Classes, instances, objets [oo]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 20 mai 2018

Table des matières

C++ - Classes, instances, objets

Mots-Clés Conception Objet, Classe, Instance, Encapsulation, Interface ■
Requis Axiomatique impérative ■
Difficulté • ○ ○ (2 h) ■

Introduction

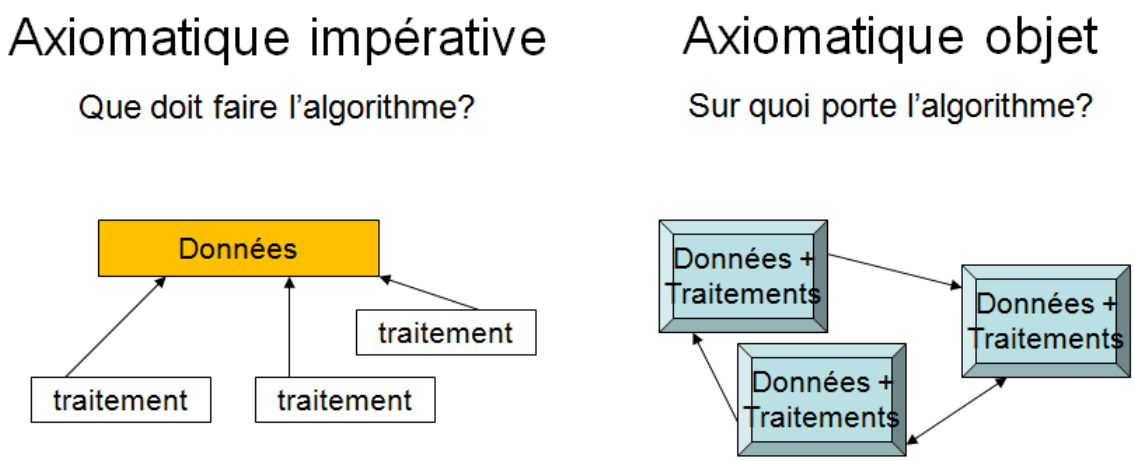
L'un des objectifs de la conception objet est de fournir des outils pour **organiser** de façon plus efficace les données et les traitements en les regroupant dans des entités appelées **objets**. Les notions mises en oeuvre par les objets sont : abstraction, encapsulation, héritage et polymorphisme.

Ce module introduit les bases de l'axiomatique objet (**abstraction**, **encapsulation**). Il présente l'approche impérative versus objet, expose les notions d'interface et d'encapsulation puis définit l'axiomatique objet. On termine par un résumé de la terminologie objet abordée dans ce module.

■

1 Approche impérative v.s. objet

1.1 Approche impérative – Approche objet



Données et Traitements

Dans l'axiomatique **impérative**, ou Programmation Procédurale PP, les données sont séparées des traitements (fonctions et procédures) qui les utilisent. Tandis que dans l'axiomatique **orientée-objet** (OO), les données et les traitements concernant ces données sont regroupés dans des entités appelées **objets**.

Organisation des programmes

Réciproquement, l'organisation des algorithmes/programmes impératifs conduit à une séparation des données et traitements. Tandis qu'un algorithme/programme objet est un réseau d'objets qui **communiquent** par l'envoi de **messages** pour réaliser un traitement.

1.2 Les types et traitements (rappel)

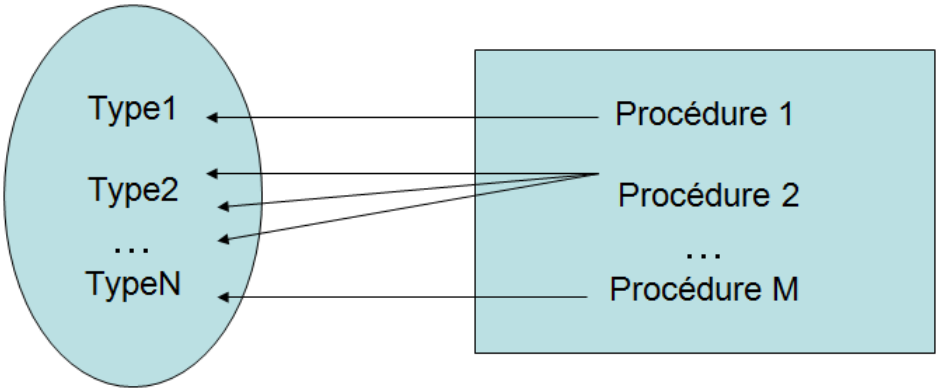
Le type d'une variable

Il détermine un ensemble de valeurs possibles pour la variable et les opérations qu'il est possible de faire sur cette variable. Il peut être un type de base (`Entier`, `Réel`, `Caractère`, `Booléen`...) ou défini par l'utilisateur.

Les traitements

Pour effectuer des traitements sur les variables, on définit des modules (fonctions et procédures). Ces derniers sont définis en dehors des types de données qu'ils manipulent. Par conséquent, les variables manipulées doivent être passées en paramètre des modules qui les manipulent.

Schéma synthétique



1.3 Définitions OO

Objet, état, comportement

Un **objet** est la modélisation d'une entité du monde réel, laquelle est concrète (une voiture, un stylo, un client...) ou abstraite (une entreprise, le temps, une relation bancaire...). Il est caractérisé par :

- Une **identité** exprimée par son nom.
- Ce qu'il est : son **état** (c.-à-d. les données sur lui-même).
- Ce qu'il sait faire : son **comportement**.

Classe, type, instance, objet

Un objet fait partie d'une catégorie d'objets appelée **classe**. La classe est le **type** de l'objet. Voici différentes manières de dire la même chose :

- Un **objet** est une **instance** de classe.
- Un objet est une variable dont le type est la classe.
- Un objet est un exemplaire d'une classe.
- La *classe* est un *moule* à partir duquel on peut faire des *gâteaux* : les *objets*.

Jargon de l'axiomatique objet

CLASSE = TYPE
INSTANCE = OBJET

Attributs, Méthodes

Une classe *encapsule* (regroupe) dans la même entité informatique les données et les traitements de ces données. Elle est composée :

- D'**attributs** (*data attributes*) : *champs* de données ou *propriétés* représentant l'*état* des objets.
- De **méthodes** (*methods*) : fonctions ou procédures applicables aux objets qui conditionnent son *comportement*.

Relation fondamentale

OBJET = ATTRIBUTS + METHODES

1.4 Dualités entre les approches

Les **classes** (resp. les **objets**) sont à l'axiomatique objet ce que les **types** (resp. les **variables**) sont à l'axiomatique impérative.

Au niveau des déclarations

- Celle d'une classe se fait de façon similaire à la déclaration d'un type structuré.
- Et celle d'une instance à la déclaration d'une variable classique.

Axiomatique impérative		Axiomatique objet	
type	TypeT … FinType	classe	classe K … FinClasse
variable	Variable v : T	objet	Variable o : K

Différences au niveau du regroupement

- En OO : Les données et les traitements sont **encapsulés** dans une classe.
- En PP : Les données sont regroupées dans une structure mais les traitements sont définis à part.

Différences au niveau des paramètres

- En OO : Les méthodes ne prennent pas en paramètre l'objet qu'elles manipulent, puisqu'une méthode est toujours appelée (*invoquée*) par un objet et la méthode appelée *sait* toujours quel objet l'a appelée.
- En PP : Les sous-programmes prennent en paramètre le nom de la structure à manipuler.

Utilisation des traitements

- En OO : C'est l'objet `o` qui fait appel au traitement `m` à effectuer : `o.m(...)`.
- En PP : Il faut appeler le traitement `m` avec l'objet `o` en paramètre : `m(o,...)`.

Résumé

En OO, tous les traitements (les actions) sont des méthodes et donc appelées (réalisées) par des objets.

2 Classes et instances

2.1 Déclaration Classe, Instances

Déclaration d'une classe

```
class K; // déclaration
class K // définition
{
  ...
}; // <- point virgule
```

Explication

Annonce (déclare ou défini) la classe nommée `K`.

Déclaration d'une instance

```
K o;
```

Explication

Déclare une instance nommée `o` de type `K`.

2.2 Déclaration d'un attribut

Déclaration d'un attribut

```
class K
{
  T attr;
};
```

Explication

Déclare un attribut nommé `attr` de type `T` pour la classe `K`.

Convention

Un attribut `attr` sera préfixé :

- Par « `m_` » signifiant *mon* (*my* en anglais – Convention Microsoft).
- Ou par « `d_` » pour *donnée* (*data* en anglais).

D'où l'écriture `m_attr`.

2.3 Écriture d'une méthode

Écriture d'une méthode

```
class K
{
  T meth(...) [const]
  {
    ...
  }
};
```

Explication

Définit la méthode `meth` pour la classe `K`. Le qualificatif `const` (éventuel) précise que la méthode ne modifie pas l'objet.

2.4 Accès aux membres

Accès aux membres

```
o . attribut // accès à l'attribut
o . methode(arg1, ...) // appel de la méthode
```

Explication

Accède au membre d'une instance de nom `o`.

Portée des attributs

Un attribut est une variable globale à la classe. D'où :

- Il est accessible dans toutes les méthodes de la classe.
- Il est inutile de les passer comme paramètres des méthodes (de ladite classe).

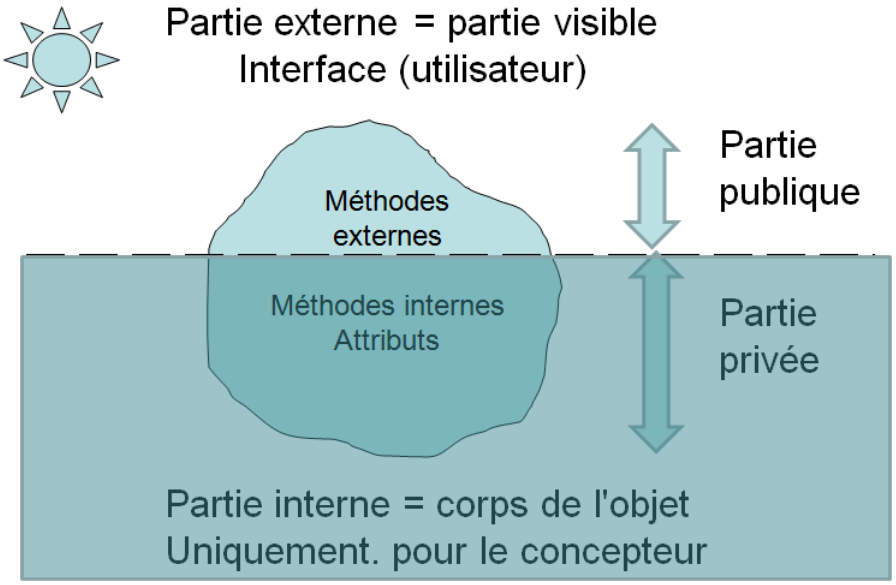
3 Encapsulation et interface

3.1 Le concept d'encapsulation

Principe de l'iceberg

Pour dissimuler son fonctionnement interne, l'axiomatique objet introduit deux niveaux de perception :

- Appelé **interface**, le niveau *externe* est la partie **visible** de l'objet.
  Ce sont les *moyens* (les méthodes) par lesquels un utilisateur peut interagir avec l'objet.
- Appelé **implémentation**, le niveau *interne* est le **corps** de l'objet.
  C'est la *définition* des méthodes et des attributs.

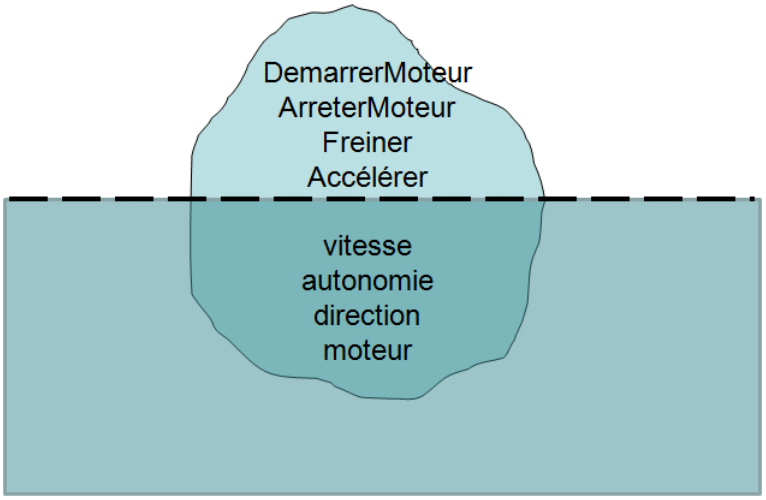


Encapsulation

Par ce terme, il faut entendre que les données d'un objet sont protégées, tout comme le médicament est protégé par la fine pellicule de sa capsule. Grâce à cette protection, il ne peut y avoir de transformation involontaire des données de l'objet.

Exemple : Interface d'une voiture

Elle concerne tout ce qu'il faut savoir pour la conduire (mais pas la réparer, ni savoir comment cela fonctionne). L'interface ne change pas même si l'on change de moteur... et/ou de voiture (dans une certaine mesure) : c'est donc l'**abstraction** de la notion de voiture (en tant qu'« objet à conduire »).



Classe : Nouvelle définition

Une classe est un ensemble :

- de méthodes,
- d'attributs et
- de qualificateurs d'accès (public ou privé).

Classe = super-structure

Une `class`(e) est donc une `struct`(ure)

- Qui contient aussi des procédures/fonctions (méthodes).
- Dont les champs (internes) peuvent être cachés.
- Et dont d'autres constituent l'interface.

3.2 Droits d'accès

C++ : Droits d'accès

```
class K {
  public: // partie Interface
  ...
  private: // partie Encapsulée
  ...
} ;
```

Par défaut, lorsque les membres sont déclarées sans type de protection, leur protection est `private`.

Explication
Précise les notions d'encapsulation et d'interface.

Erreur de compilation
Il y a **erreur** si le client fait référence à un membre privé.

Règles OO
Dans la plupart des cas :

- Privé (`private`) :
  - Tous les attributs (variables d'instance).
  - Des méthodes d'instance facilitant l'implémentation.
- Publique (`public`) :
  - Quelques méthodes d'instance bien choisies pour manipuler l'objet.

3.3 Méthodes accesseurs-mutateurs

Accès aux attributs

L'encapsulation permet d'augmenter la fiabilité, la sécurité et l'intégrité des programmes puisqu'elle empêche la modification *accidentelle* des données d'un objet. Mais l'accès aux attributs ne pouvant se faire directement,
Comment les manipuler ?

Réponse

En définissant des méthodes (publiques) qui permettent de les gérer. Il en existe de deux types :

Accesseur

Méthode d'accès à l'information **en lecture**.
Il commence souvent par `get` (*getter* en anglais, lire).

Mutateur

Méthode d'accès à l'information **en écriture**.
Il commence souvent par `set` (*setter* en anglais, fixer).

Propriété

(*Property* en anglais) Attribut de classe géré par un accesseur et un mutateur.

3.4 Modifieurs et sélecteurs

Principe de la propriété (property) (Juillet 2017)

Pratique liée au principe de l'encapsulatiion des données. L'accès à un attribut se fera via une méthode dédiée dont le nom est généralement préfixé avec « get » et la modification avec une méthode dédiée dont le nom est généralement préfixé avec « set ». Une autre solution consiste à retourner la référence de l'attribut ce qui permet de lire/modifier l'attribut.

C++ : Modifieurs et sélecteurs

```
class K
{
  ... getXXX(...) const ; // accesseur
  ... setXXX(...) ; // mutateur
} ;
```

Explication

Déclare la propriété `XXX` de l'objet de type `K`. La méthode `getXXX` **n'a pas le droit de modifier** (mot-clé `const`) l'état interne de l'objet.

3.5 Exemples : Propriétés encapsulées (property) (Juillet 2017)

Remarque
Le principe de la *property* est intégrée nativement dans le langage C# avec les mots-clés `get`, `set` et `value` ainsi que dans les langages VB.NET et Python (par exemple).

C++ : Propriétés Get/Set/Valeur

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

class test
{
private:
   int m_val;
public:
   test(int v): m_val(v){}
   inline int getVal() const { return m_val; }
   inline void setVal(int val) { m_val = val; }
   inline int& val(){ return m_val; };
};

int main()
{
   test t(10);
   // obtient la valeur
   cout<<"getVal()="<<t.getVal()<<endl;
   // fixe une nouvelle valeur
   t.setVal(50);
   cout<<"Apres setVal(50), getVal()="<<t.getVal()<<endl;
   // obtient et modifie la valeur via une reference
   int& ref = t.val();
   ref += 10; // modifiable
   cout<<"Apres ref+=10, val()="<<t.val()<<endl; // 60
   t.val() += 40;
   cout<<"Apres val()+=40, val()="<<t.val()<<endl; // 100
}
```

Exemple : C++ : Contrôle lors de la modification
L'emploi des méthodes d'accès permet de contrôler que la valeur est conforme.

```
#include <iostream>
using namespace std;
class Produit
{
private:
   double m_prix;
   int m_stock;
   int m_min, m_max;
public:
   Produit(int min, int max, int stock);
   double getPrix() const;
   void setPrix(double prix);
};
```

```
Produit::Produit(int min, int max, int stock)
: m_min(min), m_max(max), m_stock(stock)
{
   m_prix = (max - min) / 2 + min;
}

double Produit::getPrix() const
{
   return (m_stock < 10 ? m_prix * 2 : m_prix);
}

void Produit::setPrix(double prix)
{
   if (prix >= m_min && prix <= m_max){
      m_prix = prix;
   }
   else{
      cout<<"Pas de modif; doit etre entre "<<m_min<<" et "<<m_max<<endl;
   }
}

int main()
{
   Produit p1(0, 100, 20);
   cout<<p1.getPrix()<<endl; // 50
   p1.setPrix(200); // msg erreur
   p1.setPrix(75);
   cout<<p1.getPrix()<<endl; // 75

   Produit p2(500, 1000, 5);
   cout<<p2.getPrix()<<endl; // 1500
   p2.setPrix(200); // msg erreur
   p2.setPrix(950);
   cout<<p2.getPrix()<<endl; // 1900
}
```

4 Règles OO et Structeurs

4.1 Règles OO

Un objet doit être « intègre » durant toute sa durée de vie. Par conséquent :

Règle 1
La **structure interne** de l'objet est **inaccessible** directement : il faut passer par les méthodes de l'interface.

Règle 2
Les attributs **ne doivent pas être accessibles directement** depuis l'extérieur mais uniquement par des méthodes.

Règle 3
Il existe un mécanisme qui permet d'éviter les problèmes dus aux valeurs indéterminées, en rendant l'initialisation automatique à la déclaration d'un objet. Ce mécanisme est fondé sur les **constructeurs**.

Règle 4
De même, il existe une méthode particulière qui est appelée lorsqu'un objet doit être détruit. Cette méthode s'appelle **le destructeur**.

4.2 Constructeur

Constructeur

Méthode particulière qui est appelée automatiquement lors de la **création** d'une instance d'objet. Il permet d'effectuer certaines actions (tests des valeurs d'initialisation, allocation de ressources, ouverture de fichiers...) avant même qu'un objet ne soit utilisé.

C++ : Constructeur

```
K::K(...)
```

Explication

Un constructeur est une méthode qui :

- Porte le même nom que la classe dans laquelle il est placé.
- C'est la première méthode à être exécutée.
- Il peut prendre des paramètres.
- Il n'est pas typé.
- Sa définition n'est obligatoire que si elle est nécessaire au bon fonctionnement de la classe.

Multiplicité des constructeurs

Une classe peut définir **plusieurs** constructeurs à condition qu'ils diffèrent dans leurs listes de paramètres respectives selon le principe de la surcharge de fonction/procédure.

Constructeur par défaut

Chaque classe dispose d'un **constructeur par défaut** (il ne possède pas de paramètre) qui initialise à zéro les données (`0` pour les entiers, `0.0` pour les réels, `""` pour les chaînes, etc.). Lorsqu'un constructeur est défini, le constructeur proposé par défaut par le langage n'existe plus.

Il est recommandé de toujours prévoir le constructeur par défaut (il est nécessaire lors de la déclaration de tableaux d'objets par exemple).

C++ : Liste d'initialisation d'un constructeur

```
K::K(...)
: liste_d_initialisation_des_champs
{
  Corps_du_constructeur
}
```

4.3 Destructeur

Destructeur
Méthode particulière qui est appelée automatiquement à chaque **destruction** d'objet. Il permet d'effectuer certaines actions de nettoyage (comme la récupération de mémoire dynamique, la fermeture de fichiers...) avant de perdre tout lien vers l'objet.

C++ : Destructeur

```
K::~K()
```

Explication
Le destructeur :

- A le même nom que la classe précédé de title (~).
- Ne prend pas de paramètre.
- Ne retourne pas de résultat.

Unicité du destructeur
Une classe ne peut disposer que d'un **seul** destructeur.

5 Compléments

6 Terminologie objet

Classe
Abstraction qui permet de modéliser et/ou décrire un ensemble d'objets de façon générale. La classe explicite le comportement et le rôle des instances.

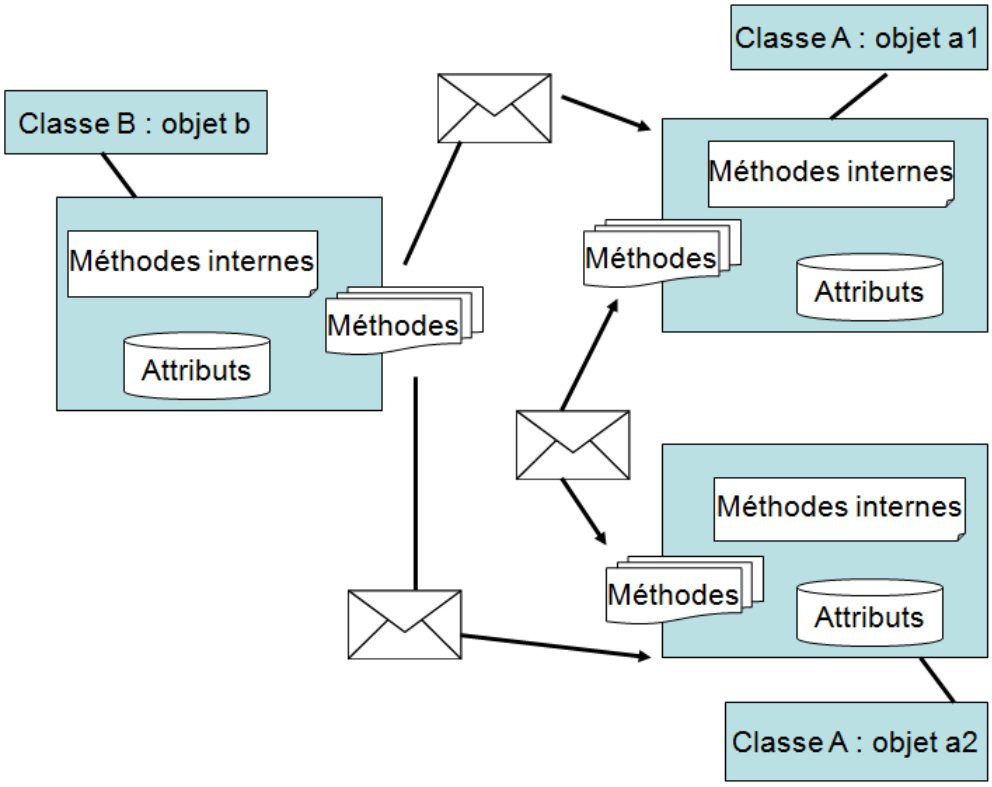
Instances (d'une classe)
Ce sont les *objets* qui vérifient les contraintes définies par la classe. Les instances sont toutes différentes : leur état est différent mais elles ont toutes le même comportement. On décrit parfois la classe comme une usine qui construit des objets.

État d'un objet
Ensemble des caractéristiques instantanées d'un objet, c.-à-d. l'ensemble des valeurs des attributs. Toutes les instances d'une classe ont les mêmes attributs avec des valeurs variables, uniques pour chaque instance.

Encapsulation
Permet à l'objet de cacher son fonctionnement interne. Ainsi les changements internes n'ont pas d'influence sur le code extérieur à l'objet. Les objets encapsulent les données (*données membres*, champs, attributs...) et les traitements (*méthodes membres*) (partie privée – `private`).

Interface
Permet à l'objet de communiquer avec d'autres objets par l'intermédiaire d'envoi de *messages* à l'aide de *méthodes* (partie publique – `public`). *Envoyer un message* à un objet, c'est lui demander d'exécuter une de ses *méthodes*.

Schéma synthétique



7 C++ : Spécificités

7.1 Écriture externe d'une méthode

Écriture externe d'une méthode

```
class K
{
  T meth(...) [const]; //<- point-virgule
};

T K::meth(...) [const] // Implémentation
{
  ...
}
```

Explication

L'opérateur de résolution de portée `::` relie la définition d'une méthode `meth` à la classe `K` pour laquelle elle est déclarée.

Intérêts de l'opérateur de portée

Il permet une meilleure lisibilité du code et la modularisation via l'écriture des définitions des méthodes à l'extérieur de la définition de la classe. Par conséquent :

- Dans la classe : uniquement les prototypes des méthodes.
- Implémentations : à l'**extérieur** de la classe.

7.2 Organisation du code source

A l'exception de classes fortement reliées, il est conseillé de placer une seule classe par fichier source. En fait, une classe a même besoin de 2 fichiers sources : un fichier *header* (`.h`, `.H`, `.hxx`, `.hh` ou `.hpp` selon les environnements) où l'on place la déclaration de la classe et un fichier de définition des méthodes et des variables de classe (`.C`, `.cpp`, `.cc` ou `.cxx` selon les environnements – l'utilisation de `.C` est fortement déconseillé car certains environnements ne font pas la distinction entre les majuscules et les minuscules et `.C` risquerait d'être interprété comme du C et non du C++).

Structure du fichier de déclaration d'une classe

Afin d'éviter les inclusions redondantes, on place des balises de compilation avec des `#ifdef` comme dans l'exemple suivant :

```
#ifndef NOM_DE_LA_CLASSE_HEADER
#define NOM_DE_LA_CLASSE_HEADER
// Placer ici les inclusions et les déclarations externes nécessaires
// Placer ici la déclaration de la classe
class NomDeLaClasse
{
}; //<- Ne pas oublier ce ";"
// Sauver sous le nom: nom_de_la_classe.hxx
#endif
```

Structure du fichier de définition d'une classe

Le fichier de définition sera :

```
#ifndef NOM_DE_LA_CLASSE_METHODS
#define NOM_DE_LA_CLASSE_METHODS
#include "nom_de_la_classe.hxx"
// autres inclusions nécessaires
// Définitions des variables de classe
// Définitions des méthodes
#endif
```

7.3 Les méthodes inline

Les méthodes inline

Il existe deux manières de spécifier le code des méthodes : l'écriture directe ou l'écriture externe. La deuxième, qui consiste à séparer la déclaration de la méthode de son implémentation, présente néanmoins un inconvénient : « Utiliser un appel de méthode pour récupérer la valeur d'un attribut c'est une perte de temps lamentable. »

Les méthodes `inline` ont été inventées pour cela ! La principale caractéristique des méthodes `inline` est leur faculté à développeur leur code en lieu et place d'un appel à la manière d'une macro. Vous saisissez tout de suite l'avantage : vous gagnez le temps nécessaire à l'appel d'une fonction, ainsi l'accès à un attribut ne coûte plus rien !

Il y a néanmoins un inconvénient, comme tout appel est remplacé par le développement du code de la méthode, il en résulte un accroissement de la taille du code cible. Aussi, ces méthodes doivent être limitées à quelques instructions sous peine d'accroître quasi indéfininiment la taille de l'exécutable. En outre, certains compilateurs refusent de mettre en ligne les méthodes qui contiennent des boucles.

Comment une méthode devient elle inline ?

Il existe deux manières de le faire :

1. Décrire l'implémentation de la méthode au niveau de sa déclaration. C'est la manière la plus simple mais elle présente un sérieux défaut : ne pas séparer l'implémentation de la déclaration ce qui est contraire au principe de dissimulation.
2. Par opposition aux méthodes décrites dans la déclaration d'une classe, on appelle *méthode déportée* une méthode dont le code n'est pas transcrit dans la déclaration de sa classe mais en dehors. Notons qu'il est néanmoins possible de faire développer `inline` une méthode déportée. Il faut alors préfixer sa déclaration ainsi que sa définition du mot clé `inline`. En outre, il faut donner son implémentation dans le fichier de déclaration à la suite de la déclaration de la classe. En effet, si vous souhaitez que le compilateur puisse développer le code de la méthode sur le lieu de l'appel, il faut qu'il connaisse sa taille !

Exemple : Classe Point

Nous allons écrire une classe `Point` en utilisant des méthodes `inline`. Nous allons mettre `inline` les deux méthodes d'accès aux attributs ainsi que le constructeur. Afin d'exploiter toutes les possibilités, les méthodes d'accès aux attributs seront placées `inline` dans la déclaration alors que le constructeur sera mis `inline` externe. En respectant la structure en deux fichiers, nous obtenons finalement :

```
#ifndef POINT_HEADER
#define POINT_HEADER
class Point
{
  public:
    // Constructeur déclaré inline
    // le compilateur va rechercher le code plus loin
    inline Point(int absc, int ordo);

    int x(void) const                // Déclaration et définition inline
```

```
    {
      return abscisse_;
    }
    int y(void) const                 // Déclaration et définition inline
    {
      return ordonnee_;
    }
    void deplacerDe(int incX, int incY); // méthode normale NON inline
    void deplacerVers(int dX, int dY);   // méthode normale NON inline
  private:
    int abscisse_;
    int ordonnee_;
};

// Definition inline deportee du constructeur
inline Point::Point(int absc, int ordo)
{
  abscisse_=absc;
  ordonnee_=ordo;
}
#endif
```

```
#include "Point.hxx"
void Point::deplacerDe(int incX, int incY)
{
  abscisse_+=incX;
  ordonnee_+=incY;
}

void Point::deplacerVers(int dX, int dY)
{
  abscisse_=dX;
  ordonnee_=dY;
}
```