, un attribut de classe existe indépendamment de toute instance et ne peut donc pas être initialisé par un constructeur. Pour l'initialiser, il faut utiliser la syntaxe illustrée en figure 2, en dehors de la classe:

`Type NomClasse::AttributClasse(Valeur);`

```
/* Initialisation de l'attribut de classe dans le fichier de
définition de la classe, mais HORS de la classe.
*/
int Rectangle::compteur(0);

/* Rectangle::compteur existe même si l'on n'a déclaré
aucune instance de la classe Rectangle */
```

FIGURE 2

8:20

13:10

Initialisation d'un attribut de classe.



Les attributs de classe permettent d'éviter des duplications inutiles et des problèmes de maintenance. Par exemple, dans une classe représentant des employés, si la donnée de l'âge officiel de départ à la retraite est la même pour tous, on déclarerait cette dernière information, commune à tout employé, comme un attribut de classe. Dans le cas contraire, il faudrait dupliquer inutilement sa valeur pour chacun des employés considérés. De plus, des problèmes de maintenance se poseraient dans le cas où cette valeur serait amenée à changer, puisqu'il faudrait alors modifier sa valeur dans chaque instance de la classe.

Enfin, il faut souligner qu'un attribut de classe stocke souvent une constante. L'utilisation d'une variable de classe non constante, comme dans l'exemple du compteur, dépendant du nombre d'instances existantes, est plus rare.

MÉTHODES DE CLASSES

Une **méthode de classe** est une méthode définie à l'intérieur d'une classe, dont la déclaration est précédée du mot `static`. Comme précédemment, une méthode de classe peut être invoquée, grâce à l'opérateur de résolution de portée, sans qu'aucun objet de la classe n'existe, comme on le constate sur la figure 3. Ce n'est pas le cas pour les **méthodes d'instances** étudiées jusqu'ici, directement liées à une instance précise. On note dans la figure 3 qu'une méthode statique d'une classe peut également être appelée au travers d'une instance, tout comme un attribut statique, s'il est accessible.

```
class A {  
public:  
    static void methode1() { cout << "Méthode 1" << endl; }  
    void methode2() { cout << "Méthode 2" << endl; }  
};  
  
int main () {  
    A::methode1(); // OK  
    A::methode2(); // ERREUR  
    A x;  
    x.methode1(); // OK  
    x.methode2(); // OK  
}
```

FIGURE 3

9:10

13:10

Exemple des appels possibles à une méthode d'instance et à une méthode de classe.

Une méthode de classe est indépendante de l'existence d'un objet courant sur lequel elle s'appliquerait, et ne peut donc pas exploiter de méthode ou d'attribut d'instance dans sa définition, mais uniquement d'autres membres de classe (« statiques »). Ainsi, une méthode de classe peut être vue comme une fonction usuelle placée dans une classe. On pourrait justifier leur usage pour afficher ou manipuler des valeurs privées d'attributs de classe par exemple ; mais on préférera les fonctions usuelles dès que possible.

Un membre statique brise l'approche orientée objet, puisqu'il est indépendant de tout objet : on limitera donc l'utilisation du `static` à des attributs de classe constants, si nécessaires.



11. SURCHARGE D'OPÉRATEURS: INTRODUCTION

Comme souligné par la figure 1, il est plus intuitif et moins fastidieux d'utiliser le symbole `+` pour additionner des complexes préalablement définis dans une classe, qu'une fonction `addition()`. De même, on pourrait vouloir les afficher de manière homogène plutôt que de devoir faire appel à une fonction `affiche()`. La surcharge d'opérateurs permet d'utiliser ces écritures plus naturelles, d'étendre l'utilisation des opérateurs usuels à des nouvelles classes.

Exemple avec les nombres complexes :

```
class Complexe { ... };
Complexe z1, z2, z3, z4;
```

Il est quand même plus naturel d'écrire :

```
z4 = z1 + z2 + z3;
que z3 = addition(addition(z1, z2), z3);
```

De même, on préfèrera unifier l'affichage :

```
cout << "z3 = " << z3 << endl;
plutôt que d'écrire :
cout << "z3 = ";
affiche(z3);
cout << endl;
```

FIGURE 1

2:30

11:09

Illustration de l'intérêt des opérateurs.

Il s'agit d'un sujet technique qui plonge le programmeur au cœur du langage puisqu'il redéfinit des opérations élémentaires. On peut aborder la surcharge d'opérateurs de différentes façons, selon le niveau de chacun ; ce cours vise un niveau qui permet la surcharge des opérateurs arithmétiques simples et de l'opérateur d'affichage. Les niveaux de la surcharge des opérateurs d'auto-affectation sans retour et enfin avec valeur de retour seront également explorés, bien que d'un niveau plus avancé.



OPÉRATEURS

Un **opérateur** est un signe qui représente une opération :

- arithmétique : `+`, `-`, ...
- logique : `and`, `or`, ...
- de comparaison : `==`, `<`, `>`, `!=`, ...
- d'auto-incrémentation : `++`
- d'affectation : `=`

Une expression qui contient un opérateur fait appel à une fonction ou à une méthode. Ecrire `a Op b` revient à appeler soit la fonction `operatorOp(a, b)`, soit la méthode `a.operatorOp(b)` qui appartient à la classe dont `a` est une instance. C'est le cas, par exemple, pour `cout << a`, qui appelle `operator<<(cout, a)` ou la méthode `cout.operator<<(a)` de la classe `ostream` à laquelle appartient `cout`. La figure 2 liste les appels possibles de quelques opérateurs.

<code>a + b</code>	correspond à	<code>operator+(a, b)</code>	ou	<code>a.operator+(b)</code>
<code>b + a</code>		<code>operator+(b, a)</code>		<code>b.operator+(a)</code>
<code>-a</code>		<code>operator-(a)</code>		<code>a.operator-()</code>
<code>cout << a</code>		<code>operator<<(cout, a)</code>		<code>cout.operator<<(a)</code>
<code>a = b</code>		—		<code>a.operator=(b)</code>
<code>a += b</code>		<code>operator+=(a, b)</code>		<code>a.operator+=(b)</code>
<code>++a</code>		<code>operator++(a)</code>		<code>a.operator++()</code>
<code>not a</code>	ou	<code>operator not(a)</code>		<code>a.operator not()</code>
		<code>operator! (a)</code>		<code>a.operator! ()</code>

FIGURE 2

7:00

11:09

Exemples d'appels d'opérateurs.

Notons que l'opérateur `operator=` correspond systématiquement à une méthode de classe et jamais à une fonction.

Pour des opérateurs unaires, c'est-à-dire qui ne sont liés qu'à un seul opérande, comme `-a` pour l'inverse par exemple, on appellera la fonction `operator-` avec `a` comme argument, ou la méthode `a.operator-` sans argument.

SURCHARGE D'OPÉRATEURS

On dit qu'il y a **surcharge** quand une fonction a le même nom qu'une autre dont elle se différencie par ses arguments. En créant une fonction `max` qui prend deux entiers comme paramètres et une autre fonction `max` qui prend deux `doubles` comme paramètres, on surcharge les fonctions `max`.

La surcharge d'opérateur consiste donc à étendre l'utilisation des opérateurs couramment employés à de nouvelles classes. Leur nom est conservé, mais pas leurs arguments. Par exemple, on peut surcharger l'opérateur d'addition, qui jusque-là n'était défini que pour des types de base, pour la classe `Complexe`:

```
Complexe operator+(Complexe, Complexe);
```

Comme vu dans le paragraphe précédent, un opérateur peut être surchargé de deux manières. La **surcharge externe** utilise une fonction qui prend les opérandes comme argument, prototypée et définie à l'extérieur de la classe. Une **surcharge interne** est une méthode de la classe, dont l'instance courante est un opérande.



12. SURCHARGE D'OPÉRATEURS : SURCHARGE EXTERNE

La surcharge externe consiste à définir les opérateurs comme des fonctions externes à la classe sur laquelle on travaille.

PROTOTYPES

Comme vu dans la leçon précédente, écrire `z3=z1+z2`, où les variables sont des instances d'une classe `Complexe`, par exemple, correspond à l'appel de la fonction `z3=operator+(z1,z2)`. Son prototype prendra donc deux arguments de type `Complexe` et retournera un `Complexe`. Pour optimiser cet appel, il est possible de faire passer les arguments par **référence constante** par le biais du `const&`, et également de qualifier de type de retour par `const` (voir leçon 14). Enfin, en C++ 2011, pour une raison qui sera aussi explicitée leçon 14, le prototype suivant serait encore une meilleure option :

```
const Complexe operator+(Complexe z1, Complexe const& z2);
```

Les différents prototypes possibles de cette fonction sont listés dans la figure 1.

Dans la définition de cette fonction, on pourra ensuite construire la nouvelle valeur qui correspond à l'addition de deux complexes, renvoyée en utilisant par exemple le constructeur de la classe `Complexe`.

- ▶ base :

```
Complexe operator+(Complexe z1, Complexe z2);
```

- ▶ optimisation :

```
const Complexe operator+(Complexe const& z1, Complexe const& z2);
```

- ▶ avancé et C++11 :

```
const Complexe operator+(Complexe z1, Complexe const& z2);
// const Complexe operator+(Complexe const& z1, Complexe&& z2);
```

FIGURE 1

2:30

17:28

Différents prototypes possibles pour la surcharge externe de l'opérateur + entre deux instances de la classe `Complexe`.



NÉCESSITÉ DE LA SURCHARGE EXTERNE

Il existe des situations, bien que peu courantes, dans lesquels le programmeur n'a pas le choix entre surcharge externe ou interne et doit employer la première: c'est le cas quand les opérandes sont de type différent ou quand on veut surcharger l'opérateur d'affichage.

Par exemple, écrire `a * z`, où `a` est un `double` et `z` un `Complexe`, appellera une méthode `a.operator*()` dans le cas d'une surcharge interne ou la fonction externe à deux arguments `operator*()`. La première option est dénuée de sens puisque `a` appartient à un type de base (`double`) et non à une classe que l'on pourrait modifier. Le programmeur doit donc utiliser la surcharge externe. En revanche, dans la définition de cette fonction, il peut faire appel, si elle existe, à une méthode qui effectue le même calcul mais dont les opérandes sont inversés: `z*a` (fig. 2). Ce calcul peut alors correspondre à une méthode `z.operator*()` au sein de la classe `Complexe`.

```
double x;
Complexe z1, z2;
// ...
z2 = x * z1;

const Complexe operator*(double a, Complexe const& z)
{
    /* Soit l'écrire explicitement,
       soit, quand c'est possible, utiliser l'opérateur interne :
    */
    return z * a;
}
```

FIGURE 2

8:00

17:28

Définition d'une surcharge externe utilisant une surcharge interne.

D'autre part, en écrivant `cout << z1`, où `z1` est un `Complexe`, les deux appels suivants sont imaginables : l'appel à la méthode de la classe `ostream` dont `cout` est une instance, ou la fonction externe qui prend les deux paramètres `cout` et `z1`. Or la classe étudiée ici est la classe `Complexe` et l'on ne cherche pas à modifier la classe `ostream` (à laquelle on n'a d'ailleurs pas accès): dans cette situation, à nouveau, on utilisera donc une surcharge externe. Le prototype de cette fonction est le suivant :

```
ostream& operator<<(ostream&, Complexe const&);
```

Il est important de passer le premier argument par référence puisque l'affichage va modifier le flot en question. Les flots ne peuvent d'ailleurs être passés que par référence. La leçon 14 justifiera le choix du type de retour de cette fonction.



Pour définir le corps de la fonction, on peut imaginer plusieurs alternatives, décrites en figure 3. La première consisterait à utiliser des accesseurs qui retournent les attributs que l'on souhaite afficher. Une deuxième définition possible, et préférable, peut utiliser une méthode préalablement définie, qui, par exemple, convertit un `Complexe` en une chaîne de caractères, comme la méthode `to_string()`, ou affiche un complexe, comme `affiche()` qui prend en paramètre le flot à modifier et réalise l'affichage demandé.

Via des accesseurs :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    sortie << '(' << z.get_x() << ", " << z.get_y() << ')';
    return sortie;
}
```

Via une autre méthode :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    sortie << z.to_string();
    return sortie;
}
```

Ou :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    return z.affiche(sortie);
}
```

FIGURE 3

13:00

17:28

Différentes définitions possibles de l'opérateur d'affichage.

DROITS D'ACCÈS AUX ÉLÉMENTS PRIVÉS

Lors de surcharges externes, il est parfois nécessaire d'accéder aux attributs privés de la classe sur laquelle on veut faire porter l'opérateur. Il est alors fortement recommandé de toujours utiliser l'interface publique de la classe, les accesseurs, mais certains préfèrent briser le principe d'encapsulation : on donne alors un accès privilégié à la fonction en copiant son prototype, précédé du mot-clé `friend`, dans la définition de la classe. Cette « friendship » entre la fonction externe et la classe lui confère le droit d'accéder à ses éléments privés. Si l'on déclare un tel lien entre la fonction de surcharge de l'opérateur d'affichage et la classe `Complexe` comme en figure 4, on peut ainsi utiliser directement les attributs privés des instances, mais, à nouveau, il est peu conseillé d'employer cette technique.

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    sortie << '(' << z.x << ", " << z.y << ')';
    return sortie;
}

// ...

class Complexe {
    friend ostream& operator<<(ostream& sortie, Complexe const& z);
    // ...
private:
    double x;
    double y;
};
```

FIGURE 4

16:20

17:28

Exemple de fonction `friend` de la classe `Complexe`.



13. SURCHARGE D'OPÉRATEURS: SURCHARGE INTERNE

Un opérateur peut également être surchargé à l'intérieur d'une classe: dans ce cas-là, il s'agira d'une méthode.

On peut donc choisir de définir un opérateur *Op*, comme une méthode d'une classe. Il s'appliquera donc à ses instances. Le premier opérande est alors l'instance courante sur laquelle on appelle la méthode, et le deuxième éventuel sera son unique argument, puisqu'elle a déjà accès aux attributs de l'instance courante. On placera le prototype de l'opérateur au sein de la définition de la classe, et on peut, comme avec une méthode quelconque, donner sa définition à l'extérieur au moyen de l'opérateur de résolution de portée.

L'opérateur `+=` d'une classe *Complexe*, par exemple, ne crée pas de nouvelle instance, comme l'opérateur `+` le fait, mais modifie son premier opérande: on dit qu'il est proche de la classe puisqu'il en modifie les objets et ceci justifie le choix d'une surcharge interne. On explicite également sa définition, à l'extérieur de la classe, dans la figure 1.

```
Complexe z1, z2;
// ...
z1 += z2;
```

```
class Complexe {
public:
    // ...
    void operator+=(Complexe const& z2); // z1 += z2;
    // ...
};

void Complexe::operator+=(Complexe const& z2) {
    x += z2.x;
    y += z2.y;
}
```

FIGURE 1

3:50

12:08

Exemple de surcharge interne d'un opérateur.



LIEN ENTRE OPÉRATEURS

Une fois l'opérateur `+=` défini, il est possible d'établir le lien sémantique entre ce dernier et l'opérateur `+`. Puisqu'il crée une nouvelle instance de la classe, ce dernier demande plus de traitements que `+=`, qu'il exploitera donc dans sa définition, comme dans la figure 2. On préfère définir l'opérateur le plus «lourd» en fonction du plus «léger».

```
class Complexe {
public:
    // ...
    void operator+=(Complexe const& z2); // z1 += z2;
    // ...
};

const Complexe operator+(Complexe z1, Complexe const& z2) {
    z1 += z2; // utilise l'opérateur += redéfini précédemment
    return z1;
}
```

FIGURE 2

4:30

12:08

Redéfinition de l'opérateur `+` en fonction de l'opérateur `+=`.

Dans un algorithme qui calculerait la somme de deux instances `c1` et `c2` de la classe `Complexe` grâce à l'opérateur `+=`, on créerait un nouveau `Complexe tmp` initialisé à la valeur de `c1`, auquel on appliquerait la méthode `+= c2`. La variable `tmp` correspondrait bien à la somme de `c1` et `c2`. Il s'agit de l'algorithme utilisé dans la définition de la fonction `operator+` de la figure 2: par un passage par valeur du premier opérande, la valeur de celui-ci est copiée dans une zone mémoire locale à la fonction. Cette variable temporaire sera la valeur de retour et a donc le même rôle que `tmp` dans l'algorithme précédent. Une telle définition est concise, optimisée et établit efficacement le lien entre des opérateurs proches; de plus, elle permet d'éviter l'usage de getters auparavant nécessaire pour accéder aux attributs, qui peuvent nuire à l'encapsulation.

SURCHARGE INTERNE OU EXTERNE ?

Dans certaines situations, le choix entre surcharge interne et externe ne se pose pas: c'est le cas des exemples cités dans la leçon 12. Mais de manière générale, on peut souvent choisir entre les deux surcharges: c'est le cas de l'opérateur `+` entre deux instances, par exemple. Il faut alors tenir compte des recommandations suivantes pour décider laquelle utiliser.

Un premier conseil est de préférer la surcharge externe lorsque le corps de la fonction peut être écrit sans avoir recours au mot-clé `friend` qui brise l'encapsulation et sans devoir coder des accesseurs inutiles autrement. Dans ce cas, l'opérateur doit pouvoir être défini par le biais de méthodes publiques de la classe et sans copies inutiles: un exemple est l'opérateur `+` vu précédemment, qui exploite le `+=`. Au contraire, ce dernier doit modifier une instance: pour lui donner un accès interne, on préférera alors une surcharge interne.



14. SURCHARGE D'OPÉRATEURS : COMPLÉMENTS

S'inscrivant au cœur du langage, la surcharge des opérateurs est un sujet assez difficile qui peut s'aborder à différents niveaux, listés en figure 1. On peut d'abord choisir de ne pas surcharger les opérateurs ou de se contenter des opérations simples ou de l'affichage. Ensuite, on pourra définir les opérateurs d'auto-affectation. Enfin on pourra choisir de leur donner une valeur de retour strictement conforme à la norme.

Dans votre pratique du C++, vous pouvez, en fonction de votre niveau :

- ① ne pas faire de surcharge des opérateurs ;
- ② surcharger simplement les opérateurs arithmétiques de base (+, -, ...) sans leur version « auto-affectation » (+=, -=, ...);
surcharger l'opérateur d'affichage (<<);
- ③ surcharger les opérateurs en utilisant leur version « auto-affectation », mais sans valeur de retour pour celles-ci :
`void operator+=(Classe const&);`
- ④ faire la surcharge avec valeur de retour des opérateurs d'« auto-affectation » :
`Classe& operator+=(Classe const&);`

FIGURE 1

1:20

24:04

Différents niveaux de surcharge d'opérateurs.

La figure 2 donne des exemples de surcharges d'une classe `Classe`, comme l'affichage, les opérations arithmétiques simples, d'auto-affectation, ou entre types différents, tous évoqués en leçons 12 ou 13. Il est également possible de surcharger des opérateurs unaires comme le signe - ou l'auto-incrément ++, méthodes internes sans argument, ou de comparaison comme ==, != ou <, qui devront être définis les uns en fonction des autres pour maintenir le lien sémantique.

```

bool operator==(Classe const&) const; // ex: p == q
bool operator<(Classe const&) const; // ex: p < q

Classe& operator+=(Classe const&); // ex: p += q
Classe& operator-=(Classe const&); // ex: p -= q

Classe& operator*=(autre_type const); // ex: p *= x;

Classe& operator++(); // ex: ++p
Classe& operator++(int inutile); // ex: p++

const Classe operator-() const; // ex: r = -p;

// ===== surcharges externes -----
const Classe operator+(Classe, Classe const&); // r = p + q
const Classe operator-(Classe, Classe const&); // r = p - q

ostream& operator<<(ostream&, Classe const&); // ex: cout << p;
const Classe operator*(autre_type, Classe const&); // ex: q = x * p;

```

FIGURE 2

2:00

24:04

Exemples de surcharges, au niveau 4 de la figure 1.



RETOUR CONSTANT

La leçon 12 annonçait que le type de retour des opérateurs arithmétiques simples serait précédé du mot-clé `const`. Ceci déclare que `z1+z2`, le résultat de l'appel à `operator+(z1, z2)` appartenant à la même classe que les opérandes, n'est pas une variable, c'est-à-dire que l'on ne peut pas modifier sa valeur. Sans déclarer que la valeur de retour est constante, celle-ci pourrait être traitée comme une instance quelconque : il serait alors possible d'écrire des formules absurdes comme `++(z1+z2)` ou `(z1+z2)=f(x)`. Le type de retour de tels opérateurs arithmétiques doit donc être un `const Classe`, afin d'empêcher des modifications hasardeuses du résultat.

RETOUR DE L'OPÉRATEUR D'AFFICHAGE

Pour afficher une instance de la classe `Complexe` suivie d'un saut de ligne, il est commode d'écrire :

```
cout << z1 << endl;
```

Cette expression est donc un appel à la fonction `operator<<(cout<<z1, endl);` qui est elle-même équivalente à `operator<<(operator<<(cout, z1), endl);`. Donc la valeur de retour de `operator<<(cout, z1)` est le premier argument d'un autre appel à `operator<<`, il s'agit donc également d'un `ostream&`, passé par référence pour pouvoir être modifié et identique à celui passé en argument dans le premier appel. Avec un type de retour `void`, enchaîner des opérateurs `<<` n'aurait plus de sens.

RETOUR DES OPÉRATEURS D'AUTO-AFFECTATION

À un premier niveau de surcharge, on peut donner un type de retour `void` aux opérateurs comme `+=`. On peut effectivement faire un appel autonome à ces méthodes en écrivant uniquement l'expression `z1+=z2;`. En C++, toute expression a pourtant une valeur et il serait donc concevable d'écrire `z3=(z1+=z2);`. Le résultat de `z1+=z2`, qui sera la valeur de `z1` après l'opération d'auto-affectation, pourra être stocké dans `z3` : il s'agit donc d'une instance de même classe que l'opérande `z1`, avec la même valeur. Pour éviter une copie superflue, on peut donc retourner une référence sur `z1` : au niveau avancé, le type de retour des opérateurs d'auto-affectation est une référence sur l'instance courante, de la forme `Classe&`.

Au sein de la définition de la méthode, il faudra bien retourner, après les diverses opérations d'affectation, la valeur de l'instance courante. Puisque `this` est un pointeur sur celle-ci, la valeur à retourner est donc `*this`.



ÉVITER LES COPIES

Dans l'exemple précédent, le type de retour est une référence puisque l'instance résultante existe déjà : ceci permet d'éviter des copies inutiles. De manière générale, il est important d'optimiser la surcharge d'opérateurs à l'aide de références afin de limiter toute copie superflue : les opérateurs étant employés fréquemment dans un programme, une telle erreur serait souvent répétée et pourrait avoir des répercussions néfastes.

La figure 3 donne deux versions de surcharge de l'opérateur `+=`. La première fait jusqu'à trois copies (selon le compilateur) contre aucune pour la deuxième. Tout d'abord, la première version effectue un passage par valeur, une copie potentielle, de l'argument qui est passé par référence constante dans la seconde. De plus, la valeur de retour est une nouvelle instance de la classe, copiée puisqu'elle n'est pas passée par référence contrairement à la deuxième version. Enfin, dans le corps de la méthode, la première version déclare la nouvelle instance `Complexe` qui sera retournée, dans laquelle on copie l'instance courante modifiée. La deuxième version est bien plus optimisée et efficace : il est conseillé de toujours employer une référence quand on en a la possibilité.

Exemple : comparez le code suivant qui fait de 1 à 3 copies inutiles :

```
Complexe Complexe::operator+=(Complexe z2)
{
    Complexe z3;
    x += z2.x;
    y += z2.y;
    z3 = *this;
    return z3;
}
```

avec le code suivant qui n'en fait pas :

```
Complexe& Complexe::operator+=(Complexe const& z2)
{
    x += z2.x;
    y += z2.y;
    return *this;
}
```

FIGURE 3

14:30

24:04

Comparaison entre deux surcharges d'un même opérateur, dont l'une n'est pas optimisée.

OPÉRATEUR D'AFFECTATION

L'opérateur `=` est particulier, étant l'unique opérateur dit **universel**, c'est-à-dire fourni d'office pour toutes les classes. On note que la différence entre construction de copie et affectation repose sur le fait qu'une construction de copie est appelée lors d'une **initialisation** d'instance. Comme pour le constructeur de copie et le destructeur, la version par défaut de l'opérateur d'affectation fait une copie de surface (notion qui sera présentée en leçon 20) souvent suffisante. Sa redéfinition, lorsqu'elle s'avère nécessaire, est souvent liée à la redéfinition du constructeur de copie et du destructeur, comme vu en leçon 9.

Dans certaines situations, on peut aussi supprimer l'usage d'un tel opérateur. Si une classe utilise beaucoup de mémoire, par exemple, on peut empêcher des copies d'instances avec une syntaxe similaire à la suppression du constructeur de copie, vue en leçon 8. Il faudrait donc placer le prototype suivant au sein de la classe :

```
EnormeClasse& operator=(EnormeClasse const&) = delete;
```

L'opérateur d'affectation ne peut être surchargé qu'en interne. Son argument doit être de la même classe que l'instance courante et sera passé par référence constante par souci d'optimisation. Enfin, pour pouvoir écrire `a = (b = c)`, la valeur de retour d'une affectation sera une référence sur la valeur finale de l'instance courante.



FONCTION SWAP

L'opérateur d'affectation fourni par défaut est souvent suffisant ; cependant, s'il est nécessaire de le redéfinir, en C++ 2011, il est conseillé d'utiliser le « **swap** », comme dans la figure 4.

La fonction **swap** échange deux arguments : **swap(a, b)** va échanger les valeurs entre les variables **a** et **b**. Elle est définie dans la bibliothèque **utility** pour les types de base, et doit donc être étendue à la classe concernée par une surcharge de la fonction **swap**, qui échangerait les valeurs des attributs entre deux instances. Elle prendrait en argument ces deux instances, par référence afin de pouvoir les modifier.

Pour redéfinir l'opérateur d'affectation de façon optimale, il faudrait utiliser la fonction **swap** entre l'instance courante et l'argument passé par valeur, qui contient la valeur du deuxième opérande. Puisque l'échange se fait avec une copie locale de celui-ci, seule l'instance courante, le premier opérande, sera modifiée pour prendre sa valeur. L'affectation est ainsi optimisée.

```
Classe& Classe::operator=(Classe source)
// Notez le passage par VALEUR
{
    swap(*this, source);
    return *this;
}
```

FIGURE 4

21:20

24:04

Méthode à suivre pour redéfinir l'opérateur d'affectation, si nécessaire.

15. HÉRITAGE: CONCEPTS

Après les notions d'encapsulation et d'abstraction, la troisième notion fondamentale de la programmation orientée objet est l'héritage.

Pour illustrer cette notion, on peut imaginer vouloir représenter les personnages d'un jeu. On pourrait créer une classe spécifique à chaque type de personnage, qui aurait comme attributs un nom, une durée de vie mais aussi des éléments propres à son identité comme une arme pour un guerrier. Ces classes partageraient également une méthode pour rencontrer d'autres personnages qui, dans le cas d'un voleur, lui permettrait de les voler. Une telle solution dupliquerait beaucoup de code et poserait des problèmes de maintenance.

HÉRITAGE

Il convient dans le cas de notre exemple d'établir une super-classe `Personnage` ayant les caractéristiques communes à tous les personnages comme attributs, par exemple nom, durée de vie ou méthode `rencontrer(Personnage&)`. Les sous-classes `Voleur` et `Guerrier` sont des versions spécifiques de la classe `Personnage`: elles ont ses caractéristiques. On dit que ces sous-classes héritent de la classe `Personnage`. Les sous-classes peuvent avoir des éléments spécifiques comme une arme pour le `Guerrier` ou le fait de voler pour le `Voleur`.

En programmation, l'**héritage** permet de regrouper des caractéristiques communes dans une **super-classe** dont héritent des **sous-classes** qui en sont des versions enrichies, étendues. Cet outil permet d'établir une relation « **est-un** »: si C1 hérite de C, alors on dit que « C1 est un C », c'est-à-dire que toute instance de C1 est aussi une instance de C, comme le montre l'exemple de la figure 3. De ce fait, l'ensemble des attributs et méthodes de C (sauf le constructeur et destructeur) est disponible dans C1. C1 est **enrichie** par des attributs et méthodes supplémentaires et **spécialisée** si elle redéfinit des méthodes héritées de C.

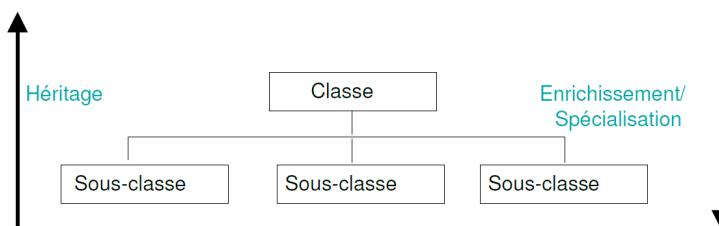


FIGURE 1

3:10

15:20

Schéma des relations d'héritage.

```
Personnage p;
Guerrier g;
// ...
p = g;
// ...
void afficher(Personnage const&);
// ...
afficher(g);
```

FIGURE 2

5:30

15:20

Une instance g de Guerrier est aussi une instance de Personnage. On peut donc copier g dans le Personnage p; en revanche, seule sa partie Personnage sera copiée dans p et non pas son attribut arme, par exemple. L'opération inverse n'est pas possible.



Dans l'exemple du jeu, les sous-classes `Voleur` et `Guerrier` sont des `Personnage` et possèdent donc toutes les caractéristiques de cette classe sans qu'il soit nécessaire de les redéfinir, mais `Guerrier` possède aussi un attribut `arme` et `Voleur` redéfinit la méthode `rencontrer(Personnage&)`.

L'héritage permet donc d'organiser et clarifier le code en explicitant les relations structurelles entre des classes, tout en limitant la redondance de code. L'héritage ne doit cependant jamais représenter une relation «possède-un», dans le même sens qu'une `Guerrier` possède une `arme`: ce lien doit être établi par l'encapsulation, par le fait de déclarer un attribut.

TRANSITIVITÉ ET HIÉRARCHIE

L'héritage est transitif, c'est-à-dire que si une super-classe hérite d'une super-super-classe, alors la sous-classe possède également tous les attributs et méthodes de cette dernière.

À travers l'héritage, les classes s'enrichissent progressivement et s'organisent selon un réseau de dépendances, une structure arborescente illustrée par la figure 3. Ces relations «est-un» définissent une **hiérarchie de classe** entre les classes les plus générales, classes **parentes** en haut (par exemple la classe `Personnage`) et les classes **enfant** spécialisées et enrichies, en bas.

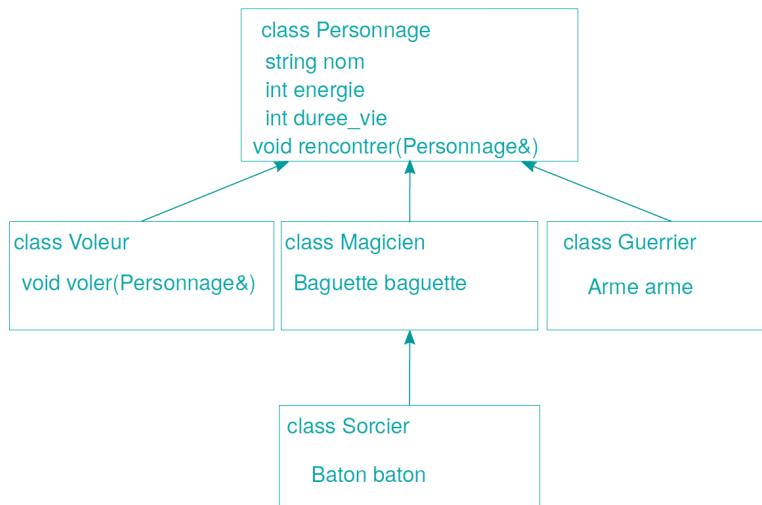


FIGURE 3

11:00

15:20

Relations d'héritage (explicitées par une flèche vers la super-classe) entre différentes classes.

DÉFINITION D'UNE SOUS-CLASSE

Pour faire hériter une classe d'une super-classe, on emploie la syntaxe suivante lors de sa déclaration:

```

class NomSousClasse : public NomSuperClasse {
    // Déclaration des attributs et méthodes spécifiques à la sous-classe
};
  
```



16. HÉRITAGE: DROIT PROTÉGÉ

DROIT D'ACCÈS PROTÉGÉ

Dans le cadre d'une relation d'héritage, la sous-classe dispose aussi des membres privés de la super-classe mais n'y a pas accès. En effet, le droit d'accès privé limite la visibilité à l'enceinte de la classe. Il existe un troisième type d'accès au sein d'une hiérarchie de classe : le **droit d'accès protégé**.

Le droit d'accès protégé assure la visibilité des membres d'une classe dans toutes les classes de sa descendance et se désigne par le mot-clé `protected` dans la même syntaxe d'utilisation que `public` et `private`. Le niveau d'accès protégé est une extension du niveau privé, qui accorde des droits d'accès privilégiés uniquement à toutes les sous-classes. On pourra donc manipuler des membres protégés dans celle-ci; en revanche, ils apparaîtront comme privés à tout autre endroit du code.

ACCÈS ET PORTÉE

Un membre protégé est uniquement accessible dans les sous-classes **dans leur propre portée**.

Pour rappel, la **portée de classe** permet de manipuler directement tous les attributs et méthodes de quelque instance d'une classe (courante ou non) au sein de sa définition. Par exemple, si une classe `B` admet un attribut privé `b`, alors on pourra accéder, dans les méthodes de `B`, au `b` de l'instance courante, `this->b`, mais également d'une autre instance `autreB` passée en paramètre, par `autreB.b`. On dit que `b` est de portée `B` pour signifier qu'il est accessible dans la classe `B` à travers tout objet de type `B`.

La figure 1 illustre l'accès possible d'une classe `B` à l'attribut protégé `a` de la super-classe `A`, mais uniquement dans sa portée. Un attribut privé de `A` n'est pas accessible en dehors de la super-classe. Au sein de `B`, on peut accéder au `a` protégé de l'instance courante ou d'une autre instance `autreB` de la même classe, mais pas à celui d'un objet d'une autre classe comme `A`: il s'agirait alors d'un accès externe à un attribut protégé, interdit.

```

class A {
    // ...
protected: int a;
private:   int prive;
};

class B: public A {
public:
    // ...
void f(B autreB, A autreA, int x) {
    a      = x; // OK A::a est protected => accès possible
    prive = x; // Erreur : A::prive est private

    a += autreB.prive; // Erreur (même raison)
    a += autreB.a     ; // OK : dans la même portée (B::)

    a += autreA.a     ; // INTERDIT ! : this n'est pas de la même
                        // portée que autreA
}
}

```

FIGURE 1

7:00

10:51

Exemple de la portée des droits d'accès protégés.

Si la classe `Personnage` de la leçon 15 admettait l'attribut protégé `énergie`, alors la classe `Guerrier` aurait accès à l'`énergie` de toute instance `Guerrier` mais pas d'un personnage d'un autre type.

Pour conclure, alors que les membres publics sont accessibles à tous les utilisateurs d'une classe et que les membres privés ne sont accessibles qu'au programmeur de la classe, les membres protégés sont accessibles à tous les programmeurs d'extension par des sous-classes.



17. HÉRITAGE: MASQUAGE

SPÉCIALISATION

La leçon 15 annonçait la possibilité de **spécialiser**, de redéfinir des méthodes de la super-classe : par exemple, cette pratique permettrait de modifier la méthode `rencontrer()` de la super-classe `Personnage` uniquement pour la sous-classe `Guerrier`. On implémenterait donc deux versions de cette méthode, comme dans la figure 1 : une dans la super-classe pour les personnages non guerriers et une autre dans cette sous-classe particulière.

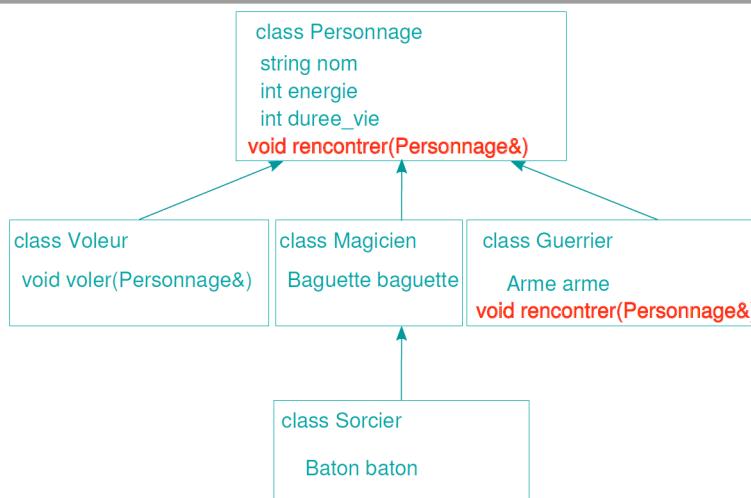


FIGURE 1

1:30

8:25

Redéfinition d'une méthode héritée dans une sous-classe.

Il s'agit donc d'une situation de **masquage**: un appel à la méthode par cette sous-classe renverra à la version spécialisée, qui l'emporte sur la version héritée de la super-classe. Le masquage désigne le partage d'un même nom d'attribut ou de méthode sur plusieurs niveaux d'une hiérarchie de classe.

Le masquage est souvent source de confusion pour les attributs : il consiste à nommer un attribut par le même nom (pas forcément le même type) qu'un attribut hérité, qui masquera ce dernier. En revanche, masquer des méthodes est courant et permet de les spécialiser. La méthode héritée est alors dite la **méthode générale**, appelée par les sous-classes qui ne la masquent pas. La méthode spécifique à la sous-classe est la **méthode spécialisée**.

ACCÈS À UNE MÉTHODE MASQUÉE

Un objet dont un des membres est spécialisé dispose également du membre hérité, bien qu'il ne soit jamais appelé directement pour des raisons de portée. Dans certains cas, il est souhaitable de pouvoir désigner un membre masqué, par exemple pour inclure la méthode générale avant de la compléter par des actions spécifiques dans la définition de la méthode spécialisée. On utilise alors l'opérateur de résolution de portée :

`NomSuperClasse::méthode ou attribut`

Cette syntaxe établit le lien entre le nom du membre et la classe à laquelle il appartient, et permet d'éviter des duplications de code lors de la spécialisation de méthodes.



18. HÉRITAGE: CONSTRUCTEURS (1/2)

L'instanciation d'une classe s'accompagne d'une initialisation des attributs. Cette tâche ne peut plus être intégralement prise en charge par le constructeur d'une sous-classe: celle-ci hérite des attributs, parfois privés, d'une super-classe, qu'elle ne peut pas initialiser. L'initialisation des attributs hérités doit se faire dans la classe où ils sont explicitement définis: chaque constructeur de la sous-classe fait donc appel à un constructeur de la super-classe.

En C++, l'appel au constructeur de la super-classe depuis le constructeur de la sous-classe se fait dans la liste d'initialisation ou section deux points (leçon 6) par la syntaxe suivante:

```
SousClasse(liste de paramètres)
    : SuperClasse(Arguments),
      Attribut1(valeur1),
      ...
{
    // Corps du constructeur
}
```

Le constructeur de la super-classe porte son nom et est placé au début de la section d'appel aux constructeurs des attributs.

```
class FigureGeometrique {
protected:  Position position;
public:
    FigureGeometrique(double x, double y) : position(x, y) {}
    // ...
};

class Rectangle : public FigureGeometrique {
protected:  double largeur; double hauteur;
public:
    Rectangle(double x, double y, double l, double h)
        : FigureGeometrique(x,y), largeur(l), hauteur(h) {}
    // ...
};
```

FIGURE 1

2:30

11:49

Exemple de constructeurs dans une relation d'héritage.



Le constructeur de la sous-classe `Rectangle` de la figure 1 fait par exemple appel au constructeur de la super-classe `FigureGeometrique` dans sa liste d'initialisation, en lui donnant les arguments pour initialiser son argument `position`.

Un autre exemple permet de souligner que les attributs supplémentaires dans une sous-classe ne sont pas nécessaires: on peut définir un carré comme une sous-classe de `Rectangle`, dont les dimensions sont égales. Cette classe `Carre` n'introduit pas d'attributs supplémentaires mais définit son constructeur de façon adaptée, tel que montré en figure 2. La classe `Carre` pourrait spécialiser des manipulateurs hérités comme `set_Hauteur()` et `set_Largeur()` afin de maintenir ses deux attributs égaux. On note que son constructeur fait appel au constructeur de `Rectangle`, avec deux valeurs égales, selon la syntaxe usitée.

```
class Carre : public Rectangle {
public:
    Carre(double taille)
        : Rectangle(taille, taille)
    {}
    /* Et c'est tout !
       (sauf s'il y avait des manipulateurs,
       il faudrait alors sûrement aussi les
       redéfinir)
    */
};
```

FIGURE 2

7:30

11:49

Exemple de constructeurs dans une relation d'héritage qui n'ajoute pas d'attribut.

Un appel explicite au constructeur de la super-classe n'est pas nécessaire si cette dernière possède un constructeur par défaut, puisque l'appel est alors effectué par le compilateur. Si au contraire la classe n'a pas de constructeur par défaut, l'appel doit être explicite, comme dans les exemples précédents, pour éviter des erreurs.

Pour rappel, le constructeur par défaut, qui ne prend pas d'arguments, existe par défaut si la classe n'a pas de constructeur, mais disparaît et doit être réécrit dès la création d'un constructeur. Il est conseillé de toujours déclarer au moins un constructeur pour chaque classe et d'effectuer un appel explicite à un constructeur de la super-classe, même à celui par défaut.

19. HÉRITAGE: CONSTRUCTEURS (2/2)

ORDRE D'APPEL DE CONSTRUCTEURS

Dans une relation d'héritage, la construction d'une sous-classe appelle d'abord le constructeur de la super-classe la plus générale puis, dans l'ordre, les constructeurs des super-classes qui en héritent, avant de terminer par l'initialisation de la partie spécifique de la classe instanciée.

Soit, par exemple, une classe **C**, héritant d'une classe **B** qui elle-même hérite d'une classe **A**, relation illustrée par la figure 1. La construction d'une instance de **C** appelle, explicitement ou non, le constructeur de **B** qui appelle celui de **A** (leçon 18). Comme ces appels sont placés en première position dans la liste d'initialisation, le constructeur de **C** va commencer par exécuter la construction de sa partie **A**, classe la plus générale dont on dérive, en donnant des valeurs à ses attributs **a1** et **a2**. Cette première étape conclue, le constructeur de **B** initialise **b1** et construit ainsi la partie **B** de l'instance. Enfin, le constructeur de **C** peut initialiser la partie spécifique à cette sous-classe, les attributs **c1** et **c2**, ce qui termine la construction de l'instance. On aura bien suivi un schéma de construction de la partie la plus générale à la plus spécifique de la classe.

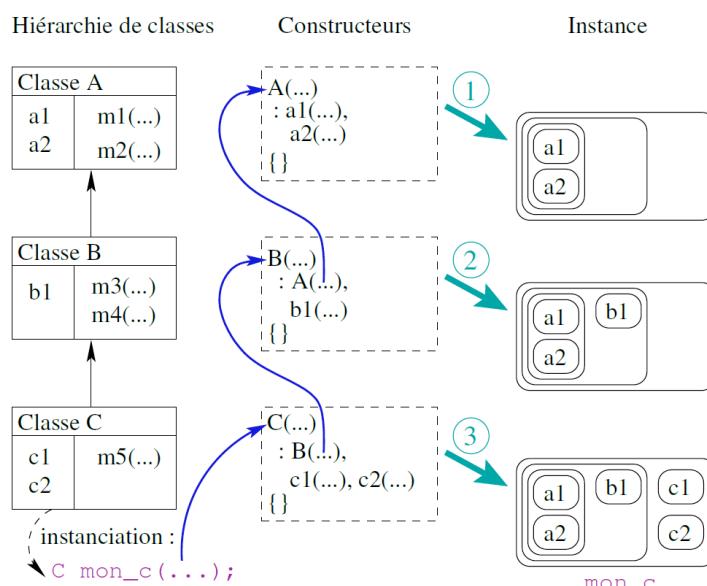


FIGURE 1

0:10

11:39

Illustration de l'ordre d'appel des constructeurs dans une relation hiérarchique.

DESTRUCTEURS

Les destructeurs sont appelés dans l'ordre inverse des constructeurs. Dans l'exemple précédent, on aurait tout d'abord fait appel au destructeur de **C** (dont le constructeur est le dernier complété), puis au destructeur de **B**, qui appelle le destructeur de **A**.



```
Rectangle(Rectangle const& autre)
: FigureGeometrique(autre),
largeur(autre.largeur),
hauteur(autre.hauteur)
{}
```

FIGURE 2

4:00

11:39

Importance de l'appel au constructeur de copie de la super-classe.

CONSTRUCTEURS DE COPIE

Une redéfinition du constructeur de copies d'une sous-classe s'accompagne toujours d'un appel explicite au constructeur de copies de la super-classe ; sinon, son constructeur par défaut est appelé et la copie est souvent mal réalisée.

Un exemple d'une telle situation est la redéfinition du constructeur de copie d'une classe `Rectangle` héritant d'une classe `FigureGeometrique`. Cette dernière classe possède un attribut `Position` à deux coordonnées et est assortie notamment d'un constructeur par défaut, qui initialise l'attribut au vecteur nul. De ce fait, si l'on ne spécifie, dans le constructeur de copie de `Rectangle`, que les valeurs des attributs spécifiques à `Rectangle`, sans appeler le constructeur de copie de `FigureGeometrique`, alors un appel implicite sera fait à son constructeur par défaut. La position du nouveau `Rectangle` sera le vecteur nul et non pas la position du `Rectangle` copié. Au contraire, en appelant, comme dans la figure 2, le constructeur de copie de `FigureGeometrique` avec l'instance copiée en argument (étant un `Rectangle`, elle est aussi une `FigureGeometrique`), l'attribut propre à cette super-classe, la position, sera copiée dans la nouvelle instance.

HÉRITAGE DES CONSTRUCTEURS

Les constructeurs ne sont pas hérités dans une relation d'héritage ; mais depuis C++ 2011, il est possible de demander à ce qu'ils le soient. On place pour cela la ligne suivante dans la définition de la sous-classe :

```
using SuperClasse::SuperClasse;
```

Cette ligne force l'héritage de tous les constructeurs de la super-classe, de sorte que la sous-classe peut être construite avec les mêmes arguments. On prend pour exemple une sous-classe `B` héritant d'une super-classe `A` qui serait dotée d'un constructeur avec un `int` et d'un avec deux `double` comme arguments. Si le concepteur de `B` écrit `using A::A` dans sa définition, il crée deux constructeurs de `B`, dont les paramètres sont un `int` ou deux `double`, qui initialiseront les attributs respectifs de `A`.

Il s'agit d'une pratique peu recommandée puisque les constructeurs de la super-classe n'initialisent pas les attributs de la sous-classe : on limitera son utilisation, si nécessaire, à des sous-classes qui n'ont pas de nouvel attribut.

20. COPIE PROFONDE

COPIE DE SURFACE

En C++, le compilateur fournit une version par défaut minimale des constructeurs et destructeurs, s'ils ne sont pas définis. Le constructeur de copie par défaut effectue une **copie de surface**, qui consiste à copier membre à membre la valeur de chaque attribut. Cette copie est d'ordinaire suffisante, mais elle peut poser problème notamment quand des attributs sont des pointeurs.

La leçon 8 conseillait de toujours considérer ensemble la redéfinition du constructeur de copie, du destructeur et de l'opérateur d'affectation. La figure 1, qui définit une classe `Rectangle` avec deux pointeurs comme attributs, illustre l'importance de ce conseil: le destructeur est bien redéfini, pour pouvoir désallouer des zones mémoires octroyées à la construction, mais pas le constructeur de copie, ce qui posera problème, comme expliqué ci-dessous.

```
class Rectangle {  
private:  
    double* largeur; // aïe, un pointeur !  
    double* hauteur;  
public:  
    Rectangle(double l, double h)  
        : largeur(new double(l)), hauteur(new double(h)) {}  
    ~Rectangle() { delete largeur; delete hauteur; }  
    double getLargeur() const;  
    double getHauteur() const;  
    // ...  
};
```

FIGURE 1

3:45

16:33

Exemple de classe dont il faut redéfinir le constructeur de copie.

On pourrait concevoir, par exemple, une fonction `afficher_largeur(Rectangle tmp)` ayant pour but d'afficher la largeur du `Rectangle` passé par valeur en argument. En appelant `afficher_largeur(r)`, le `Rectangle r` serait copié dans `tmp` par une copie de surface, puisque le constructeur de copie est celui par défaut. Ce sont les adresses contenues dans les pointeurs qui seront copiées: les attributs de `tmp` pointent vers les mêmes zones mémoires, les mêmes variables que `r`. Après l'exécution du corps de la fonction, le paramètre `tmp`, à portée locale, fait automatiquement appel au destructeur: celui-ci désalloue les zones mémoires associées aux attributs de `tmp` mais également à ceux de `r`. Cet objet est corrompu par l'appel `afficher_largeur(r)`, un risque de *Segmentation Fault*.

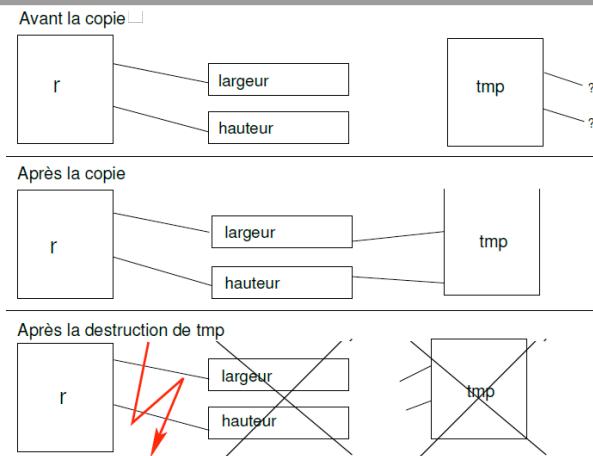


FIGURE 2

11:30

16:33

Illustration de l'état des zones mémoires lors d'un appel à la fonction `afficher_largeur`.



COPIE PROFONDE

Dans des situations où les attributs sont des pointeurs, le constructeur de copie doit être redéfini pour réaliser une **copie profonde** qui ne copie pas les adresses mais duplique les zones pointées. Les attributs d'un objet copié devraient pointer vers des variables à même valeur mais stockées dans des zones mémoires distinctes de l'objet d'origine. Cette indépendance implique que la manipulation de l'objet copié (et notamment, sa destruction) n'a plus aucune incidence sur l'objet d'origine.

Dans le constructeur de copie, il convient donc d'allouer de nouvelles zones mémoires, qui contiennent les valeurs pointées par l'objet à copier. Un exemple est proposé dans la figure 3. Dans une relation d'héritage, il faudra explicitement faire appel au constructeur de la super-classe, comme vu en leçon 19.

```

class Rectangle {
public:
    Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
    Rectangle(const Rectangle& obj);
    ~Rectangle();
    // Note: il faudrait aussi redefinir operator= !
private:
    double* largeur;      double* hauteur;
};
// constructeur de copie
Rectangle::Rectangle(const Rectangle& obj)
    : largeur(new double(*(obj.largeur))),
    hauteur(new double(*(obj.hauteur)))
{}
// destructeur
void Rectangle::~Rectangle() {
    delete largeur;
    delete hauteur;
}

```

FIGURE 3

15:25 16:33

Redéfinition du constructeur de copie de la classe Rectangle.

Pour les mêmes raisons, si une classe contient des pointeurs, il est parfois important de redéfinir, d'une façon similaire, l'opérateur d'affectation `=`: sa version par défaut effectue aussi une copie de surface. On n'oubliera pas non plus de considérer le cas du destructeur.

21. POLYMORPHISME ET RÉSOLUTION DYNAMIQUE DES LIENS

POLYMORPHISME

La dernière notion fondamentale de la programmation orientée objet est le **polymorphisme**. Il permet à un même code de s'adapter aux types des données auxquels il s'applique. Ainsi, il rend le code générique, écrit de façon unifiée pour différents types de données.

En prenant la hiérarchie de classe présentée en leçon 15, on voudrait faire rencontrer à un joueur un ensemble de `Personnage` contenus dans un tableau : on écrirait une boucle itérative sur ses éléments, appelant tout à tour leur méthode `rencontrer (Personnage)`. Ce tableau peut contenir des instances de sous-classes de `Personnage`, comme des `Guerrier`, dont la méthode `rencontrer` est spécialisée (leçon 17). Grâce au polymorphisme, le même code appliqué à différents personnages s'adapte à la forme de chacun et appelle la fonction spécifique du personnage. On peut alors utiliser un `Voleur` sous la forme d'un `Personnage` tout en conservant ses spécificités, et ainsi unifier des manipulations sur différents types d'objets.

RÉSOLUTION DYNAMIQUE DES LIENS

Dans une hiérarchie de classe, le type est hérité (leçon 15) : on dit qu'une instance de la sous-classe est aussi une instance de la super-classe et cette relation est transitive. L'héritage du type s'applique dans le cadre de l'affectation, mais aussi du passage des arguments comme l'illustre la figure 1. La fonction `faire_rencontrer ()` prend en argument deux personnages qui peuvent être un `Guerrier` et un `Voleur`, par exemple.

On peut supposer que la sous-classe `Guerrier` a redéfini la méthode `rencontrer ()` (leçon 17). Cependant, le paramètre de la fonction de la figure 1 est un `Personnage` et ce sera donc la méthode générale qui sera appelée, même si l'objet `g` possède une méthode spécialisée. En C++, par défaut, le type de la variable détermine la méthode à appeler. On parle de **résolution statique des liens** parce que le choix de la méthode se fait statiquement avant l'exécution.

```
class Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};

void faire_rencontrer(
    Personnage const& un,
    Personnage const& autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;

    faire_rencontrer(g, v);
}
```

FIGURE 1

8:00

10:58

Exemple de résolution statique des liens : l'affichage à l'exécution de ce programme sera « Bonjour ! ».



Le polymorphisme est le fait que, lors d'affectations ou de passages d'arguments, des instances d'une sous-classe soient substituables aux instances de leurs super-classes tout en gardant leurs propriétés propres. Dans l'exemple précédent, ceci permettrait d'appeler la méthode spécifique du personnage passé en argument. La méthode à invoquer est choisie pendant l'exécution, en fonction de la nature réelle des instances : on a recours à la **résolution dynamique des liens**.

Pour avoir recours à la résolution dynamique des liens, il est nécessaire de réunir les conditions suivantes :

- les méthodes concernées doivent être déclarées comme **virtuelles** et
- elles doivent s'exercer sur les instances réellement concernées grâce à des **références** ou des **pointeurs**.



22. POLYMORPHISME : MÉTHODES VIRTUELLES

En C++, pour permettre la résolution dynamique des liens (leçon 21), il faut utiliser des méthodes virtuelles au travers de références ou de pointeurs. Ainsi, le choix de la méthode se fait en fonction du type réel de l'instance.

MÉTHODES VIRTUELLES

Une méthode à résoudre dynamiquement doit être déclarée comme **virtuelle**, en précédant son prototype par le mot-clé **virtual**. On effectue ce signalement dans la classe la plus générale qui admet cette méthode, par exemple dans la classe **Personnage** pour le code donné en leçon 21. On note enfin que toute spécialisation d'une méthode virtuelle, dans une sous-classe, est aussi virtuelle par transitivité, même sans spécifier explicitement le mot-clé **virtual**.

Pour permettre du polymorphisme dans le code de la figure 1 de la leçon 21, on doit déclarer comme virtuelle, dans la super-classe, la méthode **rencontrer**: dans la figure 1, la fonction **faire_rencontrer** fait alors appel à la méthode spécifique du **Guerrier**.

```
class Personnage {
public:
    // ...
    virtual void rencontrer(
        Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};

void faire_rencontrer(
    Personnage const& un,
    Personnage const& autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;

    faire_rencontrer(g, v);
}
```

FIGURE 1

6:45

17:25

Retour sur l'exemple de la leçon 21 : version polymorphe.

RÉFÉRENCES ET POINTEURS

Pour activer la résolution dynamique des liens, il est également essentiel que le passage des arguments des méthodes virtuelles soit fait par le biais de références ou de pointeurs : elles opèrent ainsi sur les instances réelles.

Par exemple, si l'argument **un** de la fonction **faire_rencontrer** en figure 1 était passé par valeur, le polymorphisme serait impossible : lors de l'appel, l'objet **g** du **main** serait copié dans une variable de type **Personnage** en perdant ses spécificités comme son attribut **arme** ou la spécialisation de sa méthode **rencontrer**. Malgré la virtualisation de cette méthode, ce serait donc toujours la version générale qui serait invoquée par la fonction **faire_rencontrer**.



L'exemple des figures 2 et 3 illustre l'emploi de pointeurs dans le cadre du polymorphisme. Une sous-classe **Dauphin** y hérite d'une super-classe **Mammifere**. Le code de la Figure 3 déclare tout d'abord un pointeur sur **Mammifere**, qui reçoit l'adresse d'un **Dauphin**. L'objet alloué dynamiquement fait appel, dans l'ordre, au constructeur de sa super-classe puis de sa classe spécialisée : le premier affichage sera donc « Un nouveau mammifère est né ! », suivi du message « Couic, Couic ! ».

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

class Mammifere {
public:
    Mammifere() { cout << "Un nouveau mammifère est né !" << endl; }
    virtual ~Mammifere() { cout << "Un mammifère est en train de mourir :" << endl; }
    void manger() const { cout << "Miam... croumf !" << endl; }
    virtual void avancer() const { cout << "Un grand pas pour l'humanité." << endl; }
};

class Dauphin : public Mammifere {
public:
    Dauphin () { cout << "Coui, Couic !" << endl; }
    ~Dauphin() { cout << "Flipper, c'est fini..." << endl; }
    void manger() const { cout << "Sglups, un poisson." << endl; }
    void avancer() const { cout << "Je nage." << endl; }
};
```

FIGURE 2

7:15

17:25

Illustration du concept de virtualité des méthodes (à suivre figure 3).

Que produit le code suivant ?

```
int main() {
    Mammifere* lui(new Dauphin());
    lui->avancer();
    lui->manger();
    delete lui;
    return 0;
}
```

Un nouveau mammifère est né !
 Coui, Couic !
 Je nage.
 Miam... croumf !
 Flipper, c'est fini...
 Un mammifère est en train de mourir :(

Mammifere::Mammifere()
 Dauphin::Dauphin()
 Dauphin::avancer()
 Mammifere::manger()
 Dauphin::~Dauphin()
 Mammifere::~Mammifere()

FIGURE 3

13:10

17:25

Illustration du concept de virtualité des méthodes (suite).

Dans la seconde ligne du **main**, puisque la méthode **avancer** est virtuelle et qu'elle a accès à l'instance réelle par le biais du pointeur, la résolution dynamique des liens appelle la méthode **avancer** du **Dauphin**, pour afficher « Je nage ». Au contraire, une invocation de la méthode non virtuelle **manger** est résolue statiquement : la méthode liée au type de la variable, **Mammifere**, affiche « Miam... croumf ! ».

Enfin, la dernière ligne désalloue et donc détruit la zone mémoire pointée. Or, le destructeur de **Mammifere** est virtuel : le code appelle d'abord le destructeur spécifique à **Dauphin**, qui affiche « Flipper, c'est fini... », puis le destructeur de **Mammifere** (dans l'ordre inverse des constructeurs), responsable du dernier message. Au contraire, si le destructeur de la classe **Mammifere** n'avait pas été déclaré comme virtuel, la destruction de l'objet serait résolue statiquement et invoquerait immédiatement le destructeur de **Mammifere**. La part de l'objet de type **Dauphin** ne serait alors pas détruite.



CONSTRUCTEUR ET DESTRUCTEUR

Pour éviter qu'une destruction d'objet ne soit que partielle comme dans l'exemple précédent, il est conseillé de toujours déclarer les destructeurs comme virtuels. Au contraire, puisqu'un constructeur est chargé d'initialiser l'instance courante, il ne peut pas être virtuel. Enfin, s'il appelle dans son corps des méthodes virtuelles, la virtualité de ces méthodes est ignorée.

Cette notion est illustrée par le code de la figure 4, dans lequel une sous-classe **B** hérite, et redéfinit la méthode virtuelle **f**, d'une super-classe **A**. La première ligne du programme principal fait appel au constructeur par défaut de la classe **A**: le premier affichage sera **A::f()**. La deuxième ligne crée un objet **B** par son constructeur par défaut, lequel appelle le constructeur par défaut de sa super-classe **A**. La méthode **f** s'applique alors à un objet de type **B**; cependant, l'aspect virtuel de la méthode est ignoré dans un constructeur, qui appelle toujours la méthode de la classe courante. Le second affichage sera aussi **A::f()**. Au contraire, les lignes suivantes font bien du polymorphisme puisque l'on invoque la méthode virtuelle **f**, en dehors du constructeur, par un pointeur sur une instance de **B**: le dernier message affiché est **B::f()**.

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

class A {
public:
    A() { f(); }
    virtual void f() const { cout << "A::f()" << endl; }
};

class B : public A {
public:
    virtual void f() const { cout << "B::f()" << endl; }
};

int main()
{
    A a;
    B b;
    A* pa(&b);
    pa->f();
    return 0;
}
```

FIGURE 4

16:00

17:25

L'aspect polymorphique est ignoré dans les constructeurs, qui appellent toujours les méthodes des instances courantes.



23. MASQUAGE, SUBSTITUTION ET SURCHARGE

DISTINCTION ET EXEMPLE

On parle de **surcharge** (overloading) quand des fonctions ou des méthodes dans la même portée portent le même nom, mais se distinguent par des paramètres différents. Un **masquage** (shadowing) de méthodes a lieu lorsque des méthodes de portée différente ont le même nom : celles dont la portée est la plus proche masquent les méthodes plus lointaines, indépendamment de leurs paramètres. Il suffit d'une méthode avec le même nom pour masquer plusieurs méthodes surchargées dans une autre portée. La **substitution** (overriding) de méthodes virtuelles désigne la redéfinition d'une méthode virtuelle héritée d'une super-classe, dans le but de résoudre dynamiquement des liens. À nouveau, la substitution d'une seule méthode virtuelle, même avec des paramètres différents, masque toutes les autres qui portent le même nom.

Dans le cadre de méthodes virtuelles, une confusion sur la distinction entre les trois concepts précédents est courante : l'exemple de la figure 1 vise à l'éclairer. Deux classes **B** et **C** héritent d'une super-classe **A**, et présentent différentes versions de la méthode `m1()`.

```

class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};

```

FIGURE 1

4:00

19:11

Différentes classes dont les méthodes `m1()` sont en situation de surcharge, masquage ou substitution.

Dans la définition de **A**, les deux méthodes `m1()` diffèrent par leurs paramètres et sont de même portée : il s'agit d'une situation de surcharge. La classe **B** redéfinit une méthode virtuelle héritée : on parle donc de substitution de la méthode `m1(string)`. Par les règles de résolution de portée, les deux méthodes héritées de **A** sont masquées par la fonction redéfinie dans **B**, dans le programme en figure 2, il est licite d'appeler la nouvelle méthode `m1(string)` de **B** mais pas la méthode `m1(int)`, qui pourtant existe comme le montre l'appel par l'opérateur de résolution de portée.

La classe **C** introduit une troisième méthode `m1()`, avec un `double` comme paramètre. Par les mêmes règles de résolution de portée, cette nouvelle méthode masque les deux autres héritées de **A**. Dans le programme, on ne peut pas appeler ces méthodes sans l'opérateur de résolution de portée ; écrire `m1()` avec un entier comme paramètre sera un appel à `C::m1(double)` par conversion de l'entier en réel.



```
int main() {
    B b;
    //b.m1(2);      // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2);   // ... mais elle est bien là
    b.m1("2");

    C c;
    c.m1(2); // Attention ici : c'est celle avec double !!
    //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2);  // OK, et là c'est celle avec int

    return 0;
}
```

FIGURE 2

6:00

19:11

Exemples de masquage.

Le programme donné en figure 3 illustre enfin la notion de substitution. Toutes les méthodes `m1()` sont virtuelles : en employant un pointeur `pa` sur une instance de `A`, on peut permettre la résolution dynamique des liens. Par exemple, un appel à `m1(string)` d'une instance de `B` pointée par `pa` renverra à la méthode substituée dans `B`. Quand le polymorphisme n'a pas lieu, le type de la variable `prime:pa` accède aux méthodes de la portée de `A` et donc peut effectuer un appel direct à `m1(int)`. Pour la même raison, si `pa` pointe sur une instance de `C`, en passant un double comme paramètre à `m1()`, la méthode appelée sera `A::m1(int)` après conversion du réel en entier. Puisque `pa` est un pointeur sur `A`, il s'agit de l'unique méthode dans la portée de cette classe qui puisse prendre un `double` en paramètre. La méthode `C::m1(double)` n'est pas accessible par ce pointeur.

```
int main() {
    B b;
    C c;
    A* pa(nullptr);

    pa = &b;
    pa->m1("2");
    pa->m1(2);    // OK (nous sommes dans A::)

    pa = &c;
    pa->m1(2.1); /* Attention ici : c'est celle avec int !!
                    * Nous sommes dans A::                      */
    // pa->C::m1(2.1); // Impossible ! A n'hérite pas de C !!

    return 0;
}
```

FIGURE 3

9:00

19:11

Exemples de substitutions.



OVERRIDE ET FINAL

Depuis C++ 2011, deux mots-clés permettent au programmeur de mieux spécifier ses intentions lors de redéfinitions ou de surcharges. Placés après le prototype de la méthode concernée, le mot `override` indique que l'on redéfinit une méthode virtuelle héritée d'une super-classe et le mot `final` empêche toute substitution future de la méthode dans des sous-classes. Il est très conseillé d'utiliser le mot `override` pour limiter des erreurs ou oubli.

La figure 4 propose un exemple d'utilisation de ces outils dans le cadre d'une relation d'héritage entre une super-classe `A` et une sous-classe `B`. En premier lieu, on substitue la méthode virtuelle `f1()` dans `B`: le mot `override` est correctement utilisé. Au contraire, dans les lignes suivantes, on pense redéfinir les fonctions `f1()` et `f2()` mais le programmeur fait une faute de frappe ou un oubli: grâce à l'indication fournie par `override`, le compilateur vérifie s'il effectue bien une substitution de méthode virtuelle. Puisqu'aucun prototype identique n'existe dans la classe parente, il signale l'erreur. Sans utiliser `override`, le programmeur aurait créé sans le vouloir deux nouvelles fonctions indépendantes. Enfin, alors que l'on veut substituer dans `B` une méthode `f3()` tout en oubliant de la déclarer comme virtuelle, le mot `override` permet au compilateur de signaler une erreur et non de simplement masquer la méthode héritée. Ce mot constitue donc une protection contre les erreurs.

Enfin, en utilisant le mot-clé `final`, le concepteur de la classe `A` empêche toute redéfinition de la méthode `f4()`: la dernière ligne du code de la classe `B` ne compilerait donc pas. On peut également utiliser `final` pour interdire la création de sous-classes d'une classe, en plaçant ce mot après le nom de la classe dans son prototype. L'usage de ce mot reste cependant limité et souvent peu justifié à ce niveau.

```

class A {
    // ...
    virtual void f1();
    virtual void f2() const;
        void f3();           // non virtuelle (oubli?)
    virtual void f4() final; // pas de redéfinition
};

class B : public A {
    // ...
    virtual void f1() override; // OK
    virtual void f1() override; // Erreur faute de frappe : 1 <-> 1
    virtual void f2() override; // Erreur: a oublié le const
        void f3() override; // Erreur: non virtuelle
    virtual void f4();       // Erreur : f4 était final
};

```

FIGURE 4

13:10

19:11

Illustration de l'emploi des mots réservés `override` et `final`.



24. CLASSES ABSTRAITES

MÉTHODES VIRTUELLES PURES

Au niveau le plus élevé d'une hiérarchie de classe, il est parfois impossible de définir une méthode générale qui devra pourtant exister dans toutes les sous-classes. Par exemple, calculer la surface d'une figure géométrique [FigureFermee](#) quelconque se révèle difficile, tandis que pour des formes plus concrètes de figures fermées comme un carré, on peut la mettre en œuvre. Une telle méthode [surface\(\)](#) au niveau de la super-classe [FigureFermee](#) pourrait toutefois être nécessaire pour calculer un volume ou simplement par souci d'abstraction et d'unification des données.

En utilisant la hiérarchie de la leçon 15, une classe [Jeu](#), présentée en figure 1, comporte un ensemble de personnages et voudrait les afficher spécifiquement selon la sous-classe dont ils sont des instances ([Guerrier](#), [Voleur](#), [Magicien](#)...). Pour le polymorphisme, l'accès aux instances est fait avec des pointeurs (l'attribut est bien un ensemble de pointeurs); il faut également définir une méthode virtuelle [afficher\(\)](#) au niveau de la super-classe [Personnage](#). Plusieurs problèmes se posent alors: on ne sait pas très bien comment afficher un personnage générique; de plus, il faut imposer à chaque sous-classe de substituer explicitement cette méthode.

```
class Jeu {
public:
    // ...
    void afficher() const {
        for (auto un_perso : perso) {
            un_perso->afficher();
            // Tous les personnages doivent pouvoir s'afficher !...
            // ...mais comment ????
        }
    }
    // ...
private:
    vector<Personnage*> perso;
};
```

FIGURE 1

2:35

13:55

Illustration de l'importance des méthodes virtuelles pures.

Une **méthode virtuelle pure**, ou méthode abstraite, est une méthode qui doit exister et être redéfinie dans chaque sous-classe que l'on souhaite instancier, sans qu'il soit nécessaire de la définir dans la super-classe. Par conséquent, la super-classe ne donne souvent que son prototype sans la définir explicitement. On signale une méthode virtuelle pure en ajoutant = 0 à la fin de son prototype, c'est-à-dire avec la syntaxe donnée en figure 2:

```
virtual Type nom_methode(liste de paramètres) = 0;
```

Il est ainsi possible d'appeler une telle méthode du niveau de la super-classe : la fonction [surface\(\)](#) existe par exemple pour chaque [FigureFermee](#), même si l'on ne sait pas calculer une surface quelconque.



```
class FigureFermee {
public:
    virtual double surface() const = 0;
    virtual double perimetre() const = 0;

    // On peut utiliser une méthode virtuelle pure :
    double volume (double hauteur) const {
        return hauteur * surface();
    }
};
```

FIGURE 2

8:00

13:55

Exemples de déclarations de méthodes virtuelles pures.

CLASSES ABSTRAITES ET HÉRITAGE

Une classe qui contient au moins une méthode virtuelle pure est qualifiée de **classe abstraite** et ne peut pas être instanciée. Toute sous-classe héritant d'une classe abstraite l'est aussi tant qu'elle ne redéfinit pas toutes ses méthodes virtuelles pures héritées.

La figure 3 illustre cette notion: la classe `Cercle` n'est pas abstraite puisqu'elle redéfinit les méthodes virtuelles pures héritées de son ascendance (fig. 1), alors que la classe `Polygone` le reste puisqu'elle ne donne pas de définition de `surface()`. On ne pourra donc pas déclarer d'instance de `Polygone`.

```
class Cercle: public FigureFermee {
public:
    double surface() const override {
        return M_PI * rayon * rayon;
    }
    double perimetre() const override {
        return 2.0 * M_PI * rayon;
    }
protected:
    double rayon;
};
```

```
class Polygone: public FigureFermee {
public:
    double perimetre() const override {
        double p(0.0);
        for (auto cote : cotes) {
            p += cote;
        }
        return p;
    }
protected:
    vector <double> cotes;
};
```

FIGURE 3

12:00

13:55

Héritage des classes abstraites. `Cercle` n'est pas une classe abstraite. `Polygone` reste par contre une classe abstraite.

Les classes abstraites sont idéales pour construire la racine des arbres d'héritage. Grâce à l'outil des méthodes virtuelles pures, le polymorphisme complète l'abstraction. Elles permettent de définir des concepts génériques communs à toutes les sous-classes mais trop abstraits pour être codés à haut niveau.



25. COLLECTIONS HÉTÉROGÈNES

Les **collections hétérogènes** sont une application importante du polymorphisme. Il s'agit d'ensemble d'objets d'une même super-classe qui sont traités de façon polymorphe, spécifique.

EXEMPLE DE COLLECTION HÉTÉROGÈNE

Dans la figure 1 de la leçon 24, on souhaite afficher dans une classe `Jeu` un ensemble d'instances de `Personnage`, en fonction de leur appartenance à leur sous-classe `Guerrier`, `Voleur`, `Magicien`... : cette collection est hétérogène. Une approche possible d'implémentation est de séparer ces instances en plusieurs collections de même type (un tableau de `Guerrier`, un tableau de `Voleur`...) et de les afficher distinctement selon leur sous-classe : on parle alors de collections spécifiques. Une autre conception de la classe `Jeu` peut demander un traitement générique des instances, par un seul tableau de `Personnage`, et une méthode `afficher()` qui effectue un traitement adapté à leur sous-type.

Pour permettre la résolution dynamique des liens, il faut que cette méthode soit virtuelle mais aussi que les instances soient accédées par le biais de pointeurs ou de références. Il est conseillé de toujours préférer des références aux pointeurs lorsque c'est possible ; cependant, on peut ne pas réaliser de tableaux de références. Dans le cadre d'une collection hétérogène, on utilise donc un tableau de pointeurs vers des instances de la super-classe. La classe `Jeu` aurait un attribut de type `vector<Personnage*>` ou mieux encore `vector<unique_ptr<Personnage>>`.

Pour poursuivre, on peut définir une méthode pour ajouter un élément à ce tableau : celle-ci prendrait un pointeur de `Personnage` comme paramètre, alloué dynamiquement lors de l'appel. Pour ajouter un `Guerrier` au `Jeu`, on écrirait donc :

```
jeu.ajouter_personnage(new Guerrier(...));
```

La définition de cette méthode est donnée en figure 1 : par précaution, un test vérifie si le pointeur contient une adresse valable, puis il est converti en `unique_ptr` et ajouté au tableau dynamique de l'instance courante.

Une autre méthode à implémenter dans la classe `Jeu` est responsable de l'affichage de l'ensemble des personnages. Si chaque sous-type de `Personnage` a substitué la méthode `afficher()` polymorphe, une boucle sur le tableau de pointeurs, détaillée en figure 1, permet la résolution dynamique des liens. On notera l'importance de passer les éléments du tableau par référence : les `unique_ptr` ne peuvent en aucun cas être copiés ; dans le cas contraire, plusieurs pointeurs partageraient la même zone mémoire. Dans cette méthode, la référence peut être constante puisqu'aucune valeur n'est modifiée par l'affichage.

```
void Jeu::ajouter_personnage(Personnage* nouveau) {
    if (nouveau != nullptr) {
        personnages.push_back(unique_ptr<Personnage>(nouveau));
    }
}

void Jeu::afficher() const {
    for (auto const& quidam : personnages) {
        quidam->afficher();
    }
}
```

FIGURE 1

11:35

17:10



POINTEURS ET INTÉGRITÉ DES DONNÉES

Pour garantir l'intégrité d'une collection hétérogène, la durée de vie des éléments pointés doit être au moins aussi longue que celle des pointeurs. Un pointeur sur un objet déjà détruit poserait de nombreux problèmes. La figure 2 est un exemple d'une telle faute de conception. Pour ajouter un magicien à un *Jeu*, dans une fonction `creer_magicien()`, le programmeur crée une instance `mago` de *Magicien*, dont il donne l'adresse à la méthode `ajoute_personnage()` définie précédemment. La variable `mago`, cependant, est locale: elle cessera d'exister au terme de l'exécution de la fonction, contrairement au pointeur contenant son adresse stockée dans *Jeu*.

L'allocation dynamique est l'outil qui permet de préserver la zone mémoire allouée aussi longtemps que la collection qui la contient. Une définition robuste de la méthode `creer_magicien()` est proposée en figure 3, grâce à l'utilisation de `new`: la zone mémoire nouvellement allouée existe jusqu'à l'emploi du mot `delete`. Dans une collection faite de `unique_ptr`, la désallocation sera faite automatiquement: pour cette raison, il est recommandé de les favoriser aux pointeurs à la C. Ils permettent également d'assurer qu'un objet n'est pointé que par un pointeur à la fois, ce qui limite les erreurs de manipulation.

```
void creer_magicien(Jeu& jeu) {
    Magicien mago(...);
    jeu.ajouter_personnage(&mago);
}
// ...
int main() {
    Jeu mon_jeu;
    creer_magicien(mon_jeu);
    mon_jeu.afficher(); // ouille !
    return 0;
}
```

FIGURE 2

14:00

17:10

Erreur de conception par ajout d'un pointeur sur une variable locale.

```
// définition robuste de la fonction creer_magicien
void creer_magicien(Jeu& jeu) {
    jeu.ajouter_personnage(new Magicien(...));
}
```

FIGURE 3

15:45

17:10

Solution à l'erreur de la figure 2: allocation dynamique du personnage ajouté.



26. COLLECTIONS HÉTÉROGÈNES: COMPLÉMENTS AVANCÉS

POINTEURS «À LA C», POINTEURS INTELLIGENTS

La leçon 25 conseillait de représenter les collections hétérogènes, ensembles d'objets polymorphiques, par des tableaux dynamiques de `unique_ptr` pour garantir une meilleure intégrité des données. Il est également possible d'utiliser des pointeurs à la C.

L'exemple donné dans la leçon précédente et en figure 1 est celui d'une classe `Jeu`, qui a pour attribut un tableau dynamique de pointeurs intelligents sur des personnages; la seconde déclaration de cette classe, visible en figure 2, utilise des pointeurs à la C. La méthode `ajouter_personnage()` en est simplifiée : elle reçoit en paramètre un pointeur sur `Personnage`, résultant d'une allocation dynamique de sous-classe par le mot-clé `new`, qui sera ajouté au tableau attribut après vérification de sa validité. La conversion du pointeur en `unique_ptr`, nécessaire en figure 1, n'a pas lieu d'être dans la seconde version. La définition méthode `afficher()` est identique à la précédente; il n'est même plus nécessaire de passer les pointeurs par référence puisque ceux-ci peuvent être partagés.

```
class Jeu {
public:
    void afficher() const;
    void ajouter_personnage(Personnage*);

private:
    vector<unique_ptr<Personnage>> personnages;
};

void Jeu::afficher() const {
    for (auto const& quidam : personnages) {
        quidam->afficher();
    }
}

void Jeu::ajouter_personnage(Personnage* nouveau) {
    if (nouveau != nullptr) {
        personnages.push_back(unique_ptr<Personnage>(nouveau));
    }
}
```

FIGURE 1

3:00

18:25

Exemple complet d'une collection hétérogène utilisant des pointeurs intelligents.



```

class Jeu {
public:
    void afficher() const;
    void ajouter_personnage(Personnage*);

private:
    vector<Personnage*> personnages;
};

void Jeu::afficher() const {
    for (auto quidam : personnages) {
        quidam->afficher();
    }
}

void Jeu::ajouter_personnage(Personnage* nouveau) {
    if (nouveau != nullptr) {
        personnages.push_back(nouveau);
    }
}

```

FIGURE 2

1:50

18:25

Exemple complet d'une collection hétérogène utilisant des pointeurs à la C.

INTÉGRITÉ DES DONNÉES

L'intégrité des données peut poser problème lors de l'emploi de pointeurs dans une classe. Le premier enjeu, mentionné en leçon 25, est de garantir la durée de vie des données pointées au moins aussi longtemps que la classe qui les exploite; cette notion concerne tant les pointeurs intelligents que les pointeurs à la C. Malgré une interface d'utilisation qui peut paraître plus simple, d'autres problèmes se posent lors de l'emploi de pointeurs à la C: la désallocation et le partage de données entre collections. Selon la conception de la classe, il est souvent nécessaire de redéfinir des outils qui le permettent, comme le constructeur de copie, le destructeur et l'opérateur d'affectation.

INTÉGRITÉ DES DONNÉES: DÉSALLOCATION

Les pointeurs à la C, contrairement aux pointeurs intelligents, doivent être manuellement désalloués, tâche du programmeur qui a alloué la zone mémoire en premier lieu.

Dans l'exemple de la figure 2, le programmeur utilisateur alloue de la mémoire lors de ses appels à `ajouter_personnage()`, et est ainsi responsable de la libérer après utilisation. La classe `Jeu` doit donc proposer une méthode le permettant, qui peut par exemple détruire l'ensemble des zones mémoire stockées, comme illustré en figure 3. Cette méthode parcourt l'ensemble des pointeurs du tableau de personnages, les libère, avant de vider le tableau avec la méthode `clear()`.

```

void Jeu::detruire_tout()
{
    for (auto quidam : personnages) {
        delete quidam;
    }
    personnages.clear();
}

```

FIGURE 3

7:20

18:25

Exemple de méthode pour détruire l'ensemble des personnages d'une instance de `Jeu`.



Une autre solution est de libérer uniquement un `Personnage` dont on fournit l'adresse ou l'index en argument, ce que propose la figure 4. Selon la conception de la classe, le code propose ensuite plusieurs façons de supprimer le pointeur stocké afin de ne pas pointer vers une zone mémoire invalide. La première garde le même ordre et le même nombre de personnages dans le tableau: elle signale que le pointeur concerné a disparu en lui affectant la valeur `nullptr`. Faire ce choix impose des précautions dans la manipulation du tableau, qui peut donc contenir des pointeurs nuls. Une deuxième solution est de supprimer efficacement la case contenant le pointeur indiqué, mais sans conserver l'ordre du tableau. Avec la méthode `swap` incluse dans `utility`, elle échange cette case avec le dernier pointeur du tableau, que l'on peut désigner par `personnages.back()`; puis supprime le dernier élément avec la méthode `pop_back()`. Enfin, la troisième solution, plus coûteuse, supprime également la case concernée tout en préservant l'ordre. La méthode `erase()` de la classe `vector` consiste en effet à supprimer l'élément indiqué, avant de recopier les éléments qui le suivent à l'index précédent.

```
void Jeu::detruire_personnage(size_t lequel)
{
    delete personnages[lequel];
    // puis, en fonction des situations :

    // SOIT
    // préserve les index et la taille de la collection
    personnages[lequel] = nullptr;

    // SOIT
    // suppression efficace mais ne préserve pas l'ordre
    swap(personnages[lequel], personnages.back());
    personnages.pop_back();

    // SOIT
    // suppression plus couteuse qui préserve l'ordre
    personnages.erase(personnages.begin() + lequel);
}
```

FIGURE 4

9:00

18:25

Exemple de méthode pour détruire un personnage particulier. Si le test n'est pas effectué avant, il faudrait vérifier la validité de l'index donné en argument.

INTÉGRITÉ DES DONNÉES: PARTAGE DE DONNÉES

Une copie de surface de pointeurs (voir leçon 19) fait pointer plusieurs instances vers les mêmes entités et expose souvent à des erreurs de manipulations. Avec des `unique_ptr`, la question ne se pose pas: la copie est interdite puisque ces pointeurs ne peuvent pas partager une zone mémoire. Lorsque l'on utilise des pointeurs à la C, il faut s'interroger sur la nécessité de redéfinir le constructeur de copie.

On peut prendre comme collection hétérogène un `Dessin`, ensemble de figures, où `Figure` est une classe abstraite avec différentes sous-classes polymorphiques concrètes telles que `Cercle` ou `Rectangle`. Il faut alors s'interroger si le contenu d'un `Dessin` est personnel ou partagé: l'action « modifier le `Cercle` d'un certain dessin » a-t-elle un impact sur d'autres dessins ? Dans cet exemple, partager des figures entre plusieurs dessins a peu de sens: il faudrait plutôt implémenter une copie profonde de la collection `Dessin`. Dans le cas de la classe `Jeu`, un partage du contenu du jeu semble incongru et devra également être limité. En revanche, dans un jeu fait de caméras qui verraiennt des personnages selon un certain angle, chaque caméra aurait une collection hétérogène de `Personnages` visibles, partagés entre elles. La redéfinition du constructeur de copie dépend de la conception.

27. HÉRITAGE MULTIPLE: CONCEPT ET CONSTRUCTEURS

Pour rappel, les notions principales de l'orientée objet sont l'encapsulation et l'abstraction, l'héritage et le polymorphisme. La première consiste à regrouper données et traitements en une même entité et à séparer l'interface d'utilisation des détails d'implémentation. L'héritage établit une relation «est-un» entre différentes classes et permet le polymorphisme d'inclusion: ce dernier outil, nécessitant l'emploi de méthodes virtuelles au travers de références ou de pointeurs, induit une adaptation de l'exécution d'un code selon le type de données traitées.

HÉRITAGE MULTIPLE

L'**héritage multiple** est une extension de l'héritage simple étudié depuis la leçon 15: en C++, une sous-classe peut hériter directement de plusieurs classes parentes. Une sous-classe va donc hériter du type, des attributs et des méthodes (hormis les constructeurs et destructeurs) de toutes ses super-classes.

L'exemple du jeu suivant illustre un contexte d'utilisation de l'héritage multiple, mettant en scène différentes entités comme une **Balle**, une **Raquette**, un **Filet** et un **Joueur**, tous dotés d'une méthode **evolve()**. Cette conception suppose l'implémentation d'une méthode **evolve()** au sein d'une super-classe **Entite** dont héritent les sous-classes mentionnées. De plus, on peut représenter graphiquement certaines entités, ou encore rendre certaines interactives: la figure 1 représente la hiérarchie de classe qui organise cette conception. Une **Balle** ou une **Raquette** sont contrôlables de l'extérieur et ont donc aussi comme classe parente la super-classe **Interactif**, qui peut gérer un signal de la souris; de même, la **Raquette**, la **Balle** et le **Filet** sont toutes de type **Graphique**, classe qui permet la représentation graphique. Il ne serait pas correct de placer les méthodes de dessin et de gestion de l'interaction dans la super-classe **Entite**: l'entité **Joueur**, par exemple, n'est ni dessinable ni interactive.

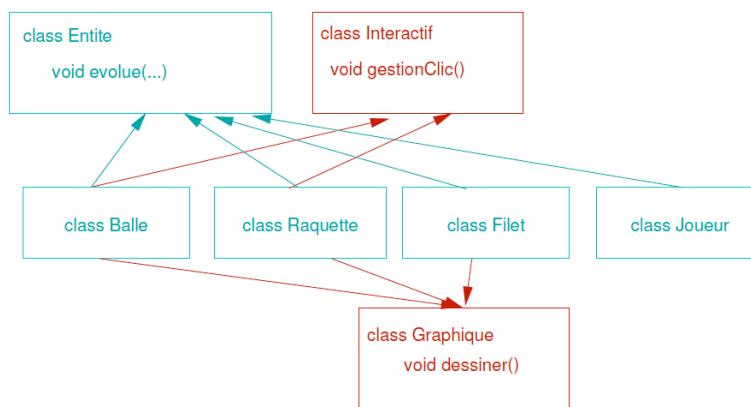


FIGURE 1

4:20

11:14

Exemple d'une hiérarchie d'héritage multiple: certaines sous-classes héritent de plusieurs super-classes.

Enfin, les classes qui gèrent les entrées et sorties en C++ sont organisées grâce à l'héritage multiple, selon une structure en diamant présentée en figure 2.

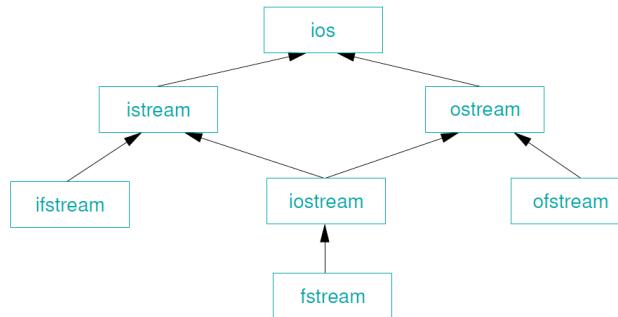


FIGURE 2

5:45

11:14

Exemple d'héritage multiple dans la hiérarchie des classes d'entrées-sorties de C++.

Pour déclarer une relation d'héritage multiple, on indique l'ensemble des super-classes dont hérite une sous-classe dans sa déclaration, séparées par des virgules. La syntaxe est la suivante :

```
class SousClasse : public SuperClasse1, ..., public SuperClasseN
{
    //...
};
```

On note que l'ordre de déclaration des liens d'héritage joue un rôle important dans la construction et la destruction d'une instance.

CONSTRUCTION, DESTRUCTION ET ORDRE DES RELATIONS

De façon analogue à l'héritage simple, l'initialisation d'attributs hérités se fait dans la liste d'initialisation du constructeur de la sous-classe, par un appel au constructeur des classes parentes, selon une telle syntaxe (présentée aussi en figure 3) :

```
SousClasse (liste de paramètres)
    : SuperClasse1 (arguments1),
    ...
    SuperClasseN (argumentsN),
    attribut1 (valeur1),
    ...
    attributK (valeurK)
{ }
```

Si une super-classe admet un constructeur par défaut, une invocation explicite n'est pas obligatoire, mais reste recommandée pour éviter tout oubli.

Une notion importante à saisir est que, dans un contexte d'héritage multiple, l'ordre d'appel des constructeurs parents n'est pas donné par la liste d'initialisation du constructeur de la sous-classe mais par l'**ordre de la déclaration de l'héritage**. L'ordre d'appel des destructeurs se fait toujours dans l'ordre inverse de l'appel des constructeurs.



L'exemple d'une classe `Ovovivipare` qui hérite de deux classes, `Ovipare` et `Vivipare`, est donné en figure 3. Le constructeur d'`Ovovivipare` se charge d'appeler les constructeurs de ces classes parentes dans sa liste d'initialisation; on notera cependant que l'ordre des appels à celles-ci ne se fera pas selon l'ordre de cette liste mais de la déclaration d'héritage. Le constructeur d'`Ovovivipare` appellera en premier lieu le constructeur d'`Ovipare`, et ensuite seulement de `Vivipare`. La plupart des compilateurs donneront un message d'alerte lorsque la liste d'initialisation ne respecte pas l'ordre des liens d'héritage.

```
class Ovovivipare : public Ovipare, public Vivipare {
public:
    Ovovivipare(unsigned int nb_oeufs ,
                 unsigned int duree_gestation ,
                 bool rarete = false );
    virtual ~Ovovivipare();
protected:
    bool espece_rare;
};

Ovovivipare::Ovovivipare(unsigned int nb_oeufs ,
                         unsigned int duree_gestation ,
                         bool rarete)
: Vivipare(duree_gestation), // Mauvais ordre !!
  Ovipare(nb_oeufs),
  espece_rare(rarete)
{}
```

FIGURE 3

9:00

11:14

Illustration de l'ordre d'appel aux constructeurs des super-classes lors d'une relation d'héritage multiple.

28. HÉRITAGE MULTIPLE: MASQUAGE

Dans une relation d'héritage multiple, les droits d'accès s'appliquent comme dans une relation d'héritage simple: une sous-classe peut directement accéder aux membres publics et protégés de ses super-classes. Des situations d'ambiguïté surviennent lorsque plusieurs superclasses contiennent un membre portant le même nom.

Dans l'exemple de la classe `Ovovivipare` donné en leçon 27, on peut supposer que les classes parentes `Vivipare` et `Ovipare` possèdent toutes deux une méthode `afficher()`. Invoquer la méthode `afficher()` d'une instance d'`Ovovivipare` provoquera une erreur de compilation due à un problème de résolution de portée. Il s'agit effectivement d'une situation de masquage, qui pose souci même si les méthodes héritées prennent des paramètres différents, comme dans la figure 1. La portée de méthodes étant différente, il ne s'agit pas d'une situation de surcharge.

```
class Ovipare {  
    // ...  
    void afficher() const;  
};  
  
class Vivipare {  
    // ...  
    void afficher(string const& entete) const;  
};  
  
class Ovovivipare : public Ovipare,  
                    public Vivipare  
{ //...  
};  
  
int main()  
{  
    Ovovivipare o(...);  
    o.afficher("Un orvet : ");  
  
    return 0;  
}
```

FIGURE 1

2:30

7:06

Problème de masquage dans l'héritage multiple: le compilateur refusera ce code: quelle méthode `afficher` est appelée dans `main`?

Une première solution lève l'ambiguïté par l'opérateur de résolution de portée, qui indique au compilateur la classe dont on appelle la méthode: dans l'exemple précédent, on pourrait choisir d'écrire `o.Ovipare::afficher()`. Cette solution est déconseillée, puisqu'elle délègue à l'utilisateur externe le choix de comment afficher un `Ovovivipare`, responsabilité qui devrait incomber au concepteur de la classe.

Une meilleure solution consiste à ajouter dans la sous-classe une déclaration spéciale explicitant quelle méthode héritée sera invoquée lors d'un appel ambigu. La syntaxe est la suivante:

```
using SuperClasse::NomMethodeouAttributAmbigu;
```

Notons l'absence de parenthèses et de type de retour: cette déclaration ne donne que le nom du membre concerné.



Enfin, une solution optimale clarifie la situation en définissant proprement une méthode au sein de la sous-classe, pour laquelle il y a ambiguïté et établissant le traitement à réaliser lors de l'appel. Le code de la figure 2 explicite le corps de la méthode `afficher()` propre à la classe `Ovovivipare`, qui peut notamment faire appel aux méthodes héritées et masquées au travers de l'opérateur de résolution de portée.

```
class Ovovivipare: public Ovipare, public Vivipare {
    public:
        // ...
        void afficher() const {
            Ovipare::afficher();
            Vivipare::afficher(" mais aussi pour sa partie"
                               " vivipare : ");
        }
        // ...
};
```

FIGURE 2

6:00

7:06

Bonne solution pour éviter les invocations ambiguës, consistant à redéfinir la méthode concernée au sein de la sous-classe.



29. CLASSES VIRTUELLES

Dans certaines situations d'héritage multiple, une sous-classe hérite plusieurs fois, indirectement, d'une même classe parente; contexte qui peut se révéler problématique. En C++, on observe de telles relations par exemple au niveau de la classe `iostream`: celle-ci hérite des deux classes `ostream` et `istream`, super-classes qui elles-mêmes héritent toutes deux de la classe `ios`. Cette organisation est présentée par le schéma en figure 2 de la leçon 27.

Plus concrètement, on pourrait concevoir que les deux classes `Ovipare` et `Vivipare`, dont hérite `Ovovivipare` (classes présentées en leçon 27), pourraient elles-mêmes hériter d'une super-classe `Animal`. Dans ce contexte, illustré en figure 1, une instance `Ovovivipare` posséderait en double chacun des attributs d'`Animal`, un jeu hérité de chaque classe parente. Ainsi, lors de l'affichage de `Ovovivipare`, deux attributs `tete` peuvent être considérés: l'un hérité via `Ovipare` et l'autre via `Vivipare`.

```
class Animal {  
public:    Animal(string const& description) : tete(description) {}  
protected: string tete;  
};  
  
class Ovipare : public Animal { public: Ovipare() : Animal("à cornes") {} };  
class Vivipare : public Animal { public: Vivipare() : Animal("de poisson") {} };  
  
class Ovovivipare : public Ovipare, public Vivipare {  
public:  
    void affiche() const { cout << "j'ai une tête " << Ovipare::tete  
                      << " et une tête " << Vivipare::tete << " !" << endl;  
    }  
};  
// ...  
Ovovivipare x;  
x.affiche();
```

j'ai une tête à cornes et une tête de poisson !

FIGURE 1

3:30

15:02

Exemple de programme où l'emploi de classes virtuelles serait justifié.

Dans l'exemple précédent, l'héritage double des attributs de la super-classe `Animal` est indésirable («deux têtes»); en revanche, certaines conceptions peuvent vouloir faire hériter un jeu d'attributs plusieurs fois. Dans une hiérarchie descendant d'une classe `Vehicule` dotée d'un attribut `moteur`, avec comme sous-classes des véhicules `Electrique` et `Essence`, on pourrait implémenter une classe de véhicule `Hybride` avec un moteur électrique et un moteur à essence. Le fait que `Hybride` ait comme classes parentes à la fois `Electrique` et `Essence`, qui pourtant héritent d'une même super-classe, est conforme à la conception du programme.

LIEN D'HÉRITAGE VIRTUEL

Pour éviter la duplication des attributs d'une super-classe plusieurs fois incluse lors d'héritages multiples, il faut déclarer son lien d'héritage avec toutes ses sous-classes comme `virtuel`. Une telle super-super-classe est appelée `super-classe virtuelle`.

Il est important de souligner la différence entre classe abstraite (leçon 24) et classe virtuelle: une classe abstraite est une classe avec des méthodes virtuelles pures, une classe virtuelle a un lien d'héritage virtuel vers ses sous-classes.

Signaler un héritage virtuel se fait lors de la déclaration du lien d'héritage:

```
class SousClasse : public virtual SuperClasseVirtuelle
```



Notons que c'est la classe pouvant être héritée plusieurs fois qui est virtuelle et non pas directement les classes utilisées dans l'héritage multiple. Dans l'exemple de la figure 1, la classe `Animal` est une classe virtuelle ; on indiquera l'héritage virtuel dans la déclaration des classes `Ovipare` et `Vivipare` et non pas dans la sous-classe `Ovovivipare` qui introduit le problème. L'héritage virtuel établit donc une dépendance au niveau de l'implémentation entre différents niveaux d'héritage, raison pour laquelle certains langages de programmation refusent l'héritage multiple.

La figure 2 illustre la structure que permettent les liens virtuels : grâce à ces derniers, la classe `Ovovivipare` n'aura qu'un seul attribut `tete` hérité de la classe `Animal`.

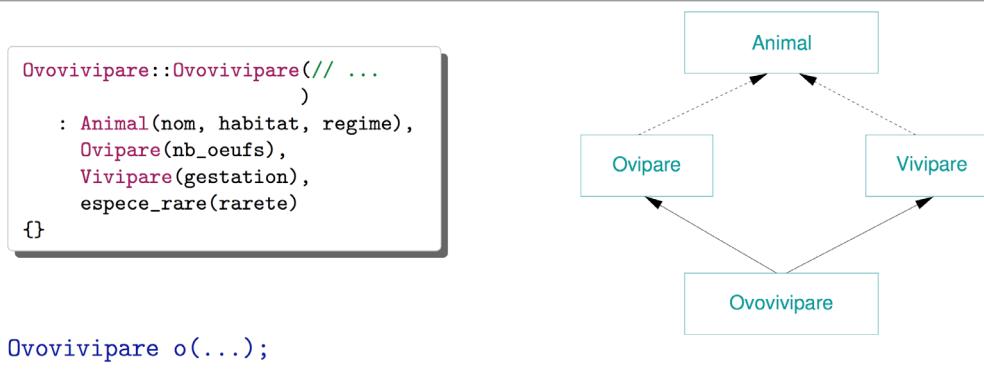


FIGURE 2

12:00

15:02

Nouvelle hiérarchie de classe et constructeur de la classe `Ovovivipare`. Les traits pointillés représentent un lien d'héritage virtuel.

HÉRITAGE VIRTUEL ET CONSTRUCTEURS

Pour rappel, dans un héritage simple, le constructeur d'une sous-classe ne fait appel qu'aux constructeurs de ses super-classes directes. En revanche, dans un contexte d'héritage virtuel, le constructeur de la super-super-classe virtuelle doit être explicitement appelé par les constructeurs des sous-classes instanciables, qui initialisent ainsi l'unique jeu d'attributs hérité de la classe virtuelle. Par la suite, les appels au constructeur de la super-classe virtuelle par le constructeur des classes intermédiaires sont ignorés.

Ainsi, le constructeur d'`Ovovivipare`, donné en figure 2, doit appeler directement le constructeur de la classe virtuelle `Animal`, pour initialiser son unique objet `Animal` hérité. Vient ensuite l'appel usuel aux super-classes directes (dans l'ordre de la déclaration d'héritage) que sont `Ovipare` et `Vivipare` : dans leur constructeur, l'appel à la classe virtuelle est alors ignoré. Enfin, le constructeur de la sous-classe `Ovovivipare` peut initialiser ses propres attributs.

Dans le cas où la super-classe virtuelle a un constructeur par défaut, il n'est pas nécessaire d'expliciter l'appel à ce dernier mais il sera toujours effectué dès la création de l'instance la plus dérivée.

L'ordre d'appel des constructeurs de copie vérifie aussi ces règles.

30. ÉTUDE DE CAS: PRÉSENTATION ET MODÉLISATION DU PROBLÈME

Terminons ce cours par l'étude d'un cas, allant de la modélisation du problème au code C++ complet.

PROBLÈME TRAITÉ

Le problème considéré ici vise à modéliser différentes montres, chacune constituée d'un mécanisme de base et différents accessoires (boîtiers, bracelets...). Ces entités sont toutes des produits dotés d'un prix, dont le calcul varie à partir d'une valeur de base fixe, et qui sont affichables en fonction de leur nature. Les mécanismes seront de trois types: analogiques, digitaux ou «doubles». Un mécanisme gère une heure. Un mécanisme double est un mécanisme avec une seule heure, qui peut être affichée de façon analogique et digitale: on choisira d'établir la virtualité des liens d'héritages entre la classe `Mecanisme` et ses sous-classes directes.

La conception d'une structure de classe à partir d'un problème passe par un repérage de mots-clés indicateurs. Le verbe **avoir** souligne une relation d'encapsulation, le verbe **être** qualifie une relation d'héritage, et enfin « **varier selon** » indique l'emploi du polymorphisme.

La figure 1 donne la hiérarchie de classe qui représente la situation. On spécifiera enfin les aspects d'encapsulation, à savoir qu'une `Montre` a comme attributs un `Mecanisme` et des instances d'`Accessoire`.

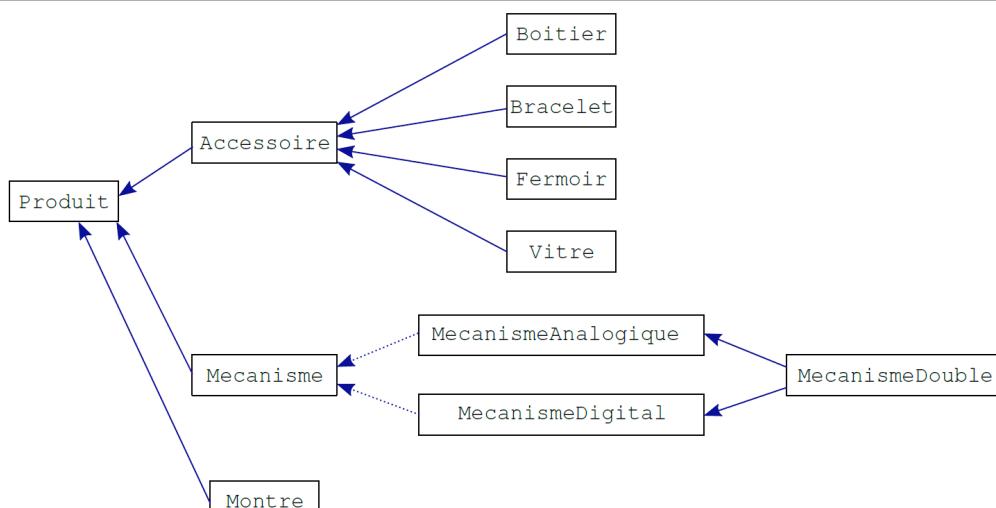


FIGURE 1

5:30

12:16

Modélisation des entités du problème choisi.



PREMIER CODE

Pour coder ces classes, on commencera par déclarer la super-classe `Produit`. Pour représenter le prix variable d'un `Produit`, on implémente une méthode qui effectue le calcul donné en fonction de la nature de l'instance courante : il s'agit donc d'une méthode virtuelle, donnée en figure 2. Un prix de base est fixé pour un produit quelconque, valeur stockée dans un attribut privé et renvoyée par défaut par la méthode `prix()`. On peut la qualifier de `const` puisqu'elle ne modifie pas l'instance ; elle renvoie un `double`. De plus, pour que chaque produit soit affichable, il est pertinent de surcharger l'opérateur d'affichage, concept étudié dans la leçon 31.

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

// =====
class Produit {
public:
    virtual double prix() const
    { return valeur; }
private:
    double valeur;
};

// On y revient plus tard
ostream& operator<<(ostream& sortie,
                      Produit const&);

// ...
```

FIGURE 2

10:30

12:16

Premier code de la classe `Produit`.

Ensuite, on peut déclarer les sous-classes `Montre`, `Mecanisme` et `Accessoire` qui héritent de `Produit`. Une montre encapsule un mécanisme et un ensemble d'accessoires sous la forme d'un tableau dynamique. Puisque ces deux dernières classes, de par leur généralité, seront traitées avec du polymorphisme, l'on choisira d'employer des pointeurs pour les attributs de la classe `Montre`, comme présenté en figure 3. Pour utiliser des pointeurs intelligents, on n'oubliera pas d'inclure la bibliothèque `memory`. Puisque la classe `Montre` a des attributs de type pointeur, il faut s'interroger sur la nature de la copie d'une telle instance : avant d'aborder le sujet en leçon 34, on se contentera de supprimer le constructeur de copie et l'opérateur d'affectation.

```
// ...
// =====
class Accessoire : public Produit {
};

// =====
class Mecanisme : public Produit {
};

// =====
class Montre : public Produit {
private:
    unique_ptr<Mecanisme> coeur;
    vector<unique_ptr<Accessoire>> accessoires;

    Montre(const Montre&) = delete;
    Montre& operator=(Montre) = delete;
};
```

FIGURE 3

11:00

12:16

Attributs de la classe `Montre`.



On déclare également les classes `MecanismeAnalogique`, `MecanismeDigital` et `MecanismeDouble` en tant que classes filles de `Mecanisme`; la spécificité de la classe `MecanismeDouble` sera redéfinie en leçon 33.



31. ÉTUDE DE CAS: AFFICHAGE POLYMORPHIQUE

Cette leçon conclut l'implémentation de la classe `Produit` en abordant la notion d'affichage polymorphique.

OPÉRATEUR D'AFFECTATION ET POLYMORPHISME

On cherche à adapter l'affichage des instances de `Produit` en fonction de leur type réel, dans le cadre d'une surcharge de l'opérateur d'affectation. Comme vu en leçon 12, cet opérateur ne peut être surchargé que par une fonction externe et ne peut donc pas être qualifié de méthode virtuelle pour permettre le polymorphisme. En revanche, son corps peut invoquer une méthode virtuelle publique `affiche()` de la super-classe `Produit`, à redéfinir dans chaque sous-classe. Par les arguments passés par référence, le comportement de l'opérateur devient ainsi polymorphique.

On décide que l'affichage d'un produit par défaut donne son prix: une idée serait d'afficher l'attribut `valeur` que détient chaque `Produit`. Cependant, selon les instances de produit, le prix ne correspond pas à la valeur de base: c'est le cas des montres, notamment, dont le prix dépasse la somme des coûts des composants. Il s'agit là d'une source d'erreurs fréquente: puisqu'**une méthode polymorphe s'adapte à la nature de l'instance traitée, les valeurs qu'elle utilise doivent le faire également**. Il est donc judicieux d'appeler la méthode virtuelle `prix()`, par un pointeur sur l'instance courante qui induit le traitement polymorphique.

```
class Produit {
public:
    virtual void afficher(ostream& sortie) const;
};

// -----
ostream& operator<<(ostream& sortie, Produit const& machin) {
    machin.afficher(sortie);
    return sortie;
}
```

FIGURE 1

1:40

8:58

Méthode d'affichage polymorphe.

Un bon réflexe à adopter, lors de l'introduction d'une méthode virtuelle, est de rendre aussi le destructeur virtuel. Ainsi, on s'assure que toute sa descendance est correctement détruite (voir leçon 22).



FINALISATION DE **Produit**

Une fois la valeur de base d'un produit donnée, on peut choisir de la laisser inchangée : pour cette raison, on qualifie cet attribut de `const`. Pour initialiser sa valeur, on définit deux constructeurs de la classe `Produit`: un premier qui prend en argument la valeur de l'instance et un second qui l'initialise par défaut à zéro. Cette implémentation est présentée en figure 2.

Enfin, il faut s'interroger sur le caractère instanciable de la classe `Produit`: en effet, cette dernière est une classe abstraite, racine du schéma d'héritage, dont on ne déclarera pas d'instance directe dans le programme. Pour rendre une classe abstraite, il faut lui donner une méthode virtuelle pure: en créer une pour cette seule raison est dénué de sens, on préférera donner cette caractéristique au destructeur, qui existe automatiquement dans chaque classe. Il est impératif de doter tout destructeur d'un corps et toute méthode virtuelle pure peut avoir un corps, à condition qu'il soit externalisé comme en figure 2.

```
class Produit {
public:
    Produit(double une_valeur = 0.0) : valeur(une_valeur) {}

    virtual ~Produit() = 0;

    virtual double prix() const { return valeur; }
    virtual void afficher(ostream& sortie) const { sortie << prix(); }

private:
    const double valeur;
};

Produit::~Produit() {}

ostream& operator<<(ostream& sortie, Produit const& machin) {
    machin.afficher(sortie);
    return sortie;
}
```

FIGURE 2

8:00

8:58

Code final de la classe `Produit`.



32. ÉTUDE DE CAS: SURCHARGE D'OPÉRATEUR ET PREMIÈRE VERSION

Après avoir déterminé la structure des classes et la question de l'affichage polymorphe, la tâche suivante de l'étude de cas consiste à surcharger l'opérateur `+=`, dans la classe `Montre`, pour permettre l'ajout d'accessoires à une montre.

SURCHARGE DE L'OPÉRATEUR `+=`

Le but de la surcharge suivante est donc d'ajouter des objets de type `Accessoire` au tableau dynamique de la classe `Montre`, déjà amorcée en leçon 30. Pour ajouter un `Bracelet` (sous-classe de `Accessoire`) à une `Montre`, `montre`, on utiliserait cet opérateur avec la syntaxe suivante :

```
montre += new Bracelet(...);
```

Puisqu'il modifie l'instance courante, cet opérateur doit être surchargé en interne, au sein de la classe `Montre`. L'exemple d'utilisation indique que son type de retour est `void` et qu'il prend comme paramètre un pointeur sur `Accessoire`.

Enfin, pour définir cet opérateur, il suffit de convertir le pointeur à la C sur `Accessoire` reçu en argument en `unique_ptr`; puis de l'ajouter à la collection d'accessoires de l'instance courante, stockée dans un tableau dynamique. Cette méthode est donnée en figure 1.

```
class Montre : public Produit {
public:
    void operator+=(Accessoire* p_accessoire) {
        accessoires.push_back(unique_ptr<Accessoire>(p_accessoire));
    }

private:
    unique_ptr<Mechanisme> coeur;
    vector<unique_ptr<Accessoire>> accessoires;

    Montre(const Montre&) = delete;
    Montre& operator=(Montre) = delete;
};
```

FIGURE 1

2:30

13:55

Surcharge de l'opérateur `+=` dans `Montre` pour l'ajout d'un `Accessoire`.

CLASSE ACCESSOIRE ET DESCENDANCE

La leçon 30 avait introduit la classe `Accessoire` comme sous-classe directe de `Produit`. On peut supposer que ses instances ont un nom donné par une chaîne de caractères, attribut qui reste constant une fois l'instance construite. Son type sera donc `const string`. On définit ensuite, comme en figure 2, un constructeur d'`Accessoire` qui prend en argument le nom et le prix du produit créé : la liste d'initialisation débute par un appel au constructeur de la super-classe `Produit` avec ce prix en paramètre, et se termine avec l'initialisation du nom. Puisque l'on s'attend à un traitement polymorphe de la classe `Accessoire`, le programmeur ne doit pas non plus oublier de rendre le destructeur virtuel.



Il convient ensuite de redéfinir la méthode d'affichage héritée, ce que l'on signale par le mot-clé `override`. Afficher un accessoire, dans la figure 2, donne son nom et son prix, auquel on accède par la méthode `afficher()` de la super-classe, démasquée par l'opérateur de résolution de portée. Enfin, on décide que le prix d'un accessoire est le même que celui d'un produit usuel: il n'est donc pas nécessaire de redéfinir la méthode `prix()` de la classe parente.

```
class Accessoire : public Produit {
public:
    Accessoire(string const& un_nom,
                double prix_de_base)
        : Produit(prix_de_base), nom(un_nom)
    {}

    virtual ~Accessoire() {}

    virtual void afficher(ostream& sortie)
        const override {
        sortie << nom << " coûtant ";
        Produit::afficher(sortie);
    }

private:
    const string nom;
};
```

FIGURE 2

6:30

13:55

Code final de la classe `Accessoire`.

Pour poursuivre, on peut définir des sous-classes d'`Accessoire`, comme les classes `Bracelet`, présentée en figure 3, ou `Fermoir`. Pour que le nom de chaque instance de ces dernières commence par son sous-type, on le concatène au nom donné en argument dans le constructeur. Ainsi, lors de la construction d'un `Bracelet`, on appelle le constructeur d'`Accessoire` avec le prix reçu et la chaîne de caractères «bracelet» + nom, où `nom` est le nom donné en argument. En prévision d'un comportement polymorphique, on signale la virtualité du destructeur des sous-classes `Bracelet` et `Fermoir`.

```
class Bracelet : public Accessoire {
public:
    Bracelet(string const& un_nom, double prix_de_base)
        : Accessoire("bracelet " + un_nom, prix_de_base)
    {}
    virtual ~Bracelet() {};
};
```

FIGURE 3

7:30

13:55

Exemple de sous-classes d'`Accessoire`.



CLASSE MONTRE

La classe `Montre` peut être opérationnelle dès lors qu'on lui définit un constructeur: la leçon 30 avait supprimé le constructeur de copie, ce qui désactive le constructeur par défaut par défaut. Avant la leçon 33, on commente la déclaration de l'attribut `unique_ptr<Mechanisme>` pour ne pas avoir à l'initialiser dans le constructeur. Puisque l'on a défini l'opérateur `+=` pour ajouter des instances d'`Accessoire` à une `Montre`, le constructeur par défaut par défaut, qui se contente de créer un tableau nul, est suffisant. On le réactive avec la ligne suivante:

```
Montre() = default;
```

On peut ensuite implémenter le calcul du prix d'une montre, comme la somme de son prix de base et du prix de ses `Accessoires`: on signale la redéfinition de la méthode virtuelle héritée par `override`. Ce calcul s'effectue par une itération sur la liste en attribut, qui accède à la méthode `prix()` de chaque instance de `Produit` constituant l'instance courante de `Montre`. Puisque le tableau est fait de `unique_ptr`, on n'oubliera pas de passer ses éléments par référence constante.

```
// ...
virtual double prix() const override {

    // Au départ, le prix est la valeur de base
    double prix_final(Produit::prix());

    for (auto const& p_acc : accessoires) {
        prix_final += p_acc->prix();
    }

    return prix_final;
}
// ...
private:
    // unique_ptr<Mechanisme> coeur;
    vector<unique_ptr<Accessoire>> accessoires;
    // ...
};
```

FIGURE 4

10:00

13:55

Méthode de calcul du prix d'une montre.

Similairement, pour afficher une `Montre`, il suffit d'afficher consécutivement l'ensemble des accessoires qui la composent, et enfin son prix total. La surcharge de l'opérateur d'affichage pour la classe `Accessoire` est en effet définie, s'agissant également d'instances de `Produit`. On redéfinit ainsi la méthode `afficher()` héritée en figure 5.



```
// ...
virtual void afficher(ostream& sortie)
    const override {

    sortie << "Une montre composée de :"
        << endl;

    for (auto const& p_acc : accessoires) {
        sortie << " * " << *p_acc << endl;
    }

    sortie << "==> Prix total : " << prix()
        << endl;
}
// ...
```

FIGURE 5

11:20

13:55

Méthode d'affichage d'une montre.

Tout le code créé peut être enfin testé avec un `main()` exécutable, comme celui fourni en figure 6. Ce programme crée une montre, lui ajoute des accessoires et l'affiche.

```
int main() {

    Montre m;

    m += new Bracelet("cuir", 54.0);
    m += new Fermoir("acier", 12.5);
    m += new Boitier("acier", 36.60);
    m += new Vitre("quartz", 44.80);

    cout << "Montre m :" << endl;
    cout << m << endl;

    return 0;
}
```

```
Montre m :
Une montre composée de :
    * bracelet cuir coutant 54
    * fermoir acier coutant 12.5
    * boitier acier coutant 36.6
    * vitre quartz coutant 44.8
==> Prix total : 147.9
```

FIGURE6

13:00

13:55

Programme principal de test.

Le code complet (à ce stade) peut être téléchargé [ici](#).



33. ÉTUDE DE CAS: MODÉLISATION DES MÉCANISMES

Cette leçon se penche sur la modélisation des mécanismes par l'héritage multiple, dans le cadre de l'étude de cas commencée en leçon 30.

RÉVISION DE LA HIÉRARCHIE

Un mécanisme double est à la fois analogique et digital, mais ne donne qu'une seule heure, modélisée par un attribut de la super-classe `Mecanisme`. L'héritage multiple permet cette relation, en faisant hériter `MecanismeDouble` à la fois de `MecanismeDigital` et de `MecanismeAnalogique`. Pour qu'un mécanisme double ne possède pas deux fois l'attribut `heure`, on introduit un lien d'héritage virtuel avec `Mecanisme` et ses sous-classes directes. Cette révision de la hiérarchie est visible en figure 1.

```
class MecanismeAnalogique : virtual public Mecanisme {
public:
    MecanismeAnalogique(double valeur_de_base, string une_heure, int une_date)
        : Mecanisme(valeur_de_base, une_heure), date(une_date)
    {}
private:
    int date;
};
// ...
class MecanismeDouble : public MecanismeAnalogique , public MecanismeDigital {
public:
    MecanismeDouble(double valeur_de_base, string une_heure, int une_date,
                    string heure_reveil)
        : Mecanisme(valeur_de_base, une_heure),
          MecanismeAnalogique(valeur_de_base, une_heure, une_date),
          MecanismeDigital(valeur_de_base, une_heure, heure_reveil)
    {}
};
```

FIGURE 1

4:45

14:46

Constructeurs des sous-classes de `Mecanisme`. Les liens d'héritages sont corrigés.

HÉRITAGE VIRTUEL ET CONSTRUCTEURS

La construction de `Mecanisme`, `MecanismeAnalogique` ou `MecanismeDigital` suit le même schéma en invoquant le constructeur de leur super-classe directe avant d'initialiser leur attribut propre, comme présenté en figure 1. Dans le constructeur de `Mecanisme`, on donne une valeur par défaut pour l'attribut `heure`.

Dans la sous-classe `MecanismeDouble`, les liens d'héritage virtuels ont une incidence sur la construction, qui doit en premier lieu appeler le constructeur de la classe virtuelle. Par la suite, les appels à ce constructeur par les super-classes directes `MecanismeAnalogique` et `MecanismeDigital` seront ignorés.



GESTION DES VALEURS PAR DÉFAUT

Afin de préserver la valeur par défaut donnée pour `heure` dans le constructeur de `Mecanisme`, il faut prévoir dans ses sous-classes un second constructeur surchargé qui ne prend pas de paramètre pour `heure` et appelle le constructeur de `Mecanisme` en ne spécifiant que le coût. Il ne serait pas possible de donner une valeur par défaut à un argument qui n'est pas en fin de liste de paramètres. Il serait également dangereux de placer cet argument en fin de liste avec une valeur par défaut: dupliquer la valeur par défaut à différents endroits du code peut être source de problèmes.

```
class MecanismeDouble : public MecanismeAnalogique , public MecanismeDigital {  
public:  
    MecanismeDouble(double valeur_de_base, string une_heure, int une_date,  
                    string heure_reveil)  
        : Mecanisme(valeur_de_base, une_heure)  
        , MecanismeAnalogique(valeur_de_base, une_heure, une_date)  
        , MecanismeDigital(valeur_de_base, une_heure, heure_reveil)  
    {}  
  
    MecanismeDouble(double valeur_de_base, int une_date, string heure_reveil)  
        : Mecanisme(valeur_de_base)  
        , MecanismeAnalogique(valeur_de_base, une_date)  
        , MecanismeDigital(valeur_de_base, heure_reveil)  
    {}  
};
```

FIGURE 2

7:05

14:46

Gestion des valeurs par défaut du constructeur de la super-classe `Mecanisme`.

AFFICHAGE DES MÉCANISMES

On décide que l'affichage d'un mécanisme s'effectue toujours selon le même schéma non modifiable: ce dernier consiste à afficher le type du mécanisme, suivi d'un cadran donnant l'heure et d'autres informations propres au mécanisme, pour enfin donner son prix. Ce schéma de base est imposé à tous les mécanismes par le biais de la méthode d'affichage héritée de `Produit`, redéfinie, comme le signale l'emploi du mot `override`, en figure 3. Pour que l'affichage d'un mécanisme soit fixé, on marque la méthode comme «`final`».



Pour permettre un affichage qui cependant s'adapte à la nature réelle de l'instance traitée, cette méthode doit donc faire appel à des méthodes virtuelles, `afficher_type()` et `afficher_cadran()`. On introduit notamment une version par défaut de cette dernière dans la super-classe, qui donne l'heure d'un mécanisme. Son droit d'accès est protégé pour la rendre utilisable dans les sous-classes. En revanche, pour imposer une redéfinition de `afficher_type()`, on la déclare comme virtuelle pure. N'étant pas invoquée dans les sous-classes, cette méthode reste privée.

```
class Mecanisme : public Produit {
public:
    //...
    // Tous les mécanismes DOIVENT s'afficher comme ceci
    virtual void afficher(ostream& sortie) const override final {
        sortie << "mecanisme" ; afficher_type(sortie);
        sortie << "(affichage : " ; afficher_cadran(sortie);
        sortie << "), prix : " ; Produit::afficher(sortie);
    }

protected: // On veut offrir la version par défaut aux sous-classes
// Par défaut, on affiche juste l'heure.
    virtual void afficher_cadran(ostream& sortie) const {
        sortie << heure;
    }

private:
    virtual void afficher_type(ostream& sortie) const = 0;
};
```

FIGURE 3

10:10

14:46

Méthodes d'affichage d'un Mecanisme.

Les sous-classes doivent définir ces méthodes pour un affichage correct et pour devenir instanciables (avec une méthode virtuelle pure, `Mecanisme` devient une classe abstraite). Pour `MecanismeAnalogique`, l'affichage du type donnerait «analogique» et l'affichage du cadran renverrait l'heure, grâce à un appel à la méthode héritée par l'opérateur de résolution de portée, et la date. La classe `MecanismeDouble` peut exploiter les redéfinitions de la méthode `afficher_cadran()` de ses deux classes parentes dans la sienne.

```
class MecanismeAnalogique: virtual public Mecanisme {
    // ...
protected:
    virtual void afficher_type(ostream& sortie) const override {
        sortie << "analogique";
    }

    virtual void afficher_cadran(ostream& sortie) const override {
        // On affiche l'heure (façon de base)...
        Mecanisme::afficher_cadran(sortie);
        // ...et en plus la date.
        sortie << ", date " << date;
    }
    // ...
};
```

FIGURE 4

12:10

14:46

Redéfinition des méthodes pour l'affichage d'une sous-classe de `Mecanisme`.



Enfin, on peut tester le code développé dans un programme principal donné en figure 5, qui construit des instances de sous-classes de `Mecanisme`, notamment sans spécifier l'heure, et les affiche. Il devient aussi pertinent d'adapter le constructeur et l'affichage de `Montre` à l'attribut `coeur` de type pointeur vers `Mecanisme`.

```
// test de l'affichage des mécanismes
MecanismeAnalogique v1(312.00, 20141212);
MecanismeDigital    v2( 32.00, "11:45", "7:00");
MecanismeDouble     v3(543.00, "8:20", 20140328, "6:30");
cout << v1 << endl << v2 << endl << v3 << endl;

// Test des montres
Montre m(new MecanismeDouble(468.00, "9:15", 20140401, "7:00"));
m += new Bracelet("cuir", 54.0);
m += new Fermoir("acier", 12.5);
m += new Boitier("acier", 36.60);
m += new Vitre("quartz", 44.80);
cout << endl << "Montre m :" << endl;
cout << m << endl;
```

FIGURE 5

13:30 14:46

Exemple de programme principal de test.

Le code complet de cette partie est téléchargeable [ici](#).



34. ÉTUDE DE CAS: COPIE PROFONDE

L'étude de cas se conclut par un travail sur la copie de montres, copie profonde et polymorphe, et leur affectation. Les attributs de `Montre` sont des pointeurs : il faut donc s'interroger sur la nature de la copie à réaliser. Par défaut, le constructeur de copie réalise une copie de surface, qui copie leur valeur, c'est-à-dire l'adresse de l'élément pointé, dans une autre instance. Dans ce contexte, deux montres copiées partageraient les mêmes accessoires et le même mécanisme ; modifier le bracelet d'une changerait aussi celui de l'autre. De plus, les pointeurs stockés sont de type `unique_ptr`, ce qui en empêche toute copie. Il faut plutôt réaliser une copie profonde, qui consiste à copier les objets pointés et non leur adresse.

Dans le constructeur de copie de `Montre`, on appelle tout d'abord explicitement un constructeur de copie de sa super-classe `Produit` : sans cette ligne visible en figure 1, la partie `Produit` de la nouvelle instance serait construite avec des valeurs par défaut.

On cherche ensuite à copier l'objet pointé par l'attribut `coeur` dans une nouvelle zone mémoire allouée, dont l'adresse sera la valeur du `coeur` de la nouvelle instance. Dans la liste d'initialisation, on pourrait être tenté d'écrire :

```
coeur(unique_ptr<Mecanisme> (new Mecanisme(*autre.coeur)))
```

où `autre` est une référence sur la montre à copier. Cette syntaxe n'est pas la solution voulue : en réalisant une copie du mécanisme pointé par `autre`, on perd sa spécificité analogique, digitale ou double. (Il est également pertinent de préciser que `Mecanisme` est une classe abstraite.)

```
Montre(const Montre& autre)
// Ne pas oublier d'appeler le constructeur DE COPIE de la super-classe
: Produit(autre),
  coeur(autre.coeur->copie())
{
    for (auto const& p_acc : autre.accessoires) {
        accessoires.push_back(p_acc->copie());
    }
}
```

FIGURE 1

7:40

14:36

Constructeur de copie de `Montre`, qui réalise une copie profonde et polymorphe.

COPIE POLYMORPHIQUE

Le constructeur de copie de `Montre` doit effectuer une copie qui conserve la nature des instances, une **copie polymorphe**. Pour effectuer cette tâche, on fait appel à une méthode virtuelle `copie()` capable de copier un `Mecanisme` ou un `Accessoire` tout en préservant sa spécificité tirée d'une sous-classe. On l'utilisera comme en figure 2.

Cette méthode est définie au niveau de la super-classe (copiée de manière polymorphe), `Accessoire` par exemple. Puisque l'on ne peut pas la définir à un niveau général, on la qualifie de virtuelle pure. Sa valeur de retour est du type des valeurs stockées dans la collection `accessoires`, à savoir `unique_ptr<Accessoire>`.



Dans les sous-classes, on redéfinit cette méthode en lui faisant retourner un `unique_ptr<Accessoire>` qui pointe sur une copie de l'instance courante `*this`. Si celle-ci est un `Bracelet`, la zone mémoire allouée contient un nouveau `Bracelet` construit par copie de l'instance courante.

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphe d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};

//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphe de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```

FIGURE 2

8:10

14:36

Définition de la méthode permettant la copie polymorphe.

OPÉRATEUR D'AFFECTATION

Pour pouvoir utiliser l'opérateur `=` entre différentes instances de `Montre`, il est possible de le surcharger. On utilisera pour cela le schéma présenté en leçon 14 avec la fonction `swap()` fournie dans la bibliothèque `utility`.

L'opérateur d'affectation, donné en figure 3, passe son argument `Montre` par valeur: le second opérande sera donc copié, par la copie profonde du constructeur de copie redéfini, dans une variable locale `source`. On peut ensuite échanger la valeur de ses attributs avec ceux de l'instance courante. On rappelle que retourner une référence sur l'instance courante permet d'enchaîner plusieurs affectations (leçon 14).

```
class Montre : public Produit {
public:
    // ...
    Montre& operator=(Montre source) // Notez le passage par VALEUR
    {
        swap(coeur      , source.coeur      );
        swap(accessoires, source.accessoires);
        return *this;
    }
};
```

FIGURE 3

11:45

14:36

Définition de l'opérateur d'affectation pour la classe `Montre`.



Enfin, on vérifie ces nouvelles implémentations en complétant le programme principal donné en leçon 33 avec une construction de `Montre` par copie, une affectation. On peut aussi définir une méthode pour modifier l'heure d'une `Montre` et s'assurer que la copie profonde est bien réalisée: dans l'exemple en figure 4, l'instance `m3` ne devrait pas être affectée par la manipulation de `m2`.

```
// ... (le reste du main() comme avant)
// Nous faisons une copie de la montre m
Montre m2(m);
cout << "Montre m2 :" << endl;
cout << m2 << endl;

// Et testons l'opérateur d'affectation
Montre m3(new MecanismeAnalogique(87.00, 20140415));
cout << "Montre m3 (1) :" << endl;
cout << m3 << endl;

m3 = m2; m2.mettre_a_1_heure("10:10");
cout << "Montre m3 (2) :" << endl;
cout << m3 << endl;
// ...
```

FIGURE 4

13:30

14:36

Exemple de programme principal de test.

Le code complet de l'étude de cas est fourni [ici](#). Cette leçon conclut l'étude de cas et termine ce cours d'introduction à la programmation orientée objet.



IMPRESSIONS

© EPFL Press, 2016.
Tous droits réservés.

Graphisme:
Emphase Sàrl, Lausanne

Résumé: Alizée Pace

Développés par EPFL Press, les BOOCs (Book and Open Online Courses) sont le support compagnon des MOOCs proposés par l'École polytechnique fédérale de Lausanne. Valeur ajoutée aux MOOCs, ils rassemblent l'essentiel à retenir pour l'obtention du certificat et constituent un atout pédagogique. Learn faster, learn better. Bonne révision!

ISBN 978-2-88914-399-3



OUVRAGE DE RÉFÉRENCE DU MÊME AUTEUR

Cet ouvrage a pour objectif d'offrir la pratique nécessaire à tout apprentissage de la programmation : un cadre permettant au débutant de développer ses connaissances sur des cas concrets. Il se veut un complément pédagogique à un support de cours. Avec près d'une centaine d'exercices gradués de programmation en C++, accompagnés d'une solution complète et souvent détaillée, l'ouvrage est structuré en deux parties : la première présente la programmation procédurale, tandis que la seconde aborde le paradigme de la programmation objet. Chacune de ces parties se termine par un chapitre regroupant des exercices généraux. Chaque chapitre contient un exercice pas à pas et une série d'exercices classés par niveaux de difficulté.