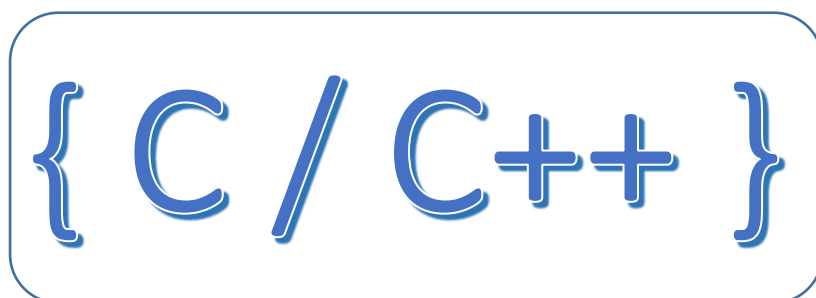


1	Premiers pas.....	3
2	Variables et types.....	4
3	Déclarations de variables et Strings .....	5
4	Entrées/Sorties.....	7
5	Constantes et caractères spéciaux.....	9
6	Opérateurs .....	10
7	Contrôles et Boucles.....	11
8	Fonctions .....	12
9	Surcharges et Templates .....	14
10	Espaces de noms, portées, variables statiques .....	15
11	Structures .....	16
12	Typedef/Using, Enumerations, Unions .....	17
13	Classes et Objets.....	18
14	Attribut et méthodes statiques (de classe) .....	20
15	Tableaux .....	21
16	Pointeurs .....	22
17	Allocation dynamique de mémoire.....	25
18	Fichiers et streams .....	28
19	Exceptions .....	30
20	UML Diagramme de classes .....	32
21	UML Diagramme de cas d'utilisation .....	34
22	UML Diagramme de séquence .....	34





# 1 PREMIERS PAS

Inclusion d'une bibliothèque pour certaines fonctions (ex : `std::cout`)

Début du programme principal (`main`)

Délimitation début et fin de bloc

Nom pas qualifié (erreur si pas le `using namespace std`)

Nom qualifié (on précise le `std::`)

```
/* Ceci est un commentaire
   réparti sur plusieurs lignes */

// ceci est un commentaire jusqu'à la fin de la ligne

#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World! " << std::endl;
    std::cout << "mon premier programme C++";
}
```

Précise d'aller chercher une fonction inconnue dans la bibliothèque `std` ( dispense d'écrire `std::` )

Affiche un message sur la console (`cout`)

Saut de ligne (`endl`)

## 2 VARIABLES ET TYPES

```
// Variables et adresse de variables

#include <iostream>
using namespace std;

int main()
{
    int maVar1 = 64;

    cout << "maVar1 vaut: " << maVar1 << endl;

    cout << "Adresse memoire de maVar1 : " << &maVar1;

    cout << endl;
}
```

Mémoire :

maVar1		
	64	
10FFBE8	10FFBEC	10FFBF0

maVar1 vaut: 64

Adresse memoire de maVar1 : 010FFBEC

Type variable (int) → `int`

Nom variable (maVar1) → `maVar1`

Valeur initiale variable (64) → `= 64`

### 4 grandes familles de types de variables de base :

- **Types caractères** : Ils peuvent représenter un caractère unique, comme 'A' ou '\$'. Le type le plus fondamental est le **char**, qui est un caractère d'un octet. D'autres types sont également prévus pour les caractères plus larges.
- **Types entiers numériques** : Ils peuvent stocker une valeur de nombre entier, tels que 7 ou 1024. Ils existent dans une grande variété de tailles et peuvent être soit **signé** ou **non signé**, selon qu'ils prennent en charge les valeurs négatives ou non. Les types **int**, **short** ou **long** sont les plus fréquents.
- **Types à virgule flottante** : Ils peuvent représenter des valeurs réelles, telles que 3.14 ou 0.01, avec différents niveaux de précision, le type **float** ou **double** sont les plus fréquents.
- **Type booléen** : Le type booléen, connu en C++ comme **bool**, ne peut représenter que deux états : vrai ou faux (true ou false).

→ Pour connaître les valeurs max contenues, par exemple pour un short (16 bits) faire  $2^{16} = 65536$ , donc de 0 à 65535 en unsigned et -32768 à 32767 en signé.

Exemple de tailles de variables (certaines dépendent de l'ordinateur) :

Taille de char :	1 octet	8 bits
Taille de wchar_t :	2 octets	16 bits
Taille de short :	2 octets	16 bits
Taille de int :	4 octets	32 bits
Taille de long :	4 octets	32 bits
Taille de long long :	8 octets	64 bits
Taille de float :	4 octets	32 bits
Taille de double :	8 octets	64 bits
Taille de long double :	8 octets	64 bits
Taille de bool :	1 octet	8 bits

### 3 DECLARATIONS DE VARIABLES ET STRINGS

```
// Variables et adresse de variables

#include <iostream>
using namespace std;

int main()
{
    int maVar1 = 64;    // déclarée et init style C
    int maVar2(26);    // déclarée et init style par constructeur
    int maVar3{ 12 };  // déclarée et init style uniformisé
    int maVar4;        // déclarée, pas initialisée

    cout << maVar1 << " " << maVar2 << " " << maVar3 << " " << maVar4;
    cout << endl;

    maVar4 = 45;        // maintenant maVar4 contient 45

    char autreVar{ 'A' }; // un caractère, entre côtes
    char caract{ '2' };   // les chiffres aussi sont des caractères
    double reel{ 3.12E-3 }; // un réel (flottant) 3,12x10-3
    float autreReel{ 4.0 }; // un autre
    bool etat{ true };    // true ou false

    auto nouvelleVar = autreReel; /* nouvelleVar prend automatiquement
                                   le type de autreReel (donc float)*/
}
```

maVar4 n'est pas  
initialisée, il y a  
n'importe quoi  
dedans.



64 26 12 997971

```
// String: chaines de caracteres
```

```
#include <iostream>
```

```
#include <string>
```

```
int main()
```

```
{
```

```
    std::string maChaine = "Hasparren";
```

```
    std::cout << maChaine.length() << std::endl;
```

```
    std::cout << maChaine.front() << std::endl;
```

```
    std::cout << maChaine.back() << std::endl;
```

```
    std::cout << maChaine.at(0) << std::endl;
```

```
    std::cout << maChaine.at(1) << std::endl;
```

```
    std::cout << maChaine.at(2) << std::endl;
```

```
    double temperature = 28.5;
```

```
    maChaine = maChaine + " " + std::to_string(temperature);
```

```
    std::cout << maChaine << " degre" << std::endl;
```

```
    return 0;
```

```
}
```

Bibliothèque contenant le type string et les « outils » qui vont avec.

9  
H  
n  
H  
a  
s

Concaténation de chaînes

Hasparren 28.500000 degre

On peut aussi faire beaucoup d'autres opérations sur les chaînes de caractères std::string (transformer un string en int, double, etc...) :

[http://en.cppreference.com/w/cpp/string/basic\\_string](http://en.cppreference.com/w/cpp/string/basic_string)

## 4 ENTREES/SORTIES

```
// Entrées/Sorties clavier/console : cin et cout
#include <iostream>
#include <string>
#include <Windows.h> // pour encodage console

using namespace std;

int main() {
    setlocale(LC_ALL, "French_France.1252"); // encodage Windows par défaut
    SetConsoleOutputCP(1252); // pour les cout, console encodée en 1252
    SetConsoleCP(1252); // pour les cin, clavier encodé en 1252

    string nom{ "Dupond" }; // déclarations et init avec valeurs par défaut
    string prenom{ "Peio" };
    string adresse{ "26, Larralde 64240 Hasparren" };
    int age{ 19 };
    double moyenne{ 20.0 };

    cout << "Bonjour, quel est votre age ?\t";
    cin >> age; cin.ignore(32767, '\n'); // cin et purge buffer

    cout << "Votre nom ? \t\t\t";
    getline(cin, nom); // getline pour le string

    cout << "et votre prénom ? \t\t";
    getline(cin, prenom); // getline pour le string

    cout << "Adresse ? \t\t\t";
    getline(cin, adresse); // getline pour le string
}
```

Gestion des caractères français  
et encodage (é, è, à, ç, etc.)

2 instructions :

- lecture valeur (cin)
- purger le buffer (cin.ignore(...))

Lecture d'une ligne entière, y  
compris les espaces (pour les  
string)

```

cout << "Moyenne au bac ? \t\t";
cin >> moyenne; cin.ignore(32767, '\n');    // cin et purge buffer

cout << "\nFélicitations " << nom
    << " " << prenom << " " << adresse << "\n"
    << "vous avez " << age << " ans,"
    << "et vous avez obtenu le bac avec une note moyenne de "
    << moyenne << " ." << endl;

return 0;
}

```

2 tab

1 saut de ligne (comme endl)

```

Bonjour, quel est votre nom ?   IRIGARAY
et quel est votre prénom ?     Julien
Adresse ?                      rue Pannecau 64100 Bayonne
Age ?                          18
Moyenne au bac ?               14.6

```

```

Félicitations IRIGARAY Julien rue Pannecau 64100 Bayonne
vous avez 18 ans,et vous avez obtenu le bac avec une note moyenne de 14.6 .

```

Pour plus d'information concernant la mise en forme des sorties sur cout : <http://en.cppreference.com/w/cpp/io/manip>

Attention : Normalement cin, cout, string, endl, getline(), cin.ignore() commencent tous par std::



## 5 CONSTANTES ET CARACTERES SPECIAUX

```
// caractères spéciaux et constantes
#include <iostream>      // pour cin, cout, etc.
#include <string>
#include <Windows.h>     // pour encodage console

#define PI 3.1416        // constante ancienne manière
int main()
{
    setlocale(LC_ALL, "French_France.1252");//pour encodage windows 1252 de base
    SetConsoleOutputCP(1252);              //pour les cout (encodage console 1252)
    SetConsoleCP(1252);                    //pour les cin (encodage clavier 1252)

    // caractères spéciaux à afficher précédés par \ (ex: \" affiche ")
    const std::string maChaine1{ "\\\"double backslash\\\""}; // chaîne constante
    std::cout << maChaine1 << '\n';          // affiche "\\\"double backslash\\"

    // caractères spéciaux à afficher encadrés par R"&&&( et )&&&
    std::string maChaine2{ R"&&&(@¥'\\"|"/'¥@)&&&" };
    std::cout << maChaine2 << '\n';          // affiche @¥'\\"|"/'¥@

    char monCar1{ '@' };                    // @ saisi depuis Notepad++
    char monCar2{ '\xA9' };                 // code hexa de © idem précédent
    std::cout << monCar1 << ' ' << monCar2;

    // efface ce qui pourrait trainer dans le buffer, équivalent à system("pause")
    std::cin.ignore(32767, '\n');
    std::cout << "Taper Entrée pour continuer";
    std::cin.ignore(32767, '\n');
}
```

## 6 OPERATEURS

```
// Opérateurs
int main()
{
    int var1(11);          // initialisation style C++ constructeur
    int var2{ 0 };         // initialisation style uniformisée
    bool compare = true;   // initialisation style C

    var2 = 4;              // affectation de 4 à var2 (de droite à gauche)

    // opérateurs arithmétiques +, -, *, /, % et cast
    float resultat = (float)var1 / (float)var2;    // resultat==2.75
    int reste = var1 % var2;                       // modulo: reste==3
    int partieEntiere = (var1 - reste) / var2;      // partieEntiere==2
    float partieDecimale = (float)reste / (float)var2; // partieDecimale==0.75

    var1 += 3;    // Ecriture composée, pareil que var1 = var1 + 3
    var2++;       // Incrément: var2 = var2 + 1
                // pareil que ++var2, sauf si on fait une affectation en même temps
    var2--;       // Décrément: var2 = var2 -1;

    // opérateurs relationnels <, <=, >, >=, ==, !=
    // opérateurs logiques &&(et), ||(ou), !(pas)
    int a{ 1 }, b{ 3 }, c{ 5 };
    compare = (a <= b) && (a > c);    // a<=b true, a>c false, true ET false compare==false
    compare = (a > 2) || (a != c);    // a>2 false, a!=c true, false OU true compare==true
    compare = !((a > 2) || (a != c)); // même que avant avec !(pas) devant compare==false

    // opérateur ternaire test vrai ? oui:non
    int result = b > c ? 12 : 14;    // b>c false donc result==14

    // opérateurs binaires &, |, ^, ~, <<, >>

Voir TP programmation communication  
série avec Arduino.


}
```

## 7 CONTROLES ET BOUCLES

```
int x{ 0 }, y{ 1 };

if (x>y)          // Test si sinon
{
    //...
}
else if (x < y)
{
    //...
}
else
{
    //...
}

while (x<y)       // tant que faire
{
    //...
}

do               // faire tant que
{
    //...
} while (y<x);

for (int i = 0; i < x; i++) // itération
{
    //...
}
```

```
for (char j : str)    // itération plage
{
    //...
}

while (x<y)
{
    // ...
    if (x==3)
        break;        // sort du while
    // ...
    if (x == 12)
        continue;     // saute lignes suivantes
                        // mais reste dans le while
    // ...
}

switch (y)            // switch
{
case 1:
    //...
    //...
    break;
case 6:
case 7:
    //...
    break;
default:
    //...
    break;
}
```

## 8 FONCTIONS

```
// déclaration de la fonction pythagore
double pythagore(double x, double y);
```



Il faut déclarer la fonction avant de l'utiliser :  
type de la valeur de retour, nom, signature.

```
int main()
{
    double u, v, w;
    cout << "Tapez la valeur de u : "; cin >> u;
    cout << "Tapez la valeur de v : "; cin >> v;

    w = pythagore(u, v); //appel de notre fonction

    cout << "Le résultat est " << w << endl;
    return 0;
}
```

La définition de la fonction peut alors être  
faite après son utilisation.



```
// définition de la fonction pythagore
double pythagore(double x, double y)
{
    double a;
    a = x*x + y*y;
    a = sqrt(a);
    return a;
}
```

L'appel passe en arguments deux valeurs de variables **u** et **v** à la fonction : ceux-ci correspondent aux paramètres **x** et **y**, déclarés pour la fonction pythagore (en tout cas le type doit correspondre, le nom n'est pas important). Le type et le nombre des paramètres est la **signature** de la fonction.

Le type de la fonction **pythagore** est double, donc nous retournons un double (**a**) et, on peut récupérer la valeur retournée (renvoyée) par la fonction pythagore : ici on la récupère dans **w**, qui est aussi un double. On ne peut retourner que la valeur d'une seule variable. Si la fonction ne fait pas de return elle est de type **void**.

Les variables **x**, **y** et **a** sont locales à la fonction.

C'est un passage de paramètres par valeurs : la valeur de **u** est copiée dans **x**, et celle de **v** dans **y**, mais pas l'inverse (**u** et **v** restent identiques).

On peut passer des paramètres par **référence** pour qu'ils soient modifiables dans la fonction : ce n'est plus la valeur qui est copiée dans une variable locale, mais un lien vers la variable de la fonction appelante.

La seule différence d'écriture est dans la déclaration-définition de la fonction ; il y a un **&** : `double pythagore(double& x, double& y)`

A l'appel de la fonction les paramètres sont passés de manière identique que précédemment (pas de **&**).

Passage par référence

```
int main()
{
    using namespace std;
    double angleDegré = 0;
    double angleRadian = 0;
    double cosPhiAngle = 0;
    cout << "Angle en degres : "; cin >> angleDegré; cin.ignore();

    if (calculcos(angleDegré, angleRadian, cosPhiAngle) == -1) cout << "Erreur" << endl;
    else cout << "L'angle en radians : " << angleRadian << "\nLe cos Phi : " << cosPhiAngle << endl;

    std::cout << "Appuyez une touche pour continuer";
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
```

angleRadian  
radian

$2\pi$

cosPhiAngle  
cosphi

angleDegré

360

Appel de la fonction et passage paramètres

Passage par valeur

```
int calculcos(double degré, double& radian, double& cosphi)
{
    const double pi = acos(-1);
    if (degré < 0 || degré > 360) return -1;
    else
    {
        radian = (degré / 360) * (2 * pi);
        cosphi = cos(radian);
        return 0;
    }
}
```

degré

360

## 9 SURCHARGES ET TEMPLATES

```
// Template functions
#include <iostream>
using namespace std;

template < typename typ1, typename typ2>
typ1 puissance(typ1 x, typ2 y)
{
    typ1 res{ 1 };
    for (typ2 i = 0; i < y; i++) res = res*x;
    return res;
}

int main()
{
    cout << puissance<int,int>(2, 4) << endl;

    cout << puissance<double,short>(3.6, 5) << endl;

    cout << puissance<long,long>(36, 15) << endl;

    return 0;
}
```

```
// Overload (surcharges) functions
#include <iostream>
using namespace std;

double puissance(double x, int y)
{
    double res{ 1 };
    for (int i = 0; i < y; i++) res = res*x;
    return res;
}

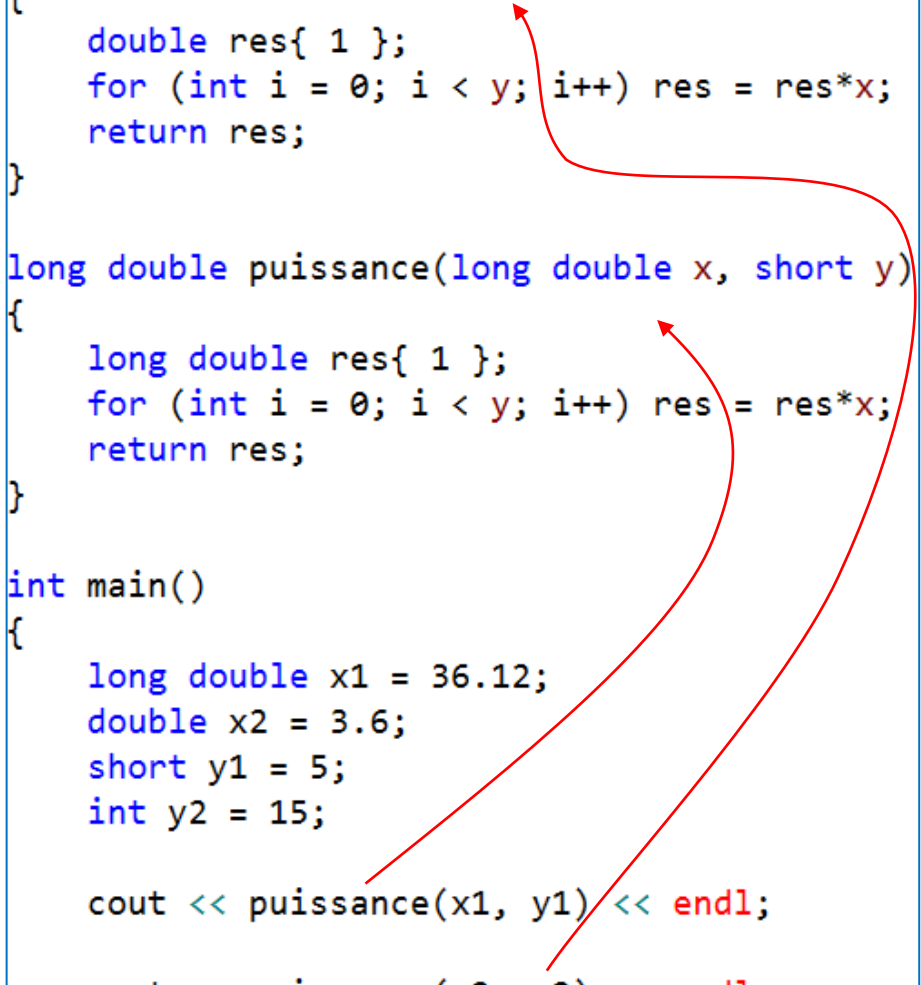
long double puissance(long double x, short y)
{
    long double res{ 1 };
    for (int i = 0; i < y; i++) res = res*x;
    return res;
}

int main()
{
    long double x1 = 36.12;
    double x2 = 3.6;
    short y1 = 5;
    int y2 = 15;

    cout << puissance(x1, y1) << endl;

    cout << puissance(x2, y2) << endl;

    return 0;
}
```



## 10 ESPACES DE NOMS, PORTEES, VARIABLES STATIQUES

```
int var1;           //globale à éviter

void fonct1 ()
{
    int var1 = 3; // var1 locale, masque l'autre
}

int main ()
{
    var1=1; // utilise la globale (à éviter)
    fonct1();
}
```

```
#include <iostream>

using std::cout;           // permet des accès non qualifiés
using std::endl;           // ciblés sur certaines fonctions

namespace finance          // namespace finance
{
    int value() { return 77; }
}

namespace mathema          // namespace math
{
    const double pi = 3.1416;
    double value() { return 2 * pi; }
}

int maxDesNombresVus(int nombre);

void main()
{
    cout << finance::value() << '\n'; // affiche: 77
    cout << mathema::value() << '\n'; // affiche: 6.2831999

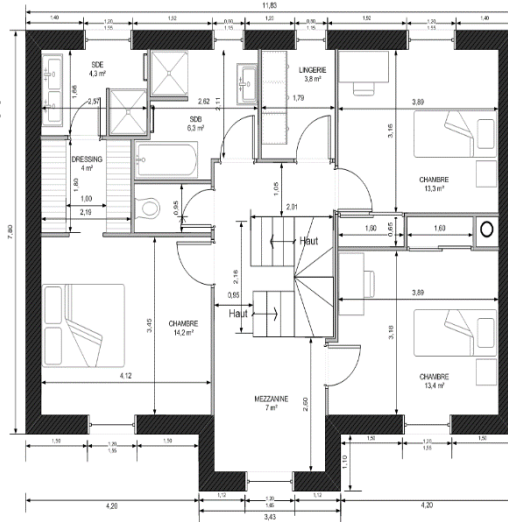
    cout << maxDesNombresVus(12) << endl; // affiche 12
    cout << maxDesNombresVus(8) << endl; // affiche 12
    cout << maxDesNombresVus(26) << endl; // affiche 26
}

int maxDesNombresVus(int nombre)
{
    static int memoMax = 0; // var static
    if (nombre > memoMax) memoMax = nombre;
    return memoMax;
}
```

# 11 STRUCTURES

## Plans de la maison :

Nbe de pièces,  
d'étages, surface,  
etc.



Construction des maisons

## Maisons (faites suivant les plans) :



```
struct maison
{
    int nbe_pieces;
    double longueur;
    double largeur;
    double surface;
    std::string adresse;
};
```

// les plans, le type

Déclaration des variables

```
int main()
{
    maison maison1{5, 10.2, 5.6, 0, "32 av du printemps"}; // objets, variables
    maison mais222{ 4, 8.2, 4.5, 0, "12 Larreko bidea" }; // de type maison

    calculSurface(maison1);
    mais222.surface = calculSurface(mais222.longueur, mais222.largeur);

    afficher(maison1);
}
```

```
void calculSurface(maison & chezMoi) {
    chezMoi.surface = chezMoi.longueur*chezMoi.largeur;
}
double calculSurface(double longueur, double largeur) {
    return longueur * largeur;
}
void afficher(const maison& maMaison) {
    std::cout << maMaison.adresse << " " << maMaison.nbe_pieces;
}
```



## 12 TYPEDEF/USING, ENUMERATIONS, UNIONS

### 12.1 Typedef et using

Permettent créer un alias (un autre nom) que l'on pourra utiliser à la place d'un type :

```
typedef char C;  
typedef unsigned int WORD;
```

, ou mieux :

```
using C = char;  
using WORD = unsigned int;
```

permet de faire :

```
C mychar, anotherchar;  
WORD myword;
```

### 12.2 Enumérations

C'est un type particulier composé d'un ensemble prédéfini de valeurs.

```
enum colors_t { black, blue, green, cyan, red, purple, yellow, white };
```

permet de faire :

```
colors_t mycolor;  
  
mycolor = blue;  
if (mycolor == green) mycolor = red;
```

Ou mieux :

```
enum class Colors { black, blue, green, cyan, red, purple, yellow, white };
```

Qui permet de faire :

```
Colors maCouleur;  
  
maCouleur = Colors::blue;  
if (maCouleur == Colors::green) maCouleur = Colors::red;
```

### 12.3 Unions

Ressemble à une struct, mais **grosse différence**, à l'intérieur il n'y a pas toutes les variables mais **une seule à la fois** !

```
union mytypes_t {  
    char c;  
    int i;  
    float f;  
} mytypes;
```

Suivant ce que l'on met dedans il y a soit un char, soit un int, soit un float, mais un seul à la fois.

Si vous avez affecté un char dans l'objet mytypes, et que plus tard vous y affectez un float, alors vous écrasez le char.

C'est un objet « fourre-tout », on peut y mettre « ce que l'on veut », mais une seule chose à la fois.

# 13 CLASSES ET OBJETS

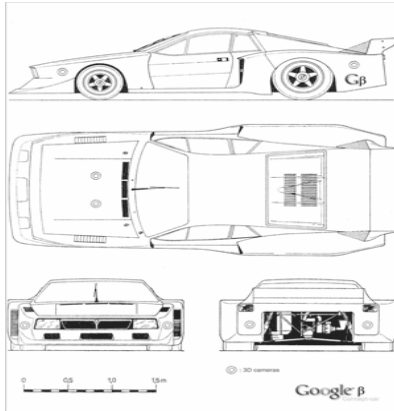
**Plans de la voiture (Classe) :**

**Propriétés :**

Nbe places, ...

**Méthodes :**

Rouler, ...



```
// Voiture.h
#pragma once
class Voiture {
    short places;           // attribut (privé)
    short puissFisc;        // attribut (privé)
public:
    Voiture();              // constructeur par défaut
    Voiture(short nbe, short puissFisc); // constructeur avec paramètres
    void demarrer();        // méthode pour démarrer
    void rouler(double vitesse); // méthode pour faire rouler l'objet
    void set_places(short nbe); // Mutateur/Setter: écrire l'attribut privé
    short get_places();     // Accesseur/Getter: lire l'attribut privé
};
```

```
// Voiture.cpp
#include "Voiture.h"

Voiture::Voiture() : places{ 5 }, puissFisc{ 4 } { } // constructeur par défaut
Voiture::Voiture(short nbe, short puissFisc) :    // avec paramètres, init commence par :
    places{ nbe }, puissFisc(puissFisc)           // init uniforme avec {} ou ()
{
    demarrer();                                     // appel méthode dans corps constructeur
}

void Voiture::demarrer() { ... } // méthode pour démarrer l'objet
void Voiture::rouler(double vitesse) { ... } // méthode pour faire rouler l'objet
void Voiture::set_places(short places) // méthode écrire l'attribut privé
{ this->places = places; }
short Voiture::get_places() { return places; } // méthode lire l'attribut privé
```

**Voitures (Objets) faits suivant les plans) :**



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

void main()
{
    // Instanciation de 2 objets de classe Voiture
    Voiture ma_voiture; // appel constructeur par défaut
    Voiture autre_voiture(3, 15); // appel constructeur
    ma_voiture.set_places(7); // écrit dans l'attribut places de ma_voiture
    ma_voiture.rouler(90); // appelle la méthode rouler de ma_voiture
    std::cout << autre_voiture.get_places(); // lit l'attribut places de autre_voiture
}
```

```
// Planete HERITE de sf::CircleShape
class Planete : public sf::CircleShape
{
public:
    Planete(std::string nom, long diametre, float distance,
            sf::Color couleur);
    ~Planete();

    std::string get_nom();
    long get_diametre();
    float get_distance_au_soleil();

private:
    std::string nom;
    long diametre;
    float distance_au_soleil;
    sf::Color couleur;
};
```

```
// Constructeur, initialisations uniformisées des attributs,
Planete::Planete(std::string nom, long diametre,
                 float distance, sf::Color couleur) :
    nom{nom}, diametre{diametre},
    distance_au_soleil{distance}, couleur{couleur},
    sf::CircleShape((float)diametre/2) //construct classe mère
{
    setFillColor(couleur); // init attribut de sf::CircleShape
}
```

```
int main()
{
    sf::RenderWindow window(sf::VideoMode(max_x, max_y), "Planètes");

    // Instanciation des objets de classe Planete
    Planete mercure("Mercure", 4878, 58, sf::Color::Yellow);
}
```

Une **Planete** est un genre de **CircleShape**, donc on peut dire que **Planete** hérite de **CircleShape**. Les objets de classe **Planete** ont leurs propres propriétés, mais **héritent**, en plus de celles de **CircleShape** qu'ils utilisent directement.

→ **HERITAGE**

```
// Toto est COMPOSE (entre autres) d'objets de la SFML
class Toto
{
public:
    Toto(float x, float y);
    ~Toto();
    void afficheDans(sf::RenderWindow &fenetre);
    void set_position(float x, float y);

private:
    sf::CircleShape tete;           // des objets de type sf:: etc.
    sf::CircleShape oeil1;         // rentrent dans la COMPOSITION
    sf::CircleShape oeil2;         // d'un objet de classe Toto
    sf::RectangleShape bouche;
    float x;
    float y;
};
```

```
int main()
{
    sf::RenderWindow window(sf::VideoMode(max_x, max_y), "Toto!");

    // Instanciation d'un objet de classe Toto
    Toto laTeteAToto1(100, 100);

    // Modification de sa position
    laTeteAToto1.set_position(200, 200);
}
```

```
void Toto::set_position(float x, float y)
{
    tete.setPosition(x, y);
    oeil1.setPosition(x + 10, y + 10);
    oeil2.setPosition(x + 30, y + 10);
    bouche.setPosition(x + 10, y + 40);
}
```

Les objets de classe **Toto** sont **composés** d'objets de classe **sf::CircleShape**, etc. On accède à leurs propriétés à travers leur nom (ex : `oeil1.setPosition()`).

→ **AGGREGATION ou COMPOSITION**

## 14 ATTRIBUT ET METHODES STATIQUES (DE CLASSE)

// Dans voiture.cpp

```
int Voiture::totalSieges = 0; // Déclare variable de classe (statique)
```

```
Voiture::Voiture(int ce_nbePlaces) : nbePlaces{ ce_nbePlaces }
```

```
{ Voiture::totalSieges += nbePlaces; // variable de classe (statique)
```

```
void Voiture::appelFournisseur()
```

```
{  
    cout << Voiture::totalSieges << " sieges";  
  
    if (Voiture::totalSieges > 20)  
        cout << " : Appel sous-traitant de sieges";  
}
```

```
void Voiture::affiche()
```

```
{  
    cout << " ...";  
}
```

### Remarque :

Ne pas confondre avec la variable statique d'une fonction (locale qui garde sa valeur) !

Ici il s'agit d'une variable commune à tous les objets de la classe ! Notez où et comment on la déclare !

Notez la différence d'appel de méthode non statique ou statique.

// Dans voiture.h

```
class Voiture
```

```
{  
    int nbePlaces;
```

```
public:
```

```
    static int totalSieges; // Attribut statique
```

```
    static void appelFournisseur(); // Méthode statique
```

```
    void affiche();
```

```
    Voiture(int);
```

```
};
```

// Dans Source.cpp

```
int main()
```

```
{
```

```
    Voiture voiture1(5);
```

```
    Voiture voiture2(2);
```

```
    Voiture voiture3(7);
```

```
    Voiture voiture4(7);
```

```
    cout << "Total: " << Voiture::totalSieges; // accès variable de classe
```

```
    voiture3.affiche(); // appel méthode non statique (de l'objet)
```

```
    Voiture::appelFournisseur(); // appel méthode statique (de classe)
```

```
    Voiture::totalSieges = 0;
```

```
    return 0;
```

```
}
```

## 15 TABLEAUX

```
#include <iostream>
#include <array>
#include <vector>;
using std::cout;
```

```
// la paramètre est un tableau de char
void affiche(char tab[])
{
    // boucle accède à toutes les cases
    // du tableau tab2
    for (int i = 0; i < 5; i++)
        cout << tab[i] << '\n';
}
```

```
void main()
{
    int tab1[3];    // déclare tableau qui contient 3 int (taille fixe)
    tab1[0] = 12;   // 1ère case: indice 0 !
    tab1[1] = 26;
    tab1[2] = 0;    // 3ème case: indice 2, attention !

    //tab1[3] = 9;  // Hors limite, écris n'importe où !!!

    // déclare et initialise un tableau de 5 char
    // on aussi char tab2[] = "ABa4" (\0 rajouté automatiquement)
    char tab2[] = { 'A', 'B', 'a', '4', '\0' }; // '\0' ou NULL

    affiche(tab2); // passe un tableau en paramètre

    // array: tableau plus sécurisé de la std (taille fixe)
    std::array<short, 4> tab3{ 0, 3, 45, -48 };

    for (int i = 0; i < tab3.size(); i++)
        cout << tab3.at(i) << '\n';

    //vector: tableau dynamique de la std (taille variable)
    std::vector<double> tab4{ 1.1, 3.3, 6.66 }; // init à 3 cases

    tab4.push_back(2.31); // ajouts dynamiques en fin de tableau
    tab4.push_back(-5.11);

    for (double nbe : tab4) // Parcours auto sur plage du tab4
        cout << nbe << '\n';

    tab4.pop_back(); // enlève dernier élément
}
```

## 16 POINTEURS

### Mémoire

```
// Pointeurs et références
```

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    int maVar = 333;           // une variable maVar
```

```
    int& maRef = maVar;        // une référence sur maVar
```

```
    int* monPointeur;          // un pointeur sur int
    monPointeur = &maVar;       // initialisé à l'adresse de maVar
```

```
    int monTab[3] = { 2, 4, 6 }; // un tableau de int de base
```

```
    std::cout << maVar << '\n';    // accès direct
```

```
    std::cout << maRef << '\n';    // accès par référence
```

```
    std::cout << *monPointeur << "\n\n"; // accès par pointeur
```

```
    std::cout << monTab[2] << '\n'; // accès tableau classique
```

```
    std::cout << *(monTab + 2) << '\n'; // accès tableau par pointeur
```

```
    return 0;
```

```
}
```

monPointeur

1efae0  
( &maVar )

1efad4

\*monPointeur

maRef  
maVar

333

1efae0

&

\*

333  
333  
333

6  
6

Ne pas confondre l'utilisation du & pour déclarer une référence et son utilisation pour affecter l'adresse d'une variable à un pointeur !

De même page suivante pour le passage de paramètre par référence par rapport au passage d'une adresse à une fonction qui attend un pointeur.

```

// Passage paramètres par Références ou Pointeurs
#include <iostream>

void parReference (int& x) // récupération paramètre dans une référence
{
    x = 3 * x; // utilisation de la référence
}

void parPointeur (int* y) // récupération paramètre dans un pointeur
{
    *y = 2 * *y; // utilisation du pointeur en déréférençant
}

int main()
{
    int x = 5;
    int y = 6;

    parReference(x); // passage de x

    parPointeur(&y); // passage de l'adresse de y

    std::cout << x << "\n"; // 15
    std::cout << y << "\n"; // 12
}

```



## // Classes: Pointeurs et Passage paramètres par Références ou Pointeurs

```
ptr-> ..... ;  
(*ptr). ..... ;
```

Note : les deux écritures ci-dessus sont équivalentes :

```
#include <iostream>
```

```
class maClasse
```

```
{
```

```
    int x;
```

```
public:
```

```
    maClasse() { x = 0; }
```

```
    void setX(int x) { this->x = x; }
```

```
    int getX() const { return this->x; }
```

```
};
```

```
void parReference (maClasse& obj) // récup paramètre dans une référence
```

```
{
```

```
    obj.setX(3 * obj.getX()); // util de la référence
```

```
}
```

```
void parPointeur (maClasse* obj) // récup paramètre dans un pointeur
```

```
{
```

```
    obj->setX(2 * obj->getX()); // util du pointeur en déréférençant
```

```
}
```

```
int main()
```

```
{
```

```
    maClasse obj1;
```

```
    maClasse obj2;
```

```
    maClasse* ptr = &obj2;
```

```
    obj1.setX(3);
```

```
    obj2.setX(4); // ou ptr->setX(4);
```

```
    parReference(obj1);
```

```
    parPointeur(&obj2); // ou parPointeur(ptr);
```

```
    std::cout << obj1.getX() << "\n"; // 9
```

```
    std::cout << obj2.getX() << "\n"; // 8
```

```
}
```



# 17 ALLOCATION DYNAMIQUE DE MEMOIRE

## 1. L'allocation statique de la mémoire :

- Se produit pour les variables **statiques (static)** et les **variables globales**.
- La mémoire pour ces types de variables est attribuée une fois lorsque votre programme est exécuté et persiste tout au long de la vie de votre programme.
- On appelle cette zone mémoire le **tas (heap)**. Sa taille peut-être très grande, mais accès moins rapide.

## 2. L'allocation automatique de la mémoire :

- Se produit pour les **paramètres de fonction** et les **variables locales**.
- La mémoire pour ces types de variables est attribuée lorsqu'on entre dans le bloc concerné et libérée lorsque qu'on sort du bloc, et ceci autant de fois que nécessaire.
- On appelle cette zone mémoire la **pile (stack)**. Taille limitée, mais accès (LIFO : Last In First Out) très rapide.

## 3. L'allocation dynamique de mémoire:

- On alloue suivant le besoin du moment, par **new** et on libère par **delete**
- La mémoire est allouée et libérée à la demande, mais c'est au programmeur de le gérer !
- La mémoire allouée est dans le **tas (heap)**, mais le pointeur qui permet d'y accéder est dans la **pile (stack)** : si le pointeur est détruit parce qu'on sort de la portée on ne peut plus libérer la mémoire correspondante → fuite mémoire → plantage à terme

### Par pointeur classique :

```
int nbeNotes = 3;  
double* ptrNotes = new double[nbeNotes]; // Alloc dyn Tableau
```

```
ptrNotes[2] = 2.3; // forme tableau  
*(ptrNotes + 2) = 2.3; // forme pointeur
```

```
delete[] ptrNotes; // Libération mémoire
```

```
maClasse * ptr1_maclasse = new maClasse; // Alloc dyn objet
```

```
ptr1_maclasse->setX(6); // Accès membres par pointeur
```

```
delete ptr1_maclasse; // Libération mémoire objet
```

### Par RAII (new et delete encapsulés dans une classe) :

```
class Notes {  
    double* notes; // pointeur sur mémoire, encapsulé  
public:  
    Notes(int nbe) { notes = new double[nbe]; } // Allocation dynamique mémoire par constructeur  
    ~Notes() { delete[] notes; } // Libération de la mémoire allouée, par le destructeur  
  
    double* getNotes() { return this->notes; } // accesseur du pointeur encapsulé  
};
```

Ici à l'instanciation de l'objet mesNotes, on passe un paramètre nbeNotes au constructeur pour donner la bonne taille au tableau

```
Notes mesNotes(nbeNotes); // Objets mesNotes de type Notes pour encapsuler le new/delete  
double* ptrNotes = mesNotes.getNotes(); // récup du pointeur pour même utilisation ensuite
```

### Par pointeurs intelligents :

```
#include <memory>  
using std::unique_ptr;
```

Pas d'étoile à la  
déclaration, c'est un  
objet de type

```
unique_ptr<double[]> ptrNotes = // smart pointeur sur tableau  
    (unique_ptr<double[]>) (new double[nbeNotes]); // de doubles, alloué dynamiquement
```

unique\_ptr qui s'utilise ensuite comme un pointeur et avec plus de possibilités (RAII améliorée).

### Déclaration d'une classe

```
class Point
{
private:
    double x, y;

public:
    Point();
    ~Point();
    void Afficher();
};
```

attributs

constructeur

destructeur

méthode

**Point.h**

### Définition d'une classe

```
Point::Point()
{
    x = 0.;
    y = 0.;
}
```

Initialisation des données  
membres de la classe  
Point

```
Point::~~Point()
{
}
```

```
void Point::Afficher()
{
    std::cout << x << y;
}
```

Définition d'une fonction  
membre de la classe  
Point

**Point.cpp**

### Instanciation d'objets

```
Point point1;
```

Allocation statique d'un objet Point

```
point1.Afficher();
```

L'opérateur d'accès aux membres d'un  
objet est le .

```
Point *point2 = new Point;
```

Allocation dynamique d'un objet Point

```
point2->Afficher();
```

L'opérateur d'accès aux membres d'un  
objet à partir d'une adresse est la ->

```
delete point2;
```

Libération mémoire de l'objet Point

**main.cpp**

## 18 FICHIERS ET STREAMS

### FICHIERS TEXTES

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

class Eleves {          // RAII tableau de strings, de taille réglable
    string* ptr;
public:
    Eleves(int nbe) { ptr = new string[nbe]; } // constructeur: new
    ~Eleves() { delete[] ptr; }                // destructeur: delete
    string* get_ptr() { return this->ptr; }     // get pointeur
};

int main() {
    int nbe{ 0 };    string nom{ "" };
    cout << "Combien d'eleves ? "; cin >> nbe; cin.ignore(32767, '\n');

    Eleves eleve(nbe); // objet qui encapsule le new/delete du tableau d'élèves
    string* ptr = eleve.get_ptr(); // récupération du pointeur sur le tableau

    fstream myfile("./eleves.txt", ios::in | ios::out | ios::trunc); //creation fichier

    if (!myfile.is_open()) { cout << "Erreur ouverture fichier \n"; return -1; }

    for (int i = 0; i < nbe; i++)
    {
        cout << "Saisissez un Nom d'eleve : "; getline(cin, nom);
        ptr[i] = nom + '\n'; // remplissage tableau, case par case
    }
}
```

```

for (int i = 0; i < nbe; i++)
{
    myfile << ptr[i];           //écriture fichier, ligne par ligne
}
myfile.close();                // fermeture fichier

myfile.open("./eleves.txt", ios::in | ios::beg);    //ouverture

if (!myfile.is_open()) { cout << "Erreur ouverture fichier \n"; return -1; }

for (int i = 0; i < nbe; i++)
{
    getline(myfile, nom);       // lecture fichier ligne par ligne
    ptr[i] = nom + '\n';        // remplissage tableau, case par case
}
return 0;
}

```

# 19 EXCEPTIONS

Vérification de l'allocation par test if-then-else

```
#include <iostream>

int main() {

    int* monTab = new (std::nothrow) int[1000000000];

    if (monTab != nullptr)
    {
        // ok, on peut travailler
        monTab[12] = 456;
    }
    else
    {
        // message d'erreur
        return -1;
    }
}
```

Vérification allocation par exception try-catch, et exception standard affichée par what

```
#include <iostream>
#include <exception>

int main() {
    try
    {
        int* monTab = new int[1000000000];
        // ici OK on peut travailler
        monTab[12] = 456;
    }
    catch (std::exception& e)
    {
        // message d'erreur
        std::cout << "Exception : " << e.what() ;
    }
}
```

Lancer une exception par  
throw

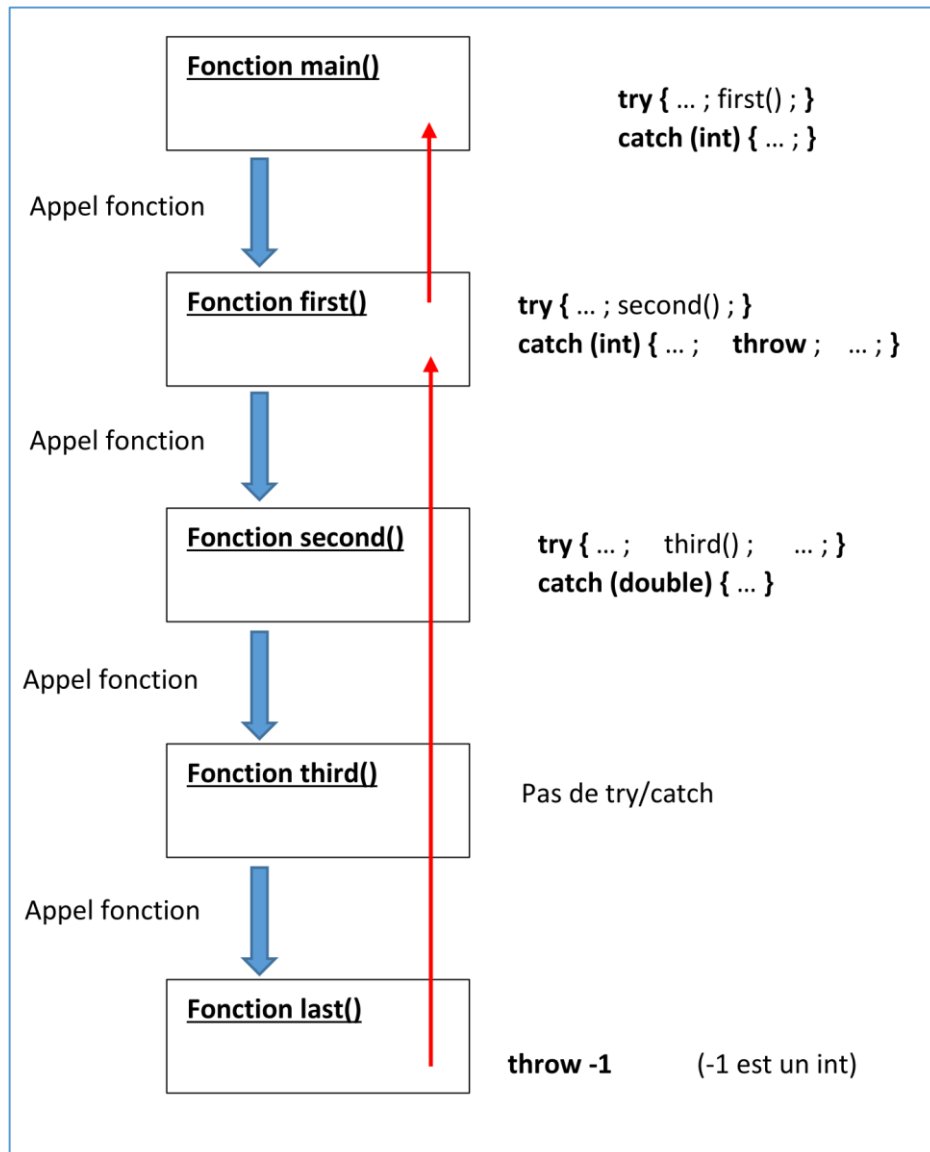
```
// exceptions
#include <iostream>
using namespace std;

int main() {
    try
    {
        throw 20;
        cout << "Vais-je passer ici ?";
    }
    catch (int e)
    {
        cout << "An exception occurred. Exception Nr. " << e << '\n';
    }
    return 0;
}
```

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

int main() {
    try {
        // code here
        throw 12.56;
    }
    catch (int param) { cout << "int exception"; }
    catch (char param) { cout << "char exception"; }
    catch (...) { cout << "default exception"; }
}
```

Remontée des exceptions dans la pile :



Libération de ressources après une exception : RAII (allocation et libération de la ressource sont encapsulées dans une classe. Le destructeur libère la ressource).

```

#include <iostream>
#include <exception>

class GrosTab      // classe RAII
{
public:
    int* ptr;
    GrosTab()
    {
        ptr = new int[1000000000];
        std::cout << "Pointeur alloue: " << ptr << std::endl;
    }
    ~GrosTab()
    {
        if (ptr != nullptr) delete[] ptr;
        std::cout << "Pointeur delete: " << ptr << std::endl;
    }
};
  
```

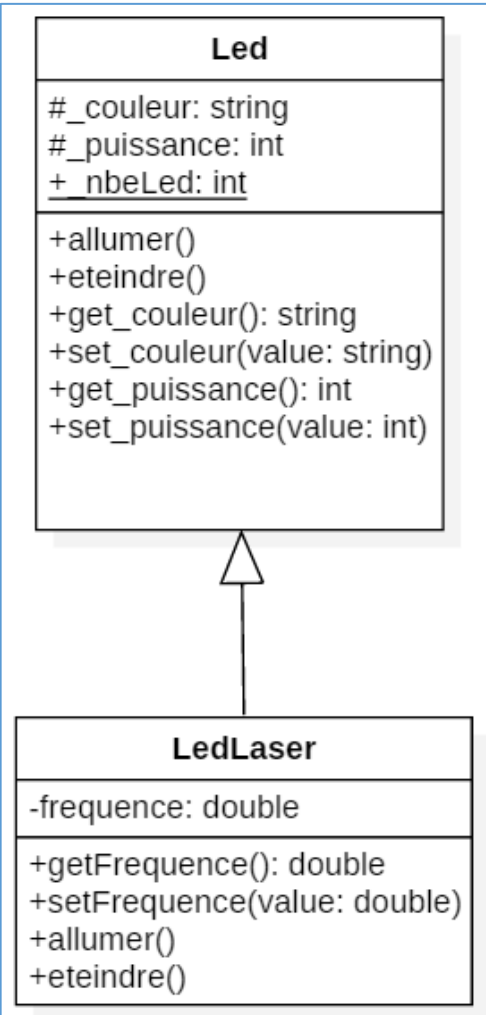
```

int main()
{
    try
    {
        GrosTab monTab[10000]; //Alloc tableau de tableaux
        //...
        // ici on utilise la mémoire allouée, car pas exception
    }
    catch (std::exception& e)
    {
        std::cout << e.what() << std::endl;
        // ici il faut gérer l'erreur, ou sortir du programme
    }
}
  
```

## 20 UML DIAGRAMME DE CLASSES

**PASSAGE à C++ : voir page suivante et cours « Passage d'UML à C++ »**

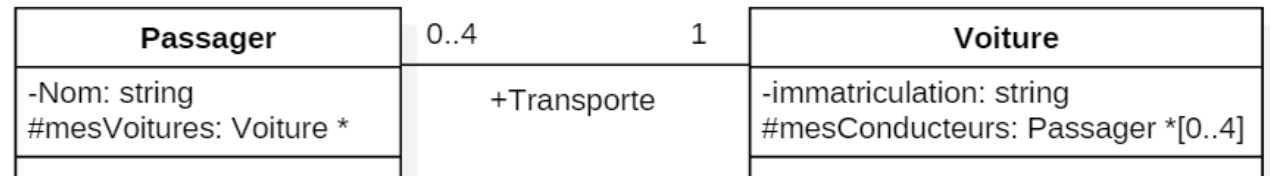
**Héritage** : une LedLaser est un genre de Led (et hérite des attributs de la Led)



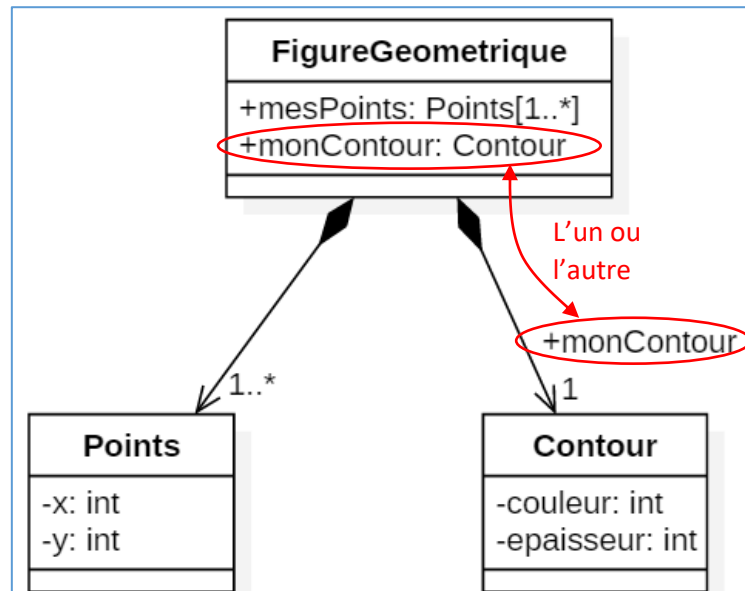
Mot-clé **public** (ou caractère +) : visible de partout.  
Mot-clé **protected** (ou caractère #) : visible dans la classe et tous ses descendants.  
Mot-clé **private** (ou caractère -) : visible uniquement dans la classe.

Attribut ou méthode soulignée : **static** à la classe.

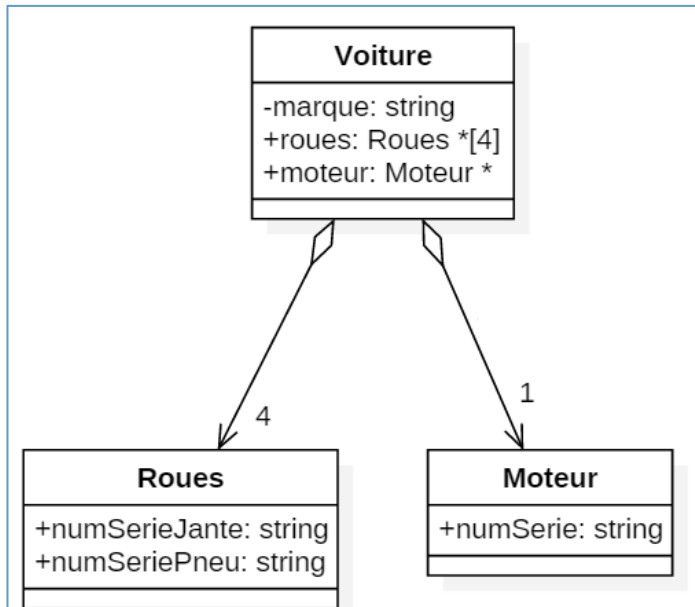
**Associations et cardinalités** : il y a un lien, mais pas de type héritage, ni de composition ni d'agrégation.  
Une Voiture transporte de **0 à 4** Passagers. Un Passager est transporté par **1** Voiture.



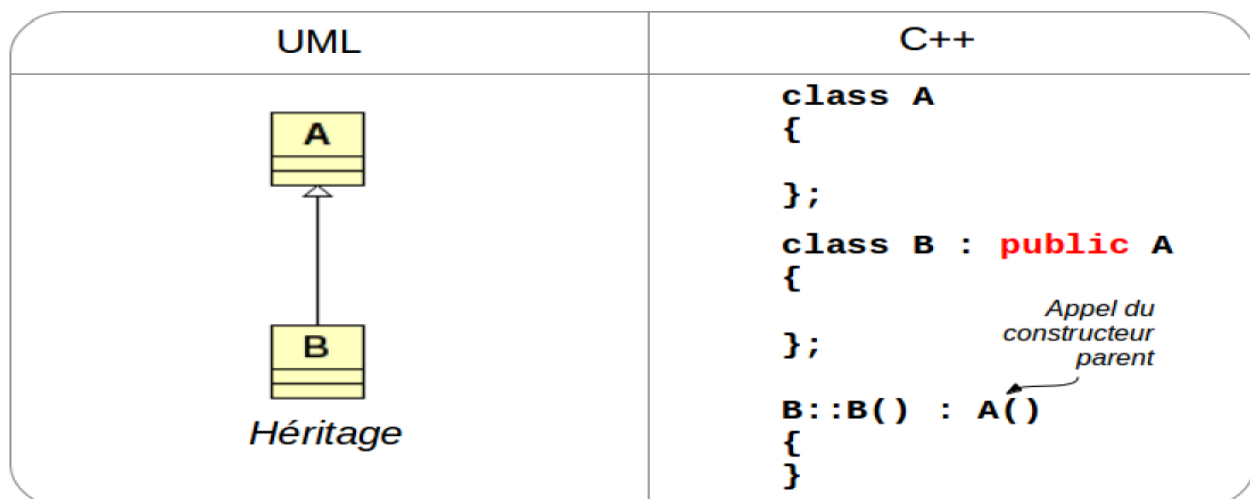
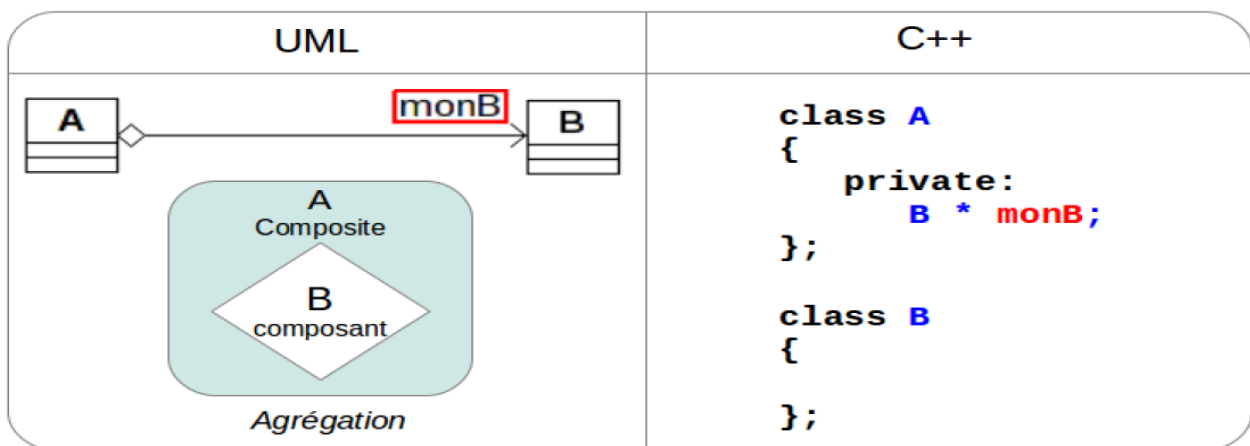
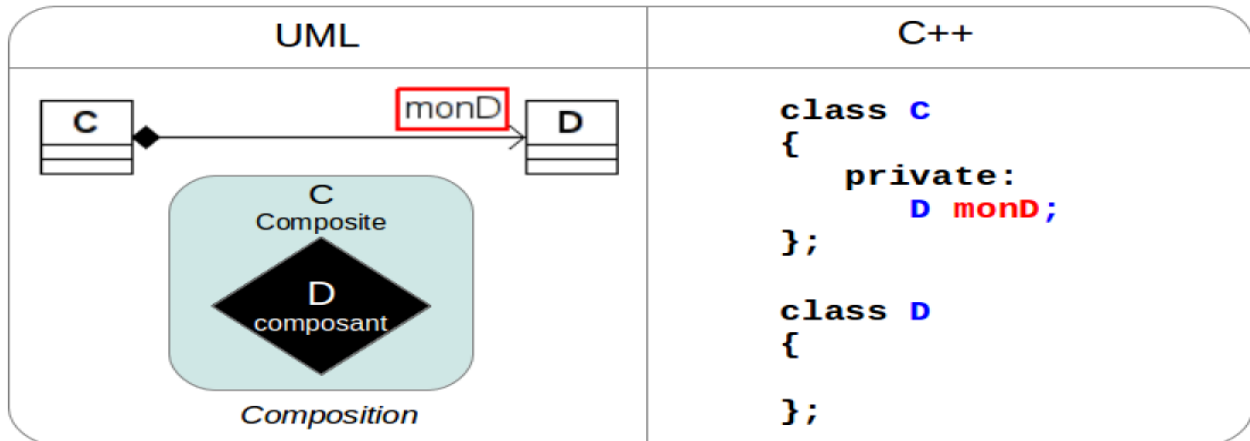
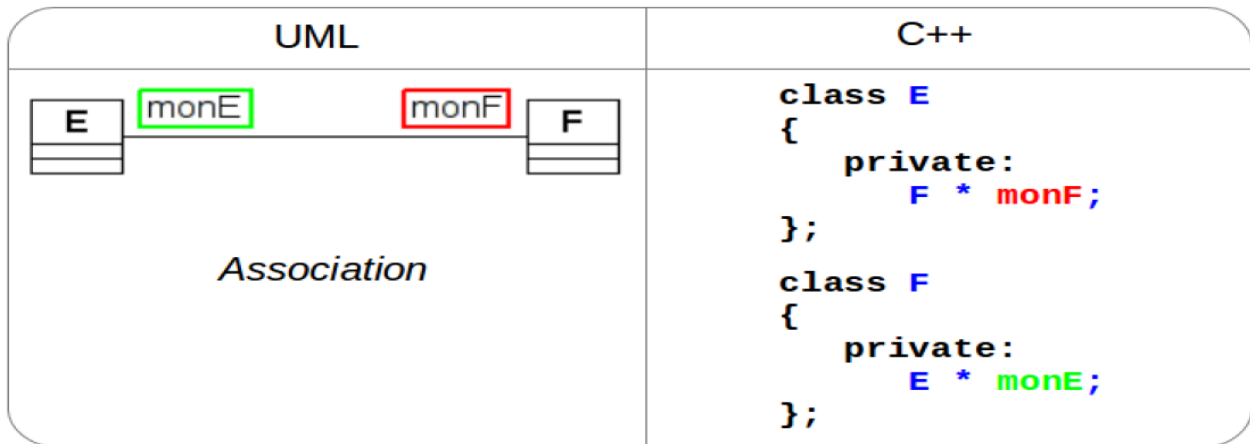
**Composition** : si on détruit le contenant les contenus sont détruits aussi. Une FigureGeometrique est composée de **1 à n** Points et **1** Contour. Relation seulement dans ce sens.



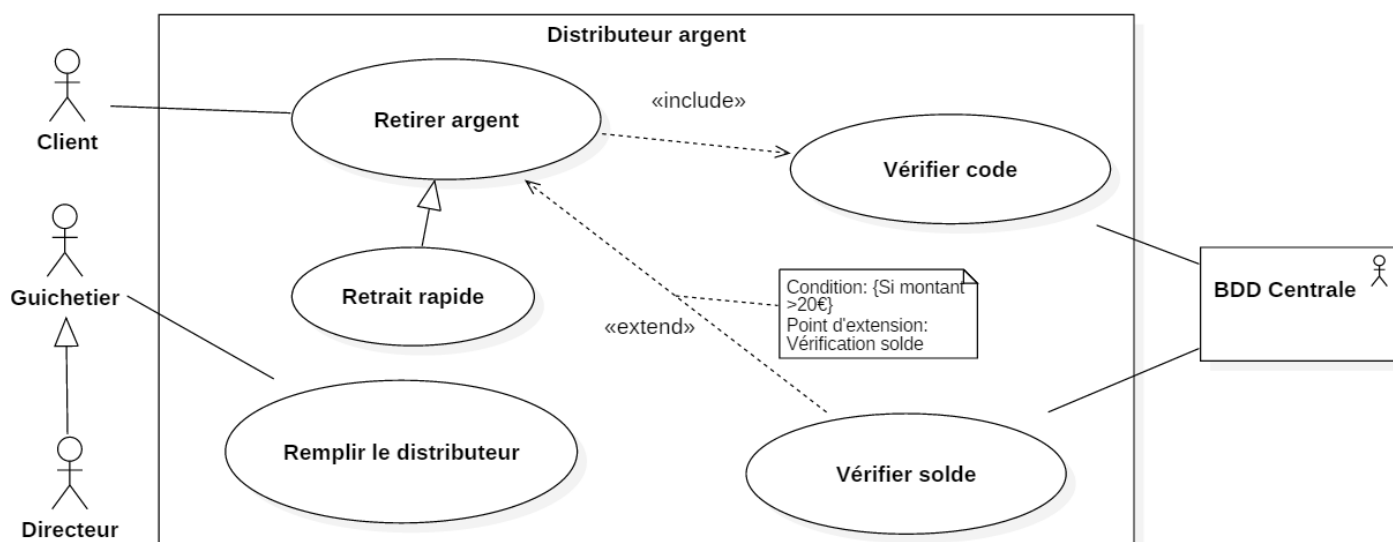
**Agrégation** : quand on détruit le contenant les contenus continuent à exister. Une Voiture contient **4** roues. Et **1** moteur. Relation seulement dans ce sens.







## 21 UML DIAGRAMME DE CAS D'UTILISATION



**Client**, **Guichetier**, **Directeur** et **BDD Centrale** sont des **acteurs**, ils sont extérieurs au système, mais ils interagissent avec lui. **BDD Centrale** est un acteur même si ce n'est pas une personne.

**Directeur** est une **spécialisation** de **Guichetier**, c'est un guichetier particulier. **Directeur** **hérite** de **Guichetier**. **Guichetier** est une **généralisation**. Un Directeur peut faire ce que fait un Guichetier, mais pas l'inverse.

**Retirer argent** est un **cas d'utilisation**. **Retirer argent** **inclus obligatoirement** **Vérifier Code**. **Vérifier solde** vient **éventuellement étendre** (compléter) **Retirer argent**. **Retrait rapide** **hérite** de **Retirer argent**.

## 22 UML DIAGRAMME DE SEQUENCE

Voir page suivante et les correspondances entre le diagramme de séquence et le programme C++.

```
Poller::Poller(Mcp_Can& mppt, Echantillon& mesures, InterfaceBDD& accesBase)
{
    bool retReceive = false;
    bool retTransmit = false;
    unsigned char *pData = nullptr;

    int cumulEnergie = accesBase.obtenirDerniereEnergie();           // 1 et 2
    accesBase.setEnergie(cumulEnergie);                             // 3 et 4

    while (true)                                                    // loop infinie
    {
        while (retReceive == false)                                  // loop retReive==false
        {
            retTransmit = mppt.transmit(0x711);                     // 5 et 6

            if (retTransmit == true)                                 // alt retTransmit==false
            {
                retReceive = mppt.receive();                        // 7 et 8
            }
        }
        pData = mppt.getData();                                     // 9 et 10
        mesures.extraireMesures(pData);                             // 11

        accesBase.enregistrerEchantillon(mesures);                 // 13
        attendre30Minutes();                                        // 14
    }
}
```

```
void main()
{
    Mcp_Can mppt;
    Echantillon mesures;
    InterfaceBDD accesBase;

    Poller acquisitionMesures(mppt, mesures, accesBase);
}
```

