

# Programmation orientée objet & autres concepts illustrés en C++11 et en Java

Eric Lecolinet - Télécom ParisTech

<http://www.telecom-paristech.fr/~elc>

**24 Septembre 2019**

# Dans ce cours

## Organisation du cours

- présentation des langages informatique par **Patrick Bellot**
- ce qui est **similaire à Java** (révision...)
- ce qui est **différent de Java**
- interfaces graphiques **Java Swing**

## Deux langages support

- **C++** : pour illustrer divers **concepts**, **mécanismes** et **difficultés** présents dans les langages courants
- **Java** : pour **comparer** et pour illustrer la programmation événementielle

## Liens

- <http://www.telecom-paristech.fr/~elc/>
- <http://www.telecom-paristech.fr/~elc/inf224/>

# Brève historique (très parcellaire)

## 1972 : Langage C

AT&T Bell Labs

## 1983 : Objective C

NeXt puis Apple

Extension **objet** du **C**

Syntaxe inhabituelle inspirée de **Smalltalk**

## 1985 : C++

AT&T Bell Labs

Extension **object** du **C** par *Bjarne Stroustrup*

## 1991 : Python

Guido van Rossum (CWI)

Vise simplicité et rapidité d'écriture

**Interprété**, typage **dynamique** (= à l'exécution)

## 1995 : Java

Sun Microsystems puis Oracle

Simplification du **C++**, **purement objet**

Egalement inspiré de **Smalltalk**, ADA, etc.

## 2001: C#

Microsoft

Originellement proche de **Java** pour **.NET**

Egalement inspiré de **C++**, Delphi, etc.

## 2011: C++11

**Révision** majeure du **C++** suivie de **C++14**, **C++17**, **C++20**

## 2014: Swift

Apple

Successeur d'**Objective C**

# C++ versus C et Java

## C++

- à l'origine : **extension du C** => un compilateur C++ peut **compiler du C**
- un même programme peut **combinaison C/C++** et d'autres langages (C#, Swift, etc.)
- **C** et **C++** existent **partout** => avantageux pour développement multi-plateformes
  - ex : même base C/C++ pour IOS et Android
- **C/C++** est généralement **plus rapide** (si on sait s'en servir !)

## Java

- à l'origine : **simplification du C++**
- beaucoup de **ressemblances** mais aussi des **différences** importantes

## Compilation

- **Java** : code source **compilé** (byte code) **puis interprété** (ou compilé à la volée)
- **C/C++** : **compilé en code natif**

# Références et liens

## Livres, tutoriaux, manuels

- **Le langage C++**, Bjarne Stroustrup (auteur du C++), Pearson  
son site : <http://www.stroustrup.com>
- **manuel de référence** : <http://cppreference.com> ou [www.cplusplus.com/reference](http://www.cplusplus.com/reference)
- **faqs, aide** : <https://isocpp.org/faq>, <https://stackoverflow.com>, etc.
- **Cours C++ de Christian Casteyde** : <http://casteyde.christian.free.fr/>

## Autres liens utiles

- **Travaux Pratiques** de ce cours : [www.enst.fr/~elc/cpp/TP.html](http://www.enst.fr/~elc/cpp/TP.html)
- **Toolkit graphique Qt** : [www.enst.fr/~elc/qt](http://www.enst.fr/~elc/qt)
- **Extensions Boost** : [www.boost.org](http://www.boost.org)
- **Petit tutoriel de Java à C++** (pas maintenu) : <http://www.enst.fr/~elc/C++/>

# Premier chapitre :

## Programme, classes, objets

# Programme C++

## Constitué

- de **classes** comme en **Java**
- et, éventuellement, de **fonctions** et **variables** « non-membre » (= hors classes) comme en **C**

## Bonne pratique : une classe principale par fichier

- mais pas de contrainte syntaxique comme en **Java**

### Car.cpp

```
#include "Car.h"

void Car::start() {
    ....
}
```

### Truck.cpp

```
#include "Truck.h"

void Truck::start(){
    ....
}
```

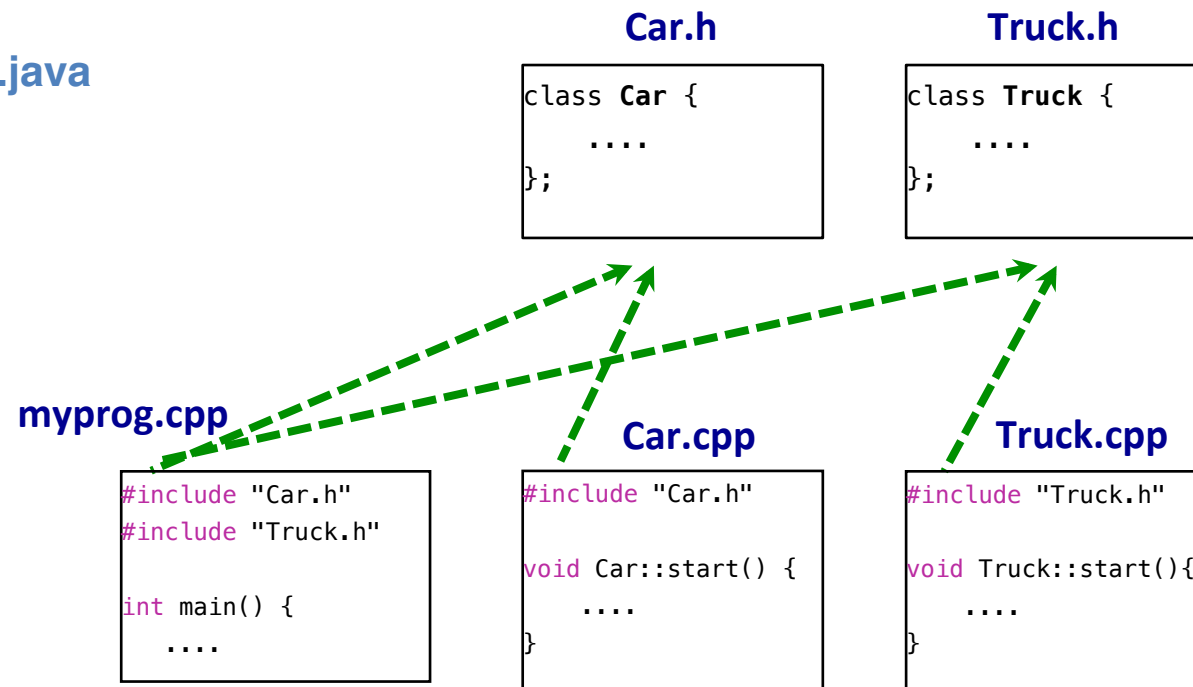
# Déclarations et définitions

## C/C++ : deux types de fichiers

- **déclarations** dans fichiers **header** (extension **.h** ou **.hpp** ou pas d'extension)
- **définitions** dans fichiers d'**implémentation** (**.cpp**)
- en général à chaque **.h** correspond un **.cpp**

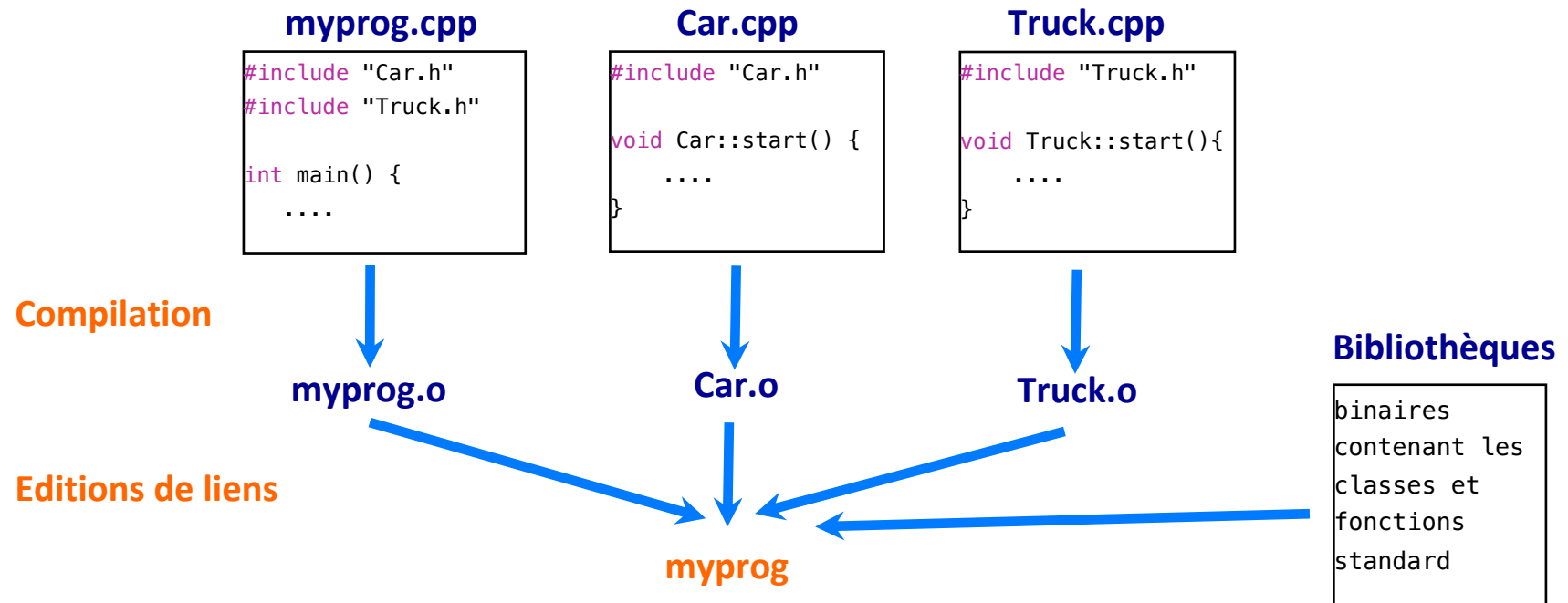
## En Java

- tout dans les **.java**





# Compilation et édition de liens



## Problèmes éventuels

- **incompatibilités syntaxiques** :
  - la compilation échoue : **compilateur** pas à jour
- **incompatibilités binaires** :
  - l'édition de liens échoue : **bibliothèques** pas à jour

## Options g++

- mode **C++11** : **-std=c++11**
- warnings : **-Wall -Wextra** ...
- débogueur : **-g**
- optimisation : **-O1 -O2 -O3 -Os -s**
- et bien d'autres ...

# Déclaration de classe

Dans le header **Circle.h** :

```
class Circle {  
private:  
    int x = 0, y = 0;  
    unsigned int radius = 0;  
  
public:  
    Circle(int x, int y, unsigned int radius);  
  
    void setRadius(unsigned int);  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
}; // ne pas oublier ; à la fin !
```

← variables d'instance

← constructeur

← méthodes d'instance

## Remarques

- même sémantique que **Java** (à part **const**)
- il faut un **;** après la **}**

# Variables d'instance

```
class Circle {  
private:  
    int x = 0, y = 0;  
    unsigned int radius = 0;  
  
public:  
    Circle(int x, int y, unsigned int radius);  
    void setRadius(unsigned int)  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
};
```

← variables d'instance

## Variables d'instance

- chaque objet possède **sa propre copie** de la variable
- normalement **private** ou **protected** (à suivre)
- **doivent être initialisées** si c'est des **types de base** ou des **pointeurs**
  - NB : avant C++11 : le faire dans les constructeurs

# Constructeurs

```
class Circle {  
private:  
    int x = 0, y = 0;  
    unsigned int radius = 0;  
  
public:  
    Circle(int x, int y, unsigned int radius); ← constructeur  
    void setRadius(unsigned int)  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
};
```

## Les constructeurs

- sont appelés quand les objets sont **créés** afin de les **initialiser**
- sont **toujours chaînés** :
  - les constructeurs des **superclasses** sont exécutés dans l'ordre **descendant**
  - pareil en **Java** (et pour tous les langages à objets)

# Méthodes d'instance

```
class Circle {  
private:  
    int x = 0, y = 0;  
    unsigned int radius = 0;  
  
public:  
    Circle(int x, int y, unsigned int radius);  
    void setRadius(unsigned int)  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
};
```

← méthodes d'instance

## Méthodes d'instance : 1<sup>er</sup> concept fondamental de l'OO

- liaison **automatique** entre fonctions et données :
  - ont **accès** aux variables d'**instance** (et de **classe**) d'un objet

## Remarques

- généralement **public**
- méthodes **const** : ne modifient **pas** les variables d'instance (n'existent pas en **Java**)

# Définition des méthodes

Dans le fichier d'implémentation  
**Circle.cpp** :

```
#include "Circle.h"

Circle::Circle(int _x, int _y, unsigned int _r) {
    x = _x;
    y = _y;
    radius = _r;
}

void Circle::setRadius(unsigned int r) {
    radius = r;
}

unsigned int Circle::getRadius() const {
    return radius;
}

unsigned int Circle::getArea() const {
    return 3.1416 * radius * radius;
}
```

Header Circle.h

```
class Circle {
private:
    int x, y;
    unsigned int radius;
public:
    Circle(int x, int y, unsigned int radius);
    void setRadius(unsigned int);
    unsigned int getRadius() const;
    unsigned int getArea() const;
    ....
};
```

insère le contenu de Circle.h

précise la classe :: typique du C++

# Définitions dans les headers

## Dans le header `Circle.h`

```
class Circle {  
private:  
    int x = 0, y = 0;  
    unsigned int radius = 0;  
  
public:  
    void setRadius(unsigned int r) {radius = r;}  
    unsigned int getRadius() const {return radius;}  
    ....  
};
```

← ..... méthodes inline

## Méthode **inline** = définie dans un header

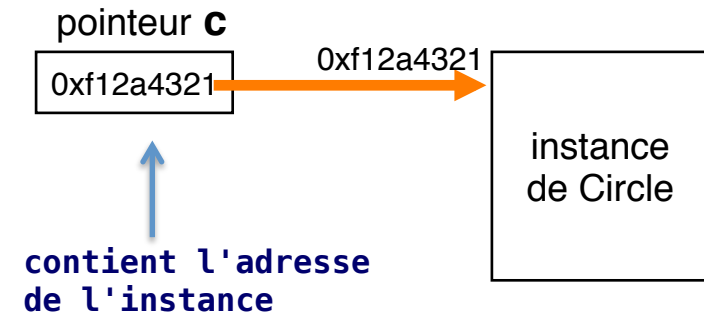
- en **théorie** : appel fonctionnel **remplacé** par son **code source**
  - exécution + rapide, *mais* exécutable + lourd et compilation + longue
- en **réalité** : c'est le compilateur qui décide !
  - pratique pour petites méthodes appelées souvent (accesseurs ...)

# Instanciation

Dans un **autre** fichier .cpp :

```
#include "Circle.h"

int main() {
    Circle * c = new Circle(0, 0, 50);
    ....
}
```



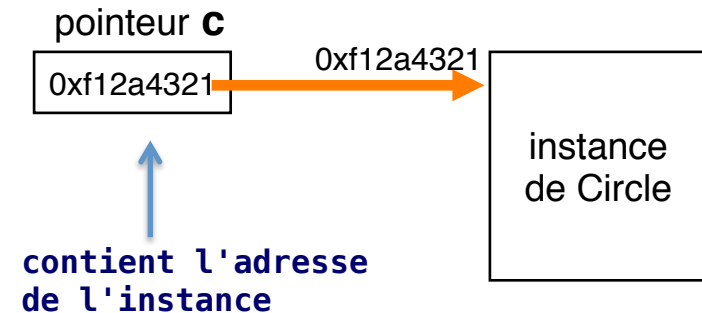


# Instanciación

Dans un **autre** fichier .cpp :

```
#include "Circle.h"

int main() {
    Circle * c = new Circle(0, 0, 50);
    ....
}
```



**new** crée un objet (= une nouvelle instance de la classe)

- 1) alloue la mémoire
- 2) appelle le **constructeur** (et ceux des superclasses !)

**c** est une variable locale qui pointe sur cet objet

- **c** est un **pointeur** (d'où l'**\***) qui contient l'**adresse mémoire** de l'instance

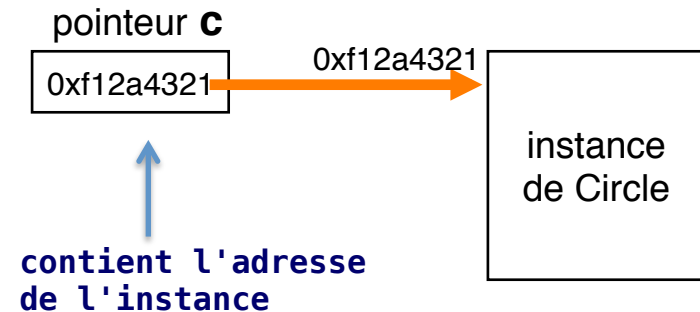
# Pointeurs C/C++ vs. références Java

C++

```
Circle * c = new Circle(0, 0, 50);
```

Java

```
Circle c = new Circle(0, 0, 50);
```



## Pointeur C/C++

- **variable** qui contient une **adresse mémoire**
- valeur **accessible**
- **arithmétique** des pointeurs :
  - calcul d'adresses **bas niveau**
  - à éviter si possible, source d'erreurs et de vulnérabilités

## Référence Java

- **pareil**
- valeur **cachée**
- **pas** d'arithmétique

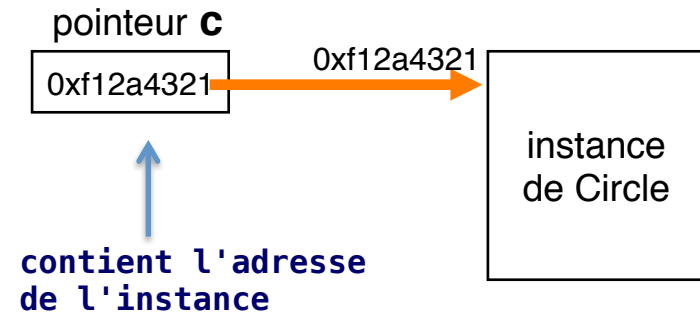
# Pointeurs C/C++ vs. références Java

C++

```
Circle * c = new Circle(0, 0, 50);
```

Java

```
Circle c = new Circle(0, 0, 50);
```



## Pointeur C/C++

- **variable** qui contient une **adresse mémoire**

## Référence Java

- **pareil**

**LES REFERENCES JAVA  
SONT SIMILAIRES AUX POINTEURS**

# Accès aux variables et méthodes d'instance

```
void foo() {  
    Circle * c = new Circle(0, 0, 50);  
    c->radius = 100;  
    unsigned int area = c->getArea();  
}
```

```
class Circle {  
private:  
    int x, y;  
    unsigned int radius;  
public:  
    Circle(int x, int y, unsigned int radius);  
    void setRadius(unsigned int);  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
};
```

## L'opérateur -> dérèfère le pointeur

- comme en C
- mais **■** en Java

## Les méthodes d'instance

- ont **automatiquement accès** aux **variables d'instance**
- sont toujours **appliquées à un objet**

***Problème ?***

# Encapsulation

```
void foo() {  
    Circle * c = new Circle(0, 0, 50);  
    c->radius = 100;           // interdit!  
    unsigned int area = c->getArea();  
}
```

```
class Circle {  
private:  
    int x, y;  
    unsigned int radius;  
public:  
    Circle(int x, int y, unsigned int radius);  
    void setRadius(unsigned int);  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
};
```

## Problème

- radius est **private** => **c** n'a pas le **droit** d'y accéder

# Encapsulation

```
void foo() {  
    Circle * c = new Circle(0, 0, 50);  
    c->radius = 100;           // interdit!  
    unsigned int area = c->getArea();  
}
```

```
class Circle {  
private:  
    int x, y;  
    unsigned int radius;  
public:  
    Circle(int x, int y, unsigned int radius);  
    void setRadius(unsigned int);  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
};
```

## Encapsulation

- séparer la **spécification** de l'**implémentation** (concept de "boîte noire")
- **spécification** : déclaration des méthodes
  - interface avec l'extérieur (API) => on ne peut interagir que **via les méthodes**
- **implémentation** : variables et définition des méthodes
  - interne à l'objet => **seul l'objet** peut accéder à ses **variables**

# Encapsulation

## Abstraire

- exhiber les **concepts**
- **cacher les détails** d'implémentation

## Modulariser

## Encapsulation

- séparer la **spécification** de l'**implémentation** (concept de "boîte noire")
- **spécification** : **déclaration des méthodes**
  - interface avec l'extérieur (API) => on ne peut interagir que **via les méthodes**
- **implémentation** : **variables et définition des méthodes**
  - interne à l'objet => **seul l'objet** peut accéder à ses **variables**

# Encapsulation

## Protéger l'intégrité de l'objet

- ne peut pas être modifié à son insu => peut assurer la **validité** de ses données
- il est **le mieux placé** pour le faire !

## Modulariser

- limiter les **dépendances** entre composants logiciels
- pouvoir **changer l'implémentation** d'un objet sans modifier les autres

## Encapsulation

- séparer la **spécification** de l'**implémentation** (concept de "**boîte noire**")
- **spécification : déclaration des méthodes**
  - interface avec l'extérieur (API) => on ne peut interagir que **via les méthodes**
- **implémentation : variables et définition des méthodes**
  - interne à l'objet => **seul l'objet** peut accéder à ses **variables**



# Encapsulation : droits d'accès

## Droits d'accès C++

- **private** : pour les objets de **cette classe** (par **défaut**)
- **protected** : également pour les **sous-classes**
- **public** : pour **tout le monde**
- **friend** : pour **certaines classes** ou **certaines fonctions**

```
class Circle {  
    friend class ShapeManager;  
    friend bool isInside(const Circle&, int x, int y);  
    ...  
};
```

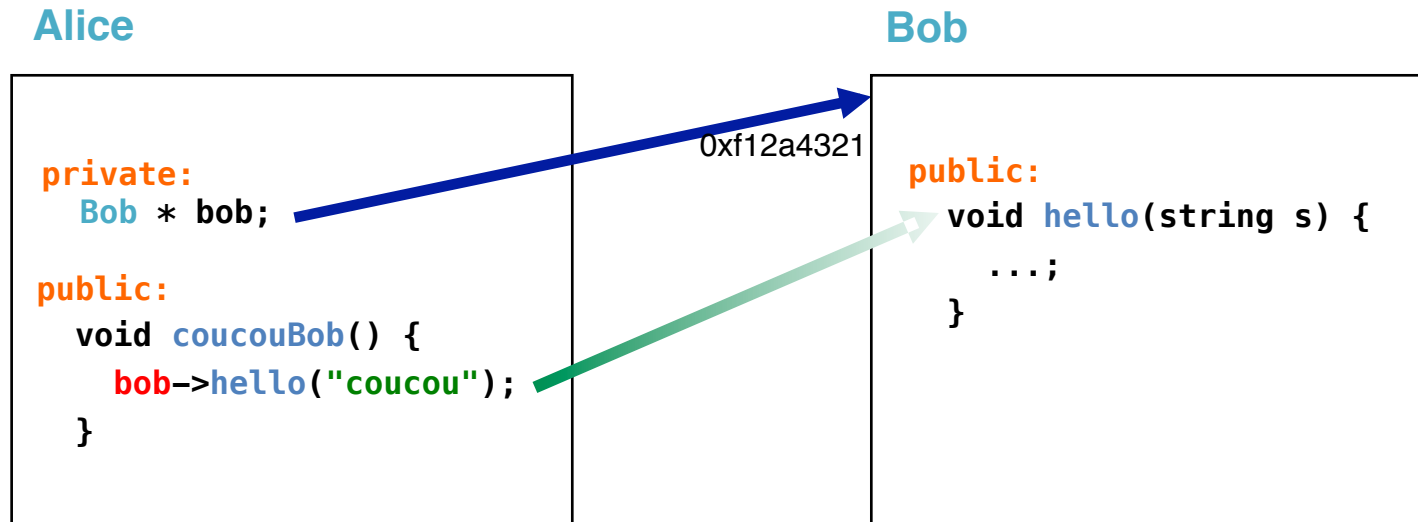
cette classe a  
droit d'accès

cette fonction a  
droit d'accès

## Droits d'accès Java

- **private, protected, public**
- **rien (défaut)** = **toutes** les classes du package

# Accès vs. droits d'accès



**Pour "envoyer un message" à un objet il faut avoir :**

- 1) son **adresse** (via un **pointeur** ou une **référence**)
- 2) le **droit** d'appeler la méthode souhaitée (via **public**, **friend**, etc.)

Il ne suffit pas d'avoir la **clé** de la porte encore faut-il savoir **où** elle se trouve !

# Retour sur les constructeurs

```
class Circle {  
private:  
    int x, y;  
    unsigned int radius;  
public:  
    Circle(int x, int y, unsigned int radius);  
    ....  
};
```

## Trois formes

```
Circle(int _x, int _y) {  
    x = _x; y = _y; radius = 0;  
}
```

←..... comme Java

```
Circle(int x, int y) : x(x), y(y), radius(0) {}
```

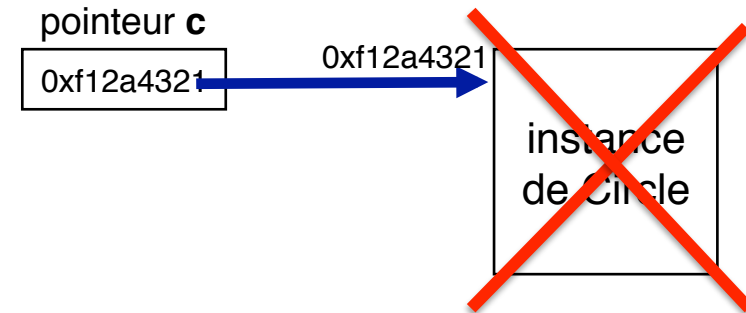
←..... C++: vérifie l'ordre  
x(x) est OK

```
Circle(int x, int y) : x{x}, y{y}, radius{0} {}
```

←..... C++11: compatible avec  
tableaux et conteneurs

# Destruction des objets

```
void foo() {  
    Circle * c = new Circle(100, 200, 35);  
    ...  
    delete c;  
}
```



**delete détruit le pointé (pas le pointeur !)**

- 1) appelle le **destructeur** (s'il y en a un)
- 2) libère la mémoire

**Pas de ramasse miettes en C/C++ !**

- sans **delete** l'objet continue d'exister jusqu'à la fin du programme
- une solution : **smart pointers** (à suivre)

# Destructeur/finaliseur

```
class Circle {  
public:  
    ~Circle() {cerr << "adieu monde cruel\n";}  
    ...  
};  
  
void foo() {  
    Circle * c = new Circle(100, 200, 35);  
    ...  
    delete c;  
}
```

← destructeur

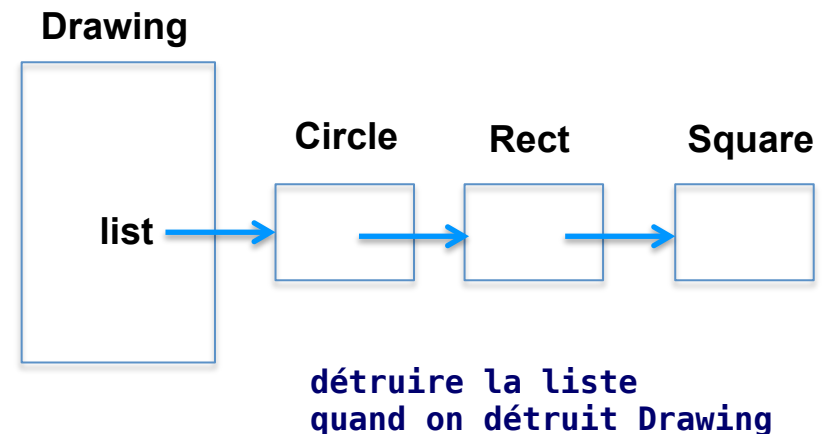
Methode appelée **automatiquement AVANT** que l'objet soit détruit

le **destructeur** ne détruit **PAS** l'objet !  
c'est **delete** qui le fait

# Destructeur/finaliseur

## Quand faut-il un destructeur ?

- **en général il n'y en a pas** : objet détruit qu'il ait ou non un destructeur
- cas particuliers
  - classes de base **polymorphes**  
(voir plus loin)
  - pour **faire le ménage** :
    - détruire **d'autres** objets créés par l'objet
    - fermer fichiers/sockets ouverts par l'objet



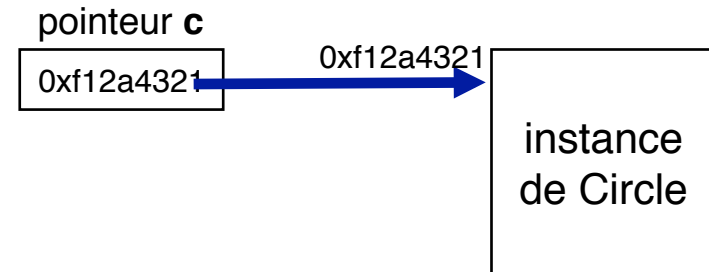
## Il y a des destructeurs en Java !

- méthode **finalise()**  
peu utile car appel **non déterministe** (et le ramasse miettes détruit les **autres** objets)

# Pointeurs nuls et pendants

```
void bar() {  
    Circle * c = new Circle(10, 20, 30);  
    foo(c);  
    delete c;  
  
    foo(c);  
    delete c;  
}
```

```
void foo(Circle * c) {  
    unsigned int area = 0;  
    if (c) area = c->getArea();  
    else perror("Null pointer");  
}
```

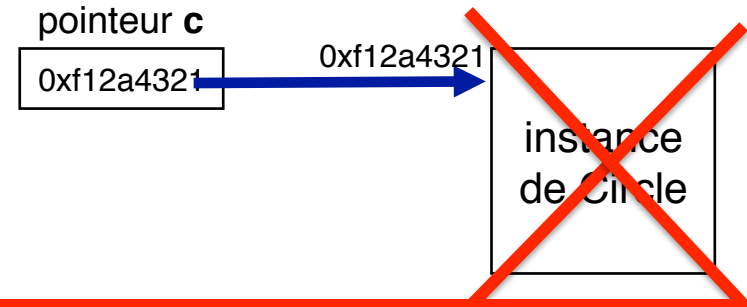


***Correct ?***

# Pointeurs nuls et pendants

```
void bar() {  
    Circle * c = new Circle(10, 20, 30);  
    foo(c);  
    delete c;  
  
    foo(c);  
    delete c;  
}
```

```
void foo(Circle * c) {  
    unsigned int area = 0;  
    if (c) area = c->getArea();  
    else perror("Null pointer");  
}
```



c est pendant :  
pointe sur un objet qui **n'existe plus**  
**!! DANGER !!**

le pointé n'existe plus

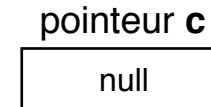
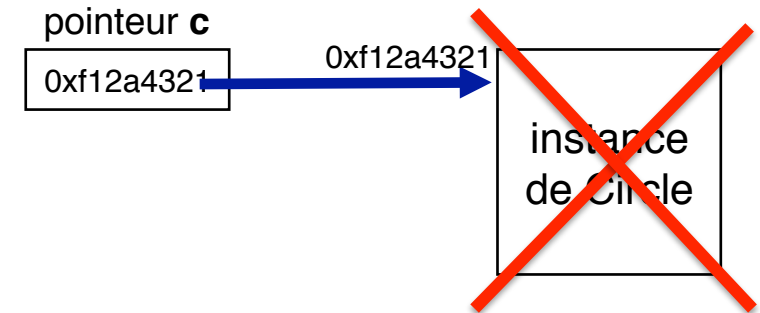
**!! CRASH !!**  
aléatoire



# Pointeurs nuls et pendants

```
void bar() {  
    Circle * c = new Circle(10, 20, 30);  
    foo(c);  
    delete c;  
    c = nullptr;   
  
    foo(c);  
    delete c;  
}
```

```
void foo(Circle * c) {  
    unsigned int area = 0;  
    if (c) area = c->getArea();  
    else perror("Null pointer");  
}
```



**c est nul : pointe sur rien**  
**OK**

# Initialisation des variables

```
Circle * c;  
Circle * c = nullptr;  
Circle * c = new Circle;
```

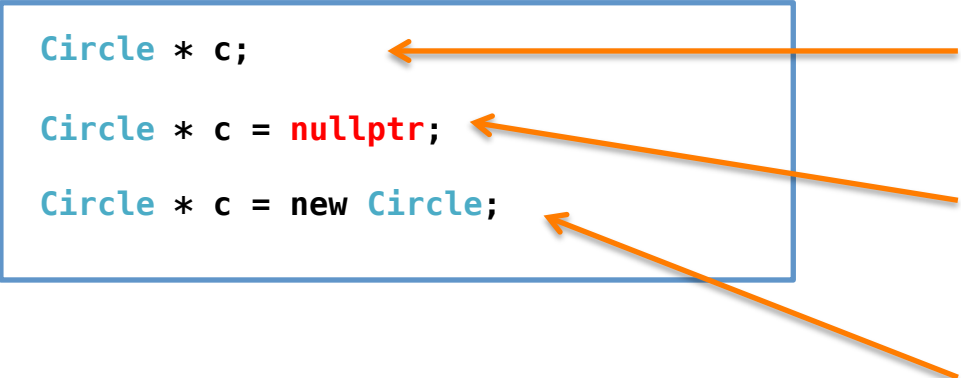
***Correct ?***

```
Circle(int _x, int _y) {  
    x = _x; y = _y;  
}  
  
Circle(int x, int y) : x(x), y(y) {}
```

***Correct ?***

# Initialisation des variables

```
Circle * c;  
Circle * c = nullptr;  
Circle * c = new Circle;
```



c est pendant : **DANGER**

c est nul : **OK**

c est initialisé : **ENCORE MIEUX**

```
Circle(int _x, int _y) {  
    x = _x; y = _y; radius = 0;  
}  
  
Circle(int x, int y) : x(x), y(y), radius(0) {}
```

sinon la valeur de radius est  
**ALEATOIRE !!!**

# Surcharge (overloading)

```
class Circle {  
    Circle();  
    Circle(int x, int y, unsigned int r);  
    void setCenter(int x, int y);  
    void setCenter(Point point);  
};
```

## Fonctions/méthodes d'une même classe

- même nom mais **signatures différentes**
- pareil en **Java**

## Attention

- ne pas confondre avec la **redéfinition de méthodes** (overriding)  
dans une **hiérarchie** de classes

# Paramètres par défaut

```
class Circle {  
    Circle(int x = 0, int y = 0, unsigned int r = 0);  
    ....  
};  
  
Circle * c1 = new Circle(10, 20, 30);  
Circle * c2 = new Circle(10, 20);  
Circle * c3 = new Circle();
```

## Alternative à la surcharge

- n'existe pas en **Java**
- les valeurs par défaut doivent être **à partir de la fin**
- erreur de compilation s'il y a des **ambiguïtés**

```
Circle(int x = 0, int y, unsigned int r = 0);    // ne compile pas !
```

# Variables de classe

```
class Circle {  
    int x, y;  
    unsigned int radius;  
    static int count;  
public:  
    static const int MAX = 10;  
    static constexpr double PI = 3.1415926535;  
    ...  
};
```

variables de classe



## Représentation unique en mémoire

- mot-clé **static** comme en **Java**
- la variable **existe toujours**, même si la classe n'a pas été instanciée

# Variables de classe

```
class Circle {  
    int x, y;  
    unsigned int radius;  
    static int count; ←  
public:  
    static const int MAX = 10;  
    static constexpr double PI = 3.1415926535;  
    ...  
};
```

doit être définie dans un .cpp

## Bizarrerie

- les variables **static** doivent être **définies** dans un .cpp (et seulement un !)
- sauf si le type est **const int** ou **constexpr** (C++11)

```
int Circle::count = 0; ←
```

dans Circle.cpp

# Méthodes de classe

```
class Circle {  
    ...  
    static int count;  
public:  
    static int getCount() {return count;}  
    ...  
};  
  
void foo() {  
    int num = Circle::getCount();  
    ....  
}
```

← variable de classe

← méthode de classe

← appel de la méthode de classe

## Ne s'appliquent pas à un objet

- mot-clé **static** comme en **Java**
- ont accès seulement aux **variables de classe**
- fonctionnent comme les **fonctions du C** mais appel **préfixé** par nom de la classe



# Namespaces

fichier math/Circle.h

```
namespace math {  
    class Circle {  
        ...  
    };  
}
```

fichier graph/Circle.h

```
namespace graph {  
    class Circle {  
        ...  
    };  
}
```

```
#include "math/Circle.h"  
#include "graph/Circle.h"  
  
int main() {  
    math::Circle * mc = new math::Circle();  
    graph::Circle * gc = new graph::Circle();  
}
```

**namespace = espace de nommage**

- évitent les **collisions de noms**
- similaires aux **packages** de **Java**, existent aussi en **C#**

# Namespace par défaut

fichier math/Circle.h

```
namespace math {  
    class Circle {  
        ...  
    };  
}
```

fichier graph/Circle.h

```
namespace graph {  
    class Circle {  
        ...  
    };  
}
```

```
#include "math/Circle.h"  
#include "graph/Circle.h"
```

```
using namespace math;
```

← math accessible par défaut

```
int main() {
```

```
    Circle * mc = new Circle();
```

← équivaut à math::Circle

```
    graph::Circle * gc = new graph::Circle();
```

```
}
```

## using namespace

- modifie la **portée** : symboles de ce **namespace** directement accessibles
- similaire à **import** en Java
- **éviter** d'en mettre dans les **headers**

# Entrées / sorties standard

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

int main() {
    Circle * c = new Circle(100, 200, 35);
    std::cout
        << "radius: " << c->getRadius() << '\n'
        << "area: " << c->getArea()
        << std::endl;
}
```

← flux d'entrées/sorties

← passe à la ligne  
et vide le buffer

## Flux standards

**std** = namespace de la **bibliothèque standard**

**std::cout** **console out** = **sortie** standard

**std::cerr** **console erreurs** = **sortie** des **erreurs** (affichage immédiat)

**std::cin** **console in** = **entrée** standard

# Fichiers

```
#include "Circle.h"
#include <iostream>
#include <fstream>

void foo(std::ostream & s, Circle * c) {
    s << c->getRadius() << ' ' << c->getArea() << std::endl;
}

void foo() {
    Circle * c = new Circle(100, 200, 35);
    foo(std::cout, c);
    std::ofstream file("log.txt");
    if (file) foo(file, c);
}
```

fichiers

ostream = flux de sortie  
noter le & (à suivre)

on peut écrire  
sur la console  
ou dans un fichier

## Flux génériques

**ostream** output stream

**istream** input stream

## Flux pour fichiers

**ofstream** output file stream

**ifstream** input file stream

# Buffers

```
#include "Circle.h"
#include <iostream>
#include <sstream>

void foo(std::ostream & s, Circle * c) {
    s << c->getRadius() << ' ' << c->getArea() << std::endl;
}

void foo() {
    Circle * c = new Circle(100, 200, 35);
    std::stringstream ss;
    foo(ss, c);

    unsigned int r = 0, a = 0;
    ss >> r >> a;
    cout << ss.str() << endl;
}
```

← buffers de texte

← écrit dans un buffer

← lit depuis un buffer

## Flux pour buffers

**stringstream** input/output buffer

**istringstream** input buffer

**ostringstream** output buffer

# Retour sur les méthodes d'instance : où est la magie ?

## Toujours appliquées à un objet :

```
void foo() {  
    Circle * c = new Circle(100, 200, 35);  
    unsigned int r = c->getRadius();  
    unsigned int a = getArea(); // problème !!!  
}
```

## Mais pas la pourquoi ?

```
unsigned int getArea() const {  
    return PI * getRadius() * getRadius();  
}
```

## Comment la méthode accède à radius ?

```
unsigned int getRadius() const {  
    return radius;  
}
```

```
class Circle {  
private:  
    int x, y;  
    unsigned int radius;  
public:  
    Circle(int x, int y, unsigned int radius);  
    void setRadius(unsigned int);  
    unsigned int getRadius() const;  
    unsigned int getArea() const;  
    ....  
};
```

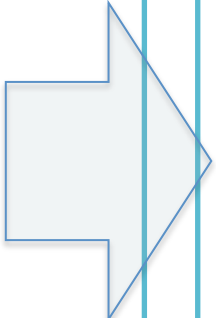
# Le *this* des méthodes d'instance

Le compilateur fait la transformation :

```
unsigned int a = c->getRadius();

unsigned int getRadius() const {
    return radius;
}

unsigned int getArea() const {
    return PI * getRadius() * getRadius();
}
```



```
unsigned int a = getRadius(c);

unsigned int getRadius(Circle * this) const {
    return this->radius;
}

unsigned int getArea(Circle * this) const {
    return PI * getRadius(this) * getRadius(this);
}
```

Le paramètre caché *this* permet :

- d'accéder aux variables d'instance
- d'appeler les autres méthodes d'instance sans avoir à indiquer l'objet

# Documentation

```
/** @brief modélise un cercle.  
 *   Un cercle n'est pas un carré ni un triangle.  
 */  
class Circle {  
    /// retourne la largeur.  
    unsigned int getWidth() const;  
  
    unsigned int getHeight() const;    ///< retourne la hauteur.  
  
    void setPos(int x, int y);  
    /**< change la position: @see setX(), setY().  
    */  
    ...  
};
```

## Doxygen : documentation automatique

- similaire à **JavaDoc** mais plus général (fonctionne avec de nombreux langages)
- documentation : [www.doxygen.org](http://www.doxygen.org)

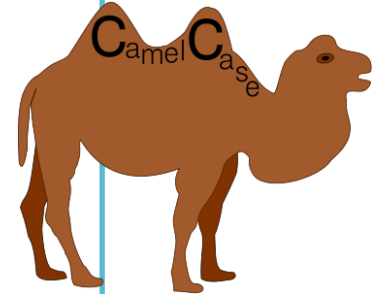


# Style et commentaires

```
/** @brief modélise un cercle.
 *  Un cercle n'est pas un carré ni un triangle.
 */
class Circle {
    /// retourne la largeur.
    unsigned int getWidth() const;

    unsigned int getHeight() const;    ///< retourne la hauteur.

    void setPos(int x, int y);
    /**< change la position: @see setX(), setY().
     */
    ...
};
```



## Règles

- être **cohérent**
- **indenter** (utiliser un IDE qui le fait **automatiquement** : **TAB** ou **Ctrl-I** en général)
- **aérer** et **passer à la ligne** (éviter plus de 80 colonnes)
- **camelCase** et mettre le **nom des variables** (pour la doc)
- **commenter** quand c'est utile

# Chapitre 2 : Héritage et polymorphisme

# Héritage

## 2<sup>e</sup> Concept fondamental de l'OO

- les sous-classes **héritent** les **méthodes** et **variables** de leurs super-classes :
  - la classe **B** a une méthode **foo()** et une variable **x**
- **héritage simple**
  - une classe ne peut hériter que d'une superclasse
- **héritage multiple**
  - une classe peut hériter de **plusieurs** classes
  - C++, Python, Eiffel, Java 8 ...
- **entre les deux**
  - héritage multiple des **interfaces**
  - Java, C#, Objective C ...

### Classe A

```
class A {  
    int x;  
    void foo(int);  
};
```



### Classe B

```
class B : public A {  
    int y;  
    void bar(int);  
};
```

# Règles d'héritage

## Constructeurs / destructeurs

- pas hérités (mais **chaînés** !)

## Méthodes

- héritées
- peuvent être **redéfinies** (overriding)
  - la nouvelle méthode **remplace** celle de la superclasse

**: public** : héritage (comme **extends** de Java)

**virtual** : définition de plus haut niveau

**override** : redéfinition (C++11)

**final** : ne peut être redéfinie (C++11)

### Classe A

```
class A {  
    int x;  
    virtual void foo(int);  
};
```

### Classe B

```
class B : public A {  
    int x;  
    int y;  
    void foo(int) override;  
    void bar(int);  
};
```

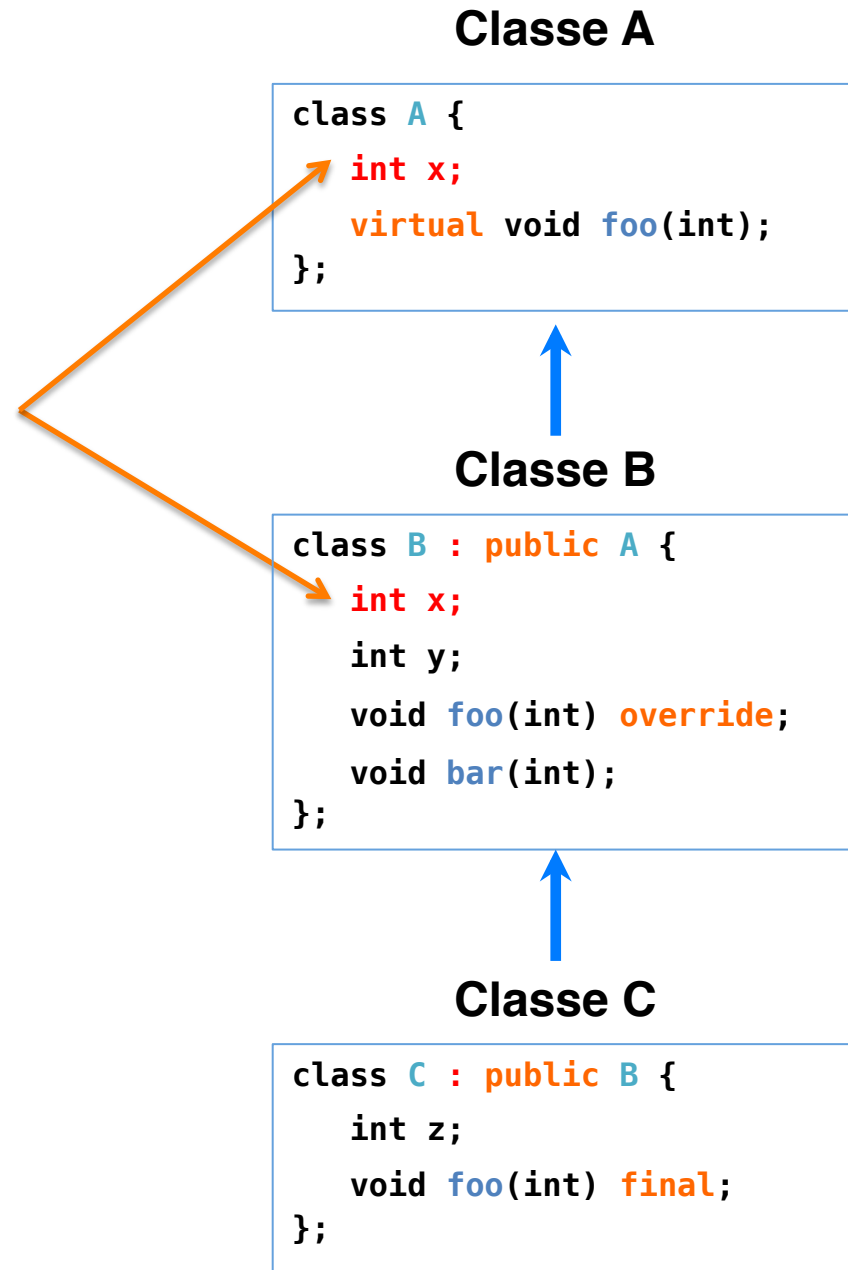
### Classe C

```
class C : public B {  
    int z;  
    void foo(int) final;  
};
```

# Règles d'héritage

## Variables

- héritées
- peuvent être **surajoutées** (shadowing)
- **attention** : la nouvelle variable **cache** celle de la superclasse :
  - **B** a deux variables **x** : **x** et **A::x**
- à éviter !



# Exemple

```
class Rect {
protected:
    int x, y;
    unsigned int width, height;
public:
    Rect();
    Rect(int x, int y, unsigned int w, unsigned int h);

    unsigned int getWidth() const;
    unsigned int getHeight() const;
    virtual void setWidth(unsigned int w);
    virtual void setHeight(unsigned int h);
    //...etc...
};
```

```
class Square : public Rect {
public:
    Square();
    Square(int x, int y, unsigned int size);

    void setWidth(unsigned int w) override;
    void setHeight(unsigned int h) override;
};
```

Rect

```
class Rect {
    ...
};
```



Square

```
class Square
    ...
};
```

Dérivation de classe:  
=> comme **extends** de Java

Redéfinition de méthode  
=> **override** (C++11)

*Pourquoi faut-il  
redéfinir ces  
deux méthodes ?*

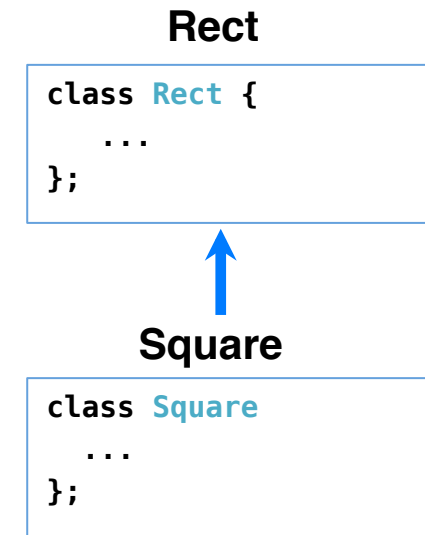
# Exemple

```
class Rect {
protected:
    int x, y;
    unsigned int width, height;
public:
    Rect();
    Rect(int x, int y, unsigned int w, unsigned int h);

    unsigned int getWidth() const;
    unsigned int getHeight() const;
    virtual void setWidth(unsigned int w) {width = w;}
    virtual void setHeight(unsigned int h) {height = h;}
    //...etc...
};

class Square : public Rect {
public:
    Square();
    Square(int x, int y, unsigned int size);

    void setWidth(unsigned int w) override {height = width = w;}
    void setHeight(unsigned int h) override {width = height = h;}
};
```



← sinon ce n'est plus un carré !

# Chaînage des constructeurs

```
class Rect {  
protected:  
    int x, y;  
    unsigned int width, height;  
public:  
    Rect() : x(0), y(0), width(0), height(0) {}  
    Rect(int x, int y, unsigned int w, unsigned int h) : x(x), y(y), width(w), height(h) {}  
    //...etc...  
};
```

```
class Square : public Rect {  
public:  
    Square() {}  
    Square(int x, int y, unsigned int size) : Rect(x, y, size, size) {}  
    //...etc...  
};
```

Chaînage implicite des constructeurs

Chaînage explicite des constructeurs

=> comme `super()` en Java

**Attention à la syntaxe :**

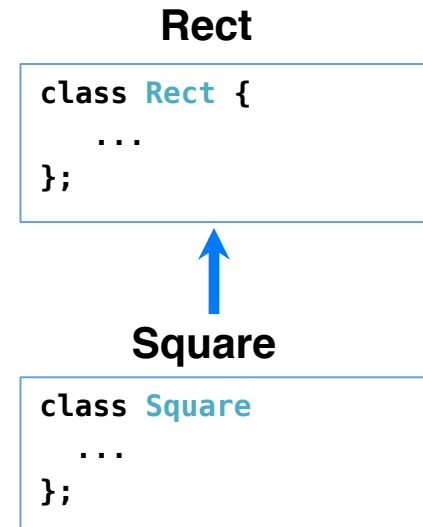
`Square(int x, int y, unsigned int size) { Rect(x, y, size, size); }` FAUX !!!

`Square() {}` identique à: `Square() : Rect() {}`



# Covariance des types de retour

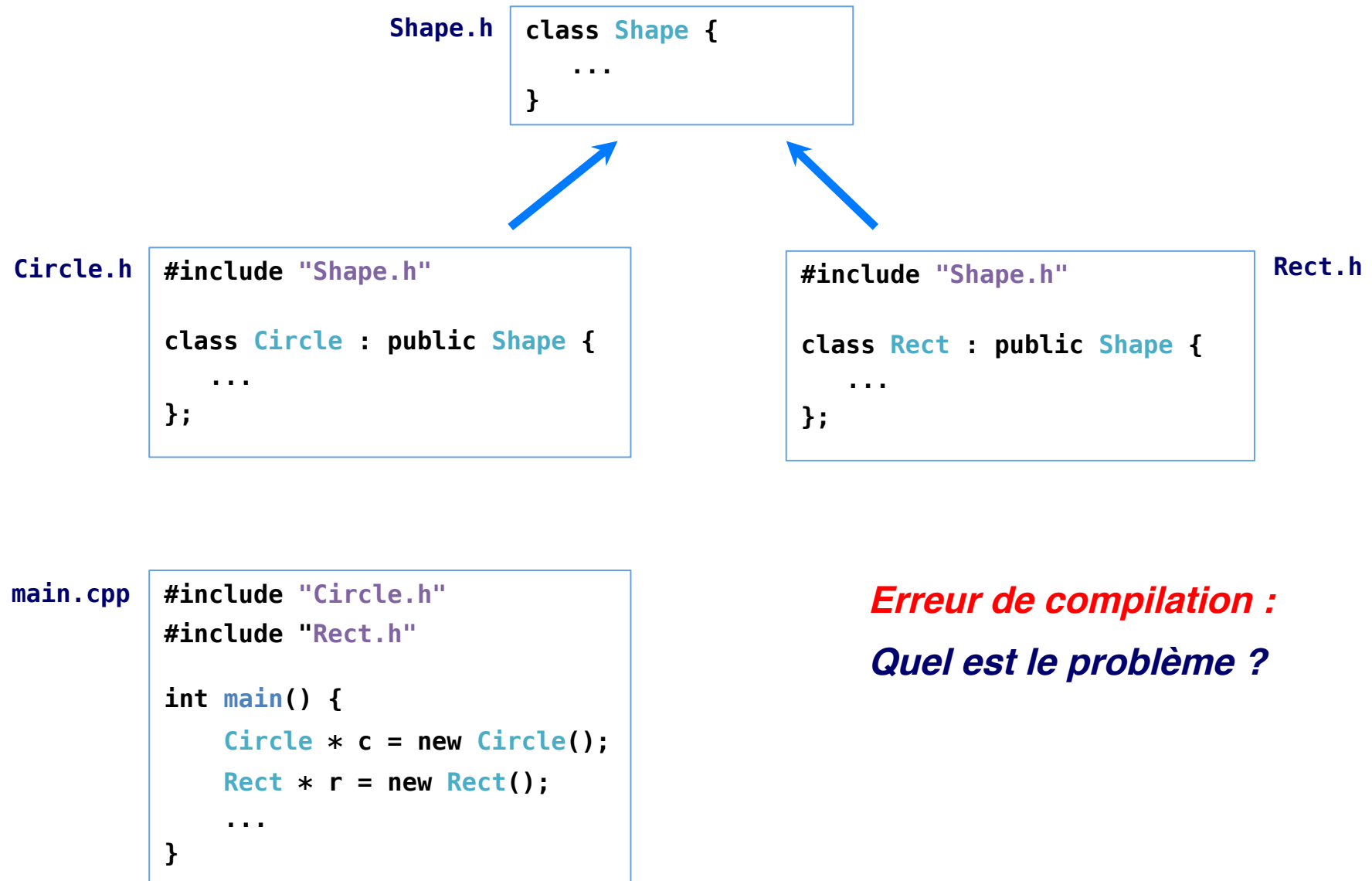
```
class RectCreator {  
    virtual Rect* makeShape() {return new Rect;}  
}  
  
class SquareCreator : public RectCreator {  
    Square* makeShape() override {return new Square;}  
}
```



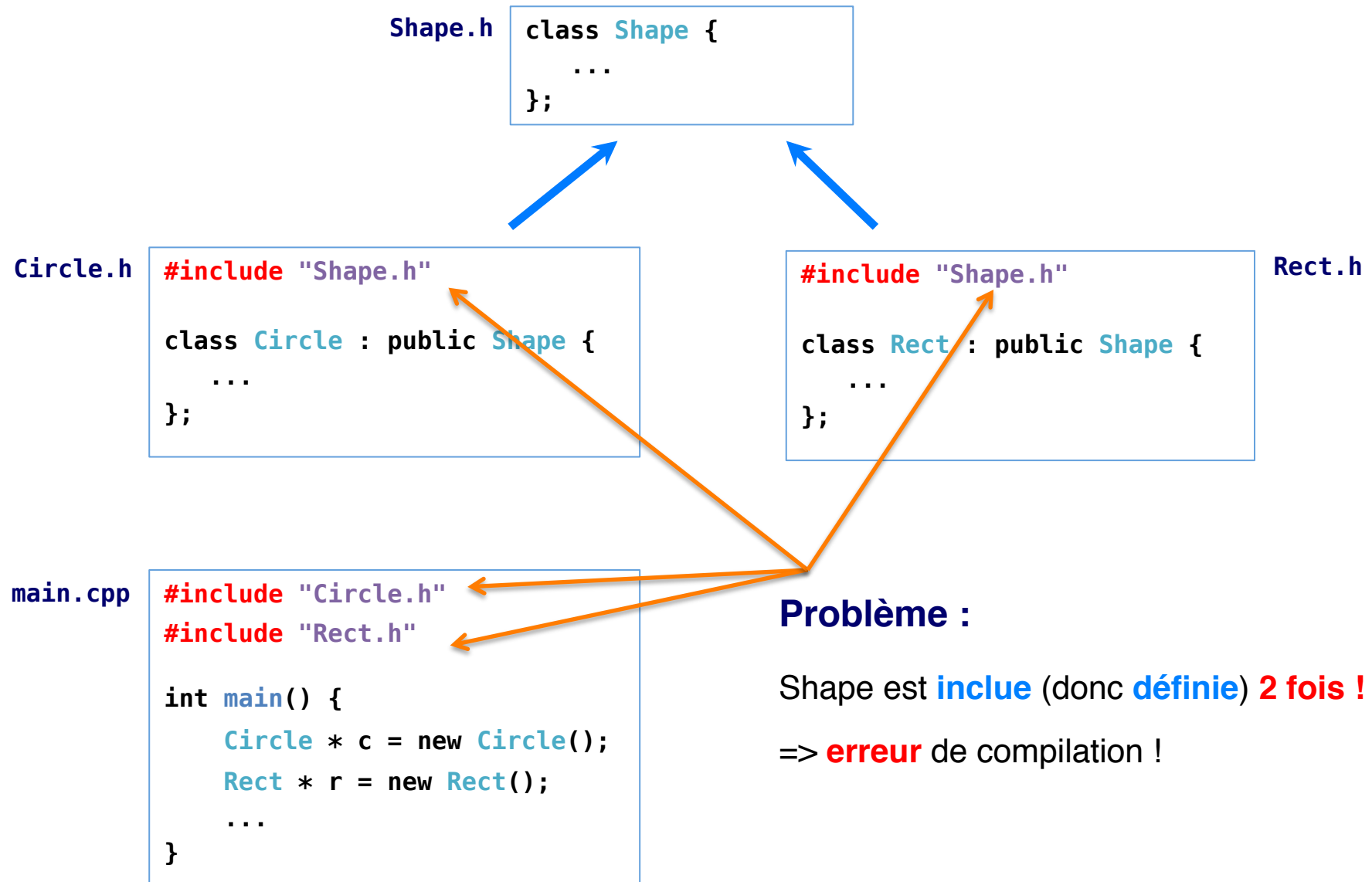
## Le type de retour n'est pas le même !

- **correct** car `Square` dérive de `Rect`
- mais les **paramètres** doivent être **identiques**, sinon c'est de la **surcharge** !

# Inclusion des headers



# Inclusion des headers



# Solution

Shape.h

```
#ifndef Graph_Shape
#define Graph_Shape

class Shape {
    ...
};

#endif
```

empêche les  
inclusions multiples

Circle.h

```
#ifndef Graph_Circle
#define Graph_Circle

#include "Shape.h"

class Circle : public Shape {
    ...
};

#endif
```

Rect.h

```
#ifndef Graph_Rect
#define Graph_Rect

#include "Shape.h"

class Rect : public Shape {
    ...
};

#endif
```

main.cpp

```
#include "Circle.h"
#include "Rect.h"

int main() {
    Circle * c = new Circle();
    Rect * r = new Rect();
    ...
}
```

## Solution :

**#ifndef** évite les inclusions multiples

# Directives du préprocesseur

Header  
Truc.h

```
#ifndef Truc
#define Truc

class Truc {
    ...
};

#endif
```

inclut ce qui suit jusqu'à **#endif**  
seulement si **Truc** n'est pas déjà défini

définit **Truc**, qui doit être **unique**  
=> à forger sur nom du header

## Directives de compilation

- **#if** / **#ifdef** / **#ifndef** pour **compilation conditionnelle**
- **#import** (au lieu de **#include**) empêche l'inclusion multiple (pas standard !)

## Recherche des headers

- **#include "Circle.h"** cherche dans **répertoire courant**
- **#include <iostream>** cherche dans **répertoires systèmes** (/usr/include, etc.)  
et dans ceux spécifiés par **option -I** du compilateur :

```
gcc -Wall -I/usr/X11R6/include -o myprog Circle.cpp main.cpp
```

# Polymorphisme d'héritage

## 3<sup>e</sup> concept fondamental de l'orienté objet

- le plus **puissant** mais pas toujours le mieux **compris** !

## Un objet peut être vu sous plusieurs formes

- un **Square** est aussi un **Rect**
- mais **l'inverse** n'est pas vrai !

```
#include "Rect.h"
```

```
void foo() {
```

```
    Square * s = new Square();    // s voit l'objet comme un Square
    Rect * r = s;                  // r voit objet comme un Rect (upcasting implicite)
    ...
    Square * s2 = new Rect();      // OK ?
    Square * s3 = r;               // OK ?
```

```
}
```

**Rect**

```
class Rect {
    ...
};
```



**Square**

```
class Square
    : public Rect {
    ...
};
```

# Buts du polymorphisme

Pouvoir choisir le **point de vue le plus approprié** selon les besoins

Pouvoir traiter un ensemble de classes liées entre elles de **manière uniforme sans considérer leurs détails**

```
#include "Rect.h"
```

```
void foo() {
```

```
    Square * s = new Square();    // s voit l'objet comme un Square
    Rect * r = s;                 // r voit objet comme un Rect (upcasting implicite)
    ...
    Square * s2 = new Rect();     // erreur de compilation! (downcasting interdit !)
    Square * s3 = r;              // erreur de compilation!
```

```
}
```

Rect

```
class Rect {
    ...
};
```



Square

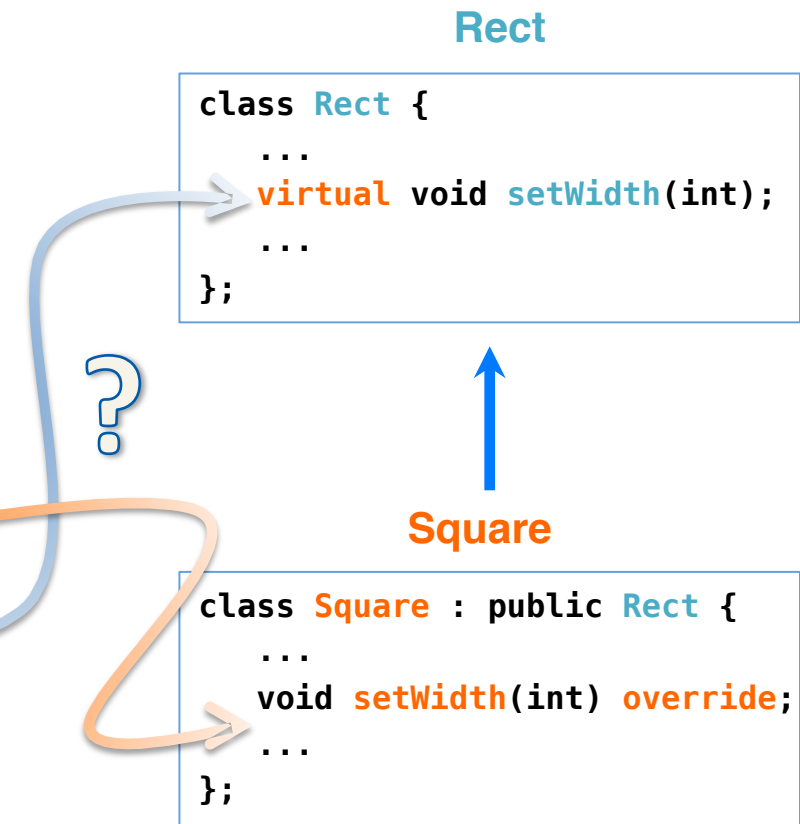
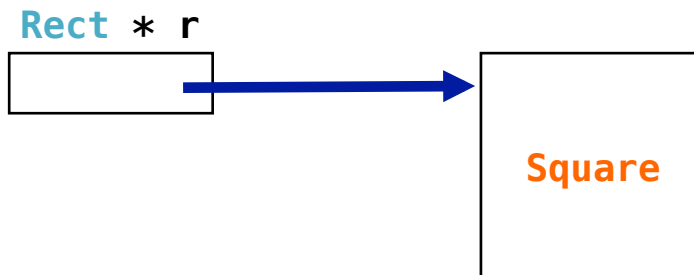
```
class Square
    : public Rect {
    ...
};
```

# Polymorphisme

## Question à \$1000

- quelle méthode `setWidth()` est appelée : celle du **pointeur** ou celle du **pointé** ?
- avec **Java** ?
- avec **C++** ?

```
Rect * r = new Square();  
r->setWidth(100);
```





# Polymorphisme : Java

## Question à \$1000

- quelle méthode `setWidth()` est appelée : celle du **pointeur** ou celle du **pointé** ?

```
Rect * r = new Square();  
r->setWidth(100);
```

Rect

```
class Rect {  
    ...  
    virtual void setWidth(int);  
    ...  
};
```



Square

```
class Square : public Rect {  
    ...  
    void setWidth(int) override;  
    ...  
};
```

## Java

- **liaison dynamique / tardive** : choix de la méthode à l'exécution
- ⇒ appelle toujours la méthode du **pointé**
  - *heureusement sinon le carré deviendrait un rectangle !*

# Polymorphisme : C++

## Question à \$1000

- quelle méthode `setWidth()` est appelée : celle du **pointeur** ou celle du **pointé** ?

```
Rect * r = new Square();  
r->setWidth(100);
```

Rect

```
class Rect {  
    ...  
    virtual void setWidth(int);  
    ...  
};
```

Square

```
class Square : public Rect {  
    ...  
    void setWidth(int) override;  
    ...  
};
```

## C++

- avec **virtual** : **liaison dynamique / tardive** => méthode du **pointé** comme **Java**
- sans **virtual** : **liaison statique** => méthode du **pointeur**  
**=> comportement incohérent dans cet exemple !**

# Règles à suivre

## Classe de base

- ⇒ mettre **virtual**
- ⇒ y compris pour les **destructeurs**

## Redéfinitions

- ⇒ mettre **override** ou **final** (C++11)
- ⇒ vérifie méthode parente **virtual** ou **override**

## Destructeurs virtuels

```
Rect * r = new Square();  
delete r;
```

si `~Shape()` n'est pas **virtual**  
`~Rect()` et `~Square()` ne sont pas appelées !!!

### Shape

```
class Shape {  
    virtual ~Shape();  
    virtual void setWidth(int);  
    ...  
};
```

### Rect

```
class Rect : public Shape {  
    ~Rect();  
    void setWidth(int) override;  
    ...  
};
```

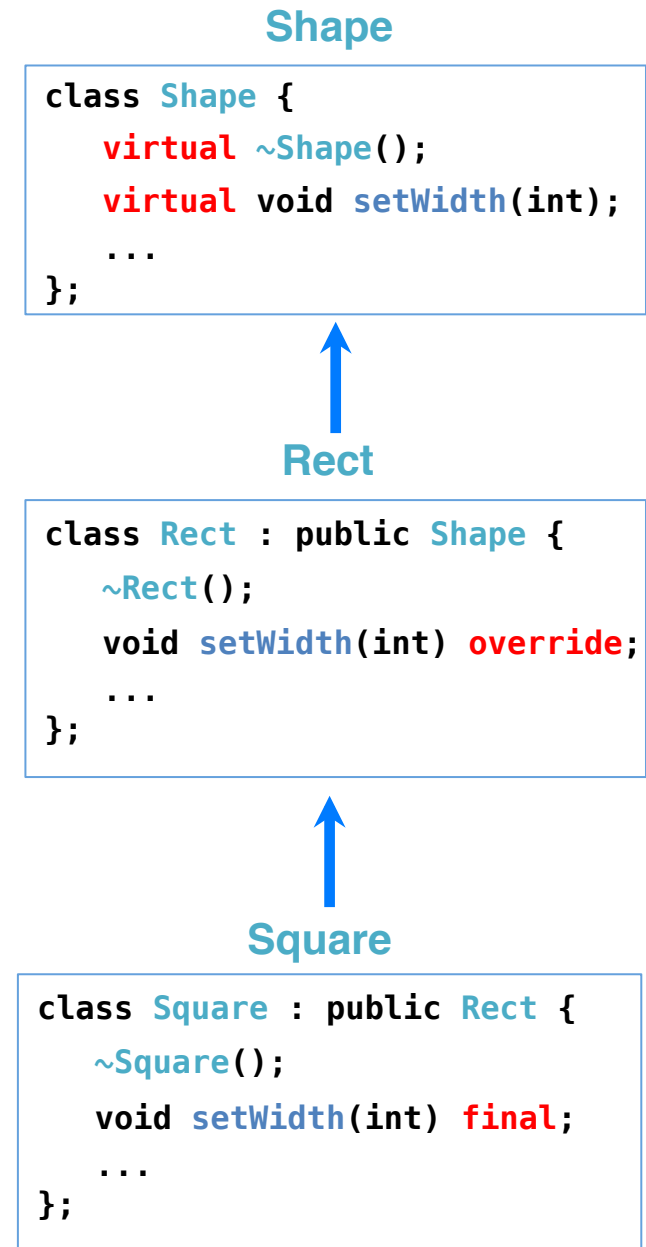
### Square

```
class Square : public Rect {  
    ~Square();  
    void setWidth(int) final;  
    ...  
};
```

# Règles à suivre

## Remarques

- une **redéfinition** de méthode **virtuelle** est automatiquement **virtuelle**
- la fonction doit avoir la **même signature** sinon c'est de la **surcharge** !
- sauf que le **type de retour** peut être une **sous-classe** (**covariance** des types de retour)
- une classe peut être **final**

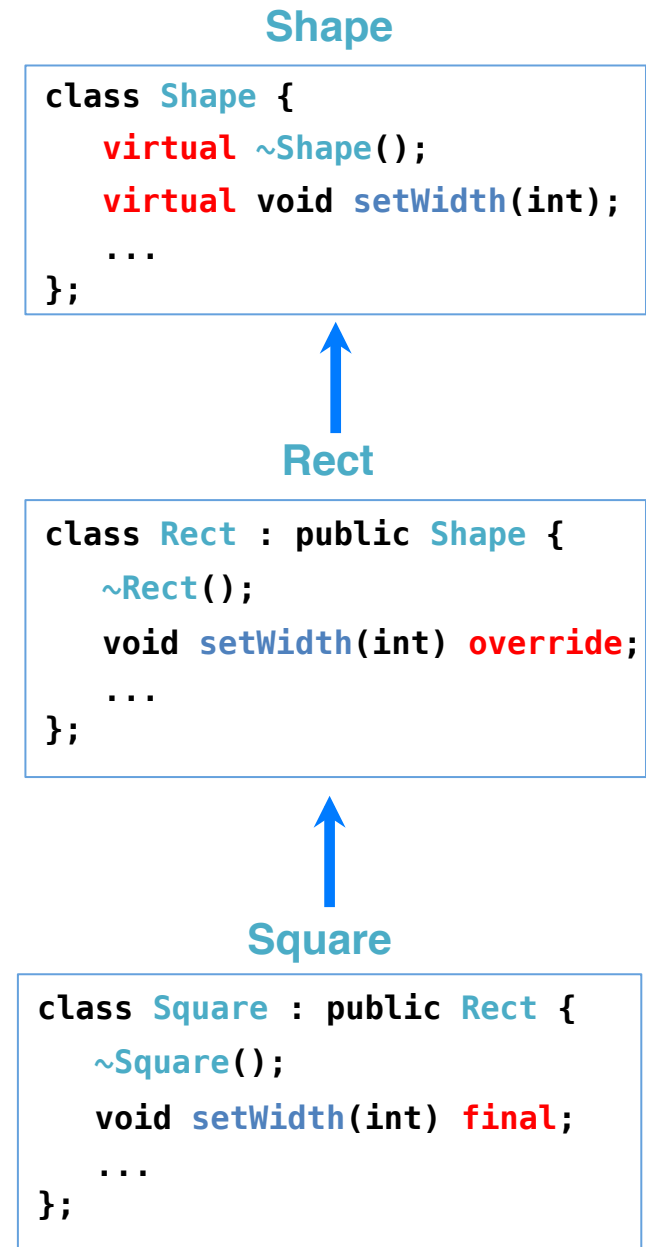


# Méthodes non virtuelles

## Utiles dans quel cas ?

- si **classe** pas héritée
- si **méthode** jamais redéfinie
  - typiquement : getters et setters
- si **méthode** appelée **très très très** souvent :
  - méthodes virtuelles plus lentes
  - sans importance dans la plupart des cas
  - gare aux erreurs !!!

Dans le doute mettre **virtual**  
et optimiser plus tard !



# Méthodes et classes abstraites

```
class Shape {  
public:  
    virtual void setWidth(unsigned int) = 0;    // méthode abstraite  
    ...  
};
```

## Méthode abstraite

- spécification d'un **concept** dont la réalisation diffère selon les sous-classes
  - pas implémentée
  - doit être **redéfinie** et **implémentée** dans les sous-classes **instanciables**

## Classe abstraite

- classe dont **au moins** une méthode est abstraite

## Java

- pareil mais mot clé **abstract**

# Bénéfices des classes abstraites

```
class Shape {  
public:  
    virtual void setWidth(unsigned int) = 0;    // méthode abstraite  
    ...  
};
```

## Méthode abstraite

- spécification d'un **concept** dont la réalisation diffère selon les sous-classes
  - **pas implémentée**
  - doit être **redéfinie et implémentée** dans les sous-classes **instanciables**

Traiter un ensemble de classes liées entre elles :

- **de manière uniforme sans considérer leurs détails**
- **avec un degré d'abstraction plus élevé**

**Imposer une spécification** que les sous-classes doivent implémenter

- sinon erreur de compilation !
- façon de « mettre l'**UML** dans le code »

# Exemple de classe abstraite

```
class Shape {  
    int x, y;  
public:  
    Shape() : x(0), y(0) {}  
    Shape(int x, int y) : x(x), y(y) {}  
  
    int getX() const {return x;}  
    int getY() const {return y;}  
    virtual unsigned int getWidth() const = 0;  
    virtual unsigned int getHeight() const = 0;  
    virtual unsigned int getArea() const = 0;  
    ....  
};
```

← implémentation commune  
à toutes les sous-classes

← méthodes abstraites:  
l'implémentation dépend des  
sous-classes

```
class Circle : public Shape {  
    unsigned int radius;  
public:  
    Circle() : radius(0) {}  
    Circle(int x, int y, unsigned int r) : Shape(x, y), radius(r) {}  
  
    unsigned int getRadius() const {return radius;}  
    virtual unsigned int getWidth() const {return 2 * radius;}  
    virtual unsigned int getHeight() const {return 2 * radius;}  
    virtual unsigned int getArea() const {return PI * radius * radius;}  
    ....  
}
```

doivent être implémentées  
dans les sous-classes



# Interfaces

```
class Shape {  
public:  
    virtual int getX() const = 0;  
    virtual int getY() const = 0;  
    virtual unsigned int getWidth() const = 0;  
    virtual unsigned int getHeight() const = 0;  
    virtual unsigned int getArea() const = 0;  
};
```

← pas de variables d'instance  
ni de constructeurs

← toutes les méthodes sont  
abstraites

## Classes totalement abstraites (en théorie)

- pure **spécification** : toutes les méthodes sont **abstraites**
- ont un rôle particulier pour l'**héritage multiple** en **Java**, **C#**, etc.
  - **C++** : pas de mot clé, cas particulier de **classe abstraite**
  - **Java** : mot clé **interface**
    - en **Java 8** les interfaces peuvent avoir des implémentations de méthodes !

# Traitements uniformes

```
#include "Rect.h"
#include "Circle.h"

void foo() {
    Shape ** shapes = new Shape * [10];

    unsigned int count = 0;
    shapes[count++] = new Circle(0, 0, 100);
    shapes[count++] = new Rect(10, 10, 35, 40);
    shapes[count++] = new Square(0, 0, 60)

    printShapes(shapes, count);
}
```

← tableau de 10 Shape \*

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

void printShapes(Shape ** tab, unsigned int count) {
    for (unsigned int k = 0; k < count; ++k) {
        cout << "Area = " << tab[k]->getArea() << endl;
    }
}
```

# Tableaux dynamiques

```
int * tab = new int[10];
```

```
...
```

```
delete [] tab;    // ne pas oublier []
```

tableau de 10 int

pointeur tab



chaque élément est un int

```
Shape ** shapes = new Shape * [10];
```

```
...
```

```
delete [] shapes;
```

tableau de 10 Shape \*

pointeur shapes



chaque élément est un Shape \*

# Traitements uniformes (2)

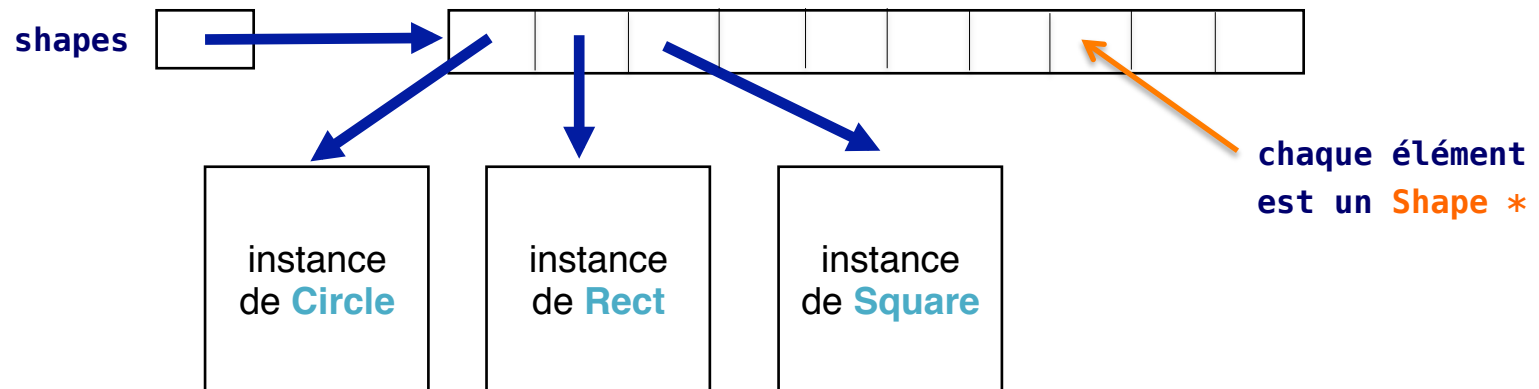
```
#include "Rect.h"
#include "Circle.h"

void foo() {
    Shape ** shapes = new Shape * [10];

    unsigned int count = 0;
    shapes[count++] = new Circle(0, 0, 100);
    shapes[count++] = new Rect(10, 10, 35, 40);
    shapes[count++] = new Square(0, 0, 60)

    printShapes(shapes, count);
}
```

équivalent à:  
shapes[count] = ...;  
count++;



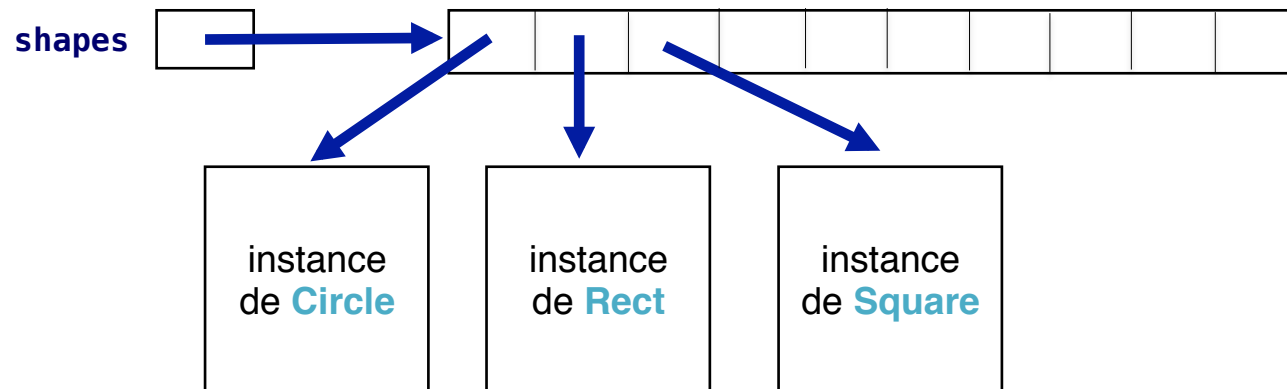
# Magie du polymorphisme

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

void printShapes(Shape ** tab, unsigned int count) {
    for (unsigned int k = 0; k < count; ++k) {
        cout << "Area = " << tab[k]->getArea() << endl;
    }
}
```

taille en paramètre  
pas d'autre moyen  
de la connaître !

C'est toujours la bonne version de `getArea()` qui est appelée !



# Magie du polymorphisme

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

void printShapes(Shape ** tab, unsigned int count) {
    for (unsigned int k = 0; k < count; ++k) {
        cout << "Area = " << tab[k]->getArea() << endl;
    }
}
```

## Remarque

- cette fonction **ignore** l'existence de **Circle**, **Rect**, **Square** !

## Mission accomplie !

- on peut traiter un ensemble de classes liées entre elles de **manière uniforme**  
**sans considérer leurs détails**
- on peut rajouter de **nouvelles classes** **sans modifier l'existant**

# Chaînage des méthodes

## Règle générale : éviter les duplications de code

- à plus ou moins long terme ca **diverge** !
  - ⇒ code difficile à **comprendre**
  - ⇒ difficile à **maintenir**
  - ⇒ probablement **buggé** !

## Solutions

- utiliser l'**héritage** !
- le cas échéant, **chaîner** les méthodes des superclasses

```
class NamedRect : public Rect {  
public:  
    virtual void draw() {           // affiche le rectangle et son nom  
        Rect::draw();               // trace le rectangle  
        // code pour afficher le nom ...  
    }  
};
```

# Concepts fondamentaux de l'orienté objet

## En résumé : 4 fondamentaux

- 1) **méthodes** : lien entre les fonctions et les données
- 2) **encapsulation** : crucial en OO (mais possible avec des langages non OO)
- 3) **héritage** : simple ou multiple
- 4) **polymorphisme d'héritage (dynamique)** : toute la puissance de l'OO !

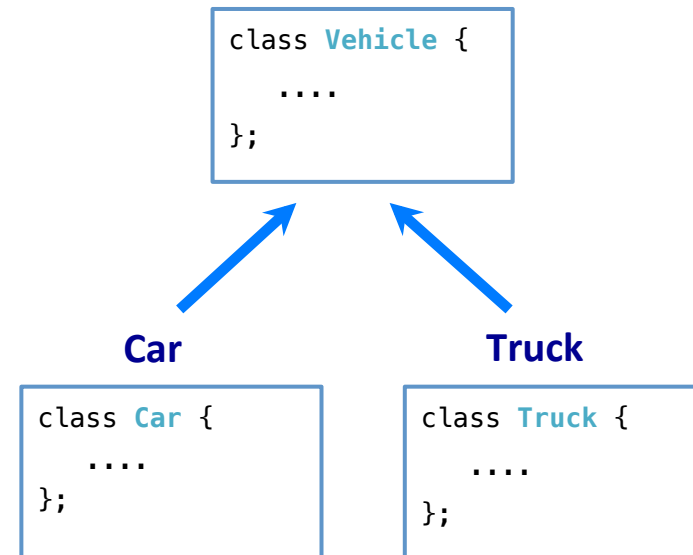


# Implémentation des méthodes virtuelles

```
class Vehicle {
public:
    virtual void start();
    virtual int getColor();
    ...
};

class Car : public Vehicle {
public:
    void start() override;
    virtual void setDoors(int doors);
    ...
};

class Truck : public Vehicle {
public:
    void start() override;
    virtual void setPayload(int payload);
    ...
};
```

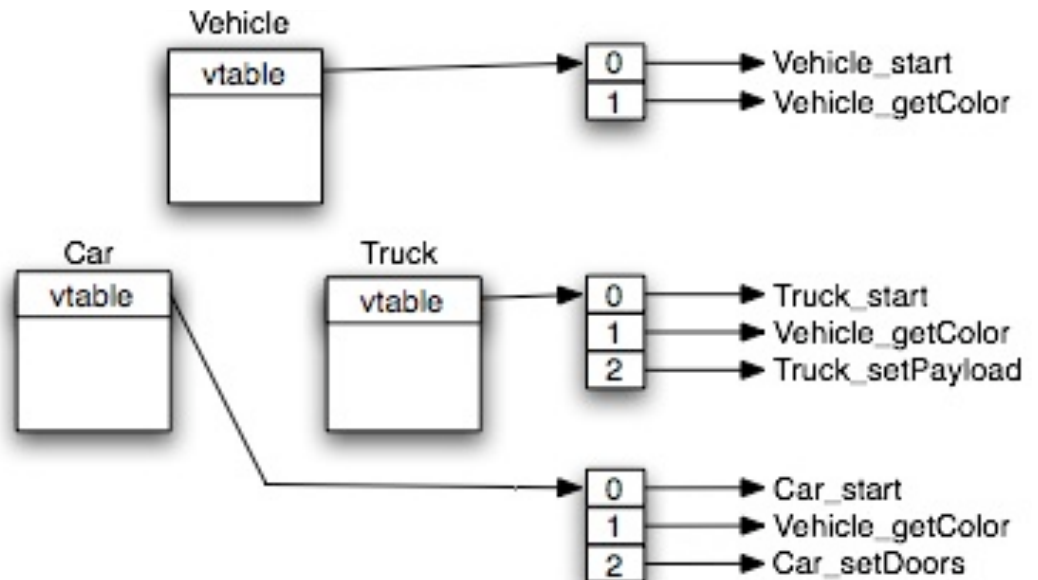


# Implémentation des méthodes virtuelles

```
class Vehicle {
    __VehicleTable * __vtable;
public:
    virtual void start();
    virtual int getColor();
    ...
};

class Car : public Vehicle {
    __CarTable * __vtable;
public:
    void start() override;
    virtual void setDoors(int doors);
    ...
};

class Truck : public Vehicle {
    __TruckTable * __vtable;
public:
    void start() override;
    virtual void setPayload(int payload);
    ...
};
```



## vtable

- chaque **objet** pointe vers la **vtable** de sa **classe**
- **vtable** = tableau de pointeurs de fonctions

```
Vehicle * p = new Car();
```

```
p->start();      == (p->__vtable[0])();
```

```
p->getColor();   == (p->__vtable[1])();
```

# Coût des méthodes virtuelles

```
class Vehicle {
    __VehicleTable * __vtable;
public:
    virtual void start();
    virtual int getColor();
    ...
};

class Car : public Vehicle {
    __CarTable * __vtable;
public:
    void start() override;
    virtual void setDoors(int doors);
    ...
};

class Truck : public Vehicle {
    __TruckTable * __vtable;
public:
    void start() override;
    virtual void setPayload(int payload);
    ...
};
```

## Coût mémoire

un pointeur (`__vtable`) par objet

⇒ méthodes virtuelles **inutiles** si :

- **aucune** sous-classe
- ou **aucune redéfinition** de méthode

## Coût d'exécution

double indirection

- coût **négligeable** sauf si 1 milliard d'appels !

# Implémentation des méthodes virtuelles

```
class Vehicle {
    __VehicleTable * __vtable;
public:
    virtual void start();
    virtual int getColor();
    ...
};

class Car : public Vehicle {
    __CarTable * __vtable;
public:
    void start() override;
    virtual void setDoors(int doors);
    ...
};

class Truck : public Vehicle {
    __TruckTable * __vtable;
public:
    void start() override;
    virtual void setPayload(int payload);
    ...
};
```

0000000100000ff0 t \_\_ZN3Car5startEv

0000000100000f40 t \_\_ZN3CarC1Ei

0000000100000f70 t \_\_ZN3CarC2Ei

00000001000010b0 t \_\_ZN7Vehicle5startEv

0000000100001c70 t \_\_ZN7Vehicle8getColorEv

0000000100000fc0 t \_\_ZN7VehicleC2Ei

0000000100002150 D \_\_ZTI3Car

0000000100002140 D \_\_ZTI7Vehicle

U \_\_ZTVN10\_\_cxxabiv117\_\_class\_type\_infoE

U \_\_ZTVN10\_\_cxxabiv120\_\_si\_class\_type\_infoE

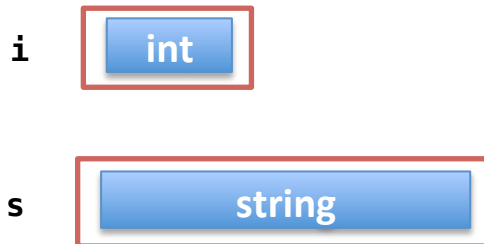
0000000100000ec0 T \_main

# Chapitre 3 : Mémoire

# Allocation mémoire

## Mémoire automatique (pile/stack)

- variables **locales** et **paramètres**
- **créées** à l'**appel** de la fonction  
**détruites** à la **sortie** de la fonction
- la variable **contient** la donnée



```
void foo(bool option) {  
    int i = 0;  
    i += 10;  
    string s = "Hello";  
    s += " World";  
    s.erase(4, 1);  
    ...  
}
```

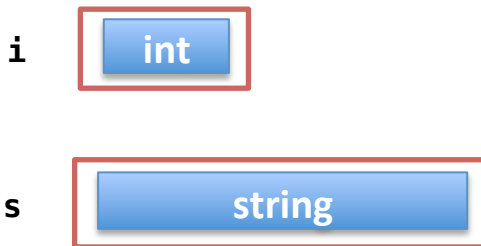
- accède aux champs de l'objet

- possible pour **types de base** et **objets** contrairement à **Java** (que types de base)

# Allocation mémoire

## Mémoire globale/statique

- variables **globales** ou **static** dont **variables de classe**
- existent du **début** à la **fin** du programme
- initialisées **une seule fois**
- la variable **contient** la donnée



```
→ int glob = 0;    // variable globale
→ static int stat = 0;

void foo() {
    → static int i = 0;
      i += 10;
    → static string s = "Hello";
      s += "World";
      s.erase(4, 1);
      ...
}
```

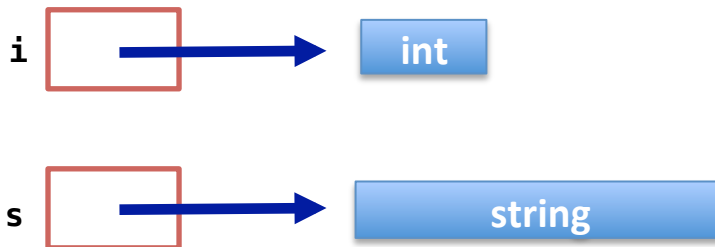
Que valent `i` et `s` si on appelle `foo()` deux fois ?

- les variables **globales** sont **dangereuses** !

# Allocation mémoire

## Mémoire dynamique (tas/heap)

- données **créées** par **new**  
**détruites** par **delete**
- la variable **pointe** sur la donnée



- possible pour **objects** et **types de base**  
contrairement à **Java** (que objets)

```
void foo() {  
    int * i = new int(0); ←  
    *i += 10;  
  
    string * s = new string("Hello"); ←  
    *s += " World";  
    s->erase(4, 1);  
    ...  
    delete i;  
    delete s;  
}
```

**\*s** est le **pointé**

-> accède aux champs de l'objet :

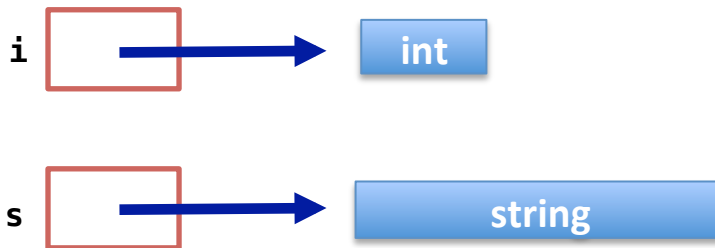
**a->x == (\*a).x**



# Allocation mémoire

## Mémoire dynamique (tas/heap)

- données **créées** par **new**  
**détruites** par **delete**
- la variable **pointe** sur la donnée



```
void foo() {  
    int * i = new int(0);  
    *i += 10;  
  
    string * s = new string("Hello");  
    *s += " World";  
    s->erase(4, 1);  
    ...  
    delete i;  
    delete s;  
}
```

## Penser à détruire les pointés !

- sinon ils existent jusqu'à la **fin du programme**
- **delete** ne détruit pas la variable mais **le pointé** !

# Objets et types de base

## C++

- traite les **objets** **comme** les **types de base**
- idem en **C** avec les **struct**
- les **constructeurs** / **destructeurs** sont appelés **dans tous les cas**

```
int glob = 0;
static int stat = 0;

void foo() {
    static int i = 0;
    int i = 0;
    int * i = new int(0);

    static string s = "Hello";
    string s = "Hello";
    string * s = new string("Hello");
    ...
    delete i;
    delete s;
}
```

C++

# Objets et types de base

## C++


- traite les **objets** comme les **types de base**

## Java

- **ne traite pas** les objets comme les types de base
- **objets toujours** créés avec **new**
- **types de base jamais** créés avec **new**
- **static** que pour **variables de classe**

~~int glob = 0;~~                      **équivalent**  
~~static int stat = 0;~~          **Java**

```
void foo() {  
    static int i = 0;  
    int i = 0;  
    int * i = new int(0);  
  
    static string s = "Hello";  
    string s = "Hello";  
    string * s = new string("Hello");  
    ...  
    delete i;  
    delete s;  
}
```



```
// en Java on écrirait:  
String s = new String("Hello");  
String s = "Hello";
```

# Sous-objets

```
class Car : public Vehicle {  
    int power;  
    Door rightDoor;  
    Door * leftDoor;  
public:  
    Car() :  
        rightDoor(this),  
        leftDoor(new Door(this)) {  
    }  
};
```

```
class Door {  
public:  
    Door(Car *);  
    ....  
};
```

rightDoor

Door

contient l'objet (pas possible en Java)

leftDoor

Door

pointe l'objet (comme Java)

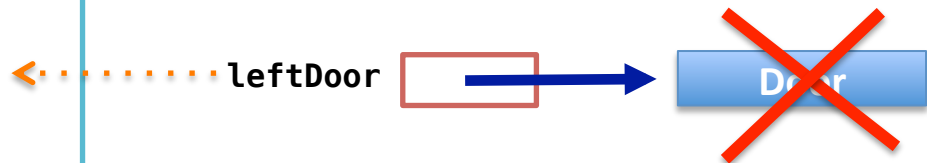
## Variables d'instance **contenant** un objet (rightDoor)

- allouées, créés, détruites **en même temps** que l'objet contenant
- appel automatique des **constructeurs** / **destructeurs**
- pas possible en **Java**

*Qu'est-ce qui manque ?*

# Objets dans des objets

```
class Car : public Vehicle {  
    int power;  
    Door rightDoor;  
    Door * leftDoor;  
public:  
    Car() :  
        rightDoor(this),  
        leftDoor(new Door(this)) {  
    }  
    virtual ~Car() {delete leftDoor;}  
    ...  
};
```



## Il faut un destructeur !

- pour détruire les **pointés** créés par **new** dans le constructeur (`leftDoor`)
- par contre, les objets **contenus** dans les variables sont **autodétruits** (`rightDoor`)

# Copie d'objets

```
void foo() {  
    Car c("Smart-Fortwo", "blue");  
    Car * p = new Car("Ferrari-599-GT0", "red");  
    Car myCar;  
  
    myCar = c;  
    myCar = *p;  
    Car mySecondCar(c);  
}
```

```
class Car : public Vehicle {  
    int power;  
    Door rightDoor;  
    Door * leftDoor;  
    ...  
};
```

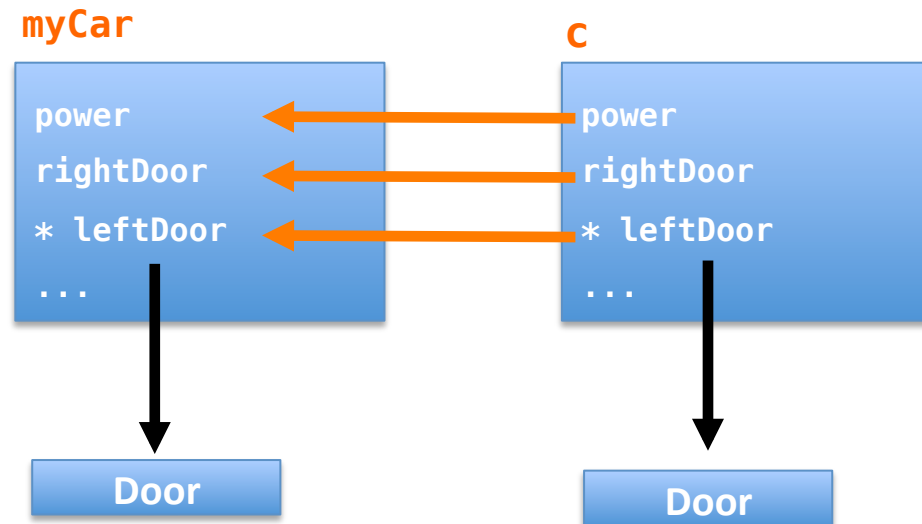
affectation (après la création)

initialisation (lors de la création)

= copie le **contenu des objets**  
**champ à champ** (comme en C)

Noter l'\* : `myCar = *p;`

**Problème ?**



# Copie d'objets

```
void foo() {  
    Car c("Smart-Fortwo","blue");  
    Car * p = new Car("Ferrari-599-GT0","red");  
    Car myCar;  
  
    myCar = *p;  
    Car mySecondCar(c);  
}
```

```
class Car : public Vehicle {  
    int power;  
    Door rightDoor;  
    Door * leftDoor;  
    ...  
};
```

myCar et p ont la même porte gauche !

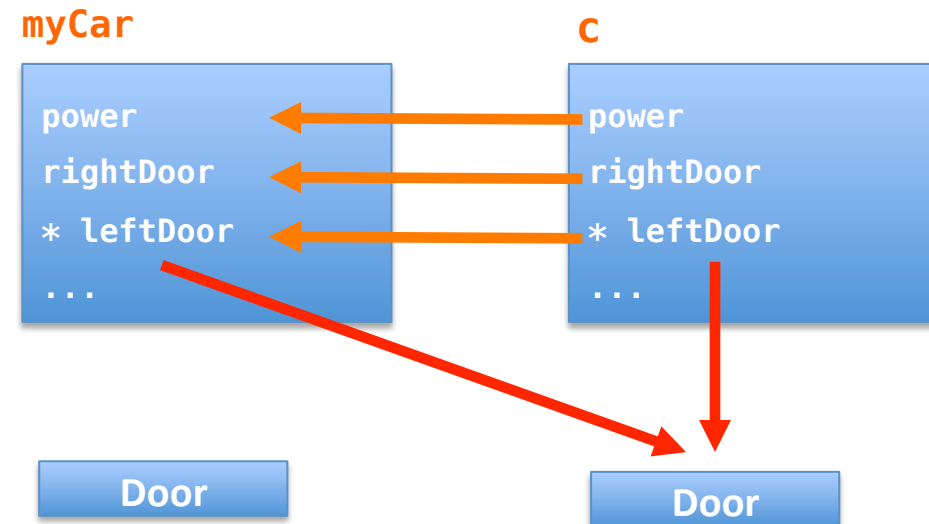
pareil pour mySecondCar et c !

## Problème

- les pointeurs pointent sur le **même objet** !
- **pas de sens** dans ce cas !

## De plus

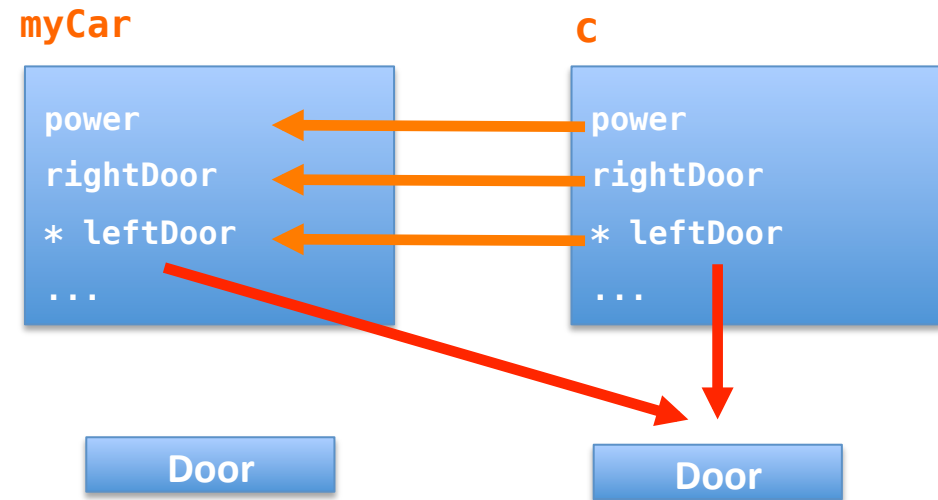
- **Door** est détruit 2 fois !  
=> risque de **plantage** !



# Copie superficielle et copie profonde

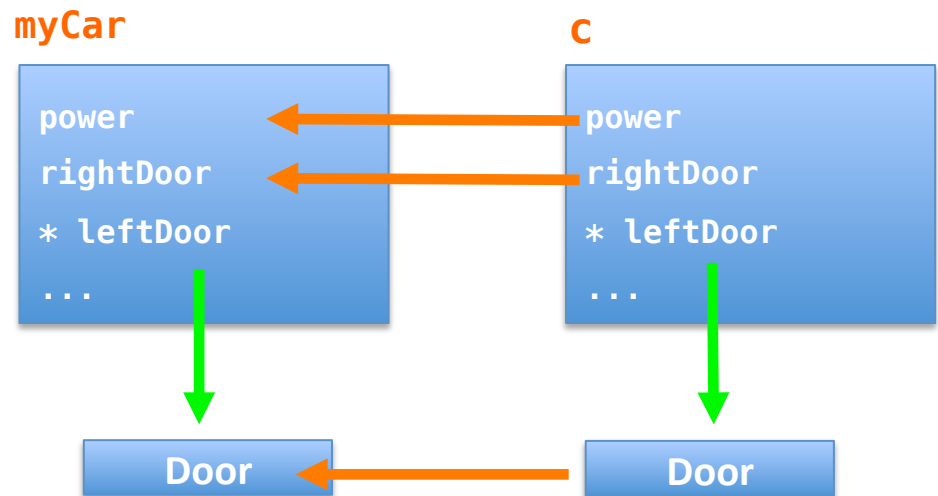
## Copie superficielle (shallow)

- copie **champ à champ**
- souvent **problématique** si l'objet contient des **pointeurs**



## Copie profonde (deep)

- copie le **contenu des pointés** et ce **récurivement**



*Et en Java ?*



# Copie superficielle et copie profonde

## Java

- **même problème** si l'objet contient des **références** Java
- mais **=** ne permet pas la **copie d'objets** (mais des méthodes peuvent le faire)

### C/C++

```
Car * a = new Car(...);  
Car * b = new Car(...);  
a = b;      <----- copie le pointeur  
*a = *b;    <----- copie l'objet pointé
```

```
Car a(...);  
Car b(...);  
a = b;      <----- copie l'objet
```

copie le pointeur

copie l'objet pointé

n'existe pas en Java

copie l'objet

### Java

```
Car a = new Car(...);  
Car b = new Car(...);  
a = b;  
a.copyFrom(b);  
a = b.clone();  
a = new(b);
```

(si ces fonctions existent)

# Opérateurs de copie

## Copy constructor et opérateur d'affectation

- on peut les **redéfinir** pour faire de la **copie profonde**, ou les **interdire**
- si on change l'un **il faut changer l'autre !**

```
class Car : public Vehicle {  
    ....  
    Car(const Car& from);  
    Car& operator=(const Car& from);  
};
```

**Copy constructor**  
appelé à l'**initialisation**

```
Car c("Smart", "blue");  
Car myThirdCar(c);
```

**operator=**  
appelé à l'**affectation**

```
Car mycar;  
myCar = c;
```

# Opérateurs de copie

## Copy constructor et opérateur d'affectation

- **= delete interdit** de les utiliser
- y compris dans les sous-classes

```
class Car : public Vehicle {  
    ....  
    Car(const Car& from) = delete;  
    Car& operator=(const Car& from) = delete;  
};
```

Copy constructor interdit

operator= interdit

## Remarque

- les **copy constructors** existent en **Java** !

# Opérateurs de copie

## Copy constructor et opérateur d'affectation

```
class Car : public Vehicle {  
    Door rightDoor;  
    Door * leftDoor;  
public:  
    Car(const Car&);  
    Car& operator=(const Car&);  
    ...  
};
```

```
Car::Car(const Car& from) : Vehicle(from) {  
    rightDoor = from.rightDoor;  
    leftDoor = from.leftDoor ? new Door(*from.leftDoor) : nullptr;  
}
```

```
Car& Car::operator=(const Car& from) {  
    Vehicle::operator=(from);  
    rightDoor = from.rightDoor;  
    if (leftDoor && from.leftDoor) *leftDoor = *from.leftDoor;  
    else {  
        delete leftDoor;  
        leftDoor = from.leftDoor ? new Door(*from.leftDoor) : nullptr;  
    }  
    return *this;  
}
```

ne pas oublier de copier  
les champs de Vehicle !

# Tableaux

tableaux  
dans la  
pile

tableaux  
dynamiques

```
void foo() {  
    int count = 10, i = 5;  
  
    double tab1[count];  
    double tab2[] = {0., 1., 2., 3., 4., 5.};  
    cout << tab1[i] <<" "<< tab2[i] << endl;  
  
    double * p1 = new double[count];  
    double * p2 = new double[count]();  
    double * p3 = new double[count]{0., 1., 2., 3., 4., 5.};  
    cout << p1[i] <<" "<< p2[i] <<" "<< p3[i] << endl;  
  
    delete [] p1;  
    delete [] p2;  
    delete [] p3;  
}
```

initialise à 0

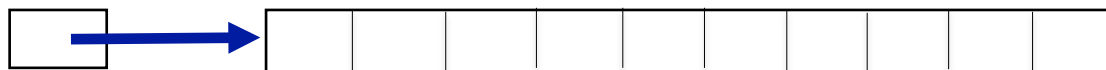
C++11 seulement

ne pas oublier []

tab



p



# Coût de l'allocation mémoire

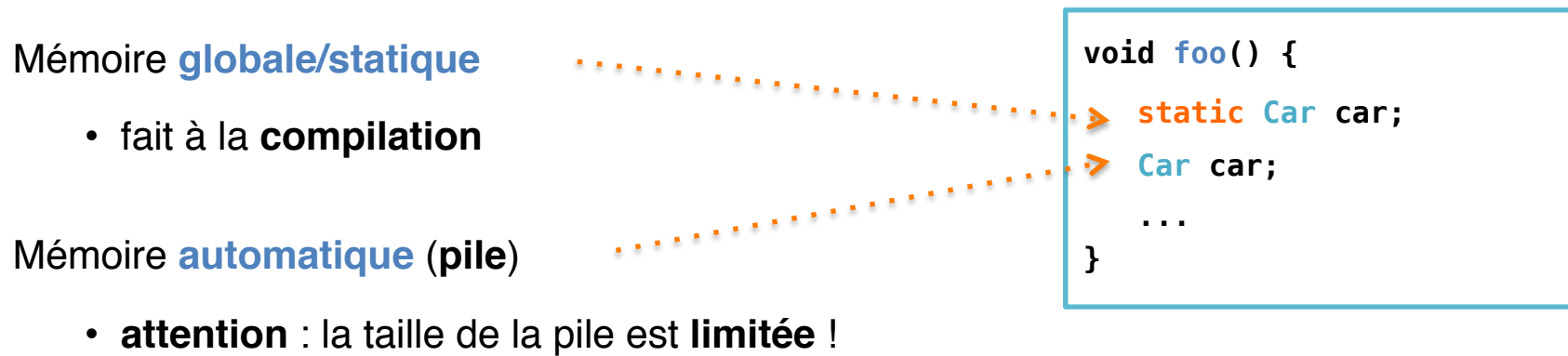
## Gratuit ou négligeable

Mémoire **globale/statique**

- fait à la **compilation**

Mémoire **automatique (pile)**

- **attention** : la taille de la pile est **limitée** !



```
void foo() {  
    > static Car car;  
    > Car car;  
    ...  
}
```

## Objets dans les objets

- généralement **aucune** allocation

# Coût de l'allocation mémoire

## Coûteux

Mémoire **dynamique** (**tas**) :

- **new** en **C++** (malloc en **C**)
  - **ramasse-miettes** en **Java**
- 

```
void foo() {  
    Car * s = new Car();  
    ...  
}
```

## Dépend de divers facteurs

- typiquement : **beaucoup** de créations / destructions entremêlées
- "beaucoup" veut dire beaucoup ! (négligeable sinon)

## Le ramasse-miettes "stops the world"

- problématique pour **temps réel**
- il existe **plusieurs types** de ramasse miettes (avantages / inconvénients différents)

# Compléments

## Mémoire constante

- parfois appelée **statique**
- ex : littéraux comme "Hello Word"

## Variables volatile

- empêchent optimisations du compilateur
- pour **threads** ou **entrées/sorties** selon le langage

## En C/C++

### Variables globales

- accessibles dans **toutes** les fonctions de **tous les fichiers** => **dangereuses !**

### Variables globales statiques

- accessibles dans **toutes** les fonctions **d'un fichier**



# Chapitre 4 : Types, constance & smart pointers

# Types de base

bool

char

short

←----- peuvent être  
signed ou unsigned

int

long

long long

wchar\_t, char16\_t, char32\_t

float

double

long double

La taille dépend de la **plateforme** !

- tailles dans `<climits>` et `<cfloat>`

Le signe de **char** également !!!

- entre [0, 255] **ou bien** [-128, 127]

float et double **ne sont pas** des réels !

- mais une **approximation**
- => attention aux **arrondis** !

# Types normalisés

`int8_t`

`int16_t`

`int32_t`

`int64_t`

`intmax_t`

`uint8_t`

`uint16_t`

`uint32_t`

`uint64_t`

`uintmax_t`

**etc.**

**Portabilité :** même taille sur toutes les plateformes

Définis dans `<stdint>`

# Noms de types

**using** (typedef en C) : crée un nouveau nom de type

```
using ShapePtr = Shape *; // en C++11
```

```
using ShapeList = std::list<Shape *>;
```

```
typedef Shape * ShapePtr; // en C et C++
```

```
typedef std::list<Shape *> ShapeList;
```

# Inférence de types

**auto** : type inféré par le compilateur (C++11)

```
auto count = 10;
```

==

```
int count = 10;
```

```
auto PI = 3.1416;
```

```
double PI = 3.1416;
```

```
ShapeList shapes;
```

*Rappel:* `using ShapeList = std::list<Shape*>;`

```
auto it = shapes.begin();
```

```
std::list<Shape*>::iterator it = shapes.begin();
```

**decltype** : type d'une variable (C++11)

```
struct Point {double x, y};
```

```
Point * p = new Point();
```

```
decltype(p->x) val;    <.....    val a le type de p->x
```

# Constantes

## Macros du C (obsolète)

- substitution textuelle **avant** la compilation

```
#define PORT 3000  
#define HOST "localhost"
```

## Enumérations

- valeurs intégrales
- commencent à 0 par défaut
- existent en **Java**

```
enum {PORT = 3000};  
enum Status {OK, BAD, UNKNOWN};  
enum class Status {OK, BAD, UNKNOWN};
```

## Variables **const**

- **final** en **Java**
- les **littéraux doivent être const**

```
const int PORT = 3000;  
const char * HOST = "localhost";
```

## **constexpr** (C++11)

- expression calculée à la **compilation**

```
constexpr const char * HOST = "localhost";
```

# Objets constants ou immuables

## Questions à se poser

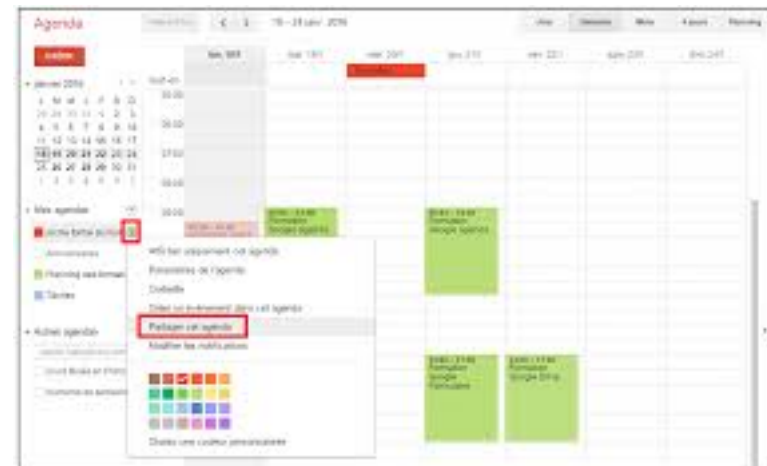
- à qui **appartient** l'objet (qui le **crée**, qui le **détruit** ?)
- qui a le droit de le **lire** ?
- qui a le droit de le **modifier** ?

## Exemple 1

- tableaux du TP !

## Exemple 2

- **Alice** a un **calendrier partagé**
  - ses collègues peuvent le **lire** mais pas le **modifier**
- ⇒ **Alice** et ses collègues n'ont pas le même **point de vue** sur l'objet !



# Objets constants ou immuables

```
void pote(User* alice) {  
    Cal* c = alice->getCal();  
    ...  
};
```

```
class User {  
    Cal* cal = new Cal;  
public:  
    User(...) {...}  
    Cal* getCal() {return cal;}  
};
```

**Problème ?**

Cal : classe du calendrier



# Objets constants ou immuables

```
void pote(User* alice) {  
    Cal* c = alice->getCal();  
    ...  
};
```

```
class User {  
    Cal* cal = new Cal;  
public:  
    User(...) {...}  
    Cal* getCal() {return cal;}  
};
```

peut modifier le contenu du pointé !

## Problème !

- `pote()` peut **modifier** le contenu du calendrier d'**Alice** !

## Solutions ?

# Objets immuables

```
void pote(User* alice) {  
    Cal* c = alice->getCal();  
    ...  
};
```

```
class User {  
    Cal* cal = new Cal;  
public:  
    User(...) {...}  
    Cal* getCal() {return cal;}  
};
```

## Solution 1 : objets immuables

- aucune méthode ne permet de **modifier** l'objet
  - cas de **String**, **Integer**... en **Java**
  - peuvent être **partagés** sans risque (y compris dans les **threads**) !
- pas applicable ici : **Alice** ne pourrait **pas modifier** son propre calendrier !

# Objets constants

```
void pote(const User* alice) {  
    const Cal* c = alice->getCal();  
    ...  
};
```

```
class User {  
    Cal* cal = new Cal;  
public:  
    User(...) {...}  
    const Cal* getCal() const {return cal;}  
};
```

## Solution 2 : objets constants

- **const\*** => le pointé ne peut pas être modifié
  - **Alice** peut modifier car `cal` n'est pas **const\***
  - **pote()** ne peut pas modifier car `getCal()` renvoie **const\***

cette methode ne permet  
pas de modifier l'objet

# Copie

```
void pote(const User* alice) {  
    Cal* c = new Cal(*alice->getCal());  
    ...  
};
```

```
class User {  
    Cal* cal = new Cal;  
public:  
    User(...) {...}  
    const Cal* getCal() const {return cal;}  
};
```

## Solution 3 : copie

- **c** pointe sur une **copie** du calendrier
- les contenus ne sont plus **synchronisés**
- parfois c'est ce qu'on veut (cf. TP) :
  - ex : garder l'historique
  - ex : l'objet d'origine peut être détruit sans qu'on soit averti

# const sert aussi à éviter les bugs !

```
class Plane {  
    double pitch;  
    ....  
public:  
    double getPitch() const {  
        return pitch = 0.12;  
    }  
    void setPitch(double angle);  
    ...  
};
```

oops typo !  
erreur de compil car méthode **const**

```
double readPitch(const Plane& c) {  
    c.setPitch(0.);  
}
```

oops typo !  
erreur de compil car paramètre **const**



## Erreur de compilation dans les deux cas

- les **const** servent à éviter des erreurs aussi **dangereuses** que bêtes !

# Pointeurs et pointés

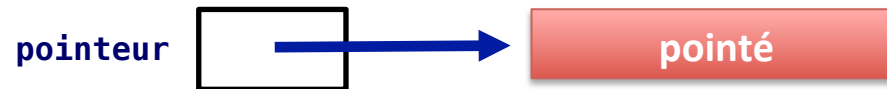
Qu'est-ce qui est constant : le **pointeur** ou le **pointé** ?

**const** porte sur « ce qui suit »

// \*s est constant:

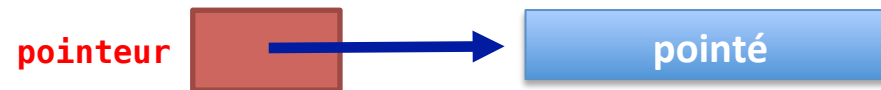
**const** char \* s

char **const** \* s



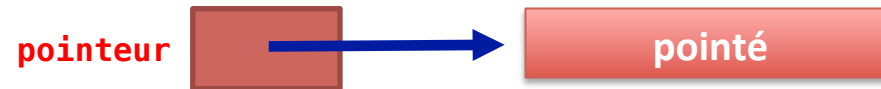
// s est constant:

char \* **const** s



// les deux sont constants:

**const** char \* **const** s



# Constance logique

```
class Doc {  
    string text;  
    mutable Printer * printer;    // peut être modifiée même si Doc est const  
public:  
    Doc() : printer(nullptr) {}  
    void print() const {  
        if (!printer) printer = new Printer();    // OK car printer est mutable  
    }  
    ....  
};
```

## Objet vu comme immuable

- l'objet n'a pas de méthode permettant de le modifier : **constance logique**

## Mais qui peut modifier son état interne

- **print()** peut allouer une ressource interne : **non-constance physique**

# Smart pointers

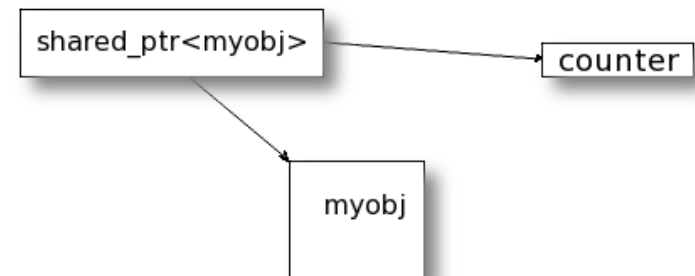
```
#include <memory>

void foo() {
    shared_ptr<Circle> p(new Circle(0, 0, 50));           // count=1
    shared_ptr<Circle> p2;

    p2 = p;                                               // p2 pointe aussi sur l'objet => count=2
    p.reset();                                             // p ne pointe plus sur rien => count=1
}                                                         // p2 est détruit => count=0 => objet auto-détruit
```

## shared\_ptr (C++11)

- **smart pointer** avec **comptage de références**
  - objet **auto-détruit** quand le compteur arrive à 0
  - mémoire gérée **automatiquement** : **plus de delete !**

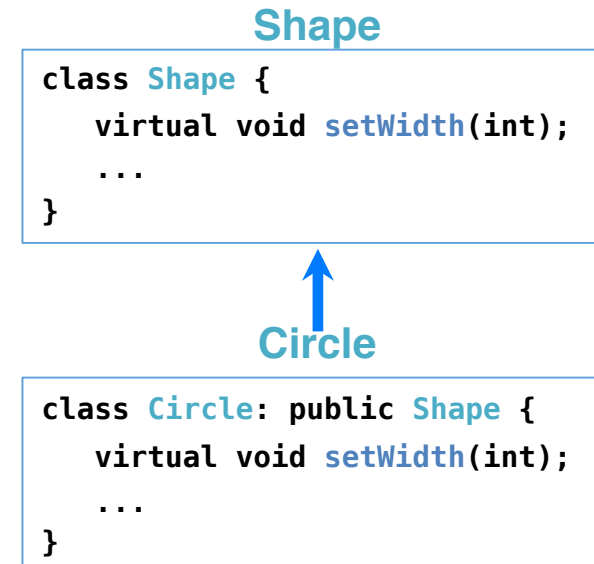




# Smart pointers

```
#include <memory>

void foo() {
    shared_ptr<Shape> p(new Circle(0, 0, 50));
    p->setWidth(20);
}
```



## S'utilisent comme des "raw pointers"

- polymorphisme
- déréférencement par opérateurs `->` ou `*`

## Attention !

- doivent pointer sur des objets créés avec **new**
- **pas convertibles en raw pointers** (perd le compteur)
- ne marchent pas si **dépendances circulaires** (voir **weak\_ptr**)

# Smart pointers

```
#include <memory>

void foo() {
    vector< unique_ptr<Shape> > vect;
    vect.push_back( unique_ptr(new Circle(0,0,50) ));
}
```

## unique\_ptr : si l'objet n'a qu'un seul pointeur

- pas de comptage de référence => moins coûteux
- utiles pour **tableaux** ou **conteneurs** pointant des objets

## weak\_ptr

- pointe un objet **sans** le "posséder"
- permet de tester si l'objet **existe encore**
- utiles si **dépendances circulaires**

# Chapitre 5 : Bases des Templates et STL

# Programmation générique

```
template <typename T>  
T mymax(T x, T y) {return (x > y ? x : y);}
```

```
i = mymax(4, 10);
```

←..... T déduit: **int**

```
x = mymax(66., 77.);
```

←..... T déduit: **double**

```
y = mymax<float>(66., 77.);
```

←..... T spécifié: **float**

```
string s1 = "aaa", s2 = "bbb";
```

←..... T déduit: **string**

```
string s = mymax(s1, s2);
```

## Templates = les types sont des paramètres

⇒ algorithmes et types **génériques**

⇒ max() est **instanciée** à la **compilation**

comme si on avait défini **4 fonctions différentes**

# Classes templates

```
template <typename T> class Matrix {
public:
    void set(int i, int j, T val) { ... }
    T get(int i, int j) const { ... }
    void print() const { ... }
    ....
};

template <typename T>
Matrix<T> operator+(Matrix<T> m1, Matrix<T> m2) {
    ....
}

Matrix<float> a, b;
a.set(0, 0, 10);
a.set(0, 1, 20);
....
Matrix<float> res = a + b;
res.print();

Matrix<complex> cmat;
Matrix<string> smat; // why not?
```

## T peut être ce qu'on veut

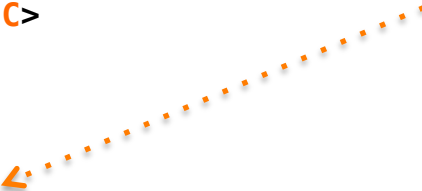
- pourvu que ce soit **compatible** avec les méthodes de **Matrix**

appelle: operator+(a,b)

# Exemple


```
template <typename T, int L, int C>
class Matrix {
    T values[L * C];
public:
    void set(int i, int j, const T & val) {values[i * C + j] = val;}
    const T& get(int i, int j) const {return values[i * C + j];}
    void print() const {
        for (int i = 0; i < L; ++i) {
            for (int j = 0; j < C; ++j) cout << get(i,j) << " ";
            cout << endl;
        }
    }
};
```

passage par const référence  
(chapitre suivant)



```
template <typename T, int L, int C>
Matrix<T,L,C> operator+(const Matrix<T,L,C> & a, const Matrix<T,L,C> & b) {
    Matrix<T,L,C> res;
    for (int i = 0; i < L; ++i)
        for (int j = 0; j < C; ++j)
            res.set(i, j, a.get(i,j) + b.get(i,j));
    return res;
}
```

NB: on verra une solution plus performante au chapitre suivant



# Standard Template Library (STL)

```
std::vector<int> v(3);    // vecteur de 3 entiers  
v[0] = 7;  
v[1] = v[0] + 3;  
v[2] = v[0] + v[1];  
reverse(v.begin(), v.end());
```

## Conteneurs

- pour traiter une **collection** d'objets
- compatibles avec **objets et types de base** (contrairement à Java)
- gèrent **automatiquement leur mémoire**
  - exles : **array, vector, list, map, set, deque, queue, stack ...**

## Itérateurs

- pour pointer sur les **éléments** des conteneurs : ex : **begin()** et **end()**

## Algorithmes

- **manipulent** les données des conteneurs : ex : **reverse()**

# Vecteurs

```
#include <vector>

void foo() {
    std::vector<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));

    for (unsigned int i=0; i < path.size(); ++i) {
        path[i].print();
    }
}
```

```
class Point {
    int x, y;
public:
    Point(int x, int y) : x(x), y(y) {}
    void print() const;
};
```

path contient les Points :

x	x	x
y	y	y

↑ chaque élément  
est un objet Point

**Accès direct** aux éléments :

- `path[i]` ou `path.at(i)`      `at(i)` vérifie l'indice (exception) mais pas `[i]`

**En théorie**

- vecteurs peu efficaces pour l'**insertion/suppression** d'éléments

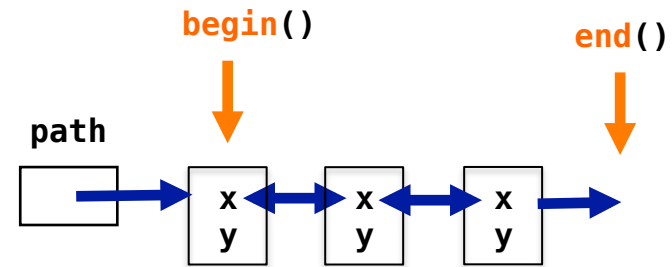


# Listes et itérateurs

```
#include <list>

void foo() {
    std::list<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));

    for (auto & it : path) it.print();
}
```



## Pas d'accès direct aux éléments

=> utiliser des **itérateurs** (le **&** est optionnel et généralement plus rapide)

## En théorie

- listes efficaces pour l'**insertion/suppression** d'éléments

## En réalité

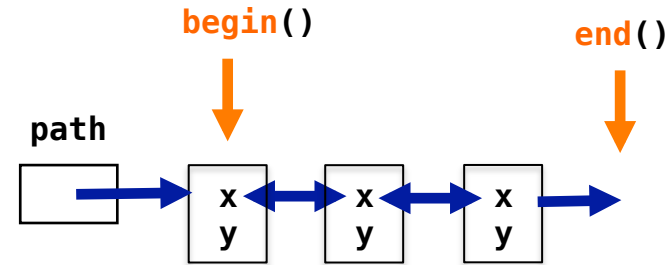
- les vecteurs sont **presque toujours plus rapides**

# Listes et itérateurs

```
#include <list>

void foo() {
    std::list<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));

    for (auto & it : path) it.print();
}
```



Equivaut à :

```
for (std::list<Point>::iterator it = path.begin(); it != path.end(); ++it) {
    (*it).print();
}
```

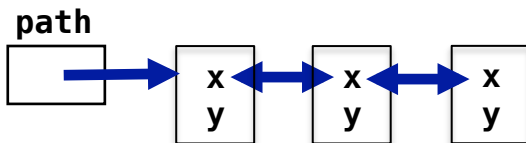
↑  
parenthèses nécessaires

# Conteneurs et pointeurs

```
#include <list>

void foo() {
    std::list<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));

    for (auto & it : path) it.print();
}
```

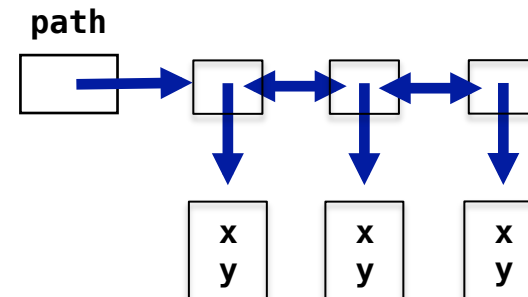


- A gauche : la **liste contient** les éléments
- A droite: la **liste pointe** sur les éléments

```
#include <list>

void foo() {
    std::list<Point*> path;
    path.push_back(new Point(20,20));
    path.push_back(new Point(50,50));
    path.push_back(new Point(70,70));

    for (auto & it : path) it->print();
}
```



*Problème ?*

# Conteneurs et pointeurs

```
#include <list>

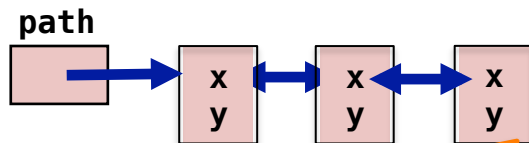
void foo() {
    std::list<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));

    for (auto & it : path) it.print();
}
```

```
#include <list>

void foo() {
    std::list<Point *> path;
    path.push_back(new Point(20,20));
    path.push_back(new Point(50,50));
    path.push_back(new Point(70,70));

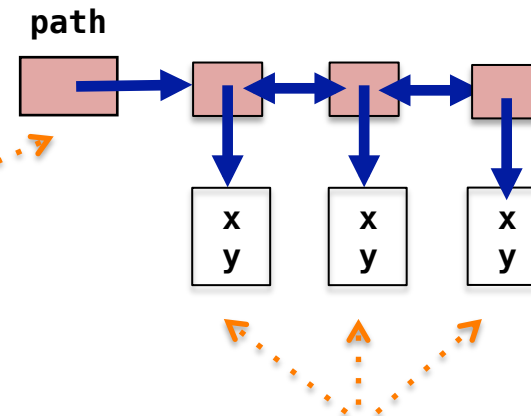
    for (auto & it : path) it->print();
}
```



Détruit

## Problème à droite !

- la liste est **détruite** (path est dans la **pile**)
- mais pas les pointés !**



Pas détruit

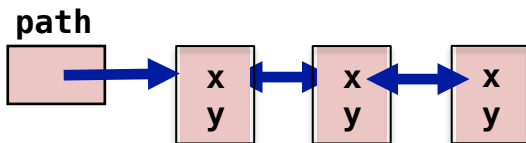
## Solutions ?

# Conteneurs et pointeurs

```
#include <list>

void foo() {
    std::list<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));

    for (auto & it : path) it.print();
}
```



## Solution 1 : pas de pointeurs

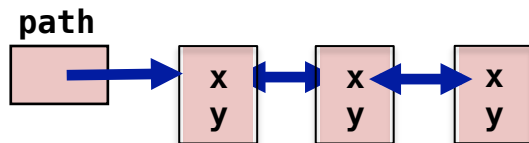
- simple et efficace (mémoire)
- *limitation ?*

# Conteneurs et pointeurs

```
#include <list>

void foo() {
    std::list<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));

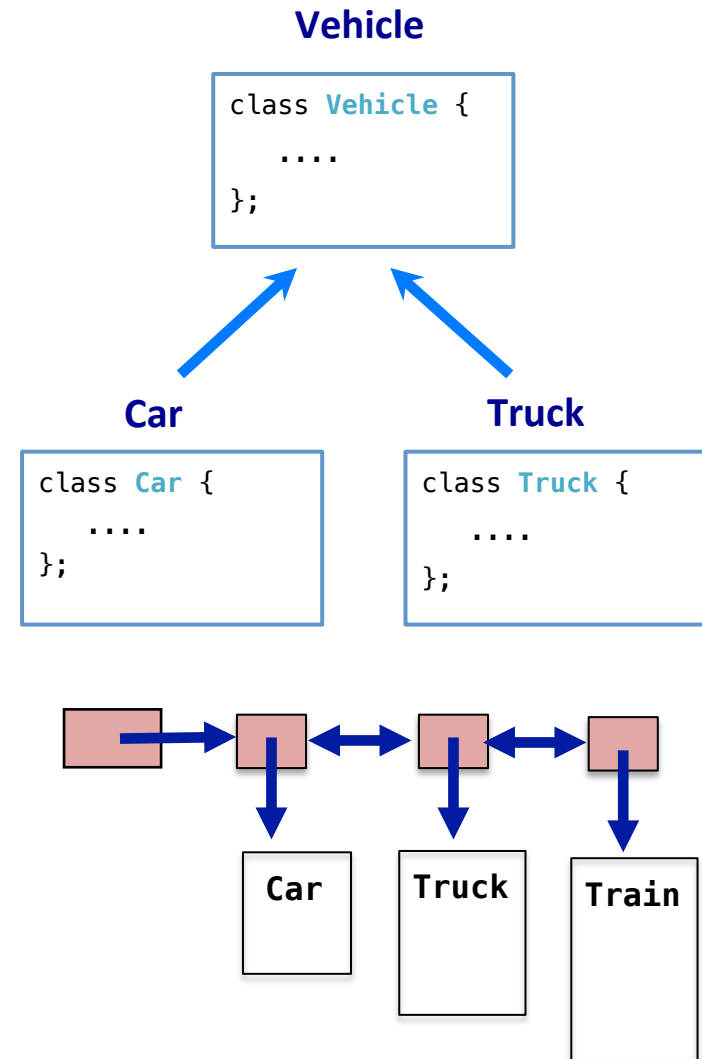
    for (auto & it : path) it.print();
}
```



## Solution 1 : pas de pointeurs

- **limitation** : les éléments doivent avoir le même type

⇒ **pointeurs** nécessaires si **polymorphisme** (toujours le cas en **Java**)



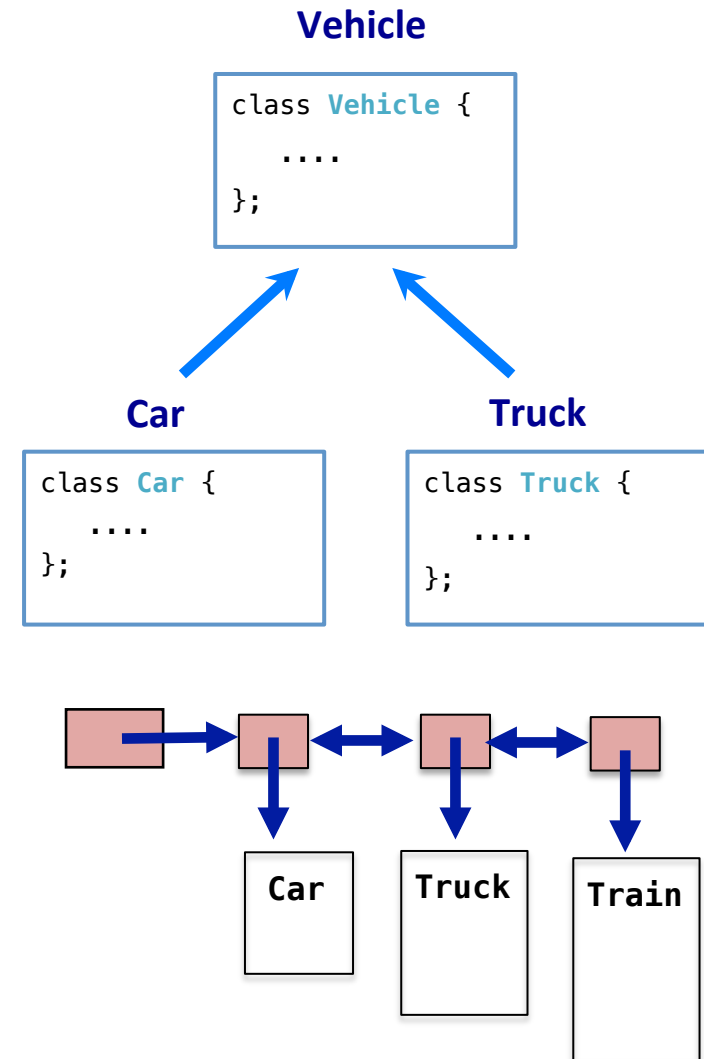
# Conteneurs et pointeurs

## Solution 2 : pointeurs

```
#include <list>

void foo() {
    std::list<Vehicule*> v;
    v.push_back(new Car());
    v.push_back(new Truck());
    v.push_back(new Train());

    for (auto & it : v) it->print();
    ...
    for (auto & it : v) delete it;
}
```



## Solution 3 : smart pointers

```
void foo() {
    std::list<shared_ptr<Vehicule>> v;
    v.push_back(make_shared<Car>());
    ....
}
```

ou: `v.push_back(shared_ptr<Car>(new Car))`

# Enlever des éléments

## Enlever tous les éléments

```
std::vector<int> v{0, 1, 2, 3, 4, 5};  
v.clear(); // enlève tout
```

## Enlever les éléments à une position ou un intervalle

```
std::vector<int> v{0, 1, 2, 3, 4, 5};  
v.erase(v.begin()+1); // enlève v[1]  
v.erase(v.begin(), v.begin()+3); // enlève de v[0] à v[2]
```

```
std::list<int> l{0, 1, 2, 3, 4, 5};  
auto it = l.begin();  
std::advance(it, 3);  
l.erase(it); // enlève l(3)  
  
l = {0, 1, 2, 3, 4, 5};  
it = l.begin();  
std::advance(it, 3);  
l.erase(l.begin(), it); // enlève de l(0) à l(2)
```



# Enlever des éléments

## Enlever les éléments ayant une certaine valeur

```
std::vector<int> v{0, 1, 2, 1, 2, 1, 2};  
v.erase(std::remove(v.begin(), v.end(), 2), v.end()); // enlève tous les 2
```

```
std::list<int> l{0, 1, 2, 1, 2, 1, 2};  
l.remove(2); // enlève tous les 2
```

## Enlever les éléments vérifiant une condition

```
bool is_odd(const int & value) {return (value%2) == 1;}  
  
std::list<int> l{0, 1, 2, 1, 2, 1, 2};  
l.remove_if(is_odd); // enlève tous les nombres impairs
```

# Deque ("deck")

```
#include <deque>

void foo() {
    std::deque<Point> path;
    path.push_back(Point(20, 20));
    path.push_back(Point(50, 50));
    path.push_back(Point(70, 70));

    for (auto & it : path) it.print();
    for (unsigned int i=0; i < path.size(); ++i) path[i].print();
}
```

## Hybride entre **liste** et **vecteur**

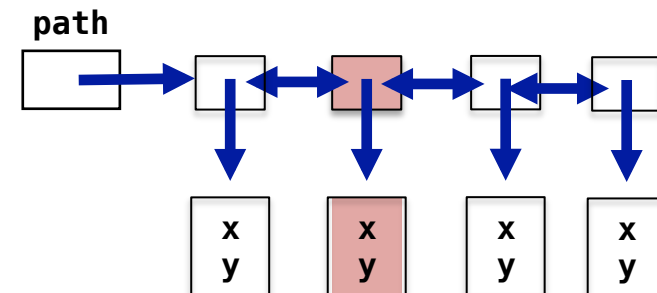
- **accès direct** aux éléments par **[i]** ou **at(i)**
- **faible coût** d'insertion / suppression
- **plus coûteux** en **mémoire**

# Enlever plusieurs éléments dans une liste

```
std::list<Point> path;  
int value = 200;           // détruire les points dont x vaut 200  
  
for (auto it = path.begin(); it != path.end(); ) {  
    if ((*it)->x != value) it++;  
    else {  
        auto it2 = it;  
        ++it2;  
        delete *it;           // détruit l'objet pointé par l'itérateur  
        path.erase(it);       // it est invalide après erase()  
        it = it2;  
    }  
}
```

## Attention

- l'itérateur **it** est **invalide** après **erase()**  
=> second itérateur **it2**



# Table associative (map)

```
#include <iostream>
#include <map>

using Dict = std::map<string, User*>;

void foo() {
    Dict dict;

    dict["Dupont"] = new User("Dupont", 666);           // ajout
    dict["Einstein"] = new User("Einstein", 314);

    auto it = dict.find("Dupont");                       // recherche
    if (it == dict.end())
        std::cout << "pas trouvé" << std::endl;
    else
        std::cout << "id: " << it->second->getID() << std::endl;
}
```

```
class User {
    string name,
    int id;
public:
    User(const string& name, int id);
    int getID() const {return id;}
};
```

On pourrait aussi utiliser le **conteneur set**

# Algorithmes exemple : trier les éléments d'un conteneur

```
#include <string>
#include <vector>
#include <algorithm>

class User {
    string name;
public:
    User(const string & name) : name(name) {}
    friend bool compareEntries(const User &, const User &);
};

// inline nécessaire si la fonction est définie dans un header
inline bool compareEntries(const User & e1, const User & e2) {
    return e1.name < e2.name;
}

void foo() {
    std::vector<User> entries;
    ....
    std::sort(entries.begin(), entries.end(), compareEntries);
    ....
}
```

# Template metaprogramming

```
template <int N> struct Factorial {  
    static const int value = N * Factorial<N-1>::value;  
};  
  
template <> struct Factorial<0> {  
    static const int value = 1;  
};  
  
void foo() {  
    int x = Factorial<4>::value;    // vaut 24  
    int y = Factorial<0>::value;    // vaut 1  
}
```

calcule factorielle  
à la **compilation** !

spécialisation  
de template

instanciation  
réursive

## Programme qui génère un programme

- valeur calculée à la **compilation** par **instanciation réursive** des templates
- **spécialisation** = définition de **cas spécifique** (ici l'appel terminal)
- le **paramètre** n'est pas forcément un type (ici un **int**)

# Polymorphisme paramétré

Comment avoir une fonction `print()` générique ?

```
void foo() {  
    print(55);  
  
    std::string s = "toto";  
    print(s);  
  
    Point p(10,20);  
    print(p);  
  
    std::vector<int> vi{0, 1, 2, 3, 4, 5};  
    print(vi);  
  
    std::vector<Point> vp{{0, 1},{2, 3},{4, 5}};  
    print(vp);  
  
    std::list<Point> lp{{0, 1},{2, 3},{4, 5}};  
    print(lp);  
}
```

```
class Point {  
    int x, y;  
public:  
    Point(int x, int y) : x(x), y(y) {}  
    void print() const;  
};
```

# Polymorphisme paramétré

cas général

spécialisation  
totale

spécialisations  
partielles

```
template <typename T> void print(const T & arg) {  
    cout << arg << endl;  
}  
  
template <> void print(const Point & p) {  
    p.print(cout);  
}  
  
template <typename T> void print(const std::vector<T> & v) {  
    for (auto& it : v) print(it);  
}  
  
template <typename T> void print(const std::list<T> & l) {  
    for (auto& it : l) print(it);  
}
```

```
void foo() {  
    print(55);  
    string s = "toto";  
    print(s);  
    Point p(10,20);  
    print(p);  
    std::vector<int> vi{0, 1, 2, 3, 4, 5};  
    print(vi);  
    std::vector<Point> vp{{0, 1},{2, 3},{4, 5}};  
    print(vp);  
    std::list<Point> lp{{0, 1},{2, 3},{4, 5}};  
    print(lp);  
}
```

C'est une autre **forme de polymorphisme**  
effectuée **à la compilation**

**Amélioration:**

même définition pour tous les conteneurs :  
=> comment les **détecter** ?



# Traits <type\_traits>

```
is_array<T>
is_class<T>
is_enum<T>
is_floating_point<T>
is_function<T>
is_integral<T>
is_pointer<T>
is_arithmetic<T>

is_object<T>
is_abstract<T>
is_polymorphic<T>
is_base_of<Base, Derived>
is_same<T, V>
etc.
```

```
#include <type_traits>
```

Permet calculs sur les **types** à la compilation :

```
void foo() {
    cout << is_integral<int>::value << endl;    // 1
    cout << is_integral<float>::value << endl;   // 0
    cout << is_class<int>::value << endl;        // 0
    cout << is_class<Point>::value << endl;      // 1
}
```

Il n'y a pas `is_container<>`  
Comment le définir ?

# Type matching

```
template <typename T> struct is_container {  
    static const bool value = false;  
};  
  
template <typename T> struct is_container<std::vector<T>> {  
    static const bool value = true;  
};  
  
template <typename T> struct is_container<std::list<T>> {  
    static const bool value = true;  
};  
  
// ... etc.
```

← cas général

← spécialisations  
partielles

**Type matching** : la définition correspondant au **type** est instanciée

```
void foo() {  
    cout << is_container<int>::value << endl;           // 0 (false)  
    cout << is_container<std::vector<int>>::value << endl; // 1 (true)  
    cout << is_container<std::list<int>>::value << endl; // 1 (true)  
}
```

# Conditions sur les types

```
template <typename T>
void print(const T& arg,
           typename std::enable_if< !is_container<T>::value, bool>::type = true) {
    cout << arg << " " << endl;
}

template <typename T>
void print(const T& arg,
           typename std::enable_if<is_container<T>::value, bool>::type = true) {
    for (auto & it : arg) print(it);
}
```

```
void foo() {
    print(55);
    string s = "toto";
    print(s);
    std::vector<int> vi{0,1,2,3,4,5};
    print(vi);
    std::list<float> li{0,1,2,3,4,5};
    print(li);
}
```

La définition **valide** est instanciée

SFINAE: "Substitution failure is not an error"

type **bool** si **is\_container<T>** est vrai, **indéfini** sinon

**print()** matche avec la définition **valide** pour type **T** de son argument

# Templates C++ vs. Generics Java

```
template <typename T>
T max(T x, T y) {return (x > y ? x : y);}

i = max(4, 10);
x = max(6666., 77777.);
```

## Templates C++

- **instanciation à la compilation** => **optimisation** prenant compte des types
- **puissants** (Turing complets) ... mais pas très lisibles !

## Generics Java

**Sémantique** et **implémentation** différentes :

- pas pour **types de base**
- pas **instanciés** à la compilation, pas de **spécialisation**
- pas de **calcul sur les types** (les types sont « effacés »)

# Chapitre 6 :

## Passage par valeur et par référence

# Passer des valeurs à une fonction

```
class Truc { C++
    void print(int n, const string * p) {
        cout << n << " " << *p << endl;
    }

    void foo() {
        int i = 10;
        string * s = new string("YES");
        print(i, s);
    }
    ...
};
```

```
class Truc { Java
    void print(int n, String p) {
        System.out.println(n + " " + p);
    }

    void foo() {
        int i = 10;
        String s = new String("YES");
        print(i, s);
    }
    ...
}
```

## Quelle est la relation

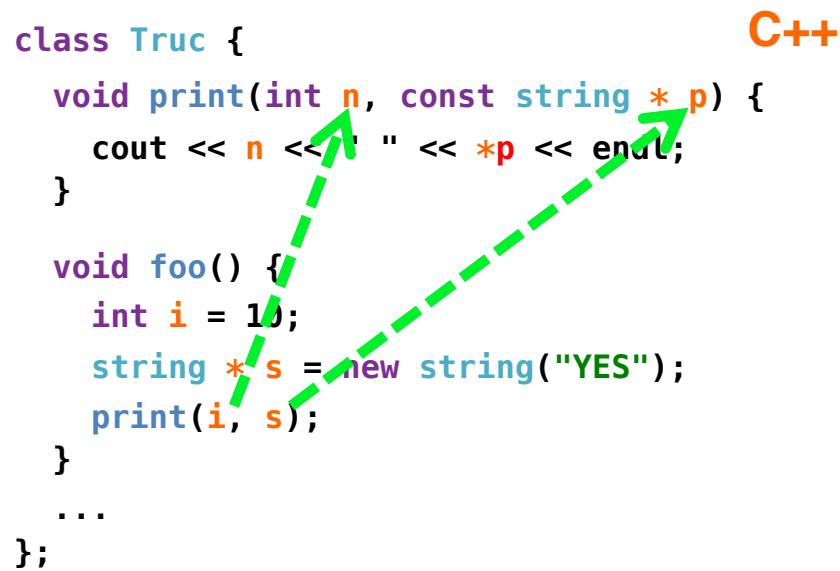
- entre les **arguments** (i, s) passés à la méthode **print()**
- et ses **paramètres formels** (n, p)



# Passer des valeurs à une fonction

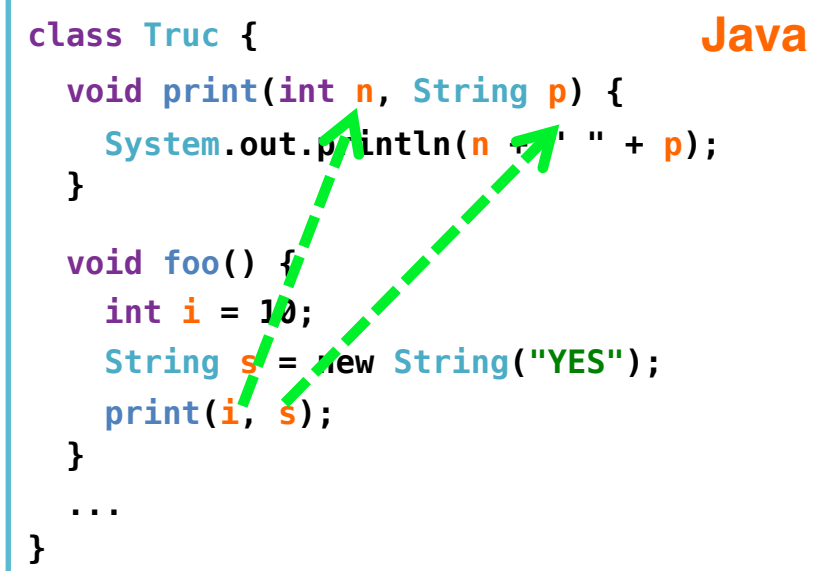
**C++**

```
class Truc {  
    void print(int n, const string *p) {  
        cout << n << " " << *p << endl;  
    }  
  
    void foo() {  
        int i = 10;  
        string *s = new string("YES");  
        print(i, s);  
    }  
    ...  
};
```



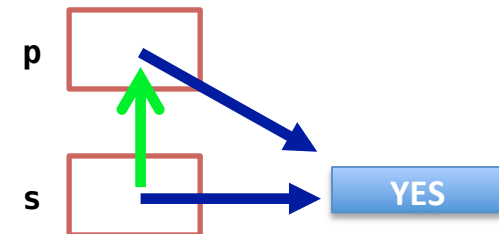
**Java**

```
class Truc {  
    void print(int n, String p) {  
        System.out.println(n + " " + p);  
    }  
  
    void foo() {  
        int i = 10;  
        String s = new String("YES");  
        print(i, s);  
    }  
    ...  
}
```



## Passage par valeur

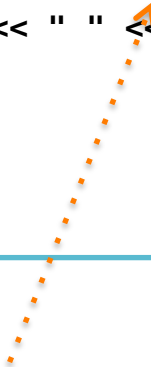
- la valeur de l'**argument** est **copiée** dans le **paramètre**
  - le **pointeur s** est **copié** dans le pointeur **p**
  - le **pointé** n'est **pas** copié !
  - références **Java** = comme **pointeurs**



# Passer des valeurs à une fonction

```
class Truc {  
    void print(int n, const string * p) {  
        cout << n << " " << *p << endl;  
    }  
    ...  
};
```

C++



```
class Truc {  
    void print(int n, String p) {  
        System.out.println(n + " " + p);  
    }  
    ...  
}
```

Java

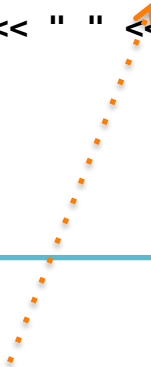
**Remarque : pourquoi const ?**



# Passer des valeurs à une fonction

```
class Truc {  
    void print(int n, const string * p) {  
        cout << n << " " << *p << endl;  
    }  
    ...  
};
```

C++



```
class Truc {  
    void print(int n, String p) {  
        System.out.println(n + " " + p);  
    }  
    ...  
}
```

Java

## Remarque : pourquoi const ?

- `print()` ne doit pas changer le **pointé \*p**
  - en C/C++ : **const \***
  - en Java : **String** est **immutable**

# Récupérer des valeurs d'une fonction

**C++**

```
class Truc {  
    void get(int n, const string * p) {  
        n = 20;  
        p = new string("NO");  
    }  
  
    void foo() {  
        int i = 10;  
        string * s = new string("YES");  
        get(i, s);  
        cout << i << " " << *s << endl;  
    }  
  
    ...  
};
```

**Java**

```
class Truc {  
    void get(int n, String p) {  
        n = 20;  
        p = new String("NO");  
    }  
  
    void foo() {  
        int i = 10;  
        String s = new String("YES");  
        get(i, s);  
        System.out.println(i + " " + s);  
    }  
  
    ...  
}
```

## Résultat

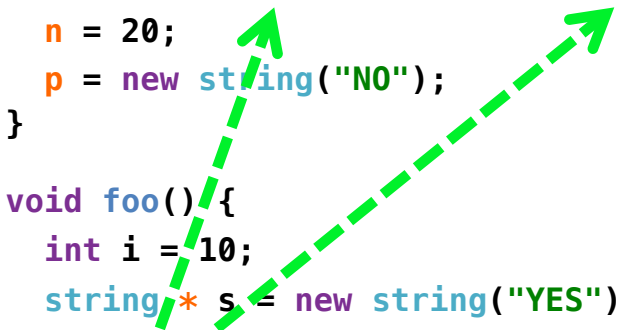
- 10 YES
- 20 NO



# Récupérer des valeurs d'une fonction

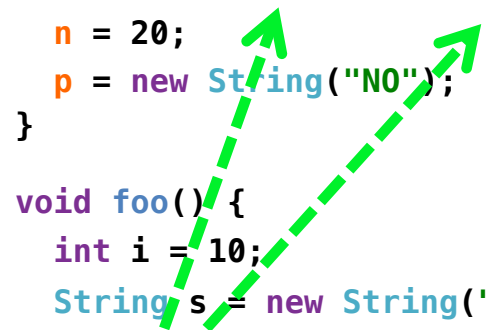
**C++**

```
class Truc {  
    void get(int n, const string * p) {  
        n = 20;  
        p = new string("NO");  
    }  
  
    void foo() {  
        int i = 10;  
        string * s = new string("YES");  
        get(i, s);  
        cout << i << " " << *s << endl;  
    }  
    ...  
};
```



**Java**

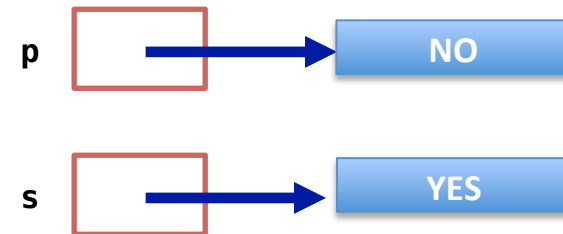
```
class Truc {  
    void get(int n, String p) {  
        n = 20;  
        p = new String("NO");  
    }  
  
    void foo() {  
        int i = 10;  
        String s = new String("YES");  
        get(i, s);  
        System.out.println(i + " " + s);  
    }  
    ...  
}
```



**Résultat : 10 YES**

- **passage par valeur => arguments inchangés**  
copie dans un seul sens !

***Solution ?***



# Récupérer des valeurs d'une fonction

```
class Truc {  
    void get(int &n, string &p) {  
        n = 20;  
        p = "NO";  
    }  
  
    void foo() {  
        int i = 10;  
        string s("YES");  
        get(i, s);  
        cout << i << " " << *s << endl;  
    }  
    ...  
};
```

C++

<... & : passage par référence

<... affiche: 20 NO

## Passage par référence avec &

- le **paramètre** est un **alias** de l'**argument**:  
si on change l'un on change l'autre

s

*Et en Java ?*

# Récupérer des valeurs d'une fonction

**LE PASSAGE PAR REFERENCE  
N'EXISTE PAS EN JAVA**

**Java : références et types de base passés par VALEUR**

Le **passage par référence** (ou similaire) existe aussi dans **C#, Pascal, Ada**, etc.

# Récupérer des valeurs d'une fonction

**C++**

```
class Truc {  
    void get(int * n, string * p) {  
        *n = 20;  
        *p = "NO";  
    }  
  
    void foo() {  
        int i = 10;  
        string * s = new string("YES");  
        get(&i, s);  
        cout << i << " " << *s << endl;  
    }  
    ...  
};
```

modifier les pointés

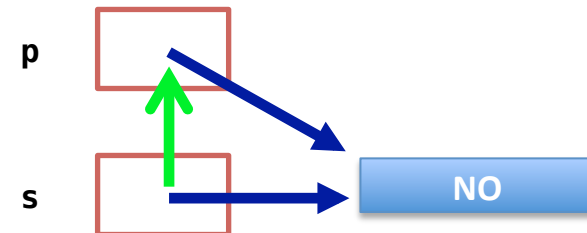
**Java**

```
class Truc {  
    void get(StringBuffer p) {  
        p.replace(0, p.length(), "NO");  
    }  
  
    void foo() {  
        StringBuffer s = new StringBufer("YES");  
        get(s);  
        System.out.println(i + " " + s);  
    }  
    ...  
}
```

## Solution : modifier les pointés

En **Java** :

- seulement avec **objets mutables**
- pas possible avec **types de base** !



# Passage des objets en C++

```
class Truc {  
    void print(int n, string p) {  
        cout << n << " " << p << endl;  
    }  
  
    void foo() {  
        int i = 10;  
        string s("YES");  
        print(i, s);  
    }  
    ...  
};
```

C++

← ..... pas de & ni de \*

## Passage par valeur

- l'objet **tout entier** est copié !  
    => problématique pour les **gros objets** comme les **conteneurs** !
- comment éviter cette **copie inutile** ?

# Passage par const référence

```
class Truc { C++
    void print(int n, const string & p) {
        cout << n << " " << p << endl;
    }

    void foo() {
        int i = 10;
        string s("YES");
        print(i, s);
    }
    ...
};
```

## Evite de copier les gros objets

- inutile pour les types de base
- indispensable pour les conteneurs !



# Retour par const référence

```
class Truc { C++
    string name_;
    const string & name() const {
        return name_;
    }

    void foo() {
        string s = name();    // OK
        string & s = name();  // INTERDIT !
        ...
    }
    ...
};
```

ne compile pas  
permettrait de modifier name\_  
romprait l'encapsulation !

## Permet d'accéder aux variables d'instance en lecture

- sans les recopier
- sans risquer de les modifier

# Références C++

## Ce sont des **alias** :

- doivent être **initialisées** : référencent toujours **la même entité**
- pas de calcul d'adresses dangereux (contrairement aux pointeurs)

```
Circle c;  
Circle & ref = c;    // ref sera toujours un alias de c
```

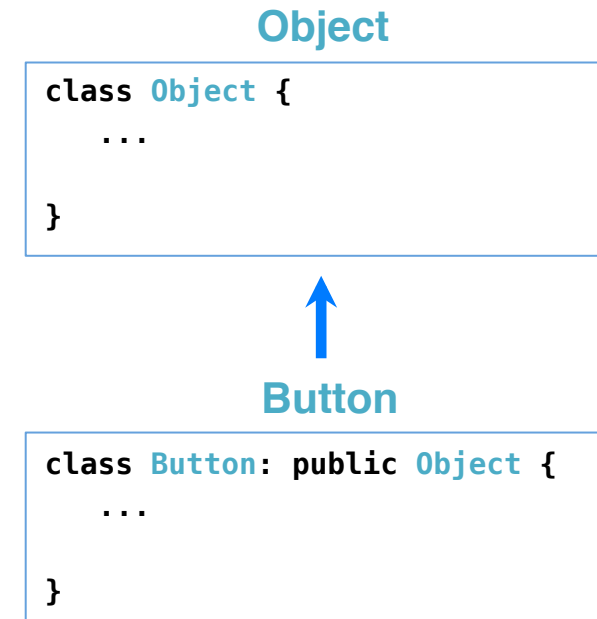
## C'est l'**objet** qui est copié :

```
Circle c1, c2;  
c1 = c2;    // copie le contenu de c2 dans c1  
  
Circle & r1 = c1;  
Circle & r2 = c2;  
  
r1 = r2;    // copie le contenu de c2 dans c1
```

# Chapitre 7 : Compléments

# Transtypage vers les superclasses

```
class Object {  
    ...  
};  
  
class Button : public Object {  
    ...  
};  
  
void foo() {  
    Object * obj = new Object();  
    Button * but = new Button();  
    obj = but;           // correct?  
    but = obj;           // correct?  
}
```



**Rappel : Correct ?**

# Transtypage vers les superclasses

```
class Object {  
    ...  
};  
  
class Button : public Object {  
    ...  
};  
  
void foo() {  
    Object * obj = new Object();  
    Button * but = new Button();  
    obj = but; <-  
    but = obj; <-  
}
```

*Tous les cèpes sont des bolets,  
mais tous les bolets ne sont pas  
des cèpes.*

OK: upcasting

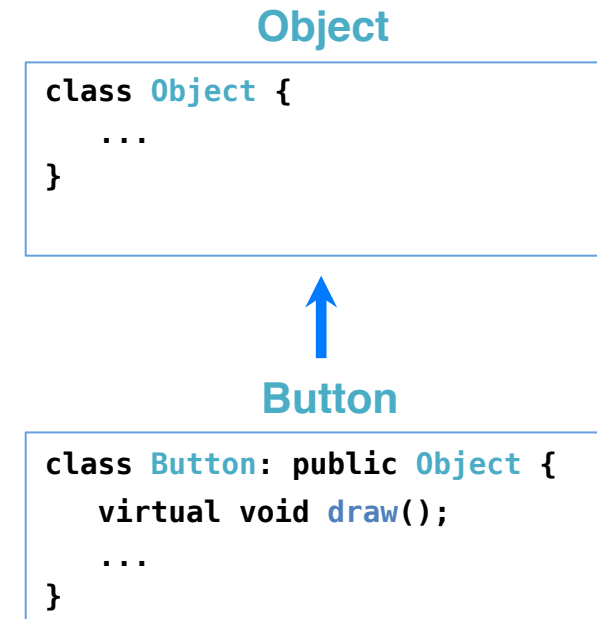
erreur de compilation:  
un Object n'est pas un Button

## Rappel : héritage

- transtypage **implicite** vers les **super-classes** (**upcasting**)
- mais **pas** vers les **sous-classes** (**downcasting**)

# Transtypage vers les sous-classes

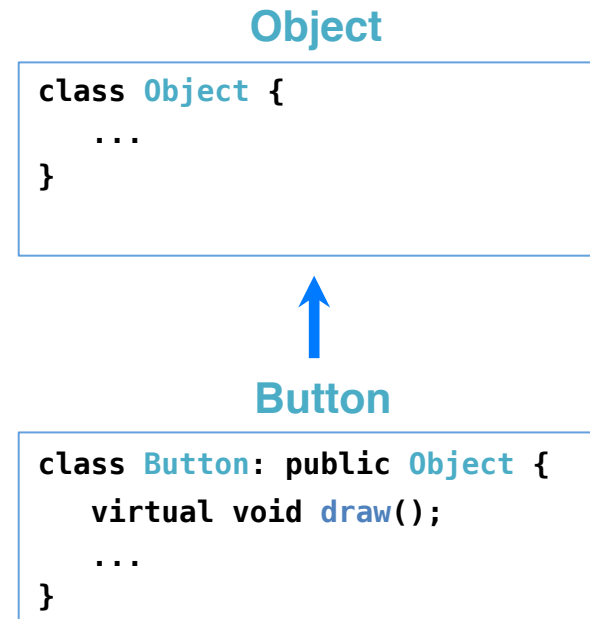
```
class Object {  
    // pas de méthode draw()  
    ...  
};  
  
class Button : public Object {  
    virtual void draw();  
    ...  
};  
  
void foo(Object * obj) {  
    obj->draw();           // correct ?  
}  
  
void bar() {  
    foo(new Button());  
}
```



**Correct ?**

# Transtypage vers les sous-classes

```
class Object {  
    // pas de methode draw()  
    ...  
};  
  
class Button : public Object {  
    virtual void draw();  
    ...  
};  
  
void foo(Object * obj) {  
    obj->draw();  
}  
  
void bar() {  
    foo(new Button());  
}
```



erreur de compilation: draw()  
pas une méthode de Object

## Que faire ?

- si on ne peut pas modifier **Object** ni la signature de **foo()**

# Transtypage vers les sous-classes

```
class Object {  
    ...  
};  
  
class Button : public Object {  
    virtual void draw();  
    ...  
};  
  
void foo(Object * obj) {  
    Button * b = (Button *) obj;  <----- cast du C : DANGEREUX !  
    b->draw();  
}  
  
void bar() {  
    foo(new Button());  
}
```

## Mauvaise solution !!!

- trompe le compilateur => **plante** si **obj** n'est pas un **Obj** !!!
- => **JAMAIS** de casts du C en C++ !!!



# Transtypage dynamique

```
class Object {  
    ...  
};  
  
class Button : public Object {  
    virtual void draw();  
    ...  
};  
  
void foo(Object * obj) {  
    Button * b = dynamic_cast<Button*>(obj);  
    if (b) b->draw();  
}  
  
void bar() {  
    foo(new Button());  
}
```

renvoie null si ce  
n'est pas un Button

## Bonne solution

- **contrôle dynamique** du type à l'exécution
- **Java** : tester avec  **instanceof** , puis faire un **cast** (ou cast + **vérifier exceptions**)

# Opérateurs de transtypage

**dynamic\_cast**<Type>(b)

- vérification du type à l'exécution : opérateur sûr

**static\_cast**<Type>(b)

- conversions de types "raisonnables" : à utiliser avec prudence

**reinterpret\_cast**<Type>(b)

- conversions de types "radicales" : à utiliser avec encore plus de prudence !

**const\_cast**<Type>(b)

- pour enlever ou rajouter const

**(Type) b** : **cast du C** : à éviter !!!!

**Note** : il y a des opérateurs spécifiques pour les **shared\_ptr** (voir la doc)

# RTTI (typeid)

```
#include <typeinfo>

void printClassName(Shape * p) {
    cout << typeid(*p).name() << endl;
}
```

## Retourne de l'information sur le type

- généralement **encodé** (mangled)

# Types incomplets et handle classes

```
#include <Widget>

class Button : public Widget {
public:
    Button(const string& name);
    void mousePressed(Event& event);
    ....
private:
    ButtonImpl * impl;
};
```

header Button.h

## Cacher l'implémentation : handle classes

- implémentation **cachée** dans `ButtonImpl`
  - `ButtonImpl` déclarée dans **header privé** `ButtonImpl.h` pas donné au client

## Références à des objets non déclarés

- `mousePressed()` dépend d'une classe `MouseEvent` déclarée ailleurs

# Types incomplets

```
#include <Widget>

class Button : public Widget {
public:
    Button(const string& name);
    void mousePressed(Event& event);
    ....
private:
    ButtonImpl * impl;
};
```

header Button.h

## Problème

- erreur de compilation: `ButtonImpl` et `MouseEvent` sont inconnus !

## Solution ?

# Types incomplets

```
#include <Widget>
#include <Event.h>
#include "ButtonImpl.h"

class Button : public Widget {
public:
    Button(const string& name);
    void mousePressed(Event& event);
    ....
private:
    ButtonImpl * impl;
};
```

header Button.h

## Mauvaise solution

- l'implémentation **n'est plus cachée** : il faut donner `ButtonImpl.h` au client !
- plein de headers qui s'incluent les uns les autres !

# Types incomplets

```
#include <Widget>
class Event;
class ButtonImpl;

class Button : public Widget {
public:
    Button(const string& name);
    void mousePressed(Event& event);
    ....
private:
    ButtonImpl * impl;
};
```

header Button.h

## Bonne solution : types incomplets

- déclarent l'**existence** d'une classe sans spécifier son **contenu** (même chose en **C** avec les **struct**)
- les variables (**event**, **impl**) doivent être des **pointeurs** ou des **références**


# Pointeurs de fonctions et lambdas

```
class Data {
public:
    std::string firstName, lastName;
    int id, age;
    ....
};

using DataList = std::list< Data* >;

class DataBase {
public:
    DataList search( std::function<bool(const Data&)> test ) const;
    ....
};
```

test = fonction  
appelée par search()



## Problème

- **DataBase** contient des **Data**
- **search()** renvoie les **Data** qui vérifient la fonction **test**
- **test** = **pointeur d'une fonction** :
  - qui prend un **Data&** en argument et renvoie un **bool**



# Pointeurs de fonctions

```
class Data {
public:
    std::string firstName, lastName;
    int id, age;
    ....
};

using DataList = std::list< Data* >;

class DataBase {
public:
    DataList search(std::function<bool(const Data&)> test) const;
    ....
};
```

## Exemple

```
bool test10(const Data& d) {return d.age > 10;}

void foo(const DataBase& base) {
    DataList res = base.search(test10);
    ....
}
```

## Limitation

- il faut écrire une fonction pour **chaque age** ! (et pour tous les autres critères)

# Lambdas

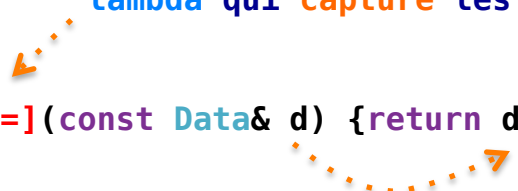
Plus puissant !

```
class Data {
public:
    std::string firstName, lastName;
    int id, age;
    ....
};

using DataList = std::list< Data* >;

class DataBase {
public:
    DataList search(std::function<bool(const Data&)> test) const;
    ....
};
```

```
void foo(const DataBase& base) {    lambda qui capture les vars de foo()
    int age = 10;
    DataList res = base.search( [=](const Data& d) {return d.age > age;} );
}
```



**Lambdas = fonctions anonymes qui capturent les variables**

- la lambda **possède une copie** des variables de `foo()`

# Lambdas et capture de variables

```
void foo(const DataBase& base) {  
    int age = 10;  
    DataList res = base.search( [age](const Data& d) {return d.age > age;} );  
}
```

lambda qui capture seulement age

## Options

- `[]` : capture **rien**
- `[=]` : capture par **valeur** (y compris **this** dans un objet)
- `[&]` : capture par **référence** (pour **modifier** les variables)
- `[age]` : capture **age** par **valeur** (age est copié)
- type de retour **implicite** sinon écrire :

```
[age](const Data& d) -> bool {return d.age > age;}
```

## Existent aussi en Python, Java 8, etc.

- simplifient considérablement le code !

# Types de pointeurs de fonctions

## Plusieurs solutions

- pointeurs de **fonctions**
- pointeurs de **méthodes**
- pointeurs de **fonctions généralisés**
  - peuvent pointer n'importe quel type de fonction ou une **lambda**

## Exemple

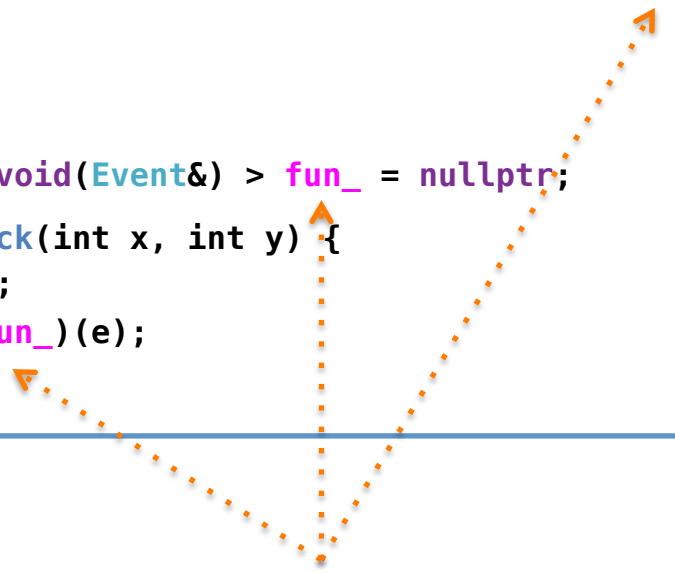
- fonctions de **callback** des boites à outils graphiques

```
void doIt(Event&) {  
    cout << "Done!" << endl;  
}  
  
void foo() {  
    Button * btn = new Button("OK");  
    btn->addCallback(doIt);  
}
```

**doIt()** sera appelée  
quand on cliquera le bouton

# Pointeurs de fonctions généralisés

```
class Button : public Widget {
public:
    void addCallback(std::function< void(Event&) > fun) {
        fun_ = fun;
    }
protected:
    std::function< void(Event&) > fun_ = nullptr;
    void callCallback(int x, int y) {
        Event e(x,y);
        if (fun_) (fun_)(e);
    }
};
```



```
void doIt(Event&) {
    cout << "Done!" << endl;
}

void foo() {
    Button * btn = new Button("OK");
    btn->addCallback(doIt);
}
```

pointeur de fonction généralisé (C++11)

# Pointeurs de fonctions non-membres

Pour les fonctions :

- **non-membres**
- **ou static**

```
void doIt(Event&) {  
    cout << "Done!" << endl;  
}  
  
void foo() {  
    Button * btn = new Button("OK");  
    btn->addCallback(doIt);  
}
```

```
class Button : public Widget {  
public:  
    void addCallback( void (*fun)(Event&) ) {  
        fun_ = fun;  
    }  
protected:  
    void (*fun_)(Event&) >= nullptr;  
    void callCallback(int x, int y) {  
        Event e(x,y);  
        if (fun_) (fun_)(e);  
    }  
};
```

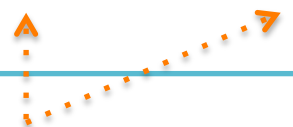


pointeur de fonction (comme en C)  
(noter l'\*)

# Pointeurs de méthodes

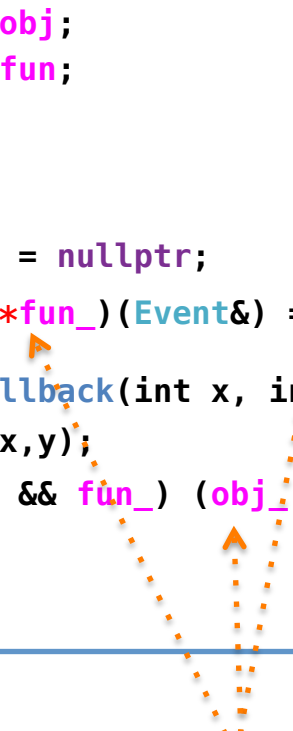
Pour les **méthodes** d'instances

```
class Truc {  
    string result;  
public:  
    void doIt(Event& e) {  
        cout << "Result:" << result << endl;  
    }  
};  
  
void foo() {  
    Truc * truc = new Truc();  
    Button * btn = new Button("OK");  
    btn->addCallback(truc, &Truc::doIt);  
}
```



on passe l'objet et la méthode en argument  
noter le &

```
class Button : public Widget {  
public:  
    void addCallback(Truc* obj,  
                    void(Truc::*fun)(Event&)){  
        obj_ = obj;  
        fun_ = fun;  
    }  
  
protected:  
    Truc * obj_ = nullptr;  
    void(Truc::*fun_)(Event&) = nullptr;  
    void callCallback(int x, int y) {  
        Event e(x,y);  
        if (obj_ && fun_) (obj_ -> *fun_)(e);  
    }  
};
```



pointeur de méthode de la classe Truc  
(noter l'\*)

# Foncteurs

- l'objet est considéré comme une **fonction**
- plus besoin de passer l'objet en argument !
- il suffit de définir **operator()**

```
class Truc {  
    string result;  
public:  
    void operator()(Event& e) {  
        cout << "Result:" << result << endl;  
    }  
};  
  
void foo() {  
    Truc * truc = new Truc();  
    Button * btn = new Button("OK");  
    btn->addCallback(truc);  
}
```

on ne passe que l'objet en argument

```
class Button : public Widget {  
public:  
    void addCallback(Truc* obj){  
        obj_ = obj;  
    }  
  
protected:  
    Truc * obj_ = nullptr;  
  
    void callCallback(int x, int y) {  
        Event e(x,y);  
        if (obj_) (*obj_)(e);  
    }  
};
```

l'objet est considéré comme une fonction !



# Surcharge des opérateurs

```
#include <string>

string s = "La tour";
s = s + " Eiffel";
s += " est bleue";
```

```
class string {
    friend string operator+(const string&, const char*);
    string& operator+=(const char*);
    ....
};
```

## Possible pour presque tous les opérateurs

- sauf `::` `.` `.*` `?`
- la priorité est inchangée
- à utiliser avec discernement !
- existe en **C#**, **Python**, **Ada**... (mais pas **Java**)

```
#include <vector>

vector tab(3);
tab[0] = tab[1] + tab[2];
```

`operator[]`

## Permet en particulier de redéfinir:

- `operator[]`, `operator()`
- `operator*`, `operator->`
- **new** et **delete**
- conversions de types

```
class Number {
    Number & operator++(); // prefixe
    Number operator++(int); // postfixe
};
```

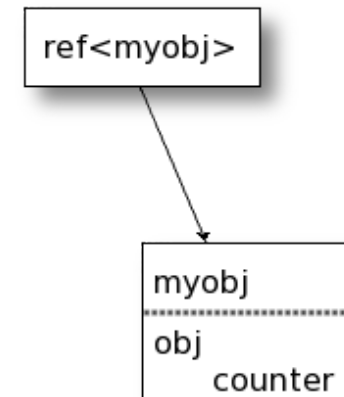
`operator++`

# Exemple : smart pointers intrusifs

```
class Shape {
public:
    Shape() : counter(0) {}
private:
    long counter;
    friend void intrusive_ptr_add_ref(Pointable* p);
    friend void intrusive_ptr_release(Pointable* p);
    friend long intrusive_ptr_get_count(Pointable* p);
};

inline void intrusive_ptr_add_ref(Shape* p) {
    ++(p->counter);
}

inline void intrusive_ptr_release(Shape* p) {
    if (--(p->counter) == 0) delete p;
}
```



## Principe

- la classe de base possède un compteur de références
- les smart pointers détectent les affectations et modifient le compteur

# Exemple : smart pointers intrusifs

```
template <class T>
class intrusive_ptr {
    T* p;
public:
    intrusive_ptr(T* obj) : p(obj) {if (p != NULL) intrusive_ptr_add_ref(p);}
    ~intrusive_ptr() {if (p) intrusive_ptr_release(p);}
    intrusive_ptr& operator=(T* obj) {...}
    T* operator->() const {return p;}          // la magie est là !
    T& operator*() const {return *p;}
    .....
};

void foo() {
    intrusive_ptr<Shape> ptr = new Circle(0, 0, 50);
    ptr->setX(20);      // fait ptr.p->setX(20)
}                      // ptr est détruit car dans la pile => appelle destructeur
                      // => appelle intrusive_ptr_release()
```

## Le smart pointer

- encapsule un raw pointer
- surcharge le **copy constructor**, et les **opérateurs =** , **->** et **\***

# Exceptions

```
class MathErr {};  
  
class Overflow : public MathErr {};  
  
struct Zerodivide : public MathErr {  
    int x;  
    Zerodivide(int x) : x(x) {}  
};  
  
void foo() {  
    try {  
        int z = calcul(4, 0)  
    }  
    catch(Zerodivide & e) { cerr << e.x << "divisé par 0" << endl; }  
    catch(MathErr)        { cerr << "erreur de calcul" << endl; }  
    catch(...)            { cerr << "autre erreur" << endl; }  
}  
  
int calcul(int x, int y) {  
    return divide(x, y);  
}  
  
int divide(int x, int y) {  
    if (y == 0) throw Zerodivide(x);    // throw leve l'exception  
    else return x / y;  
}
```

# Exceptions

## But : faciliter le traitement des erreurs

- remontent dans la pile des appels de fonctions
- jusqu'à un point de contrôle

## Avantages

- gestion **centralisée** et **systématique** des erreurs
  - évitent d'avoir tester et propager des **codes d'erreurs** dans une myriade de fonctions

## Inconvénient

- peuvent rendre le flux d'exécution **difficile à comprendre** si on en abuse
  - => à utiliser à bon escient et avec modération !

```
void foo() {  
    try {  
        int z = calcul(4, 0)  
    }  
    catch(Zerodivide & e) {...}  
    catch(MathErr)      {...}  
    catch(...)          {...}  
}
```

# Exceptions

## Différences entre C++ et Java

- en **C++** on peut renvoyer **ce qu'on veut** (pas seulement des objets)
- en **Java** les fonctions doivent **spécifier les exceptions**

## Spécification d'exceptions de Java

```
int divide(int x, int y) throws Zerodivide, Overflow {...}    // Java  
int divide(int x, int y);    // C++
```

- n'existent pas en **C#**, obsolètes en **C++**
- compliquent le code et entraînent des **limitations** :
  - en **Java** une méthode redéfinie dans une sous-classe **ne peut pas spécifier** de nouvelles exceptions !

# Exceptions

## Exceptions standards

- `exception` : classe de base ; header : `<exception>`
- `runtime_error`
- `bad_alloc`, `bad_cast`, `bad_typeid`, `bad_exception`, `out_of_range` ...

## Handlers

- `set_terminate()` et `set_unexpected()` spécifient ce qui se passe en dernier recours

## Redéclenchement d'exceptions

```
try {  
    ..etc..  
}  
  
catch (MathErr& e) {  
    if (can_handle(e)) {  
        ..etc..  
        return;  
    }  
    else {  
        ..etc..  
        throw;           // relance l'exception  
    }  
}
```

# Assertions

```
#include <Square.h>
#include <assert.h>

void changeSize(Square * obj, unsigned int size) {
    assert(obj);
    obj->setWidth(size);
    assert(obj->getWidth() == obj->getHeight());
}
```

← ..... précondition

← ..... postcondition

## Pour faire des tests en mode débog

- en mode **débug** : **assert()** **aborte** le programme si valeur = 0
- en mode **production** : définir la macro **NDEBUG** et **assert()** ne fait plus rien
  - option de compilation `-DNDEBUG`
  - ou `#define NDEBUG` avant `#include <assert.h>`

## Remarques

- il est **dangereux** de ne faire aucun test en mode production (**exceptions** faites pour cela)
- préférer les **tests unitaires** (ex : [GoogleTest](#), [CppTest](#), [CppUnit](#))



# Une source d'erreur fréquente...

## Attention

- le pointeur peut être **nul** !
- ca arrive **souvent** ...

```
void changeSize(Square * obj, unsigned int size) {  
    obj->setWitdth(size);  
}
```

## Mieux !

- lancer une **exception**
- c'est ce que fait **Java**

```
void changeSize(Square * obj, unsigned int size) {  
    if (obj) obj->setWitdth(size);  
    else throw NullPointerException("changeSize");  
}
```

## Encore mieux !

- une **référence C++** ne peut pas être nulle
- mais ne **pas** faire :

```
void foo(Square * obj) {  
    changeSize(*obj, 200)  
}
```

← ..... DANGER : tester que obj n'est pas nul !

# Une source d'erreur fréquente...

```
#include <string>
#include <stdexcept>

struct NullPointer : public runtime_error {
    explicit NullPointer(const std::string & what)
        : runtime_error("Error: Null pointer in" + what) {}
    explicit NullPointer(int line, const char * file)
        : runtime_error("Error: Null pointer at line "+to_string(line)+" of file: "+file) {}
};

#define CheckPtr(obj) (obj ? obj : throw NullPointer(__LINE__,__FILE__),obj)
```

```
void changeSize(Square * obj, unsigned int size) {
    if (obj) obj->setWidth(size);
    else throw NullPointer("changeSize");
}
```

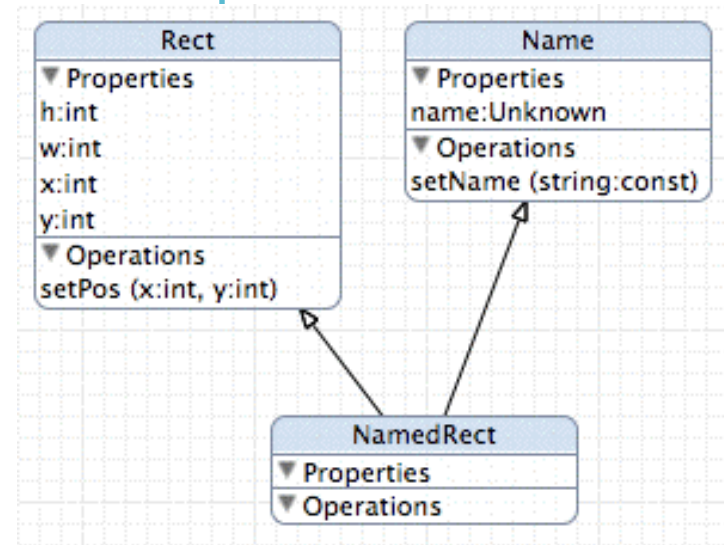
```
void changeSize(Square * obj, unsigned int size) {
    CheckPtr(obj)->setWidth(size);
}
```

# Héritage multiple

```
class Rect {
    int x, y, w, h;
public:
    virtual void setPos(int x, int y);
    ....
};

class Name {
    string name;
public:
    virtual void setName(const string&);
    ....
};

class NamedRect : public Rect, public Name {
public:
    NamedRect(const string& s, int x, int y, int w, int h)
        : Rect(x,y,w,h), Name(s) {}
};
```

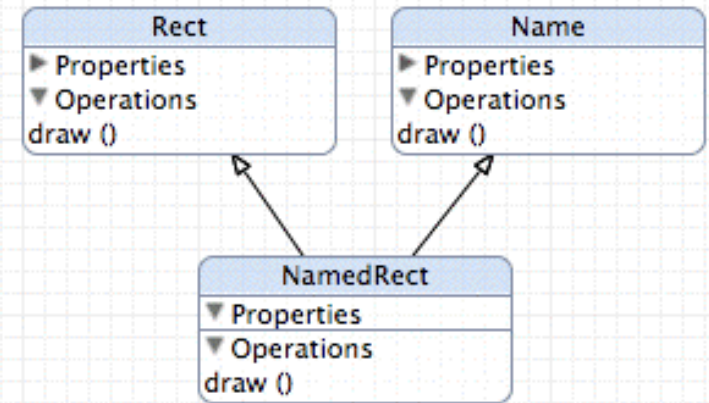


## Principe

- la classe hérite des variables et méthodes de **toutes** ses superclasses

# Collisions de noms

```
class Rect {  
    int x, y, w, h;  
public:  
    virtual void draw();  
    ....  
};  
  
class Name {  
    string x;  
public:  
    virtual void draw();  
    ....  
};  
  
class NamedRect : public Rect, public Name {  
public:  
    ....  
};
```



Variables ou méthodes ayant le **même nom**  
dans les superclasses

=> il faut les **préfixer** pour pouvoir y accéder

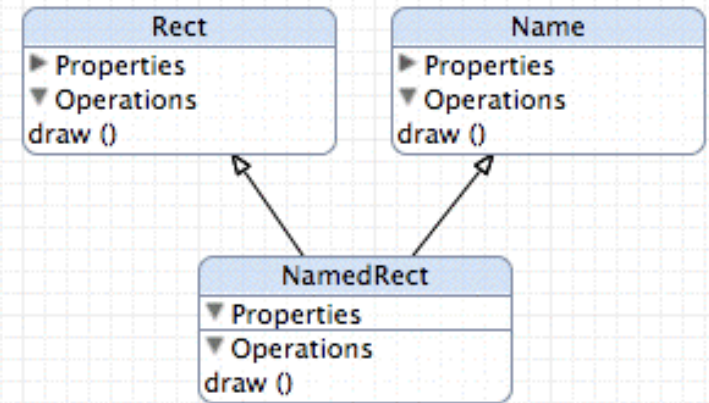
```
NamedRect * p = ...;  
p->draw();           // ERREUR!  
p->Rect::draw();     // OK  
p->Name::draw();     // OK
```

# Collisions de noms

```
class Rect {
    int x, y, w, h;
public:
    virtual void draw();
    ....
};

class Name {
    string x;
public:
    virtual void draw();
    ....
};

class NamedRect : public Rect, public Name {
public:
    void draw() override {
        Rect::draw();
        Name::draw();
    }
    // ou bien
    using Rect::draw();
    ...
};
```

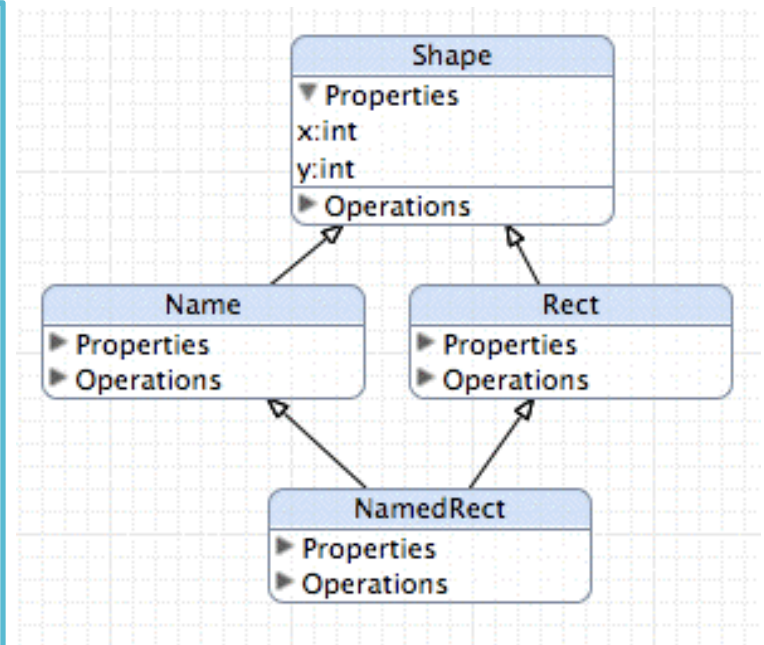


## Solutions

- **redéfinir** les méthodes concernées
- ou
- **choisir** la méthode héritée avec **using**

# Héritage en diamant

```
class Shape {  
    int x, y, w, h;  
public:  
    virtual void draw();  
    ....  
};  
  
class Rect : public Shape {  
    ....  
};  
  
class Name : public Shape {  
    ....  
};  
  
class NamedRect : public Rect, public Name {  
public:  
    ....  
};
```

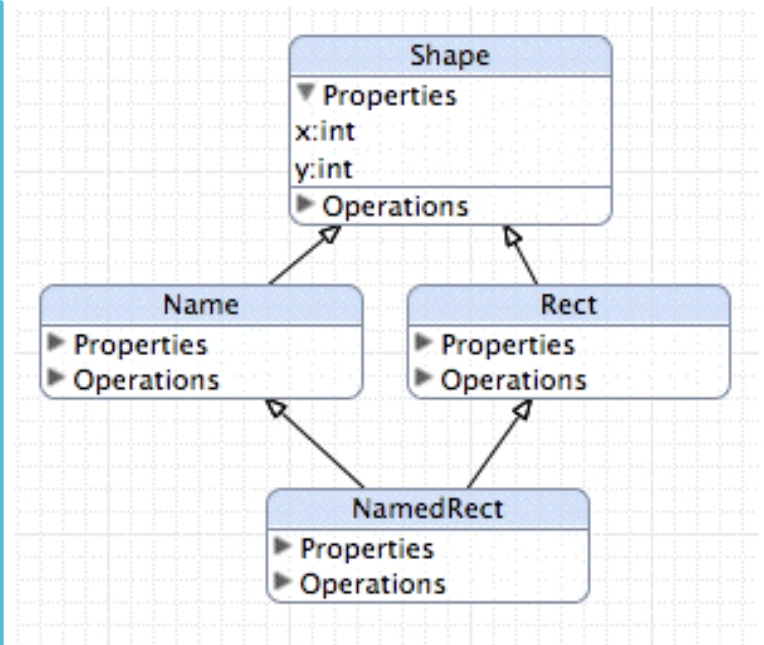


## Problème

- la classe de base (**Shape**) est **dupliquée** car elle est héritée des **deux côtés**
- fait **rarement sens** !

# Héritage en diamant

```
class Shape {  
    int x, y, w, h;  
public:  
    virtual void draw();  
    ....  
};  
  
class Rect : public Shape {  
    ....  
};  
  
class Name : public Shape {  
    ....  
};  
  
class NamedRect : public Rect, public Name {  
public:  
    ....  
};
```

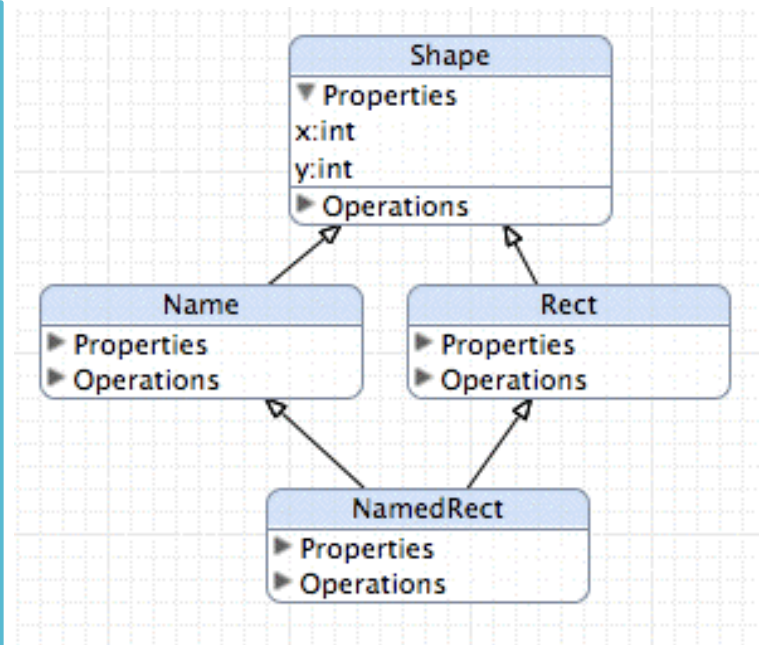


## Solution 1 : pas de variables

- **que des méthodes** dans les **classes de base**
- c'est ce que fait **Java 8** avec les **default methods** des **interfaces**

# Héritage virtuel

```
class Shape {  
    int x, y, w, h;  
public:  
    virtual void draw();  
    ....  
};  
  
class Rect : public virtual Shape {  
    ....  
};  
  
class Name : public virtual Shape {  
    ....  
};  
  
class NamedRect : public Rect, public Name {  
public:  
    ....  
};
```



## Solution 2 : héritage sans duplication avec **virtual**

- un peu plus **coûteux** en mémoire et en temps
- ne pas faire de **casts** mais des **dynamic\_cast**



# Classes imbriquées

```
class Car : public Vehicle {  
    class Door {  
    public:  
        virtual void paint();  
        ....  
    };  
    Door leftDoor, rightDoor;  
    string model, color;  
public:  
    Car(string model, string color);  
    ...  
};
```

← ..... classe imbriquée

## Technique de **composition** souvent préférable à l'**héritage multiple**

- évite problèmes précédents
- l'**héritage multiple** peut entraîner des **dépendances complexes**

# Classes imbriquées (2)

```
class Car : public Vehicle {  
    class Door {  
    public:  
        virtual void paint();  
        ....  
    };  
    Door leftDoor, rightDoor;  
    string model, color;  
public:  
    Car(string model, string color);  
    ...  
};
```

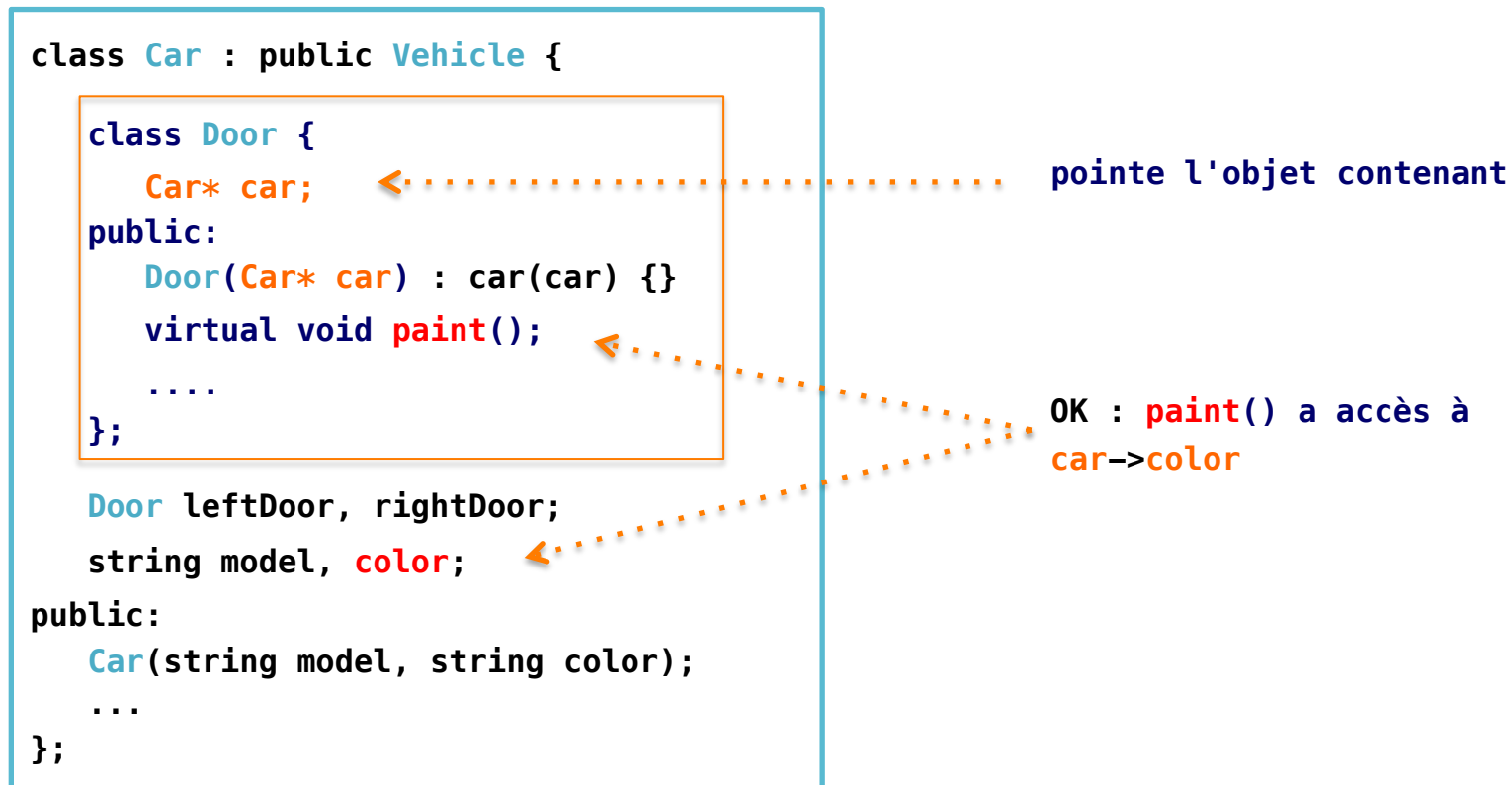
problème: `paint()` n'a pas accès à `color`

## Java

- les méthodes des **classes imbriquées** ont **automatiquement accès** aux variables et méthodes de la **classe contenant**

## Pas en C++ !

# Classes imbriquées (3)



## Solution (rappel)

- pour « **envoyer un message** » à un objet il faut son **adresse**

# Sérialisation

## But

- transformer l'**information en mémoire** en une **représentation externe non volatile** (et vice-versa)

## Cas d'usage

- **persistance** : sauvegarde sur / relecture depuis un fichier
- **transport réseau** : communication de données entre programmes

## Implémentation

- **Java** : en **standard**, mais spécifique à **Java**
- **C/C++** : pas standard (pour les objets) mais diverses extensions :
  - **Cereal, Boost, Qt, Protocol Buffers** (Google), OSC ...

# Sérialisation binaire vs. texte

## Sérialisation binaire

- objets stockés en **binaire**
- codage **compact** mais pas **lisible** par un humain
- **pas compatible** d'un ordinateur à l'autre sauf si **format standardisé**
  - exemple: **Protocol Buffers**
  - raisons :
    - little/big endian
    - taille des nombres
    - alignement des variables

## Sérialisation au format texte

- tout est converti en **texte**
- prend **plus de place** mais **lisible** et un peu plus **coûteux** en CPU
- **compatible** entre ordinateurs
- il existe des **formats standards**
  - **JSON**
  - **XML/SOAP**
  - etc.

# Ecriture/lecture d'objets (format texte)

**Principe** : définir des fonctions d'écriture **polymorphiques**

```
#include <iostream>

class Vehicle {
public:
    virtual void write(std::ostream & f);    <..... ne pas oublier virtual !
    virtual void read(std::istream & f);
    ....
};

class Car : public Vehicle {
    string model;
    int power;
public:
    void write(std::ostream & f) override {
        Vehicule::write(f);    <..... chaîner les méthodes
        f << model << '\n' << power << '\n';
    }

    void read(std::istream & f) override {
        Vehicule::read(f);
        f >> model >> power;
    }
    ....
};
```

**Fichier:**

```
whatever\n    <..... écrit par Véhicule
whatever\n
Ferrari 599 GTO\n    <... écrit par Car
670\n
whatever\n
whatever\n
Smart Fortwo\n
71\n
```

# Lecture avec espaces

```
void read(std::istream & f) override {  
    Vehicule::read(f);  
    f >> power >> model;  
}
```

## Problème

>> s'arrête au **premier espace** (' ', '\n', '\r', '\t', '\v', '\f')

## Solution

**getline()** : lit **toute la ligne** (ou jusqu'à un **certain caractère**)

```
void read(std::stream & f) override {  
    Vehicule::read(f);  
    getline(f, model);  
    string s;  
    getline(f, s);  
    model = stoi(s);  
}
```

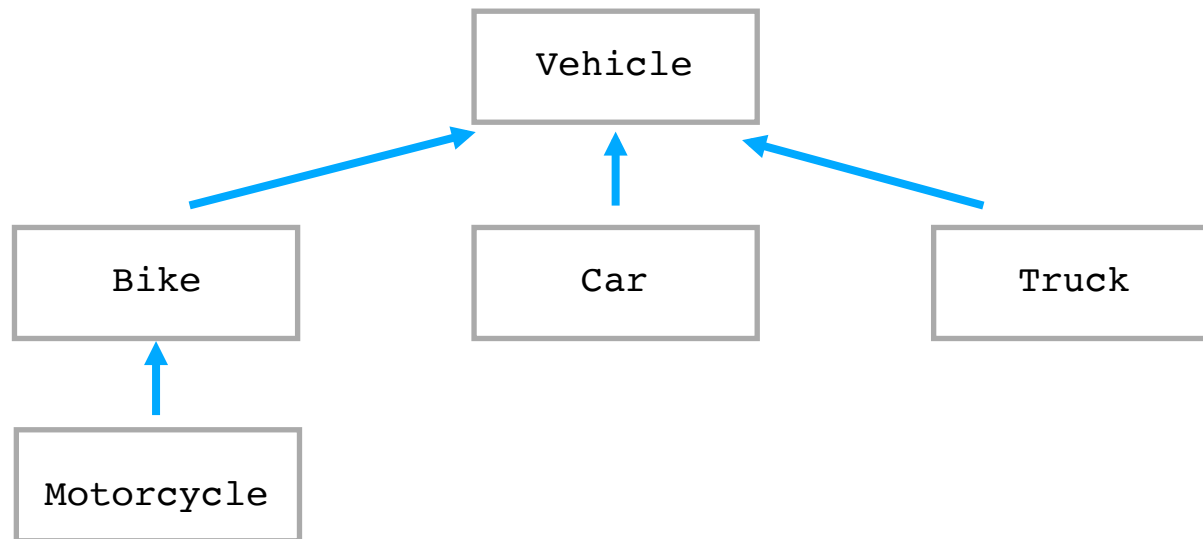
### Fichier:

```
whatever\n  
whatever\n  
Ferrari 599 GTO\n  
670\n  
whatever\n  
whatever\n  
Smart Fortwo\n  
71\n
```

# Classes polymorphes

## Problème

- les objets ne sont **pas tous du même type** (mais dérivent d'un même type)
- => pour pouvoir les **lire** il faut connaître leur **classe**





# Classes polymorphes

## Solution

écrire le **nom de la classe**  
des objets dans le fichier

## En écriture :

- 1) écrire le **nom de la classe**
- 2) écrire les **attributs** de l'objet

## En lecture :

- 1) lire le **nom de la classe**
- 2) **créer l'objet** correspondant
- 3) lire ses **attributs**

```
#include <iostream>

class Vehicle {
public:
    virtual std::string classname() const = 0;
    // ... le reste est identique
};

class Car : public Vehicle {
public:
    std::string classname() const override {
        return "Car";
    }
    // ... le reste est identique
};
```

# Sauver des objects

```
#include <iostream>
#include <fstream>
```

passer vecteur par **référence**  
sinon il est **recopié** !



```
bool saveAll(const std::string & filename, const std::vector<Vehicle *> & objects) {
```

```
    std::ostream f(filename);
```

```
    if (!f) {
        cerr << "Can't open file " << filename << endl;
        return false;
    }
```

vérifie que le  
fichier est **ouvert**

```
    for (auto it : objects) {
        f << it->classname();
        it->write(f);
        if (f.fail()) {
            cerr << "Write error in " << filename << endl;
            return false;
        }
    }
    return true;
}
```

écrire la **classe**  
puis les **attributs**

**erreur** d'écriture

# Lire des objets

passer vecteur par référence  
sinon il est recopié !



```
bool readAll(const std::string & filename, std::vector<Vehicle *> & objects) {  
    std::istream f(filename);  
    if (!f) {  
        cerr << "Can't open file " << filename << endl;  
        return false;  
    }  
  
    while (f) {  
        std::string classname;  
        getline(f, classname);  
        Vehicle * obj = createVehicle(classname);  
        obj->read(f);  
        if (f.fail()) {  
            cerr << "Read error in " << filename << endl;  
            delete obj;  
            return false;  
        }  
        else objects.push_back(obj);  
    }  
    return true;  
}
```

tant que pas en fin de  
fichier et pas d'erreur

factory qui sert  
à créer les objets

erreur de lecture

# stringstream

## Flux de caractères

- fonctionne de la même manière que `istream` et `ostream`

```
#include <string>
#include <iostream>
#include <sstream>
void foo(const string& str) {
    std::stringstream ss(str);
    int power = 0;
    string model;
    ss >> power >> model;
    cout << "Vehicle: power:" << power << " model: " << model << endl;

    Vehicle * obj = new Car();
    obj->read(ss);
}

foo("670 \n Ferrari-599-GT0");
```

# Compléments

## Améliorations

- meilleur traitement des erreurs
- gérer les pointeurs et les conteneurs  
=> utiliser **Boost**, **Cereal**, etc.

## JSON

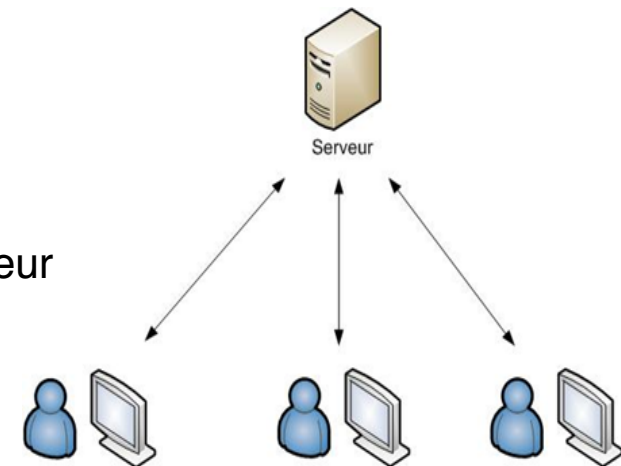
- **JavaScript Object Notation**
- commode pour les échanges textuels

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

# Client / serveur

## Cas typique

- **un** serveur de calcul
- **des** interfaces utilisateur pour interagir avec le serveur
- cas du TP INF224



source:  
maieutapedia.org

## Principe

- le client émet une requête, obtient une réponse, et ainsi de suite
- dialogue **synchrone** ou **asynchrone**

# Client / serveur

## Dialogue synchrone

- le client émet une **requête** et **bloque** jusqu'à réception de la **réponse**
- le plus simple à implémenter
- problématique si la réponse met du temps à arriver ou en cas d'erreur

## Dialogue asynchrone

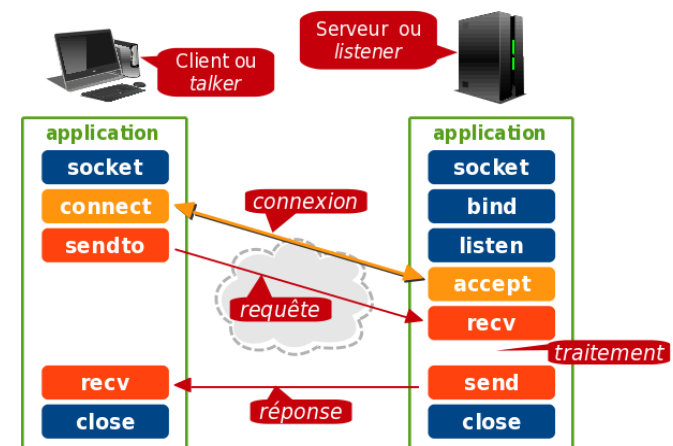
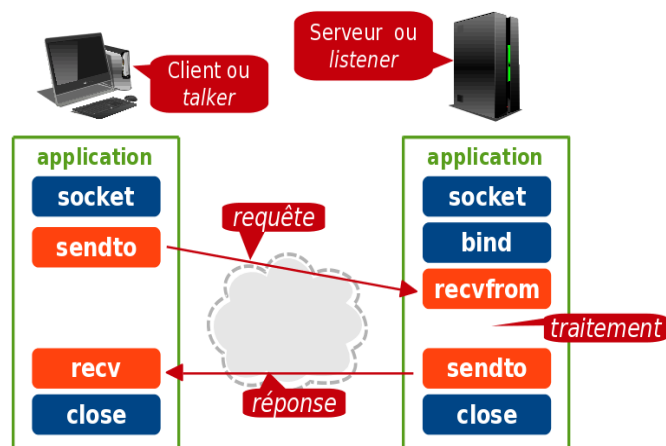
- le client vaque à ses occupations après l'émission de la requête
- quand la réponse arrive une **fonction de callback** est activée
- exemples :
  - thread qui attend la réponse
  - **XMLHttpRequest** de JavaScript



# Sockets

## Principe

- **canal de communication bi-directionnel** entre 2 programmes
- programmes éventuellement sur des machines différentes
- divers protocoles, **UDP** et **TCP** sont les plus courants



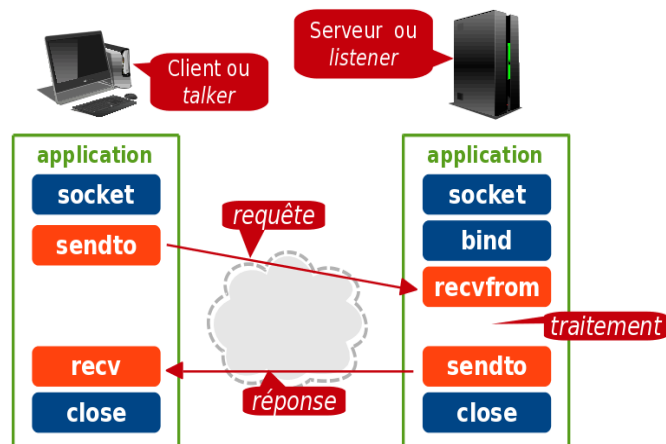
source: [inetdoc.net](http://inetdoc.net)



# Sockets

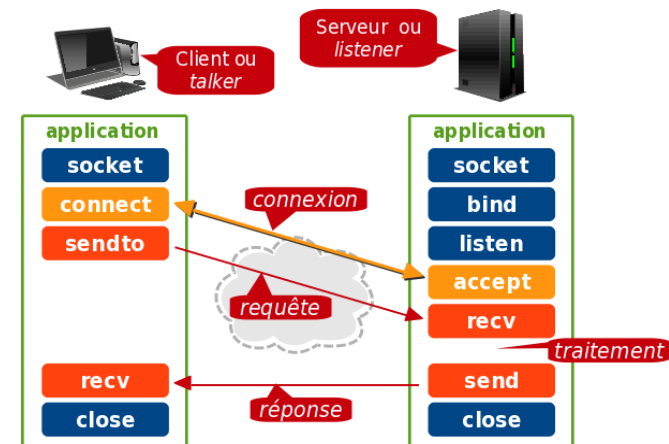
## Protocole UDP

- **Datagram sockets** (type SOCK\_DGRAM)
- protocole "léger", «**non connecté**»
- peu coûteux en ressources
- rapide mais des paquets peuvent être **perdus** ou arriver dans le **désordre**



## Protocole TCP

- **Stream sockets** (type SOCK\_STREAM)
- protocole **connecté**
- un peu plus coûteux en ressources
- flux d'octets entre 2 programmes, **pas** de paquets perdus et toujours dans l'**ordre**
- ex : HTTP, TP INF224



source: inetdoc.net

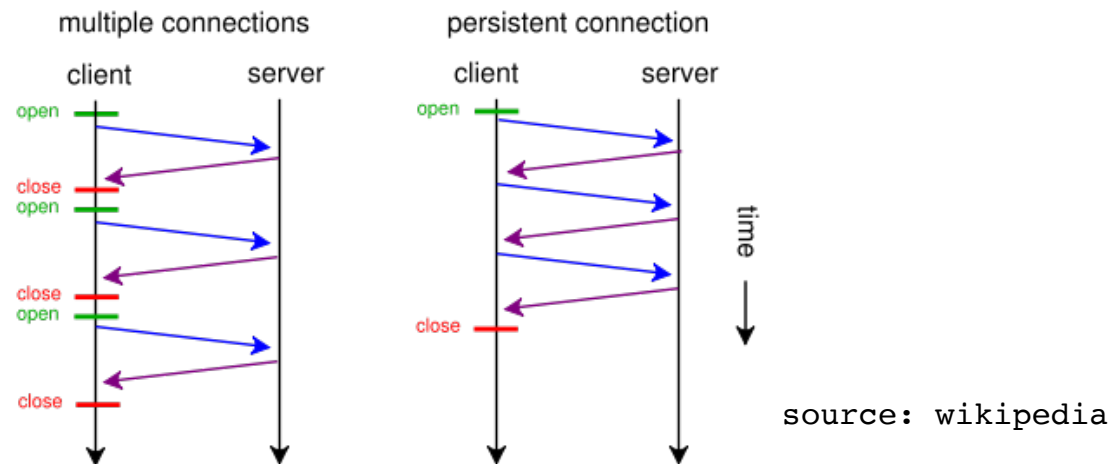
# Sockets

## Connexion TCP persistante

- le client est **toujours** connecté au serveur
- solution utilisée dans le TP

## Connexion TCP non persistante

- le client n'est connecté que **pendant l'échange de messages**
- moins rapide, moins de flexibilité
- mais consomme moins de ressources côté serveur



# Mémoire et sécurité

```
#include <stdio.h>           // en langage C
#include <stdbool.h>
#include <string.h>

#define CODE_SECRET "1234"

int main(int argc, char**argv)
{
    bool is_valid = false;
    char code[5];

    printf("Enter password: ");
    scanf("%s", code);

    if (strcmp(code, CODE_SECRET) == 0)
        is_valid = true;

    if (is_valid)
        printf("Welcome dear customer ;-)\n");
    else
        printf("Invalid password !!!\n");

    return 0;
}
```

## Questions :

**Que fait ce programme ?**

**Est-il sûr ?**

# Mémoire et sécurité

```
#include <stdio.h>           // en langage C
#include <stdbool.h>
#include <string.h>

#define CODE_SECRET "1234"

int main(int argc, char**argv)
{
    bool is_valid = false;
    char code[5];

    printf("Enter password: ");
    scanf("%s", code);

    if (strcmp(code, CODE_SECRET) == 0)
        is_valid = true;

    if (is_valid)
        printf("Welcome dear customer ;-)\n");
    else
        printf("Invalid password !!!\n");

    printf("Adresses: %p %p %p %p\n",
           code, &is_valid, &argc, argv);

    return 0;
}
```

Avec LLVM sous MacOSX 10.7.1 :

**Enter password: 111111**  
**Welcome dear customer ;-)**

Adresses:

0x7fff5fbff98a 0x7fff5fbff98f  
0x7fff5fbff998 0x7fff5fbff900

**Débordement de chaînes :**  
technique typique de **piratage**  
**informatique**

# Mémoire et sécurité

```
#include <iostream>           // en C++
#include <string>

static const string CODE_SECRET{"1234"};

int main(int argc, char**argv)
{
    bool is_valid = false;
    string code;

    cout << "Enter password: ";
    cin >> code;

    if (code == CODE_SECRET) is_valid = true;
    else is_valid = false;

    if (is_valid)
        cout << "Welcome dear customer ;-)\n";
    else
        cout << "Invalid password !!!\n";

    return 0;
}
```

**string au lieu de char\***

**pas de débordement :  
taille allouée automatiquement**

**rajouter une clause else  
ne mange pas de pain  
et peut éviter des erreurs**

# Mélanger C et C++

## De préférence

- tout compiler (y compris les .c) avec compilateur C++

## Si on mélange compilation en C et compilation en C++

- édition de liens avec compil C++
- main() doit être dans un fichier C++
- une fonction C doit être déclarée comme suit dans C++

```
extern "C" void foo(int i, char c, float x);
```

ou

```
extern "C" {  
    void foo(int i, char c, float x);  
    int  goo(char* s, char const* s2);  
}
```

# Mélanger C et C++

## Dans un header C

- pouvant indifféremment être inclus dans un .c ou un .ccp, écrire :

```
#ifdef __cplusplus  
extern "C" {  
#endif
```

```
void foo(int i, char c, float x);  
int  goo(char* s, char const* s2);
```

```
#ifdef __cplusplus  
}  
#endif
```

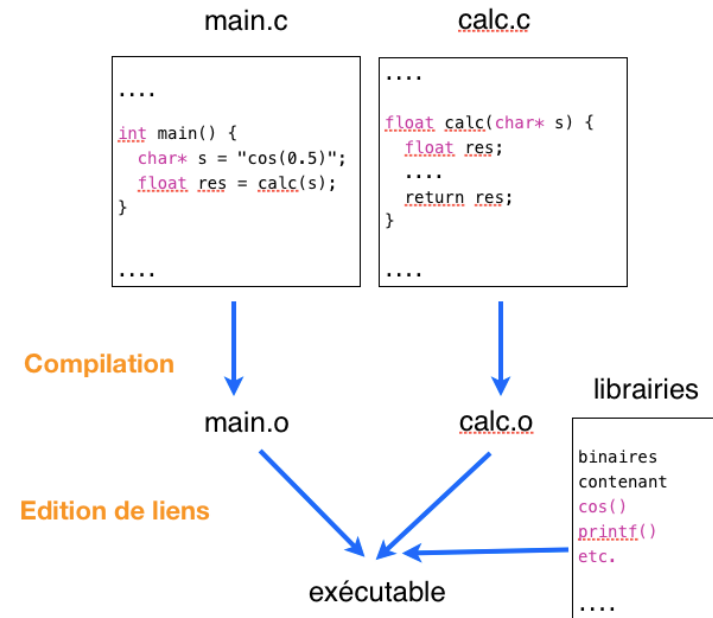
# Librairies statiques et dynamiques

## Librairies statiques

- code binaire **inséré dans l'exécutable** à la compilation
- extension .a (Unix)

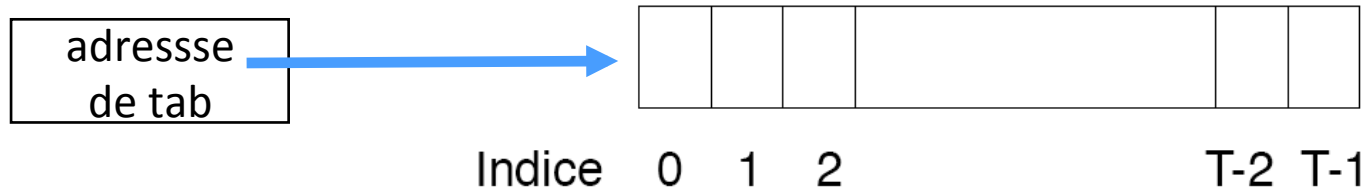
## Librairies dynamiques

- code binaire chargé **dynamiquement** à l'exécution
- .dll (Windows), .so (Linux), dylib (Mac)
- avantages :
  - programmes **moins gros** et **plus rapides** (moins de swap si DLL partagée)
- inconvénient :
  - nécessite la présence de la **DLL** (cf. licences et versions)  
(cf. variable `LD_LIBRARY_PATH` (ou équivalent) sous Unix)





# Arithmétique des pointeurs



## Tableaux

```
int tab[10];
```

```
tab[k] == *(tab + k)
```

```
&tab[k] == tab + k
```

```
// valeur du kième élément du tableau
```

```
// adresse du kième élément du tableau
```

## Pointeurs : même notation !

```
int* p = tab;
```

```
p[k] == *(p + k)
```

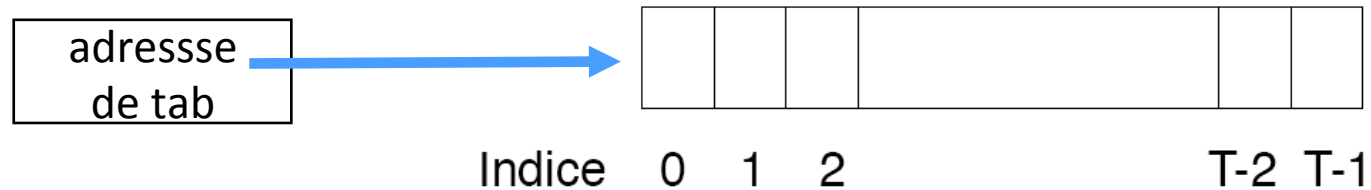
```
&p[k] == p + k
```

```
// équivaut à : p = &tab[0];
```

```
// valeur du kième élément à partir de p
```

```
// adresse du kième élément à partir de p
```

# Tableaux et pointeurs



**Même notation mais ce n'est pas la même chose !**

```
int tab[10];  
int* p = tab;
```

`sizeof(tab)` vaut 10

`sizeof(p)` dépend du processeur (4 si processeur 32 bits)

# Manipulation de bits

## Opérateurs

&	ET
	OU inclusif
^	OU exclusif
<<	décalage à gauche
>>	décalage à droite
~	complément à un

```
int n = 0xff, m = 0;  
m = n & 0x10;  
m = n << 2;      /* équivalent à: m = n * 4 */
```

**Attention:** ne pas confondre & avec && (et logique) ni | avec || (ou logique)

# Orienté objet en C

## C

```
typedef struct {  
    char* name;  
    long id;  
} User;
```

```
User* createUser (const char* name, int id);  
void destroyUser (User*);  
void setUserName (User*, const char* name);  
void printUser (const User*);  
....
```

```
void foo() {  
    User* u = createUser("Dupont");  
    setUserName(u, "Durand");  
    ....  
    destroyUser(u);  
    u = NULL;
```

## C++

```
class User {  
    char* name;    // en fait utiliser string  
    long id;  
public:  
    User (const char* name, int id);  
    virtual ~User( );  
    virtual void setName(const char* name);  
    virtual void print() const;  
    ....  
};
```

```
void foo() {  
    User* u = new User("Dupont");  
    u->setName("Durand");  
    ....  
    delete u;  
    u = NULL;
```

# Orienté objet en C

```
typedef struct User {  
    int a;  
    void (*print) (const struct User*);  
} User;
```

```
typedef struct Player { // subclass  
    User base;  
    int b;  
} Player;
```

```
void print(const User* u) {  
    (u->print)(u);  
}
```

```
void printUser(const User *u) {  
    printf("printUser a=%d \n", u->a);  
}
```

```
void printPlayer(const Player *u) {  
    printf("printPlayer a=%d b=%d\n",  
        u->base.a, u->b);  
}
```

```
User* newUser() {  
    User* p = (User*) malloc(sizeof(User));  
    p->a = 0;  
    p->print = printUser;  
    return p;  
}
```

```
Player* newPlayer() {  
    Player* p = (Player*) malloc(sizeof(Player));  
    p->base.a = 0;  
    p->base.print = printPlayer; // cast nécessaire  
    p->b = 0;  
    return p;  
}
```

```
int main() {  
    Player* p = newPlayer();  
    p->base.a = 1;  
    p->b = 2;  
    print(p);  
}
```

// NB: en fait il faudrait partager les pointeurs  
// de fonctions de tous les objets d'une même  
// classe via une vtable