

Programmation C++



Facile



Normal



Difficile



Professionnel



Expert

https://wiki.waze.com/wiki/Your_Rank_and_Points



- 1 - Généralités
- 2 - Les Notions de Base C/C++
- 3 - Les Classes et les Objets en C++
- 4 - Compléments
- 5 - Bibliographie



1 - Généralités



1.1 - Programmation Orientée Objet

1.2 - Du C au C++

1.3 - Environnement de Développement



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



1 – Généralités

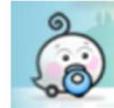
-> Programmation Orientée Objet : Les Concepts



www.yantra-technologies.com

Programmation Orientée Objet

Les Concepts



Facile



Normal



Difficile



Professionnel



Expert

https://wiki.waze.com/wiki/Your_Rank_and_Points

david.palermo@yantra-technologies.com

1



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020

Programmation Orientée Objet

Les Concepts



Facile



Normal



Difficile



Professionnel



Expert

https://wiki.waze.com/wiki/Your_Rank_and_Points



Sommaire

- 1 - Présentation
- 2 - Concepts de bases
- 3 - Quelques Design Patterns
- 4 - Différences entre les langages





- 1.1 - Les styles de programmation
- 1.2 - L'approche Objet et Langage UML
- 1.3 - Qu'est-ce qu'un Objet ?
- 1.4 - La modélisation avec UML
- 1.5 - UML
- 1.6 - Utilisation UML
- 1.7- Les diagrammes UML 2.0
- 1.8 - L'approche orientée objet
- 1.9 - Langage Objet
- 1.10 - Les avantages de la POO

1.1 - Les styles de programmation



- **Fonctionnel** : évaluation d 'expressions comme une formule, et d'utiliser le résultat pour autre chose *Lisp, Caml, APL* (récursivité)
- **Procédural ou Impératif** : exécution d 'instructions étape par étape *Fortran, C, Pascal, Cobol* (itératif)
- **Logique** : répondre à une question par des recherches sur un ensemble, en utilisant des axiomes, des demandes et des règles de déduction *Prolog*
- **Objet** : *Simula, Smalltalk, C++, Java, ADA 95*
 - ensemble de composants autonomes (objets) qui disposent de moyens d 'interaction,
 - utilisation de classes,
 - échange de message.

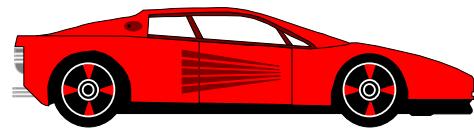


L'Approche Objet est une **démarche** qui consiste à utiliser l'**objet** pour **modéliser** le monde réel.

- Une démarche => Une méthodologie pour décrire comment s'organiser le système
- Une modélisation => Une Notation **UML**



1.2 - Qu'est-ce qu' un Objet ? (1)



propriétés

Ferrari
rouge
En_stationnement

comportements

rouler
se_garer

identité

ma_ferrari
305 XV 13





En 1994, plus de 50 méthodes OO

- Fusion, Shlaer-Mellor, ROOM, Classe-Relation, Wirfs-Brock, Coad-Yourdon, MOSES, Syntropy, BOOM, OOSD, OSA, BON, Catalysis, COMMA, HOOD, Ooram, DOORS...

Les métamodèles se ressemblent de plus en plus

Les notations graphiques sont toutes différentes

L'industrie a besoin de standards

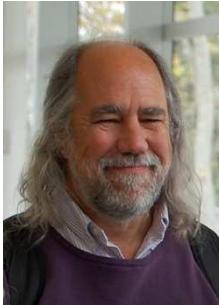


La pratique des méthodes a permis de faire le tri entre les différents concepts

Jim Rumbaugh, Grady Booch (1993) et plus tard ***Ivar Jacobson*** (1994) décident d'unifier leurs travaux:

- Methode OMT(Object Modeling Technique)
- Methode Booch
- Methode OOSE (Object Oriented Software Engineering)

1.1.3 - Les créateurs de la notation UML



Grady Booch

Méthode de Grady Booch

La méthode proposée par G. Booch est une méthode de conception, définie à l'origine pour une programmation Ada, puis généralisée à d'autres langages. Sans préciser un ordre strict dans l'enchaînement des opérations



James Rumbaugh

Méthode OMT

La méthode OMT (Object Modeling Technique) permet de couvrir l'ensemble des processus d'analyse et de conception en utilisant le même formalisme. L'analyse repose sur les trois points de vue: statique, dynamique, fonctionnel. donnant lieu à trois sous-modèles.



Ivar Jacobson

Méthode OOSE

Object Oriented Software Engineering (OOSE) est un langage de modélisation objet créé par Ivar Jacobson. OOSE est une méthode pour l'analyse initiale des usages de logiciels, basée sur les « cas d'utilisation » et le cycle de vie des logiciels.

1.2 - Qu'est-ce qu 'un Objet ? (2)



un **objet** représente un concept du monde réel possédant un **état** auquel peuvent être associés des **propriétés** et des **comportements**.

- L '**état** d'un objet comprend toutes les propriétés d'un objet (habituellement statiques) plus les valeurs courantes de celles-ci (habituellement dynamiques).
- Une **propriété** d'un objet est une caractéristique inhérente et distincte qui contribue à l 'unicité de l 'objet.
- Le **comportement** d'un objet c'est comment l'objet agit et réagit en termes de changement d'état et de passage de message.



UML : Unified Modeling Language



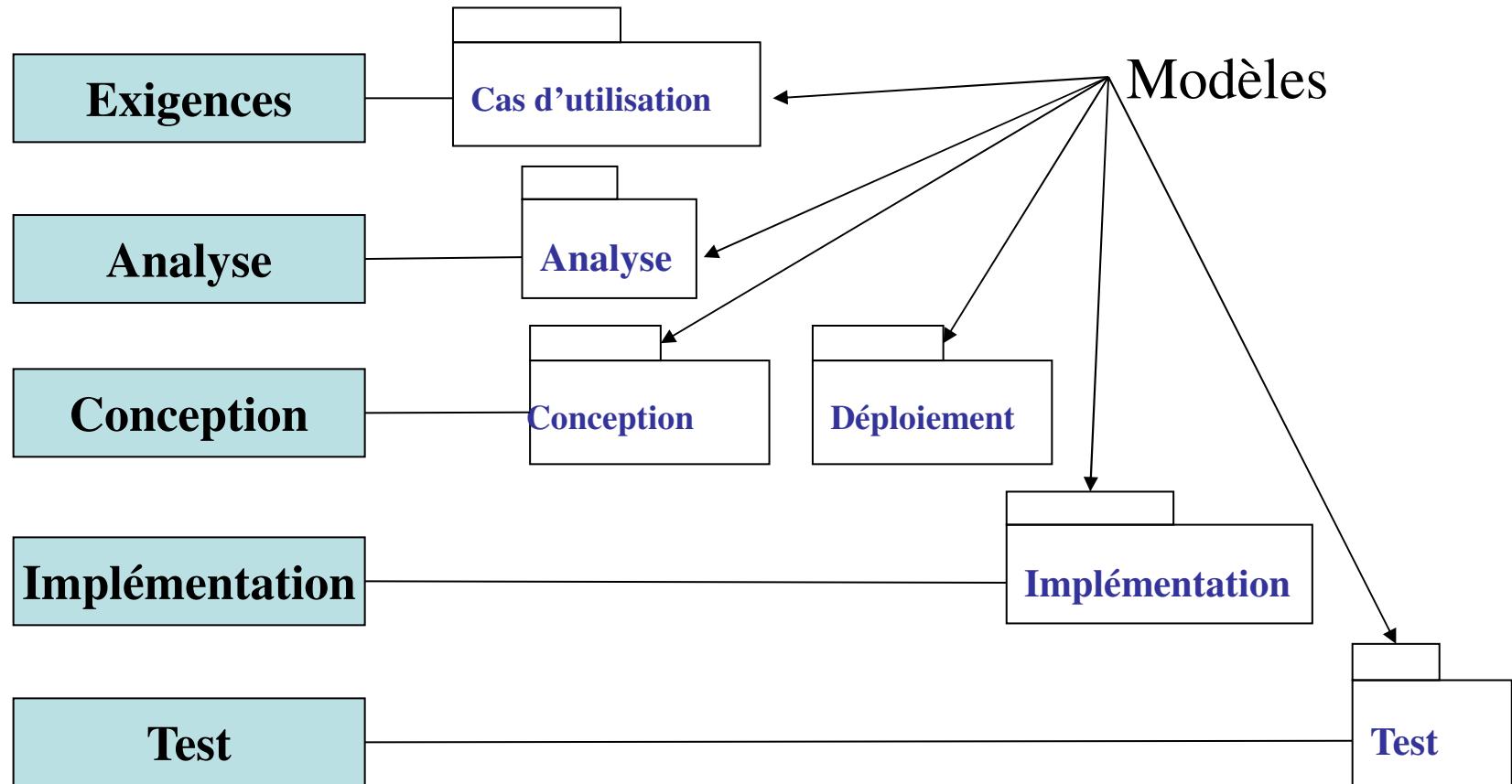
- **UML permet d'exprimer et d'élaborer des modèles objet indépendamment de tout langage de programmation.** Il a été pensé pour servir de support à une analyse basée sur les concepts objet.
- UML est un **langage formel**, défini par un **métamodèle**.
- Le métamodèle d'UML décrit de manière très précise tous les éléments de modélisation et la sémantique de ces éléments (leur définition et le sens de leur utilisation).
UML normalise les concepts objet.
- UML est avant tout **un support de communication performant**, qui facilite la représentation et la compréhension de solutions objet



- Les points forts d'UML
 - UML est un langage formel et normalisé
 - UML est un support de communication performant
- Les points faibles d'UML
 - La mise en pratique d'UML nécessite un apprentissage et passe par une période d'adaptation.
 - Le processus (non couvert par UML) est une autre clé de la réussite d'un projet.

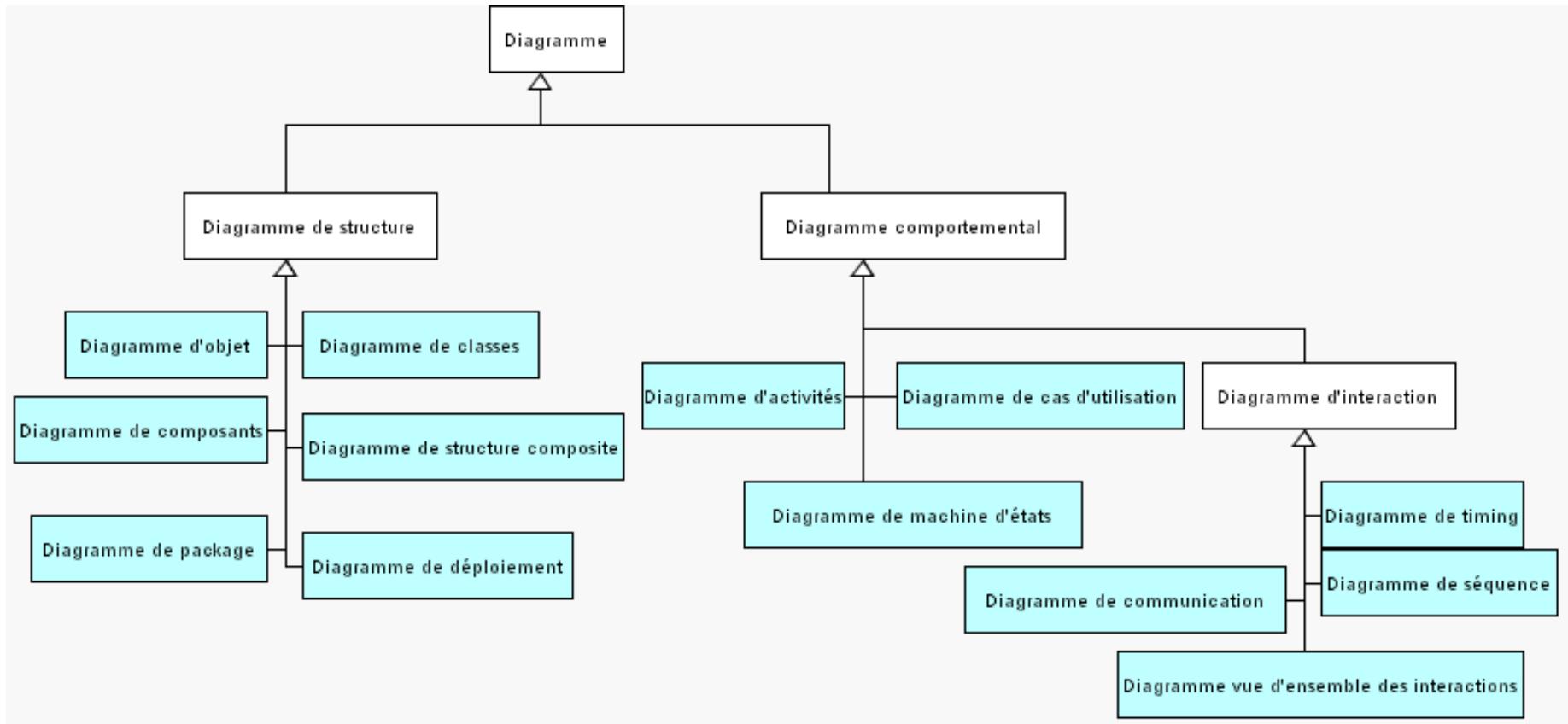


Modélisation UML





Modélisation UML 2.0



1.8 - L'approche orientée objet

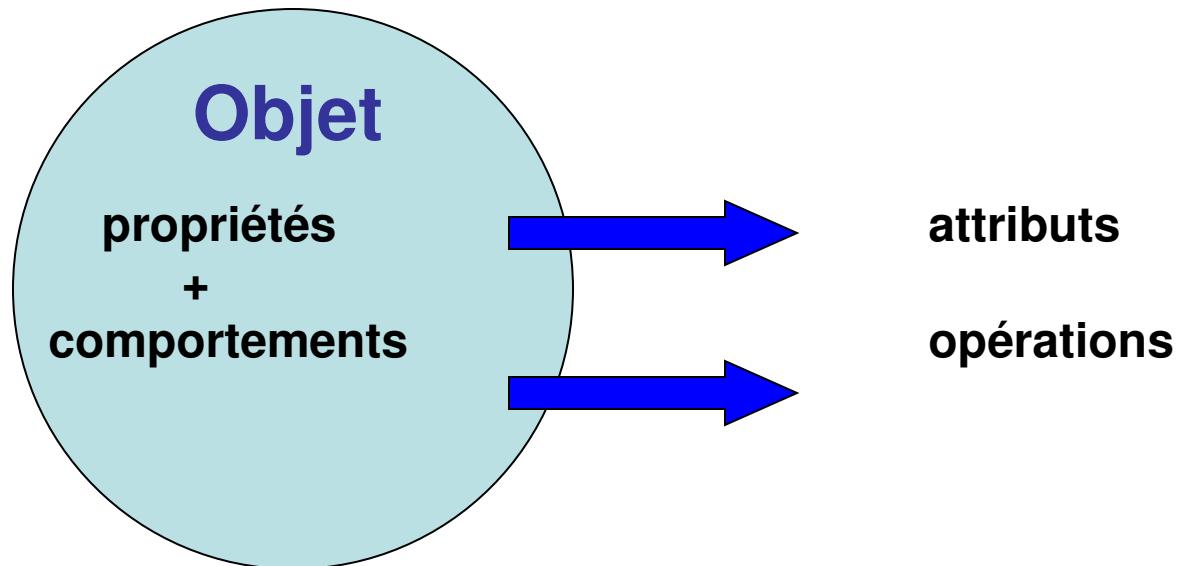


L'approche orientée objet considère le logiciel comme une collection d'objets dissociés définis par des **propriétés**. (propriété : **attribut** ou une **opération**)

- ⇒ Un objet comprend à la fois une structure de données et une collection d'opérations (son comportement).
- ⇒ Un certain nombre de caractéristiques pour qu'une approche soit dite orientée objet il faut : l'identité, la classification, le polymorphisme et l'héritage.



Le modèle objet





3 concepts pour faire un langage objet :

- **Encapsulation** : combiner des données et un comportement dans un emballage unique,
- **Héritage** : chien et chat sont des mammifères, ils héritent du comportement du mammifère.
- **Polymorphisme** : Cercle et rectangle sont des formes géométriques, chacun doit calculer sa surface.



- **Facilite la programmation modulaire :**
 - composants réutilisables,
 - un composant offre des services et en utilise d'autres,
 - il expose ses services au travers d'une interface.
- **Facilite l'abstraction :**
 - elle sépare la définition de son implémentation,
 - elle extrait un modèle commun à plusieurs composants,
 - le modèle commun est partagé par le mécanisme d'héritage.
- **Facilite la spécialisation :**
 - elle traite des cas particuliers,
 - le mécanisme de dérivation rend les cas particuliers transparents.



Yantra Technologies



D.Palermo

Programmation Orientée Objet Les Concepts

Version 1.0 - 02/2019

19

Copyright : Yantra Technologies 2004-2019



Yantra Technologies



D.Palermo

Programmation Orientée Objet Les Concepts

Version 1.0 - 02/2019

20

Copyright : Yantra Technologies 2004-2019



- 1- Présentation de la notion d 'Objet
- 2- La modélisation avec UML
- 3- Le principe de l'encapsulation
- 4- La Classe
- 5 -L'Héritage
- 6- Agrégation
- 7- Polymorphisme
- 8- Les classes abstraites
- 9- Relations d'association
- 10- Le Langage Objet
- 11- Développement d 'un projet orienté Objet

2.1- Présentation de la notion d'Objet

Un Objet peut :

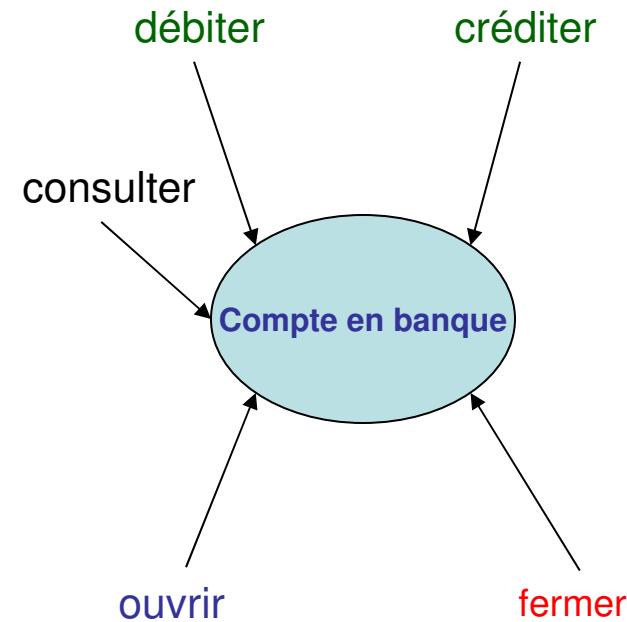
- changer d 'état,
- se comporter de façon discernable,
- être manipulé par diverses formes de stimuli,
- être en relation avec d 'autres objets.

Chaque objet a sa propre **identité** et donc une existence indépendante des autres.

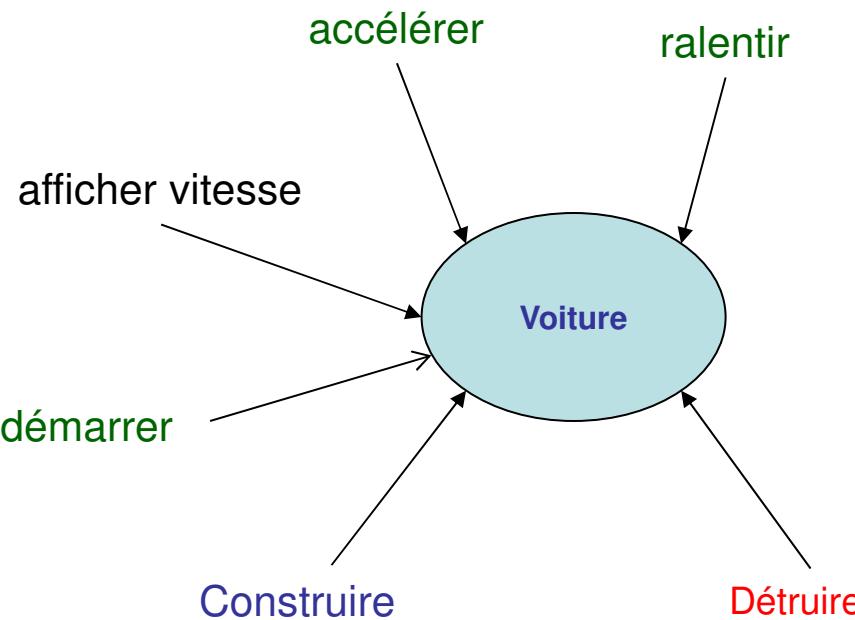
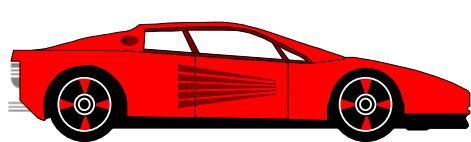


Types d'opération sur le comportement

- **Modificateur** : une opération qui altère l'état d'un objet,
- **Sélecteur** : une opération qui accède à l'état d'un objet
- **Itérateur** : une opération qui permet d'avoir accès à toutes les parties d'un objet dans un ordre défini,
- **Constructeur** : une opération qui crée un objet et initialise son état,
- **Destructeur** : une opération qui détruit un objet.



2.1- Présentation de la notion d'Objet



propriétés

Ferrari
rouge
en_stationnement

comportements

rouler
se_garer

identité

ma_ferrari
305 XV 13

class Test Model

```

Voiture
+ afficherVitesse(): void
+ demarrer(): void
+ accelerer(): void
+ ralentir(): void
<<constructor>>
+ Voiture(): void
<<destructor>>
+ ~Voiture(): void

```

2.1- Objet : C++



```
void Exemple01_Cpp()
{
    std::cout << "Exemple 1 : C++" << std::endl;
    Voiture* maFerrari = new Voiture("305 XV 13", "Ferrari", "rouge");

    maFerrari->demarrer();
    maFerrari->afficherVitesse();
    maFerrari->accelerer();
    maFerrari->afficherVitesse();
    maFerrari->ralentir();
    maFerrari->afficherVitesse();
    maFerrari->arreter();
    maFerrari->afficherVitesse();

    delete maFerrari;
}
```

2.1 – Objet : procédural C



```
void Exemple01_C()
{
    std::cout << "Exemple 1 : C" << std::endl;
    Voiture_struct* maFerrari = Voiture_construire("305 XV 13", "Ferrari", "rouge");

    Voiture_demarrer(maFerrari);

    Voiture_afficherVitesse(maFerrari);
    Voiture_accelerer(maFerrari);
    Voiture_afficherVitesse(maFerrari);
    Voiture_ralentir(maFerrari);
    Voiture_afficherVitesse(maFerrari);
    Voiture_arreter(maFerrari);
    Voiture_afficherVitesse(maFerrari);

    Voiture_detruire(&maFerrari);
}
```



L' Encapsulation

C'est le processus qui consiste à cacher tous les détails d'un objet. L'objet peut être vu :

- de **l'intérieur** pour le concepteur : détail de la structure et du comportement, données et méthodes privées,
- de **l'extérieur** pour l'utilisateur : interface « publique » qui décrit l'utilisation de l'objet.

Permet de rendre indépendante la spécification de l'objet et son implémentation (*la vue externe doit être la plus indépendante possible de la vue interne*).

2.4 – Notion de classe



On appelle **classe** la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc « issu » d'une classe, c'est le produit qui sort d'un moule.

On dit qu'un objet est une **instanciation** d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'**objet** ou d'**instance** (éventuellement d'*occurrence*).

Une classe est composée de deux parties :

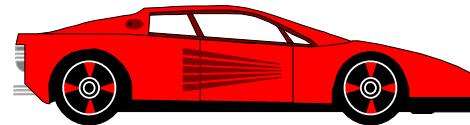
- **Les attributs** (parfois appelés *données membres*) : il s'agit des données représentant l'état de l'objet
- **Les méthodes** (parfois appelées *fonctions membres*): il s'agit des opérations applicables aux objets

Si on définit la classe *voiture*, les objets *Peugeot 406*, *Renault 18* seront des instantiations de cette classe. Il pourra éventuellement exister plusieurs objets *Peugeot 406*, différenciés par leur numéro de série. Mieux: deux instantiations de classes pourront avoir tous leurs attributs égaux sans pour autant être un seul et même objet. C'est le cas dans le monde réel, deux T-shirts peuvent être strictement identiques et pourtant ils sont distincts. D'ailleurs, en les mélangeant, il serait impossible de les distinguer...



Une classe

C'est un ensemble d'objets ayant les mêmes propriétés et un comportement commun.





- Une **classe** est une construction du langage de programmation :
 - décrit les propriétés communes à des objets
Class Point { }
 - un objet est une *instance* de classe
- Un **objet** est une entité en mémoire :
 - il a un état, un comportement, une identité.
Point* a = new Point(3,5); (C++)
Point a = new Point(3,5); (java)

Un objet sans classe n'existe pas



La classe possède :

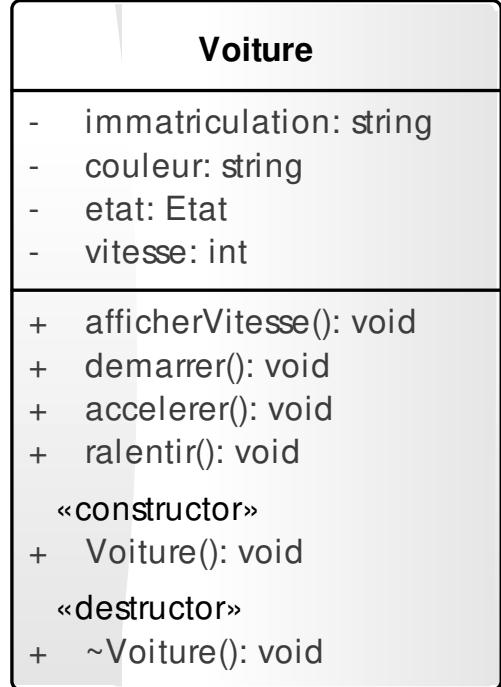
- un nom unique,
- une composante **statique** : les données qui sont des champs (attributs)
 ⇒ Etat
- une composante **dynamique** : les méthodes (opérations)
 ⇒ Comportement

Article
 Référence : int  Désignation : String  PrixHT : float  Quantité : int
 PrixTTC() : float  PrixTransport() : float  Retirer(q : int) : int  Ajouter(q : int) : int

2.4- La Classe



```
class Test Model
```



Etat est soit "arreter" soit "demarrer"

2.4- La Classe : C++



```
class Voiture
{
public:
    Voiture(std::string immatriculation, std::string type, std::string couleur);

    ~Voiture();
    void demarrer();
    void afficherVitesse() const;
    void accelerer();
    void ralentir();
    void arreter();

protected:
    const std::string _immatriculation;
    const std::string _type;
    std::string _couleur;

    std::string _etat;
    unsigned int _vitesse;

};
```

2.4- La Classe : C++



```

Voiture::Voiture(std::string immatriculation, std::string type, std::string couleur):
    _immatriculation(immatriculation),
    _type(type),
    _couleur(couleur),
    _etat("arreter"),
    _vitesse(0)
{
    std::cout << "Voiture : construction " << endl;
}

Voiture::~Voiture()
{
    std::cout << "Voiture : destruction " << endl;
}
void Voiture::Voiture::demarrer()
{
    std::cout << "Voiture : demarrer " << endl;
    this->_etat="demarrer";
}
void Voiture::afficherVitesse() const
{
    std::cout << "Vitesse :" << this->_vitesse << std::endl;
}
void Voiture::accelerer()
{
    std::cout << "Voiture : accelerer " << endl;
    if ( this->_etat=="demarrer") this->_vitesse++;
}
void Voiture::ralentir()
{
    std::cout << "Voiture : ralentir " << endl;
    if ( this->_vitesse != 0 ) this->_vitesse--;
}
void Voiture::arreter()
{
    std::cout << "Voiture : arreter " << endl;
this->_vitesse=0;
this->_etat="arreter";
}

```

2.1- Objet : C++



```
void Exemple01_Cpp()
{
    std::cout << "Exemple 1 : C++" << std::endl;
    Voiture* maFerrari = new Voiture("305 XV 13", "Ferrari", "rouge");

    maFerrari->demarrer();
    maFerrari->afficherVitesse();
    maFerrari->accelerer();
    maFerrari->afficherVitesse();
    maFerrari->ralentir();
    maFerrari->afficherVitesse();
    maFerrari->arreter();
    maFerrari->afficherVitesse();

    delete maFerrari;
}
```

2.4- La Classe : procédural C



```
typedef struct voiture_struct
{
    char* _immatriculation;
    char* _type;
    char* _couleur;

    char _etat[260];
    unsigned int _vitesse;
} Voiture_struct;

Voiture_struct* Voiture_construire(const char* immatriculation, const char* type, const char* couleur);
void Voiture_demarrer(Voiture_struct* voiture);
void Voiture_afficherVitesse(const Voiture_struct* voiture);
void Voiture_accelerer(Voiture_struct* voiture);
void Voiture_arreter(Voiture_struct* voiture);
void Voiture_detruire(Voiture_struct** voiture);
```

2.4- La Classe : procédural C



```
Voiture_struct* Voiture_construire(const char* immatriculation,const char* type,const char* couleur){  
    Voiture_struct* res = NULL;  
    printf("Voiture : construction\n");  
    if ( (immatriculation == NULL) || type == NULL || couleur == NULL ) exit(2);  
    res = (Voiture_struct* ) malloc( sizeof(Voiture_struct));  
    res->_immatriculation= (char*) malloc(sizeof(char)*strlen(immatriculation));  
    strcpy(res->_immatriculation,immatriculation);  
    res->_type= (char*) malloc(sizeof(char)*strlen(type));  
    strcpy(res->_type,type);  
    res->_couleur= (char*) malloc(sizeof(char)*strlen(couleur));  
    strcpy(res->_couleur,couleur);  
  
    strcpy(res->_etat,"arreter");  
    res->_vitesse = 0;  
    return res;  
}
```

2.4- La Classe : procédural C



```

void Voiture_demarrer(Voiture_struct* voiture)
{
    printf("Voiture : demarrer \n");
    strcpy(voiture->_etat,"demarrer");
}
void Voiture_afficherVitesse(const Voiture_struct* voiture) {
    printf("Vitesse : %d \n" , voiture->_vitesse);
}
void Voiture_accelerer(Voiture_struct* voiture)
{
    printf("Voiture : accelerer \n");
    if ( strcmp(voiture->_etat,"demarrer") == 0 )  voiture->_vitesse++;
}
void Voiture_ralentir(Voiture_struct* voiture) {
    printf("Voiture : ralentir\n");
    if ( voiture->_vitesse != 0 ) voiture->_vitesse--;
}
void Voiture_arreter(Voiture_struct* voiture) {
    printf("Voiture : arreter \n");
    strcpy(voiture->_etat,"arreter");
    voiture->_vitesse = 0;
}
void Voiture_detruire(Voiture_struct** voiture){
    printf("Voiture : destruction\n");
    if ( voiture == NULL) return;
    if ( *voiture == NULL) return;
    if ( (*voiture)->_immatriculation != NULL ) free((*voiture)->_immatriculation);
    if ( (*voiture)->_type != NULL ) free((*voiture)->_type);
    if ( (*voiture)->_couleur != NULL ) free((*voiture)->_couleur);
    free(*voiture);
    *voiture=NULL;
}

```



1. Naissance d'un objet (*constructeur*)

- Allouer de la mémoire
- Initialiser cette mémoire

2. Vie d'un objet

- Utilisation des méthodes en modification, sélection et itération

3. Mort d'un objet (*destructeur*)

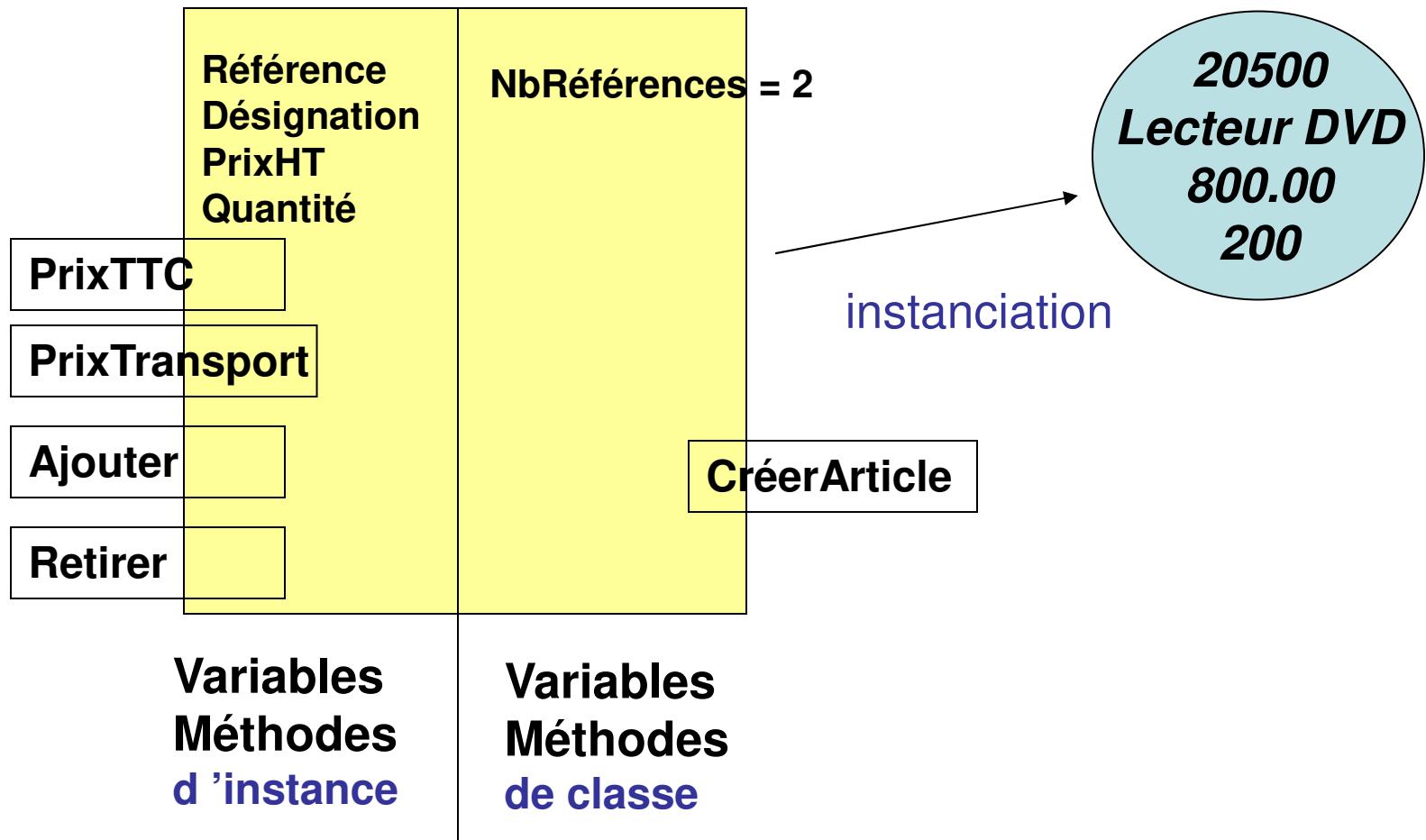
- Libérer la mémoire allouée dynamiquement
- Rendre les ressources systèmes
- autres.....



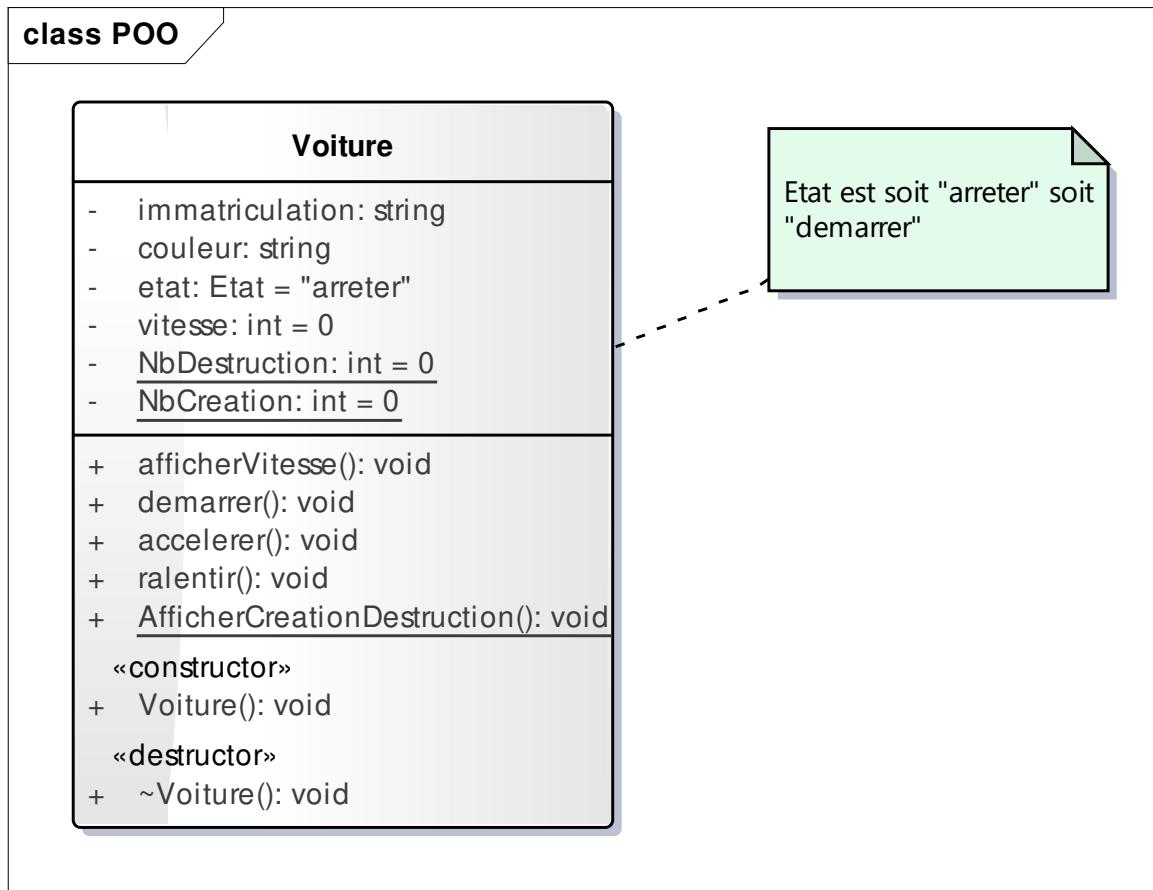
2.4- La Classe



Méthodes, variables d'instance et de classe



2.4- La Classe



2.4- La Classe : C++



```
class Voiture
{
public:
    Voiture(std::string immatriculation, std::string type, std::string couleur);

    ~Voiture();
    void demarrer();
    void afficherVitesse() const;
    void accelerer();
    void ralentir();
    void arreter();

    static void AfficherCreationDestruction();

protected:
    const std::string _immatriculation;
    const std::string _type;
    std::string _couleur;

    std::string _etat;
    unsigned int _vitesse;

    static unsigned int NbCreation;
    static unsigned int NbDestruction;
};


```

2.4- La Classe : C++



```
unsigned int Voiture::NbCreation = 0;
unsigned int Voiture::NbDestruction = 0;

void Voiture::AfficherCreationDestruction() {
    std::cout << "Nombre de voiture creee : " << Voiture::NbCreation << std::endl;
    std::cout << "Nombre de voiture detruite : " << Voiture::NbDestruction << std::endl;
}

Voiture::Voiture(std::string immatriculation, std::string type, std::string couleur):
    _immatriculation(immatriculation),
    _type(type),
    _couleur(couleur),
    _etat("arreter"),
    _vitesse(0)
{
    std::cout << "Voiture : construction " << endl;
    Voiture::NbCreation++;
}

Voiture::~Voiture()
{
    std::cout << "Voiture : destruction " << endl;
    Voiture::NbDestruction++;
}
```

2.4- Objet : C++



```
void Exemple01_Cpp()
{
    std::cout << "Exemple 1 : C++" << std::endl;
    Voiture* maFerrari = new Voiture("305 XV 13", "Ferrari", "rouge");

    maFerrari->demarrer();
    maFerrari->afficherVitesse();
    maFerrari->accelerer();
    maFerrari->afficherVitesse();
    maFerrari->ralentir();
    maFerrari->afficherVitesse();
    maFerrari->arreter();
    maFerrari->afficherVitesse();

    delete maFerrari;
}
```

2.4- La Classe : procédural C



```

static unsigned int Voiture_NbCreation = 0;
static unsigned int Voiture_NbDestruction = 0;

void Voiture_AfficherCreationDestruction() {
    printf("Nombre de voiture creee : %d \n", Voiture_NbCreation );
    printf("Nombre de voiture detruite : %d \n", Voiture_NbDestruction);
}

Voiture_struct* Voiture_construire(const char* immatriculation,const char* type,const char* couleur) {
    Voiture_struct* res = NULL;
    printf("Voiture : construction\n");
    if ( (immatriculation == NULL) || type == NULL || couleur == NULL ) exit(2);
    res = (Voiture_struct*) malloc( sizeof(Voiture_struct));
    res->_immatriculation= (char*) malloc(sizeof(char)*strlen(immatriculation));
    strcpy(res->_immatriculation,immatriculation);
    res->_type= (char*) malloc(sizeof(char)*strlen(type));
    strcpy(res->_type,type);
    res->_couleur= (char*) malloc(sizeof(char)*strlen(couleur));
    strcpy(res->_couleur,couleur);

    strcpy(res->_etat,"arreter");
    res->_vitesse = 0;
    Voiture_NbCreation++;
    return res;
}

void Voiture_detruire(Voiture_struct** voiture){
    printf("Voiture : destruction\n");
    if ( voiture == NULL) return;
    if ( *voiture == NULL) return;
    if ( (*voiture)->_immatriculation != NULL ) free((*voiture)->_immatriculation);
    if ( (*voiture)->_type != NULL ) free((*voiture)->_type);
    if ( (*voiture)->_couleur != NULL ) free((*voiture)->_couleur);
    free(*voiture);
    *voiture=NULL;
    Voiture_NbDestruction++;
}

```



En POO les attributs et méthodes d'une classe peuvent être visibles depuis les instances de toutes les classes d'une application.

En POO il faudrait que :

- Les attributs ne soient lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels.
- les méthodes "utilitaires" ne soient pas visibles, seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.

2.4- La Classe : encapsulation des attributs & méthodes



class POO

PetiteCalculatrice

- `resultat: float`
- + `additioner(float, float): float`
- + `getResultat(): float`
- + `effacer(): void`
- «constructor»
- + `PetiteCalculatrice(): void`
- «destructor»
- + `~PetiteCalculatrice(): void`

2.4- La Classe : C++



```
class PetiteCalculatrice
{
private :
    double resultat;

public:
    PetiteCalculatrice():resultat(0) {}

    double additionner(double nb1, double nb2)
    {
        return this->resultat = nb1 + nb2 ;
    }

    void effacer()
    {
        this->resultat = 0;
    }

    double getResultat() const
    {
        return this->resultat;
    }

};
```

2.4- La Classe : procédural C



```
typedef struct
{
    double resultat;
} PetiteCalculatrice_struct;

PetiteCalculatrice_struct* PetiteCalculatrice() {
    PetiteCalculatrice_struct* res = (PetiteCalculatrice_struct*)malloc(sizeof(PetiteCalculatrice_struct));
    res->resultat = 0;
    return res;
}

double PetiteCalculatrice_additionner(PetiteCalculatrice_struct* calc1, double nb1, double nb2)
{
    return calc1->resultat = nb1 + nb2 ;
}

void PetiteCalculatrice_effacer(PetiteCalculatrice_struct* calc)
{
    calc->resultat = 0;
}

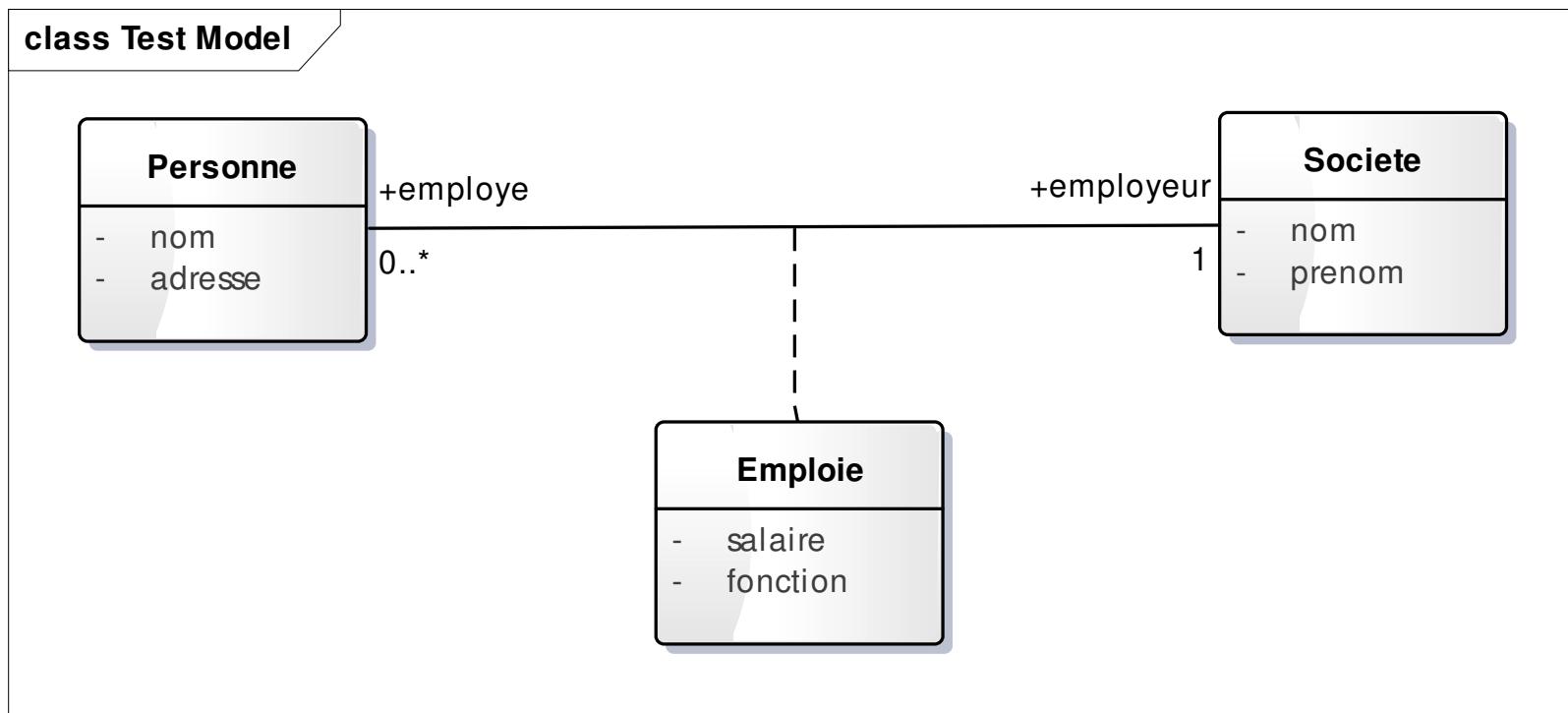
double PetiteCalculatrice_getResultat(const PetiteCalculatrice_struct* calc)
{
    return calc->resultat;
}
```



La relation d'association

- exprime une dépendance sémantique entre des classes
- une association est bidirectionnelle
- une association a une cardinalité
 - 0 ou 1
 - un pour un
 - un pour n
 - n pour n

2.9- Les Relations d'association



2.4- La Classe : C++



```
|class Personne {
private :
    std::string _nom;
    std::string _adresse;
public:
    Personne(std::string nom, std::string adresse) :_nom(nom), _adresse(adresse) {}
    Personne(const Personne& per) :_nom(per._nom), _adresse(per._adresse) {}
    std::string getAdresse() const { return this->_adresse; }
    std::string getNom() const { return this->_nom; }

};

|class Societe {
private :
    std::string _nom;
    std::string _adresse;
public:
    Societe(std::string nom, std::string adresse) :_nom(nom), _adresse(adresse) {}
    Societe(const Societe& soc) :_nom(soc._nom), _adresse(soc._adresse) {}
    std::string getAdresse() const { return this->_adresse; }
    std::string getNom() const { return this->_nom; }

};
```

2.4- La Classe : C++



```
class Emploi {  
private :  
    Societe _employeur;  
    std::map< unsigned, Personne>      _employe;  
    std::map< unsigned, std::string>     _emploi;  
    std::map< unsigned, unsigned int>    _salaire;  
  
public:  
  
    Emploi(std::string employeur);  
    void set(unsigned cle, Personne per, unsigned int salaire, std::string emploi );  
  
    Personne getPersonne(unsigned cle) const { return this->_employe.at(cle); }  
    unsigned int getSalaire(unsigned cle) const { return this->_salaire.at(cle); }  
    std::string getEmploi(unsigned cle) const { return this->_emploi.at(cle); }  
  
    unsigned nbEmployer() const { return this->_emploi.size(); }  
    Societe getSociete() const { return this->_employeur; }  
};
```

2.4- La Classe : procédural C



```

typedef struct
{
    char _nom[256];
    char _adresse[256];
} Personne_struct ;

Personne_struct *CreatePersonne(const char* nom,const char* adresse)
{
    Personne_struct *per=(Personne_struct *) malloc(sizeof(Personne_struct));
    strcpy(per->_nom,nom);
    strcpy(per->_adresse,adresse);
    return per;
}

Personne_struct * CopyPersonne(const Personne_struct* per){ return CreatePersonne(per->_nom,per->_adresse); }
std::string Personne_getAdresse(const Personne_struct* per){ return per->_adresse; }
std::string Personne_getNom(const Personne_struct* per) { return per->_nom; }

typedef struct
{
    char _nom[256];
    char _adresse[256];
} Societe_struct ;

Societe_struct *CreateSociete(const char* nom,const char* adresse)
{
    Societe_struct *per=(Societe_struct *) malloc(sizeof(Societe_struct));
    strcpy(per->_nom,nom);
    strcpy(per->_adresse,adresse);
    return per;
}

```

2.4- La Classe : procédural C



```

#define NBMAX 1000
typedef struct
{
    Societe_struct* _employeur;
    Personne_struct* _employe[NBMAX];
    char _emploi[NBMAX][256];
    unsigned _salaire[NBMAX];
    unsigned _cle;
} Emploi_struct;

Emploi_struct* CreateEmploi( std::string employeur) {
    Emploi_struct* lien=(Emploi_struct*) malloc(sizeof(Emploi_struct));
    lien->_employeur = employeur;
    return lien;
}

void Emploi_set(Emploi_struct* lien,Personne_struct* per,unsigned int salaire,const char* emploi )
{
    if ( lien->_cle >= NBMAX ) {printf("Nombre maximum de personnel atteint\n");return;}
    lien->_employe[lien->_cle]=per;
    lien->_salaire[lien->_cle]=salaire;
    strcpy(lien->_emploi[lien->_cle],emploi);
    lien->_cle++;
}

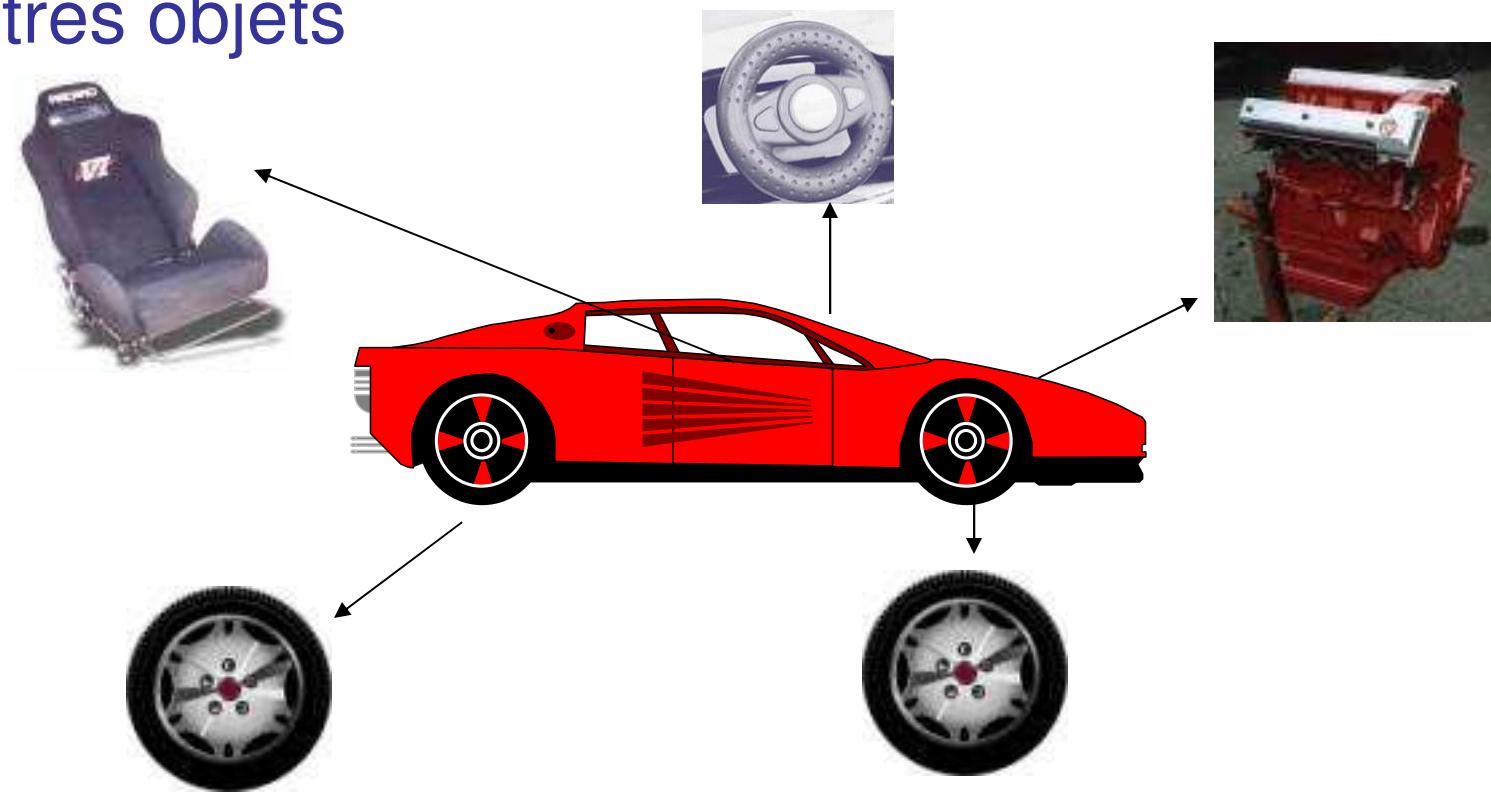
Personne_struct* Emploi_getPersonne(const Emploi_struct* lien,unsigned cle) {return lien->_employe[cle];}
unsigned int Emploi_getSalaire(const Emploi_struct* lien,unsigned cle) { return lien->_salaire[cle]; }
const char* Emploi_getEmploi(const Emploi_struct* lien,unsigned cle) { return lien->_emploi[cle];}
unsigned Emploi_nbEmployer(const Emploi_struct* lien){ return lien->_cle; }
Societe_struct* Emploi_getSociete(const Emploi_struct* lien) { return lien->_employeur; }

```

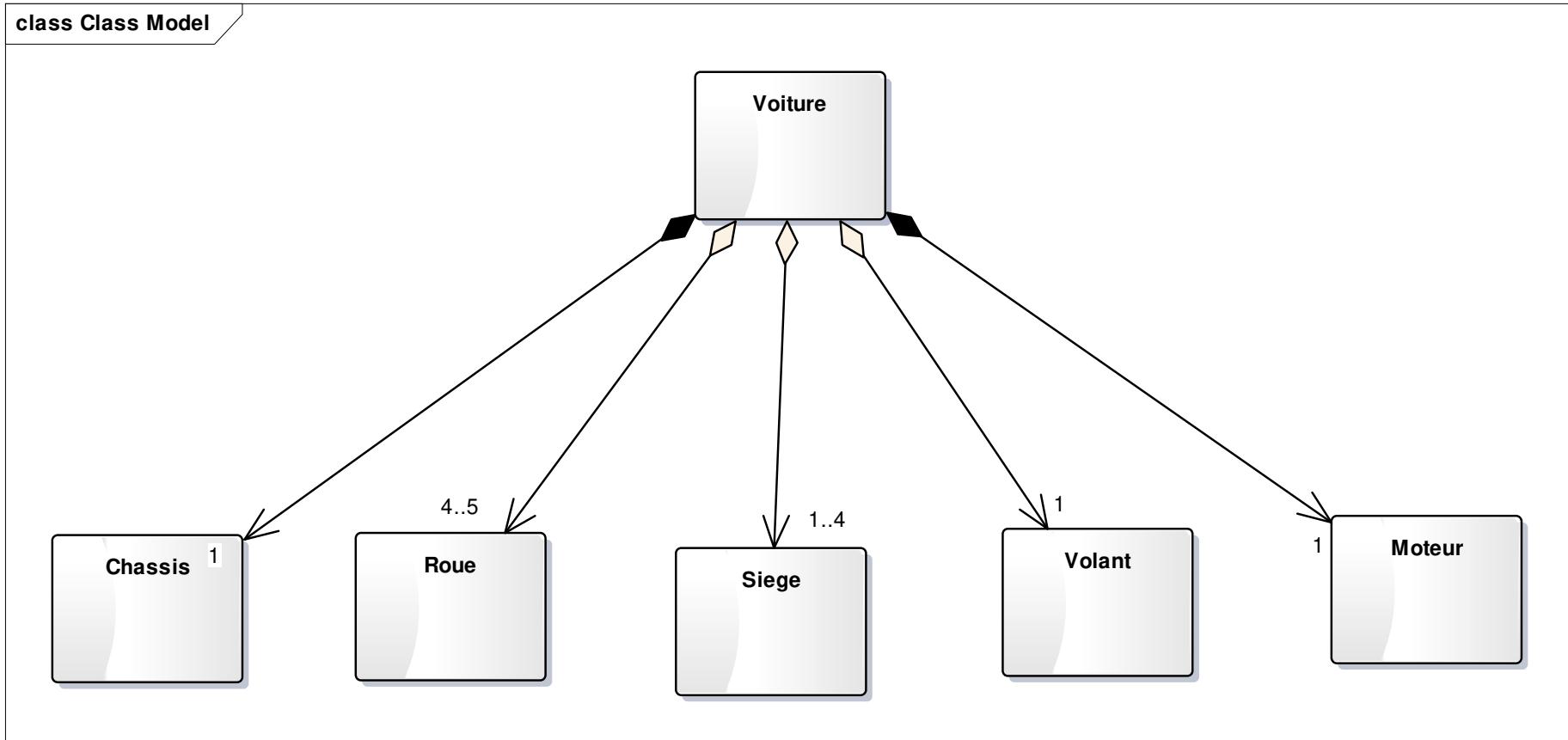


Agrégation

Les objets peuvent contenir des références à d'autres objets



2.6- L'Agrégation



2.4- La Classe : C++



```
class Chassis{};  
class Roue{};  
class Siege{};  
class Volant{};  
class Moteur{};  
  
class Voiture  
{  
private:  
    Chassis* _chassis;  
    Roue* _roue[4];  
    Siege* _siege[4];  
    Volant* _volant;  
    Moteur* _moteur;  
public:  
    Voiture();  
    ~Voiture(){  
        delete this->_chassis;  
        delete this->_moteur;  
    }  
};
```

2.4- La Classe : procédural C



```
typedef struct {} Chassis_struct;
typedef struct {} Roue_struct;
typedef struct {} Siege_struct;
typedef struct {} Volant_struct;
typedef struct {} Moteur_struct;

typedef struct
{
    Chassis_struct* _chassis;
    Roue_struct* _roue[4];
    Siege_struct* _siege[4];
    Volant_struct* _volant;
    Moteur_struct* _moteur;
} Voiture_struct;

Voiture_struct* CreateVoiture();
void Voiture_detruire(Voiture_struct* vo)
{
    free( vo->_chassis);
    free( vo->_moteur);
}
```



L 'Héritage

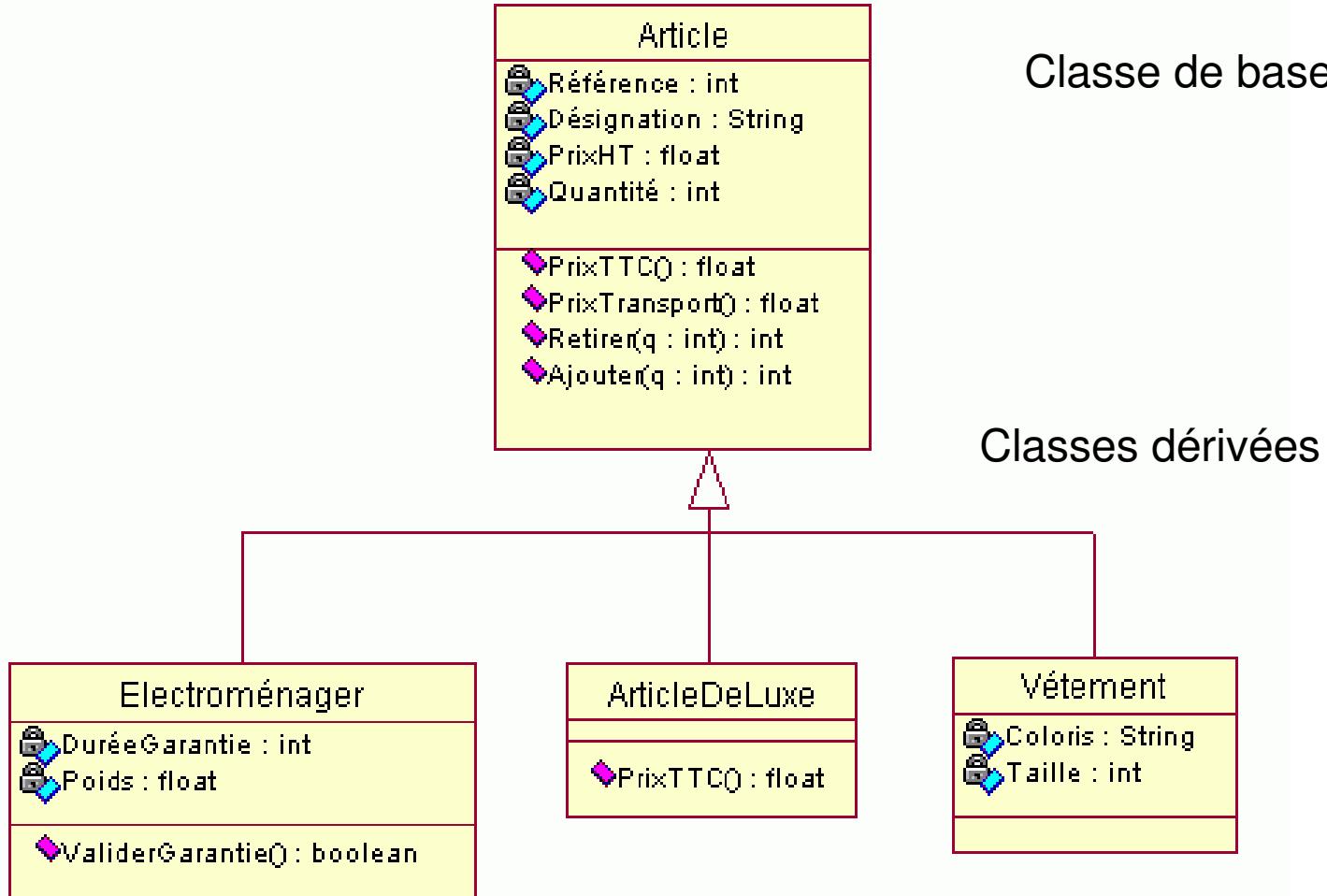
Par héritage, une classe dérivée possède les attributs et les méthodes de la superclasse.

- ⇒ La classe dérivée possède les attributs et méthodes de la classe de base,
- ⇒ la classe dérivée peut en ajouter ou en masquer,
- ⇒ facilite la programmation par raffinement,
- ⇒ facilite la prise en compte de la spécialisation.

2.5- L 'Héritage



GENERALISATION ↑ ↓ SPECIALISATION





Le Polymorphisme

C'est un concept selon lequel le même nom peut désigner des méthodes différentes.

La méthode désignée dépend de l'objet auquel on s'adresse.

- Le polymorphisme **Statique**: l'objet est connu à la compilation
- Le polymorphisme **Dynamique**: l'objet n'est connu qu'ultérieurement. L'identité ne sera qualifiée qu'au moment de l'exécution.

2.7- Le Polymorphisme



```
class Forme
```

```
{ afficher(); }
```

```
class Rectangle
```

```
{afficher();}
```

```
class Cercle
```

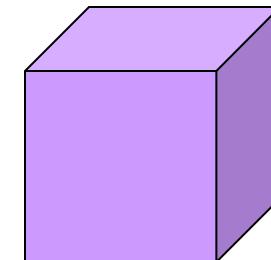
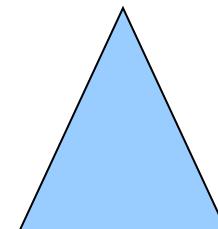
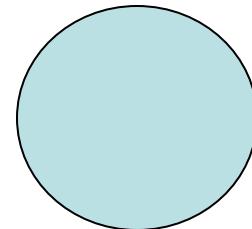
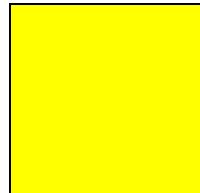
```
{afficher();}
```

STATIQUE

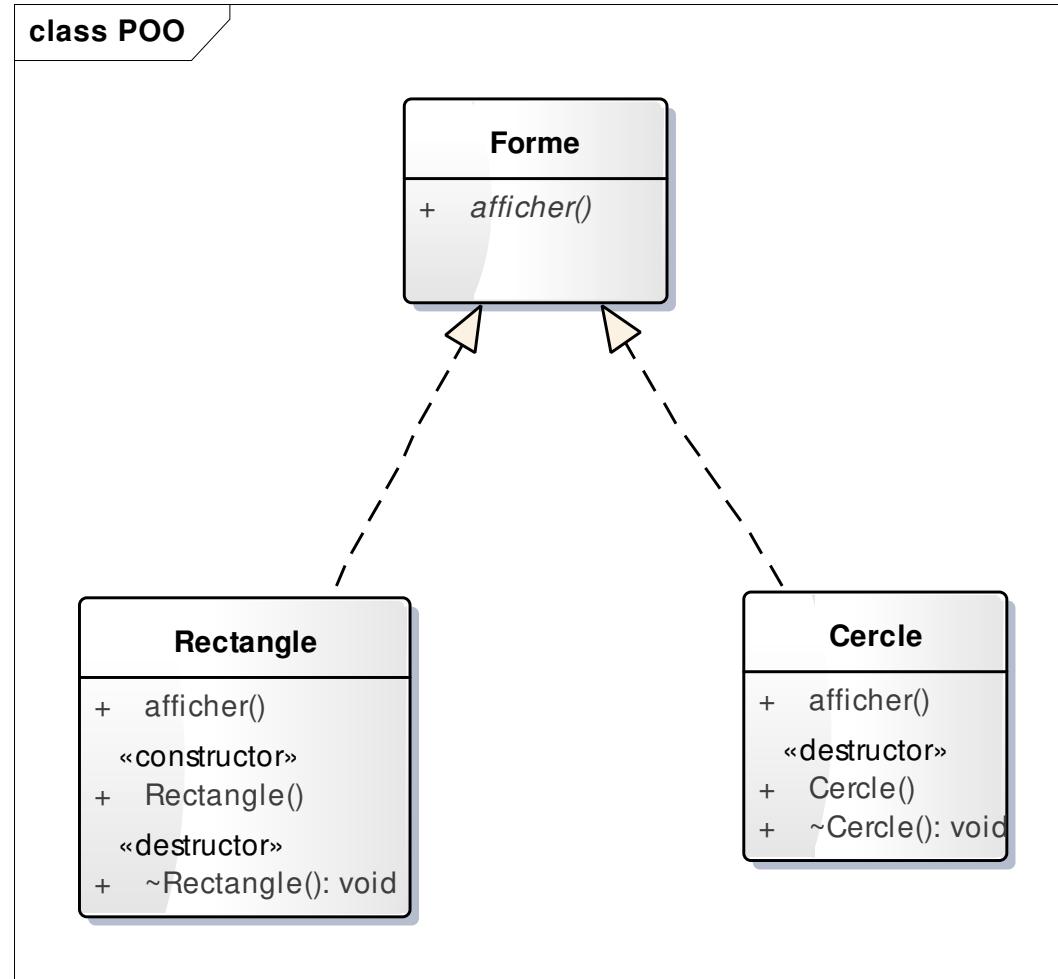
```
    un Rectangle.afficher();  
    un Cercle.afficher();
```

DYNAMIQUE

```
forme.afficher();
```



2.7- Le Polymorphisme





2.4- La Classe : C++



```
#include <iostream>

using namespace std;

class Forme
{
public:
    virtual void afficher() const = 0;
};

class Rectangle:public Forme
{
public:
    void afficher() const { cout << "je suis un rectangle" << endl; }

};

class Cercle:public Forme
{
public:
    void afficher() const{cout << "je suis un cercle" << endl; }

};

void afficherForme(const Forme& f) { f.afficher();}

int main()
{
    Cercle cer;
    Rectangle rec;
    cer.afficher();
    rec.afficher();

    afficherForme(cer);
    afficherForme(rec);
}
```

```
D:\YantraTechnologies\Yantra-Client\CEA\CEA_250315\POO6\bin\Debug>
je suis un cercle
je suis un rectangle
je suis un cercle
je suis un rectangle

Process returned 0 (0x0)   execution time : 0.041 s
Press any key to continue.
```



2.4- La Classe : procédural C



```
typedef enum { FORME, CERCLE, RECTANGLE } TypeForme;

typedef struct {
    TypeForme _type;
    void* _fils;
} Forme_struct;

typedef struct {
    Forme_struct* _parent;
} Rectangle_struct;

typedef struct {
    Forme_struct* _parent;
} Cercle_struct;
```

2.4- La Classe : procédural C



```
|Forme_struct * creerForme() {
    Forme_struct* res=(Forme_struct*)malloc(sizeof(Forme_struct));
    res->_type=FORME;
    res->_fils=NULL;
    return res;
}

|Cercle_struct * creerCercle() {
    Cercle_struct* res=(Cercle_struct*)malloc(sizeof(Cercle_struct));
    res->_parent=creerForme();
    res->_parent->_type=CERCLE;
    res->_parent->_fils=(void*)res;
    return res;
}
|Rectangle_struct *creerRectangle() {
    Rectangle_struct* res=(Rectangle_struct*)malloc(sizeof(Rectangle_struct));
    res->_parent=creerForme();
    res->_parent->_type=RECTANGLE;
    res->_parent->_fils=(void*)res;
    return res;
}
```

2.4- La Classe : procédural C



```

void afficherRectangle(Rectangle_struct *) { printf("je suis un rectangle\n"); }
void afficherCercle(Cercle_struct *) { printf("je suis un cercle\n"); }
void afficherForme(const Forme_struct* f)
{
    switch(f->_type) {
        case CERCLE:afficherCercle((Cercle_struct*)f);break;
        case RECTANGLE:afficherRectangle((Rectangle_struct*)f);break;
        default:break;
    };
}
int main()
{
    Cercle_struct *cer = creerCercle();
    Rectangle_struct *rec = creerRectangle();
    afficherCercle(cer);
    afficherRectangle(rec);
    afficherForme(cer->_parent);
    afficherForme(rec->_parent);
    free(cer->_parent);
    free(cer);
    free(rec->_parent);
    free(rec);
}

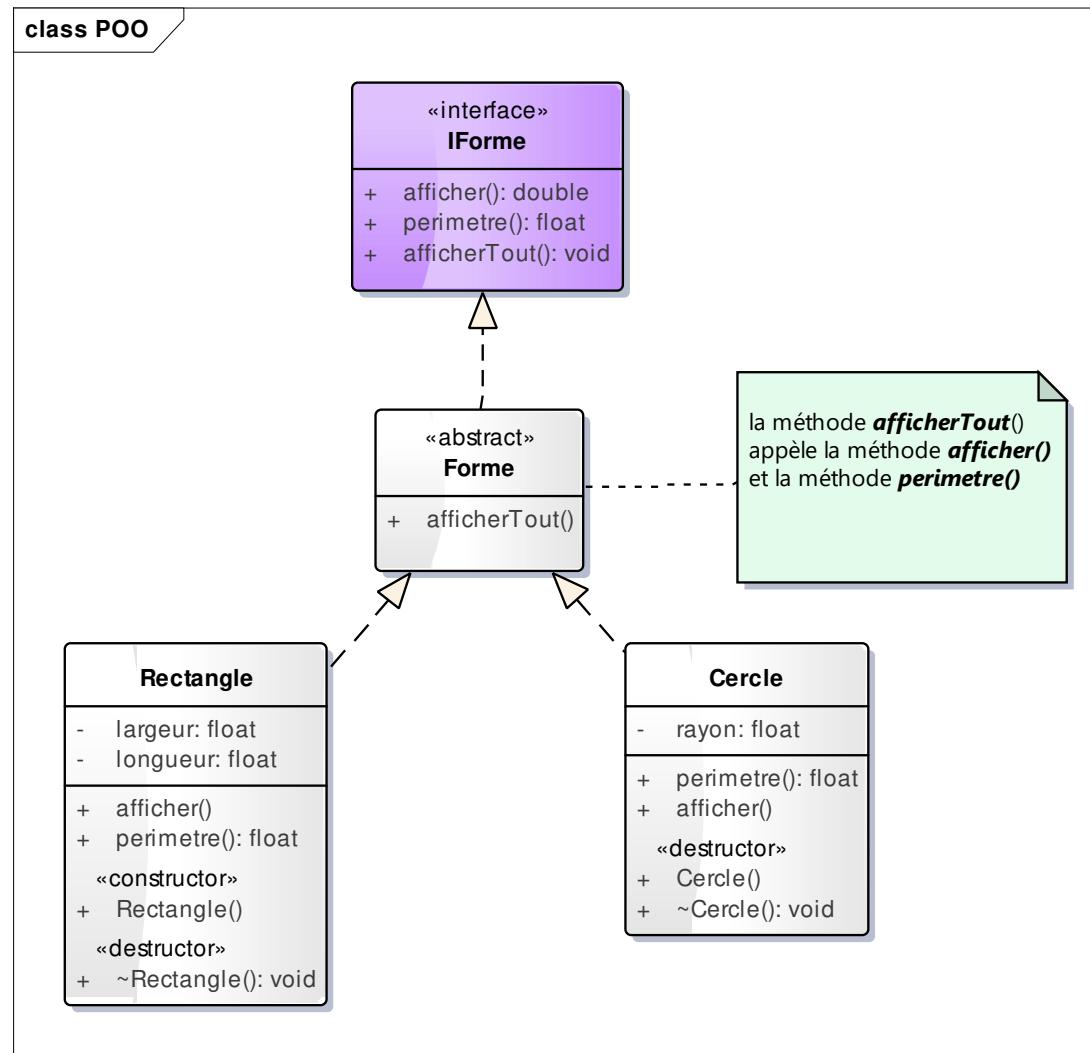
```



Les Classes Abstraites

- Elles représentent des concepts : *Mammifère*
- Elles peuvent contenir des méthodes abstraites :
 - méthodes sans implémentation (corps),
 - la mise en œuvre pour chaque sous-classe peut être différente (*calculerSurface*)
 - polymorphisme

2.8- Les Classes Abstraites



2.8- La Classe : C++



```

class Forme
{
public:
    virtual void afficher() const = 0;
    virtual double perimetre() const = 0;
    void afficherTout()
    {
        this->afficher();
        cout << "le perimetre de la forme est : " << this->perimetre() << endl;
    }
};

class Rectangle:public Forme
{
private:
    double _longeur;
    double _largeur;
public:
    Rectangle(double longeur,double largeur):
        _longeur(longeur),_largeur(largeur) {}
    void afficher() const{cout << "je suis un rectangle" << endl; }
    double perimetre() const { return (this->_longeur+this->_largeur) * 2 ; }
};

class Cercle:public Forme
{
private:
    double _rayon;
public:
    Cercle(double rayon):
        _rayon(rayon) {}
    void afficher() const{cout << "je suis un cercle" << endl; }
    double perimetre() const{return 2 * 3.1415 * this->_rayon; }
};

```

2.8- La Classe : C++



```
int main()
{
    Cercle cer(10);
    Rectangle rec(5, 2);
    cer.afficherTout();
    rec.afficherTout();

    return 0;
}
```

```
je suis un cercle
le perimetre de la forme est : 62.83
je suis un rectangle
le perimetre de la forme est : 14
```

2.8 - La Classe : procédural C



```
typedef enum { FORME, CERCLE, RECTANGLE} TypeForme;

typedef struct {
    TypeForme _type;
    void* _fils;
    double (*perimetre) (void* );
    void (*afficher) (void* );
} Forme_struct;

typedef struct {
    Forme_struct* _parent;
    double _largeur;
    double _longeur;
} Rectangle_struct;

typedef struct {
    Forme_struct* _parent;
    double _rayon;
} Cercle_struct;
```

2.8 - La Classe : procédural C



```
double Rectangle_perimetre(void * ele) {
    Rectangle_struct *rec = (Rectangle_struct *) ele;
    return (rec->_largeur + rec->_longeur)*2; }

double Cercle_perimetre(void *ele) {
    Cercle_struct *rec = (Cercle_struct *) ele;
    return 2*3.1415*rec->_rayon; }

double Forme_perimetre(const Forme_struct* f) {
    return f->perimetre(f->_fils); }
```

2.8 - La Classe : procédural C



```

Forme_struct * creerForme() {
    Forme_struct* res=(Forme_struct*)malloc(sizeof(Forme_struct));
    res->_type=FORME;
    res->_fils=NULL;
    return res;
}

Cercle_struct * creerCercle(double rayon) {
    Cercle_struct* res=(Cercle_struct*)malloc(sizeof(Cercle_struct));
    res->parent=creerForme();
    res->rayon = rayon;
    res->parent->_type=CERCLE;
    res->parent->_fils=(void*)res;
    res->parent->perimetre = Cercle_perimetre;
    res->parent->afficher = Cercle_afficher;
    return res;
}

Rectangle_struct *creerRectangle(double longueur,double largeur) {
    Rectangle_struct* res=(Rectangle_struct*)malloc(sizeof(Rectangle_struct));
    res->parent=creerForme();
    res->longueur=longueur;
    res->largeur=largeur;
    res->parent->_type=RECTANGLE;
    res->parent->_fils=(void*)res;
    res->parent->perimetre = Rectangle_perimetre;
    res->parent->afficher = Rectangle_afficher;
    return res;
}

```

2.8 - La Classe : procédural C



```

void libererForme( Forme_struct* f) {
    switch(f->_type) {
        case CERCLE:free((Cercle_struct*)f->_fils);break;
        case RECTANGLE:free((Rectangle_struct*)f->_fils);break;
        default:break;
    }
    free(f);
}

int main()
{
    Forme_struct *cer = creerCercle(10)->_parent;
    Forme_struct *rec = creerRectangle(5,2)->_parent;
    Forme_afficher(cer);
    printf("perimetre cercle %f\n",Forme_perimetre(cer));
    Forme_afficher(rec);
    printf("perimetre rectangle %f\n",Forme_perimetre(rec));
    libererForme(cer);
    libererForme(rec);
}

```

```

je suis un cercle
perimetre cercle 62.830000
je suis un rectangle
perimetre rectangle 14.000000

```



Il s'agit donc de l'ensemble des méthodes accessibles depuis l'extérieur de la classe, par lesquelles on peut modifier l'objet. Pour rappel la différenciation publique/privée ou portée de variable permet :

- d'éviter de manipuler l'objet de façon non voulue, en limitant ses modifications à celles autorisées comme publiques par le concepteur de la classe
- Au concepteur, de modifier l'implémentation interne de ces méthodes de manière transparente.

[https://fr.wikipedia.org/wiki/Interface_\(programmation_orient%C3%A9e_objet\)](https://fr.wikipedia.org/wiki/Interface_(programmation_orient%C3%A9e_objet))

2.10- Interface



class POO

«interface»

IForme

- + afficher(): double
- + perimetre(): float
- + afficherTout(): void

2.4- La Classe : C++



```
]class IForme {
    public:
        virtual void afficher() const = 0;
        virtual double perimetre() const = 0;
        virtual void afficherTout() const =0;
};
```

2.4- La Classe : procédural C



```
]typedef struct  {
    TypeForme _type;
    void* _fils;
    double (*perimetre) (void*);
    void (*afficher) (void*);
} Forme_struct;
```



Clonage d'un Objet

- Créer un objet *identique* à un autre

Comparaison Objet :

- Comparer 2 objets pour savoir si il sont *identique*

Assignation Objet

- Copier l'état d'un objet dans un autre afin qu'il soit *identique*



3 concepts pour faire un langage objet :

- **Encapsulation** : combiner des données et un comportement dans un emballage unique,
- **Héritage** : chiens et chat sont des mammifères, ils héritent du comportement du mammifère.
- **Polymorphisme** : Cercle et rectangle sont des formes géométriques, chacun doit calculer sa surface.



Cycle de vie itératif

- **Spécification**
 - étude du contexte : objets du domaine, objets applicatifs
- **Analyse**
 - analyse du domaine : classes du domaine
 - analyse applicative : classes applicatives
- **Conception**
 - conception préliminaire : classes d'interface
 - conception objet : classes techniques
- **Implémentation**
 - codage : classes techniques spécifiques au langage utilisé

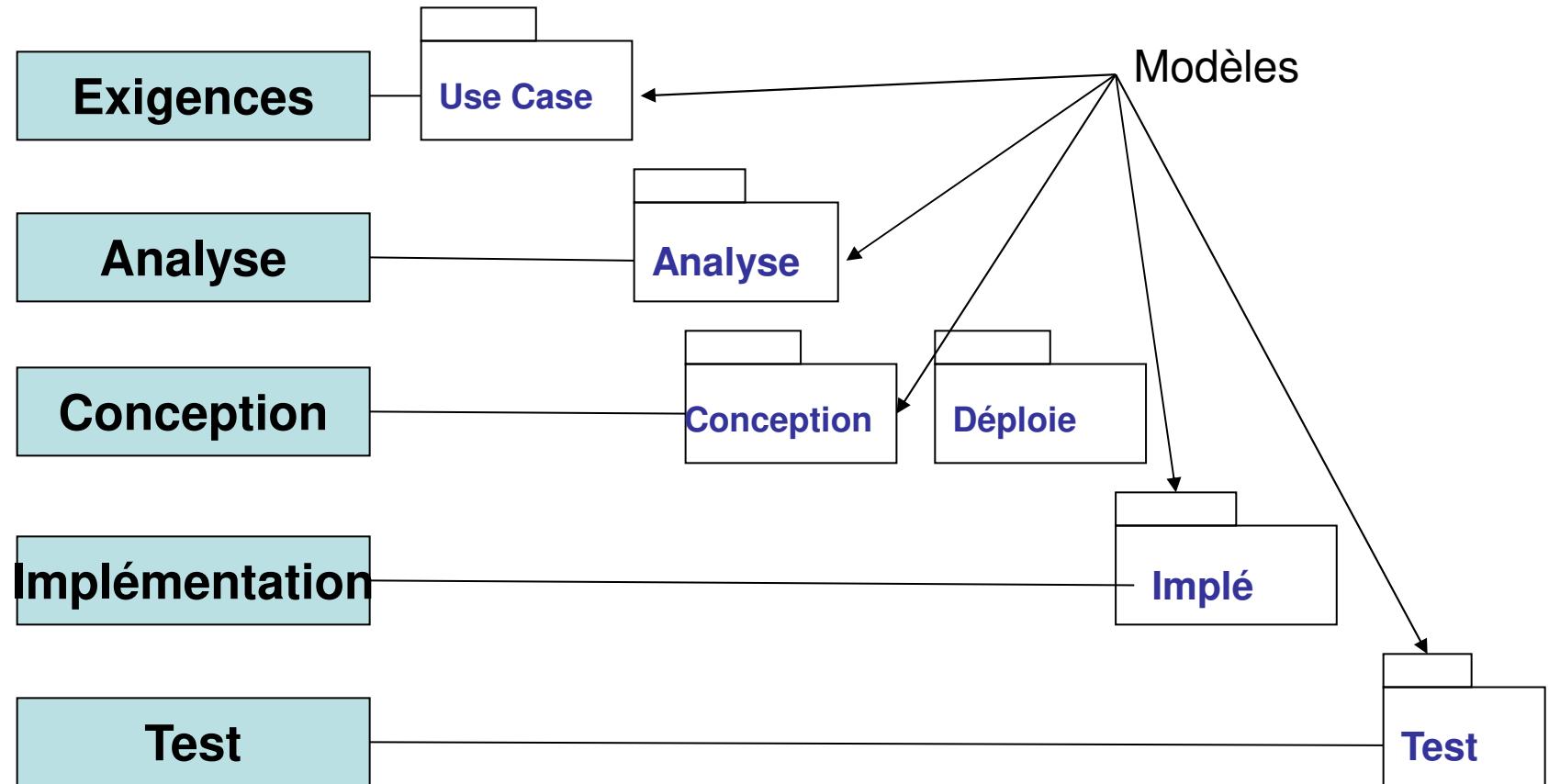


Approche itérative et incrémentale

- **Développement par approches successives**
 - **1er cycle** : prototype de base
 - **2ème cycle** : amélioration du prototype de base, intégration de nouvelles fonctionnalités
 - **3ème ...** : continuer jusqu'à épuisement des fonctionnalités
- **Étapes**
 - **définition du domaine** : design général
 - **définition des sous-systèmes** : design du sous-système
 - **remonter les problèmes dans le design général**
 - **réitérer pour chaque sous-système.**

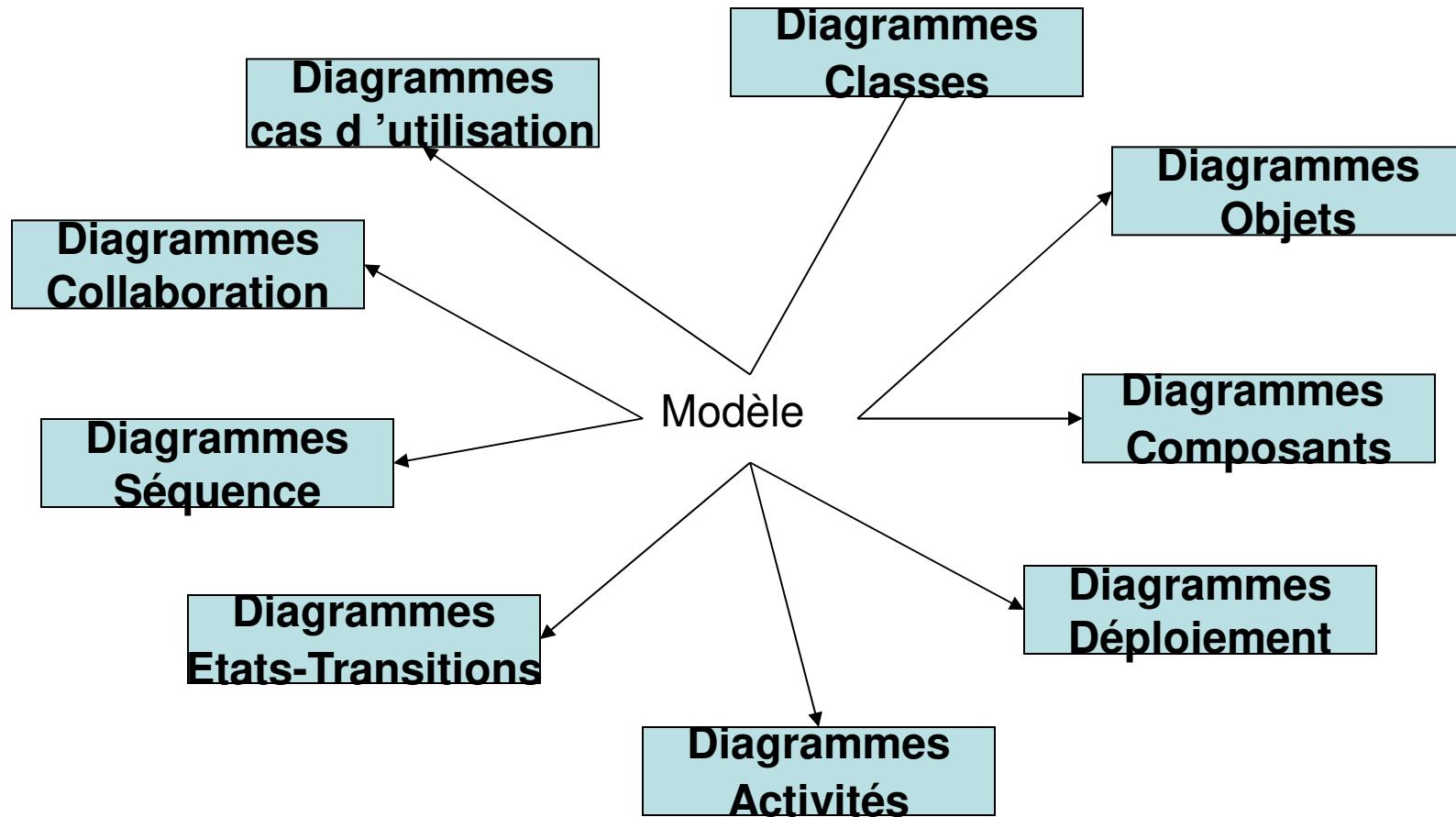


Modélisation UML



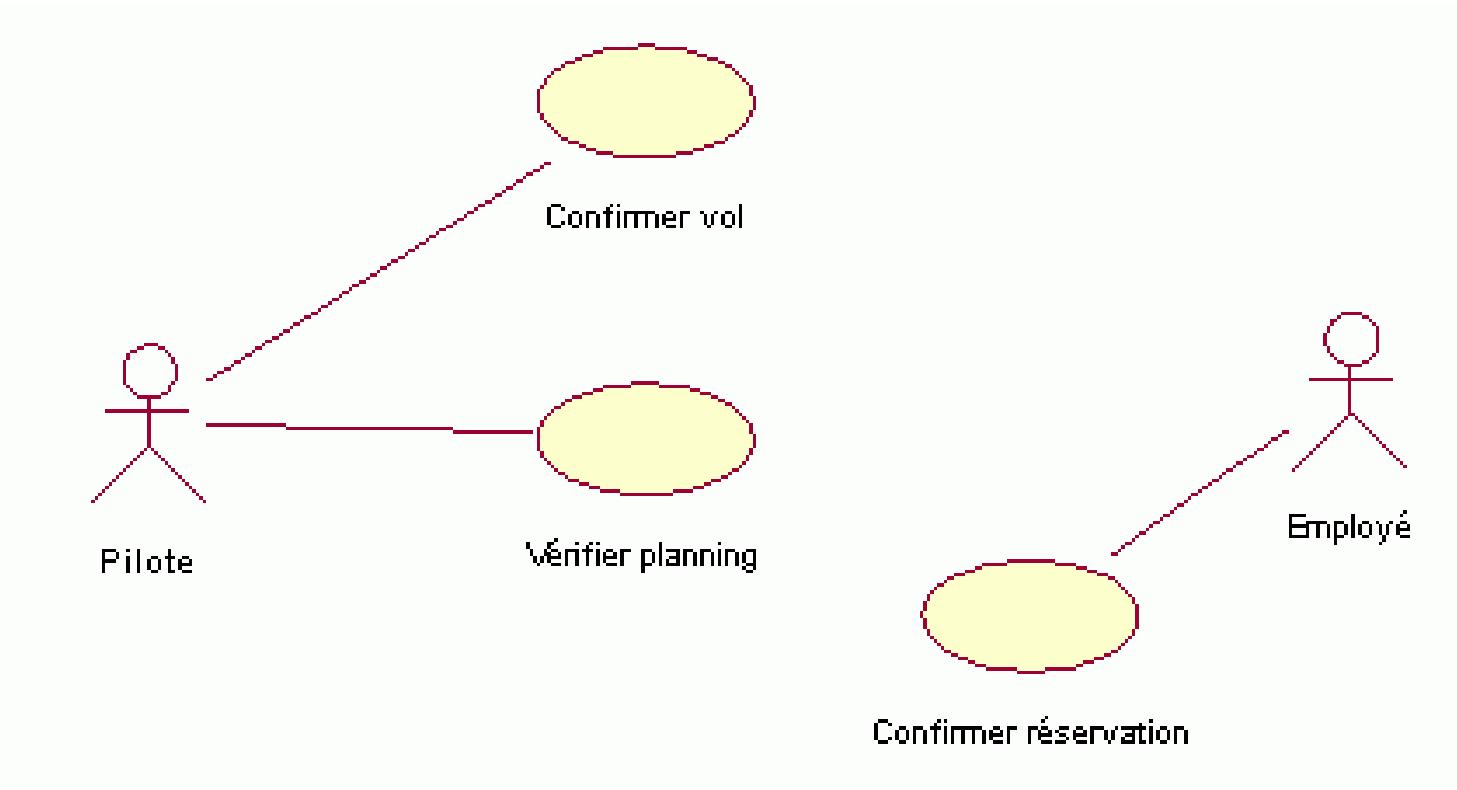


Modélisation UML





Diagrammes de cas d 'utilisation : Use Case





Avantages du style objet

- **Facilite la programmation modulaire :**
 - composants réutilisables,
 - un composant offre des services et en utilise d'autres,
 - il expose ses services au travers d'une interface
- **Facilite l'abstraction :**
 - elle sépare le définition de son implémentation,
 - elle extrait un modèle commun à plusieurs composants,
 - le modèle commun est partagé par le mécanisme d'héritage.
- **Facilite la spécialisation :**
 - elle traite des cas particuliers,
 - le mécanisme de dérivation rend les cas particuliers transparents.



- 3.1 - Introduction aux design Patterns
- 3.2 - Exemples simples intuitifs (non officiels)
- 3.3 - Les patterns ?
- 3.4 - Comment ?
- 3.5 - Description des modèles de conception
- 3.6 - Classification
- 3.7 - Avantage
- 3.8 - Inconvénient
- 3.9 - Exemples



Un **Design Pattern** est un **motif ou patron** de Conception dont le rôle est d'implémenter un **mécanisme générique** répondant à une problématique **récurrente** sur un projet.

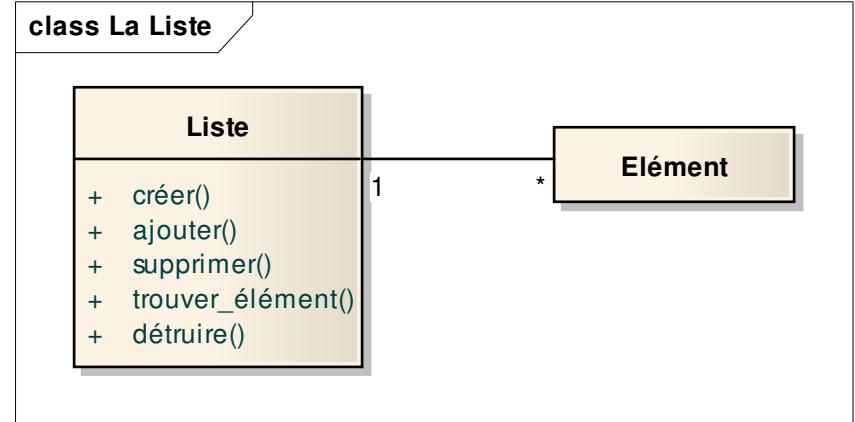
Les patrons de conception les plus répandus sont au nombre de 23.

Ils sont couramment appelés « patrons GoF » « Gang of Four » : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (1995).

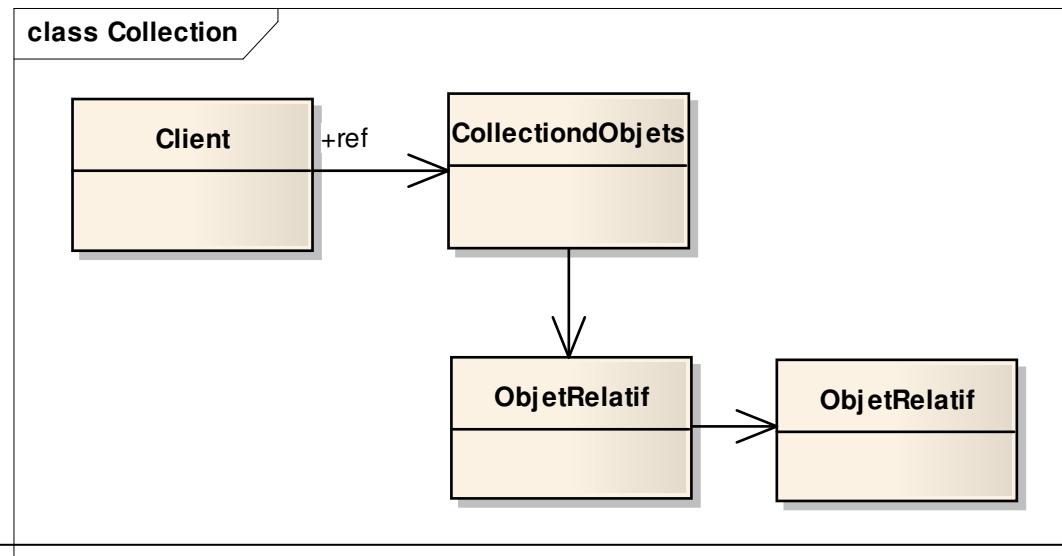
3.2 – Exemples simples intuitifs (non officiels)



La Liste : des défauts ?



Gestion de collections via une classe : UML ? des défauts ?





Design Pattern

=

Schéma de conception réutilisable

=

Organisation et hiérarchie de *plusieurs* modèles de classes réutilisables par simple implémentation, adaptation et extension

3.4 – Comment ?



- Les Design Patterns sont présentés en utilisant la notation graphique standard UML.
- En général, seule la partie statique (diagrammes de classe) de UML est utilisée.
- La description d'un patron de conception suivant un formalisme fixe :
 - Nom
 - Description du problème à résoudre
 - Description de la solution : les éléments de la solution, avec leurs relations. La solution est appelée **patron de conception**.
 - Conséquences : résultats issus de la solution
 - ...

3.5 - Description des modèles de conception



- Nom et classification
- Implémentation
- Intention
- Exemple de code
- Autres noms connus
- Usages connus
- Motivation (scénario)
- Formes associées
- Applicabilité
- Structure
- Participants (classes...)
- Collaborations
- Conséquences



- **Modèles Créateurs ou Patterns de construction**

Création d'objets sans instantiation directe d'une classe

- **Modèles structuraux**

Composition de groupes d'objets

- **Modèles de comportement**

Modélisation des communications inter-objets et du flot de données

3.6 - Classification



		Modèles		
		Créateur	Structure	Comportement
Classe	Factory Method	Adapter	Interpreter Template Method	
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor	



- L'utilisation des Design Patterns offre de nombreux avantages. Tout d'abord cela permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts. Ainsi on gagne en rapidité de conception (on diminue également les coûts) et en qualité.
- De plus, les Design Patterns sont réutilisables et permettent de mettre en avant les bonnes pratiques de conception.
- Les Design Patterns étant largement documentés et connus d'un grand nombre de développeurs, ils permettent également de faciliter la communication. Si un développeur annonce que sur ce point du projet il va utiliser le Design Pattern Observateur il est compris des informaticiens sans pour autant rentrer dans les détails de la conception (diagramme UML, objectif visé...).



- Truffer le code de design pattern n'est pas forcément bon pour la compréhension. Il faut le plus possible se poser la question de l'intérêt d'utiliser un design pattern.
- Trouver le juste dosage entre l'intérêt de l'utilisation d'un pattern et les inconvénients inhérents à son utilisation.



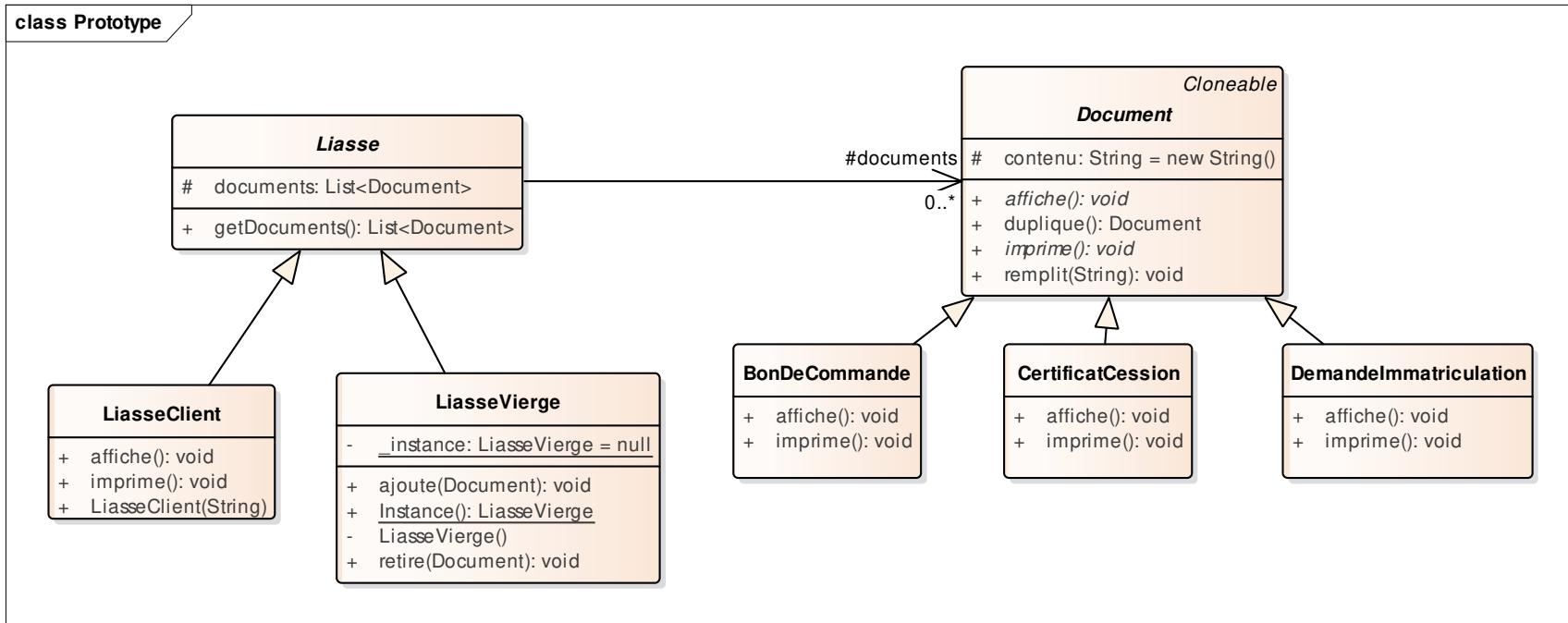
- ### Description

Assurer qu'une classe ne possède qu'une instance et fournir une méthode de classe unique retournant cette instance.

- ### Domaine d'utilisation

- ✓ Il ne doit y avoir qu'une seule instance de classe;
- ✓ Cette instance ne doit être accessible qu'au travers d'une méthode de classe.

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Prototype : Exemple

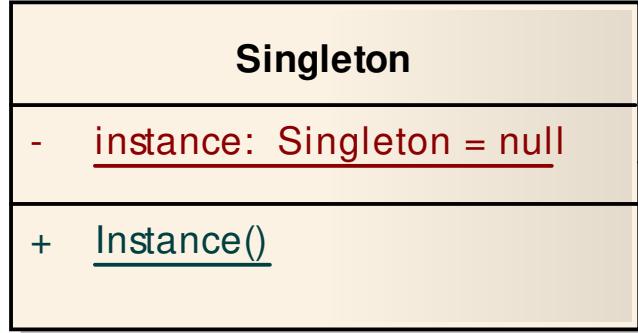


<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Singleton : Structure



```
class Singleton
```



```
if (instance == null)  
    instance = new Singleton();  
return instance;
```



- Participant

- Singleton :

- Offre l'accès à l'unique instance par sa méthode de classe Instance.
 - Possède un mécanisme qui bloque la création d'autres instances.

- Collaborations

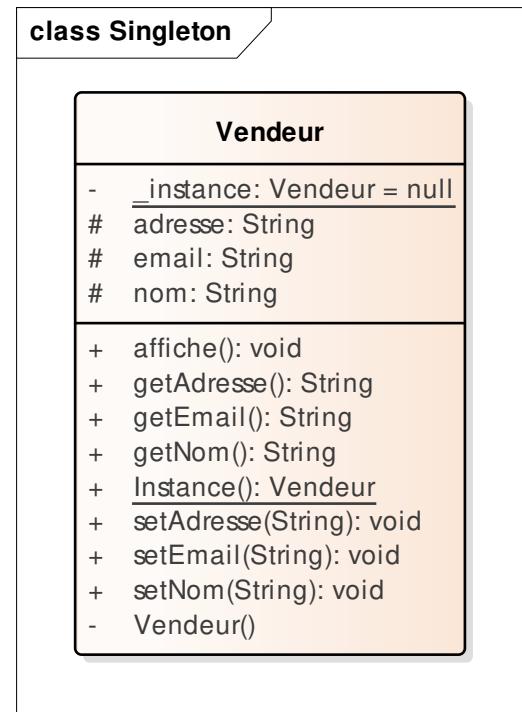
Chaque client accède à l'unique instance par la méthode de classe Instance. Le new est bloqué.



- ### Exemples

- Singleton permet de gérer la création à la demande d'un objet unique.
- L'objet est créé au premier accès de façon invisible.
- Spooler d'impressions.

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Singleton : Exemple



<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles Créateurs ou Patterns de construction

-> Pattern Singleton : Exemple Java wikipedia



```
public class Singleton
{
    private static Singleton INSTANCE = null;
    /**
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
    */
    private Singleton() {}

    /**
     * Le mot-clé synchronized sur la méthode de création
     * empêche toute instantiation multiple même par
     * différents threads.
     * Retourne l'instance du singleton.
    */
    public synchronized static Singleton getInstance()
    {
        if (INSTANCE == null)
            INSTANCE = new Singleton();
        return INSTANCE;
    }
}
```

```
public class Singleton
{
    /**
     * Crédit de l'instance au niveau de la variable
    */
    private static final Singleton INSTANCE =
        new Singleton();

    /**
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
    */
    private Singleton() {}

    /**
     * Dans ce cas présent, le mot-clé synchronized n'est pas utile.
     * L'unique instantiation du singleton se fait avant
     * l'appel de la méthode getInstance(). Donc aucun risque *
     * d'accès concurrents.
     * Retourne l'instance du singleton.
    */
    public static Singleton getInstance()
    {
        return INSTANCE;
    }
}
```

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Singleton : Exemple C++ wikipedia



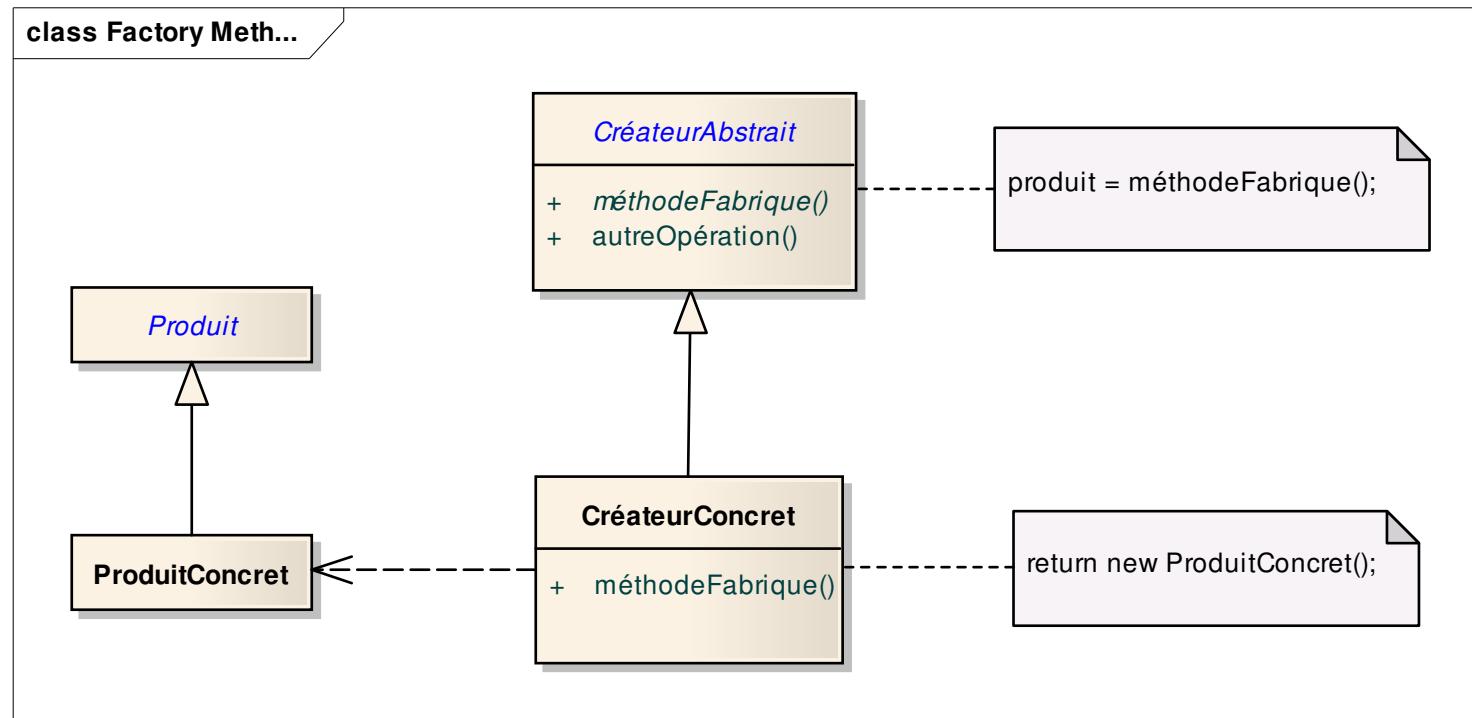
```
template<typename T> class Singleton
{
public:
    static T& Instance()
    {
        static T theSingleton;
        // suppose que T a un constructeur par défaut
        return theSingleton;
    }
};

class OnlyOne : public Singleton<OnlyOne>
{
    // constructeurs/destructeur de OnlyOne accessibles au Singleton friend class Singleton<OnlyOne>;
    // ...définir ici le reste de l'interface
};
```



- Description
 - Introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective.
- Domaine d'utilisation
 - Une classe ne connaît que les classes abstraites des objets avec lesquels elle possède des relations.
 - Une classe veut transmettre à ses sous-classes les choix d'instanciation en profitant du mécanisme de polymorphisme.

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Factory Method : Structure





• Participants

- **CréateurAbstrait** : classe abstraite qui introduit la méthode de fabrique et l'implantation de méthodes qui invoquent ma méthode de fabrique.
- **CréateurConcret** : classe concrète implémentant la méthode de fabrique. Il peuvent être plusieurs.
- **Produit** : classe abstraite définissant les propriétés communes des produits.
- **ProduitConcret** : classe concrète décrivant un produit.

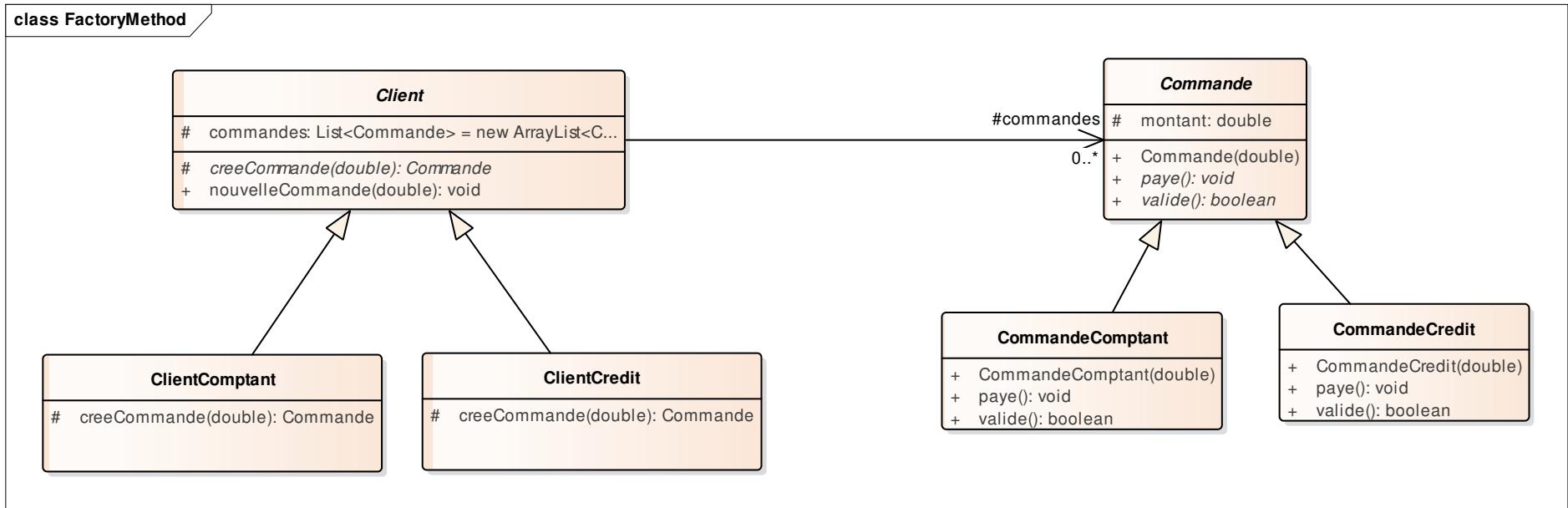
• Collaborations

Les méthodes concrètes de la classe **CréateurAbstrait** se basent sur l'implantation de la méthode de fabrique dans les sous-classes. Cette implantation crée une instance de la sous-classe adéquate de **Produit**.



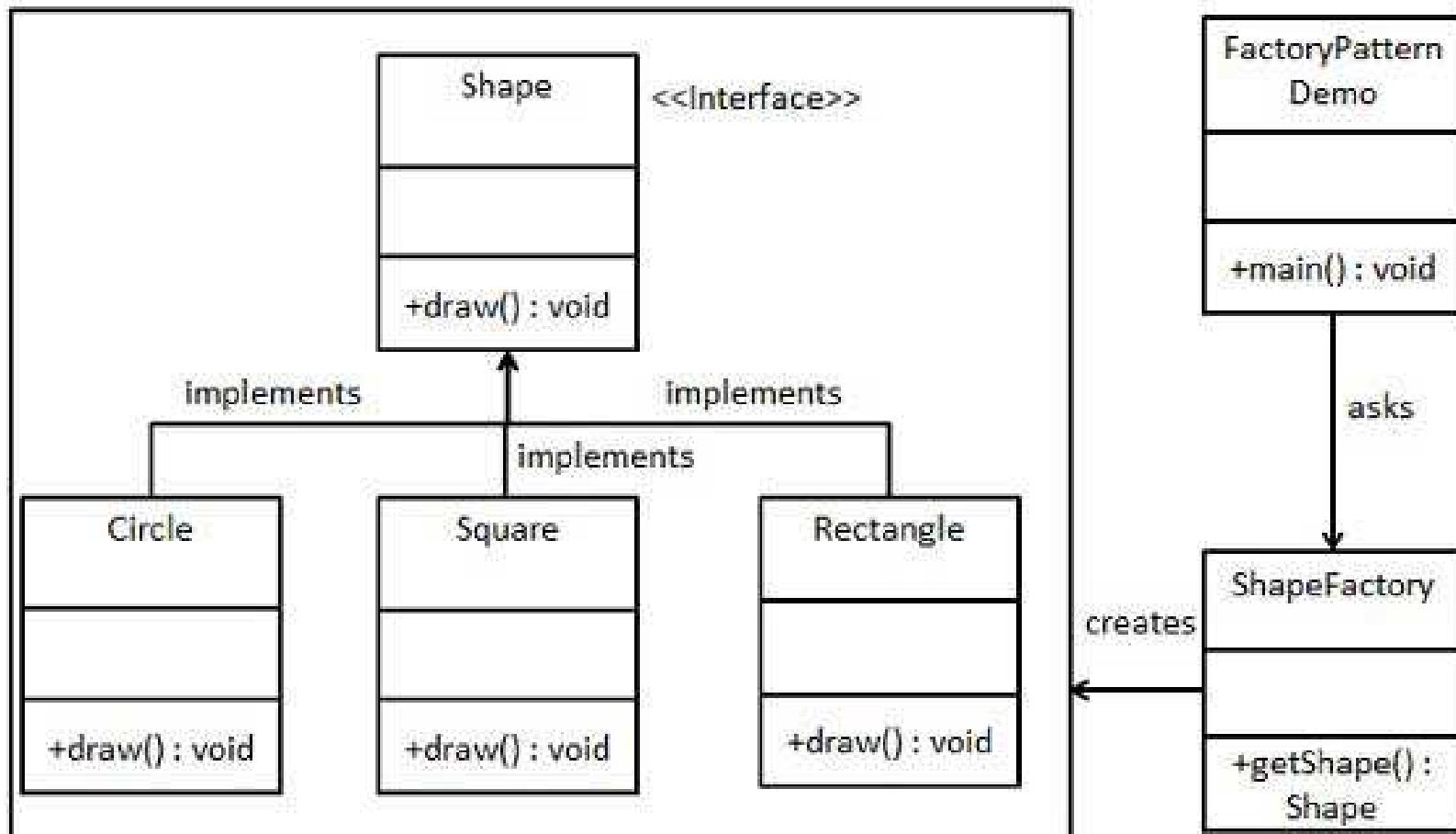
- Exemples
 - Fragment de Abstract Factory
 - Crédit d'itérateurs. Chaque classe d'une hiérarchie de containers implante sa version de `createIterator()`, qui retourne un objet du type adéquat.

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Factory Method : Exemple



<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Factory Method : Exemple



https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

3.9 – Exemple : Modèles Créateurs ou Patterns de construction -> Pattern Factory Method : ExempleJava wikipedia



```
public class FabriqueAnimal
{
    Animal getAnimal(String typeAnimal)
        throws ExceptionCreation
    {
        if (typeAnimal.equals("chat"))
            { return new Chat(); }
        else if (typeAnimal.equals("chien"))
            { return new Chien(); }
        throw new ExceptionCreation("Impossible de créer un " +typeAnimal);
    }
}
```



• Description

Séparer l'aspect d'implantation d'un objet de son aspect de représentation et d'interface.

L'implémentation peut être encapsulée.

Implémentation et représentation peuvent évoluer indépendamment.

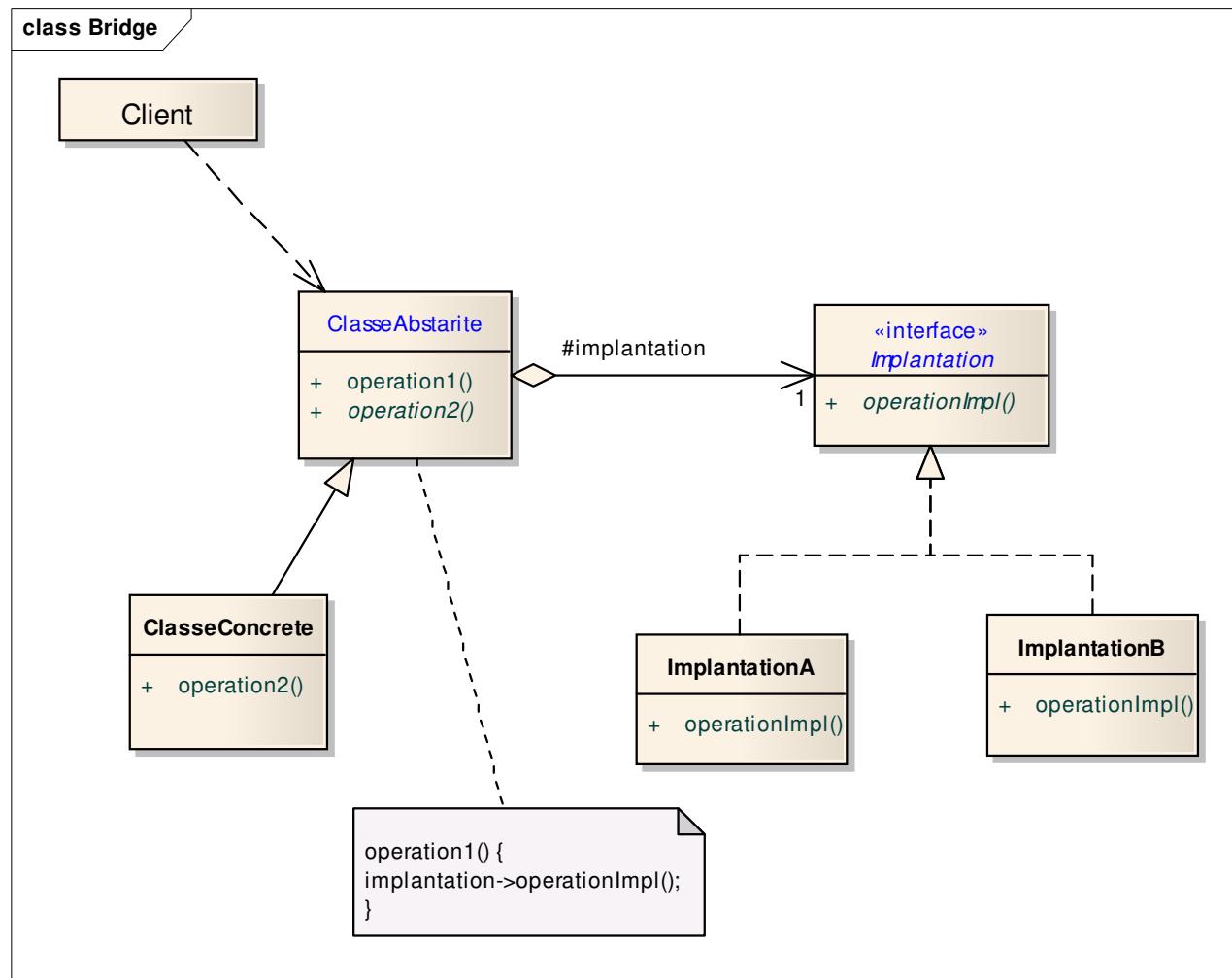
• Domaine d'utilisation

- ✓ Eviter une liaison forte entre la représentation des objets et leur implantation notamment
- ✓ Pour que le changement d'implantation n'est pas d'impact dans les interactions entre les objets et leurs clients
- ✓ Pour conserver la capacité d'extension par la création de nouvelles sous-classes.

3.9 – Exemple : Modèles structuraux -> Pattern Bridge



- Structure





- Participants

- ClasseAbstraite : des objets du domaine qui contient une référence sur l'interface Implantation.
- ClasseConcrète : classes concrètes implémentant les méthodes de ClasseAbstraite.
- Implantation : interface des classes d'implantation.
- ImplantationAbstraitA, ImplantationAbstraitB : classes concrètes qui réalisent les méthodes d'Implantation.

- Collaborations

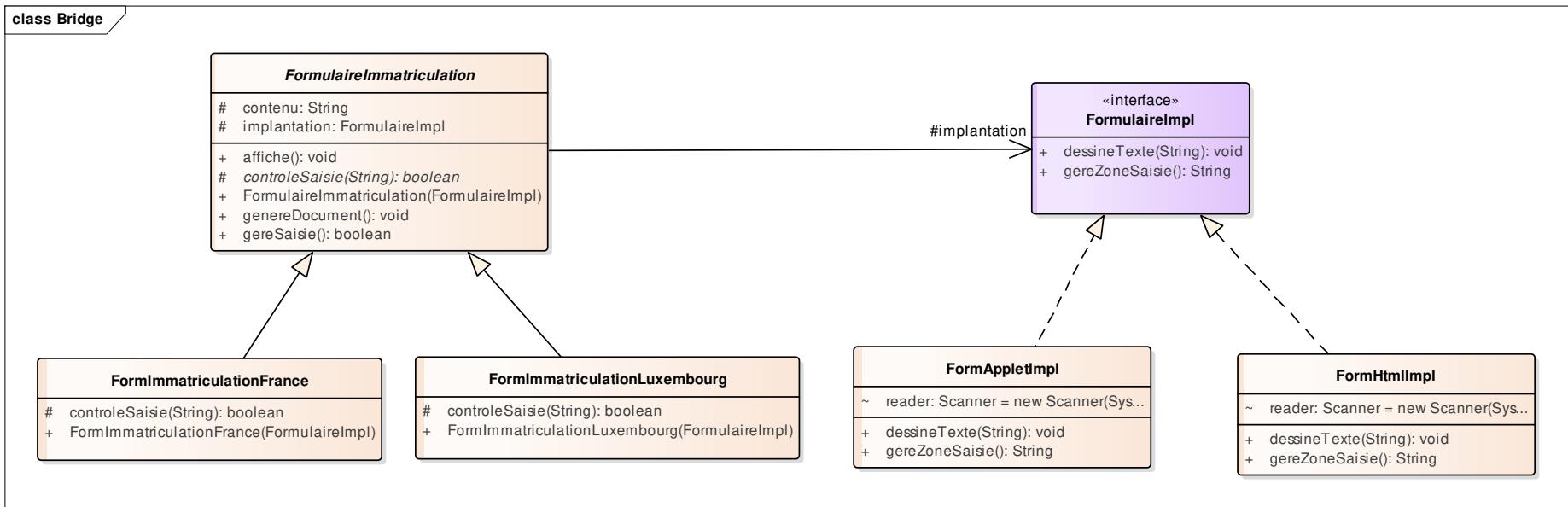
Les opérations de ClasseAbstraite et des sous-classes invoquent les méthodes introduites dans l'interface Implantation.



- ### Exemples

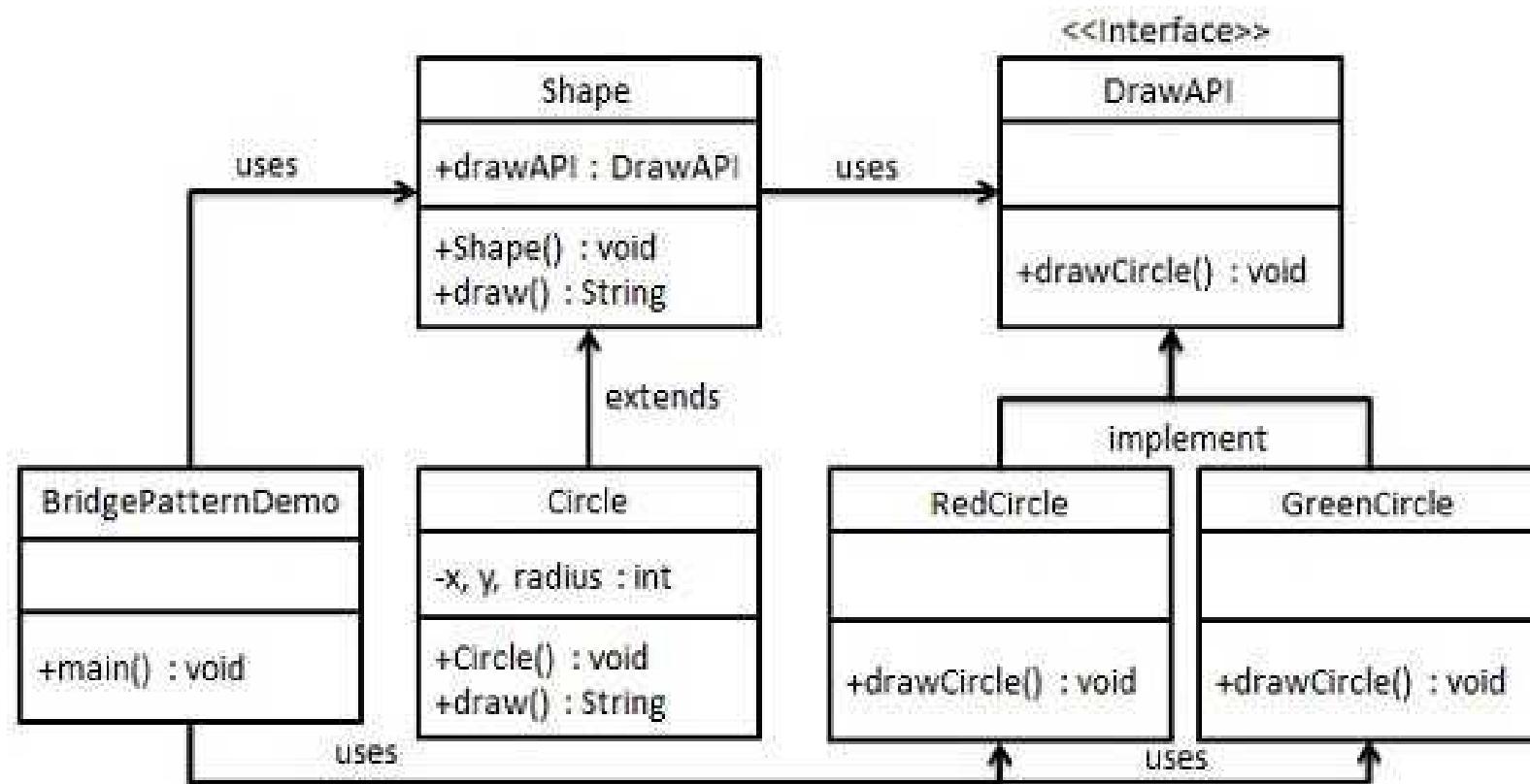
- On doit maintenir une hiérarchie d'abstraction avec plusieurs hiérarchies d'implantation différentes : interfaces graphiques portables.
- Cacher des interfaces de programmation : type de la classe Handle.

3.9 – Exemple : Modèles structuraux -> Pattern Bridge : Exemple



<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles structuraux -> Pattern Bridge : Exemple



https://www.tutorialspoint.com/design_pattern/bridge_pattern.htm



• Description

Cadre de conception d'une composition d'objets, de profondeur variable et basée sur un arbre.

La composition est encapsulée vis-à-vis des clients des objets.

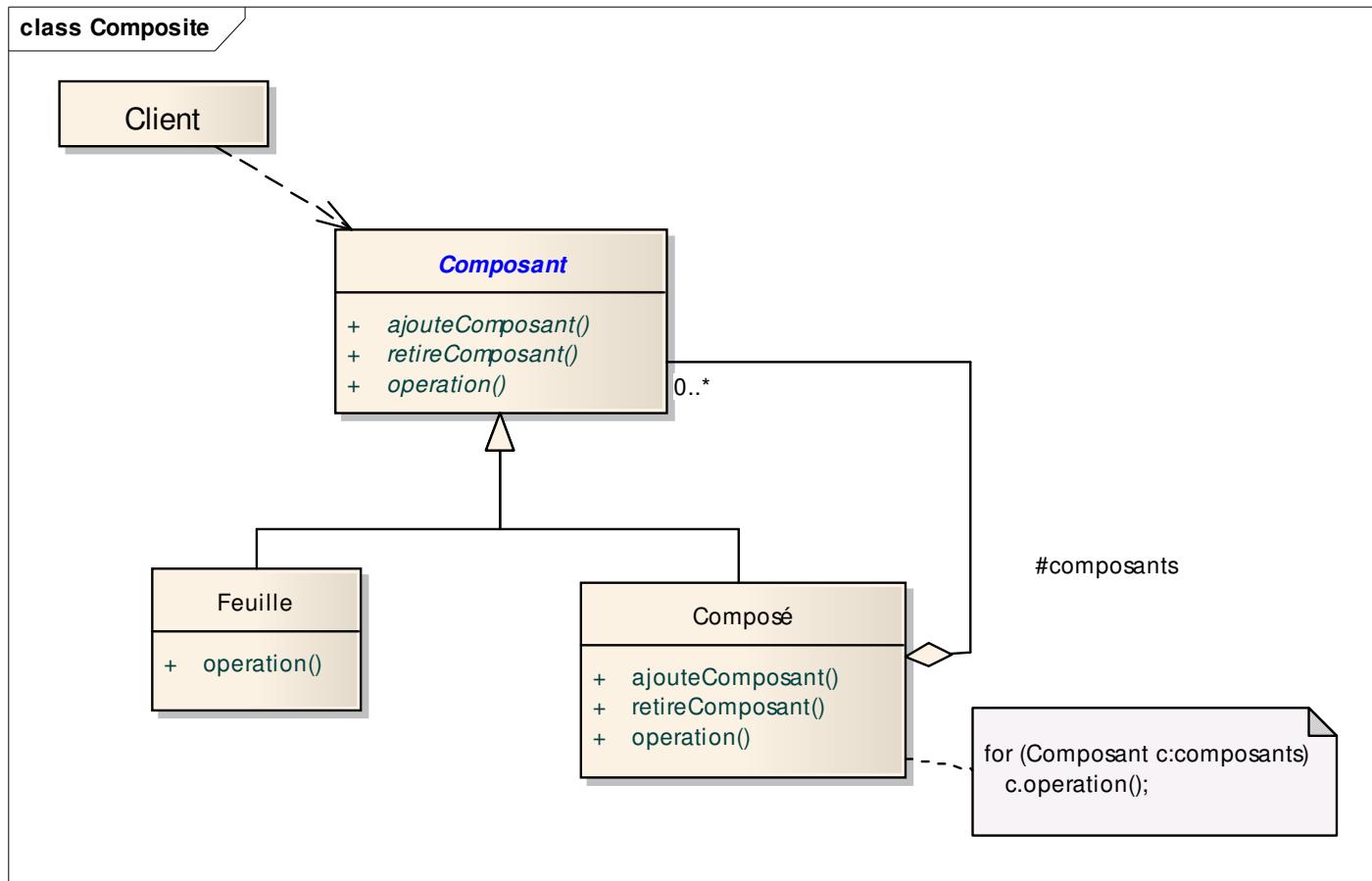
• Domaine d'utilisation

- ✓ Les clients envoient leurs requêtes aux composants au travers d'une interface
- ✓ Lorsqu'un composant reçoit une requête, il réagit en fonction de sa classe.

3.9 – Exemple : Modèles structuraux -> Pattern Composite



- Structure





- Participants

- Composant : classe abstraite qui introduit l'interface des objets de la composition. Implante les méthodes communes et introduit la signature des méthodes qui gèrent la composition en ajoutant ou supprimant des composants.
- Feuille : classe concrète qui décrit les feuilles de la composition.
- Composé : classe concrète qui décrit les objets composés de la hiérarchie. Possède une association d'agrégation avec la classe Composant.
- Client : classes des objets qui accèdent aux objets de la composition.

- Collaborations

Les clients envoient leurs requêtes aux composants au travers de l'interface de la classe Composant.

Lorsqu'un composant reçoit une requête, il réagit en fonction de sa classe.

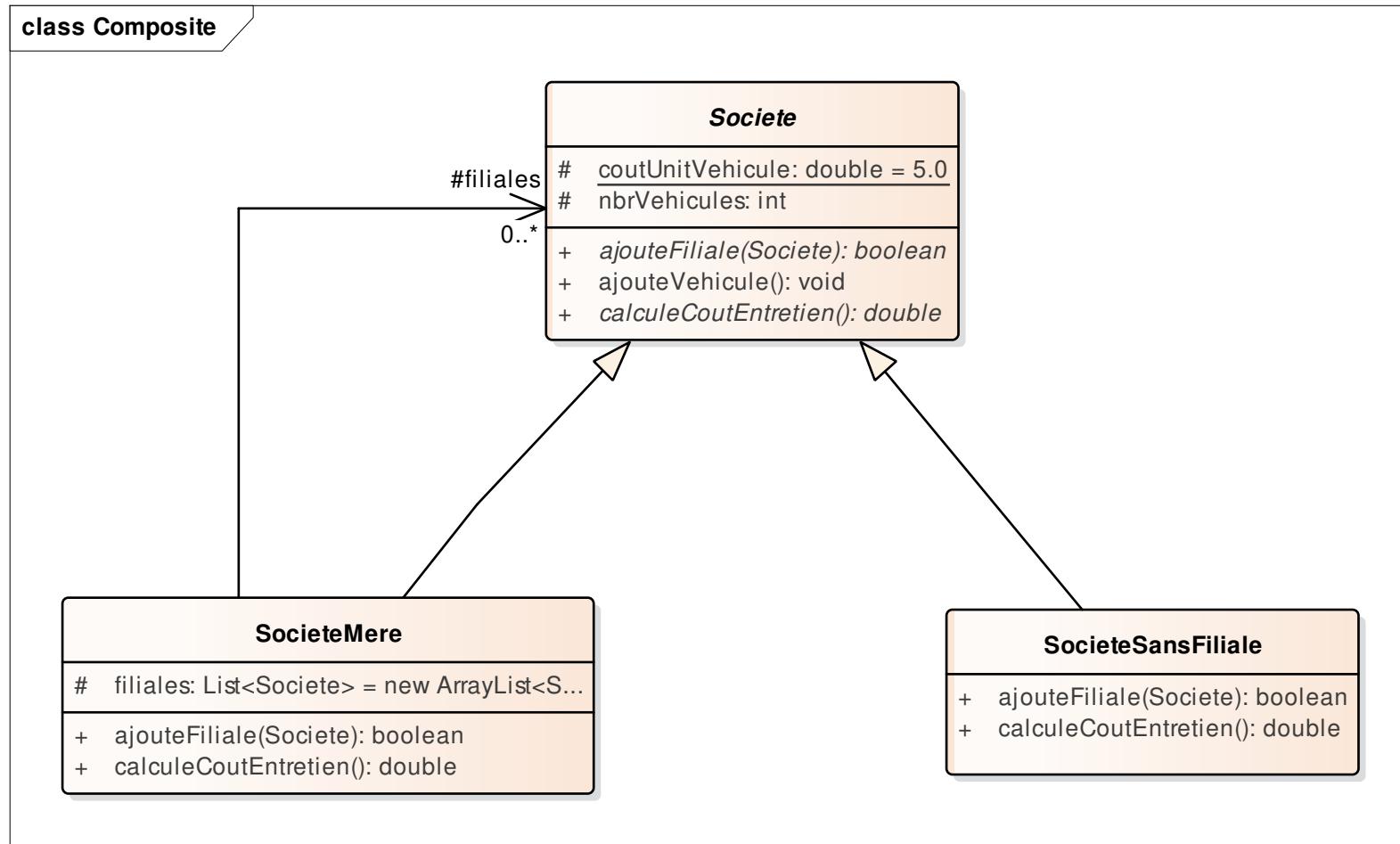


- ### Exemples

Toute structure de données récursives :

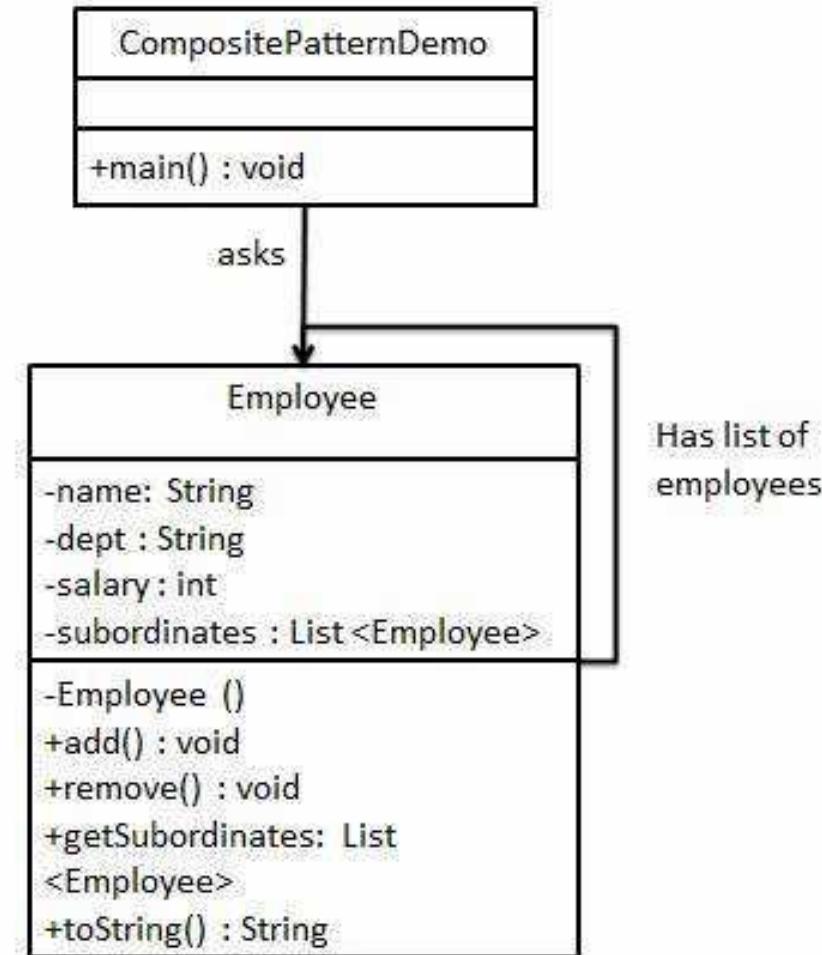
- Container graphique,
- Structure de document
(chapitre/section/paragraphe ...),
- Container conceptuel.

3.9 – Exemple : Modèles structuraux -> Pattern Composite : Exemple



<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles structuraux -> Pattern Composite : Exemple



https://www.tutorialspoint.com/design_pattern/builder_pattern.htm



3.9 – Exemple : Modèles structuraux
-> Pattern Composite : Exemple C++ wikipedia



```
//Cet exemple represente la hiérarchie d'un système de fichiers : fichiers/répertoires
class Composant {
    //L'objet abstrait qui sera 'composé' dans le composite
    //Dans notre exemple, il s'agira d'un fichier ou d'un répertoire
public:
    //Parcours récursif de l'arbre
    //(Utilisez plutôt le Design Pattern "Visiteur" à la place de cette fonction)
    virtual void ls_r() = 0 ;
protected:
    std::string name;
};

class File : public Composant { //Leaf
public:
    void ls_r() { std::cout << name << std::endl; }
};

class Repertoire : public Composant { //Composite
    //Le repertoire est aussi un 'composant' car il peut être sous-répertoire
protected:
    std::list<Composant*> files; //List des composants
public: void ls_r() {
    std::cout << "[" << name << "]" << std::endl;
    for( std::list<Composant*>::iterator it = files.begin();
        it != files.end(); it ++ ) { it->ls_r(); }
}
};
```





3.9 – Exemple : Modèles structuraux -> Pattern Composite : Exemple Java wikipedia



```
import java.util.ArrayList;  
  
interface Graphic {  
    //Imprime le graphique.  
    public void print(); }  
  
class CompositeGraphic implements Graphic {  
    //Collection de graphiques enfants.  
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();  
    //Imprime le graphique.  
    public void print() {  
        for (Graphic graphic : mChildGraphics) {  
            graphic.print(); } }  
    //Ajoute le graphique à la composition composition.  
    public void add(Graphic graphic) { mChildGraphics.add(graphic); }  
    //Retire le graphique de la composition.  
    public void remove(Graphic graphic) { mChildGraphics.remove(graphic); } }  
  
class Ellipse implements Graphic {  
    //Imprime le graphique.  
    public void print() {  
        System.out.println("Ellipse"); } }
```

```
public class Program {  
    public static void main(String[] args) {  
        //Initialise quatre ellipses  
        Ellipse ellipse1 = new Ellipse();  
        Ellipse ellipse2 = new Ellipse();  
        Ellipse ellipse3 = new Ellipse();  
        Ellipse ellipse4 = new Ellipse();  
        //Initialise three graphiques composites  
        CompositeGraphic graphic = new CompositeGraphic();  
        CompositeGraphic graphic1 = new CompositeGraphic();  
        CompositeGraphic graphic2 = new CompositeGraphic();  
        //Composes les graphiques  
        graphic1.add(ellipse1);  
        graphic1.add(ellipse2);  
        graphic1.add(ellipse3);  
        graphic1.add(ellipse4);  
        graphic.add(graphic1);  
        graphic.add(graphic2);  
        //Imprime le graphique complet (quatre fois la chaîne "Ellipse").  
        graphic.print(); } }
```





• Description

Regrouper les interfaces d'un ensemble d'objets en une interface unifiée pour rendre cet ensemble plus facile à utiliser pour un client.

Encapsule l'interface de chaque objet considérée comme interface de bas niveau dans une interface unique de niveau plus élevé.

Cela peut nécessiter d'implanter des méthodes destinées à composer des interfaces de bas niveau..

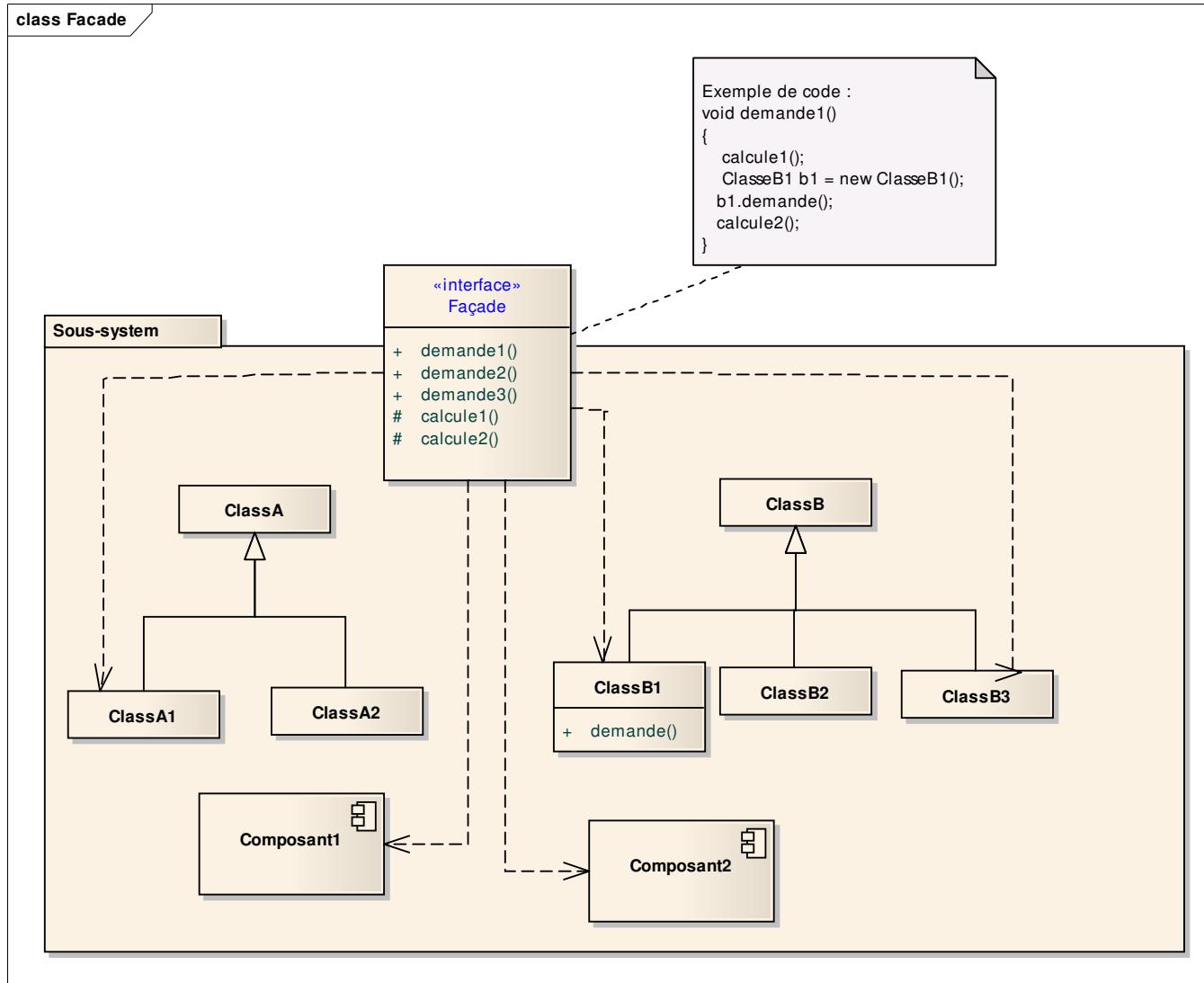
• Domaine d'utilisation

- ✓ Fournir une interface simple d'un système complexe.
- ✓ Diviser un système en sous-systèmes, la communication entre sous-systèmes étant mise en œuvre de façon abstraite de leur implantation grâce aux façades.
- ✓ Systématiser l'encapsulation de l'implantation d'un système vis-à-vis de l'extérieur.

3.9 – Exemple : Modèles structuraux -> Pattern Facade



- Structure





- Participants

- Façade : interface qui constitue la partie abstraite exposée aux clients. Elle possède des références vers les classes et composants du système. Les méthodes sont utilisées par la façade pour implanter l'interface unifiée.
- Les classes et composants du systèmes : implantent les fonctionnalités du système et répondent aux requêtes de la façade. Ils n'ont pas besoin de la façade pour travailler.

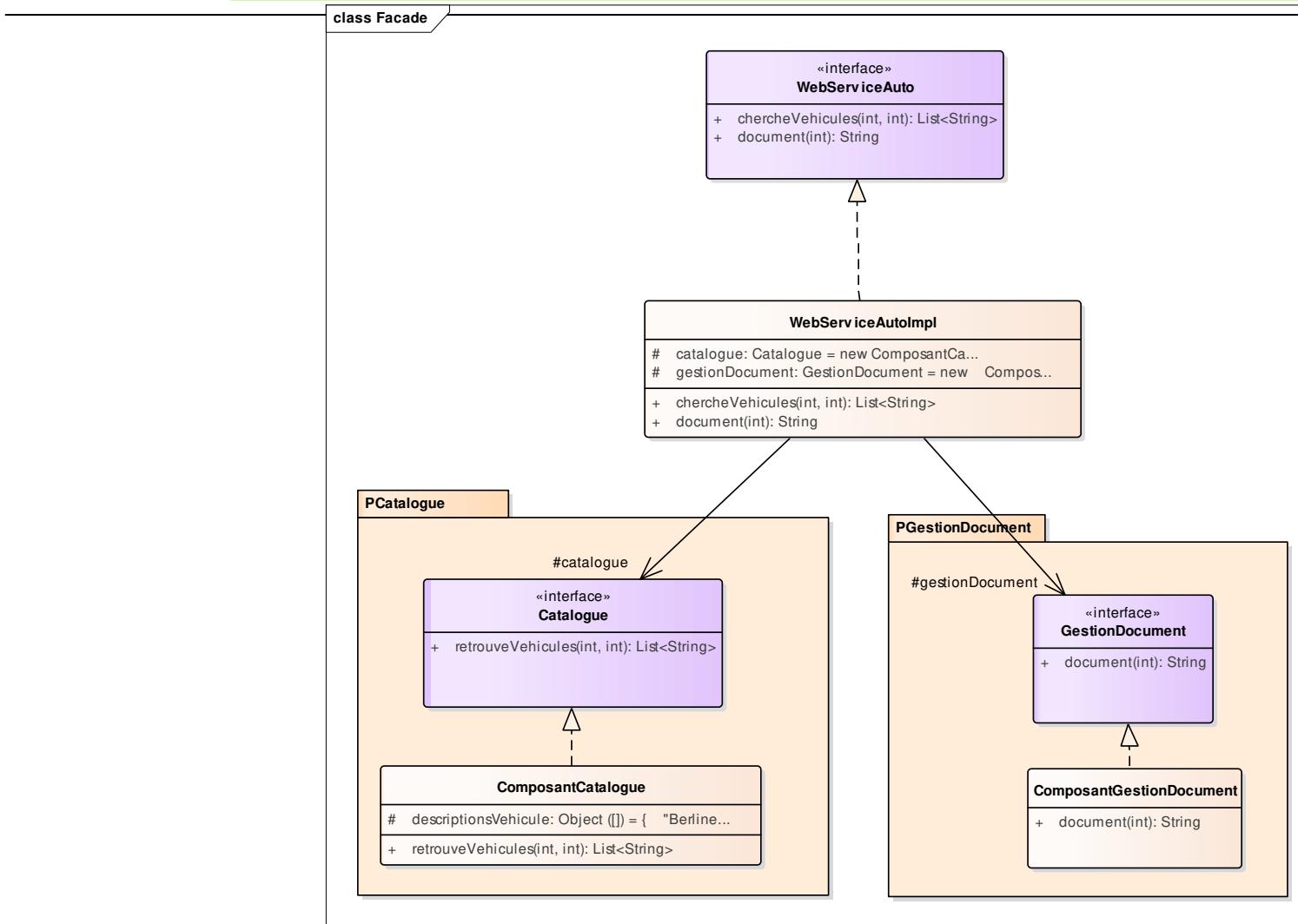
- Collaborations

Les clients communiquent avec le système au travers de la façade. Celle-ci se charge d'invoquer les classes et composants du système. La façade doit réaliser l'adaptation entre son interface et l'interface des objets du système au moyen de code spécifique.



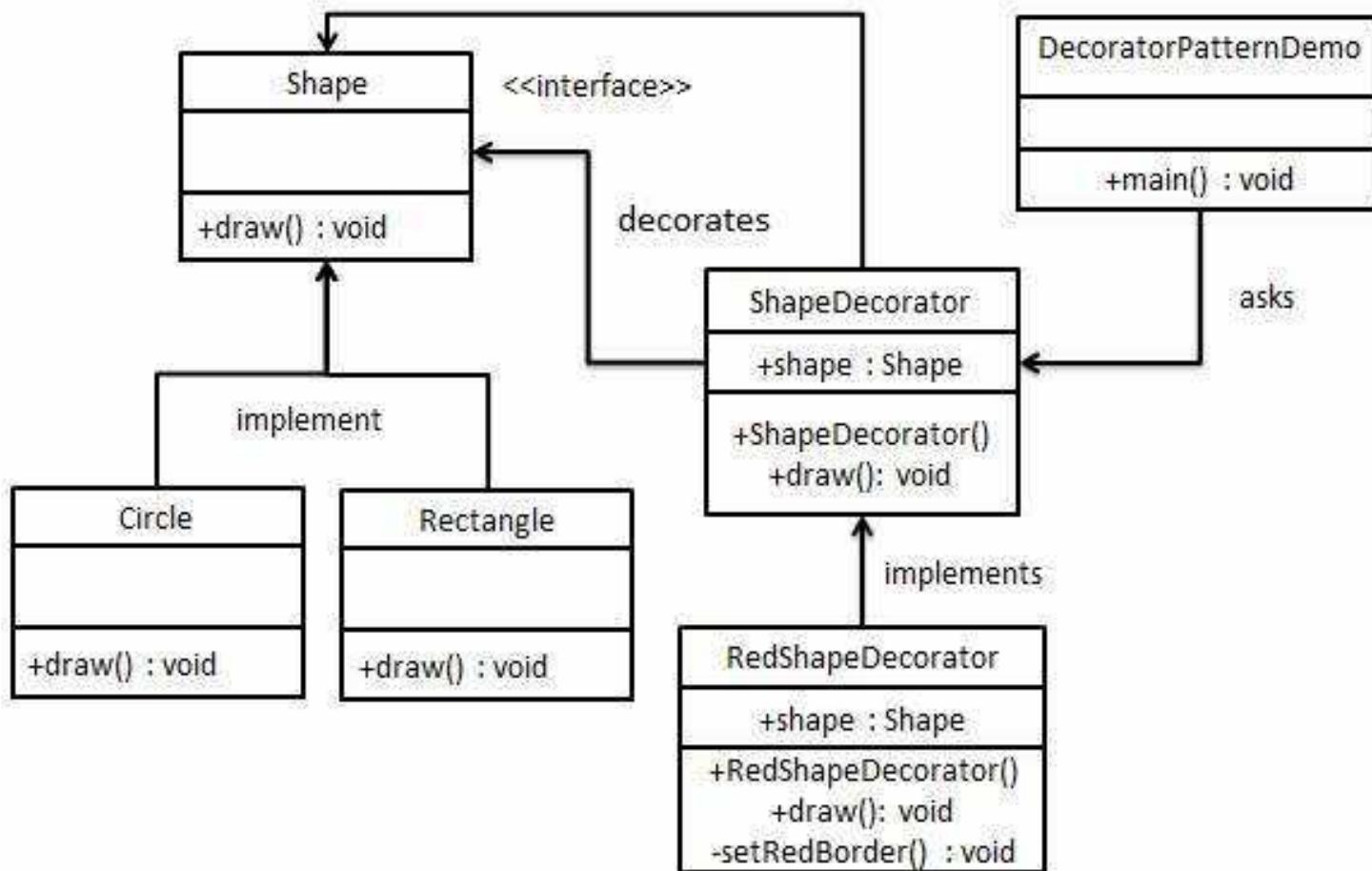
- Exemples
 - Un compilateur C++ en ligne de commande : permet l'invocation d'une session complète, ou du préprocesseur, ou du linker ...

3.9 – Exemple : Modèles structuraux -> Pattern Facade : Exemple



<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles structuraux -> Pattern Facade : Exemple



https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

3.9 – Exemple : Modèles structuraux

-> Pattern Facade : Exemple Java wikipedia



```
import java.util.*;
/* Façade */
class UserfriendlyDate {
    GregorianCalendar gcal;
    public UserfriendlyDate(String isodate_ymd) {
        String[] a = isodate_ymd.split("-");
        gcal = new GregorianCalendar(Integer.valueOf(a[0]).intValue(),
            Integer.valueOf(a[1]).intValue()-1 /* !!! */,
            Integer.valueOf(a[2]).intValue());
    }
    public void addDays(int days) {
        gcal.add(Calendar.DAY_OF_MONTH, days);
    }
    public String toString() {
        return new Formatter().format("%tY-%tm-%td",
            gcal).toString();
    }
}
```

```
/* Client */
class FacadePattern {
    public static void main(String[] args) {
        UserfriendlyDate d = new UserfriendlyDate("1980-08-20");
        System.out.println("Date : "+d);
        d.addDays(20);
        System.out.println("20 jours après : "+d);
    }
}
```



- **Description**

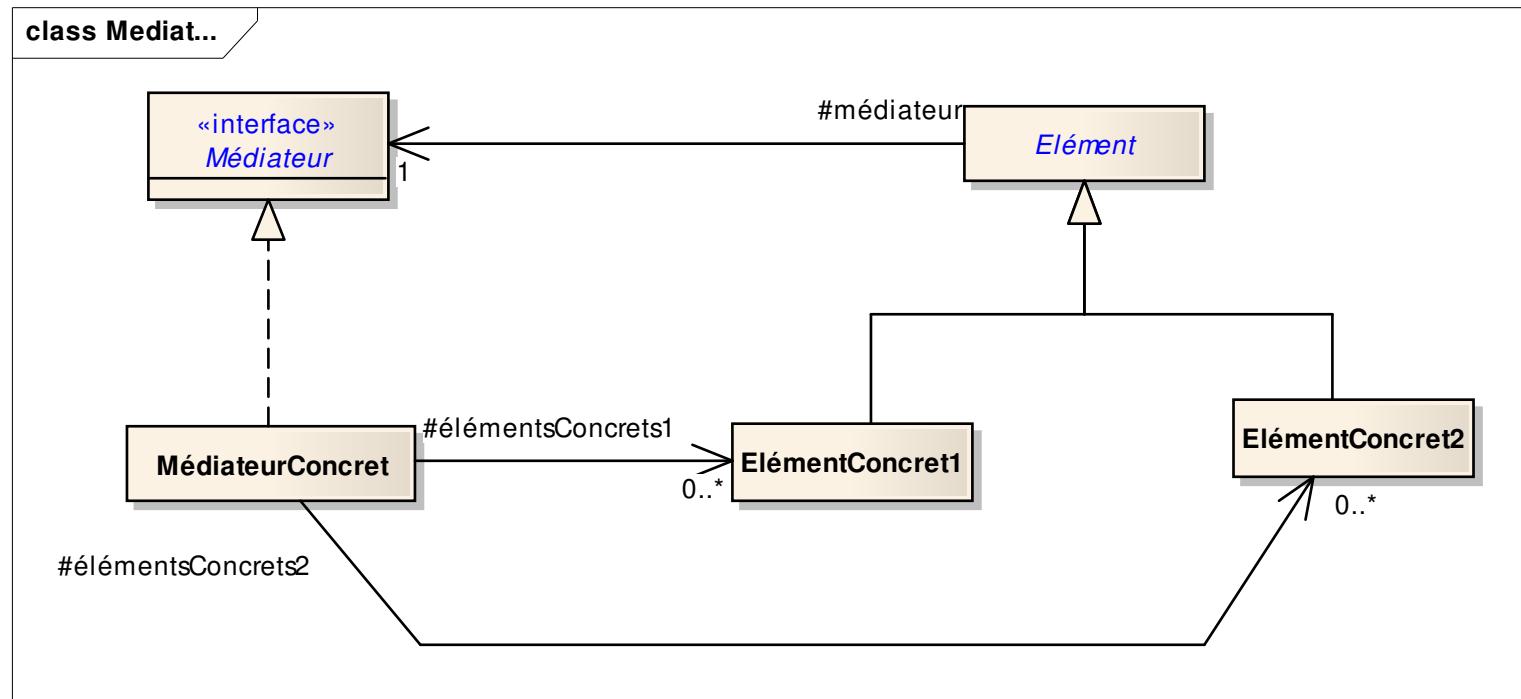
Construire un objet dont la vocation est la gestion et le contrôle des interactions dans un ensemble d'objets sans que les éléments doivent se connaître mutuellement.

- **Domaine d'utilisation**

- ✓ Système formé d'un ensemble d'objets basé sur une communication complexe conduisant à associer des objets entre eux.
- ✓ Système d'objets difficiles à réutiliser car ils possèdent de nombreuses associations entre eux.



- Structure





• Participants

- Médiateur : interface du médiateur pour les objets Elément.
- MédiateurConcret : implante la coordination entre les éléments et gère les associations avec les éléments.
- Elément : classe abstraite des éléments qui introduit leurs attributs, associations et méthodes communes.
- ElémentConcret1 et ElémentConcret2 : classes concrètes des éléments qui communiquent avec le médiateur au lieu de communiquer avec les autres éléments.

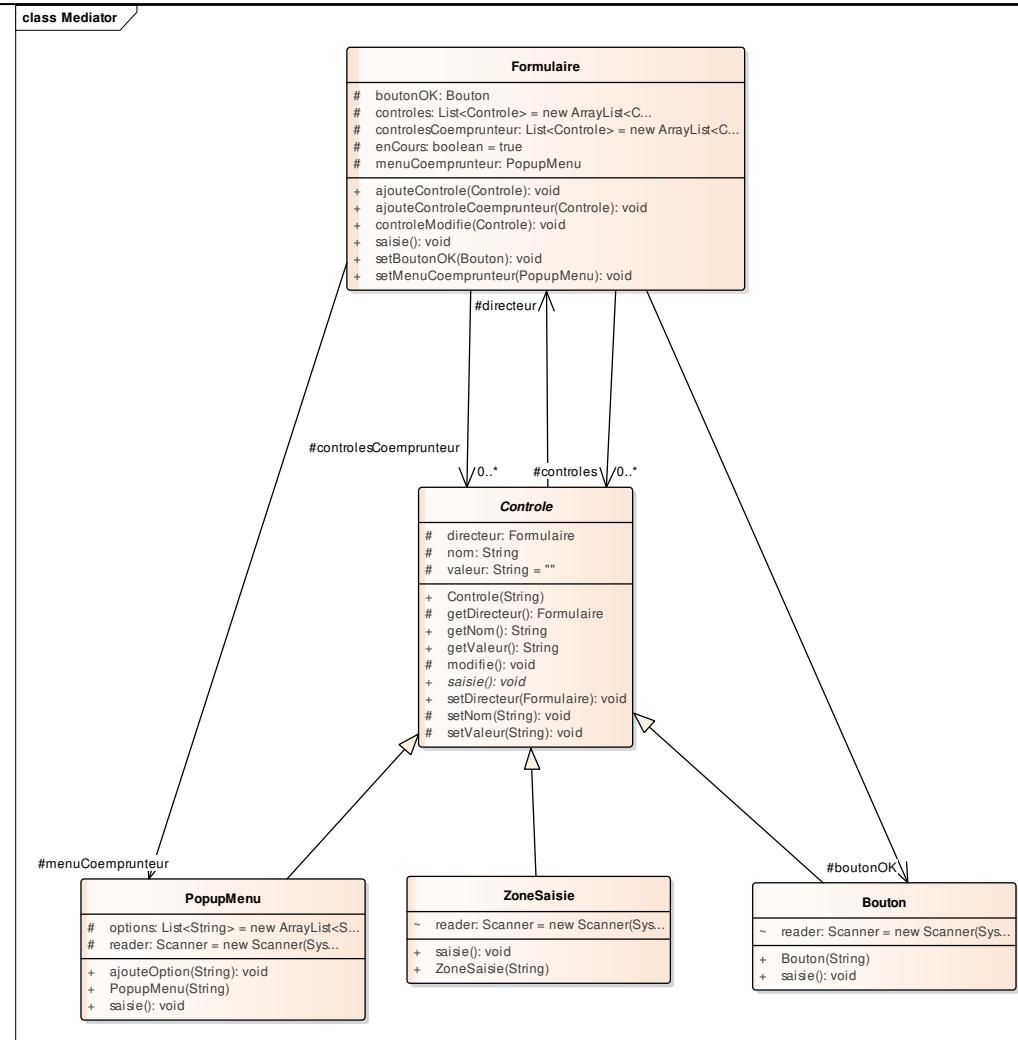
• Collaborations

Les éléments envoient des messages au médiateur et en reçoivent. Le médiateur implante la collaboration et la coordination entre les éléments.



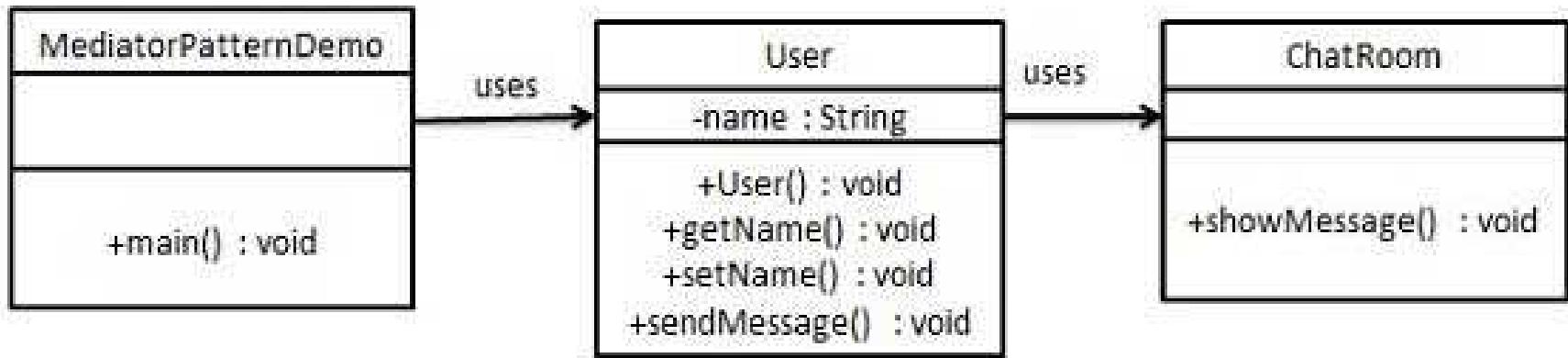
- Exemples
 - Dans une interface graphique gérer des dépendances complexes entre des composants de saisie/visualisation.

3.9 – Exemple : Modèles de comportement -> Pattern Mediator : Exemple



<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles de comportement -> Pattern Mediator : Exemple



https://www.tutorialspoint.com/design_pattern/builder_pattern.htm



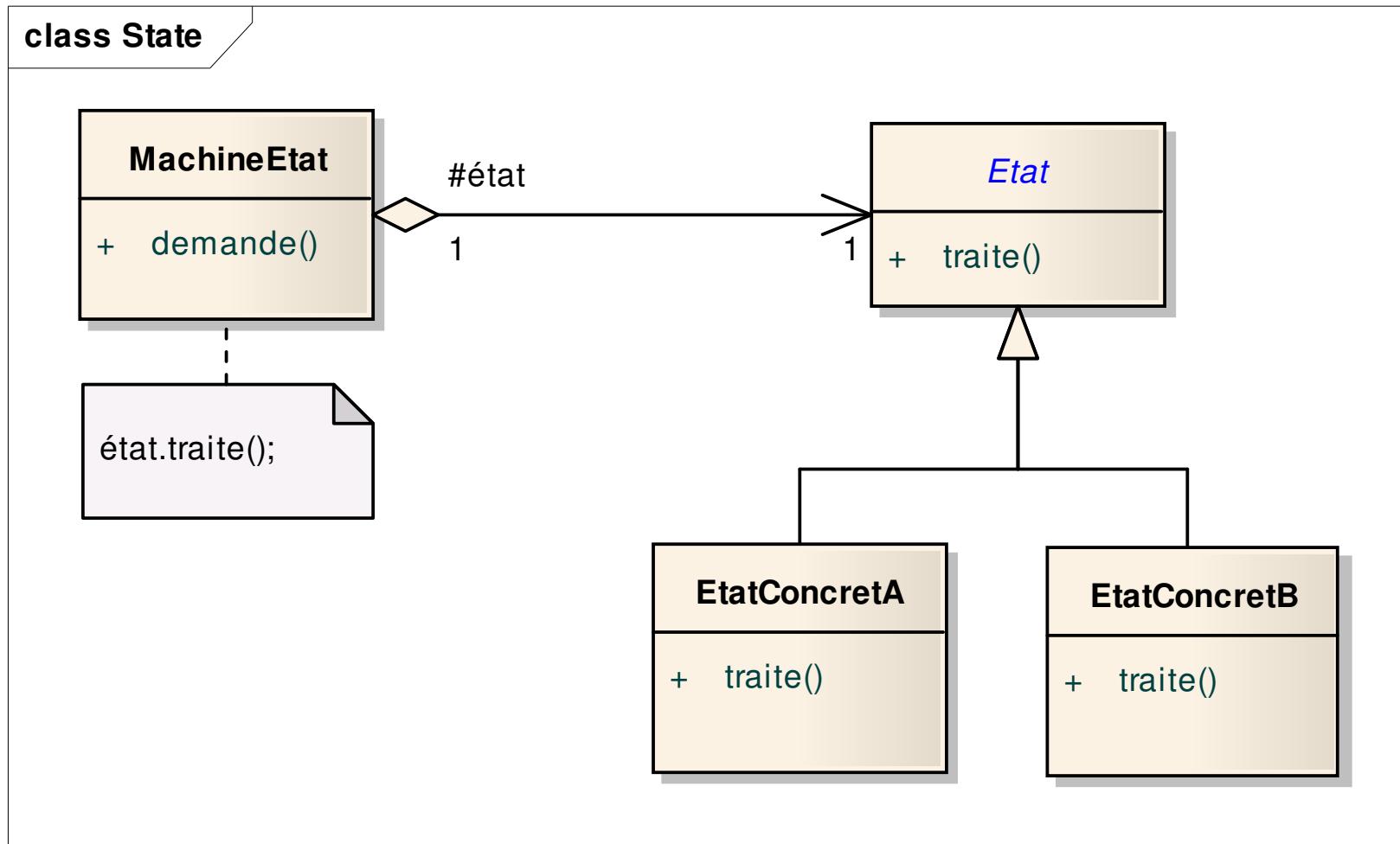
- **Description**

Permet à un objet d'adapter son comportement en fonction de son état interne .

- **Domaine d'utilisation**

- ✓ Le comportement d'un objet dépend de son état.
- ✓ L'implantation de cette dépendance à l'état par des instructions conditionnelles est trop complexe.

3.9 – Exemple : Modèles de comportement -> Pattern State : Structure





- Participants

- MachineEtat : classe concrète décrivant des objets qui sont des machines à état, qui possèdent un ensemble d'état pouvant être décrits dans un diagramme d'états-transitions. Maintient une référence vers une sous-classe d'Etat qui définit l'état courant.
- Etat : classe abstraite qui introduit la signature des méthodes liées à l'état et qui gère l'association avec la machine à états.
- EtatConcretA et EtatConcretB : sous-classes concrètes qui implantent le comportement des méthodes relativement à chaque état.

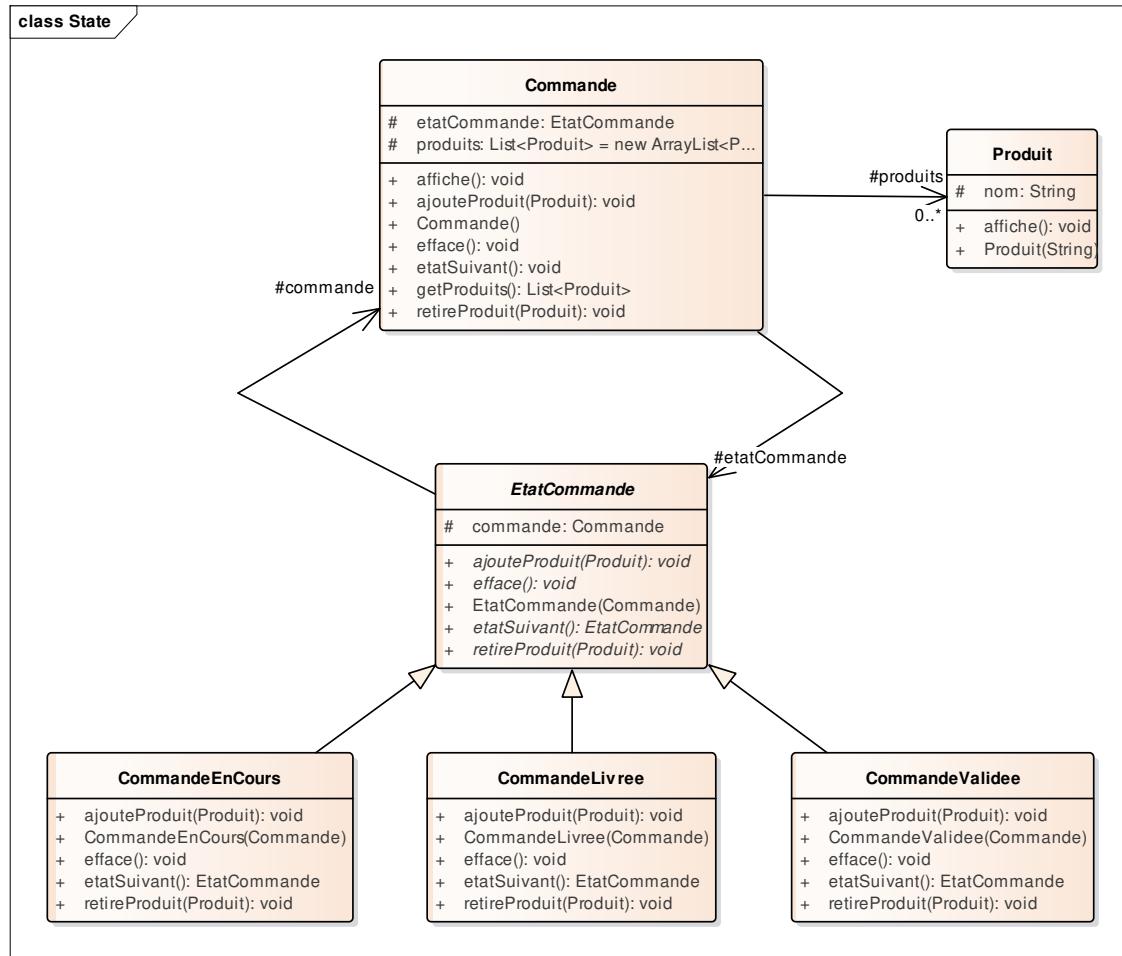
- Collaborations

La machine à état délègue les appels des méthodes dépendant de l'état courant vers un objet d'état. Elle peut transmettre à l'objet d'état un référence vers elle-même.



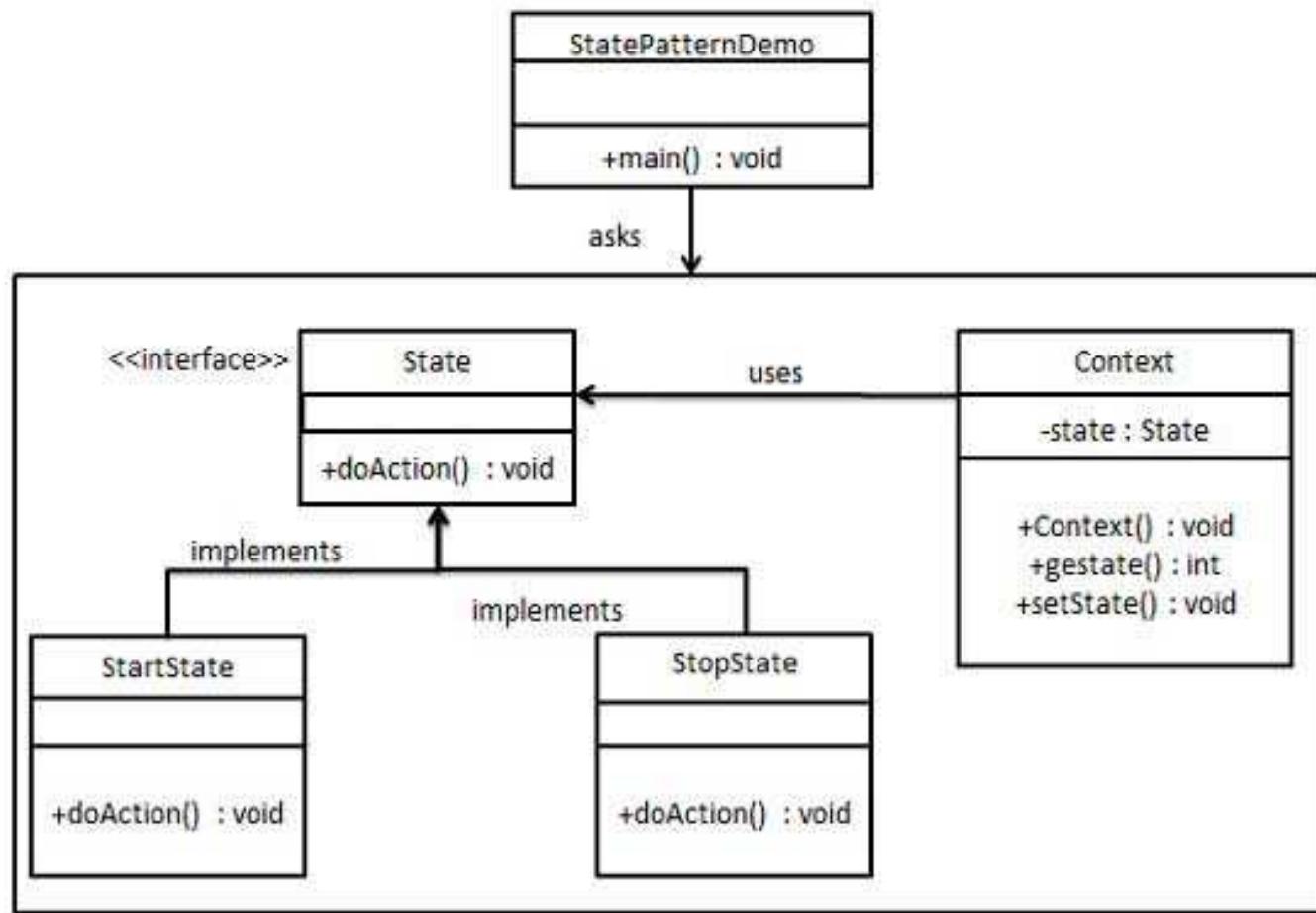
- Exemples
 - IHM : la fonction « display » d'une icône représentant une connexion change selon son état.
 - Changement d'état de la matière : solide, liquide, gazeux.

3.9 – Exemple : Modèles de comportement -> Pattern State : Exemple



<https://www.editions-eni.fr/livre/design-patterns-en-java-les-23-modeles-de-conception-descriptions-et-solutions-illustrees-en-uml-2-et-java-4e-edition-9782409012815>

3.9 – Exemple : Modèles de comportement -> Pattern State : Exemple



https://www.tutorialspoint.com/design_pattern/builder_pattern.htm



Modèle-Vue-Contrôleur (MVC) :

c'est une architecture et une méthode de conception qui organise l'IHM d'une application logicielle.

Ce paradigme divise l'IHM en :

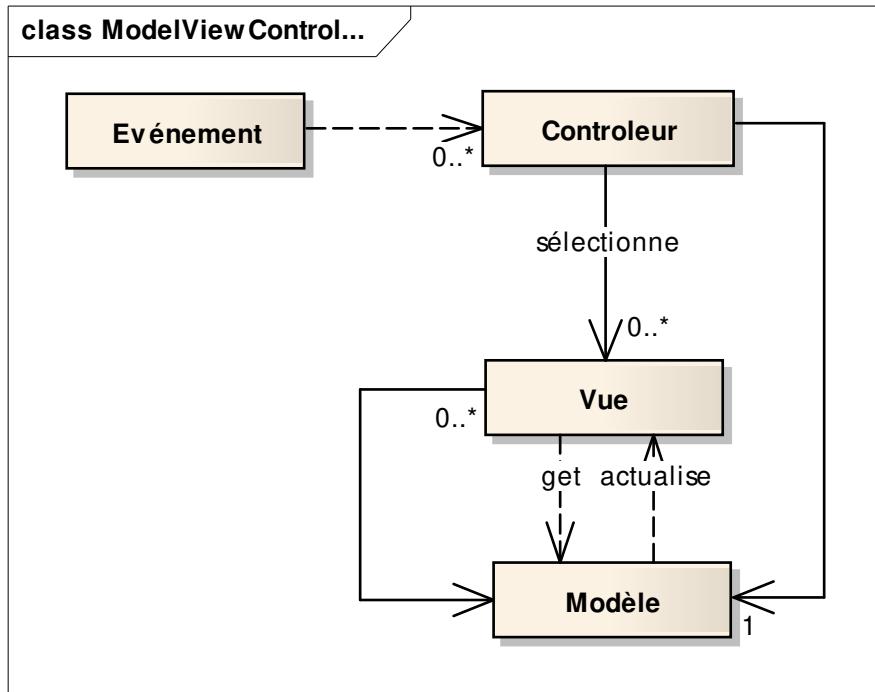
- un **modèle** (modèle de données),
- une **vue** (présentation, interface utilisateur)
- un **contrôleur** (logique de contrôle, gestion des événements, synchronisation),

chacun ayant un rôle précis dans l'interface.

3.9 – Exemple : Autres Patterns -> Model View Controller



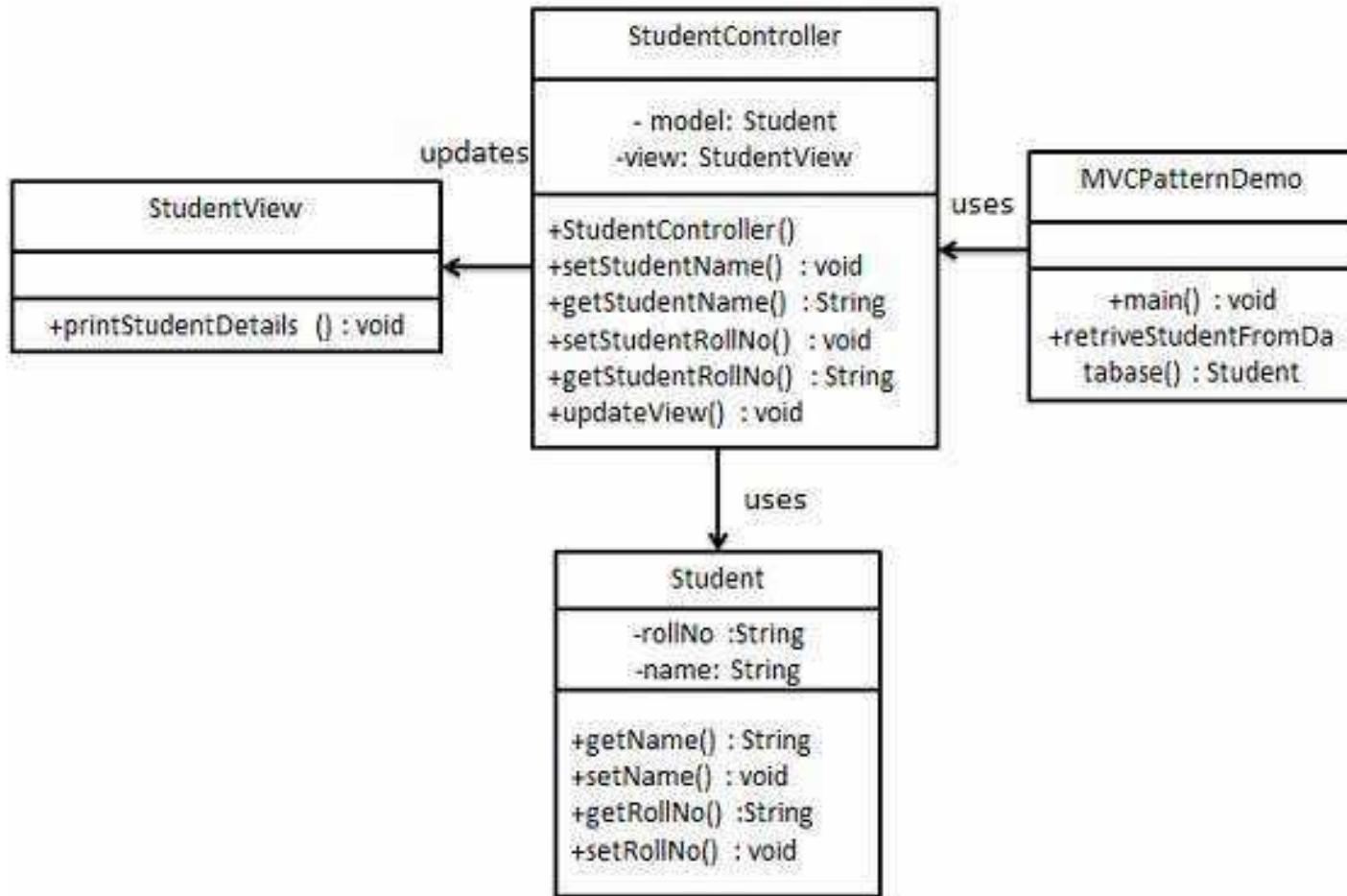
Le MVC peut être vu comme une combinaison de design patterns.



- Contrôleur : Strategy attachées aux vues encapsule les requêtes de modification du modèle que l'utilisateur lance sur la vue
- Vues : organisées en Composite
- Le lien entre Vue et Modèle : Observer
- Modèle : souvent un Mediator

- Les arbres de Vues mettent en œuvre Decorator
- On a souvent besoin d'Adapter pour convertir l'interface d'un Modèle de façon à ce qu'elle soit adaptée à la Vue
- Les Vues créent les contrôleurs avec Factory Method

3.9 – Exemple : Autres Patterns -> Model View Controller : Exemple



https://www.tutorialspoint.com/design_pattern/builder_pattern.htm



Exemples d'architecture MVC :

- En java : Swing (hors conteneurs), JavaServer Faces, Struts, Spring MVC, SWT
- En ECMA Script : Adobbe Flex, XUL
- En C++ : Qt
- En PHP : Zend Framework ...
- En Ruby : Ruby on Rail



Yantra Technologies



D.Palermo

Programmation Orientée Objet Les Concepts

Version 1.0 - 02/2019

151

Copyright : Yantra Technologies 2004-2019



Yantra Technologies



D.Palermo

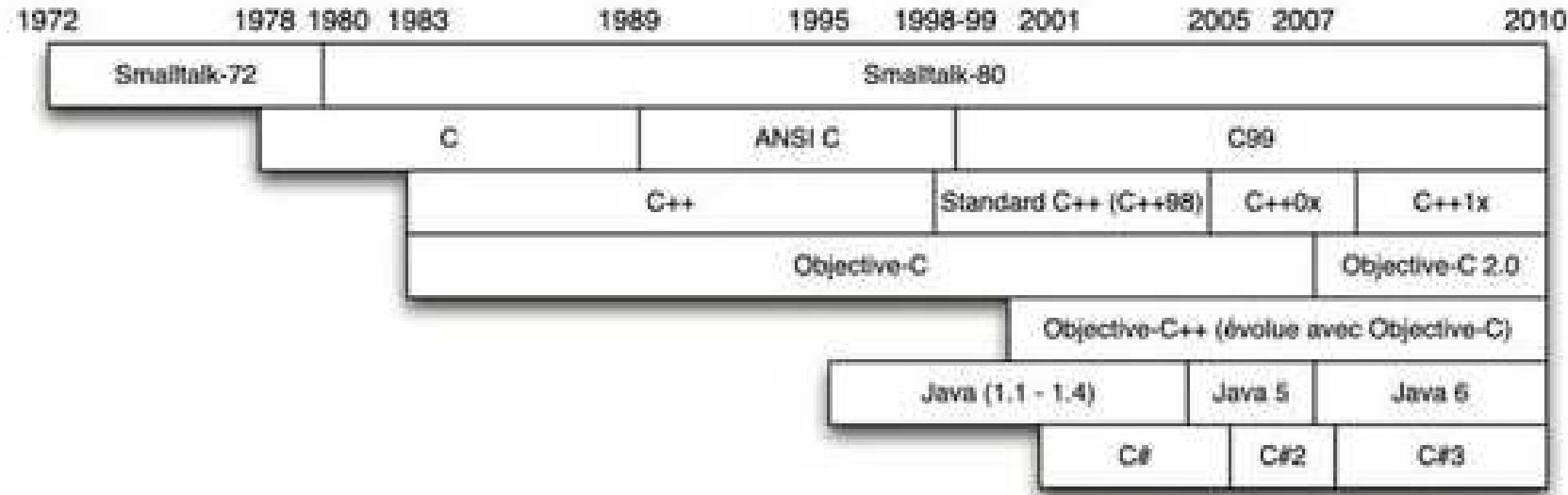
Programmation Orientée Objet Les Concepts

Version 1.0 - 02/2019

152

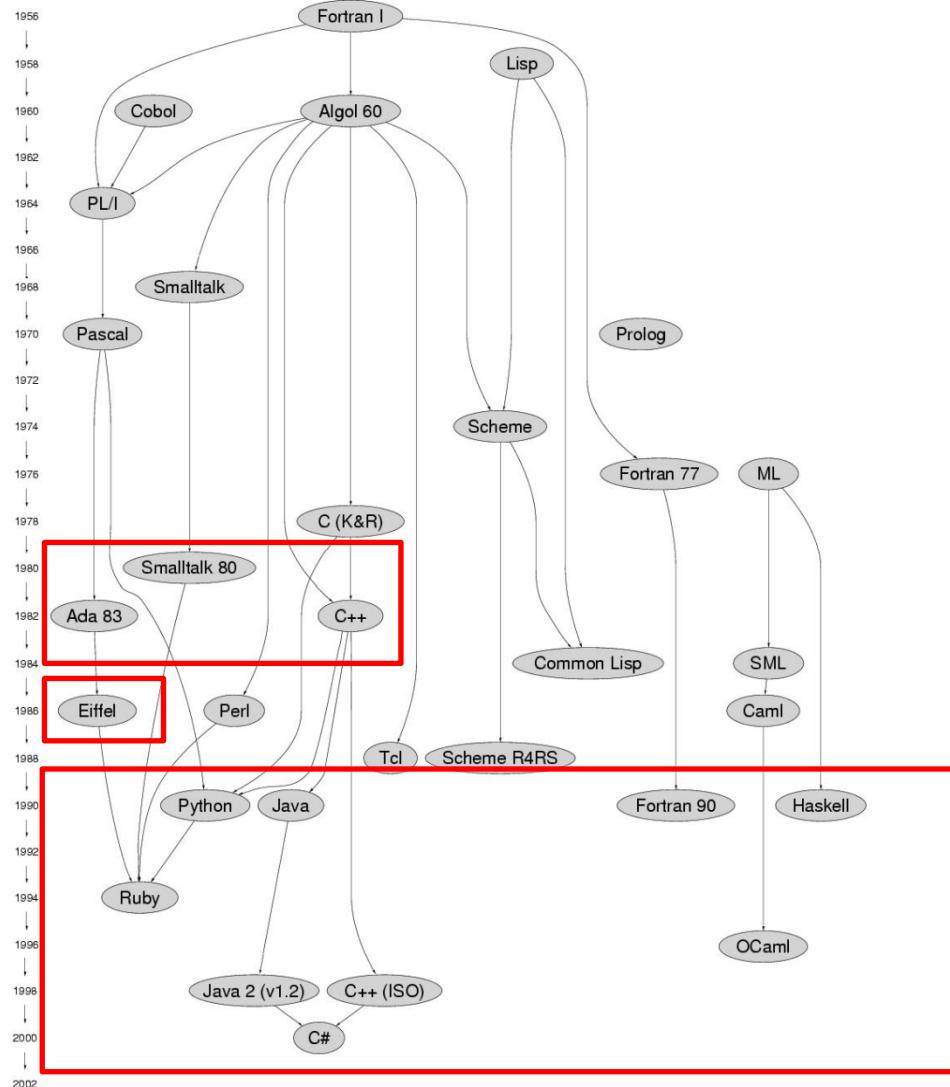
Copyright : Yantra Technologies 2004-2019

4 – Les Différents langages objets



http://librairie.immateriel.fr/fr/read_book/9782212127515/chap001

4 – Les Différents langages objets



4 – Les Différents langages objets



- Ada
- C++
- C#
- Delphi (=Pascal orienté objet)
- Java
- Kylix
- Objective-C
- Objective Caml (ocaml)
- Perl
- PHP (Depuis la version 4)
- Python
- SmallTalk (totalement objet)
- Ruby

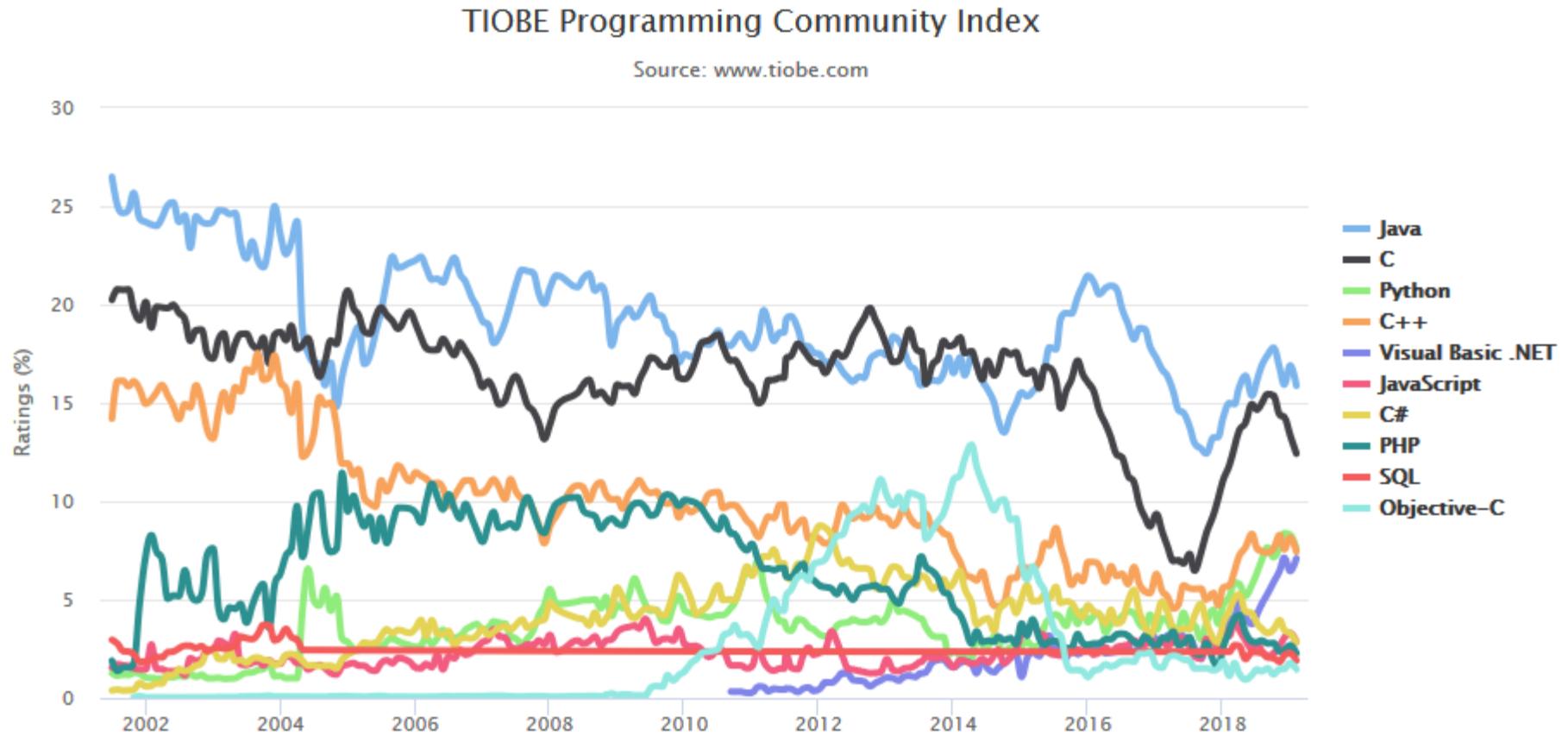
4 – Les Différents langages objets



Feb 2019	Feb 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.876%	+0.89%
2	2		C	12.424%	+0.57%
3	4	▲	Python	7.574%	+2.41%
4	3	▼	C++	7.444%	+1.72%
5	6	▲	Visual Basic .NET	7.095%	+3.02%
6	8	▲	JavaScript	2.848%	-0.32%
7	5	▼	C#	2.846%	-1.61%
8	7	▼	PHP	2.271%	-1.15%
9	11	▲	SQL	1.900%	-0.46%
10	20	▲	Objective-C	1.447%	+0.32%
11	15	▲	Assembly language	1.377%	-0.46%
12	19	▲	MATLAB	1.196%	-0.03%

<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

4 – Les Différents langages objets



<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>



Le langage C++ se veut un langage C amélioré.

- Il possède des fonctionnalités supplémentaires, et notamment :
 - la surcharge de fonctions
 - le passage par référence
 - l'emplacement des déclarations
 - l'allocation dynamique
- Les apports spécifiques de C++ sont :
 - l'aide à l'abstraction de données: définition de types de données, et de leur implémentation concrète.
 - l'aide à la programmation objet: hiérarchie de classes et héritage.
 - l'aide à la programmation générique: classes patron et algorithmes génériques.



1.2.1 – Généralités -> Du C au C++ -> Le C++ façon C



<pre>#include <string.h> #include <stdio.h> #include <stdlib.h> #include <time.h> static void sleep(int delais) { time_t tempsInit,tempsFinal; time(&tempsInit); while ((time(&tempsFinal) - tempsInit) < delais){ } } static int lexsort(const void *a, const void *b) { return strcmp(*(const char**)a, *(const char**)b); } int main(int argc, char **argv) { int i=0; const char **args= malloc(argc * sizeof *args); memcpy(args, argv, argc * sizeof *args); qsort(args, argc, sizeof *args, lexsort); for(i=0; i<argc; ++i) { printf("%s ", args[i]); } putchar('\n'); free(args); sleep(5); return 0; }</pre>	<pre>#include <iostream> #include <string> #include <vector> #include <time.h> static void sleep(int delais) { time_t tempsInit,tempsFinal; time(&tempsInit); while ((time(&tempsFinal) - tempsInit) < delais){ } } using namespace std; int main(int argc, char **argv) { vector<string> args(argv, argv+argc); sort(args.begin(), args.end()); copy(args.begin(), args.end(), ostream_iterator<string>(cout, " ")); cout << endl; sleep(5); }</pre>
---	---

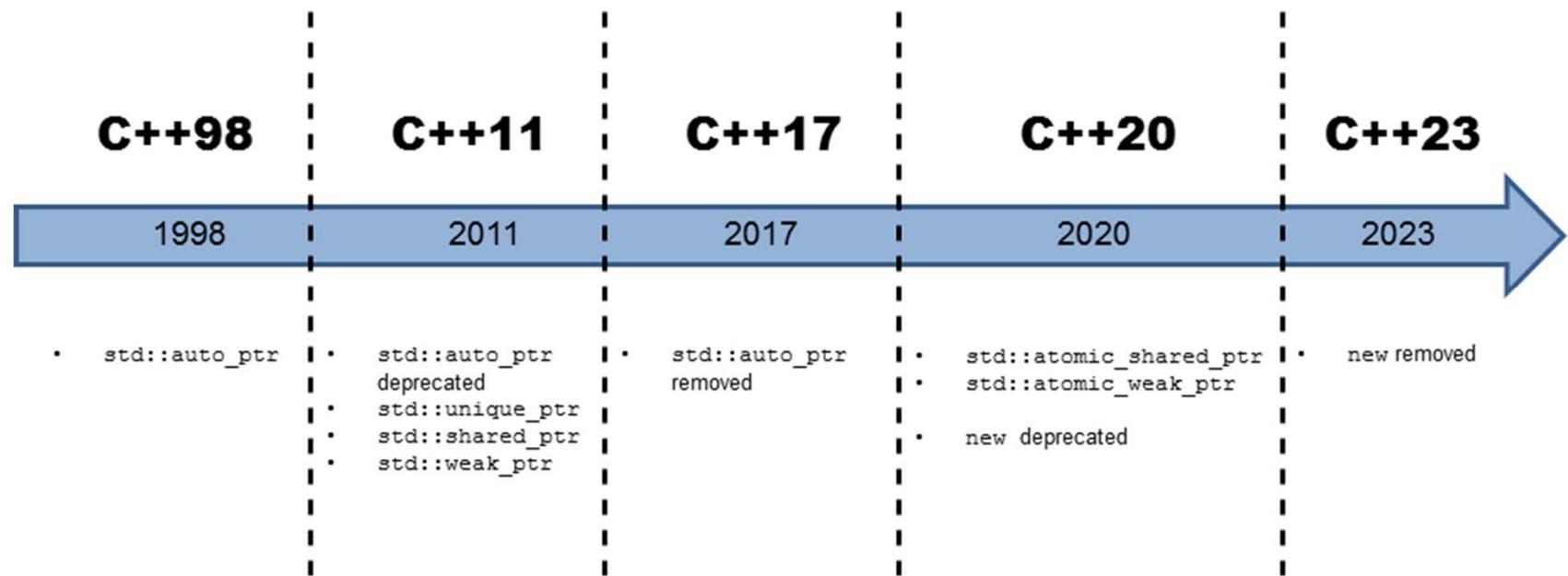


1.2.2 – Généralités -> Du C au C++ -> Historique



- Le langage C a été créé au début des années 1970 par Dennis Ritchie (Laboratoires Bell AT&T). Son but était de ré-écrire le système d'exploitation Unix en langage evolué.
En 1978, le premier standard est publié ("The C Programming Language") par B. W. Kernighan et D.M. Ritchie.
- En 1983, l' "American National Standards Institute" (ANSI) définit le standard ANSI-C qui est une définition explicite et indépendante de la machine pour le langage C".
- Le langage C++ est né en 1983 et a pour but de développer un langage qui garderait les avantages de ANSI-C (portabilité, efficacité) et qui permettrait en plus la programmation orientée objet. Premier standard ANSI-C++ en 1998 corrigée en 2003,
- Nouvelle version 12/08/2011 nouvelle norme du C++ C++11
- Une mise à jour mineure a été publiée en 2014 C++14
- Une mise a jour à été publié en 2017 C++17
- Une mise a jour à été publié en 2020 C++20

1.2.2 – Généralités -> Du C au C++ -> Historique



<https://www.modernescpp.com/>



17

C++20

2020

is
ing
`_view`
ms of the STL
ry
`std::optional,`
`:itant`

The Big Four

- Concepts
- Ranges library
- Coroutines
- Modules

Core Language

- Three-way comparison operator
- Strings literals as template parameters
- `constexpr` virtual functions
- Redefinition of `volatile`
- Designated initializers
- Various lambda improvements
- New standard attributes
- `consteval` and `constinit` keyword
- `std::source_location`

Library

- Calender and time-zone
- `std::span` as a view on a contiguous array
- `constexpr` containers such as `std::string` and `std::vector`
- `std::format`

Concurrency

- `std::atomic_ref<T>`
- `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>`
- Floating point atomics
- Waiting on atomics
- Semaphores, latches, and barriers
- `std::jthread`

<https://www.modernescpp.com/>

1.2.3 – Généralités -> Du C au C++ -> Différences



C	C++
C a été développé par Dennis Ritchie entre 1969 et 1973 chez AT & T Bell Labs.	C++ a été développé par Bjarne Stroustrup en 1979 avec le prédecesseur de C++.
Comparé à C++, C est un sous-ensemble de C++.	C++ est un sur-ensemble de C. C++ peut exécuter la plupart du code C, alors que C ne peut pas exécuter de code C++.
C ne supporte pas la programmation orientée objet; par conséquent, il ne prend pas en charge le polymorphisme, l'encapsulation et l'héritage.	En tant que langage de programmation orienté objet, C++ prend en charge le polymorphisme, l'encapsulation et l'héritage.
En C (parce que c'est un langage de programmation procédural), les données et les fonctions sont des entités séparées et libres.	En C++ (lorsqu'il est utilisé comme langage de programmation orienté objet), les données et les fonctions sont encapsulées ensemble sous la forme d'un objet. Pour la création d'objets, la classe fournit un plan de structure de l'objet.
En C, les données sont des entités libres et peuvent être manipulées par un code extérieur. En effet, C ne prend pas en charge l'encapsulation d'informations.	En C++, Encapsulation masque les données pour garantir que les structures de données et les opérateurs sont utilisés comme prévu.
C étant un langage de programmation procédurale, c'est un langage basé sur les fonctions.	Alors que C++, étant un langage de programmation orientée objet, c'est un langage orienté objet.
C ne prend pas en charge la surcharge des fonctions et des opérateurs.	C++ supporte à la fois surcharge des fonctions et d'opérateurs.

Ref : <https://waytolearnx.com/2018/12/differences-entre-c-et-c.html>

1.2.3 – Généralités -> Généralités -> Du C au C++ -> Différences



C	C++
C ne permet pas de définir des fonctions dans des structures.	En C++, les fonctions peuvent être utilisées dans une structure.
C n'a pas de fonctionnalité de NAMESPACE	C++ utilise NAMESPACE pour éviter les conflits de noms de variables/fonctions/classes...
C utilise des fonctions pour les entrées/sorties. Par exemple, scanf et printf.	C++ utilise des objets pour la sortie. Par exemple cin et cout.
C ne supporte pas les variables de référence.	C++ supporte les variables de référence.
C n'a pas de support pour les fonctions virtuelles et friend.	C++ prend en charge les fonctions virtuelles et friend.
C fournit les fonctions malloc() et calloc() pour l'allocation dynamique de la mémoire et free() pour la dés-allocation de mémoire.	C++ fournit un nouvel opérateur pour l'allocation de mémoire et l'opérateur de suppression pour la dés-allocation de mémoire.
C ne fournit pas de support direct pour le traitement des erreurs (également appelé traitement des exceptions)	C++ fournit un support pour la gestion des exceptions. Les exceptions sont utilisées pour des erreurs « difficiles » qui rendent le code incorrect.
Extension du fichier C est .C	Extension du fichier C++ est .CPP

Ref : <https://waytolearnx.com/2018/12/differences-entre-c-et-c.html>



Déclarations de fonctions :

- En C++ : Toute fonction doit être définie avant utilisation ou être déclarée par un prototype :
 - C++ : float fct (int, double, char*);
 - C : float fct ();

Fonctions sans argument :

- En C++ : une fonction sans argument se définit et se déclare en fournissant une liste d'arguments vide :
 - C++ : float fct ();
 - C : float fct (void);

Fonctions sans valeur de retour

- En C++ : une fonction sans valeur de retour se définit et se déclare à l'aide du mot clé void :
 - C++ : void fct (int, double, char*)
 - C : fct (); // facultatif

Le qualificatif const

- En C++ : un symbole global déclaré avec le qualificatif const peut être utilisé pour une expression constante
 - const int N = 10; // remplace #define N 10
 - int valeurs[N];

1.3.1 – Généralités -> Environnement de développement



Compilateurs gratuits :

Eclipse CDT : (<http://www.eclipse.org/cdt/>) fournit un environnement de développement moderne et modulaire pour la réalisation de projets C et C++.

Sous Windows :

Cygwin (<http://www.cygwin.com/>). Un environnement qui permet d'émuler Linux sous Windows, et donc en particulier de disposer du compilateur Gcc.

Code :: Blocks (<http://www.codeblocks.org/>) est un environnement de développement en C++ entièrement configurable et extensible à l'aide de nombreux plugins. A la fois complet et simple d'utilisation, il supporte divers compilateurs : GCC, Microsoft Visual C++ Toolkit 2003, Microsoft Visual C++ Express 2005, Borland C++ 5.5, Intel C++ compiler, etc.

NetBeans (<https://netbeans.org/>) est un environnement de développement intégré (EDI), placé en open source par Sun en juin 2000

Dev-C++ (<http://www.bloodshed.net/devcpp.html>). c'est un système complet de développement C/C++, tournant cette fois sous Windows directement. Comme Dlgpp, il comprend un éditeur multi-fichiers, un compilateur, un gestionnaire de projet et un débogueur. Le compilateur de base est d'ailleurs le même : Gcc

MinGW (<http://www.mingw.org/>) C'est un compilateur C/C++ qui produit des exécutables Win32. Il constitue une alternative au Visual C++ de Microsoft. Le package comprend un pré-processeur, un compilateur, un linker de même qu'une très grande partie des fichiers en-têtes pour programmer sous Win32. MinGW s'exécute par la ligne de commande

Sous Unix : Gcc/G++. Les sources sont téléchargeables à partir de <http://gcc.gnu.org/> avec un débogueur graphique (ddd) à l'adresse <http://www.gnu.org/software/ddd/>.



1.3.1 – Généralités -> Environnement de développement



Compilateurs C gratuits pour Windows

- **MinGw (<http://www.mingw.org/>)**
- **Borland C++ Compiler**
- **Digital Mars**

Environnements de développement intégrés C gratuits

- **Visual C++ Express**
- **CodeBlocks Studio (<http://www.codeblocks.org/>)**
- **NetBeans I.D.E.**
- **Qt C++**
- **Anjuta**
- **KDevelop**
- **Glade (GTK a& GNOME)**

Compilateurs / EDI C commerciaux disponibles en évaluation

- **Microsoft Visual Studio .NET**

Divers

- **Doxygen**
- **Valgrind**
- **Intel VTune Performance Analyzer**
- **Intel Parallel Studio**
- **TotalView Debugger**

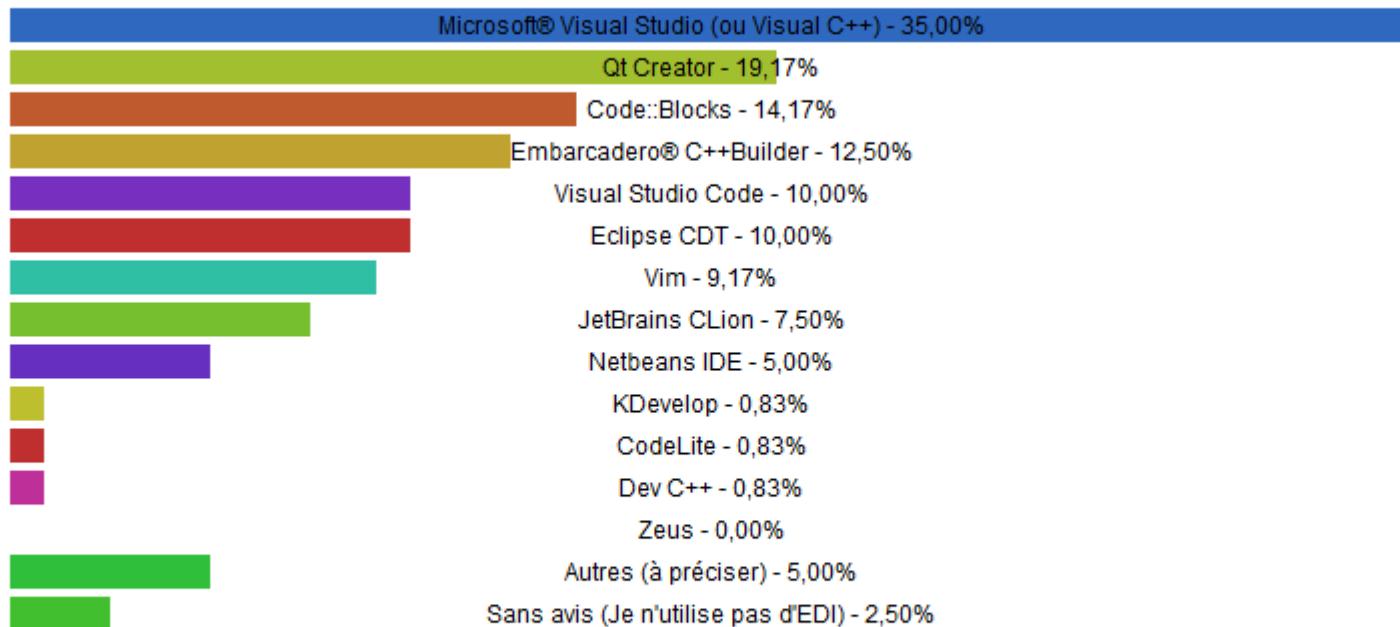
Voir : <http://c.developpez.com/compilateurs/>



1.3.1 – Généralités -> Environnement de développement



Quels environnements de développement intégrés C/C++ utilisez-vous en 2018 ? Et pourquoi ?



240 votants

[Voter](#)

<https://www.developpez.com/actu/189329/Quels-sont-les-EDI-C-Cplusplus-les-plus-utilises-en-2018-Un-developpeur-nous-livre-les-resultats-de-son-etude/>



1.3.1 – Généralités -> Environnement de développement



1. **Microsoft Visual Studio**
2. **Qt Creator**
3. **Code::blocks**
4. **Embarcadero® C++Builder**
5. **Eclipse CDT**
6. ***GNAT Programming Studio***
7. ***CodeLite***
8. ***NetBeans 8***
9. ***Sublime Text***
10. ***Dev C++***
11. ***Anjuta***
12. ***Clion***
13. ***MonoDevelop***
14. ***Linx***

<https://codecondo.com/top-10-ide-for-c-and-cplusplus-for-programmers/>

<https://www.easypartner.fr/blog/les-meilleurs-ide-c/>



1.3.2 – Généralités -> Liens divers



- <http://www.cplusplus.com/reference/>
- https://fr.wikibooks.org/wiki/Exercices_en_langage_C%2B%2B
- <http://slauncha.dyndns.org/index.php?article12/compilateur-c-online>
- <http://cours.sebastien-pastore.com/cours/PPE/Epreuves%20orales/E4.php>
- <http://www.reseaucerta.org/exonet-bts-sio-slam2>



- 2.0 - Mots-clés du C++
- 2.1 - Les types prédéfinis
- 2.2 - Les commentaires
- 2.3 - Les opérateurs
- 2.4 - Les structures de contrôle
- 2.5 - Les fonctions
- 2.6 - Les pointeurs
- 2.7 - Les tableaux
- 2.8 - Les directives de précompilation
- 2.9 - Le type struct en C
- 2.10 - Le type union
- 2.11 - Le type enum
- 2.12 - type auto (C++11)
- 2.13 - decltype (C++11)
- 2.14 - constexpr (C++11)
- 2.15 - rvalue références && (C++ 11)



2.0 - Les Notions de Base C/C++ -> Mots-clés du C++



alignas (since C++11)	default(1)	register(2)
alignof (since C++11)	delete(1)	reinterpret_cast
and	do	requires (since C++20)
and_eq	double	return
asm	dynamic_cast	short
atomic_cancel (TMTS)	else	signed
atomic_commit (TMTS)	enum	sizeof(1)
atomic_noexcept (TMTS)	explicit	static
auto(1)	export(1)(3)	static_assert (since C++11)
bitand	extern(1)	static_cast
bitor	false	struct(1)
bool	float	switch
break	for	synchronized (TMTS)
case	friend	template
catch	goto	this
char	if	thread_local (since C++11)
char8_t (since C++20)	inline(1)	throw
char16_t (since C++11)	int	true
char32_t (since C++11)	long	try
class(1)	mutable(1)	typedef
compl	namespace	typeid
concept (since C++20)	new	typename
const	noexcept (since C++11)	union
constexpr (since C++20)	not	unsigned
constexprr (since C++11)	not_eq	using(1)
constinit (since C++20)	nullptr (since C++11)	virtual
const_cast	operator	void
continue	or	volatile
co_await (since C++20)	or_eq	wchar_t
co_return (since C++20)	private	while
co_yield (since C++20)	protected	xor
decltype (since C++11)	public	xor_eq
	reflexpr (reflection TS)	

<https://en.cppreference.com/w/cpp/keyword>

Mots clé spéciaux

override (C++11)
final (C++11)
import (C++20)
module (C++20)
transaction_safe (TMTS)
transaction_safe_dynamic (TMTS)

Preprocesseur

if	ifdef	include	defined
elif	ifndef	line	_has_include (since C++17)
else	define	error	_has_cpp_attribute (since C++20)
endif	undef	pragma	

_Pragma (since C++11)

TMTS :Transactional Memory Technical Specification

https://en.cppreference.com/w/cpp/language/transactional_memory





2.1 - Les Notions de Base C/C++ -> Types prédéfinis



Type	Taille (en octets)	Type	Format	Plage de valeurs	
				Approximate	Exact
caractère	1	signed char	signé (complément à un)	-127 to 127	
			signé (complément à deux)	-128 to 127	
		unsigned char	non-signé	0 to 255	
entier	2	short int int	signé (complément à un)	$\pm 3.27 \cdot 10^4$	-32767 to 32767
			signé (complément à deux)		-32768 to 32767
		unsigned short int unsigned int	non-signé	$0 \text{ to } 6.55 \cdot 10^4$	0 to 65535
	4 (32 bits)	int long	signé (complément à un)	$\pm 2.14 \cdot 10^9$	-2,147,483,647 to 2,147,483,647
			signé (complément à deux)		-2,147,483,648 to 2,147,483,647
		unsigned int unsigned long int	non-signé	$0 \text{ to } 4.29 \cdot 10^9$	0 to 4,294,967,295
	8 (64 bits)	long int long long int	signé (complément à un)	$\pm 9.22 \cdot 10^{18}$	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
			signé (complément à deux)		-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned long long int	non-signé	$0 \text{ to } 1.84 \cdot 10^{19}$	0 to 18,446,744,073,709,551,615
floating point	4 (32 bits)	float	IEEE-754	$\pm 3.4 \cdot 10^{\pm 38}$	min subnormal: $\pm 1.401,298,4 \cdot 10^{-47}$
				(~ 7 chiffres)	min normal: $\pm 1.175,494,3 \cdot 10^{-38}$
					max: $\pm 3.402,823,4 \cdot 10^{38}$
	8 (64 bits)	double	IEEE-754	$\pm 1.7 \cdot 10^{\pm 308}$	min subnormal: $\pm 4.940,656,458,412 \cdot 10^{-324}$
				(~ 15 chiffres)	min normal: $\pm 2.225,073,858,507,201,4 \cdot 10^{-308}$
					max: $\pm 1.797,693,134,862,315,7 \cdot 10^{308}$
		long double			-3.4*10^-4932 à 3.4*104932



2.1 - Les Notions de Base C/C++ -> Types prédéfinis



Déterminant du type	Type équivalent	Largeur (bits) en mémoire				
		C++ standard	LP32 / Win16	ILP32 / Win32 & Unix	LLP64 / Win64	LP64 / Unix
short	short int	au moins 16	16	16	16	16
short int						
signed short						
signed short int						
unsigned short						
unsigned short int						
int	int	au moins 16	16	32	32	32
signed						
signed int						
unsigned						
unsigned int						
long	long int	au moins 32	32	32	32	64
long int						
signed long						
signed long int						
unsigned long						
unsigned long int						
long long	long long int (C++11)	au moins 64	64	64	64	64
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C++11)					
unsigned long long int						



void	type vide
wchar_t	type représentant caractère large. (ex. UTF-8)
char16_t	type représentant un caractère UTF-16. (depuis C++11)
char32_t	type représentant un caractère UTF-32. (depuis C++11)
bool	true/false

- les tableaux à une dimension, dont les indices sont spécifiés par des crochets ('[' et ']').
- les structures, unions et énumérations, les fonctions
- Le type auto pour faire déduire le type par le compilateur

Remarque :

- la taille des types peut dépendre des compilateurs,
- en hexadécimal, ils s'écrivent de la manière suivante : 127 -> 0x7f
- un bit est réservé pour le signe



Commentaire C :

```
/* Ceci est un commentaire C */
```

Commentaire C++ :

```
// Ceci est un commentaire C++
```

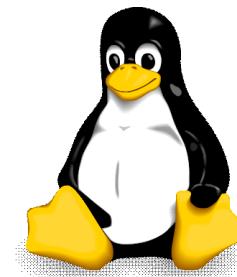
(Remarque : voir doxygen)



2.2 - Les Notions de Base C/C++ -> doxygen



Doxxygen



Facile



Normal



Difficile



Professional



Expert



D.Palermo

Outilage : Doxygen
Version 4.4 - 02/2020

Copyright : Yantra Technologies 2004-2020



D.Palermo

Programmation C++
Version 4.4 - 02/2020

Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



- 2.3.1 - Listes des opérateurs
- 2.3.2 - Priorité des opérateurs
- 2.3.3 - Les opérateurs : arithmétiques
- 2.3.4 - Les opérateurs : binaires
- 2.3.5 - Les opérateurs : conditionnels
- 2.3.6 - Les opérateurs : simplifiés
- 2.3.7 - Les opérateurs : logiques

2.3.1 - Les Notions de Base C/C++ -> Les opérateurs : Listes des opérateurs



<u>affectation</u>	<u>incrémantion décrémentation</u>	<u>arithmétique</u>	<u>logique</u>	<u>comparaison</u>	<u>accès aux membre</u>	<u>autre</u>
$a = b$	$++a$	$+a$	$!a$	$a == b$	$a[b]$	$a(...)$
$a = rvalue$	$--a$	$-a$	$a \&& b$	$a != b$	$*a$	a, b
$a += b$	$a++$	$a + b$	$a b$	$a < b$	$\&a$	$(type) a$
$a -= b$	$a--$	$a - b$		$a > b$	$a->b$	$? :$
$a *= b$		$a * b$		$a <= b$	$a.b$	
$a /= b$		a / b		$a >= b$	$a->*b$	
$a %= b$		$a \% b$			$a.*b$	
$a \&= b$		$\sim a$				
$a = b$		$a \& b$				
$a ^= b$		$a b$				
$a <<= b$		$a ^ b$				
$a >>= b$		$a << b$				
		$a >> b$				

Opérateurs spéciaux

static_cast convertit un type à un autre type compatible

dynamic_cast convertit classe de base virtuelle à class dérivée

const_cast convertit un type à l'autre compatible avec cv qualifiers

reinterpret_cast convertit type type incompatibles

new alloue memory

delete libère memory

sizeof interroge la taille d'un type

sizeof... interroge la taille d'un Paquet de paramètre (depuis C++11)

typeid interroge les informations de type d'une type

noexcept vérifie si une expression peut jeter une exception (depuis C++11)

alignof requête les exigences d'alignement d'un (depuis C++11) type



2.3.2 - Les Notions de Base C/C++ -> Les opérateurs : Priorité des opérateurs (1/3)



Priorité	Opérateur	Description	Associativité
1	::	Scope resolution	Gauche à droite
2	++ --	Incrémantation et décrementation suffixe/postfixe	
	()	Appel de fonction	
	[]	Array subscripting	
	.	Sélection membre par référence	
	->	Sélection membre par pointeur	
3	++ --	Incrémantation et décrémentation préfixe	Droite à gauche
	+ -	Plus et moins unaires	
	! ~	NON logique et NON binaire	
	(type)	Transtypage	
	*	Indirection (déréférence)	
	&	Adresse de	
	sizeof	Taille de	
	new, new[]	Allocation mémoire dynamique	
	delete, delete[]	Désallocation mémoire dynamique	



2.3.2 - Les Notions de Base C/C++ -> Les opérateurs : Priorité des opérateurs (2/3)



Priorité	Opérateur	Description	Associativité
4	<code>.* ->*</code>	Pointeur vers un membre	Gauche à droite
5	<code>* / %</code>	Multiplication, division, et reste	
6	<code>+ -</code>	Addition et soustraction	
7	<code><< >></code>	Décalage binaire à gauche et à droite	
8	<code>< <=</code>	Respectivement pour les opérateurs de comparaison < et ≤	
	<code>> >=</code>	Respectivement pour les opérateurs de comparaison > et ≥	
9	<code>== !=</code>	Respectivement pour les comparaisons = et ≠	
10	<code>&</code>	ET binaire	
11	<code>^</code>	XOR binaire (ou exclusif)	
12	<code> </code>	OU binaire (ou inclusif)	
13	<code>&&</code>	ET logique	
14	<code> </code>	OU logique	

2.3.2 - Les Notions de Base C/C++ -> Les opérateurs : Priorité des opérateurs (3/3)



Priorité	Opérateur	Description	Associativité
15	?:	Ternary conditional	Droite à gauche
	=	Affectation directe (fourni par défaut pour les classes C++)	
	+ = - =	Affectation par somme ou différence	
	* = / = % =	Affectation par produit, division ou reste	
	<<= >>=	Affectation par décalage binaire à gauche ou à droite	
	& = ^ = =	Affectation par ET, XOR ou OU binaire	
16	throw	Throw operator (for exceptions)	
17	,	Virgule	Gauche à droite



Les entrées et sorties sont gérées en C++ à travers des objets particuliers appelés **streams** ou **flots**.

```
#include <iostream>
```

Deux opérateurs sont surchargés de manière appropriée pour les flots:

- l'opérateur d'insertion **<<** (écriture)
- l'opérateur d'extraction **>>** (lecture)

Trois flots prédéfinis sont :

- **cout** attaché à la sortie standard
- **cerr** attaché à la sortie erreur standard
- **cin** attaché à l'entrée standard (**sécuriser le cin => <http://sdz.tdct.org/sdz/la-saisie-securisee-en-c.html>**)

On retrouve les flots du C **stdin**, **stdout**, **stderr**.



- Sur les réels : + - / *
- Sur les entiers : + - / (quotient de la division), % (reste de la division)

l'affectation se fait grâce au signe '='

Hiérarchie habituelle (* et / prioritaires par rapport aux + et -)

2.3.3 - Les Notions de Base C/C++ -> Les opérateurs : arithmétiques -> Exemple



```
#include <iostream>

int main(int , char *[])
{
    int x = 0;
    int a = 0;
    int ag = 0;
    int age = 0;
    int AgeDUnePersonne=0;

    x=5;
    a=10;
    ag=15;
    age=20;
    AgeDUnePersonne = 50 ;

    std::cout << x << " " << a << " " << ag
                  << " " << age << " " << AgeDUnePersonne << std::endl;
    return 0;
}
```

2.3.3 - Les Notions de Base C/C++ -> Les opérateurs : arithmétiques -> Exemple



```
int main(int , char *[]){
{
    //ENTIER
    int x=0;
    int y=0;
    int z=0;
    int a=0;

    x = 1 + 2;
    y = 1 - 2;
    z = x * y;
    a = x / 2;
    std:: cout << x << " " << y << " " << z << " " << a << std::endl;
}
{
    //REEL
    double x=0.0;
    double y=0.0;
    double z=0.0;
    double a=0.0;

    x = 1.2 + 2.2;
    y = 1.3 - x;
    z = x * y;
    a = x / 2.3;
    std:: cout << x << " " << y << " " << z << " " << a << std::endl;
}
return 0;
}
```

2.3.4 - Les Notions de Base C/C++ -> Les opérateurs : binaires



- &, bitand (et)

Table de vérité AND		
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

- |, bitor (ou)

Table de vérité OR		
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

- ^, xor (ou exclusif)

Table de vérité XOR		
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



- `&=`, `and_eq`, // $a \&= b \Leftrightarrow a = a \& b$
- `|=`, `or_eq`, // $a | b \Leftrightarrow a = a | b$
- `^=`, `xor_eq`, // $a ^= b \Leftrightarrow a = a ^= b$
- `~`, `compl` : complément à 1
- `<<`, décalage à gauche
- `>>`, décalage à droite

Table de vérité NOT	
A	NOT A
0	1
1	0

Exemple : `p = n << 3;`
`p` égal `n` décalé de 3 bits sur la gauche



`expr1 ? expr2 : expr3`

si (expr1) alors expr2 sinon expr3

Exemple : fonction min

```
if (y<z) x=y; else x=z;
```

Équivalent à

```
x = ( y<z ) ? y : z ;
```



- $i=i+1$ peut s'écrire $i++;$ $(++i;)$
- $i=i-1$ peut s'écrire $i--;$ $(--i;)$
- $a=a+b$ peut s'écrire $a+=b;$
- $a=a-b$ peut s'écrire $a-=b;$
- $a=a&b$ peut s'écrire $a&=b;$



2.3.7.1 - Quelle utilisation ?

2.3.7.2 - Opérateurs classiques

2.3.7.3 - Les expressions

2.3.7.4 - Et - Ou - !



- Tous les tests :
 - si (expression vraie) ...
 - tant que (expression vraie) ...
- Valeurs booléennes
 - Vrai (1)
 - Faux (0)



- Le 'ET'

A	B	ET
0	0	0
0	1	0
1	0	0
1	1	1

- Le 'OU'

A	B	OU
0	0	0
0	1	1
1	0	1
1	1	1



- Egalité

$a == b$

- Inégalité

$a != b$

- Relations d'ordre

$a < b$ $a <= b$ $a > b$ $a >= b$

- Expression

a est vraie si a est différent de 0



- Et Logique
 - expression1 **&&** expression2
 - expression1 **and** expression2
 - *si expression1 et expression2 sont vraies*
- Ou Logique
 - expression1 **||** expression2
 - expression1 **or** expression2
 - *si expression1 ou expression2 est vraie*
- **!** (Non)
 - **!expression**
 - **not expression**
 - *si expression est fausse*



```
int main(int , char *[]){
    bool a = true;
    bool b = false;
    std::cout << std::boolalpha << a << " " << b << std::endl;
    a = !b;
    std::cout << std::boolalpha << a << " " << b << std::endl;
    a = a && b;
    std::cout << std::boolalpha << a << " " << b << std::endl;
    b = a || b;
    std::cout << std::boolalpha << a << " " << b << std::endl;

    return 0;
}
```



2.4.0 - Notion bloc

2.4.1 - Si ... alors ... sinon ... => if () {} else {}

2.4.2 - Au cas ... ou faire ... => switch () { case }

2.4.3 - Tant que ... faire ... => while () {}

2.4.4 - Répéter ... tant que ...=> do {} while ()

2.4.5 - Boucle Pour ... faire ... => for () {}

2.4.6 - Boucle Pour ... faire ... => for () {} - (C++11)

2.4.7 - Saut => goto

2.4.8 - Rupture de séquence => break, continue, return



Yantra Technologies

2.4.0 - Les Notions de Base C/C++ -> Les structures de contrôle : Notion bloc



```
#include <iostream>

int i = 12;

int main()
{
    std::cout << i << std::endl;
    int i = 4;
    {
        double nom = 5.;
        nom = 2.3;
        std::cout << i << std::endl;
        int i = 0;
        i = nom / 2.;
        bool v = false;
        std::cout << v << std::endl;
        std::cout << i << std::endl;
    }
    // std::cout << v << std::endl; error: 'v' was not declared in this scope
    std::cout << i << std::endl;
    return 0;
}
```

12
4
0
1
4



Yantra Technologies



Si (expression conditionnelle vraie)
alors { instructions 1 }
sinon { instructions 2 }

```
if (expression) {
    instructions 1;
}
else {
    instructions 2;
}

if (expression) {
    instructions 1; /* bloc else
        optionnel */
}
```

```
if (expression)
    instruction; /* bloc d'1
                    instruction */

if (expression) instruction1;
else instruction2;
```



2.4.1 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[]) {
    char quit = '\0';
    cout << " Bonjour ! " << endl;
    cout << " Appuyer sur la touche q pour quitter " << endl;
    cin >> quit;
    if ( quit == 'q' ) return 0;

    if ( quit == 'Q' ) {
        cout << " Appuyer sur la touche q et pas Q pour quitter " << endl;
        cin >> quit;
    }
    else
    {
        cout << " Appuyer sur la touche q pour quitter " << endl;
        cin >> quit;
    }

    if ( quit == 'q' ) return 0;
    else cout << "Dernier essai, Appuyer sur la touche q pour quitter " << endl;

    cin >> quit;
    return 0;}
}
```





2.4.2 - Les Notions de Base C/C++ ->

Les structures de contrôle : Au cas ... ou faire ...
=> **switch () { case }**



Au cas où la variable vaut :

valeur 1 : faire ... ;
valeur 2 : faire ... ; etc.

```
switch (variable de type char ou int) {  
    case valeur 1 : ...;  
        ...;  
        break;  
    case valeur 2 : ...;  
        ...;  
        break;  
    default :      ...;  
        ...;  
}
```



2.4.2 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
    char quit = '\0';
    cout << " Bonjour ! " << endl;
    cout << " Appuyer sur la touche q pour quitter " << endl;
    cin >> quit;
    switch ( quit )
    {
        case 'q' :
            return 0; break;
        case 'Q' :
            cout << " Appuyer sur la touche q et pas Q pour quitter " << endl;
            cin >> quit;
            break;
        case 'r':case 'R':case 'S' : case 'T' :
            cout << " Appuyer sur la touche q et pas " << quit << " pour quitter " << endl;
            cin >> quit;
            break;
        default :
            cout << " Appuyer sur la touche q et pas " << quit << " pour quitter " << endl;
            cin >> quit;
            break;
    }
    if ( quit == 'q' ) return 0;
    else cout << "Dernier essai, Appuyer sur la touche q pour quitter " << endl;
    cin >> quit;
    return 0;
}
```



2.4.3 - Les Notions de Base C/C++ -> Les structures de contrôle : Tant que ... faire => while () {}



Tant que (expression vraie) faire
{bloc d 'instructions}

```
while (expression vraie) {  
    bloc d 'instructions }
```



2.4.3 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
    char quit = '\0';
    while ( quit != 'q' )
    {
        cout << " Bonjour ! " << endl;
        cout << " Appuyer sur la touche q pour quitter " << endl;
        cin >> quit;
    }
    return 0;
}
```



2.4.4 - Les Notions de Base C/C++ -> Les structures de contrôle : Répéter ... tant que ...=> do {} while ()



Répéter {bloc d 'instructions}
tant que (expression vraie)

```
do {bloc d 'instructions} while  
(expression vraie);
```



2.4.4 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
    char quit = '\0';
    do
    {
        cout << " Bonjour ! " << endl;
        cout << " Appuyer sur la touche q pour quitter " << endl;
        cin >> quit;
    }
    while ( quit != 'q' )
    return 0;
}
```



Pour (initialisation;condition;modification) faire {
 bloc d 'instructions}

```
for (initialisation;condition;modification) {  
    bloc d 'instructions}
```

2.4.5 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
    char quit = '\0';
    for ( ; quit != 'q'; )
    {
        for ( int i = 0 ; i < 5; i++)
            cout << " Bonjour ! " << endl;
        cout << " Appuyer sur la touche q pour quitter " << endl;
        cin >> quit;
        cout << endl;
    }

    return 0;
}
```



Pour (déclaration: expression) faire {
 bloc d 'instructions }

```
for (déclaration: expression) {  
    bloc d 'instructions}
```

2.4.6 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {0, 1, 2, 3, 4, 5}// C++11

    for (int &i : v) // accès par référence
    {
        std::cout << i << ' ';
        i *= 2;
    }

    std::cout << '\n';

    for (int i : v) // accès par valeurs
    {
        std::cout << i << ' ';
    }

    for (const int& i : v) // accès par référence valeur contante
    {
        std::cout << i << ' ';
    }

    std::cout << std::endl;;
    return 0;
}
```

2.4.7 - Les Notions de Base C/C++ -> Les structures de contrôle : Saut => goto



goto étiquette ;
étiquette :

goto étiquette ;
étiquette :

2.4.7 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
    char quit = '\0';

    reprise :

    cout << " Bonjour ! " << endl;
    cout << " Appuyer sur la touche q pour quitter " << endl;
    cin >> quit;
    if ( quit != 'q' ) goto reprise;
    cout << endl;

    return 0;
}
```



- ***Return[valeur]*** : permet de quitter immédiatement la fonction en cours.
- ***break*** : permet de passer à l'instruction suivant l'instruction *while*, *do*, *for* ou *switch* la plus imbriquée.
- ***continue*** : saute directement à la dernière ligne de l'instruction *while*, *do* ou *for* la plus imbriquée.

2.4.8 - Les Notions de Base C/C++ -> Les structures de contrôle : Exemple



```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
    char quit = '\0';

    while ( quit!='q' )
    {
        for (int i=0;i< 5;i++)
        {
            cout << " Bonjour ! : num => " << i << endl;
            cout << " Appuyer sur la touche 'o' pour quitter la boucle" << endl;
            cin >> quit;
            if ( quit=='o' ) break;
        }
        cout << " Appuyer sur la touche 'r' pour recommencer la boucle" << endl;
        cout << " Appuyer sur la touche 'q' pour quitter " << endl;
        cin >> quit;

        if ( quit=='r' ) continue;
        if ( quit!='q' ) return 0;
    }
    return 0;
}
```



Écrire un jeu dont le but est de trouver un nombre choisi par la machine compris entre -N et N

Exemple d'exécution :

> NombreMystere 10

Le nombre à trouver est compris entre -10 et 10

Entrer un nombre :

0

Trop petit

5

Trop grand

1

Bravo le chiffre est bien 1, vous avez gagné en 3 coups .

Voulez vous refaire une partie o[O] ? N

En C : srand() et rand();

>

En C++ : #include <random> https://fr.cppreference.com/w/cpp/numeric/random/uniform_int_distribution



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Une fonction est définie de la façon suivante :

```
type nom(type var1, type var2, ..., type varn)
{ instruction; ... ; return valeur; }
```

Remarque :

- En C une procédure est une fonction
- Pas de déclaration de fonction dans une autre fonction
- Possible appel d'une fonction dans une autre fonction
- Si une fonction ne renvoie pas de valeur, on lui donnera le type void
- Si elle n'attend pas de paramètre sa liste de paramètres sera void ou n'existera pas (C++) .
- Il n'est pas nécessaire de mettre une instruction return à la fin d'une fonction qui ne renvoie pas de valeur.

Exemple :

```
int mult(int i, int j)
{
    return i*j;
}
```

2.5.1 - Les Notions de Base C/C++ -> Les Fonctions : Surcharge des fonctions



```
int min(int i, int j)
{
    if ( i < j ) return i;
    return j;
}
```

```
float min (float i, float j)
{
    if ( i < j ) return i;
    return j;
}
```

```
int min(int i, int j, int k )
{
    int local = min(i,j) ;
    if ( local < k ) return local;
    return k;
}
```

```
float min (float i, float j, float k)
{
    double local = min(i,j) ;
    if ( local < k ) return local;
    return k;
}
```



```
inline int min(int i, int j)
{
    if ( i < j ) return i;
    return j;
}
```

Remarque :

- Une fonction inline ne peuvent pas être récursive.
- On ne peut pas faire de pointeur sur une fonction inline.



```
// Déclaration de fonction statique  
static int min(int i, int j);
```

```
// Définition de fonction statique  
static int max(int i, int j)  
{  
    if ( i > j ) return i;  
    return j;  
}
```

Si une fonction à uniquement un intérêt local à un fichier, le mot clé static, la rend visible uniquement dans ce fichier.



```
// Déclaration de la variable statique
static int j = 0;

// Définition de la variable statique dans une fonction
int add()
{
    static int i = 0; // variable non libérée à la fin de la fonction
    i++;
    return i;
}
```

La variable static rend la visibilité de la variable limitée au fichier la contenant ou fonction.



```
int main() /* Plus petit programme C/C++. */  
{  
    return 0;  
}
```

La fonction **main** est une fonction spéciale, lorsqu'un programme est chargé, son exécution commence par l'appel de celle-ci, elle marque le début du programme.

Remarque :
Elle ne peut pas être récursive.



La fonction lambda est une fonction inline créeé directement dans votre code et éventuellement passée directement dans les paramètres d'une de vos méthodes.

• Avantages

- La fonction est inline : **plus de performance.**
- La fonction est privée à votre code
- Pour des fonctions courtes : code plus concis (elle évite la création de fonctions)



[capture](parameters)->return-type{body}

```
[](int x, int y) -> int { int z = x + y; return z; }
```

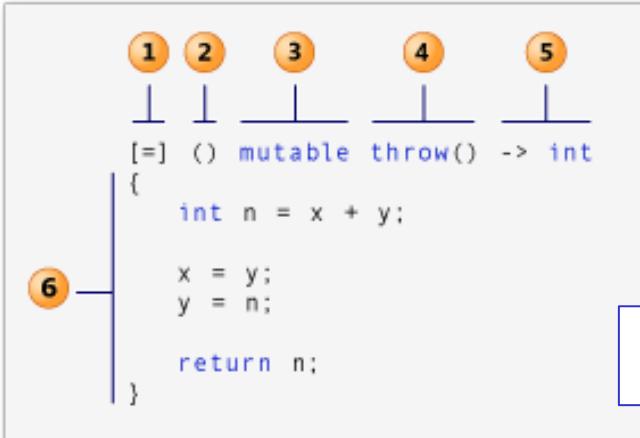
```
[](int x, int y) { return x + y; }
```

```
[](int& x) { ++x; }
```

```
[global_x]() { ++global_x; }
```

```
[global_x]{ ++global_x; }
```

2.5.6.2 - Les Notions de Base C/C++ -> Les Fonctions : Expressions lambda



```

auto p = [] () { cout << "Je suis une fonction lambda" << endl; };
p();
  
```

- Partie capture qui spécifie les symboles visibles dans le champ où la fonction est déclarée sera visible à l'intérieur du corps de la fonction. Une liste de symboles peut être adoptée comme suit:
 - [a,&b] où a est capturé par valeur et b est capturé par référence.
 - [this] capte le pointeur this
 - [&] captures all symbols by reference
 - [=] captures all by value
 - [] captures nothing
- liste des paramètres, comme dans fonctions nommées
- mutable** : permet à body de modifier les paramètres capturés par copie, et d'appeler leurs fonctions de membre non-const
- throw** : fournit la spécification d'exception ou la noexcept clause pour l'opérateur () du type de fermeture
- type de retour : s'il n'est pas présent il est implicite dans les énoncés de retour de fonction (ou void si elle ne retourne aucune valeur)
- corps de la fonction lambda

2.5.6.3 - Les Notions de Base C/C++ -> Les Fonctions

Expressions lambda : Exemple



```
vector<int> v = {30,-20, 40, -40,10,20};

std::sort(v.begin(), v.end(), myfunction); // avec bool myfunction(int a, int b) { return a < b; }
for (auto i: v) std::cout << i << " ";
std::cout << std::endl;

std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });

for (auto i: v) std::cout << i << " ";
std::cout << std::endl;
std::cout << std::endl;
```

-40 -20 10 20 30 40
40 30 20 10 -20 -40

```
int a = 2;
auto p1 = [&a] () { return a = a*a; }; // Retour implicite
auto p2 = [&a] () -> int { return a = a*a; }; // Retour explicite
auto p3 = [&a] () -> decltype(a) { return a = a*a; }; // Retour explicite

cout << "p1 = " << p1() << " a = " << a << endl;
cout << "p2 = " << p2() << " a = " << a << endl;
cout << "p3 = " << p3() << " a = " << a << endl;
```

p1 = 4 a = 2
p2 = 16 a = 4
p3 = 256 a = 16

2.5.6.4 — Les Notions de Base C/C++ -> Les Fonctions Expressions lambda : Exemple



```
int plus(const int& lhs, const int& rhs) { return lhs + rhs; }
```

```
auto plus2 = std::bind(&plus, std::placeholders::_1, 2);
std::cout << plus2(123) << std::endl;
```

```
auto plus_lambda2 = std::bind([](const int& lhs, const int& rhs)->int{ return lhs + rhs; }, std::placeholders::_1, 2);
std::cout << plus_lambda2(123) << std::endl;
```

125
125



2.5.6.5 - Les Notions de Base C/C++ -> Les Fonctions : Expressions lambda : Exemple



```
int i = 10;
int j = 200;
int res = 0;
auto divparam = [](const int& ip,const int& jp) throw(std::exception) { if ( ip == 0) throw std::exception(); return jp/ip;};
auto divvalue = [=]() throw(std::exception) { if ( i == 0) throw std::exception(); return j/i;};
auto divref = [&]() throw( std::exception ){ if ( i == 0) throw std::exception(); res = j/i ; return res;};
auto divmutable = [i,j,res]() mutable throw( std::exception ) -> int { if ( i == 0) throw std::exception(); res = j/i ; return res;};
auto divrefmutable = [i,j,&res]() mutable throw( std::exception ) -> int& { if ( i == 0) throw std::exception(); res = j/i ; return res;};

res=0;
std::cout << res << " ";
std::cout << "divparam => " << divparam(i,j) << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " ";
std::cout << " divvalue => " << divvalue() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " ";
std::cout << " divref => " << divref() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " ";
std::cout << " divmutable => " << divmutable() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " ";
std::cout << " divrefmutable => " << divrefmutable() << " ";
std::cout << res << std::endl;
res=0;
std::cout << res << " ";
std::cout << "[&]() mutable throw( std::exception ) -> int& => " <<
    ( [&]() mutable throw( std::exception ) -> int& { if ( i == 0) throw std::exception(); res = j/i ; return res;})() << " ";
std::cout << res << std::endl;
```

```
0 divparam => 20 0  
0 divvalue => 20 0  
0 divref => 20 20  
0 divmutable => 20 0  
0 divrefmutable => 20 20
```





2.5.6.6 - Les Notions de Base C/C++ -> Les Fonctions : Fonction anonyme lambda (C++11)



```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

int main()
{
    std::vector<int> c { 1,2,3,4,5,6,7 };
    int x = 5;
    c.erase(std::remove_if(c.begin(), c.end(), [x](int n) { return n < x; }), c.end());

    std::cout << "c: ";
    for (auto i: c) {
        std::cout << i << ' ';
    }
    std::cout << '\n';

    std::function<int (int)> func = [](int i) { return i+4; };
    std::cout << "func: " << func(6) << '\n';
}
```

Résultat :

```
c: 5 6 7
func: 10
```

<http://fr.cppreference.com/w/cpp/language/lambda>



D.Palermo

Programmation C++

Version 4.4 - 02/2020

Copyright : Yantra Technologies 2004-2020



Faire un programme qui calcule une suite jusqu'à ce que $|U_n - U_{n-1}| < \text{epsilon}$ ou $n < \text{NMax}$

Suite : $U_n = a + b/U_{n-1}$

Exemple d'exécution :

> Suite 5 3 1 10 0.001

Calcul de la suite $U_n = 5 + 3 * U_{n-1}$ avec $U_0 = 1$ et $NMAX = 10$
avec epsilon = 0.001

Résultat 5.54137 pour un nombre d'itérations n = 6

>



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



- Les **pointeurs** ont pour valeur une **adresse mémoire**
- Les **références** sont des identificateurs qui permettent de manipuler des notions introduites avec les pointeurs
- Les références n'existent qu'en C++.



Notion :

- Un objet manipulé est stocké dans sa mémoire.
- Une mémoire est constituée d'une série de « cases », où sont stockées des valeurs (variables ou instructions)
- Pour accéder au contenu de la case mémoire où l'objet est enregistré, il faut connaître l'emplacement en mémoire de l'objet à manipuler.

Cet emplacement est appelé **l'adresse de la case mémoire**, ou
l'adresse de la variable ou **l'adresse de la fonction**.

Toute case mémoire a une adresse unique.



Notation adresse

adresse x = &x

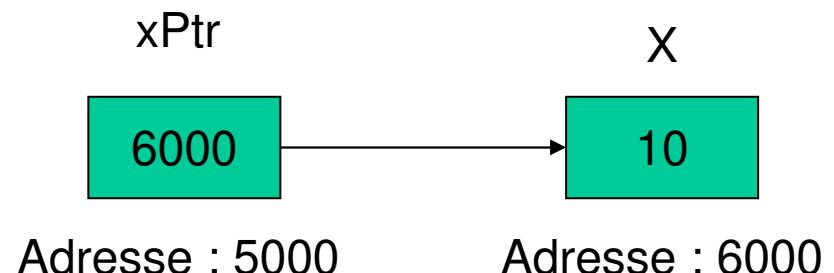


- Tout emplacement mémoire a une adresse
- Un pointeur est une variable qui contient une adresse mémoire
- On dit que le pointeur pointe sur cette adresse

```
int main ()
{
    int x = 10;
    int * xPtr; /* xPtr pointeur pointant sur type int */
    xPtr = &x;
    cout << &xPtr << ' ' << xPtr << ' ' << *xPtr << endl;
    cout << &x << ' ' << x << endl;
}
```

5000 6000 10
6000 10

Déclaration pointeur
*pointeur = type * p*
Récupération valeur pointée
** p*





L'allocation dynamique de mémoire permet de réserver de la mémoire (appelée encore *allocation*) pendant l'exécution d'un programme. La quantité de mémoire utilisée par le programme est variable.



Allocation dynamique : **malloc**

- signature de la fonction :
 - `void* malloc(int) ;`
- Prototype dans alloc.h
- Exemples :
 - `char* pc;`
 - `pc=(char*) malloc(10);`
 - `int * pi;`
 - `pi=(int*) malloc(sizeof(int)*16);`

Libération mémoire : **free**

- Libération par la fonction
 - `free(pointeur)`
- Évite l'encombrement mémoire par la destruction des pointeurs
- Exemples :
 - `free(pi);`
 - `free(pc);`



Allocation dynamique : `new`

- Allocation par les fonctions :
 - `new type ;`
 - `new type [taille] ;`
- Exemples :
 - `char* pc = new char;`
 - `int * pi = new int[16];`

Libération mémoire : `delete`

- Libération par les fonctions :
 - `delete pointeur;`
 - `delete [] pointeur ;`
- Exemples :
 - `delete pc ;`
 - `delete [] pi;`



```
int i = 0;  
int *pi = 0; // en C++11 int* pi=nullptr  
pi = &i;  
*pi = *(pi+1);
```

Attention $*\text{pi} + 1 \neq *(\text{pi} + 1)$

$$*\text{pi} + 1 == \text{pi}[0] + 1$$

$$*(\text{pi} + 1) == \text{pi}[1]$$



```
int i=1;  
int &ri = i;           // int & = synonyme  
ri = ri+i;
```

Il est possible de faire des références sur des valeurs numériques.

```
const int &ri=3; // Référence sur 3.  
int &error=4; // Erreur ! La référence n'est pas constante.
```



Passage de paramètre par valeur

```
double mult( double x1, double y1 )
{
    y1 = x1 * y1;
    return y1;
}

void main()
{
    double x = 2;
    double y = 3;
    double z = mult(x,y);
    /*en sortie z = 6 , x=2 , y=3*/
}
```

Passage de paramètre par variable

```
const double& mult(
    const double& x1, double& y1 )
{
    y1 = x1 * y1;
    return y1;
}

void main()
{
    double x = 2;
    double y = 3;
    double z = mult(x,y);
    /*en sortie z = 6 , x=2 ,
     Attention y a été modifié y=6*/
}
```

2.6.6 - Les Notions de Base C/C++ -> Les Pointeurs : Exemple : Façon C



```
inline double mult1 (const double* x, const double* y) // z = x * y
{
    return (*x)* (*y);
}

inline void mult1(const double* x , double *y ) // y *= x
{
    (*y)=(*x)* (*y);
}

int main() {
    double x = 5;
    double y = 2;
    double z= 0;
    z = mult1(&x,&y); // z == 10
    mult2(&x,&y); // y ==10

    double * x1 = new double(5);
    double * y1 = new double(2);
    double * z1 = new double(0);
    *z1 = mult1(x1,y1); // *z1 == 10
    mult2(x1,y1);      // *y1 ==10
}
```



2.6.6 - Les Notions de Base C/C++ -> Les Pointeurs : Exemple : Façon C++



```
inline double mult1 (cons double& x, const double& y) // z = x * y
{
    return x * y;
}

inline void mult1(const double& x , double & y ) // y *= x
{
    y=x* y;
}

int main() {
    double x = 5;
    double y = 2;
    double z= 0;
    z = mult1(x,y); // z == 10
    mult2(x,y); // y == 10

    double * x1 = new double(5);
    double * y1 = new double(2);
    double * z1 = new double(0);
    *z1 = mult1(*x1,*y1); // z1 == 10
    mult2(*x1,*y1);      // y1 ==10
}
```



2.6.6 - Les Notions de Base C/C++ -> Les Pointeurs : Exemple : Mixte



```

double& mult( const double& x, double& y )
{
    y = x * y;
    return y;      /* ou          return (y = x * y); */

    // return (x * y); Errreur
}

double mult( const double x, double* y )
{
    (*y) *= x;
    return *y;
}

int main() {

    double x=10,y=2;
    mult(x,y)=5; // ⇔ mult(x,y); y = 5;
    cout << "5 == " << y << endl;
    mult(2.0,y)=6;
    cout << "6 == " << y << endl;

    // mult(2.0,&y)=8; Errreur
    // mult(2.0,3)=8; Errreur
    return 0;
}

```



```
int const *pi;
```

```
const int *pi; /* pi est un pointeur sur un entier constant. */
```

```
int i=0;
```

```
int j=0;
```

```
pi = &i;
```

```
*pi = 4; // error
```

```
pi= &j;
```

```
int j;
```

```
int * const pj=&j; /* pj est un pointeur constant sur un entier non  
constant */
```

```
const int *pj[10]; /* pj est un tableau de 10 pointeurs d'entiers  
constants */
```



Ecrire une fonction « constructeur » qui prend un pointeur sur un double en argument et lui affecte une valeur.

Écrire une fonction « affiche » qui affiche un pointeur sur un double , son adresse et sa valeur.

Ecrire une fonction « destructeur » qui prend un pointeur sur un double en argument et libère la mémoire.



2.6.7 - Les Notions de Base C/C++ -> Les Pointeurs : Exercices



```
#include <iostream>

typedef double* ptrDouble;

void constructeur(ptrDouble dd,const unsigned & taille) { ACOMPLETER }
void afficher (const ptrDouble dd, const unsigned& taille) { ACOMPLETER }
void modifier (ptrDouble const dd, const unsigned& taille, const unsigned& index, const double& valeur)
{ACOMPLETER}
void destructeur (ptrDouble dd) { ACOMPLETER }
const double& get(const ptrDouble dd, const unsigned& taille, const unsigned& index ) { ACOMPLETER }
double& get(ptrDouble dd, const unsigned& taille, const unsigned& index) { ACOMPLETER }

void C_2_6b() {
    ptrDouble d1 = nullptr;
    unsigned t1=5;
    constructeur(d1,t1);
    afficher(d1,t1);
    modifier(d1,t1,2, 3.13589985);
    afficher(d1,t1);
    std::cout<< get(d1,t1,2) << std::endl;
    get(d1,t1,2) = 62.1;
    std::cout<< get(d1,t1,2) << std::endl;
    afficher(d1,t1);
    destructeur(d1);
    afficher(d1,t1);
}
```



Accès aux éléments d'un tableau par pointeur

```
int tableau[100];  
int *pi=tableau;
```

```
tableau[3]=5; /* Le 4ème élément est initialisé à 5 */  
*(tableau+2)=4; /* Le 3ème élément est initialisé à 4 */  
pi[5]=1; /* Le 6ème élément est initialisé à 1 */
```

```
f(int & t []);  
g(int* &t);
```



type (*identificateur) (paramètres);

Exemple :

```
int (*pf)(int, int);
```

/* pf est un pointeur de fonction */.

```
typedef int (*PtrFonct)(int, int);
```

```
PtrFonct pf;
```



std::function : Le template de classe std::function est un adaptateur générique de fonction

Exemple :

```
int (*pf)(int, int);  
std::function<int(int,int)> pf ;
```

```
using PtrFonct = std::function<int(int,int)> ;  
PtrFonct = pf;
```

<http://www.cplusplus.com/reference/functional/function/function/>
https://h-deb.clg.qc.ca/Sujets/TrucsScouts/intro_std_function.html

2.6.9 - Les Notions de Base C/C++ -> Les Pointeurs : Pointeur de fonctions (C++11) - Exemple



```
#include <functional>
#include <iostream>

struct Foo {
    Foo(int num) : num_(num) {}
    void print_add(int i) const { std::cout << num_+i << '\n'; }
    int num_;
};

void print_num(int i)
{
    std::cout << i << '\n';
}

int main()
{
    // store a free function
    std::function<void(int)> f_display = print_num;
    f_display(-9);

    // store a lambda
    std::function<void()> f_display_42 = []() { print_num(42); };
    f_display_42();

    // store the result of a call to std::bind
    std::function<void()> f_display_31337 = std::bind(print_num, 31337);
    f_display_31337();

    // store a call to a member function
    std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
    Foo foo(314159);
    f_add_display(foo, 1);
}
```

Résultat :

```
-9
42
31337
314160
```



- `std::shared_ptr<>` : pointeur intelligent avec sémantique de propriétaires partagés
- `std::weak_ptr<>` : référence faible à un objet géré par `std::shared_ptr`
- `std::unique_ptr<>` : pointeur intelligent avec sémantique de propriétaire unique



`std::shared_ptr` est un pointeur intelligent qui permet le partage d'un objet via un pointeur.

Plusieurs instances de `shared_ptr` peuvent posséder le même objet, et cet objet est détruit dans l'un des cas suivant :

- le dernier `shared_ptr` possédant l'objet est détruit
- le dernier `shared_ptr` possédant l'objet est assigné à un autre pointeur via `operator=()` or `reset()`.



2.6.10.1 - Les Notions de Base C/C++ -> Les Pointeurs : Pointeurs intelligents de bibliothèque standard (C++11) std::shared_ptr<>



```
1 // shared_ptr constructor example
2 #include <iostream>
3 #include <memory>
4
5 struct C {int* data;};
6
7 int main () {
8     std::shared_ptr<int> p1;
9     std::shared_ptr<int> p2 (nullptr);
10    std::shared_ptr<int> p3 (new int);
11    std::shared_ptr<int> p4 (new int, std::default_delete<int>());
12    std::shared_ptr<int> p5 (new int, [](int* p){delete p;}, std::allocator<int>());
13    std::shared_ptr<int> p6 (p5);
14    std::shared_ptr<int> p7 (std::move(p6));
15    std::shared_ptr<int> p8 (std::unique_ptr<int>(new int));
16    std::shared_ptr<C> obj (new C);
17    std::shared_ptr<int> p9 (obj, obj->data);
18
19    std::cout << "use_count:\n";
20    std::cout << "p1: " << p1.use_count() << '\n';
21    std::cout << "p2: " << p2.use_count() << '\n';
22    std::cout << "p3: " << p3.use_count() << '\n';
23    std::cout << "p4: " << p4.use_count() << '\n';
24    std::cout << "p5: " << p5.use_count() << '\n';
25    std::cout << "p6: " << p6.use_count() << '\n';
26    std::cout << "p7: " << p7.use_count() << '\n';
27    std::cout << "p8: " << p8.use_count() << '\n';
28    std::cout << "p9: " << p9.use_count() << '\n';
29
30 }
```

Output:

```
use_count:
p1: 0
p2: 0
p3: 1
p4: 1
p5: 2
p6: 0
p7: 2
p8: 1
p9: 2
```

Réf : <http://www.cplusplus.com/reference/memory>





`weak_ptr` fournit l'accès à un objet appartenant à une ou plusieurs instances de `shared_ptr` (travailler en collaboration), mais ne participe pas au décompte de références (aucune responsabilité associée).

2.6.10.2 - Les Notions de Base C/C++ -> Les Pointeurs : Pointeurs intelligents de bibliothèque standard (C++11) `std::weak_ptr<>`



```

1 // weak_ptr constructor example
2 #include <iostream>
3 #include <memory>
4
5 struct C {int* data;};
6
7 int main () {
8     std::shared_ptr<int> sp (new int);
9
10    std::weak_ptr<int> wp1;
11    std::weak_ptr<int> wp2 (wp1);
12    std::weak_ptr<int> wp3 (sp);
13
14    std::cout << "use_count:\n";
15    std::cout << "wp1: " << wp1.use_count() << '\n';
16    std::cout << "wp2: " << wp2.use_count() << '\n';
17    std::cout << "wp3: " << wp3.use_count() << '\n';
18    return 0;
19 }
```

Output:

```

use_count:
wp1: 0
wp2: 0
wp3: 1
```

Réf : <http://www.cplusplus.com/reference/memory>



`std::unique_ptr` est un pointeur intelligent qui :

- Garantie la propriété unique d'un objet à travers un pointeur, et
- détruit l'objet pointé lorsque le `unique_ptr` devient inaccessible.

Remarque : Remplace `auto_ptr`, qui est déconseillé.

2.6.10.3 - Les Notions de Base C/C++ -> Les Pointeurs : Pointeurs intelligents de bibliothèque standard (C++11) `std::unique_ptr<>`



```

1 // unique_ptr constructor example
2 #include <iostream>
3 #include <memory>
4
5 int main () {
6     std::default_delete<int> d;
7     std::unique_ptr<int> u1;
8     std::unique_ptr<int> u2 (nullptr);
9     std::unique_ptr<int> u3 (new int);
10    std::unique_ptr<int> u4 (new int, d);
11    std::unique_ptr<int> u5 (new int, std::default_delete<int>());
12    std::unique_ptr<int> u6 (std::move(u5));
13    std::unique_ptr<void> u7 (std::move(u6));
14    std::unique_ptr<int> u8 (std::auto_ptr<int>(new int));
15
16    std::cout << "u1: " << (u1?"not null":"null") << '\n';
17    std::cout << "u2: " << (u2?"not null":"null") << '\n';
18    std::cout << "u3: " << (u3?"not null":"null") << '\n';
19    std::cout << "u4: " << (u4?"not null":"null") << '\n';
20    std::cout << "u5: " << (u5?"not null":"null") << '\n';
21    std::cout << "u6: " << (u6?"not null":"null") << '\n';
22    std::cout << "u7: " << (u7?"not null":"null") << '\n';
23    std::cout << "u8: " << (u8?"not null":"null") << '\n';
24
25    return 0;
26 }

```

Output:

```

u1: null
u2: null
u3: not null
u4: not null
u5: null
u6: null
u7: not null
u8: not null

```



Les pointeurs sont très utilisés en C/C++, mais ils sont très dangereux, quelques règles de survie :

Règle 1 : TOUJOURS INITIALISER LES POINTEURS QUE VOUS UTILISEZ.

Règle 2 : VÉRIFIEZ QUE TOUTE DEMANDE D'ALLOCATION MÉMOIRE A ÉTÉ SATISFAITE.

Règle 3 : LORSQU'ON UTILISE UN POINTEUR, IL FAUT VÉRIFIER S'IL EST VALIDE.



2.6.12 - Les Notions de Base C/C++ -> Les Pointeurs : Pointeurs intelligents de bibliothèque standard (C++11) Exercices



```
#include <iostream>
#include <memory>

typedef std::shared_ptr<double> ptrStdDouble;

void constructeur(ptrStdDouble dd,const unsigned & taille) { ACOMPLETER }
void destructeur (ptrStdDouble dd) { ACOMPLETER }
void afficher (const ptrStdDouble dd, const unsigned& taille) { ACOMPLETER }
void modifier (ptrStdDouble const dd, const unsigned& taille, const unsigned& index, const double& valeur) { ACOMPLETER }
const double& get(const ptrStdDouble dd, const unsigned& taille, const unsigned& index) { ACOMPLETER }
double& get( ptrStdDouble dd, const unsigned& taille, const unsigned& index) { ACOMPLETER }

void C_2_6c() {
    ptrStdDouble d1 = nullptr;
    unsigned t1=5;
    constructeur(d1,t1);
    afficher(d1,t1);
    modifier(d1,t1,2, 3.13589985);
    afficher(d1,t1);
    std::cout<< get(d1,t1,2) << std::endl;
    get(d1,t1,2) = 62.1;
    std::cout<< get(d1,t1,2) << std::endl;
    afficher(d1,t1);
    destructeur(d1);
}
```





Tableau caractérisé par

- sa taille
- ses éléments

Tableau à une dimension

- Déclaration : **Type nom[dim]; /* réservation de dim places de grandeur Type */**
- Exemples
 - int Tab[10];
 - float vecteur[20];

Tableau à deux dimensions

- Déclaration : **type nom[dim1][dim2];**
 - int compteur[4][5];
 - float nombre[2][10];
- Utilisation : nom[indice1][indice2];
 - compteur[3][4]=3;
 - printf("%d",compteur[1][2]);



On peut initialiser au moment de la déclaration

- Exemple
 - `int liste[10]={1,2,45,5,67,120,32,48,3,10};`
 - `int tab[2][3]={{1,4,8},{8,5,3}}`

 - `for(int &x: liste) { x*=2; } (C++11)`



Déclaration d'un pointeur (adresse du premier élément) + allocation de l'espace mémoire
On peut donc déclarer nous-même un pointeur et allouer la mémoire 'manuellement'

```
int *tab = new int[5];
```

Avantage : allocation de l'espace mémoire au moment de l'exécution, attention ne pas oublier de détruire la mémoire:

```
delete [] tab; // ATTENTION n'est pas équivalent à delete tab;
```

Il est possible de pointer sur un élément particulier d'un tableau :

```
int tab[5];
int *ptr = &tab[2]; /* ptr pointe sur le 3eme élément du tableau */
```

Un nom de tableau utilisé dans une expression est converti en une adresse du début de ce tableau :

```
int tab[5];
int *ptr = tab; /* équivalent à : "ptr = &tab[0]" */
```

2.7.2 - Les Notions de Base C/C++ > Les Tableaux : Extensions de la bibliothèque standard -> `<array>`



Fournit des méthodes pour la création, la manipulation, la recherche ainsi que le tri des tableaux

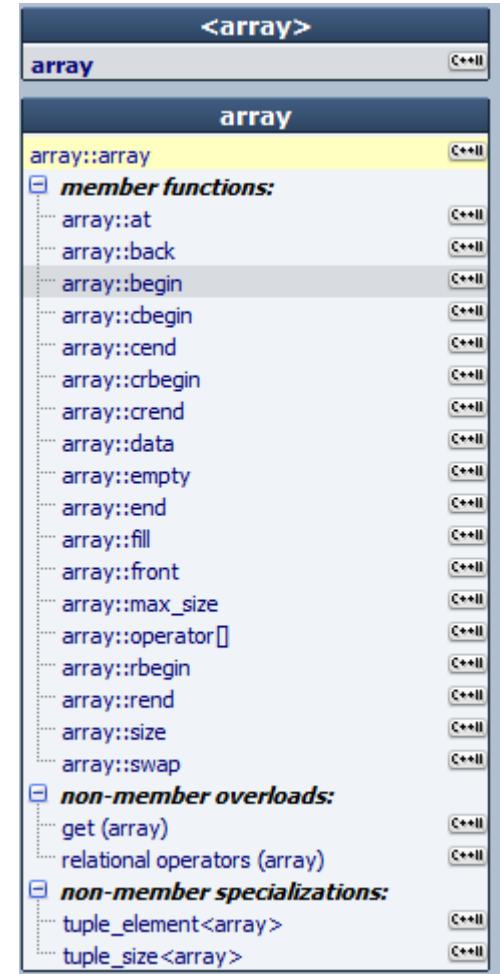
Example

```

1 // array::operator[]
2 #include <iostream>
3 #include <array>
4
5 int main ()
6 {
7     std::array<int,10> myarray;
8     unsigned int i;
9
10    // assign some values:
11    for (i=0; i<10; i++) myarray[i]=i;
12
13    // print content
14    std::cout << "myarray contains:";
15    for (i=0; i<10; i++)
16        std::cout << ' ' << myarray[i];
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```



<http://www.cplusplus.com/reference/array/array/>



Toutes les commandes du préprocesseur commencent :

- en début de ligne,
- par un signe dièse (#).

#include :

```
#include "fichier" ou #include <fichier>
```

```
#define POSIX_VERSION 1001 ( en C++ const int POSIX_VERSION = 1001; )
```

Le préprocesseur définit un certain nombre de constantes :

- __LINE__ : donne le numéro de la ligne courante
- __FILE__ : donne le nom du fichier courant
- __DATE__ : renvoie la date du traitement du fichier par le préprocesseur
- __TIME__ : renvoie l'heure du traitement du fichier par le préprocesseur
- __func__ : donne le nom de la fonction courante
- __cplusplus : définie uniquement dans le cas d'une compilation C++



Compilation conditionnelle :

```
#ifdef identificateur
```

```
...
```

```
#endif
```

```
#ifndef NAMEFILE_H  
#define NAMEFILE_H  
  
/* Contenu de votre fichier .h */  
  
#endif
```

d'autres directives de compilation conditionnelle :

- #ifndef (if not defined ...)
- #elif (sinon, si ...)
- #if (si ...)

Option de compilation :

Des options de compilation permettent de définir ou non des constantes symbolique utilisées par le préprocesseur et ainsi d'orienter ses traitements.

- D nom permet de définir la constante symbolique nom.
- D nom=definition permet en plus, lui donner la valeur definition.
- U nom permet d'annuler la définition de la constante symbolique nom.



autres commandes :

- # : ne fait rien (directive nulle) ;
- #error message : permet de stopper la compilation en affichant le message d'erreur donné en paramètre ;
- #line numéro [fichier] : permet de changer le numéro de ligne courant et le nom du fichier courant lors de la compilation ;
- #pragma texte : permet de donner des ordres spécifiques à une implémentation du compilateur

2.8.3 - Les Notions de Base C/C++ -> Les Directives de Précompilation : Macro



```
#if defined(HAVE_DIRENT_H) && defined(HAVE_SYS_TYPES_H)
#include <dirent.h>
#include <sys/types.h>
#else
/* Arret : dirent.h et sys/types.h non trouves sur ce système */
#error "Readdir non implemente sur cette plateforme"
#endif
```

https://fr.wikiversity.org/wiki/Introduction_au_langage_C/Le_processeur



```
#define macro(paramètre[, paramètre [...]]) définition
```

Exemple :

```
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)<(y)?(x):(y))
#define CHAINE(s) #s // transforme son argument en chaîne de caractères.
#define NOMBRE(chiffre1,chiffre2) chiffre1##chiffre2 // permet de construire un
nombre à deux chiffres.
```

2.8.3 - Les Notions de Base C/C++ -> Les Directives de Précompilation : Macro



```
int x1 = 1;
int x2 = 2;

/* Exemple de macro utilisant la concatenation de variables */
#define debug(s, t) \
(void)printf("x" # s "= %d, x" # t "= %d\n", \
    x ## s, x ## t)

(void)puts("\nCreation de variable par concatenation de chaine");
debug(1, 2);
```

https://fr.wikiversity.org/wiki/Introduction_au_langage_C/Le_processeur



```
#ifndef DEBUG
#define TRACE1(msg) {}
#else
#include <iostream>
#define TRACE1(msg) { \
std::cerr << __FILE__ <<< [ << << __LINE__ <<< ] >> \
<< #msg << std::endl; \
std::cerr.flush(); \
}
#endif
```



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Types particuliers composés d'un ensemble de types

```
typedef struct {  
    types; } nomdutype;
```

Exemple

- **typedef struct { char* nom; char* prenom;} Personne;// en C et C++**
- **struct Personne {....};// en C++**
- **Personne toto;**



- Syntaxe : **objet.champ**
 - Exemple : **personne toto ;**
 - **toto.nom = new char[5];**
 - **strcpy(toto.nom, « david »); // toto.nom = « david »;**
 - **cout << toto.nom << endl;**
- Pas de protection sur les champs

https://cpp.developpez.com/cours/cpp/?page=page_5#LV



Pointeurs sur des structures

- allocation dynamique de structures
- modification d'une structure dans une fonction
- Exemple : **Personne* toto = new Personne;**
(*toto).age=12;

Notation équivalente : ->

- Exemple : **(*toto).age** **toto->age**

2.9.3 - Les Notions de Base C/C++ -> Le type struct en C : Exemple



```
struct Client
{
    unsigned char Age;
    unsigned char Taille;
};

struct Client Jean, Philippe;
Client Christophe; // Valide en C++ mais invalide en C
```

```
struct
{
    unsigned char Age;
    unsigned char Taille;
} Jean;
```

https://fr.wikibooks.org/wiki/Programmation_C-C%2B%2B/Structures_et_types

2.9.3 - Les Notions de Base C/C++ -> Le type struct en C : Exemple



```
#include <iostream>

const unsigned NPOINTS = 5;

struct Point
{
    int num;
    float x;
    float y;
};

int main()
{
    Point courbe[NPOINTS];
    unsigned i;

    for(i=0; i<NPOINTS; i++)
    {
        std::cout << "numero : ";
        std::cin >> courbe[i].num;
        std::cout << "x : ";
        std::cin >> courbe[i].x;
        std::cout << "y : ";
        std::cin >> courbe[i].y;
    }

    std::cout << "**** structure fournie ****" << std::endl;
    for (i=0; i<NPOINTS; i++)
        std::cout << "numero=" <<courbe[i].num<<" x="<< courbe[i].x << " y="<< courbe[i].y << std::endl;
    return 0;
}
```



Syntaxe :

```
union <nom de l'union>
{<type de donnée> <nom de la variable>;
 ::<type de donnée> <nom de la variable>;
};
```

Description:

Le type union permet de définir des variables qui occupent le **même emplacement mémoire**.

La taille d'une union est celle de la donnée la plus grande de cette union.

L'écriture dans une variable **écrase la valeur** des autres variables de l'union.

Il faut utiliser le sélecteur d'enregistrement (.) pour accéder aux éléments d'une union.

Remarque:

Les unions, contrairement aux structures, sont assez peu utilisées, sauf en programmation système où l'on doit pouvoir interpréter des données de différentes manières selon le contexte.

Depuis C++11 accepte les type complexe,



union Nombre

```
{  
    int i;  
    double d;  
    char cNombre[12];  
};
```

Nombre nb;

nb.d = 2.3



Syntaxe :

```
enum [<nom de l'énumération>] {<nom d'une constante> [= <valeur>], ...} [<liste de variables>];
```

Description:

Le mot clé enum permet de définir un ensemble de constantes de type entier(int).

Une énumération fournit des identificateurs mnémoniques pour un ensemble de valeurs entières et finies

Une variable énumérée ne peut se voir affecter qu'un de ses énumérateurs(<constante>).

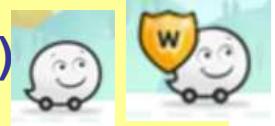
En l'absence d'une <valeur>, la première constante prend la valeur zéro.

Toute constante sans <valeur> sera augmentée d'un par rapport à la constante précédente.

Exemple :

```
enum jour { Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche};  
enum jour { Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche} variable_jour;  
enum { Lundi=1, Mardi=2, Mercredi=3, Jeudi=4, Vendredi=5, Samedi=6, Dimanche=7};
```

<http://carl.seleborg.free.fr/cpp/cours/chap2/structures.html>



Syntaxe :

enum class name : type attr { enumerator = constexpr , enumerator = constexpr , ... None}

enum struct name : type attr { enumerator = constexpr , enumerator = constexpr , ... None}

```
enum class Couleur {
    Noir, Blanc, Rouge, Jaune, Bleu, Orange, Vert, Violet
};

Couleur une_couleur { Couleur::Rouge }; // initialisation
une_couleur = Couleur::Bleu;           // affectation
```



2.11 - Les Notions de Base C/C++ -> Le type enum (C++11)



```
#include <iostream>
enum color { red,yellow,green = 20,blue };

enum class altitude : char {
    high='h',
    low='l'
};
enum { d, e, f=e+2 };
int main()
{
    color col = red;
    altitude a;
    a = altitude::low;

    std::cout << "red = " << col << " blue = " << blue << '\n'
        << "a = " << static_cast<char>(a) << '\n' // ((char)a)
        << "f = " << f << '\n';
}
```





Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020

2.12 - Les Notions de Base C/C++ -> Le mot clé : auto - définition



```
auto i = 27;
auto d = 3.4;
auto f = 3.4f;

cout << "i : " << typeid(i).name() << endl;
cout << "d : " << typeid(d).name() << endl;
cout << "f : " << typeid(f).name() << endl;
```

i	:	i
d	:	d
f	:	f

auto : permet de déduire le type à la compilation et indique au compilateur d'utiliser l'expression d'initialisation d'une variable déclarée, ou d'un paramètre d'expression lambda, pour déduire son type.

- Le mot clé **auto** est un moyen simple de déclarer une variable qui a un type complexe
- La déduction du type **auto** est identique à ceux des template



- **Robustesse** : si le type de l'expression est modifié, y compris quand un type de retour de fonction est modifié, cela fonctionne. Les variables auto doivent être initialisées.
- **Performances** : vous êtes certain qu'il n'y aura aucune conversion implicites silencieuses.
- **Usage** : vous n'avez pas à vous soucier des fautes de frappe ni des problèmes liés à l'orthographe du nom de type.
- **Efficacité** : votre codage peut être plus efficace.

2.12 - Les Notions de Base C/C++ -> Le mot clé :: auto - inconvénient



Les types proxy (invisible) peuvent conduire auto à une mauvaise déduction du type.

```
class ImageReel {
public:
    ImageReel() {
        std::cout << "nom de la fonction : " << __func__ << "()\\t" << typeid (*this).name() << std::endl;
    }
    ImageReel(const ImageReel&) {
        std::cout << "nom de la fonction : " << __func__ << "(const ImageReel&)" << typeid (*this).name() << std::endl;
    }
};

class ImageProxy {
public:

    ImageProxy(ImageReel&) {
        std::cout << "nom de la fonction : " << __func__ << "(ImageReel&)" << typeid (*this).name() << std::endl;
    }
};
```

```
nom de la fonction : ImageReel()      9ImageReel
nom de la fonction : ImageProxy(ImageReel&)    10ImageProxy

nom de la fonction : ImageReel(const ImageReel&)      9ImageReel
nom de la fonction : ImageProxy(ImageReel&)    10ImageProxy
```

```
ImageReel img;
ImageProxy imgproxy = img;
sautligne();
auto imgautoerr = img;
sautligne();
auto imgauto = static_cast<ImageProxy> (img);
```



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



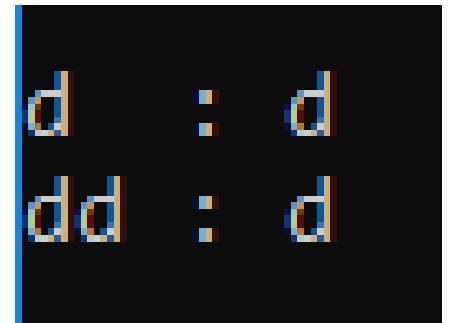
Copyright : Yantra Technologies 2004-2020



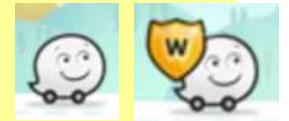
- decltype permet de typer une variable à partir du type d'une autre variable

```
auto d = 5.5;
decltype(d) dd;

cout << "d : " << typeid (d).name() << endl;
cout << "dd : " << typeid (dd).name() << endl;
```



d : d
dd : d



constexpr : expressions constantes généralisées

Une fonction est une fonction expression constante si :

- Elle renvoie une valeur (le type void n'est pas autorisé)
- Le corps de la fonction est composé d'une seule instruction, l'instruction return : return expr;
- Elle est déclarée avec le mot clé constexpr.

Une variable ou un membre de données est une valeur expression constante si :

- Il est déclaré avec le mot clé constexpr
- Elle ou il est initialisé avec une expression constante ou une rvalue construite par un constructeur d'expression constante avec des arguments d'expression constante.



```
constexpr int add(int x, int y) {  
    return x + y;  
}
```

```
constexpr double var = 2 * add(3, 4);
```

```
Int table[2 * add(3, 4)];
```



2.14 - Les Notions de Base C/C++ -> constexpr (C++11)



```
constexpr int fac(int n)
{
    return n == 1 ? 1 : n*fac(n - 1);
}
```

```
class Carre {
public:

    constexpr Carre(double d):a_cote(d) {}

    void afficher() const {
        std::cout << "Carre de cote : " << this->a_cote << std::endl;
        std::cout << "perimetre      : " << this->perimetre() << std::endl;
    }

    constexpr double perimetre() const { return this->a_cote * 2; }

private:
    double a_cote = 0.0;
};
```

```
constexpr Carre a2(5.5);
a2.afficher();

auto start = std::chrono::steady_clock::now();
constexpr auto x = fac(12);
auto stop = std::chrono::steady_clock::now();
auto dur = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
std::cout << "fac(12)=" << x << " a durer " << static_cast<double> (dur.count()) << " milli seconds!" << std::endl;
```

```
Carre de cote : 5.5
perimetre      : 11
fac(12)=479001600 a durer 0 milli seconds!
```





- ***lvalues*** (*left-value*) : variables constantes ou non
- ***rvalues*** (*right-value*) : valeurs temporaires littérales ou expressions
- ***lvalue-reference*** : n'acceptent que des *lvalues*
- ***rvalue-reference*** : n'acceptent que des *rvalues*. Mise en place avec l'opérateur **&&**, les variables ne sont jamais constantes
- **universal reference** : variable qui sera définie lors de l'instanciation du template. La substitution du type définira si c'est une ***lvalue-reference*** ou une ***rvalue-reference*** (opérateur **&&**)



&& permet d'avoir une référence sur une rvalue (objet temporaire)

```
std::string getName () { return "Alex"; }
```

```
const std:: string& name = getName(); // ok
```

```
string& name = getName(); // KO
```

```
const std:: string&& name = getName(); // ok
```

```
std:: string&& name = getName(); // ok
```



```
enum Sexe { INCONNUE=0,MASCULIN=1,FEMININ=2};

struct Personne {
    int numero;
    char nom[10];
    Sexe sexe;
};
```

Écrire les fonctions nommées suivantes :

- **créer** permettant de créer un pointeur de la structure Personne
- **détruire** permettant de détruire un pointeur de la structure Personne
- **initialiser** permettant d'initialiser les champs de la structure Personne
- **afficher** permettant d'afficher les champs de la structure Personne



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



3.1 - Classes

3.2 - Surcharge des opérateurs

3.3 – Membre static

3.4 - Fonction et classes amies

3.5 - Méthodes constantes

3.6 - Héritages

3.7 - Polymorphisme

3.8 - Le traitement des exceptions

3.9 - Les espaces de nom

3.10 - Les patrons



3.1.1 - Déclaration

3.1.2 - Définition

3.1.3 - Encapsulation

3.1.4 - Constructeur/Destructeur

3.1.5 - Délégation de constructeurs (C++11)

3.1.1 - Les Classes et les Objet en C++ -> Classes : Déclaration



```
class NomClass
{
    public :
        [ type attribut;
          type attribut;
          ... ]
        [ méthode ;
          méthode ;
          ...]
    protected :
        [ type attribut;
          type attribut;
          ... ]
        [ méthode ;
          méthode ;
          ...]

    private :
        [ type attribut;
          type attribut;
          ... ]
        [ méthode ;
          méthode ;
          ...]
};

};
```

3.1.1 Les Classes et les Objets en C++

-> Classes : Exemple



```
#ifndef VOITURE_HPP
#define VOITURE_HPP

class Voiture {
public :
    Voiture();
    Voiture(const Voiture& chaine);
    Voiture( const char * chaine );           // constructeur
    ~Voiture();                                // destructeur

    void afficher() const;                     // méthode constante
    const double& getVitesse() const;          // méthode constante
    void setVitesse(const double& );
    const std::string& getImmatriculation() const; // méthode constante
    void setImmatricualtion (const std::string& ); // méthode

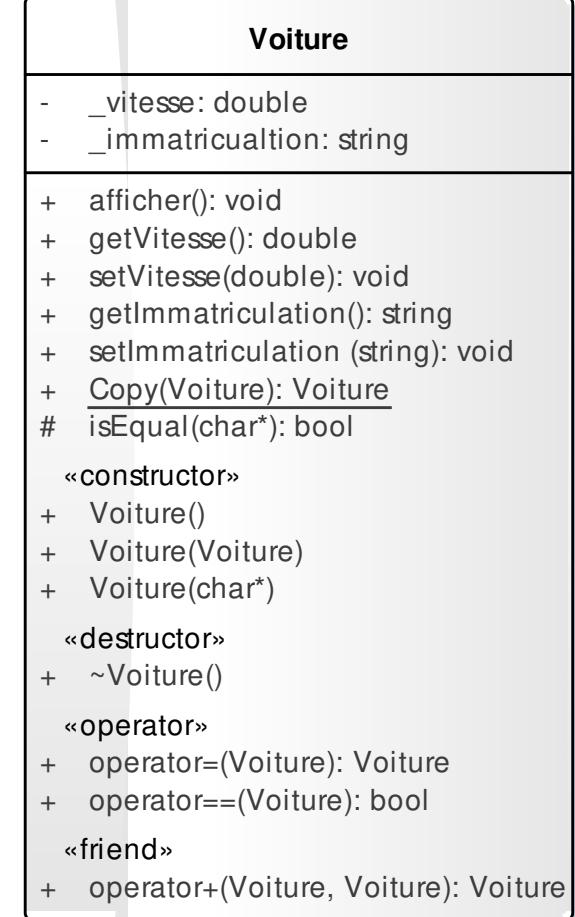
    Voiture& operator=(const Voiture& chaine) ; // opérateur d'affectation
    bool operator==(const Voiture& chaine ); // opérateur d'égalité

    friend Voiture operator+(Voiture& chaine1,Voiture& chaine2);
    static Voiture Copy(const Voiture& chaine) ;

protected :
    bool isEqual(const char * chaine );
private :
    double _vitesse;
    std::string _immatriculation;
};

#endif
```

class Cpp





3.1.1 Les Classes et les Objets en C++ -> Classes : Exemple



```
#ifndef ChaineCaractere_hpp
#define ChaineCaractere_hpp

class ChaineCaractere
{
public :
    ChaineCaractere();
    ChaineCaractere(const ChaineCaractere& chaine);
    ChaineCaractere( const char * chaine ); // constructeur
    ~ChaineCaractere(); // destructeur

    void afficher() const; // méthode constante
    const char* getChaine() const; // méthode constante
    void setChaine(const char* chaine); // méthode

    // opérateur d'affectation
    ChaineCaractere& operator=(const ChaineCaractere& chaine) ;
    // opérateur d'égalité
    bool operator==(const ChaineCaractere& chaine );
    friend ChaineCaractere operator+(ChaineCaractere& chaine1,
                                    ChaineCaractere& chaine2);
    static ChaineCaractere Copy(const ChaineCaractere& chaine) ;

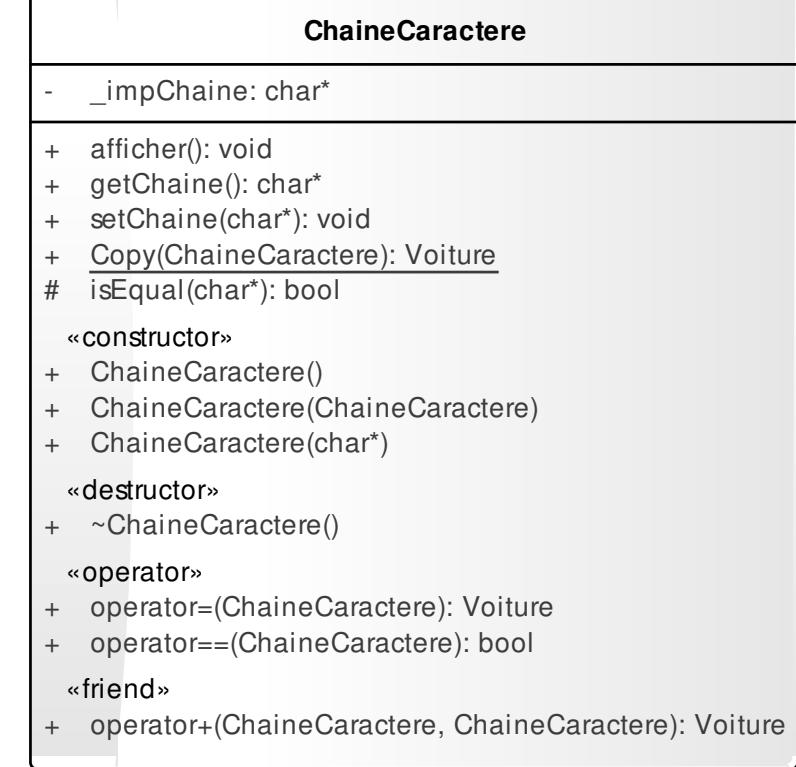
protected :
    bool isEqual(const char * chaine );
private :
    char * _impChaine;
};

#endif
```



D.Palermo

class Cpp





```
type NomClasse::methode(paramètres)
{
    /* Définition de la méthode. */
}
```

3.1.2 - Les Classes et les Objets en C++

-> Classes : Exemple



```
#ifndef CARTE_HH
#define CARTE_HH

namespace Exemple {

class Carte {
public:
    void afficher() const;
    void setValeur(int i);

private:
    int valeur;
};

#endif
```

class Cpp



Carte.hh

```
#include "Carte.hh"
#include <iostream>

using namespace std;

namespace Exemple {

void Carte::afficher() const { cout << "Carte => " << this->valeur << endl; }

void Carte::setValeur(int i) { this->valeur = i; }

}
```

Carte.cpp



Yantra Technologies

3.1.2 - Les Classes et les Objet en C++ -> Classes : Exemple



```
#ifndef CARTE_HH
#define CARTE_HH
namespace Exemple {
    class Carte {
        public:
            void afficher() const;
            void setValeur(int i);

        private:
            int valeur;
    };
}
#endif
```

Carte.hh

```
#include "Carte.hh"

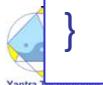
int main() {
    using Exemple::Carte;
    Carte carteObjet;
    carteObjet.afficher();
    carteObjet.setValeur(12);
    carteObjet.afficher();
    return 0;
}
```

```
#include "Carte.hh"

int main() {
    using namespace Exemple;
    Carte* carteObjet = new Carte();
    carteObjet->afficher();
    carteObjet->setValeur(12);
    carteObjet->afficher();
    delete carteObjet;
    return 0;
}
```

```
#include "Carte.hh"
using namespace Exemple;

int main() {
    const Carte carteObjet;
    carteObjet.afficher();
    // erreur carteObjet.setValeur(12);
    carteObjet.afficher();
    return 0;
}
```



Yantra Technologies

3.1.2 - Les Classes et les Objet en C++ -> Classes : Exemple



```
#ifndef CARTE_HH
#define CARTE_HH
namespace Exemple {

    class Carte {
        public:

            // Méthodes implicites
            Carte();
            Carte(const Carte&);
            ~Carte();

            Carte& operator=(const Carte&);
            bool operator==(const Carte&) const;

            void afficher() const ;
            void setValeur(int i);

        private:
            int valeur;
    };
}

#endif
```



```
#include "ChaineCaractere.hpp"
#include <iostream>
#include <stdio.h>

using namespace std;

void ChaineCaractere::afficher() const
{ cout << "ChaineCaractere => " << this->_impChaine << endl; }

const char* ChaineCaractere::getChaine() const
{ return this->_impChaine; }

bool ChaineCaractere::isEqual(const char * chaine )
{ return strcmp(chaine,this->_impChaine) ==0 ; }
```

ChaineCaractere.cpp

3.1.2 - Les Classes et les Objets en C++ -> Classes : Exemple



```
class A
{
public:
    int getValue() const { return this->value; } // définition au sein de la classe
    void setValue(int value);
    void print() const;
private:
    int value;
};
```

```
void A::setValue(int value)
{
    this->value = value;
}

void A::print() const // définition en dehors de la classe (non inline)
{
    std::cout << "Value=" << this->value << std::endl;
}
```

https://fr.wikibooks.org/wiki/Programmation_C%2B%2B/Les_classes#Les_fonctions_membres



- **public** : les accès sont libres
- **private** : les accès sont autorisés dans les fonctions de la classe seulement
- **protected** : les accès sont autorisés dans les fonctions de la classe et de ses descendantes. Le mot clé **protected** n'est utilisé que dans le cadre de l'héritage des classes.



```
#include "ChaineCaractere.hpp"
```

```
int main(int argc, char *argv[])
{
    ChaineCaractere chaine("Bonjour");
    chaine.afficher();
    // chaine isEqual("Bonjour"); Méthode protected ne peut pas être appelée
    // chaine._impChaine ; Attribut privé ne peut pas être appelé
    return 0;
}
```



- Un Constructeur porte le même nom que la classe
- Un Destructeur porte le nom de la classe avec ~ devant
- Un Constructeur/Destructeur n'a aucun type de retour

3.1.4 - Les Classes et les Objet en C++ -> Classes : Exemple



```
class ChaineCaractere
{
public :
    ChaineCaractere(const ChaineCaractere& chaine);
    ChaineCaractere( const char * chaine ="" );
    ~ChaineCaractere();
    ...
};
```

ChaineCaractere.hh

```
ChaineCaractere::ChaineCaractere(const ChaineCaractere& chaine) {
    this->_impChaine = new char[ strlen(chaine._impChaine) +1];
    strcpy(this->_impChaine,chaine._impChaine);
}

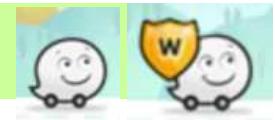
ChaineCaractere::ChaineCaractere( const char * chaine ) {
    this->_impChaine = new char[ strlen(chaine) +1];
    strcpy(this->_impChaine,chaine);
}

ChaineCaractere::~ChaineCaractere() {
    delete [] this->_impChaine;
}
```

ChaineCaractere.cpp



3.1.4 - Les Classes et les Objets en C++ -> Classes : Exemple / Exercice



class PForm

Triangle

```
# a_cote_a: int
# a_cote_b: int
# a_cote_c: int

+ Triangle(int, int, int)
+ Triangle(Triangle&)
+ ~Triangle()
+ calcule aire(): double {query}
+ afficher(): void {query}
+ getA(): int& {query}
+ getB(): int& {query}
+ getC(): int& {query}
+ setA(int): void
+ setB(int): void
+ setC(int): void
```

namespace PForme

class Triangle

public:

```
Triangle(int a = -1, int b = -1, int c = -1);
Triangle(const Triangle& tr);
```

```
~Triangle();
```

```
double calcule aire() const;
void afficher() const;
```

```
const int& getA() const;
const int& getB() const ;
const int& getC() const;
```

```
void setA(int a);
void setB(int b);
void setC(int c);
```

protected:

```
int a_cote_a;
int a_cote_b;
int a_cote_c;
```

Triangle.hh



D.Palermo

Programmation C++

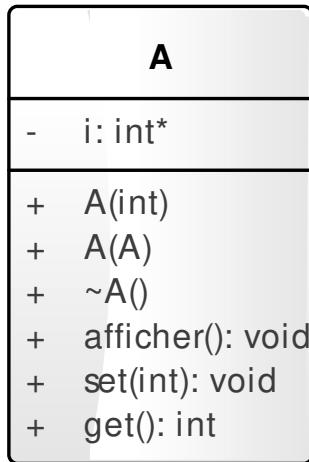
Version 4.4 - 02/2020

Copyright : Yantra Technologies 2004-2020

3.1.4 - Les Classes et les Objets en C++ -> Classes : Exemple / Exercice



class Cpp



```
#ifndef A_H
#define A_H

class A {
public:
    A(int i = 2);
    A(const A&);
    ~A();
    void afficher() const;
    void set(int);
    const int& get() const;
private:
    int *i;
};

#endif
```

```
#include "A.h"
int main() {
    A a = 2; // === A a(2); != a = 2;
    const A b = a;
{
    A *c = new A[5];
    delete [] c;
}
for (int i=0;i < 5;i++){
    A c[5];
    char buff[256];
}
a.afficher();
b.afficher();
a.set(4);
b.set(3);

A.h l.afficher();
l.afficher();
return 0;
}
```



C++11 introduit la possibilité de déléguer des constructeurs => les constructeurs de classe peuvent être invoqués au sein d'autres constructeurs de la même classe.

```
class B {  
    B(int newA) : a(newA) { ...  
    }  
  
public:  
    B() : B(15) { ... }  
  
public:  
    int a;  
};
```



3.1.4 - Les Classes et les Objets en C++

-> Classes : Exercices : Jeu de carte (1)



class Domain Model

Carte

```
- _couleur: Couleur  
- _valeur: std::string  
  
+ Carte(Couleur, std::string&)  
+ Carte(Carte&)  
+ ~Carte()  
+ setType(Couleur): void  
+ setValeur(std::string&): void  
+ afficher(): void {query}  
+ equal(Carte&): bool {query}  
+ affecter(Carte&): void
```

«Enumeration»

Couleur

```
PIQUE  
COEUR  
CARREAU  
TREFLE
```





3.1.4 - Les Classes et les Objet en C++

-> Classes : Exercices : Jeu de carte (1)



```
#include <iostream>
#include "Carte.h"

using namespace std;

//enum Couleur{ PIQUE, COEUR, CARREAU, TREFLE};

int main()
{
    cout << "Jeu de carte" << endl;
    Carte c1(PIQUE, "As");
    c1.afficher();
    Carte c2 (c1);■
    c2.afficher();
    c2.setType(TREFLE);
    c2.setValeur("Queen");
    c2.afficher();
    Carte c3(PIQUE, "2");
    c2.affecter(c3);
    c2.afficher();
    c3.afficher();

    if ( c1.equal(c2) ) {
        cout << "is ok :-)" << endl;
    } else {
        cerr << " problem bug" << endl;
        c1.afficher();
        c2.afficher();
    }
    return 0;
}
```





3.2.1 - Définition

3.2.2 - Surcharge des opérateurs internes

3.2.3 - Surcharge des opérateurs externes

3.2.4 - Pointeur this

3.2.5 - explicite (C++11)

<http://casteyde.christian.free.fr/cpp/cours/online/x3244.html>

<https://docs.microsoft.com/fr-fr/cpp/cpp/operator-overloading?view=vs-2019>



3.2.1 - Les Classes et les Objets en C++ -> Surcharge des opérateurs : Définition



type operatorOp(paramètres)

Opérateur surchargeable	Opérateur non surchargeable
+ - * / % ^ & ~ ! = < > += -= *= /= %= ^= &= = << >> >>= <<= == != <= >= && ++ -- -<* , -> [] () new delete new[] delete[]	. . * :: ?: sizeof typeid static_cast dynamic_cast const_cast reinterpret_cast





type operatorOp(paramètres)

l'écriture => A Op B

se traduisant par => A.operatorOp(B)

```
class complexe {  
private:  
    ...  
public:  
    ...  
    complexe& operator+=(const complexe&);  
};  
  
complexe& complexe::operator+=(const complexe& c) {  
    r += c.r;  
    i += c.i;  
    return *this;  
};
```

<https://www.calmip.univ-toulouse.fr/c++/surch.html>



3.2.2 - Les Classes et les Objet en C++ -> Surcharge des opérateurs : Surcharge des opérateurs internes Exemple



```
class ChaineCaractere
{
public :
...
ChaineCaractere& operator=(const char * chaine) ;      // opérateur d'affectation
bool operator==(const ChaineCaractere& chaine) ;      // opérateur d'égalité
...
};

int main(int argc, char *argv[])
{
    ChaineCaractere chaine1("Bonjour");
    ChaineCaractere chaine2("Hello");
    if ( chaine1 == chaine2 ) // chaine1.operator==(chaine2)
        cout << "Chaine egale "<< endl;
    else
        cout << "Chaine differente"<<endl;
    chaine2 = chaine1;           // chaine1.operator=(chaine2)
    if ( chaine1 == chaine2 ) // chaine1.operator==(chaine2)
        cout << "Chaine egale"<<endl;
    else
        cout << "Chaine differente"<<endl;
}
```



3.2.3 - Les Classes et les Objets en C++ -> Surcharge des opérateurs : Surcharge des opérateurs externes



friend type operatorOp(paramètres)

l'écriture => A Op B

se traduisant par => operatorOp(A,B)

```
complexe operator+(const complexe& a, const complexe& b) {  
    complexe r=a;  
    r += b;  
    return r;  
};
```

<https://www.calmip.univ-toulouse.fr/c++/surch.html>





3.2.3 - Les Classes et les Objet en C++

-> Surcharge des opérateurs :

Surcharge des opérateurs externes : Exemple



```
class ChaineCaractere
{
public :

    friend ChaineCaractere operator+(ChaineCaractere& chaine1,ChaineCaractere& chaine2);
    ...
};

ChaineCaractere operator+(ChaineCaractere& chaine1,ChaineCaractere& chaine2) {
    ChaineCaractere chaine;
    chaine._impChaine = new char[ strlen(chaine1._impChaine) + strlen(chaine2._impChaine) + 1 ];
    strcpy(chaine._impChaine,chaine1._impChaine);
    strcat(chaine._impChaine,chaine2._impChaine);
    return chaine;
}

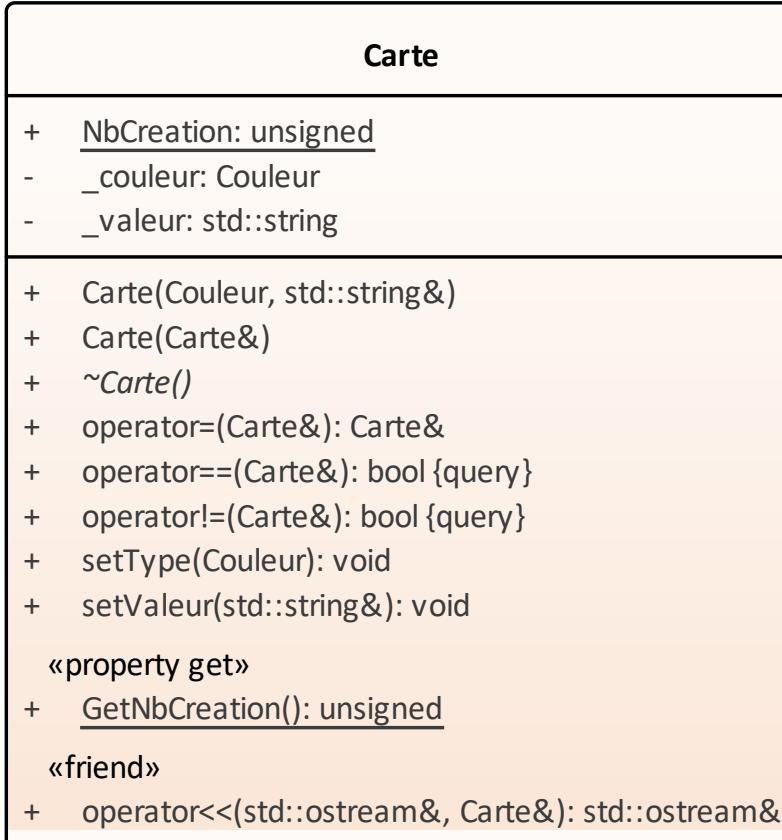
int main(int argc, char *argv[]) {
    ChaineCaractere chaine1("Bonjour");
    ChaineCaractere chaine2("Hello");
    ChaineCaractere chaine3 ;
    chaine3 = chaine1 + chaine2; // operator+(chaine1,chaine2);
    chaine3.afficher();
    return 0;
}
```



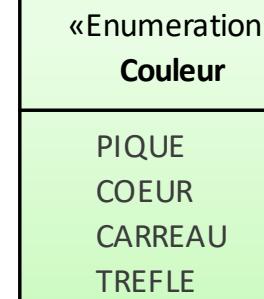
3.2.3 - Les Classes et les Objet en C++ -> Surcharge des opérateurs : Exercices : Jeu de carte (2)



class Domain Model



class Domain Model





3.2.3 - Les Classes et les Objet en C++ -> Surcharge des opérateurs : Exercices : Jeu de carte (2)



```
#include <iostream>
#include "Carte.h"

using namespace std;

int main()
{
    cout << "Jeu de carte" << endl;
    Carte c1(PIQUE, "As");
    cout << c1 << endl;

    Carte c2 (c1);
    cout << c2 << endl;
    c2.setType(TREFLE);
    c2.setValeur("Queen");
    cout << c2 << endl;

    if ( c1 != c2 ) {
        cout << "is ok :-)" << endl;
    } else {
        cout << " problem bug" << endl;
    }
    return 0;
}
```





3.2.4 - Les Classes et les Objets en C++ -> Pointeur this



Un objet peut accéder à son adresse mémoire par un pointeur appelé **this**.

Exemple :

```
ChaineCaractere& ChaineCaractere::operator=(const ChaineCaractere& chaine )
{
    if (this == &chaine) return *this;
    delete [] this->_impChaine;
    this -> _impChaine = new char[ strlen(chaine._impChaine) +1];
    strcpy( this-> _impChaine,chaine._impChaine);
    return *this;
}

bool ChaineCaractere::operator==(const ChaineCaractere& chaine )
{
    if (this == &chaine) return true;
    return this->isEqual(chaine.getChaine());
}
```





explicite empêche le compilateur de convertir une donnée en une autre (constructeur ou opérateur de conversion)

```
class Age
{
    private:
        int theage;
    public :
        Age(int): theage(a){}
        int& operator int() { return this->theage; }
};

void fonction(Age b);

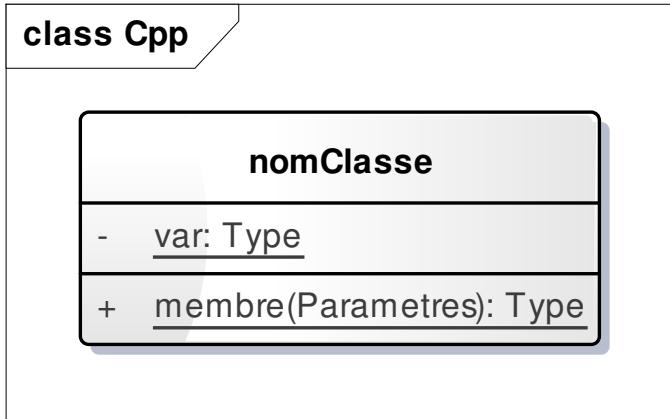
fonction(67) //OK implicite
Age a(32) ;
Int i=a; // OK
```

```
class Age
{
    private:
        int theage;
    public :
        explicite Age(int): theage(a){}
        explicite int& operator int() {
            return this->theage; }

};

void fonction(Age b);

fonction(67) //KO
fonction(Age(67));
Age a(32) ;
Int i=a; // KO
Int i = (int)a;
```



Déclaration :

```
class nomClass
{
    static Type membre(paramètre);
    static Type var;
};
```

Définition :

```
Type nomClass::var = « valeur par défaut »;  
Type nomClass:: membre(paramètre) { ... }
```

Utilisation :

```
nomClass::membre(paramètre);
```

3.3 - Les Classes et les Objet en C++ -> Membres static : Exemple



```
class ChaineCaractere
{
public :
    ...
    static ChaineCaractere Copy(const ChaineCaractere& chaine) ;
    ...
};

ChaineCaractere ChaineCaractere::Copy(const ChaineCaractere& chaine)
{   return chaine; }

int main(int argc, char *argv[])
{
    ChaineCaractere chaine1("Bonjour");
    ChaineCaractere chaine2 = ChaineCaractere::Copy (chaine1);
    if ( chaine1 == chaine2 )
        cout << "Chaine égale "<< endl;
    else
        cout << "Chaine différente"<<endl;
return 0;
}
```



3.3 - Les Classes et les Objets en C++ -> Membres static : Exemple



```
class Example {
    static int Num_instances;
    int i;
    std::string a_str;
public:
    static std::string Static_str;
    static int Static_func();

    Example():i(0),a_str("") { ++Num_instances; }
    void set_str(const std::string& str) { this->a_str=str; }
};
```

Example.hh

```
int     Example::Num_instances = 0;
std::string Example::Static_str = "Hello.";
int     Example::Static_func() { return Example::Num_instances; }
```

Example.cpp





- Fonction amies:

```
Class nom class
{
    friend Type méthode (paramètre);
}
```

- Class amie :

```
Class nom class
{
    friend class nomClassAmie;
}
```



3.4 - Les Classes et les Objet en C++ -> Fonction et classes amies: Exemple



```
class ChaineCaractere
{
public :
    ...
    friend ChaineCaractere operator+(ChaineCaractere& chaine1,ChaineCaractere& chaine2);
    friend class Conversion;
    ...
};

class Conversion
{
public :
    static int getInt(const ChaineCaractere& chaine);
private :
    Conversion() {};
};

int Conversion::getInt(const ChaineCaractere& chaine)
{ return atoi(chaine._impChaine); }

int main(int argc, char *argv[])
{
    ChaineCaractere chaine("3333");
    cout << "conversion : " << Conversion::getInt(chaine) << endl;
    return 0;
}
```



3.5 - Les Classes et les Objet en C++ -> Méthodes constantes



Déclaration :

```
class nomClass
{
public :
    nomClass();
    const Type membreConst(paramètre) const;
    Type membre(paramètre);

private :
    Type var1;
    mutable Type var2;
};
```

Utilisation :

```
const nomClass instance ;
instance.membreConst(parametre);
```



3.5 - Les Classes et les Objet en C++ -> Méthodes constantes : Exemple



```
class ChaineCaractere
{
public :
    ...
    void afficher() const;
    const char* getChaine() const const { _compteur++; return _impChaine; }
    ...
private :
    char * _impChaine;
    mutable int _compteur;
};

void ChaineCaractere::afficher() const
{
    cout << "ChaineCaractere => " << _impChaine << endl;
    cout << "la méthode getChaine a été appelée " << _compteur << " fois" << endl;
}

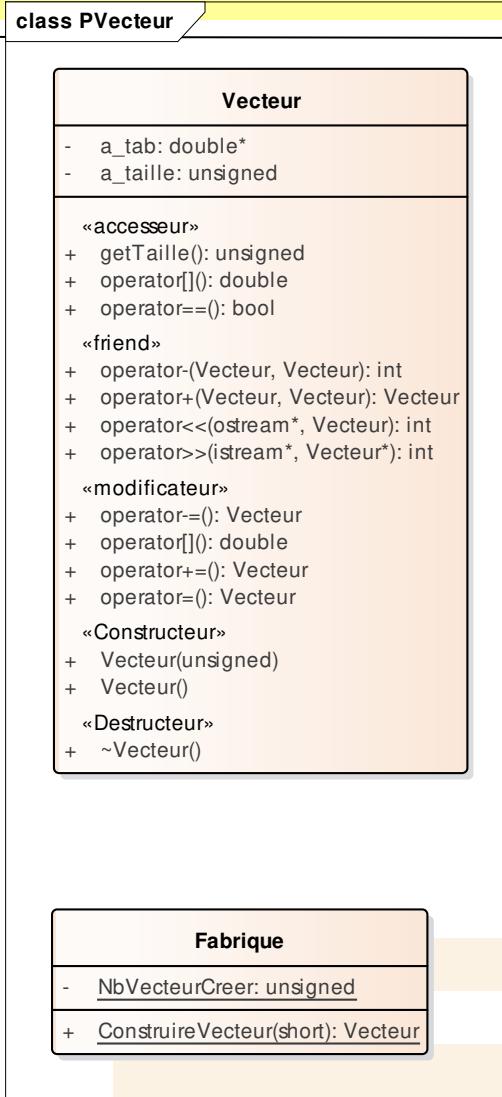
int main(int argc, char *argv[])
{
    const ChaineCaractere chaine1("Bonjour");
    cout << chaine1.getChaine() << endl;
    chaine1.afficher();
    return 0;
};
```



3.5 - Les Classes et les Objets en C++ -> Exercice : Vecteur



- Créer une classe Vecteur
 - Constructeur avec la dimension du vecteur
 - Destructeur
 - operator = (Vecteur)
 - operator [] const
 - operator []
 - getTaille() const
 - operator+=(Vecteur) : Vecteur
 - operator-=(Vecteur) : Vecteur
- Créer des opérateurs amies de Vecteur :
 - operator+(Vecteur, Vecteur) : Vecteur
 - operator-(Vecteur, Vecteur) : double
 - operator << (ostream, Vecteur): ostream
 - operator >> (istream, Vecteur): istream
- Créer une classe Fabrique contenant :
 - Une méthode static ConstruireVecteur
 - Un compteur de construction de vecteur





Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



- 3.6.1 - Déclaration
- 3.6.2 - Droits d'accès sur les membres hérités
- 3.6.3 - Constructeur / Destructeur dans les classes dérivées
- 3.6.4 - Constructeur et constexpr (C++11)
- 3.6.5 - Héritage des constructeurs (C++11)
- 3.6.6 - Contrôle des méthodes vitales (C++11)
- 3.6.7 - Classe virtuelle
- 3.6.8 - Fonction virtuelle

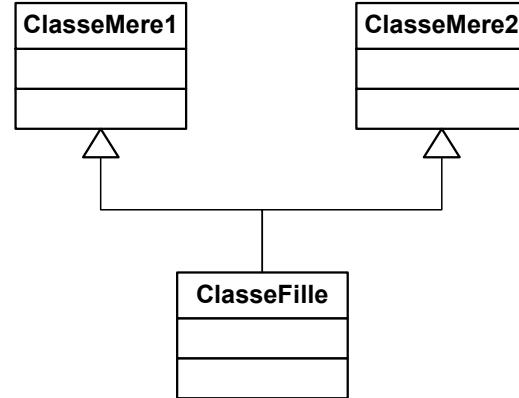


```
class ClasseMere1
{
/* Contenu de la classe mère 1. */
};
```

```
[class ClasseMere2
{
/* Contenu de la classe mère 2. */
};]
```

[...]

```
class ClasseFille : public|protected|private ClasseMere1
[, public|protected|private ClasseMere2 [...]]
{
/* Définition de la classe fille. */
};
```





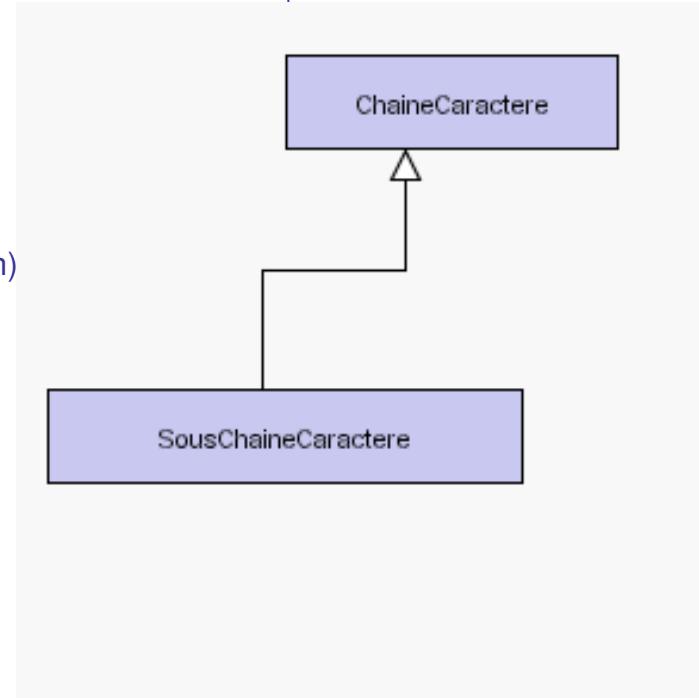
3.6.1 - Les Classes et les Objets en C++ -> Héritage : Exemple



```
#ifndef SousChaineCaractere_hpp
#define SousChaineCaractere_hpp
#include "ChaineCaractere.hpp"
class SousChaineCaractere : public ChaineCaractere
{
public :
    SousChaineCaractere(const SousChaineCaractere& chaine);
    SousChaineCaractere(const ChaineCaractere& chaine, int debut, int fin)
    ~SousChaineCaractere();
    ChaineCaractere copy() const;
    void setDebut(int debut);
    void setFin (int debut);

    void afficher() const;
    const char* getChaine() const;
    void setChaine(const char* chaine );
    ChaineCaractere& operator=(const SousChaineCaractere& chaine) ;
    bool operator==(const SousChaineCaractere& chaine );

protected :
    bool isEqual(const char * chaine );
private :
    int _debut;
    int _fin;
}
#endif
```





3.6.2 - Les Classes et les Objets en C++

-> Héritage :

Droits d'accès sur les membres hérités



Type d'accès aux membres de la classe de base	Type d'héritage		
	Héritage public	Héritage protected	Héritage private
public	public	protected	private
protected	protected	protected	private
private	Masqué pour la classe dérivée	Masqué pour la classe dérivée	Masqué pour la classe dérivée

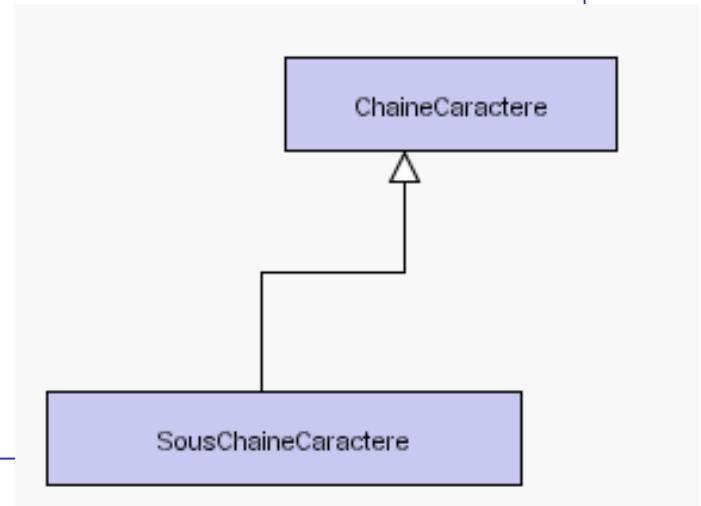




```
SousChaineCaractere::SousChaineCaractere(const SousChaineCaractere& chaine)
:ChaineCaractere(chaine)
{}
```

```
SousChaineCaractere::SousChaineCaractere(const ChaineCaractere& chaine, int debut,
int fin)
:ChaineCaractere(chaine),_debut(debut),_fin(fin)
{}
```

```
SousChaineCaractere::~SousChaineCaractere()
{}
```





Un constructeur est un constructeur d'expression constante si :

- Il est déclaré avec le mot clé `constexpr`
- La partie initialiseur-membre n'implique que des expressions constantes potentielles
- Son corps est vide.

```
class A {  
    constexpr A(int newA, int newB) : a(newA), b(newB) {}  
private:  
    int a;  
    int b;  
};
```



3.6.4 - Les Classes et les Objets en C++

-> Héritages :

Constructeur et constexpr (C++11)



```
constexpr int fac(int n)
{
    return n == 1 ? 1 : n*fac(n - 1);
}
```

```
class Carre {
public:

    constexpr Carre(double d):a_cote(d) {}

    void afficher() const {
        std::cout << "Carre de cote : " << this->a_cote << std::endl;
        std::cout << "perimetre      : " << this->perimetres() << std::endl;
    }

    constexpr double perimetres() const { return this->a_cote * 2; }

private:
    double a_cote = 0.0;
};
```

```
constexpr Carre a2(5.5);
a2.afficher();

auto start = std::chrono::steady_clock::now();
constexpr auto x = fac(12);
auto stop = std::chrono::steady_clock::now();
auto dur = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
std::cout << "fac(12)=" << x << " a durer " << static_cast<double>(dur.count()) << " milli seconds!" << std::endl;
```

```
Carre de cote : 5.5
perimetre      : 11
fac(12)=479001600 a durer 0 milli seconds!
```





Permet d'initialiser une classe dérivée avec le même ensemble de constructeur que la classe de base

```
struct A {  
    A(int );  
};  
struct B: A {  
    B(int,int );  
};
```

```
struct C: A {  
    using A::A;  
};  
  
struct D: B {  
    using B::B;  
};
```

```
struct C: A {  
    C();  
    C(int);  
    C(const C&);  
};  
  
struct D: B {  
    D();  
    D(int);  
    D(int,int)  
    D(const B&);  
};
```



ClassName() = default;

l'utilisateur peut toujours forcer la génération du constructeur implicitement déclarée avec le mot-clé default

ClassName() = delete;

l'utilisateur peut empêcher la génération du constructeur déclarée avec le mot-clé delete



```
struct A {
    int x;
    A(int x = 1) : x(x) {};
}
struct B : A {
    // B::B() implicite
};

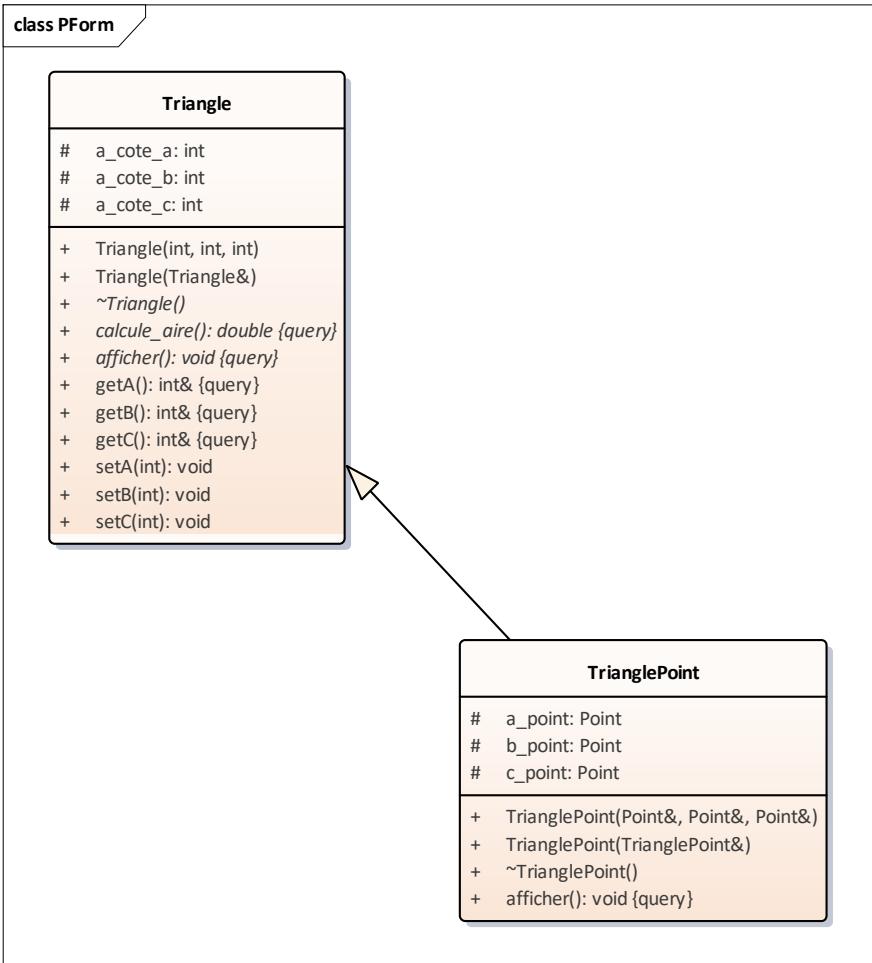
struct D : A {
    D(int y) : A(y) {}
    D(const D&)=delete; // désactive la génération
    // automatique
    // D::D() n'existe pas
};

struct E : A {
    E(int y) : A(y) {}
    E() = default; // explicite et appelle A::A()
};
```

```
int main()
{
    A a;
    B b;
    // D d; compile error
    D d1(1);
    // D d2( d1 ); compile error
    E e;
}
```



3.6.6 - Les Classes et les Objets en C++ -> Héritages : Exemple & Exercices



```
namespace PForme
```

```
class Triangle
```

```
public:
```

```
Triangle(int a = -1, int b = -1, int c = -1);  
Triangle(const Triangle& tr);
```

```
virtual ~Triangle();
```

```
double calcule_aire() const;  
virtual void afficher() const;
```

```
const int& getA() const;  
const int& getB() const;  
const int& getC() const;
```

```
void setA(int a);  
void setB(int b);  
void setC(int c);
```

```
protected:
```

```
int a_cote_a;  
int a_cote_b;  
int a_cote_c;
```

```
;
```

```
Triangle.hh
```





3.6.6 - Les Classes et les Objet en C++ -> Héritages : Exemple & Exercices



```
namespace PForme {
    class Point {
        public :
            enum TypeCoordonne2D { X = 0, Y = 1 };

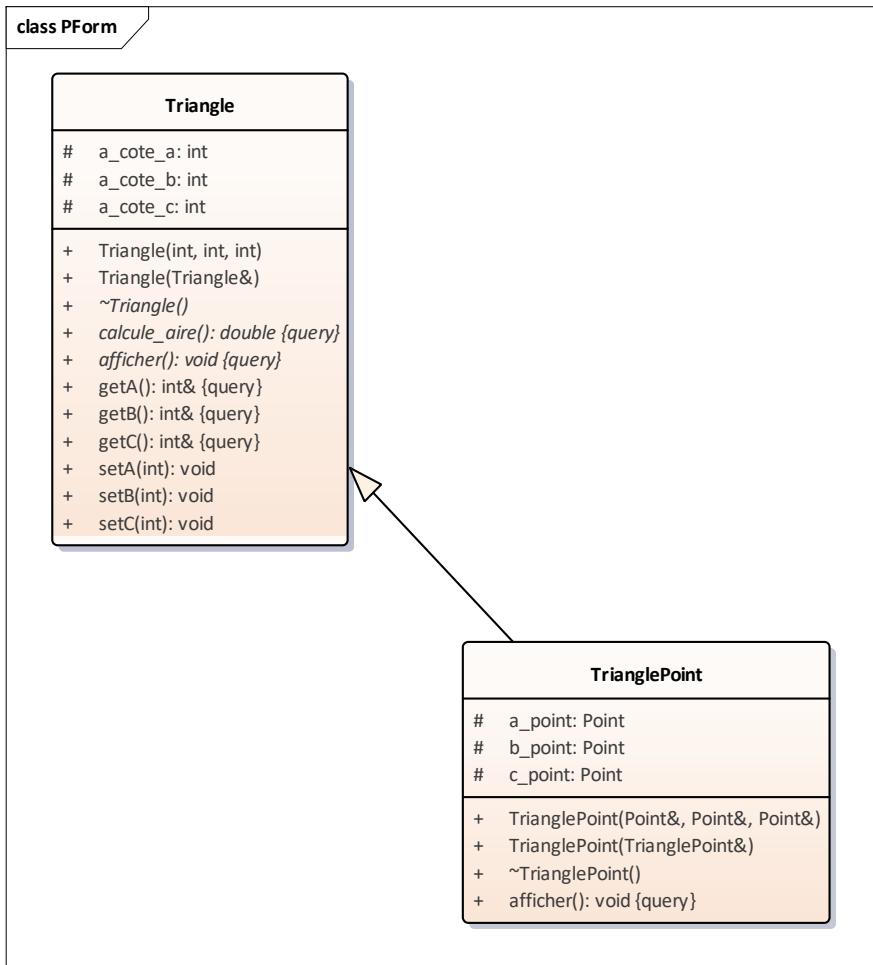
            Point( int x,int y):a_point()
            {
                a_point[X] = x;
                a_point[Y] = y;
            }
            Point( const Point& origin):a_point()
            {
                a_point[X] = origin.a_point[X];
                a_point[Y] = origin.a_point[Y];
            }

            void afficher()const { std::cout << "[" << a_point[X] << "," << a_point[Y] << "]" << std::endl; }
            int distance(const Point& b) const
            {
                return sqrt( (a_point[X] - b.a_point[X]) * (a_point[X] - b.a_point[X]) +
                            (a_point[Y] - b.a_point[Y]) * (a_point[Y] - b.a_point[Y]));
            }
        protected:
            int a_point[2];
    };
}
```

Point.hh



3.6.6 - Les Classes et les Objets en C++ -> Héritage : Exemple & Exercices



```
namespace PForme
{
    class Point;
    class TrianglePoint : public Triangle
    public:
        TrianglePoint(const Point&, const Point&, const Point&);
        TrianglePoint(const TrianglePoint& tr);

        ~TrianglePoint();

        void afficher() const override;

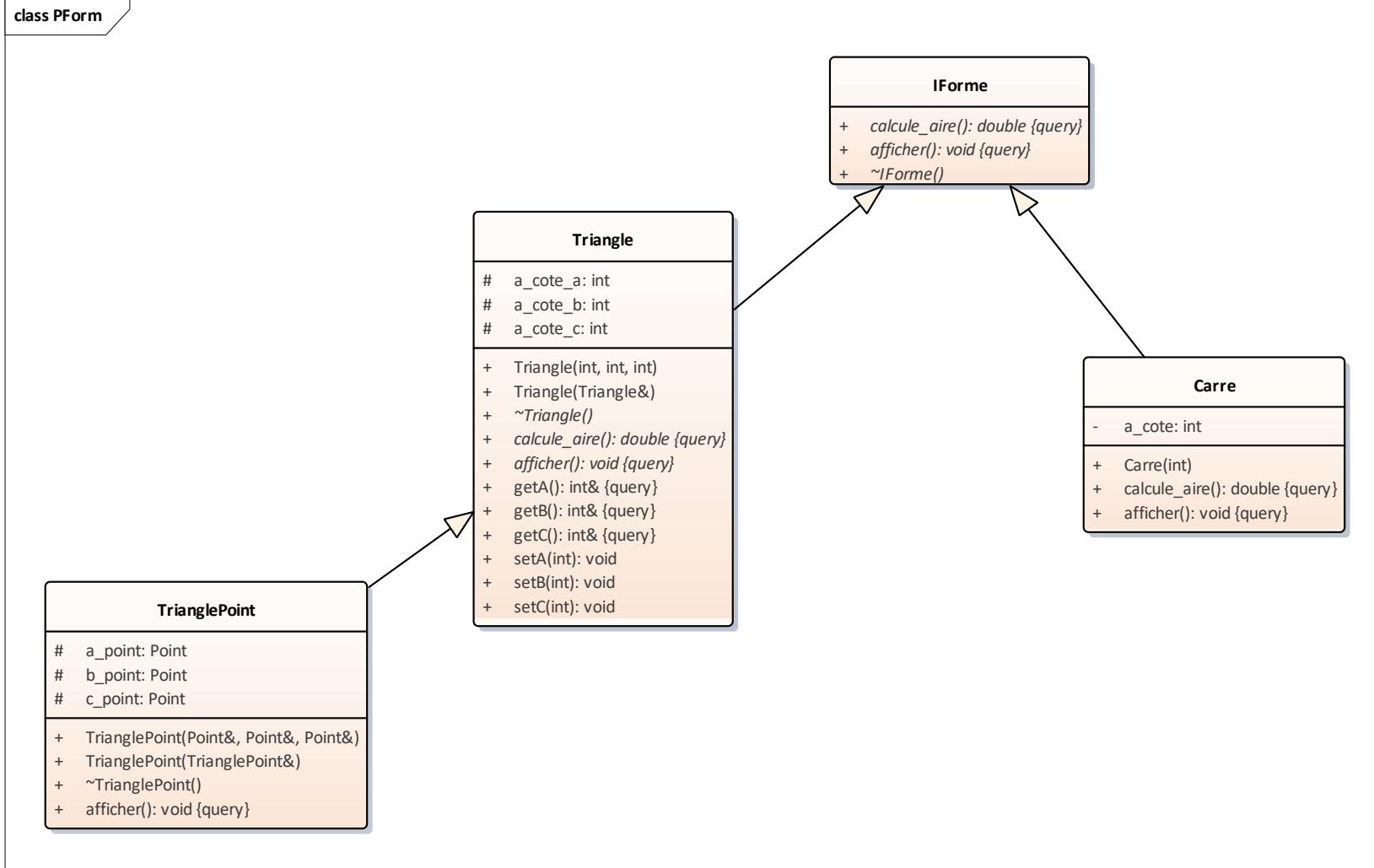
    protected:
        int a_point;
        int b_point;
        int c_point;
}
```

TrianglePoint.hh





3.6.6 - Les Classes et les Objet en C++ -> Héritages : Exemple & Exercices





3.6.6 - Les Classes et les Objets en C++ -> Héritages : Exemple & Exercices



```
namespace PForme {
```

```
class iForme {
public:
    virtual double calcule_aire() const = 0;
    virtual void afficher() const = 0;
    virtual ~iForme() {}
```

IForme.hh

```
namespace PForme {
```

```
class Carre : public IForme {
public:
    Carre(int cote = 2);
    double calcule_aire() const override;
    void afficher() const override;
protected:
    int a_cote;
```

Carree.hh

```
namespace PForme {
```

```
class Triangle : public IForme {
public:
    Triangle(int a = -1, int b = -1, int c = -1);
    Triangle(const Triangle& tr);
    ~Triangle();
    double calcule_aire() const override;
    void afficher() const override;
    const int& getA() const;
    const int& getB() const;
    const int& getC() const;
```

```
void setA(int a);
void setB(int b);
void setC(int c);
```

```
protected:
```

```
int a_cote_a;
int a_cote_b;
int a_cote_c;
```

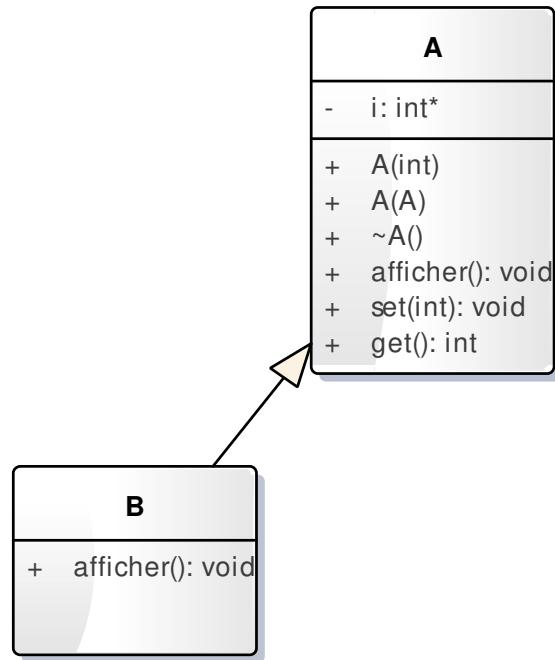
Triangle.hh



3.6.6 - Les Classes et les Objet en C++ -> Héritages : Exemple & Exercices



```
class Cpp
```



```
class B : public A {
public:
    void afficher() ;
};
```

```
int main() {
B a = 2;
const B b = a;
```

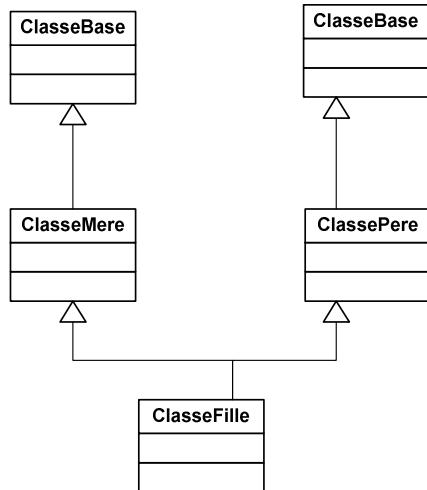
```
a.afficher();
b.afficher()
a.set(4);
b.set(3);
```

```
a.afficher();
b.afficher();
```

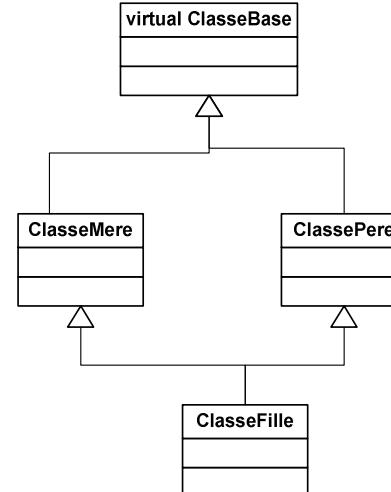
```
return 0;
}
```



3.6.7 - Les Classes et les Objet en C++ -> Héritages : Classe virtuelle



```
ClasseMere : public ClasseBase {}  
ClassePere : public ClasseBase {}  
ClasseFille : public ClasseMere, ClassePere {}
```



```
ClasseMere : virtual public ClasseBase {}  
ClassePere : virtual public ClasseBase {}  
ClasseFille : public ClasseMere, ClassePere {}
```



Déclaration :

```
virtual méthode(paramètre);
```

Fonction virtuelle pure :

```
virtual méthode(paramètre) = 0 ;
```



Le **polymorphisme** est la capacité, pour une fonction, d'être interprétée différemment en fonction de l'objet auquel elle s'applique.

- Le **polymorphisme statique** (la surcharge) . Le sens de la fonction est déterminé statiquement par la signature des arguments à l'appel.
- Le **polymorphisme dynamique** : Le sens de la fonction est déterminé par le type effectif de l'objet à l'appel.

Le polymorphisme ne s'applique qu'aux méthodes d'instances.



3.7.2 - Les Classes et les Objet en C++ -> Polymorphisme : Exemple



```
class ChaineCaractere
{
public :
...
virtual void afficher() const;
};

class SousChaineCaractere : public ChaineCaractere
{
public :
...
virtual void afficher() const;
};

int main(int argc, char *argv[])
{
    ChaineCaractere chaine1("Bonjour");
    SousChaineCaractere souschaine1(chaine1,1,2);
    souschaine1.afficher();

    ChaineCaractere *souschaine2 = new SousChaineCaractere(chaine1,1,2);
    souschaine2->afficher();

    delete souschaine2;
    return 0;
}
```





C + 11 ajoute également la capacité d'empêcher d'hériter de classes ou simplement prévenir méthodes impérieuses classes dérivées. Cela se fait avec l'identificateur spécial `final`.

```
struct Base1 final { };
```

```
struct Derived1 : Base1 { }; // ill-formed because the class Base1 has been marked final
```

```
struct Base2 {  
    virtual void f() final;  
};
```

```
struct Derived2 : Base2 {  
    void f(); // ill-formed because the virtual function Base2::f has been marked final  
};
```



La override identificateur spécial signifie que le compilateur vérifie la classe de base (es) pour voir si il ya une fonction virtuelle avec cette signature exacte. Et s'il n'y a pas, le compilateur indiquera une erreur

```
struct Base {  
    virtual void some_func(float);  
};
```

```
struct Derived : Base {  
    virtual void some_func(int) override;  
        // ill-formed - doesn't override a base class method  
};
```



Une *exception* est l'interruption de l'exécution du programme à la suite d'un événement particulier.

Les exceptions sont un mécanisme de déroutement conditionnel.

- **throw** lève une exception;
- **catch** capte l'exception.

throw objet;

```
try
{
// Code susceptible de générer des exceptions...
}
catch (classe [&][temp])
{
// Traitement de l'exception associée à la classe
}
```

3.8.1 - Les Classes et les Objets en C++ -> Le Traitement des Exceptions : Exemple



```
void SousChaineCaractere::setDebut(int debut)
{
    if ( ( debut < 0 ) || ( debut >= strlen( this->getChaine() ) ) ) throw runtime_error("setDebut");
    _debut = debut;
}

void SousChaineCaractere::setFin (int fin)
{
    if ( ( fin < 0 ) || ( fin >= strlen( this->getChaine() ) ) ) throw runtime_error("setFin");
    _fin = fin;
}

int main(int argc, char *argv[])
{
    ChaineCaractere chaine1("Bonjour");
    try
    {
        souschaine1.setDebut(-6);
    }
    catch ( runtime_error e )
    {
        cout << "Exception : " << e.what() << endl;
    }
    return 0;
}
```



noexcept(expression) noexcept

noexcept et son synonyme noexcept(true) spécifient que la fonction ne lèvera jamais d'exception

noexcept est une version améliorée de throw () , qui est « deprecated »en C++11

```
template < class T > void foo ( ) noexcept ( noexcept ( T ( ) ) ) { }
void bar ( ) noexcept ( true ) { }
void baz ( ) noexcept { throw 42 ; } // noexcept equivalent à noexcept(true)
```

```
int main ( )
{
    foo < int > ( ) ; // noexcept(noexcept(int())) => noexcept(true), so this is fine

    bar ( ) ; // fine
    baz ( ) ; // compiles, but at runtime this calls std::terminate
}
```



3.8.2 - Les Classes et les Objets en C++ -> Le Traitement des Exceptions : Le mot clé : noexcept



noexcept : Spécifie si une fonction peut lever des exceptions.

```
int division ( int i, int j) {  
    if ( j == 0) throw std::runtime_error("division par 0 : division");  
    return i/j;  
}  
  
int division_noexcept_false ( int i, int j) noexcept(false) {  
    if ( j == 0) throw std::runtime_error("division par 0 : division_noexcept_false");  
    return i/j;  
}  
  
int division_noexcept_test_false ( int i, int j) noexcept(noexcept(division(i,j))) {  
    return division(i,j);  
}
```

```
try {  
    division (0,0);  
} catch ( std::exception &e ){  
    std::cerr << e.what() << std::endl;  
};  
try {  
    division_noexcept_false (0,0);  
} catch ( std::exception &e ){  
    std::cerr << e.what() << std::endl;  
};  
try {  
    division_noexcept_test_false (0,0);  
} catch ( std::exception &e ){  
    std::cerr << e.what() << std::endl;  
};
```

```
division par 0 : division  
division par 0 : division_noexcept_false  
division par 0 : division
```





3.8.2 - Les Classes et les Objets en C++

-> Le Traitement des Exceptions :

Le mot clé : noexcept



```
int division_noexcept ( int i, int j) noexcept {
    if ( j == 0) throw std::runtime_error("division par 0 : division_noexcept");
    return i/j;
}

int division_noexcept_true ( int i, int j) noexcept(true) {
    if ( j == 0) throw std::runtime_error("division par 0 : division_noexcept_true");
    return i/j;
}

int division_noexcept_test_true ( int i, int j) noexcept( ! noexcept(division(i,j))) {
    return division(i,j);
}
```

```
try {
    division_noexcept(0,0);
} catch ( std::exception &e){
    std::cerr << e.what() << std::endl;
};
```

```
terminate called after throwing an instance of 'std::runtime_error'
what(): division par 0 : division_noexcept
```

```
try {
    division_noexcept_true(0,0);
} catch ( std::exception &e){
    std::cerr << e.what() << std::endl;
};
```

```
terminate called after throwing an instance of 'std::runtime_error'
what(): division par 0 : division_noexcept_true
```

```
try {
    division_noexcept_test_true (0,0);
} catch ( std::exception &e ){
    std::cerr << e.what() << std::endl;
};
```

```
terminate called after throwing an instance of 'std::runtime_error'
what(): division par 0 : division
```



Lorsque le programmeur donne un nom à un espace de nommage, celui-ci est appelé un *espace de nommage nommé*.

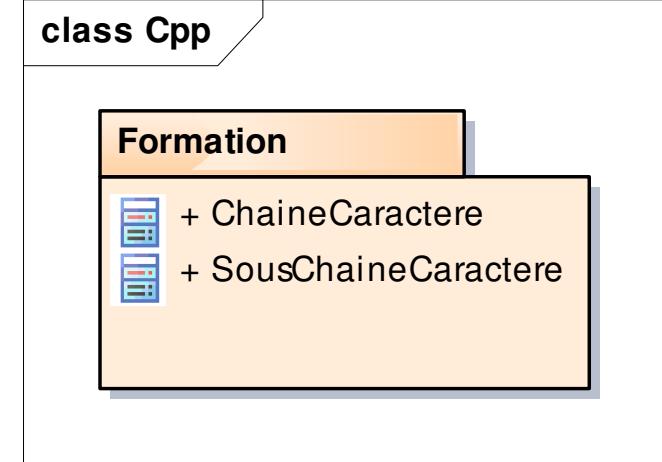
Définition:

```
namespace nom  
{  
déclarations | définitions  
}
```



namespace Formation

```
{  
    class ChaineCaractere {...}  
    class SousChaineCaractere {...}  
}
```



using namespace Formation;

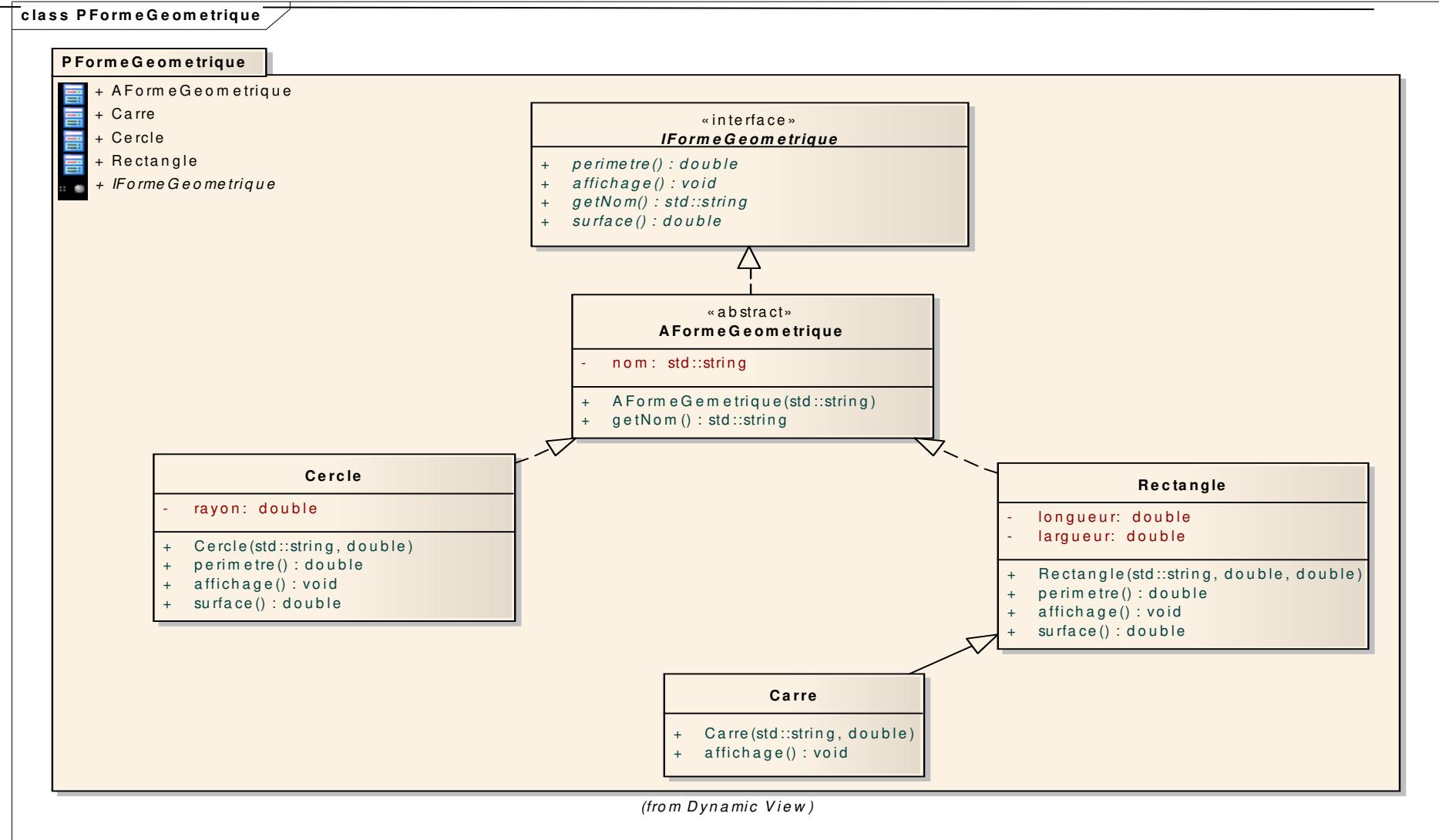
```
#include <time.h>  
int main(int argc, char *argv[])  
{ ... }
```



3.9 - Les Classes et les Objets en C++

-> Les espaces de nom :

Exercice - Package PFormeGeometrique





Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



3.10.1 - Déclaration

3.10.2 - Définition

3.10.3 - Instanciation des Patrons

3.10.4 - extern template class (C++11)

3.10.5 - Liste d'initialisation (C++11)



3.10.1 - Les Classes et les Objets en C++ -> Les Patron: Déclaration (1)



```
#ifndef VECTEUR_HH
#define VECTEUR_HH

namespace Formation
{
    template < typename T > class Vecteur {

        public:
            Vecteur(const Vecteur<T>& vec) : taille(vec.taille), valeur(new T[vec.taille])
            { for (int i=0;i < this->taille;i++) valeur[i] = vec.valeur[i]; }

            Vecteur(int ataille=1) : taille(ataille), valeur(new T[taille])
            { for (int i=0;i < this->taille;i++) this->valeur[i] = T(0); }

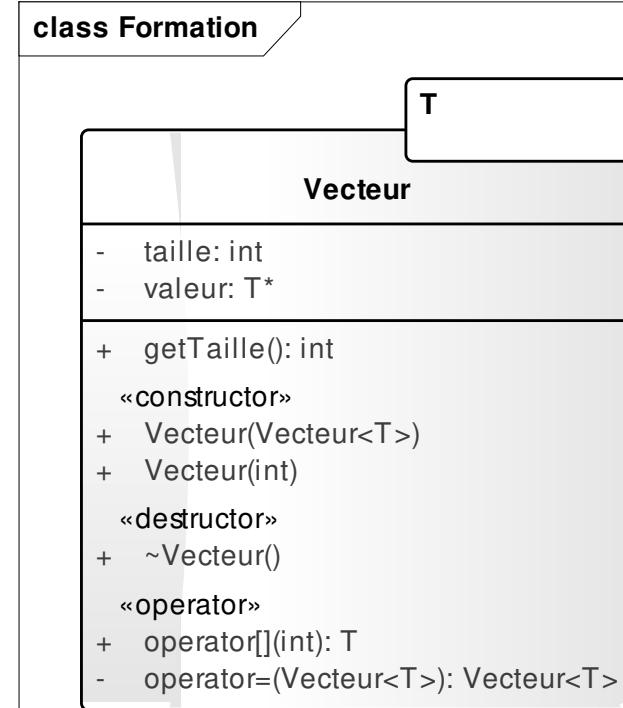
            ~Vecteur() { delete this->valeur; }

            const T& operator[](const int& i) const { return valeur[i]; }
            T& operator[](const int& i) ; { return valeur[i]; }

            const int& gettaille() const { return taille; }

        private:
            Vecteur<T>& operator=(const Vecteur<T>& vec);
            const int taille;
            T* valeur;
    };
}

#endif
```





Déclaration des types template

```
template <class|typename nom[=type] [, class|typename  
nom[=type] [...]>
```

Déclaration des constantes template

```
template <type paramètre[=valeur][, ...]>
```

Fonctions template

```
template <paramètres_template> type  
fonction(paramètres_fonction);
```



3.10.2 - Les Classes et les Objets en C++

-> Les Patrons : Définition



Les classes template

Déclaration

```
template <paramètres_template> class|struct|union nom;
```

Définition

```
template <paramètres_template>
type classe<paramètres>::nom(paramètres_méthode)
{ ... }
```

Fonctions membres template

```
class A
{
public:
    template <class T> void add(T valeur);
};

template <class T> void A::add(T valeur)
{ ... }
```





- **Declaration**

```
template <class T> T Min(T a, T b) { if (a > b) return b; return a; }
```

- **Instanciation implicite**

```
int i=Min(2,3);
```

- **Instanciation explicite**

```
int i=Min<int>(2,3.0);
double i=Min<double>(2,3.0);
```



- **Declaration**

```
template <class T> class Type
{
public:
    Type( const T& val): _imp(val) {}
    T& get() { return this->_imp;}
    const T& get() const { return this->_imp;}
private:
    T _imp;
};
```

- **Instanciation explicite**

```
Type<int> val(5);
```

```
using TypeDouble Type<double>;
TypeDouble vald(2.3);
```



3.10.3 - Les Classes et les Objets en C++

-> Les Patrons : Instanciation des Patrons (2)



```
#include <iostream>
#include "Vecteur.hh"

using Formation::Vecteur;

int main(int argc, char *argv[])
{
    Vecteur<int> vint(5);
    Vecteur<double> vdbl(2);

    std::cout << vint << std::endl;
    std::cout << vdbl << std::endl;

    for (int i=0;i < vint.gettaille();i++) vint[i] = i+1;
    std::cout << vint << std::endl;

    return 0;
}
```





Les templates ne sont actuellement pas pris en compte par l'éditeur de liens : il est nécessaire d'incorporer leur définition dans tous les fichiers sources les utilisant en programmation modulaire. => compilation longue et gourmande puisque la classe était recompilée dans chaque fichier source, pour chaque type utilisé.

C++11 permettra l'utilisation du mot-clé `extern` pour rendre les templates globaux. Les fichiers désirant utiliser le template n'ont qu'à le déclarer.



Pour initialiser un conteneur à l'aide de valeurs connues, il fallait le faire élément par élément.

C++11 introduit le patron de classe `std::initializer_list` qui permet d'initialiser les conteneurs avec la même syntaxe que celle permettant en C d'initialiser les tableaux , donc à l'aide d'une suite de valeurs entre accolades.

```
int a[] = { 1, 2, 3 };

template<class T> void maFunction(initializer_list<T> values)
{
    cout << " Nombre élément : " << values.size() << endl;
    for (auto i = begin(values); i < end(values); ++i)
    {
        cout << *i << " ";
    }
    cout << endl;
}

maFunction<int>({ 1, 2, 3 });

vector<string> a = { "Ignoring", "The", "Voices" };
```



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



4 - Compléments



- 4.1 - Les Flots de Données
- 4.2 - La bibliothèque standard (STL)
- 4.3 - Identification dynamique des types
- 4.4 – C librairies (C++11)
- 4.5 – C++ librairies (C++11)



4.1.1 - Les Flots de Données prédéfinis

4.1.2 - Flux de sortie

4.1.3 - Flux d'entrée

4.1.4 - Mode d'ouverture des flux

4.1.5 - État d'erreur des flux : Flag

4.1.6 - État d'erreur des flux : Méthode

4.1.7 - État du format des flux

4.1.1 – Compléments -> Les Flots de Donnés : prédefinis



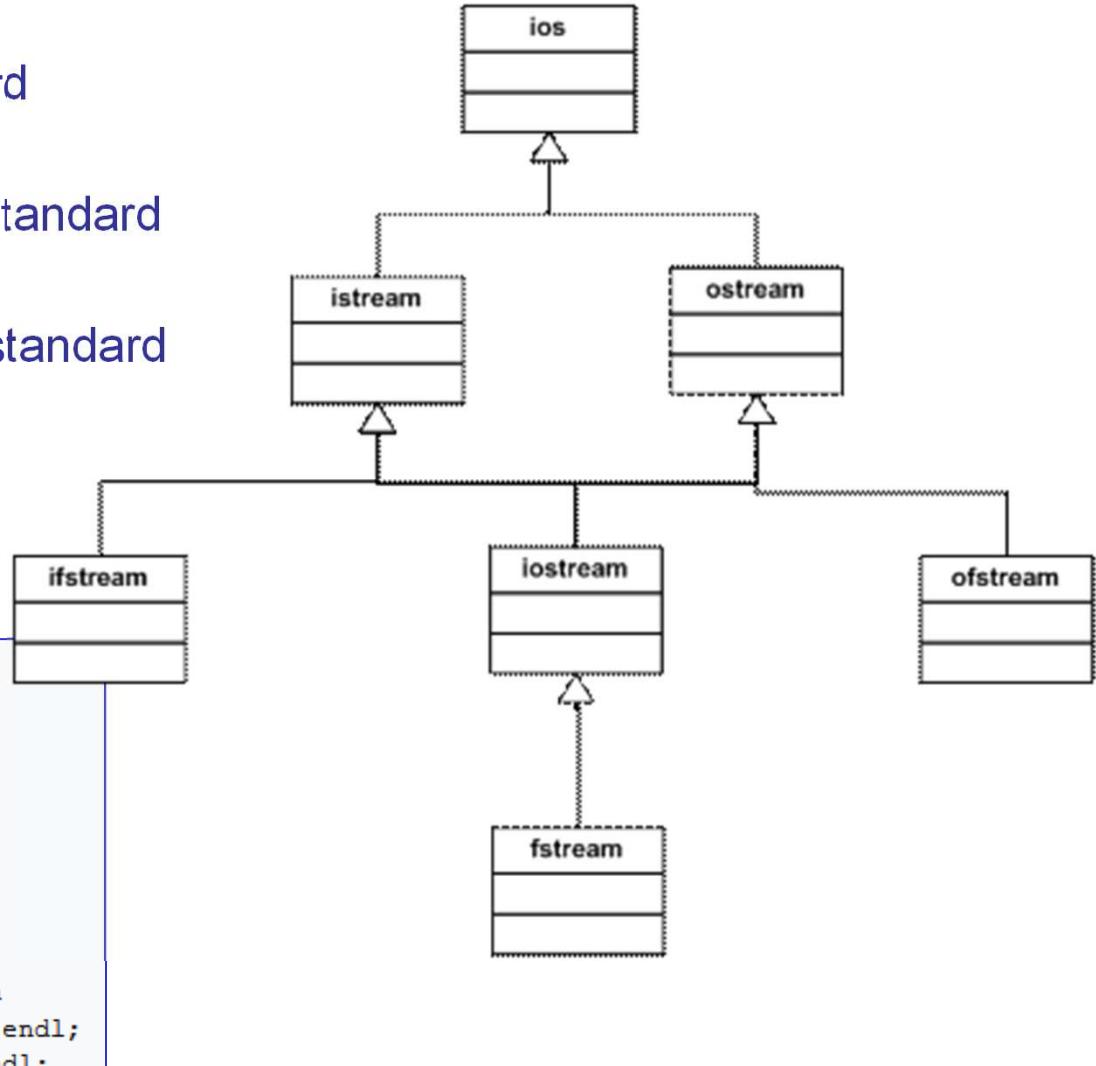
Flots prédefinis sont

- **cout** attaché à la sortie standard
 - instance de la classe ostream
- **cerr** attaché à la sortie erreur standard
 - instance de la classe ostream
- **clog** attaché à la sortie erreur standard mises en tampon
 - instance de la classe ostream
- **cin** attaché à l'entrée standard
 - instance de la classe istream;

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    cout << "Entrez un nombre positif : ";
    cin >> n;
    if (n<0) cerr << "Erreur: Le nombre " << n
        << " n'est pas positif " << endl;
    else cout << "Vous avez entré " << n << endl;
    return 0;
}
```





- Opérateur : <<
- Manipulateur de flux : endl, flush, precision, setprecision
- Fonction membre : put, write

```
ifstream fichier("test.txt");
while (fichier.good())
    cout << (char) fichier.get();
fichier.close();
```

https://fr.wikibooks.org/wiki/Programmation_C%2B%2B/Les_entrées-sorties

4.1.3 – Compléments -> Les Flots de Donnés : Flux d'entrée



- Opérateur : >>
- Fonction membre : get, getline, eof, read, gcount

```
fstream fichier("test.txt");
fichier << setw(10) << a << endl;
fichier.seekg(0, ios_base::beg);
fichier >> b;
fichier.close();
```

https://fr.wikibooks.org/wiki/Programmation_C%2B%2B/Les_entr%C3%A9es-sorties

4.1.4 – Compléments -> Les Flots de Donnés : Mode d'ouverture des flux



Mode	Description
ios::in	ouverture en lecture seule (obligatoire pour la classe ifstream)
ios::out	ouverture en écriture seule (obligatoire pour ofstream)
ios::binary	ouverture en mode binaire
ios::app	ouverture en ajout de données (écriture en fin de flux)
ios::ate	déplacement en fin de flux après ouverture
ios::trunc	si le flux existe, son contenu est effacé (obligatoire si ios::out est activé sans ios::ate ni ios::app)
ios::noreplace	le flux ne doit pas exister préalablement à l'ouverture (sauf si ios::ate ou ios::app est activé)
ios::nocreate	le flux doit exister préalablement à l'ouverture

4.1.5 – Compléments -> Les Flots de Donnés : État d'erreur des flux : Flag



Attribut de ios_base	Description
ios_base::goodbit	Lorsque ce bit vaut 0, ainsi que tous les autres, tout va bien. La fonction membre int good(void) renvoie 1 si tous les bits d'état sont à zéro (tout va bien), 0 sinon.
ios_base::eofbit	Lorsque ce bit vaut 1, la fin du fichier est atteinte. La fonction membre int eof() renvoie 1 dans ce cas, 0 sinon
ios_base::failbit	Ce bit est à 1 lorsqu'une opération a échoué. Le flot peut être réutilisé.
ios::hardfail	Ce bit est à 1 lorsqu'une erreur grave s'est produite ; il ne faut plus utiliser le flot.
ios_base::badbit	Ce bit est à 1 lorsqu'une opération invalide a été tentée ; en principe le flot peut continuer à être utilisé mais ce n'est pas certain.

4.1.6 – Compléments -> Les Flots de Donnés : État d'erreur des flux : Méthode



Fonctions membres de ios_base	Description
bool good()	renvoie une valeur non nulle si aucun bit d'erreur est activé
bool eof()	retourne une valeur non nulle si ios::eofbit est activé
bool fail()	renvoie 1 si l'un des trois bits ios::badbit ou ios::failbit ou ios::hardfail est à 1, et 0 sinon
operator void* () operator const void* ()	convertit le flux courant en pointeur pour pouvoir le comparer à NULL. Retourne 0 si ios::failbit ou ios::badbit sont activés.
iostate rdstate()	retourne toutes les valeurs courantes de l'état d'erreur
bool operator()()	renvoie 1 si l'un des bits d'état est à 1 (flux incorrect), 0 sinon
bool bad()	renvoie 1 si l'un des deux bits ios::badbit ou ios::hardfail est à 1, 0 sinon
void clear(int i = 0)	permet de modifier l'état du flux. Par exemple, l'écriture fl.clear(ios::failbit) positionne le bit ios::failbit du flux fl, indiquant une erreur grave.

4.1.7 -- Compléments -> Les Flots de Donnés : État du format des flux



Option de Format Rôle	Description
ios::skipws	supprime les espaces tapés pendant la saisie de nombres
ios::left1	justifie les valeurs à gauche -1.25e10xxx
ios::right1	justifie les valeurs à droite xxx-1.25e10
ios::internal1	ajoute des caractères de remplissage interne -x1.25ex10
ios::dec2	conversion décimale
ios::oct2	conversion octale
ios::hex2	conversion hexadécimale
ios::showbase	conversion en constante entière
ios::showpos	ajoute le caractère + pour les valeurs décimales positives +3.
ios::showpoint	affiche toujours la virgule et complète par des zéros 2.000
ios::scientific	affichage en notation scientifique 3.4e10
ios::fixed	affichage en notation décimale 123.987
ios::uppercase	affichage des caractères en majuscules 12F4, 10E34
ios::unitbuf	réalise un flush à chaque insertion



- 4.2.1 - Présentation
- 4.2.2 - Les containers
- 4.2.3 - Les containers : Caractéristiques
- 4.2.4 - Les itérateurs
- 4.2.5 - Les itérateurs : Les 5 Types
- 4.2.6 - Les algorithmes
- 4.2.7 - La classe string
- 4.2.8 - Opérations les plus courantes de la classe string



La bibliothèque STL (*Standard Template Library*) fournit un ensemble de composants C++, fonctions, classes, constantes et objets, qui permettent d'étendre le langage C++ en fournissant des fonctions basiques et quelques classes, objets et algorithmes standards régulièrement utilisés.

STL contient cinq types de composants : des containers, des itérateurs, des algorithmes, des objets fonctions et des adaptateurs.

Nous nous intéressons dans ce chapitre aux trois premiers composants.

4.2.2 – Compléments -> La bibliothèque standard (STL) : Les containers



Les containers sont des objets qui permettent de stocker d'autres objets.

- **vector** : container implantant les tableaux
- **list** : container implantant les listes doublement chaînées
- **deque** : container similaire au vector
- **set** : container implantant les ensembles où les éléments ne peuvent être présents au plus qu'en un seul exemplaire
- **multiset** : container implantant les ensembles où les éléments peuvent être présents en plusieurs exemplaires
- **map** : container implantant des ensembles où un type de données appelé clé est associé aux éléments à stocker. On ne peut associer qu'une seule valeur à une clé unique
- **multimap** : container similaire au map supportant l'association de plusieurs valeurs à une clé unique
- **stack** : adaptateur permettant de donner à un container le comportement d'une pile
- **queue** : adaptateur permettant de donner à un container le comportement d'une file
- **priority_queue** : adaptateur permettant de donner à un container le comportement d'une file avec priorités
- **array (C++11)**
- **forward_list (C++11)**
- **unordered_set (C++11)**
- **unordered_multiset (C++11)**
- **unordered_map (C++11)**
- **unordered_multimap (C++11)**



<http://www.cplusplus.com/reference/stl/>



Certains types prédéfinis (typedefs) :

size_type
pointer
const_pointer
reference
const_reference

Des types prédéfinis pour la création d'itérateurs :

iterator
const_iterator
reverse_iterator
const_reverse_iterator

constructor()

: pour la création de containers vides

copy-constructor()

destructor()

bool empty() const

: nombre d'éléments du container

size_type size() const

: capacité (mémoire occupée) maximum du container

size_type max_size() const

: remplacement de tout le contenu

container-ref operator=(const-container-ref)

: inverse tout le contenu

void swap(container-ref)

: teste l'égalité sur tous les éléments

bool operator==(const-container-ref)

bool operator<(const-container-ref)

: méthodes pour accéder au contenu

begin() et end()

insert()

4.2.3 – Compléments -> La bibliothèque standard (STL) : Les containers : Exemple



```
void ExempleVector()
{
    vector<int> v1;
    v1.push_back(1); v1.push_back(2); v1.push_back(3);
    for (int i=0;i< v1.size() ; i++) cout << "ExempleVector :
vector1["<< i << "]" << v1[i] << endl;

    vector<int> v2(3);      // Declare un vector de 3 éléments.
    v2[0] = 7;
    v2[1] = v2[0] + 3;
    v2[2] = v2[0] + v2[1]; // v2[0] == 7, v2[1] == 10, v2[2] == 17
    for (int i=0;i< v2.size() ; i++) cout << "ExempleVector :
vector2["<< i << "]" << v2[i] << endl;
}
```

4.2.3 – Compléments -> La bibliothèque standard (STL) : Les containers : Exemple <map>



```
// constructing maps
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
    bool operator() (const char& lhs, const char& rhs) const
    {return lhs<rhs;}
};

int main ()
{
    std::map<char,int> first;

    first['a']=10;
    first['b']=30;
    first['c']=50;
    first['d']=70;

    std::map<char,int> second (first.begin(),first.end());

    std::map<char,int> third (second);

    std::map<char,int,classcomp> fourth;           // class as Compare

    bool(*fn_pt) (char,char) = fncomp;
    std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare

    return 0;
}
```

<http://www.cplusplus.com/reference/map/map/map/>

4.2.3 – Compléments -> La bibliothèque standard (STL) : Les containers : Exemple <vector>



```
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
    // constructors used in the same order as described above:
    std::vector<int> first;                                // empty vector of ints
    std::vector<int> second (4,100);                         // four ints with value 100
    std::vector<int> third (second.begin(),second.end());   // iterating through second
    std::vector<int> fourth (third);                          // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    std::cout << "The contents of fifth are:";
    for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

<http://www.cplusplus.com/reference/vector/vector/vector/>

4.2.3 – Compléments -> La bibliothèque standard (STL) : Les containers : Exemple <list>



```
// constructing lists
#include <iostream>
#include <list>

int main ()
{
    // constructors used in the same order as described above:
    std::list<int> first;                                // empty list of ints
    std::list<int> second (4,100);                         // four ints with value 100
    std::list<int> third (second.begin(),second.end());   // iterating through second
    std::list<int> fourth (third);                          // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    std::cout << "The contents of fifth are: ";
    for (std::list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
        std::cout << *it << ' ';

    std::cout << '\n';

    return 0;
}
```

<http://www.cplusplus.com/reference/list/list/list/>



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Les itérateurs sont une généralisation des pointeurs.

Ils permettent de spécifier une position à l'intérieur d'un container, peuvent être incrémentés ou déréférencés.

Tous les containers sont dotés d'une méthode `begin` qui renvoie un itérateur sur le premier de leurs éléments, et d'une méthode `end` qui renvoie un itérateur.

On ne peut ainsi pas déréférencer l'itérateur renvoyé par la méthode `end`.

4.2.5 – Compléments -> La bibliothèque standard (STL) : Les itérateurs : Les 5 Types



Il existe 5 types d'itérateurs: accès aléatoire, bidirectionnel, en avant, lecture, écriture.

- Les itérateurs d'entrée (*input iterators*) : ils permettent d'accéder séquentiellement à des sources de données. Cette source peut-être un container, un flot.
- Les itérateurs de sortie (*output iterators*) : ils permettent de préciser la localisation d'une destination permettant de stocker des données. Cette source peut-être un container, un flot.
- Les itérateurs à sens-unique (*forward iterators*) : ils sont dotés de toutes les méthodes des itérateurs d'entrée et de sortie. Ils sont utilisés pour parcourir séquentiellement une séquence de données dans un sens. Ils ne peuvent pas être utilisés pour effectuer des retours en arrière.
- Les itérateurs à double-sens (*bidirectional iterators*) : ils sont dotés de toutes les méthodes des itérateurs à sens-unique. Ils sont également utilisés pour effectuer des parcours séquentiels de données, qu'ils peuvent effectuer dans les deux sens.
- Les itérateurs à accès direct (*random-access iterators*) : ils sont dotés de toutes les méthodes des itérateurs à double-sens. Ils permettent d'accéder directement à des valeurs contenues dans un container, sans être obligé d'y accéder séquentiellement.



```
void ExempleIteratorVector()
{
    vector<int> v1;
    for (int i=0;i< 10 ; i++) v1.push_back(i);
    vector<int>::iterator itv;
    for (itv = v1.begin(); itv != v1.end(); itv++)
    {
        (*itv) = (*itv) + 1;
    }

    for (int i=0;i< v1.size() ; i++) cout << "ExempleIteratorVector :
vector1["<< i << "] " << v1[i] << endl;
}
```

4.2.5 – Compléments -> La bibliothèque standard (STL) : Les itérateurs : <map>



```
// map::begin/end
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;

    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;

    // show content:
    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
        std::cout << it->first << " => " << it->second << '\n';

    return 0;
}
```

4.2.5 – Compléments -> La bibliothèque standard (STL) : Les itérateurs : Exemple <vector>



```
// std::begin / std::end example
#include <iostream>          // std::cout
#include <vector>           // std::vector, std::begin, std::end

int main () {
    int foo[] = {10,20,30,40,50};
    std::vector<int> bar;

    // iterate foo: inserting into bar
    for (auto it = std::begin(foo); it!=std::end(foo); ++it)
        bar.push_back(*it);

    // iterate bar: print contents:
    std::cout << "bar contains:";
    for (auto it = std::begin(bar); it!=std::end(bar); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

4.2.5 – Compléments -> La bibliothèque standard (STL) : Les itérateurs : Exemple <list>



```
// list::begin
#include <iostream>
#include <list>

int main ()
{
    int myints[] = {75,23,65,42,13};
    std::list<int> mylist (myints,myints+5);

    std::cout << "mylist contains:";
    for (std::list<int>::iterator it=mylist.begin(); it != mylist.end(); ++it)
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



4.2.6 – Compléments -> La bibliothèque standard (STL) : Les algorithmes



Les algorithmes sont des fonctions C++ génériques qui permettent d'effectuer des opérations sur les containers.

```
void ExempleAlgorithme()
{
    vector<int> v1;
    for (int i=0;i< 10 ; i++) v1.push_back(i);
    vector<int>::iterator itv1 = v1.begin();
    for_each(itv1,v1.end(), Add());
    cout << "ExempleAlgorithme : for_each " << *itv1 << endl;
    for (int i=0;i< v1.size() ; i++) cout << "ExempleAlgorithme : vector1["<< i <<
    "]" << v1[i] << endl;
    vector<int>::iterator itv2 = find ( v1.begin(),v1.end(),10);
    cout << "ExempleAlgorithme : find " << *itv2 << endl;
    itv2++;
    cout << "ExempleAlgorithme : find " << *itv2 << endl;
}
```

<http://www.cplusplus.com/reference/algorithm/>

<algorithm>	
adjacent_find	[C++11]
all_of	[C++11]
any_of	[C++11]
binary_search	
copy	
copy_backward	
copy_if	
copy_n	[C++11]
count	
count_if	
equal	
equal_range	
fill	
fill_n	
find	
find_end	
find_first_of	
find_if	
find_if_not	
for_each	
generate	
generate_n	
includes	
inplace_merge	
is_heap	
is_heap_until	
is_permutation	
is_sorted	
is_sorted_until	
iter_swap	





4.2.6 – Compléments -> La bibliothèque standard (STL) : Les algorithmes : Exemple std::search



Example

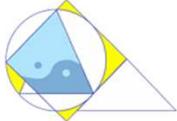
```
1 // search algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::search
4 #include <vector>              // std::vector
5
6 bool mypredicate (int i, int j) {
7     return (i==j);
8 }
9
10 int main () {
11     std::vector<int> haystack;
12
13     // set some values:      haystack: 10 20 30 40 50 60 70 80 90
14     for (int i=1; i<10; i++) haystack.push_back(i*10);
15
16     // using default comparison:
17     int needle1[] = {40,50,60,70};
18     std::vector<int>::iterator it;
19     it = std::search (haystack.begin(), haystack.end(), needle1, needle1+4);
20
21     if (it!=haystack.end())
22         std::cout << "needle1 found at position " << (it-haystack.begin()) << '\n';
23     else
24         std::cout << "needle1 not found\n";
25
26     // using predicate comparison:
27     int needle2[] = {20,30,50};
28     it = std::search (haystack.begin(), haystack.end(), needle2, needle2+3, mypredicate);
29
30     if (it!=haystack.end())
31         std::cout << "needle2 found at position " << (it-haystack.begin()) << '\n';
32     else
33         std::cout << "needle2 not found\n";
34
35     return 0;
36 }
```

- Output:

```
needle1 found at position 3
needle2 not found
```



<http://www.cplusplus.com/reference/algorithm/search/>



4.2.6 – Compléments -> La bibliothèque standard (STL) : Les algorithmes : Exemple std::sort



Example

```
1 // sort algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::sort
4 #include <vector>             // std::vector
5
6 bool myfunction (int i,int j) { return (i<j); }
7
8 struct myclass {
9     bool operator() (int i,int j) { return (i<j);}
10} myobject;
11
12 int main () {
13     int myints[] = {32,71,12,45,26,80,53,33};
14     std::vector<int> myvector (myints, myints+8);           // 32 71 12 45 26 80 53 33
15
16     // using default comparison (operator <):
17     std::sort (myvector.begin(), myvector.begin()+4);         //(12 32 45 71)26 80 53 33
18
19     // using function as comp
20     std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)
21
22     // using object as comp
23     std::sort (myvector.begin(), myvector.end(), myobject);    //(12 26 32 33 45 53 71 80)
24
25     // print out content:
26     std::cout << "myvector contains:";
27     for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
28         std::cout << ' ' << *it;
29     std::cout << '\n';
30
31     return 0;
32 }
```

Output:

```
myvector contains: 12 26 32 33 45 53 71 80
```

<http://www.cplusplus.com/reference/algorithm/sort/>



Example

```
1 // for_each example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::for_each
4 #include <vector>             // std::vector
5
6 void myfunction (int i) {    // function:
7     std::cout << ' ' << i;
8 }
9
10 struct myclass {           // function object type:
11     void operator() (int i) {std::cout << ' ' << i;}
12 } myobject;
13
14 int main () {
15     std::vector<int> myvector;
16     myvector.push_back(10);
17     myvector.push_back(20);
18     myvector.push_back(30);
19
20     std::cout << "myvector contains:";
21     for_each (myvector.begin(), myvector.end(), myfunction);
22     std::cout << '\n';
23
24 // or:
25     std::cout << "myvector contains:";
26     for_each (myvector.begin(), myvector.end(), myobject);
27     std::cout << '\n';
28
29     return 0;
30 }
```

Output:

```
myvector contains: 10 20 30
myvector contains: 10 20 30
```

http://www.cplusplus.com/reference/algorithm/for_each/

020



4.2.6 – Compléments -> La bibliothèque standard (STL) : Les algorithmes : Exemple std::generate



Example

```
1 // generate algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::generate
4 #include <vector>              // std::vector
5 #include <ctime>                // std::time
6 #include <cstdlib>             // std::rand, std::srand
7
8 // function generator:
9 int RandomNumber () { return (std::rand()%100); }
10
11 // class generator:
12 struct c_unique {
13     int current;
14     c_unique() {current=0;}
15     int operator()() {return ++current;}
16 } UniqueNumber;
17
18 int main () {
19     std::srand ( unsigned ( std::time(0) ) );
20
21     std::vector<int> myvector (8);
22
23     std::generate (myvector.begin(), myvector.end(), RandomNumber);
24
25     std::cout << "myvector contains:";
26     for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
27         std::cout << ' ' << *it;
28     std::cout << '\n';
29
30     std::generate (myvector.begin(), myvector.end(), UniqueNumber);
31
32     std::cout << "myvector contains:";
33     for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
34         std::cout << ' ' << *it;
35     std::cout << '\n';
36
37     return 0;
38 }
```

A possible output:

```
myvector contains: 57 87 76 66 85 54 17 15
myvector contains: 1 2 3 4 5 6 7 8
```

<http://www.cplusplus.com/reference/algorithm/generate/>



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020

4.2.7 – Compléments -> La bibliothèque standard (STL) : La classe string



La bibliothèque standard de C++ définit le type générique `basic_string` ainsi que ses instanciations standard `string` et `wstring`.

Ce dernier permet d'utiliser des jeux de caractères plus étendus que le type standard `char` (Unicode ou des jeux de caractères asiatiques).



The screenshot shows the structure of the `<string>` header file. It includes:

- class templates:**
 - `basic_string`
 - `char_traits`
- classes:**
 - `string`
 - `u16string`
 - `u32string`
 - `wstring`
- functions:**
 - `stod`
 - `stof`
 - `stoi`
 - `stol`
 - `stold`
 - `stoll`
 - `stoul`
 - `stoull`
 - `to_string`
 - `to_wstring`

Below this, a separate list shows the member functions of the `string` class:

- `string::string`
- `string::~string`
- member functions:**
 - `string::append`
 - `string::assign`
 - `string::at`
 - `string::back`
 - `string::begin`
 - `string::capacity`
 - `string::cbegin`
 - `string::cend`
 - `string::clear`
 - `string::compare`
 - `string::copy`
 - `string::crbegin`
 - `string::crend`
 - `string::c_str`
 - `string::data`
 - `string::empty`
 - `string::end`
 - `string::erase`
 - `string::find`
 - `string::find_first_not_of`
 - `string::find_first_of`
 - `string::find_last_not_of`
 - `string::find_last_of`
 - `string::front`
 - `string::get_allocator`

4.2.7 – Compléments -> La bibliothèque standard (STL) : La classe string (C++11)



Type	Définition
std::string	std::basic_string<char>
std::wstring	std::basic_string<wchar_t>
std::u16string	std::basic_string<char16_t> (C++11)
std::u32string	std::basic_string<char32_t> (C++11)

4.2.8 – Compléments -> La bibliothèque standard (STL) : Opérations les plus courantes de la classe string.



Opération	Effet
Constructeurs	<i>Il y a beaucoup d'autres constructeurs que nous ne détaillons pas ici</i>
string s	Crée une chaîne vide
string s(str)	Crée une copie de str
string s(beg,end)	Crée une chaîne initialisée par les caractères que l'on trouve entre les itérateurs beg et end
Méthodes	<i>il y a beaucoup d'autres méthodes que nous ne détaillons pas ici</i>
=, assign()	Affectation
+=, append(), push_back()	Concaténation de caractères
+	Concaténation de chaînes
==, !=, <, etc.	Comparaisons de chaînes
clear()	Efface tous les caractères (la chaîne devient vide)
empty()	Teste si la chaîne est vide
size(), length()	Taille de la chaîne (nombre de caractères)
[]], at()	Accède à un caractère donné par son indice dans la chaîne
data()	Donne un tableau de caractères construit à partir de la chaîne
substr()	Extraction de sous-chaînes
begin(), end()	Fournit un accès de type itérateur sur une chaîne ;
Fonctions de recherche	<i>Il y a beaucoup d'autres fonctions que nous ne détaillons pas ici</i>
find()	Trouve la première occurrence d'un caractère
rfind()	Trouve la dernière occurrence d'un caractère
find_first_of()	Trouve la première occurrence d'un des caractères de la chaîne donnée en paramètre

4.2.8 – Compléments -> La bibliothèque standard (STL) : Exemple



```
void ExempleString() {  
    //déclaration d'une chaîne de caractères vide  
    string chaîne1, chaîne2, résultat;  
  
    //assignation  
    chaîne1 = "Je suis chaîne1";  
    chaîne2 = "Et moi chaîne2";  
  
    //on affiche grâce à la méthode c_str() de cette classe.  
    //c_str() nous renvoie un const char *  
  
    cout << chaîne1.c_str() << "\n" << chaîne2.c_str() << endl;  
  
    //copie dans une troisième chaîne le contenu de chaîne1  
    résultat = chaîne1;  
  
    //on ajoute un espace à la fin de la chaîne résultat  
    résultat += ' ';  
  
    //on ajoute chaîne2 à la fin de résultat  
    résultat += chaîne2;  
    cout << "Après concaténation : " << résultat << endl;  
}
```



4.2.8 – Compléments -> La bibliothèque standard (STL) : Exemple



Example

```
1 // string constructor
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string s0 ("Initial string");
8
9     // constructors used in the same order as described above:
10    std::string s1;
11    std::string s2 (s0);
12    std::string s3 (s0, 8, 3);
13    std::string s4 ("A character sequence");
14    std::string s5 ("Another character sequence", 12);
15    std::string s6a (10, 'x');
16    std::string s6b (10, 42);      // 42 is the ASCII code for '*'
17    std::string s7 (s0.begin(), s0.begin()+7);
18
19    std::cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: " << s3;
20    std::cout << "\ns4: " << s4 << "\ns5: " << s5 << "\ns6a: " << s6a;
21    std::cout << "\ns6b: " << s6b << "\ns7: " << s7 << '\n';
22    return 0;
23 }
```

Output:

```
s1:
s2: Initial string
s3: str
s4: A character sequence
s5: Another char
s6a: xxxxxxxxxxxx
s6b: ****
s7: Initial
```

<http://www.cplusplus.com/reference/string/string/string/>





Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



4.3.1 - Run-Time Type Information

4.3.1.1 - L'opérateur typeid

4.3.1.2 - La classe type_info

4.3.2 - Transtypages C++

4.3.2.1 - Transtypage dynamique

4.3.2.2 - Transtypage statique

4.3.2.3 - Transtypage de constance et de volatilité

4.3.2.4 - Réinterprétation des données



Le RTTI (Run-Time Type Information) permet de déterminer le type d'un objet au moment de l'exécution grâce à :

- L'opérateur typeid
- la classe type_info

<typeinfo>
bad_cast
bad_typeid
type_info



L'opérateur typeid permet de récupérer les informations de type.

Déclaration :

`type_info typeid(expression)`

Example

```
1 // bad_typeid example
2 #include <iostream>           // std::cout
3 #include <typeinfo>           // operator typeid, std::bad_typeid
4
5 class Polymorphic {virtual void Member(){}};
6
7 int main () {
8     try
9     {
10         Polymorphic * pb = 0;
11         std::cout << typeid(*pb).name();
12     }
13     catch (std::bad_typeid& bt)
14     {
15         std::cerr << "bad_typeid caught: " << bt.what() << '\n';
16     }
17     return 0;
18 }
```

Possible output:

```
bad_typeid caught: St10bad_typeid
```

4.3.1.1 – Compléments -> Identification dynamique des types: Exemple



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

class Base
{
public:
    virtual ~Base( void );
};

Base::~Base( void )
{
return ;
}

class Derivee : public Base
{
public:
    virtual ~Derivee( void );
};

Derivee::~Derivee( void )
{
return ;
}

int main( void )
{
Derivee* pd = new Derivee;
Base* pb = pd;
const type_info &t1=typeid( *pd ); // t1 qualifie le type de *pd.
const type_info &t2=typeid( *pb ); // t2 qualifie le type de *pb.
return 0 ;
}
```



Yantra Technologies

4.3.1.1 – Compléments -> Identification dynamique des types: Exemple



Example

```
1 // comparing type_info objects
2 #include <iostream>    // std::cout
3 #include <typeinfo>    // operator typeid
4
5 struct Base {};
6 struct Derived : Base {};
7 struct Poly_Base {virtual void Member(){}};
8 struct Poly_Derived: Poly_Base {};
9
10 typedef int my_int_type;
11
12 int main() {
13     std::cout << std::boolalpha;
14
15     // fundamental types:
16     std::cout << "int vs my_int_type: ";
17     std::cout << ( typeid(int) == typeid(my_int_type) ) << '\n';
18
19     // class types:
20     std::cout << "Base vs Derived: ";
21     std::cout << ( typeid(Base)==typeid(Derived) ) << '\n';
22
23     // non-polymorphic object:
24     Base* pbase = new Derived;
25
26     std::cout << "Base vs *pbase: ";
27     std::cout << ( typeid(Base)==typeid(*pbase) ) << '\n';
28
29     // polymorphic object:
30     Poly_Base* ppolybase = new Poly_Derived;
31
32     std::cout << "Poly_Base vs *ppolybase: ";
33     std::cout << ( typeid(Poly_Base)==typeid(*ppolybase) ) << '\n';
34
35     return 0;
36 }
```

Output:

```
int vs my_int_type: true
Base vs Derived: false
Base vs *pbase: true
Poly_Base vs *ppolybase: false
```

http://www.cplusplus.com/reference/typeinfo/type_info/operator==/



4.3.1.1 – Compléments -> Identification dynamique des types: Exemple



Example

```
1 // type_info::name example
2 #include <iostream>           // std::cout
3 #include <typeinfo>          // operator typeid
4
5 int main() {
6     int i;
7     int * pi;
8     std::cout << "int is: " << typeid(int).name() << '\n';
9     std::cout << " i is: " << typeid(i).name() << '\n';
10    std::cout << " pi is: " << typeid(pi).name() << '\n';
11    std::cout << "*pi is: " << typeid(*pi).name() << '\n';
12
13    return 0;
14 }
```

Possible output (depends on library implementation):

```
int is: int
 i is: int
 pi is: int *
*pi is: int
```

http://www.cplusplus.com/reference/typeinfo/type_info/name/

4.3.1.2 – Compléments -> Identification dynamique des types: La classe type_info



Les informations de type sont enregistrées dans des objets de la classe type_info.

```
class type_info
{
public:
    virtual ~type_info( );
    bool operator==( const type_info &rhs ) const;
    bool operator!=( const type_info &rhs ) const;
    bool before( const type_info &rhs ) const;
    const char *name( ) const;
private:
    type_info( const type_info &rhs );
    type_info &operator=( const type_info &rhs );
};
```



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Le C++ fournit un jeu d'opérateurs de transtypage qui permettent de faire des vérifications dynamiques, et qui est donc plus sûr que le transtypage du C.

Ces opérations sont :

- transtypage dynamique, "*dynamic_cast*"
- transtypage statique, "*static_cast*"
- transtypage de constance, "*const_cast*"
- transtypage de réinterprétation, "*reinterpret_cast*"

<http://www.cplusplus.com/doc/tutorial/typecasting/>

4.3.2.1 – Compléments -> Identification dynamique des types: Transtypage dynamique



Le *transtypage dynamique* permet de convertir une expression en un pointeur ou une référence d'une classe, ou un pointeur sur void. Il effectue une vérification de la validité du transtypage.

Déclaration :

Type dynamic_cast<Type>(expression)

// La classe B hérite de la classe A :

```
B *pb;  
A *pA=dynamic_cast<A *>( pB ); équivalent à A *pA=pB;
```

Une exception de type `bad_cast` est lancée si l'expression caractérise un objet ou une référence d'objet
Attention : en cas d'erreur si le type cible est un pointeur, le pointeur nul est renvoyé mais pas d'exception .

La classe `bad_cast` est définie comme suit dans l'en-tête `typeinfo` :

```
class bad_cast : public exception  
{  
public:  
    bad_cast( void ) throw( );  
    bad_cast( const bad_cast& ) throw( );  
    bad_cast &operator=( const bad_cast& ) throw( );  
    virtual ~bad_cast( void ) throw( );  
    virtual const char* what( void ) const throw( );  
};
```

4.3.2.1 – Compléments -> Identification dynamique des types: Exemple



```
struct A
{
    virtual void f( void ) { return ; }

};

struct B : virtual public A
{ };

struct C : virtual public A, public B
{ };

struct D
{
    virtual void g( void ) { return ; }

};

struct E : public B, public C, public D
{ };

int main( void )
{
    E e;
    A *pA=&e;
    // C *pC=( C * ) pA;           // Illégal : A est une classe de base virtuelle ( erreur de compilation ).
    C *pC=dynamic_cast<C *>( pA ); // Légal. Transtypage

    D *pD=dynamic_cast<D *>( pC ); // Légal. Transtypage
    B *pB=dynamic_cast<B *>( pA ); // Légal, mais échouera à l'exécution ( ambiguïté ).

    return 0 ;
}
```

4.3.2.1 – Compléments -> Identification dynamique des types: Exemple



```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main () {
    try {
        Base * pba = new Derived;
        Base * pbb = new Base;
        Derived * pd;

        pd = dynamic_cast<Derived*>(pba);
        if (pd==0) cout << "Null pointer on first type-cast.\n";

        pd = dynamic_cast<Derived*>(pbb);
        if (pd==0) cout << "Null pointer on second type-cast.\n";

    } catch (exception& e) {cout << "Exception: " << e.what();}
    return 0;
}
```

Null pointer on second type-cast.



dynamic_cast, l'opérateur static_cast permet donc d'effectuer les conversions entre les types autres que les classes définies par l'utilisateur. Aucune vérification de la validité de la conversion n'a lieu cependant

Déclaration :

Type static_cast<Type>(expression)



```
int i = static_cast<int>(4.3)/3;  
std::cout << i << std::endl;  
double d = static_cast<double>(4)/3.;  
std::cout << d << std::endl;
```

```
. class Base {};  
. class Derived: public Base {};  
. Base * a = new Base;  
. Derived * b = static_cast<Derived*>(a);
```



L'opérateur `const_cast` peut travailler essentiellement avec des références et des pointeurs. Il permet de réaliser les transtypages dont le type destination est moins contraint que le type source vis-à-vis des mots clés `const` et `volatile`.

Déclaration:

Type `const_cast<Type>(expression)`

Exemple:

```
const char* nom ="nom";
char* mynom = const_cast<char*>(nom);
std::cout<< mynom << std::endl;
```

4.3.2.3 – Compléments -> Identification dynamique des types: Transtypage de constance et de volatilité : Exemple



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

void print (char * str)
{
    cout << str << '\n';
}

int main () {
    const char * c = "sample text";
    print ( const_cast<char *> (c) );
    return 0;
}
```

```
sample text
```

4.3.2.4 – Compléments -> Identification dynamique des types: Réinterprétation des données



L'opérateur de transtypage le plus dangereux est reinterpret_cast.

Déclaration:

Type reinterpret_cast<Type>(expression)

Aucune vérification de la validité de cette opération n'est faite.

```
double f=2.3;  
int i=1;  
reinterpret_cast<int &>( f )=i; *( ( int * ) &f )=i;
```

```
class A { /* ... */ };  
class B { /* ... */ };  
A * a = new A;  
B * b = reinterpret_cast<B*>(a);
```

L'opérateur reinterpret_cast doit cependant respecter les règles suivantes :

il ne doit pas permettre la suppression des attributs de constance et de volatilité ;

il doit être symétrique (c'est-à-dire que la réinterprétation d'un type T1 en tant que type T2, puis la réinterprétation du résultat en type T1 doit redonner l'objet initial).



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020



Yantra Technologies



D.Palermo

Programmation C++

Version 4.4 - 02/2020



Copyright : Yantra Technologies 2004-2020

4.4 – Compléments -> C librairies



```
... <cassert> (assert.h)
... <cctype> (ctype.h)
... <cerrno> (errno.h)
... <cfloat> (float.h)
... <ciso646> (iso646.h)
... <climits> (limits.h)
... <clocale> (locale.h)
... <cmath> (math.h)
... <csetjmp> (setjmp.h)
... <csignal> (signal.h)
... <cstdarg> (stdarg.h)
... <cstdbool> (stdbool.h) C++11
... <cstddef> (stddef.h) C++11
... <cstdint> (stdint.h) C++11
... <cstdio> (stdio.h)
... <cstdlib> (stdlib.h)
... <cstring> (string.h)
... <ctime> (time.h)
... <cuchar> (uchar.h) C++11
... <cwchar> (wchar.h) C++11
... <cwctype> (wctype.h) C++11
```

<http://www.cplusplus.com/reference/new/>



4.4 – Compléments -> C librairies : <cwctype> (wctype.h)

fx Functions

Character classification functions

They check whether the character passed as parameter belongs to a certain category:

iswalnum	Check if wide character is alphanumeric (function)
iswalpha	Check if wide character is alphabetic (function)
iswblank	Check if wide character is blank (function)
iswcntrl	Check if wide character is a control character (function)
iswdigit	Check if wide character is decimal digit (function)
iswgraph	Check if wide character has graphical representation (function)
iswlower	Check if wide character is lowercase letter (function)
iswprint	Check if wide character is printable (function)
iswpunct	Check if wide character is punctuation character (function)
iswspace	Check if wide character is a white-space (function)
iswupper	Check if wide character is uppercase letter (function)
iswdxdigit	Check if wide character is hexadecimal digit (function)

<http://www.cplusplus.com/reference/new/>



4.4 – Compléments -> C librairies : <cwctype> (wctype.h)



Character conversion functions

Two functions that convert between letter cases:

tolower	Convert uppercase wide character to lowercase (function)
towupper	Convert lowercase wide character to uppercase (function)

Extensible classification/conversion functions

iswctype	Check if wide character has property (function)
towctrans	Convert using transformation (function)
wctrans	Return character transformation (function)
wctype	Return character property (function)

Types

wctrans_t	Wide character transformation (type)
wctype_t	Wide character type (type)
wint_t	Wide character integral type (type)

Constants

WEOF	Wide End-of-File (constant)
----------------------	---

<http://www.cplusplus.com/reference/new/>

4.5 - Compléments -> C++ librairies : Conténaires



<code><array></code>	C++11
<code><bitset></code>	
<code><deque></code>	
<code><forward_list></code>	C++11
<code><list></code>	
<code><map></code>	
<code><queue></code>	
<code><set></code>	
<code><stack></code>	
<code><unordered_map></code>	C++11
<code><unordered_set></code>	C++11
<code><vector></code>	

<http://www.cplusplus.com/reference/new/>



template < class T, size_t N > class array;

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,10> myarray;
    unsigned int i;

    for (i=0; i<10; i++) myarray[i]=i;

    std::cout << "myarray contains:";
    for (i=0; i<10; i++)
        std::cout << ' ' << myarray[i];
    std::cout << '\n';

    return 0;
}
```

<http://www.cplusplus.com/reference/new/>



4.5 -- Compléments -> C++ librairies : Divers



```
<algorithm>  
<chrono>  
<codecvt>  
<complex>  
<exception>  
<functional>  
<initializer_list>  
<iterator>  
<limits>  
<locale>  
<memory>  
<new>
```

C++11

C++11

C++11

```
<numeric>  
<random>  
<ratio>  
<regex>  
<stdexcept>  
<string>  
<system_error>  
<tuple>  
<typeinfo>  
<type_traits>  
<utility>  
<valarray>
```

C++11

C++11

C++11

C++11

C++11

C++11

<http://www.cplusplus.com/reference/new/>



D.Palermo

Programmation C++

Version 4.4 - 02/2020

Copyright : Yantra Technologies 2004-2020

4.5 — Compléments -> C++ librairies : Divers <algorithm>



Functions in <algorithm>

Non-modifying sequence operations:

all_of <small>C++11</small>	Test condition on all elements in range (function template)
any_of <small>C++11</small>	Test if any element in range fulfills condition (function template)
none_of <small>C++11</small>	Test if no elements fulfill condition (function template)
for_each	Apply function to range (function template)
find	Find value in range (function template)
find_if	Find element in range (function template)
find_if_not <small>C++11</small>	Find element in range (negative condition) (function template)
find_end	Find last subsequence in range (function template)
find_first_of	Find element from set in range (function template)
adjacent_find	Find equal adjacent elements in range (function template)
count	Count appearances of value in range (function template)
count_if	Return number of elements in range satisfying condition (function template)
mismatch	Return first position where two ranges differ (function template)
equal	Test whether the elements in two ranges are equal (function template)
is_permutation <small>C++11</small>	Test whether range is permutation of another (function template)
search	Search range for subsequence (function template)
search_n	Search range for elements (function template)

<http://www.cplusplus.com/reference/new/>

4.5 -- Compléments -> C++ librairies : Divers <algorithme>



Modifying sequence operations:

<code>copy</code>	Copy range of elements (function template)
<code>copy_n</code> <small>C++11</small>	Copy elements (function template)
<code>copy_if</code> <small>C++11</small>	Copy certain elements of range (function template)
<code>copy_backward</code>	Copy range of elements backward (function template)
<code>move</code> <small>C++11</small>	Move range of elements (function template)
<code>move_backward</code> <small>C++11</small>	Move range of elements backward (function template)
<code>swap</code>	Exchange values of two objects (function template)
<code>swap_ranges</code>	Exchange values of two ranges (function template)
<code>iter_swap</code>	Exchange values of objects pointed by two iterators (function template)
<code>transform</code>	Transform range (function template)
<code>replace</code>	Replace value in range (function template)
<code>replace_if</code>	Replace values in range (function template)
<code>replace_copy</code>	Copy range replacing value (function template)
<code>replace_copy_if</code>	Copy range replacing value (function template)
<code>fill</code>	Fill range with value (function template)
<code>fill_n</code>	Fill sequence with value (function template)
<code>generate</code>	Generate values for range with function (function template)
<code>generate_n</code>	Generate values for sequence with function (function template)
<code>remove</code>	Remove value from range (function template)
<code>remove_if</code>	Remove elements from range (function template)
<code>remove_copy</code>	Copy range removing value (function template)
<code>remove_copy_if</code>	Copy range removing values (function template)
<code>unique</code>	Remove consecutive duplicates in range (function template)
<code>unique_copy</code>	Copy range removing duplicates (function template)
<code>reverse</code>	Reverse range (function template)
<code>reverse_copy</code>	Copy range reversed (function template)
<code>rotate</code>	Rotate left the elements in range (function template)
<code>rotate_copy</code>	Copy range rotated left (function template)
<code>random_shuffle</code>	Randomly rearrange elements in range (function template)
<code>shuffle</code> <small>C++11</small>	Randomly rearrange elements in range using generator (function template)

<http://www.cplusplus.com/reference/new/>

4.5 — Compléments -> C++ librairies : Divers <algorithms>



Partitions:

is_partitioned	Test whether range is partitioned (function template)
partition	Partition range in two (function template)
stable_partition	Partition range in two - stable ordering (function template)
partition_copy	Partition range into two (function template)
partition_point	Get partition point (function template)

Sorting:

sort	Sort elements in range (function template)
stable_sort	Sort elements preserving order of equivalents (function template)
partial_sort	Partially sort elements in range (function template)
partial_sort_copy	Copy and partially sort range (function template)
is_sorted	Check whether range is sorted (function template)
is_sorted_until	Find first unsorted element in range (function template)
nth_element	Sort element in range (function template)

Binary search (operating on partitioned/sorted ranges):

lower_bound	Return iterator to lower bound (function template)
upper_bound	Return iterator to upper bound (function template)
equal_range	Get subrange of equal elements (function template)
binary_search	Test if value exists in sorted sequence (function template)

Merge (operating on sorted ranges):

merge	Merge sorted ranges (function template)
inplace_merge	Merge consecutive sorted ranges (function template)
includes	Test whether sorted range includes another sorted range (function template)
set_union	Union of two sorted ranges (function template)
set_intersection	Intersection of two sorted ranges (function template)
set_difference	Difference of two sorted ranges (function template)
set_symmetric_difference	Symmetric difference of two sorted ranges (function template)

<http://www.cplusplus.com/reference/new/>



4.5 — Compléments -> C++ librairies : Divers <algorithm>



Heap:

push_heap	Push element into heap range (function template)
pop_heap	Pop element from heap range (function template)
make_heap	Make heap from range (function template)
sort_heap	Sort elements of heap (function template)
is_heap C++11	Test if range is heap (function template)
is_heap_until C++11	Find first element not in heap order (function template)

Min/max:

min	Return the smallest (function template)
max	Return the largest (function template)
minmax C++11	Return smallest and largest elements (function template)
min_element	Return smallest element in range (function template)
max_element	Return largest element in range (function template)
minmax_element C++11	Return smallest and largest elements in range (function template)

Other:

lexicographical_compare	Lexicographical less-than comparison (function template)
next_permutation	Transform range to next permutation (function template)
prev_permutation	Transform range to previous permutation (function template)

<http://www.cplusplus.com/reference/new/>





4.5 -- Compléments -> C++ librairies : Divers <memory>



Allocators

allocator	Default allocator (class template)
allocator_arg C++11	Allocator arg (object)
allocator_arg_t C++11	Allocator arg type (class)
allocator_traits C++11	Allocator traits (class template)

Managed pointers

auto_ptr	Automatic Pointer [deprecated] (class template)
auto_ptr_ref	Reference to automatic pointer (class template)
shared_ptr C++11	Shared pointer (class template)
weak_ptr C++11	Weak shared pointer (class template)
unique_ptr C++11	Unique pointer (class template)
default_delete C++11	Default deleter (class template)

Functions and classes related to `shared_ptr`:

make_shared C++11	Make shared_ptr (function template)
allocate_shared C++11	Allocate shared_ptr (function template)
static_pointer_cast C++11	Static cast of shared_ptr (function template)
dynamic_pointer_cast C++11	Dynamic cast of shared_ptr (function template)
const_pointer_cast C++11	Const cast of shared_ptr (function template)
get_deleter C++11	Get deleter from shared_ptr (function template)
owner_less C++11	Owner-based less-than operation (class template)
enable_shared_from_this C++11	Enable shared_from_this (class template)

<http://www.cplusplus.com/reference/new/>





4.5 — Compléments -> C++ librairies : Divers <memory>



Uninitialized memory

Raw storage iterator:

[raw_storage_iterator](#) Raw storage iterator (class template)

Temporary buffers:

[get_temporary_buffer](#) Get block of temporary memory (function template)

[return_temporary_buffer](#) Return block of temporary memory (function template)

Specialized algorithms:

[uninitialized_copy](#) Copy block of memory (function template)

[uninitialized_copy_n](#) C++11 Copy block of memory (function template)

[uninitialized_fill](#) Fill block of memory (function template)

[uninitialized_fill_n](#) Fill block of memory (function template)

Memory model

[pointer_traits](#) C++11 Pointer traits (class template)

[pointer_safety](#) C++11 Pointer safety enum (enum class)

[declare_reachable](#) C++11 Declare pointer as reachable (function)

[undeclare_reachable](#) C++11 Undeclare pointer as reachable (function template)

[declare_no_pointers](#) C++11 Declare memory block as containing no pointers (function)

[undeclare_no_pointers](#) C++11 Undeclare memory block as containing no pointers (function)

[get_pointer_safety](#) C++11 Get pointer safety (function)

[align](#) C++11 Align in range (function)

[addressof](#) C++11 Address of object or function (function template)

<http://www.cplusplus.com/reference/new/>





Bibliothèque de support des threads

<http://fr.cppreference.com/w/cpp/thread>

http://www.justsoftwaresolutions.co.uk/files/c++11_concurrency.pdf

<http://www.stdthread.co.uk/doc/headers/thread/thread.html>



4.5 -- Compléments -> C++ librairies : Divers <thread>



```
int a, b, c;
int calculateA()
{
    return a+a*b;
}
int calculateB()
{
    return a*(a+a*(a+1));
}
int calculateC()
{
    return b*(b+1)-b;
}
```

```
int main(int argc, char *argv[])
{
    getUserData();
    future<int> f1 = async(calculateB), f2 = async(calculateC);
    c = (calculateA() + f1.get()) * f2.get();
    showResult();
}
```

```
int main(int argc, char *argv[])
{
    getUserData(); // initializes a and b
    c = calculateA() * (calculateB() + calculateC());
    showResult();
}
```

<http://msdn.microsoft.com/fr-fr/magazine/hh852594.aspx>

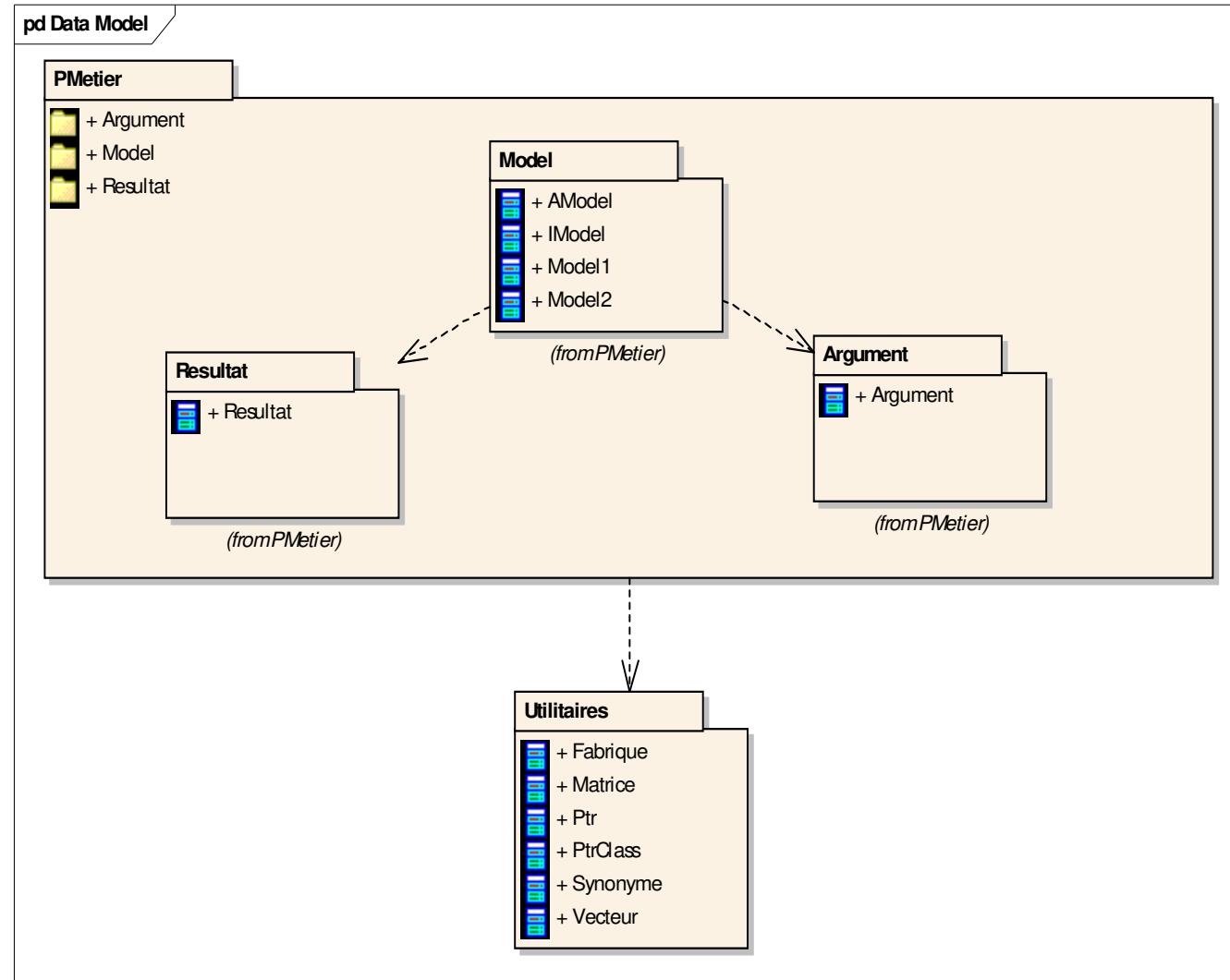


D.Palermo

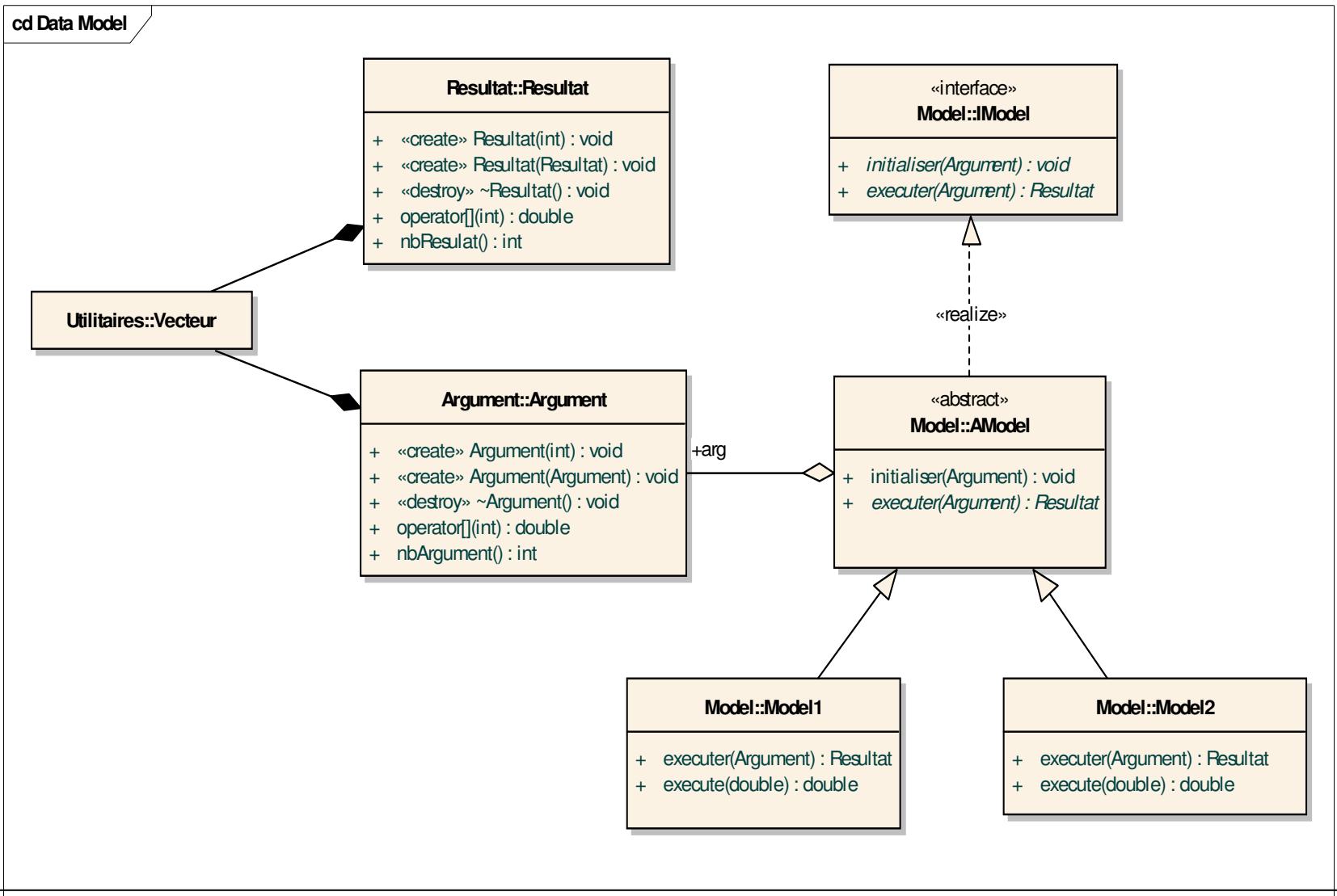
Programmation C++
Version 4.4 - 02/2020

 Copyright : Yantra Technologies 2004-2020

4.6 - - Compléments -> Exercices : Couplage



4.6 -- Compléments -> Exercices : Couplage



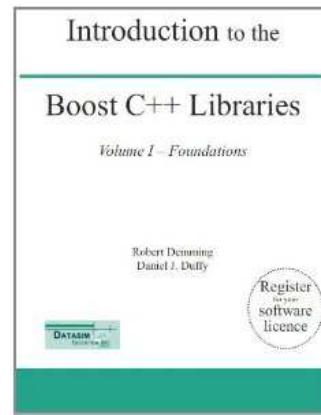
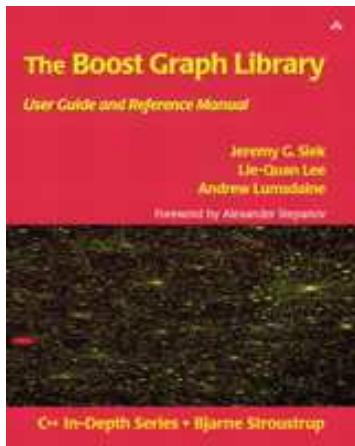
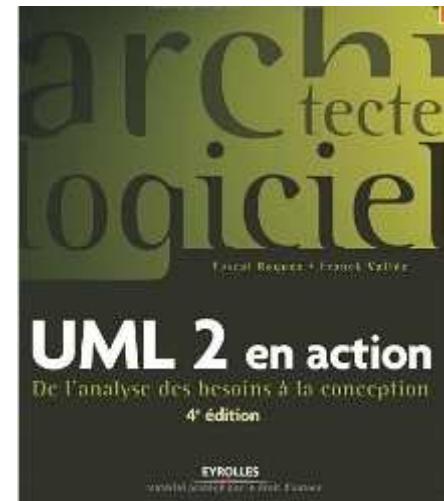
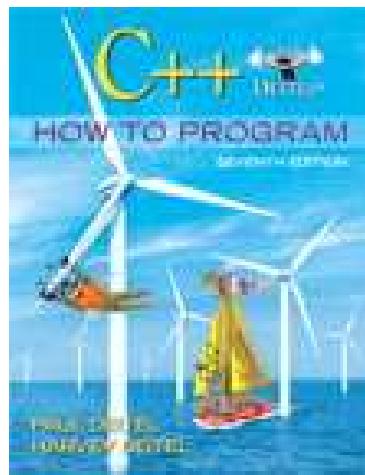
5.1 – Bibliographie



- **Design Patterns**, E.Gamma R.Helm, *Vuibert*
- **Le langage C++**, B.Stroustrup, *Campus Press*
- **C++ Comment Programmer**, Deitel and Deitel, *Reynald Goulet*
- **Programmation avancée en C++**, James O. Coplein, *Addison Wesley*
- **Effective STL**, S.Meyer , *Addison Wesley*
- **Effective C++**, S.Meyer , *Addison Wesley*
- **Modern C++ Design**, Andrei Alexandrescu
- **C++ Template Metaprogramming**, David Abrahams, Aleksey Gurtovoy
- **Exceptional C++ Style**, Herb Suter
- **More Exceptional C++**, Herb Suter
- **Pour mieux développer avec C++**, Aurélien Géron, Fatmé Tawbi
- **Modélisation objet avec UML**, P. Muller, N. Gaertner, *Eyrolles*
- **UML 2.0**, M. Fowler, Campus Press
- **Modélisation objet avec UML**, P. Muller, N. Gaertner, *Eyrolles*
- **L'eXtreme Programming**, J.Bénard, L.Bossavit, R.Médina, D.Williams , *Eyrolles*
- **Guide pratique du RUP**, P. Kruchten, P. Kroll , *Campus Press*
- **RUP, XP, architectures et outils**, P.Cloux , *Dunod*



5.2 – Bibliographie

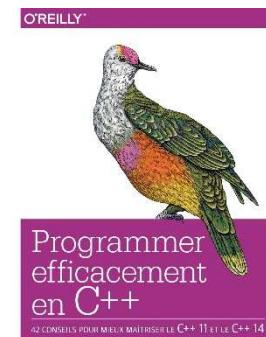
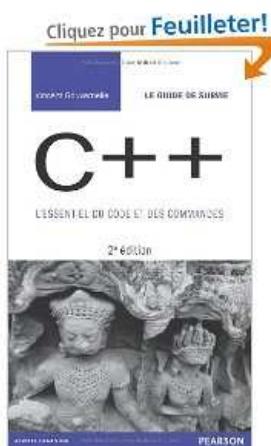
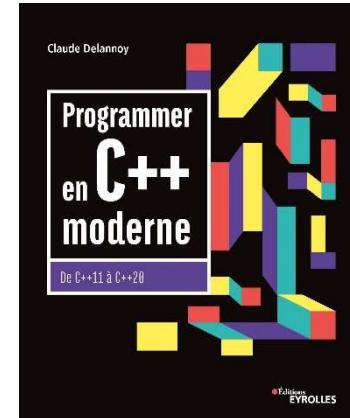
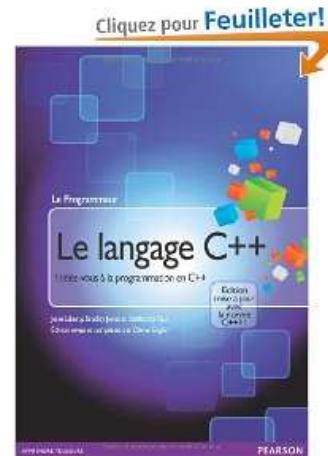
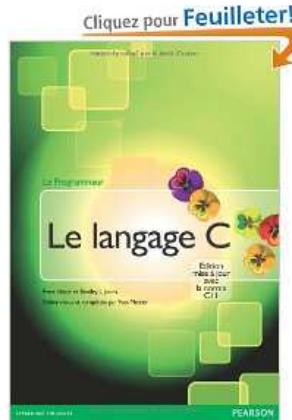


D.Palermo

Programmation C++
Version 4.4 - 02/2020



5.2 – Bibliographie



5.3 – Bibliographie internet



- Wikipédia :
 - C++ : <http://fr.wikipedia.org/wiki/C%2B%2B>
 - Design Patterns : http://fr.wikipedia.org/wiki/Patron_de_conception
 - Boost :
 - <http://www.boost.org>
 - <http://www.cplusplus.com>
 - <http://cpp.developpez.com>
 - <http://www.cs.brown.edu/~jwicks/boost>
- Divers :
 - <http://www.cppfrance.com/>
 - <http://cpp.developpez.com/>
 - <http://casteyde.christian.free.fr/>
 - <https://msdn.microsoft.com/fr-fr/library/3bstk3k5.aspx> : Référence du langage C++
 - <http://fr.cppreference.com>
 - <http://cpp.developpez.com/redaction/data/pages/users/gbdivers/cpp11/> : Nouvelles fonctionnalités du C++11