

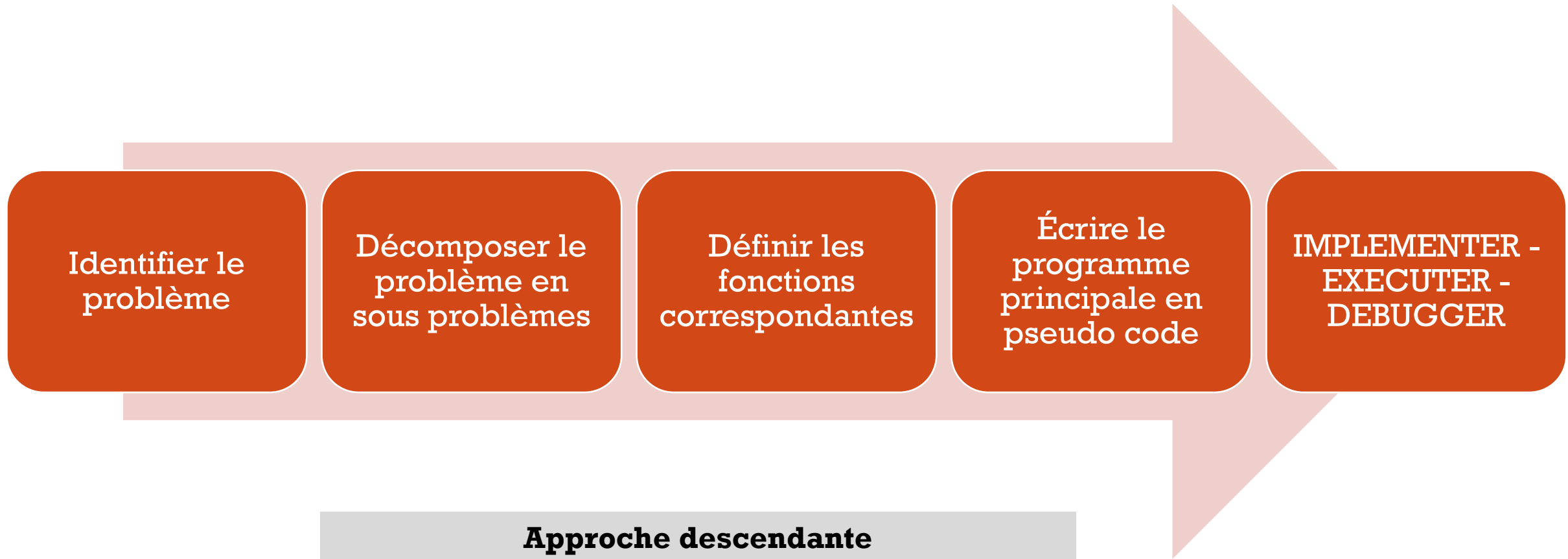
# ATELIER DE PROGRAMMATION C++

Assili Mohamed – CPI2



Groupe de discussion sur Discord: <https://discord.gg/gncxVNBQ4y>

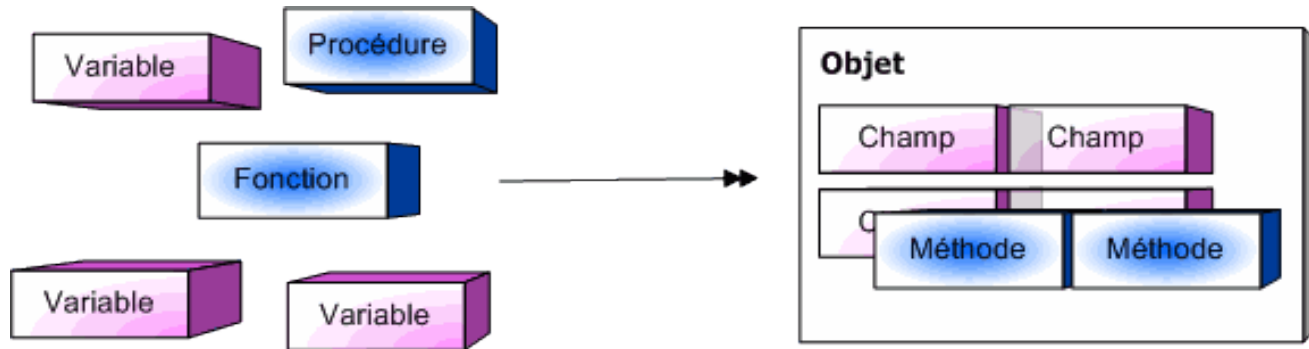
# LA PROGRAMMATION PROCÉDURALE



# LIMITES

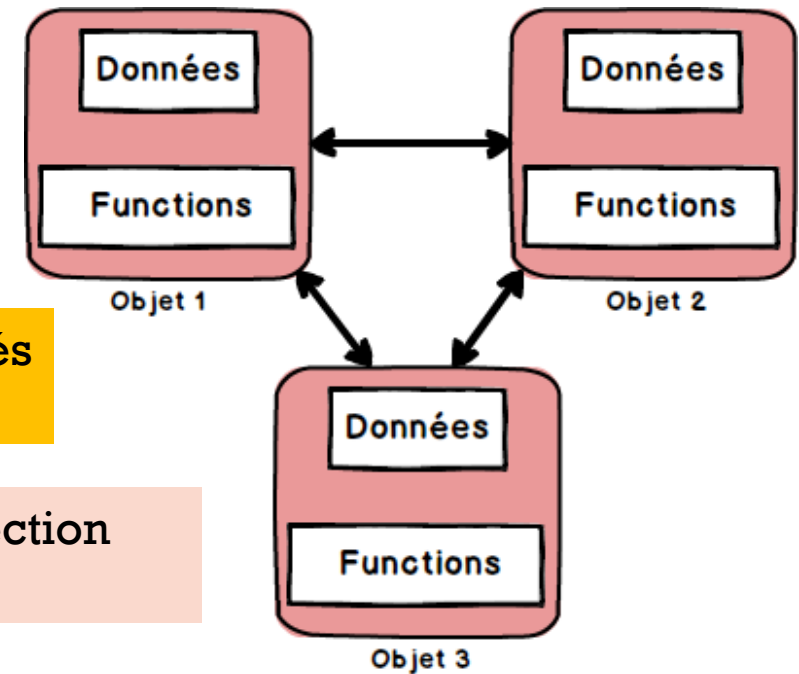
- Difficulté de réutilisation du code ( modifications des paramètres selon le cas)
- Difficulté de maintenir les grandes applications
- Risque du phénomène « spaghetti code »
- Critères de qualité du code facilement violés: lisibilité, modularité, ...

# LA PROGRAMMATION ORIENTÉE OBJET



En OO, le logiciel est considéré comme une collection d'objets dissociés définis par des propriétés.

Un objet comprend à la fois une structure de données et une collection d'opérations (son comportement).



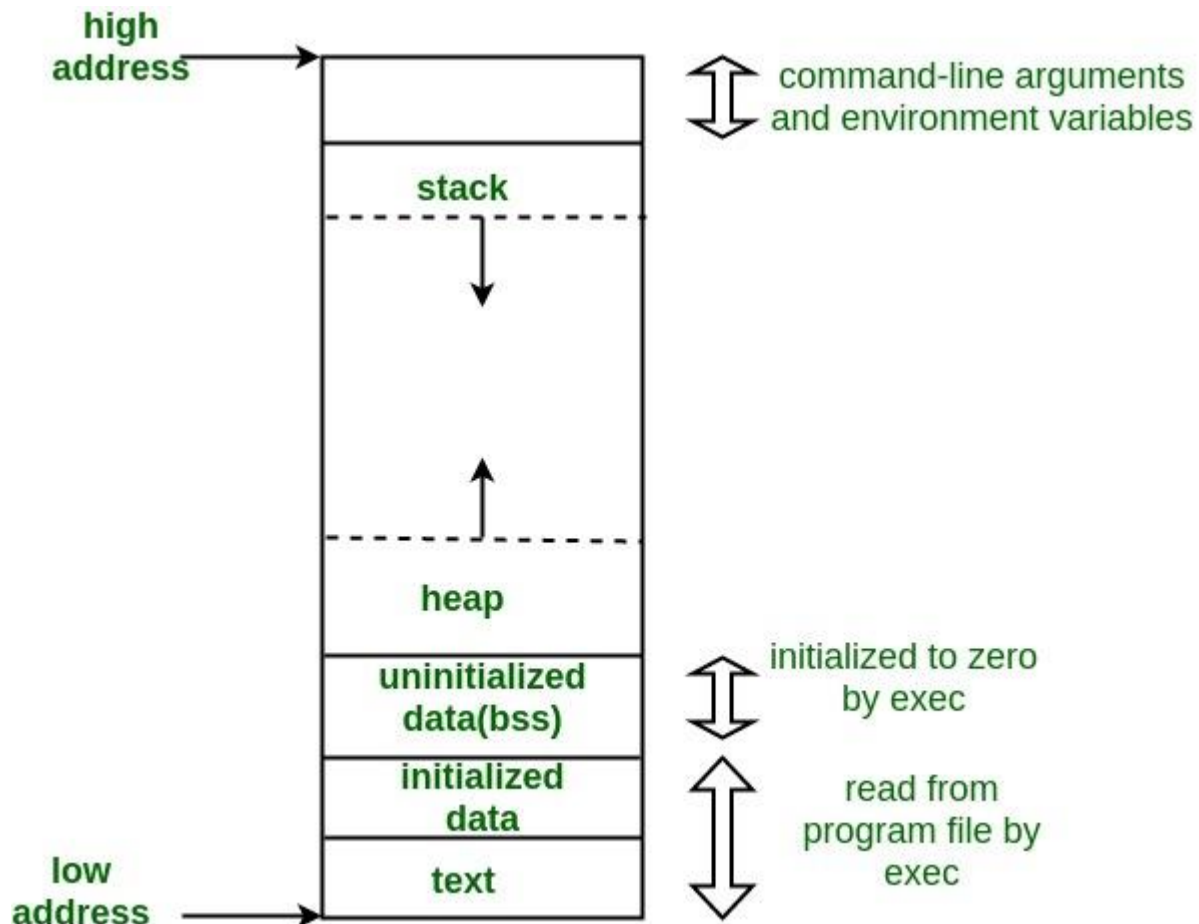
**Approche ascendante**

les méthodes orientées objets sont ascendantes.

# POINTS FORTS DE L'OO

- **Modularité** : les objets forment des modules compacts regroupant des données et un ensemble d'opérations.
- **Abstraction** : Les entités objets de la POO sont proches de celles du monde réel. Les concepts utilisés sont donc proches des abstractions familières que nous exploitons.
- **Productivité et réutilisabilité** : Plus l'application est complexe et plus l'approche POO est intéressante en terme de productivité.
- **Sûreté** : L'encapsulation et le typage des classes offrent une certaine robustesse aux applications.

# FORME DE LA MÉMOIRE D'UN PROG C/C++



- Segment de code (text segment) : les instructions du programme
- Segment des données initialisées : les variables globales et statiques initialisées par le programmeur
- Segment des données non-initialisées : contient les données globales et statiques qui sont par défaut (initialisé par le kernel) 0 ou non initialisée explicitement.
- Heap : allocation dynamique des objets (le programmeur alloue et désalloue la mémoire)
- Stack : allocation temporaire des objets (existent seulement durant l'exécution d'une fonction)

# STACK VS HEAP

## Stack

- Allocation contiguë
- Accès plus rapide
- LIFO
- Allocation / Désallocation automatique
- Taille limitée

## Heap

- Allocation non contiguë (en général)
- Accès moins rapide
- Allocation / Désallocation manuelle
- Taille à la demande

Exemple:

```
int* creerTab()
{
    int tab[20];
    return tab;
}
```



```
int* creerTab()
{
    int* tab = new int[20];
    return tab;
}
```

# LE LANGAGE C++

**Multiparadigme** : Il se base sur le langage C. Version améliorée du langage C. Permet la programmation procédurale et aussi la programmation orientée objet.

**Rapidité**: langage de bas niveau par rapport aux autres langages connus (java, python, ...) ce qui lui permet un control total sur les ressources matérielles. Il est particulièrement efficace pour développer des jeux 3D et pour développer des applications temps réel.

**Riche** : le langage C++ est très utilisé dans les communautés des développeurs donc il dispose d'une panoplie de bibliothèque et d'une documentation abondante.

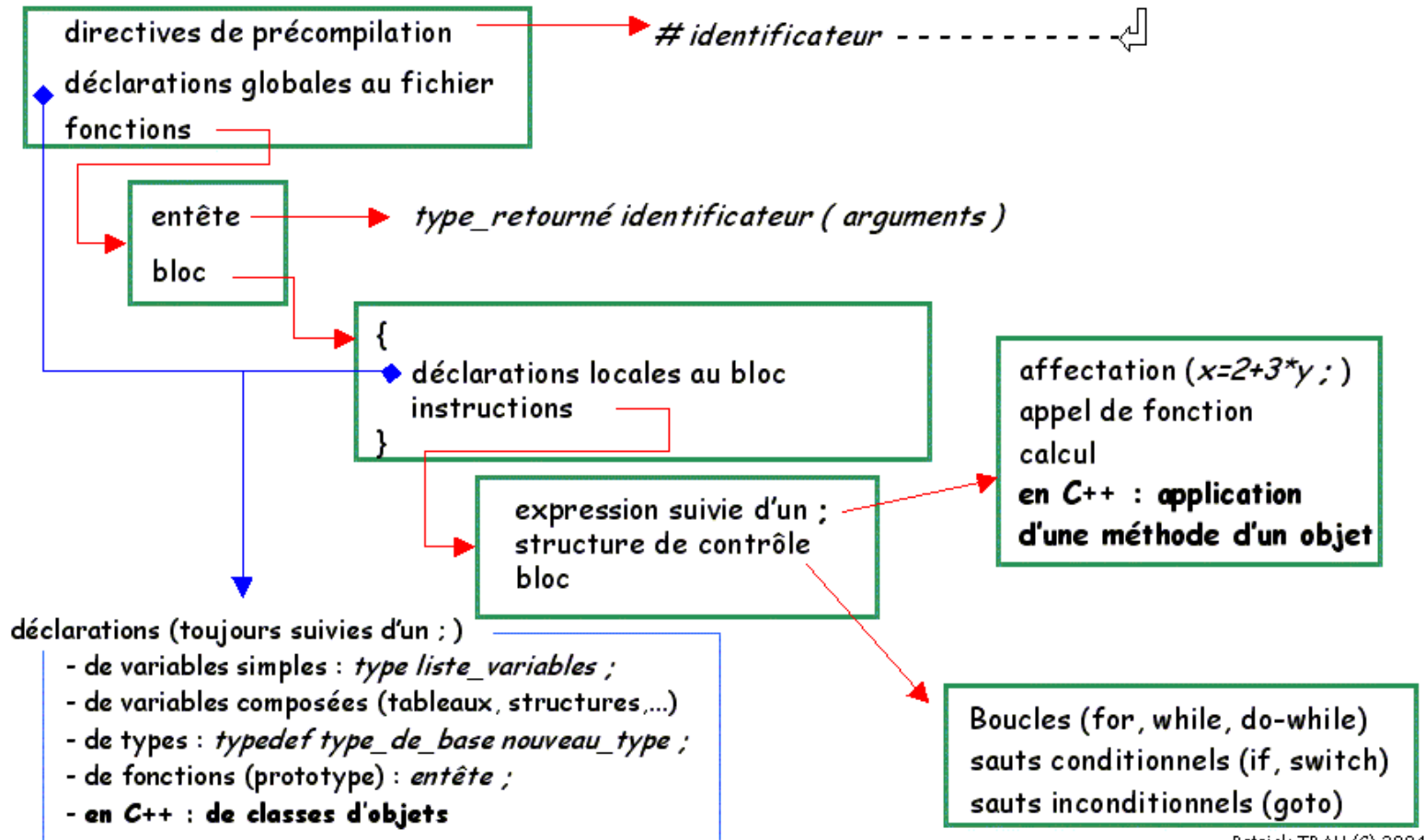
**Fiable** : sa large utilisation lui confère une maturité ce qui le rend fiable.



# LE LANGAGE C++

Le C++ est un langage typé (typed language); c'est à dire le type de tout objet utilisé doit être connu par le compilateur au moment de son utilisation dans le programme (le type de l'objet détermine l'ensemble des opérations qui lui sont applicable).

# STRUCTURE D'UN PROGRAMME EN C++



Patrick TRAU (C) 2004

```
fcts.h X
#ifndef FCTS_H_INCLUDED
#define FCTS_H_INCLUDED

int somme(int a,int b);

#endif // FCTS_H_INCLUDED
```

```
fcts.cpp X
#include<iostream>
int somme(int a,int b){
    return a+b;
}
```

```
main.cpp X
1  #include <iostream>
2  #include "fcts.h"
3  using std::cout;
4  using std::cin;
5  using std::endl;
6
7  int main()
8  {
9      int a,b;
10     cout << "Entrer un entier" << endl;
11     cin >> a;
12     cout << "Entrer un entier" << endl;
13     cin >> b;
14
15     cout << "La somme est " << somme(a,b) << endl;
16     return 0;
17 }
```

# CONCEPTS DE BASE (1)

```
main.cpp x
1  #include <iostream>
2  #include "fcts.h"
3  using std::cout;
4  using std::cin;
5  using std::endl;
6
7  int main()
8  {
9      int a,b;
10     cout << "Entrer un entier" << endl;
11     cin >> a;
12     cout << "Entrer un entier" << endl;
13     cin >> b;
14
15     cout << "La somme est " << somme(a,b) << endl;
16     return 0;
17 }
```

Directive préprocesseur. Utiliser une bibliothèque **standard** iostream

Utilisation d'une bibliothèque personnelle

Utilisation des objets prédéfinis dans le nom de domaine **std**

Fonction principale du programme

**cout** : la sortie standard du programme (console)  
**cin** : l'entrée standard des données saisies au clavier  
**endl** : fin de la ligne  
<< : opérateur de sortie du flux de données  
>> : opérateur d'entrée du flux de données

# INITIALISATION

- 3 manières d'initialiser une variable :

---

<code>( <i>expression-list</i> )</code>	(1)
---	-----

---

<code>= <i>expression</i></code>	(2)
----------------------------------	-----

---

<code>{ <i>initializer-list</i> }</code>	(3)
--	-----

---

```
int b = 1;  
int c(2);  
int d{}; // d vaut 0
```

```
std::cout << "b= " << b << std::endl;  
std::cout << "c= " << c << std::endl;  
std::cout << "d= " << d << std::endl;
```

Console d

```
b= 1  
c= 2  
d= 0
```

Plus de détails via ce lien: <https://en.cppreference.com/w/cpp/language/initialization>

# CONCEPTS DE BASE (2)

## ALLOCATION STATIQUE – ALLOCATION DYNAMIQUE

### Allocation Statique

```
int x =0 ;  
int y(5);
```

Les variables sont allouées de manière permanente

L'allocation est faite avant l'exécution du programme

Pas de possibilité de réutilisation de la mémoire

### Allocation Dynamique

```
int *pt = new int(10);
```

les variables ne sont allouées que si votre unité de programme est active

L'allocation est faite pendant l'exécution du programme

La mémoire est réutilisable et peut être libérée lorsqu'elle n'est pas nécessaire (opérateur **delete**).

**En c/c++, la gestion de la mémoire est à la charge du programmeur. Il doit libérer de la mémoire à chaque fois qu'elle n'est plus pointée.**

# CONCEPTS DE BASE (3) – LES BOOLÉENS

```
#include <iostream>

using namespace std;

int main()
{
    bool v_bool1 = true;
    bool v_bool2(false);

    cout << v_bool1 << endl;
    cout << v_bool2 << endl;
    cout << boolalpha << v_bool1 << endl;
    cout << boolalpha << v_bool2 << endl;
    return 0;
}
```

Déclaration et initialisation

Une variable de type **bool** ne peut avoir que deux états.  
**true** (1) ou **false** (0)

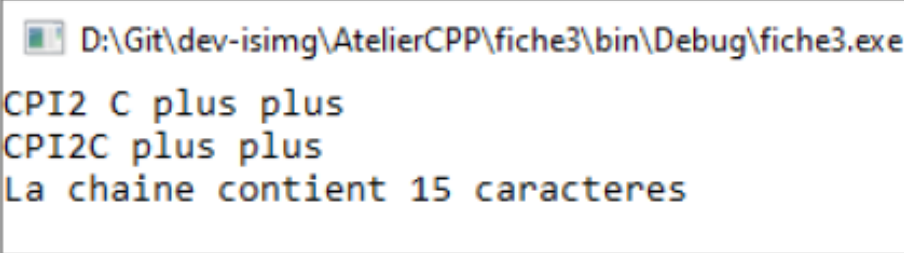
```
D:\Git\dev-ising\AtelierCPP\fiche3\bin\Debug\fiche3.exe
1
0
true
false
```

# CONCEPTS DE BASE (4) — LES STRINGS

```
#include <iostream>

using namespace std;

int main()
{
    string lang("C plus plus");
    string niv = "CPI2";
    cout << niv << " " << lang << endl;
    // ajouter à la fin
    cout << niv.append(lang) << endl;
    // on peut utiliser aussi size() pour avoir la longueur
    cout << "La chaine contient " << niv.length() << " caracteres" << endl;
    return 0;
}
```



## Opérateurs et fonctions :

**+** : concaténation

**length() / size()** : longueur

**== / !=** : comparaison

**compare(str)** : comparer

**at(index)** : lire un caractère

**insert(index,str)** : insérer

Pus de fonctions via ce lien:

<https://devdocs.io/cpp/>

**Remarque:** on peut parcourir un objet de type string avec un indice comme un tableau de caractères.



# CONCEPTS DE BASE (5) — LES STRINGS

```
#include <iostream>

using namespace std;

int main()
{
    string lang;

    cout << "C'est quoi votre langage prefere?" << endl;
    cin >> lang;
    cout << "Vous aimez programmer avec " << lang << endl;
    return 0;
}
```

D:\Git\dev-ising\AtelierCPP\fiche3\bin\Debug\fiche3.exe  
C'est quoi votre langage prefere?  
C plus plus  
Vous aimez programmer avec C

**cin >>** : arrête la lecture au premier caractère espace

**getline(cin, str):** arrête la lecture au premier retour à la ligne

```
#include <iostream>

using namespace std;

int main()
{
    string lang;

    cout << "C'est quoi votre langage prefere?" << endl;
    getline(cin, lang);
    cout << "Vous aimez programmer avec " << lang << endl;
    return 0;
}
```

D:\Git\dev-ising\AtelierCPP\fiche3\bin\Debug\fiche3.exe  
C'est quoi votre langage prefere?  
C plus plus  
Vous aimez programmer avec C plus plus



# CONSTANTES

Deux types de constantes; compile-time constant et runtime constant

```
#include <iostream>

int main()
{
    const int x { 3 }; // x is a compile-time const
    const int y { 4 }; // y is a compile-time const

    const int z { x + y }; // x + y is a compile-time expression

    std::cout << z << '\n';

    return 0;
}
```

```
int main()
{
    const int x{ 3 }; // x is a compile time constant

    const int y{ getNumber() }; // y is a runtime constant

    const int z{ x + y }; // x + y is a runtime expression
    std::cout << z << '\n'; // this is also a runtime expression

    return 0;
}
```

# CONSTANTES - CONSTEXPR

```
int five()
{
    return 5;
}

int main()
{
    constexpr double gravity { 9.8 }; // ok: 9.8 is a constant expression
    constexpr int sum { 4 + 5 };      // ok: 4 + 5 is a constant expression
    constexpr int something { sum };  // ok: sum is a constant expression

    std::cout << "Enter your age: ";
    int age{};
    std::cin >> age;

    constexpr int myAge { age };      // compile error: age is not a constant expression
    constexpr int f { five() };       // compile error: return value of five() is not a constant
expression

    return 0;
}
```

## Best practice

Any variable that should not be modifiable after initialization and whose initializer is known at compile-time should be declared as `constexpr`.

Any variable that should not be modifiable after initialization and whose initializer is not known at compile-time should be declared as `const`.

# LES TABLEAUX – RAW ARRAYS

```
constexpr int SIZE = 5;
int t[SIZE];           // mémoire contiguë

// initialiser t
for (int i = 0; i < SIZE; i++)
    t[i] = i;

// afficher t
for (int i = 0; i < SIZE; i++)
    std::cout << t[i] << "  ";
```

## Stack Memory Allocation

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

// initialiser tab
for (int i = 0; i < SIZE; i++)
    tab[i] = i;

// afficher tab
for (int i = 0; i < SIZE; i++)
    std::cout << tab[i] << "  ";

delete[] tab;
```

## Heap Memory Allocation

# LES TABLEAUX – STATIC ARRAYS

```
#include <iostream>
#include <array>
#include <algorithm>

int main()
{
    std::array<int, 5> tab{33,5,8,-1, 88};

    // trier tab avec la fonction prédéfinie sort
    std::sort(tab.begin(),tab.end());

    // afficher tab
    for (int i = 0; i < tab.size(); i++)
        std::cout << tab[i] << " ";

    return 0;
}
```

Autre façon de parcourir

```
for (const auto& ele : tab)
    std::cout << ele << " ";
```

**Stack Memory Allocation**

# LES CONTENEURS - VECTOR

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

int main()
{
    std::array<int, 10> tab{ { -33,5,8,-1, 88, -3, 9, 22, 0,-12} };

    // afficher tab
    for (int i = 0; i < tab.size(); i++)
        std::cout << tab[i] << " ";

    // remplir vec avec les int positifs
    std::vector<int> vec;
    for (const int& ele : tab)
        if (ele >= 0)
            vec.push_back(ele);

    // afficher un vector
    std::cout << std::endl;
    for (const int& e : vec)
        std::cout << e << " ";

    return 0;
}
```

- Une taille dynamique (non connue à l'avance) .
- Augmente sa taille par duplication.
- Dispose de plusieurs méthodes
- Allocation contiguë
- On peut connaître la quantité de mémoire allouée et faire retourner la mémoire en excès vers le système (capacity(), shrink\_to\_fit() ).

Références :

<https://en.cppreference.com/w/cpp/container/vector>

# LES CONTENEURS – VECTOR

```
#include <iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> vl;
    // remplissage de vl
    for (int i = 1; i <= 5; i++)
        vl.push_back(i);
    // affichage avec un itérateur
    cout << "affichage dans l'ordre" << endl;
    for (auto it = vl.begin(); it != vl.end(); ++it)
        cout << *it << " ";
    cout << endl;
    cout << "affichage dans l'ordre inverse" << endl;
    for (auto it = vl.rbegin(); it != vl.rend(); ++it)
        cout << *it << " ";
    return 0;
}
```

D:\Git\dev-ising\AtelierCPP\fiche3\bin\De

affichage dans l'ordre

1 2 3 4 5

affichage dans l'ordre inverse

5 4 3 2 1

Inclure la bibliothèque vector.

Voir :

<https://cplusplus.com/reference/vector/vector/>

Création statique d'un tableau d'entiers

Insérer des éléments dans le tableau

Début du tableau partant de la gauche

Fin du tableau partant de la gauche

Début et fin du tableau partant de la droite.

# LES CONTENEURS – MAP / UNORDRED\_MAP

```
#include <iostream>
#include <array>
#include <vector>
#include <map>

struct ville {
    std::string name;
    uint64_t population;
};

int main()
{
    std::map<std::string, ville> villes;
    villes["Gabes"] = ville{ "Gabes", 150000 };
    villes["Tunis"] = ville{ "Tunis", 900000 };
    villes["Sfax"] = ville{ "Sfax", 900000 };
    villes["Mednine"] = ville{ "Mednine", 120000 };
    villes["Gafsa"] = ville{ "Gafsa", 110000 };

    // disponible depuis c++17
    for (auto& [name, vil] : villes)
        std::cout << name << "\n Population=" << vil.population << std::endl;
    return 0;
}
```

Console de debogage Microsoft Visual Studio

```
Gabes
Population=150000
Gafsa
Population=110000
Mednine
Population=120000
Sfax
Population=900000
Tunis
Population=900000
```

## Conteneur Ordonné

clé	valeur
Gabes	Gabes, 150000
Gafsa	Gafsa, 110000
Mednine	Mednine, 120000
Sfax	Sfax, 900000
Tunis	Tunis, 900000

→ Structured bindings depuis c++17  
(Espace mémoire structuré)  
pour manipuler des pairs ou des tuples

# LES CONTENEURS - EXERCICE

Etant donné un tableau d'entiers « **tab** » contenant les N (saisie au clavier) premiers entiers partant de 0. on veut l'éclater en deux autres tableaux de manière à avoir un tableau d'entiers premiers et un tableau pour les entiers non premiers.

- Ecrire la fonction « **estPremier** » qui retourne un booléen indiquant si l'entier est premier ou non.
- Ecrire une fonction « **remplir** » qui lit la valeur maximale N et remplit le tableau passé en paramètre.
- Ecrire une fonction « **afficher** » qui affiche les éléments d'un tableau.
- Ecrire la fonction principale main qui crée et remplit le tableau « **tab** » et l'éclater en deux autres tableaux telle que demander.



# LES CONTENEURS - EXERCICE

```
#include <iostream>
#include <cmath>
#include <vector>
bool estPremier(int& n) {
    bool prem = true;
    for (int i = 2; i <= sqrt(n); i++)
        if (n % i == 0)
            prem = false;
    return prem;
}
```

```
void afficher(std::vector<int>& vec) {
    for (int& el : vec)
        std::cout << "[" << el << "]";
    std::cout << std::endl;
}
```

```
void remplir(std::vector<int>& v) {
    int n;
    std::cout << "Donner la limite des entiers:\n";
    std::cin >> n;
    for (int i = 0; i < n; i++)
        v.push_back(i);
}
```

# LES CONTENEURS - EXERCICE

```
int main()
{
    std::vector<int> tab{};
    remplir(tab);
    std::vector<int> vecPremier{}, vecNonPremier{};
    for (int& ele : tab)
        if (estPremier(ele) == true)
            vecPremier.push_back(ele);
        else
            vecNonPremier.push_back(ele);
    std::cout << "Les nombres premiers:\n";
    afficher(vecPremier);
    std::cout << "Les nombres non premiers:\n";
    afficher(vecNonPremier);
    return 0;
}
```

# UNE VARIABLE PLUSIEURS TYPES

Depuis c++17, il est possible d'avoir une seule variable capable d'avoir plusieurs types.

**std::variant<>**

```
#include <iostream>
#include <variant>

int main()
{
    std::variant<std::string, int, float> data;

    data = std::string("hello");
    std::cout << std::get<std::string>(data) << std::endl;

    data = 120;
    std::cout << std::get<int>(data) << std::endl;

    data = 1.03f;
    std::cout << std::get<float>(data) << std::endl;
    return 0;
}
```

Accès aux membres du variant

# UNE VARIABLE PLUSIEURS TYPES - EXERCICE

Soit **f** une fonction définie par  $\sqrt{(x-1) * (2-x)}$

La fonction retourne une valeur réelle lorsqu'elle est définie, sinon elle retourne un message d'erreur « la fonction n'est pas définie !! ».

Ecrire une fonction principale qui appelle la fonction **f** avec une valeur saisie au clavier.

```
#include <iostream>
#include <variant>
std::variant<double, std::string> f(double& x) {
    double r = (x - 1) * (2 - x);

    if (r > 0)
        return sqrt(r);
    else
        return "Erreur sur le signe de l'expression !!";
}
```

# UNE VARIABLE PLUSIEURS TYPES - EXERCICE

```
int main()
{
    double val;
    std::cout << "Donner une valeur:";
    std::cin >> val;
    auto resultat = f(val);
    if (resultat.index() == 0)
        std::cout << "f(" << val << ")=" << std::get<0>(resultat);
    else
        std::cout << std::get<1>(resultat);
    return 0;
}
```

Vérifier si le premier objet est actif

Accéder (lire) le contenu du variant

# STD::TUPLE / STD::PAIR

- Un `std::tuple` est un objet qui peut contenir plusieurs objets. Les objets peuvent être de différents types.
- Les objets contenus dans un tuple sont construits dans leur ordre d'accès.
- `#include <tuple>`

- Un `std::pair` est utilisé pour combiner deux objets qui peuvent être de types différents.
- `#include <utility>`
- On utilise `first` et `second` pour accéder aux objets contenus dans le pair

# STRUCTURED BINDINGS / TUPLE / PAIR

```
#include <iostream>
#include <tuple>

std::tuple<std::string, std::string, int> creerMoi()
{
    return { "Mohamed", "assili", 2022 };
}

int main()
{
    auto [prenom, nom, annee] = creerMoi();

    std::cout << prenom << " " << nom << " "
              << annee << std::endl;

    return 0;
}
```

Espace mémoire structuré (en général utilisé une seule fois, pour éviter de créer un objet Personne par exemple)

# STRUCTURED BINDINGS / TUPLE / PAIR

## Exercice

Ecrire une fonction *décomposer* qui prend en paramètre une chaîne de caractère et retourne le jour, le mois et l'année séparément.

Ecrire une fonction principale *main* qui saisie la date sous forme mm/jj/aaaa et appelle la fonction *décomposer*, puis affiche le résultat.

On peut utiliser **std::stoi** pour convertir un **string** en un **int**



# STRUCTURED BINDINGS / TUPLE / PAIR

Solution

```
#include <iostream>
#include <string>
#include <tuple>

std::tuple<int, int, int> decomposer(std::string& date) {
    int day = std::stoi(date.substr(0, 2));
    int month = std::stoi(date.substr(3, 5));
    int year = std::stoi(date.substr(6, 9));

    return { day, month, year };
}
```

```
int main()
{
    std::string maDate;
    std::cout << "Donner une date sous forme aa/jj/aaaa";
    std::cin >> maDate;
    auto [jour, mois, annee] = decomposer(maDate);
    std::cout << "Jour:" << jour << std::endl;
    std::cout << "Mois:" << mois << std::endl;
    std::cout << "Annee:" << annee << std::endl;
}
```

# LVALUE ET RVALUE

*En général, on les utilise selon l'expression suivante:*

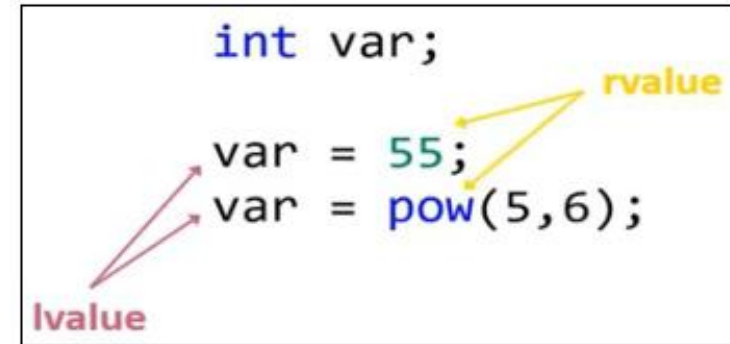
`lvalue = rvalue`

**Lvalue** est un objet qui a une location identifiée en mémoire (a une adresse en mémoire).

- Capable de stocker une information
- Ne peut pas être une fonction, une constante ou une expression

**Rvalue** est un objet n'ayant pas un identifiant en mémoire (désigne la valeur dans un espace mémoire).

- Toute chose pouvant retourner une expression constante ou une valeur.



# LVALUE ET RVALUE - EXEMPLES

```
int main()
{
    int i, j, *p;

    // Correct usage: the variable i is an lvalue and the literal 7 is a prvalue
    i = 7;

    // Incorrect usage: The left operand must be an lvalue (C2106). `j * 4` is a
    7 = i; // C2106
    j * 4 = 7; // C2106

    // Correct usage: the dereferenced pointer is an lvalue.
    *p = i;

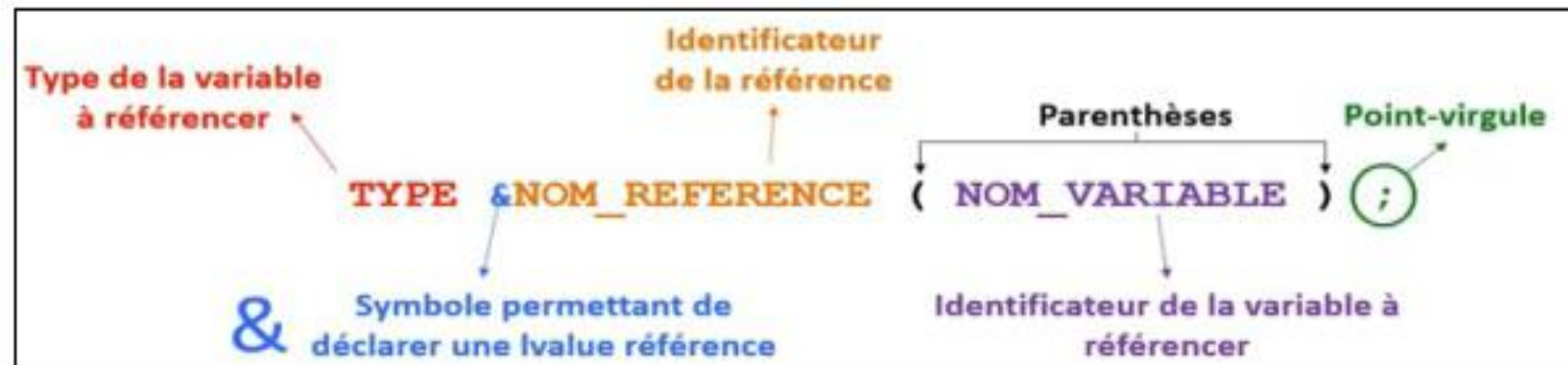
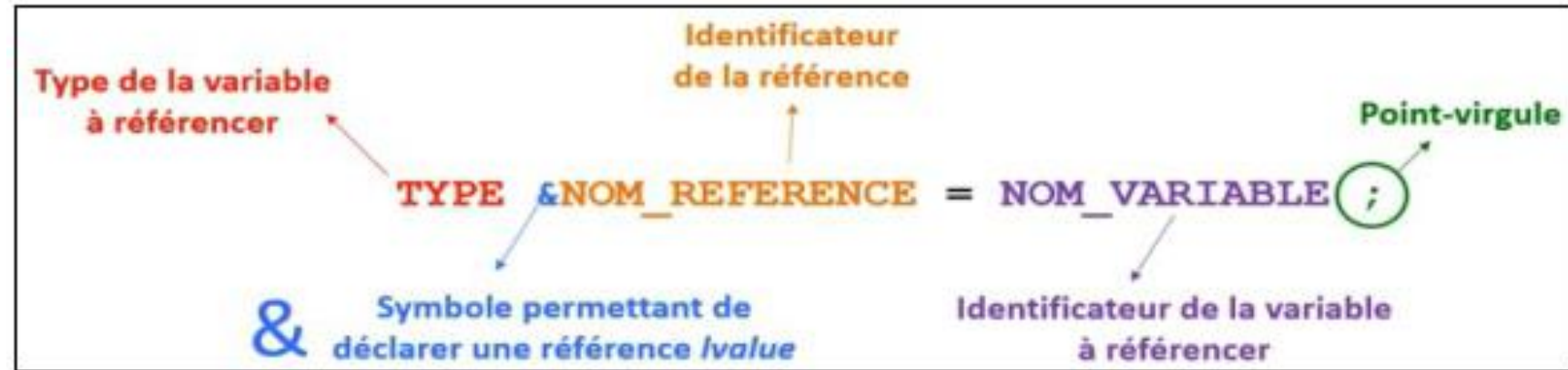
    // Correct usage: the conditional operator returns an lvalue.
    ((i < 3) ? i : j) = 7;

    // Incorrect usage: the constant ci is a non-modifiable lvalue (C3892).
    const int ci = 7;
    ci = 9; // C3892
}
```

# LES RÉFÉRENCES LVALUE

Contient l'adresse de l'objet, mais se comporte comme un objet.

Si & est précédé par un type de données alors c'est une référence de l'objet. Sinon on l'utilise comme adresse de l'objet.



# LES RÉFÉRENCES LVALUE

- impossible de modifier une référence lvalue
- une référence lvalue ne peut être initialisée qu'une seule fois (à la déclaration); ce qui implique qu'elle ne peut pas être nulle.



Une référence lvalue ne peut référencer qu'une seule variable tout au long de sa durée de vie.

```
#include <iostream>

using namespace std;

int main()
{
    int a=10, b= 55;
    int &r=a;

    cout << "A= " << a << endl;
    cout << "&r= " << r << endl;
    r++;
    cout << "A= " << a << endl;
    r=b;
    cout << "A= " << a << endl;
    return 0;
}
```

D:\Git\dev-i  
A= 10  
&r= 10  
A= 11  
A= 55

```
#include <iostream>

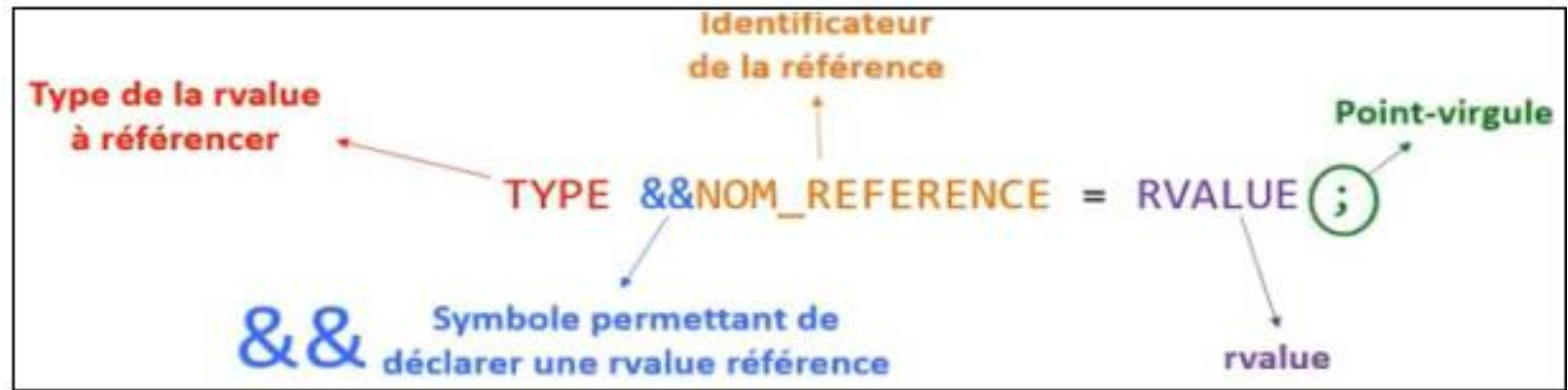
using namespace std;
void increment(int& a){
    a++;
}

int main()
{
    int a = 4;
    increment(a);
    cout << "Le resultat: " << a << endl;
    return 0;
}
```

D:\Git\dev-ising\AtelierCPP\refs  
Le resultat: 5

# LES RÉFÉRENCES RVALUE

Une référence rvalue est destinée à référencer seulement des rvalues (des valeurs temporaires).



On peut réaffecter une nouvelle valeur à une référence rvalue, un emplacement mémoire est alors créé pour y stocker la valeur. Donc la référence rvalue va référencer l'espace mémoire créé et non pas la rvalue elle même.

# LES RÉFÉRENCES RVALUE - EXEMPLE

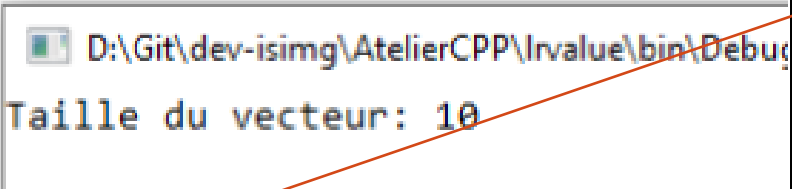
```
#include <iostream>
#include <vector>

using namespace std;

vector<int> getData()
{
    vector<int> data;
    for(int i=0; i<10; i++)
        data.push_back(i);

    return data;
}

int main()
{
    vector<int> &&vec=getData();
    cout << "Taille du vecteur: " << vec.size() << endl;
    return 0;
}
```



**data** qui est un objet temporaire créé dans la fonction *getData()* sera référencé par la référence sur rvalue, **vec**.

Donc l'espace temporaire persistera tant qu'il est référencé.

# TEMPLATES — FONCTIONS TEMPLATES

```
template<typename T>
T maxi(T a, T b) {
    return a>b?a:b;
}

int main()
{
    float res3 = maxi(5.3, 2.3);
    int res4 = maxi(10, 12);
    cout << "float maxi: " << res3 << endl;
    cout << "int maxi: " << res4 << endl;
    return 0;
}
```

On peut définir plusieurs *parameters* templates pour une fonction

```
1 template <class T, class U>
2 T GetMin (T a, U b) {
3     return (a<b?a:b);
4 }
```

Les **modèles de fonction** sont des fonctions spéciales qui peuvent fonctionner avec des types génériques.

Un **paramètre de modèle** est un type spécial de paramètre qui peut être utilisé pour passer un type comme argument

On peut écrire aussi:

```
template<class T>
T maxi(T a, T b) {
    return a>b?a:b;
}
```



# POINTEURS SUR LES FONCTIONS

Un pointeur de fonction contient l'adresse du début du code binaire constituant la fonction.

```
#include <iostream>
void afficherInt(int a){
    std::cout << "c'est un " << a << std::endl;
}
int main()
{
    void(*aff)(int); // pointeur sur la fonction
    aff = afficherInt;
    aff(6);
    return 0;
}
```

Le nom de la fonction représente son adresse de début.

Nom de la fonction

type (\*identificateur)(paramètres);

Les paramètre de la fonction

Type renvoyé par la fonction

# POINTEUR SUR LES FONCTIONS - EXERCICE

Etant donné un tableau d'entiers `tab`, on se propose de faire différentes manipulations sur ce tableau:

- Ecrire une fonction « `affichePair` » qui affiche un entier s'il est pair.
- Ecrire une fonction « `afficheImpair` » qui affiche un entier s'il est impair.
- Ecrire une fonction « `pourChaque` » qui applique une fonction (passée en paramètre) sur les éléments des tableaux.
- Ecrire une fonction principale `main` qui crée un tableau d'entiers et affiche pour chaque élément pair du tableau puis chaque élément impair du tableau.

# POINTEUR SUR LES FONCTIONS - CORRECTION

```
#include <iostream>
#include <vector>
void afficherPair(int a){
    if(!(a%2))
        std::cout << a << " est pair" << std::endl;
}
void afficherInPair(int a){
    if(a%2)
        std::cout << a << " est impair" << std::endl;
}
void PourChaque(const std::vector<int>& vec, void(*aff)(int)){
    for(int val : vec)
        aff(val);
}
int main()
{
    std::vector<int> tab = {3, 9, 2, 5, 7, 4, 12};
    PourChaque(tab, afficherPair);
    std::cin.get();
    PourChaque(tab, afficherInPair);
    return 0;
}
```

# LES FONCTIONS LAMBDA

```
[zone de capture](paramètres de la lambda) -> type de retour { instructions }
```

**La zone de capture :** par défaut, une lambda est en totale isolation et ne peut manipuler aucune variable de l'extérieur. Grace à cette zone, la lambda va pouvoir modifier des variables extérieures.

**Les paramètres de la lambda :** exactement comme pour les fonctions, les paramètres de la lambda peuvent être présents ou non, avec utilisation de références et/ou `const` possible.

**Le type de retour :** encore un élément qui nous est familier. Il est écrit après la flèche

# LES FONCTIONS LAMBDA - EXEMPLE

```
#include <iostream>
#include <string>

int main()
{
    [](std::string const & message) -> void { std::cout << "Message reçu : " << message << std::endl; };
    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <functional>
void PourChaque(const std::vector<int>& vec, const std::function<void(int&)>& aff) {
    for(int val : vec)
        aff(val);
}
```

```
int main()
{
    std::vector<int> tab = {3, 9, 2, 5, 7, 4, 12};
    int b = 6;
    PourChaque(tab, [&](int& a) {
        std::cout << a + b << std::endl;
    });
    std::cin.get();

    return 0;
}
```

Passage par référence des objets externes capturés par le lambda



# LES ARGUMENTS PAR DÉFAUT D'UNE FONCTION

Les arguments par défaut d'une fonction permettent à cette dernière d'être appelée sans fournir tout ses paramètres.

```
#include <iostream>

void afficherMessage(std::string msg = "Bonjour") {
    std::cout << msg << std::endl;
}

int main()
{
    afficherMessage("Bonsoir");
    afficherMessage("Salut");
    afficherMessage();
}
```

Les paramètres ayant une valeur par défaut **devront être placés à la fin de la liste** des paramètres de la fonction

# LES CLASSES EN C++

Une **classe** est le résultat d'un processus *d'abstraction* et *d'encapsulation*.

Une classe encapsule les *données* (attributs) avec les *traitements*  
(méthodes).

Une classe définit un **type**.

Une réalisation particulière d'une classe s'appelle un **objet**.

# LES CLASSES EN C++

```
class Rectangle{  
private:  
    double largeur ;  
    double hauteur ;  
  
public:  
    double perimetre() const  
    {  
        return (largeur+hauteur)*2;  
    }  
    void agrandir( double tau)  
    {  
        largeur *= (1+tau);  
        hauteur *= (1+tau);  
    }  
};
```

Nom de la classe

Visibilité des attributs (en général **privée**)

Attributs de la classe (données)

Visibilité des méthodes (en général **publique**)

Méthode prédicat (ne change pas les valeurs des attributs)

Méthode action (peut changer les valeurs des attributs)

Si jamais une méthode déclarée **const** doit changer la valeur d'une variable alors on devrait déclarer cette variable mutante (**mutable**).



# LES CLASSES EN C++ - THIS

En cas d'ambiguïté de nom entre les attributs et d'autres variables dans les méthodes, on dispose du pointeur sur l'instance courante, **this**, qui nous aide à pointer les attributs de la classe.

```
void setLargeur(double largeur)
{
    this->largeur = largeur;
}
void setHauteur(double hauteur)
{
    this->hauteur = hauteur;
}
```

# DÉFINITION EXTERNE D'UNE CLASSE

Pour une meilleure représentation d'une classe en C++, on peut définir un prototype de la classe dans un fichier .h et la définition de ses méthodes dans un autre fichier .cpp.

```
#include <iostream>
#include "Rectangle.h" ★
using namespace std;
int main()
{
    Rectangle r1;
    r1.setHauteur(1);
    r1.setLargeur(1);
    cout << "Perimetre de r1= " << r1.perimetre() << endl;
    r1.agrandir(0.5);
    cout << "Perimetre de r1 devient = " << r1.perimetre() << endl;
    return 0;
}
```

```
Rectangle.h X Rectangle.cpp X
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    public:
        double perimetre() const;
        void agrandir( double tau);
        double getLargeur() const;
        double getHauteur() const;
        void setLargeur(double largeur);
        void setHauteur(double hauteur);

    private:
        double largeur;
        double hauteur;
}; ★

#endif // RECTANGLE_H
```

```
Rectangle.cpp X Rectangle.h X
#include "Rectangle.h" ★
double Rectangle::perimetre() const
{
    return (largeur+hauteur)*2;
}
void Rectangle::agrandir(double tau)
{
    this->hauteur *= (1+tau);
    this->largeur *= (1+tau);
}
double Rectangle::getLargeur() const
{
    return largeur;
}
double Rectangle::getHauteur() const
{
    return hauteur;
}
void Rectangle::setLargeur(double largeur)
{
    this->largeur = largeur;
}
void Rectangle::setHauteur(double hauteur)
{
    this->hauteur = hauteur;
}
```

# CONSTRUCTEURS

- Les constructeurs sont des méthodes particulières qui ont pour responsabilité d'initialiser les attributs de la classe.
- Le constructeur porte le même nom que la classe, n'a pas de type de retour et est invoqué automatiquement où on crée une instance.
- Une classe peut avoir plusieurs constructeurs
- Si aucun constructeur n'est spécifié le compilateur fournit un constructeur par défaut à la classe ( **attention** : laisse non initialisé les attributs de type de base).

# CONSTRUCTEURS

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
    public:
        // les constructeurs
        Rectangle(double h, double l);
        Rectangle();
        // les methodes
        double perimetre() const;
        void agrandir( double tau);
        double getLargeur() const;
        double getHauteur() const;

    private:
        double largeur;
        double hauteur;
};
```

```
#include "Rectangle.h"

Rectangle::Rectangle(double h, double l)
{
    this->hauteur = h;
    this->largeur = l;
}

Rectangle::Rectangle()
{
    this->hauteur = 0;
    this->largeur = 0;
}
```

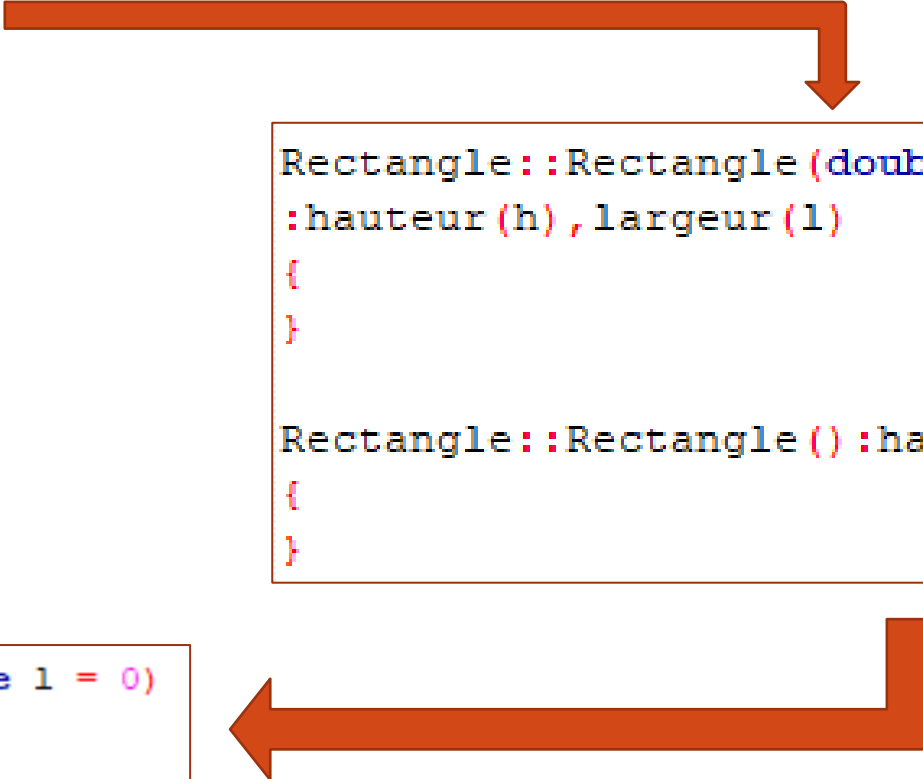
# CONSTRUCTEURS – LISTE D'INITIALISATION

les listes d'initialisation servent pour initialiser les attributs et/ou d'appeler leurs constructeurs (s'ils sont des objets).

```
#include "Rectangle.h"

Rectangle::Rectangle(double h,double l)
{
    this->hauteur = h;
    this->largeur = l;
}

Rectangle::Rectangle()
{
    this->hauteur = 0;
    this->largeur = 0;
}
```



```
Rectangle::Rectangle(double h,double l)
:hauteur(h),largeur(l)
{
}

Rectangle::Rectangle():hauteur(0),largeur(0)
{
}
```

```
Rectangle::Rectangle(double h = 0,double l = 0)
:hauteur(h),largeur(l)
{
}
```

# CONSTRUCTEURS PAR DÉFAUT

A :

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

Constructeur  
par défaut  
implicite

Constructeur par  
défaut qui  
initialise les  
attributs à 0

C :

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

B :

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

Constructeur  
par défaut qui  
initialise les  
attributs à 0

Pas de  
constructeur  
par défaut

D :

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

# EXERCICE

- On désire représenter l'objet Point caractérisé par un abscisse et un ordonné de type réel. Un point est construit de deux manières:
  - Les coordonnées sont par défaut à 0
  - Les coordonnées sont initialisés par des valeurs utilisateurs
- Un point devrait être afficher sous cette forme (1.3 , 2.9)
- Un point peut être déplacer selon l'axe des abscisses et/ou l'axe des ordonnés.

Définir la classe point.

Ecrire une fonction main qui crée un point A à l'origine du repère. Puis on crée un autre point B ayant des coordonnées saisis au clavier. Afficher les deux points.

En fin, faire confondre les deux points et les afficher de nouveau.



# EXERCICE - SOLUTION

```
class point
{
private:
    double abs;
    double ord;
public:
    point(double ab=0,double o=0);
    void deplacer(double, double) ;
    void afficher() const;
    double get_abs() const;
    double get_ord() const;
    void set_abs(double);
    void set_ord(double);
};
```

**point.h**

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

point::point(double ab, double o):abs(ab),ord(o){}
void point::deplacer(double x, double y)
{
    this->abs += x;
    this->ord += y;
}
void point::afficher() const
{
    std::cout << "(" << abs << " , " << ord << ")\n";
}
double point::get_abs() const
{
    return abs;
}
double point::get_ord() const
{
    return ord;
}
void point::set_abs(double abs)
{
    this->abs = abs;
}
void point::set_ord(double ord)
{
    this->ord = ord;
}
```

**point.cpp**



# EXERCICE - SOLUTION

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

int main()
{
    double x, y;
    point A;
    std::cout << "Donner les coordonnes de B:";
    std::cin >> x;
    std::cin >> y;

    point B(x,y);
    std::cout << "Le point A:";
    A.afficher();
    std::cout << "Le point B:";
    B.afficher();
    std::cout << "confondre A et B ....\n";
    A.set_abs(B.get_abs());
    A.set_ord(B.get_ord());
    std::cout << "Le point A:";
    A.afficher();
    std::cout << "Le point B:";
    B.afficher();
}
```

# CONSTRUCTEURS DE COPIE

Ce genre de constructeur permet d'initialiser un objet avec un autre objet.

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

Le compilateur génère automatiquement un constructeur de copie par défaut en absence d'un fourni par le développeur. Ce constructeur de copie par défaut il est semblable au constructeur de copie définie en haut. Pour cela, on se contente en général par celui fourni par le compilateur et on ne définit pas de constructeur de copie.

# COPIE DE SURFACE VS COPIE PROFONDE

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

```
void afficher_largeur(Rectangle tmp)  
{  
    cout << "Largeur: " << tmp.getLargeur() << endl;  
}
```

```
int main()  
{  
    Rectangle r(2.0,3.5);  
    afficher_largeur(r);  
    cout << "Hauteur:" << r.getHauteur() << endl;  
    r.setHauteur(4.6);  
    afficher_largeur(r);  
    cout << "Hauteur:" << r.getHauteur() << endl;  
  
    return 0;  
}
```

```
Largeur :2  
Hauteur:3.5  
Largeur :2  
Hauteur:1.33484e-306
```

# CONSTRUCTEUR DE COPIE PROFONDE

Cette fonction crée une copie tmp (via le constructeur de copie fournit par défaut). À la fin de l'exécution de cette fonction l'objet tmp sera détruit, ce qui provoque la perte des espaces mémoires de largeur et hauteur. De ce fait, l'objet initial (utilisé lors de l'appel de la fonction `afficher_largeur()` sera biaisé).



```
Rectangle(const Rectangle& obj)
: largeur(new double(*(obj.largeur))) ,
  hauteur(new double(*(obj.hauteur)))
{}
```

# DEFAULTED AND DELETED FUNCTIONS

---

On peut marquer un constructeur comme un constructeur par défaut en utilisant le mot clé **default**.

```
Rectangle() = default;
```

Indiquer explicitement que c'est un constructeur par défaut

On peut interdire l'accès à un constructeur ou à une méthode avec l'opérateur **delete**.

```
Point(const Point& pt) = delete;
```


Interdire le copiage d'un objet

# DEFAULTED AND DELETED FUNCTIONS

```
#include <iostream>
class Point{
public:
    Point() = default;
    Point(const Point& pt) = delete;
    Point(double x = 0, double y = 0) : x(x), y(y)
    { }
    void afficher() const
    {
        std::cout << "(" << x << "," << y << ")" << std::endl;
    }
private:
    double x, y;
};

int main()
{
    Point p1{2.0f, 3.5f};
    Point p2 = p1;
    p1.afficher();
    p2.afficher();

    return 0;
}
```




Code::Blocks X Search results X Cccc X Build log X Build messages X CppChe		
File	Line	Message
Enseignement ...		In function 'int main()':
Enseignement ...	18	error: use of deleted function 'Point::Point(const Point&)'

# LES DESTRUCTEURS

- Il est important de pouvoir gérer l'espace mémoire alloué pour les objets de notre programme après usage. Le C++ offre une méthode appelée destructeur invoquée automatiquement en fin de vie de l'instance.
- Le destructeur d'une classe est une méthode ayant le même nom de la classe précédé par le signe ~, sans paramètres et ne supporte pas de surcharge. Donc pour une classe il y a un seul destructeur.

```
class Livre
{
private:
    string *titre;
public:
    Livre(string *tit):titre(tit){}
    ~Livre(){ delete titre; }
};
```



Si on définit pas un destructeur explicitement pour une classe, le compilateur nous fournit un par défaut.

Destructeur qui libère l'espace alloué dynamiquement

# LES DESTRUCTEURS - UTILISATION

- A local (automatic) object with block scope goes out of scope.
- An object allocated using the new operator is explicitly deallocated using delete.
- The lifetime of a temporary object ends.
- A program ends and global or static objects exist.
- The destructor is explicitly called using the destructor function's fully qualified name.



# SURCHARGE D'OPÉRATEURS

L'intérêt de la surcharge des opérateurs est d'adapter leur comportement à des cas particuliers voulus par le développeur.

La surcharge des opérateurs peut être une surcharge externe ou interne par rapport à la classe où ils s'appliquent.

```
class Complexe
{
    .....
};

class Matrice
{
    .....
};

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

Surcharge externe (via une fonction)

```
const Complexe operator+(Complexe const& z1, Complexe const& z2) {
    // retourner un objet anonyme
    return Complexe(z1.get_x()+z2.get_x(), z1.get_y()+z2.get_y());
}
```

```
Complexe c1(2.0, 3.0);
Complexe c2(2.0, 4.0);
Complexe c3;
c3 = c1 + c2;
```

# SURCHARGE D'OPÉRATEURS – SURCHARGE INTERNE

```
class Complexe
{
private:
    double x;
    double y;
public:
    Complexe operator+(Complexe const& z2) const
    {
        return Complexe(x+z2.get_x(),y+z2.get_y());
    }
    Complexe():x(0),y(0){}
    Complexe(double x, double y):x(x),y(y){}
    double get_x()const {return x;}
    double get_y() const {return y;}
};
```

→

```
Complexe c1(2.0,3.0);
Complexe c2(2.0,4.0);
Complexe c3;
c3 = c1 + c2;
```

# L'HÉRITAGE

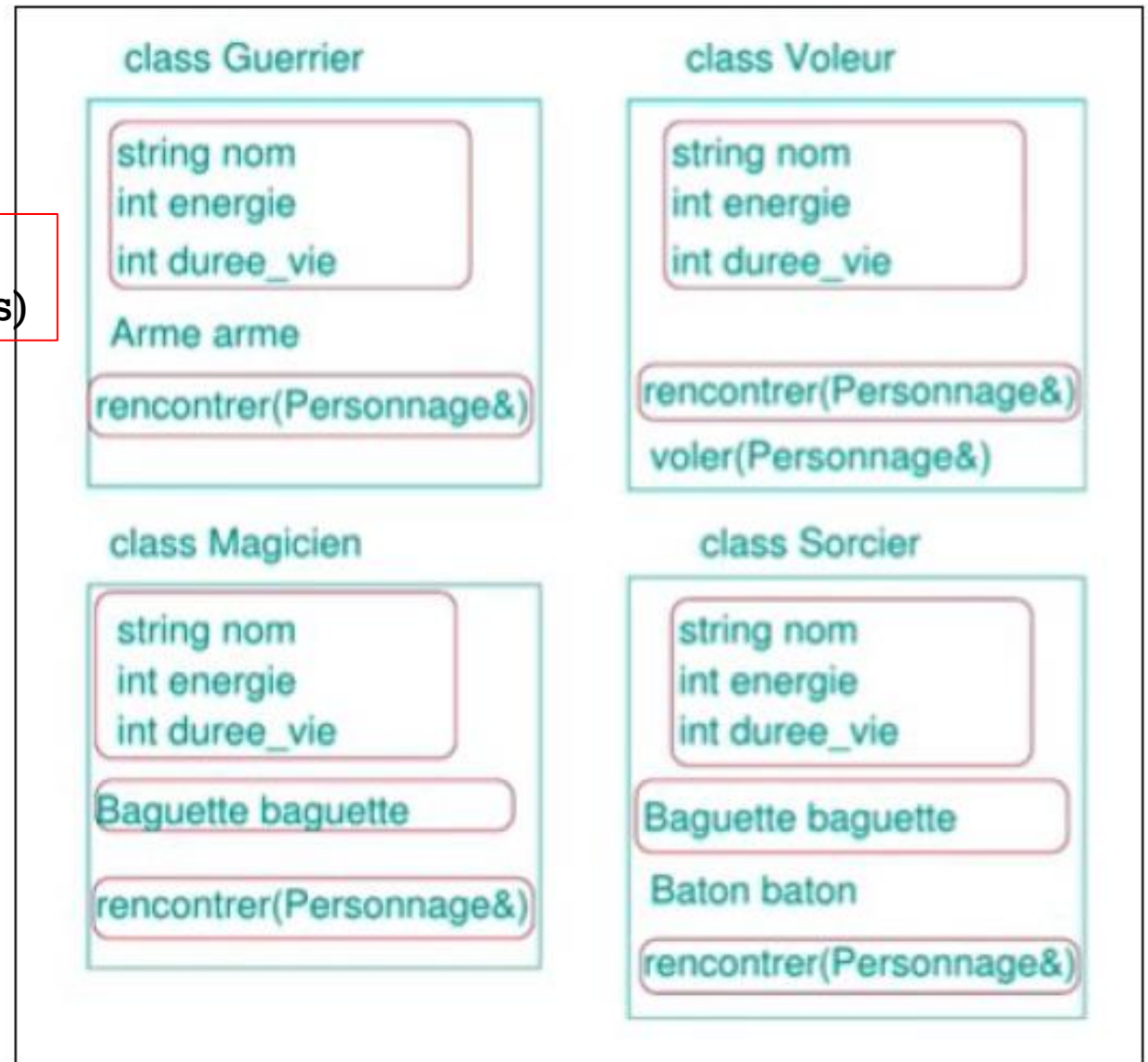
- Un code moins facile à entretenir
- Redondance de code (données et traitements)



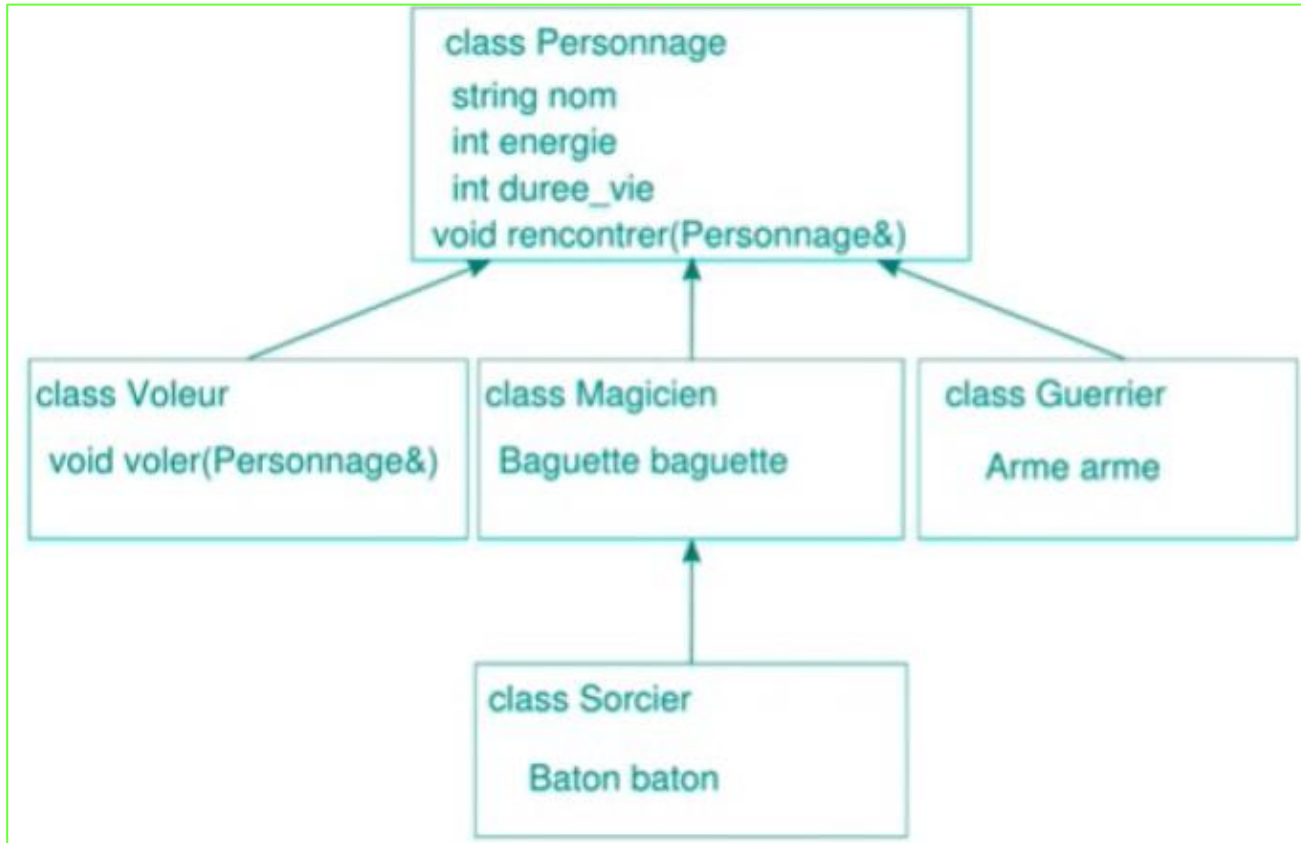
- Regrouper les informations et les comportements en commun
- Plus intuitif
- Plus facile à maintenir
- Permettre la réutilisation du code
- Une forte capacité d'évolution



Héritage (Eng: *inheritance*)



# L'HÉRITAGE



```
class Personnage {
    // ...
};
// ...
class Guerrier : public Personnage {
public:
    // constructeurs, etc.
private:
    Arme arme;
};
```

Pour mettre en oeuvre l'héritage, on utilise l'opérateur " : ".

# L'HÉRITAGE

**public**: visibilité totale à l'intérieur et à l'extérieur de la classe.

**private**: visibilité uniquement à l'intérieur de la classe.

**protected**: visibilité dans la classe et dans les classes de sa descendance.

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

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

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

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

Attention les attributs **protected** ne sont accessibles dans les classes filles que s'ils sont dans la portée de cette classe fille (c-à-d directement via **this** ou indirectement via une instance de la classe fille).

# L'HÉRITAGE - CONSTRUCTEURS

Lors de l'instanciation d'une sous classe, il faut initialiser:

- les attributs de la sous classe
- les attributs hérités des super-classes

**L'initialisation des attributs hérités doit se faire dans la classe où ils sont explicitement définis.**



**L'initialisation des attributs privés doit se faire en invoquant les constructeurs des super-classes**

```
SousClasse(liste de paramètres)
: SuperClasse(Arguments),
  attribut1(valeur1),
  ...
  attributN(valeurN)
{
    // corps du constructeur
}
```

Syntaxe



# L'HÉRITAGE - CONSTRUCTEURS

## Remarques:

- Lorsque la super classe admet un constructeur par défaut, alors son invocation explicite au niveau de la sous-classe *ne sera pas obligatoire*. Le compilateur réalise implicitement l'invocation du constructeur par défaut.
- L'appel du constructeur de la super classe doit se faire **en premier lieu** dans la liste d'initialisation
- Il n'est pas nécessaire que la sous classe ait ses propres attributs (supplémentaires)
- Le **constructeur de copie** d'une sous-classe doit *invoquer explicitement* le constructeur de copie de la super-classe (sinon c'est le constructeur par défaut qui sera invoqué par le compilateur).

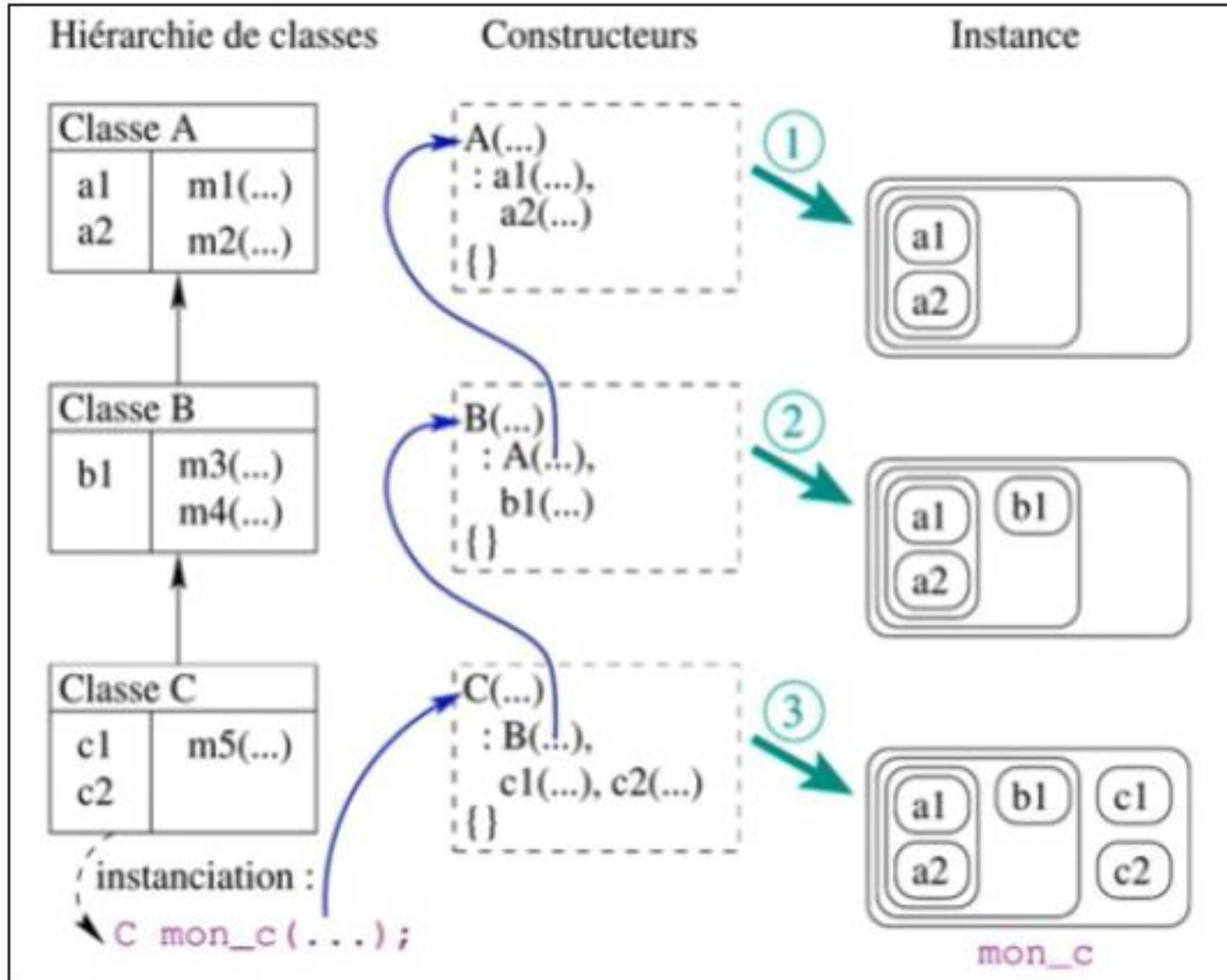
# L'HÉRITAGE — CONSTRUCTEURS - EXEMPLE

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

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



# L'HÉRITAGE — CONSTRUCTEURS



La destruction des objets se fait dans le sens inverse de la construction.

# HÉRITAGE - POLYMORPHISME

Résolution statique des liens

```
class Rectangle
{
    private:
        double largeur;
        double hauteur;
    public:
        Rectangle(double l, double h);
        ~Rectangle();
        void quiEstTu();
};
```

```
Rectangle::Rectangle(double l, double h)
: largeur(l), hauteur(h) {}

Rectangle::~~Rectangle() { }
void Rectangle::quiEstTu()
{
    std::cout << "je suis un rectangle" << std::endl;
}
```

L'objet est reconnu par son type de déclaration

```
class Carre : public Rectangle
{
    public:
        Carre(double);
        void quiEstTu();
};
```

```
Carre::Carre(double l) : Rectangle(l, l) {}
void Carre::quiEstTu()
{
    std::cout << "Je suis un carre" << std::endl;
}
```

```
#include <vector>
#include "Rectangle.h"
#include "Carre.h"
using namespace std;
int main()
{
    Rectangle r(1.5, 2.9);
    Carre car(9);
    vector<Rectangle> vec;
    vec.push_back(r);
    vec.push_back(car);
    for(auto ob:vec)
        ob.whoIs();
    return 0;
}
```

je suis un rectangle  
je suis un rectangle

# HÉRITAGE - POLYMORPHISME

Résolution dynamique des liens

```
class Rectangle
{
    private:
        double largeur;
        double hauteur;
    public:
        Rectangle(double l, double h);
        ~Rectangle();
        virtual void quiEstTu();
};
```

- Déclarer la méthode comme virtuelle avec le mot clé **virtual**, cette déclaration se fait dans la classe la plus générale qui admet cette méthode.
- Utiliser les **références** et/ou les **pointeurs** pour invoquer les méthodes virtuelles (passage par références ou par adresse, la construction dynamique des instances avec new)

La résolution dynamique des liens permet de choisir la méthode à exécuter au moment de **l'exécution** selon la nature réelle des instances dans la mémoire.

```
#include <iostream>
#include<vector>
#include"Rectangle.h"
#include"Carre.h"
using namespace std;
int main()
{
    Rectangle* r=new Rectangle(1.5,2.9);
    Carre *car=new Carre(9);
    vector<Rectangle*> vec;
    vec.push_back(r);
    vec.push_back(car);
    for(auto ob:vec)
        ob->quiEstTu();
    return 0;
}
```

je suis un rectangle  
Je suis un carre

# HÉRITAGE - POLYMORPHISME

- La virtualité est transmise par transitivité à la méthode de la classe fille.
- Lorsque la méthode virtuelle est invoquée à partir d'une référence ou d'un pointeur vers une instance, c'est la méthode du type réel de l'instance qui sera exécutée.
- Il est conseillé de toujours définir des destructeurs comme virtuels (pour éviter la destruction partielle des objets)
- Un constructeur ne peut pas être virtuel (s'il utilise des méthodes virtuelles le caractère virtuel de ces méthodes sera ignoré)

# FINAL — CLASSE VS MÉTHODE

- Depuis C++11, si une classe est déclarée **final** alors on ne peut pas dériver d'elle une autre classe.

```
class BaseClass final
{
};

class DerivedClass: public BaseClass // compiler error: BaseClass is
                                     // marked as non-inheritable
{
};
```

- Une méthode déclarée **final** ne peut pas être redéfinie au niveau des classes dérivées.

```
class BaseClass
{
    virtual void func() final;
};

class DerivedClass: public BaseClass
{
    virtual void func(); // compiler error: attempting to
                        // override a final function
};
```

# CLASSE ABSTRAITE ET MÉTHODE VIRTUELLE PURE

Une méthode virtuelle pure dans une classe n'a pas de comportement significatif. On ne sait son comportement que dans les classes dérivées de la classe où elle est défini.

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

## Exemple:

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

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

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

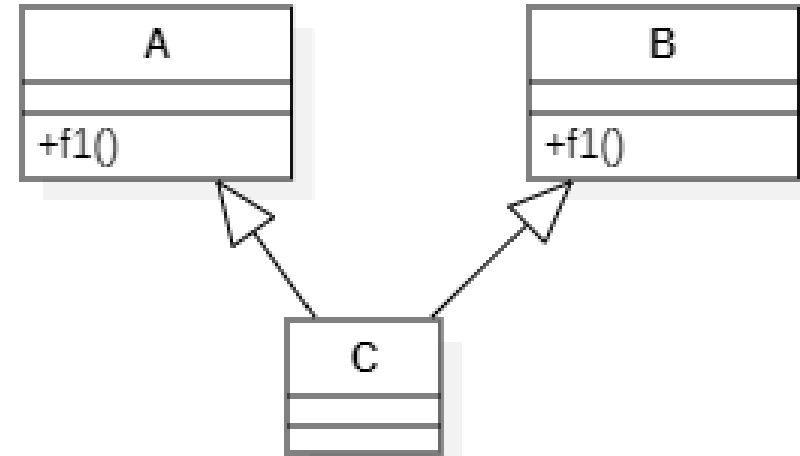


# HÉRITAGE MULTIPLE

```
class A {  
public:  
    void f1(){}  
    void f2(){}  
};
```

```
class B {  
public:  
    void f1(){}  
    void f3(){}  
};
```

```
class C :public A, public B {  
public:  
    // void f1(){} ici f1 de C  
    // masque les deux hérités  
    void g() {  
        A::f1(); // ou B::f1()  
        // il faut résoudre l'ambiguïté  
        f3();  
    }  
};
```



```
int main()  
{  
    C c;  
    c.A::f1(); // résoudre l'ambiguïté  
    c.f2(); // Ok pas d'ambiguïté  
}
```

L'appel des constructeurs des superclasses se fait dans l'ordre d'héritage.

# H. MULTIPLE – RÉOLUTION DE L'AMBIGUÏTÉ

Solution 1

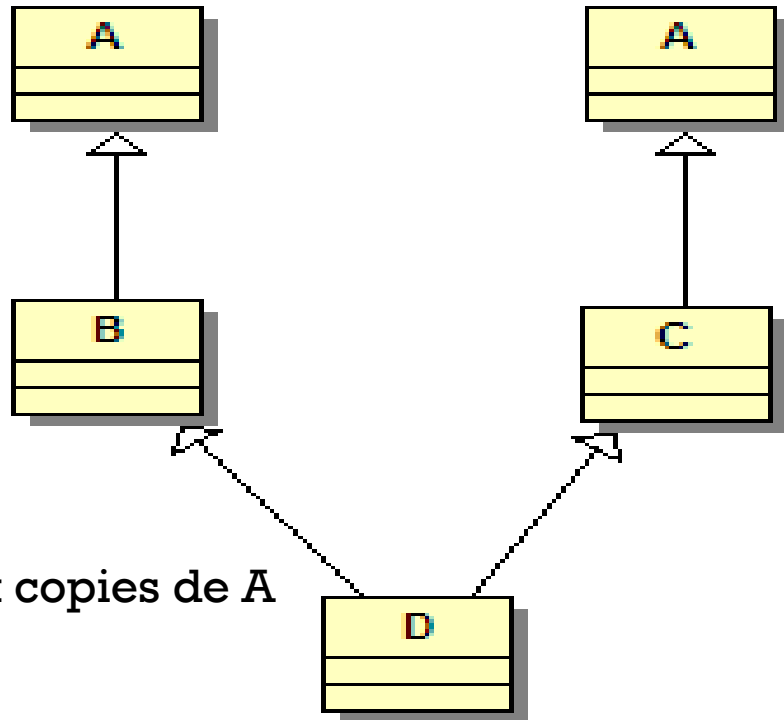
```
class C : public B, public A {  
public:  
    using B::f1;  
    void g() {  
        f1();  
        f3();  
    }  
};
```

Solution 2

```
class C : public B, public A {  
public:  
    void f1() {  
        // cette méthode masque les f1 hérités  
    }  
    void g() {  
        f1();  
        f3();  
    }  
};
```



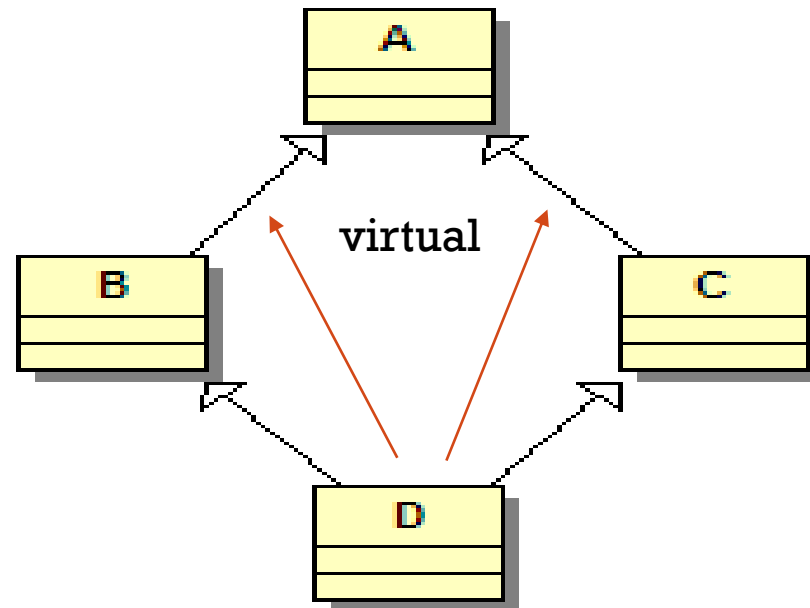
# HÉRITAGE MULTIPLE



D a deux copies de A

```
class A {};  
class B : public A {};  
class C : public A {};  
class D : public B, public C {};
```

## Héritage en Diamond



Un seul objet A est hérité pour la classe D

```
class A {};  
class B : public virtual A {};  
class C : public virtual A {};  
class D : public B, public C {};
```

# HÉRITAGE MULTIPLE

```
class Animal { ... };  
class Ovipare : public virtual Animal { ... };  
class Vivipare : public virtual Animal { ... };  
class Ovovivipare : public Ovipare, public Vivipare { public:  
    Ovovivipare(...)  
    : Animal(...), Ovipare(...), Vivipare(...), ...  
    { ... }  
};
```

Si la classe `Animal` ne possède pas un constructeur par défaut alors la classe la plus dérivée doit invoquer explicitement le constructeur de la classe de Base (`Animal`). Alors les appels dans les classes intermédiaires seront ignorés.

# HÉRITAGE MULTIPLE

```
class A {
public:
    A() { std::cout << "Constructeur A\n"; }
    void f1(){}
};
class B :public A{
public:
    B() { std::cout << "Constructeur B\n"; }
    void f2(){}
};
class C :public A {
public:
    C(){ std::cout << "Constructeur C\n"; }
    void f3() {}
};
class D :public B, public C {
public:
    D() { std::cout << "Constructeur D\n"; }
};
int main()
{
    D d;
}
```

Console de débogage Micro

```
Constructeur A
Constructeur B
Constructeur A
Constructeur C
Constructeur D
```

```
class A {
public:
    A() { std::cout << "Constructeur A\n"; }
    void f1(){}
};
class B :public virtual A{
public:
    B() { std::cout << "Constructeur B\n"; }
    void f2(){}
};
class C :public virtual A {
public:
    C(){ std::cout << "Constructeur C\n"; }
    void f3() {}
};
class D :public B, public C {
public:
    D() { std::cout << "Constructeur D\n"; }
};
int main()
{
    D d;
}
```

Console de débogage Micro

```
Constructeur A
Constructeur B
Constructeur C
Constructeur D
```

