

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

Eric Lecolinet - Télécom Paris – Institut Polytechnique de Paris
<http://www.telecom-paris.fr/~elc>

Octobre 2022

Brève historique

1972 : Langage C

AT&T Bell Labs

1983 : Objective C

NeXt puis Apple

1985 : C++

AT&T Bell Labs

1991 : Python

Guido van Rossum

1995 : Java

Sun, puis Oracle

2001: C#

Microsoft

2011: C++11

Consortium ISO

Extension objet du C

Syntaxe inhabituelle inspirée de Smalltalk

Extension object du C par *Bjarne Stroustrup*

Simplicité et rapidité d'écriture

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

Purement objet

Inspiré de C++ mais aussi Smalltalk, ADA, etc.

Originellement proche de Java

Inspiré de C++, Delphi, etc.

Révision majeure

Suivie de : C++14, C++17, C++20, C++23

Et aussi : Kotlin, Scala, Swift, Rust, Javascript, Ruby, Go, Ada, etc.

C vs C++ vs Java

C++

- à l'origine : **extension** du **C** (peut compiler du code **C**)

Java

- à l'origine : **simplification** de **C++** (+ inspiré de **Smalltalk, ADA**, etc.)
- **ressemblances**, mais aussi **différences** majeures avec C++

C/C++

- **disponibles** sur quasi toutes les plateformes
- **compatibles** entre eux + avec la plupart des **autres** langages
=> avantageux pour **développement multi-plateformes**

Points

- popularité
- rapidité, conso électrique

Références et liens

Liens utiles

- **Travaux Pratiques** de ce cours : www.enst.fr/~elc/cpp/TP.html
- **Toolkit graphique Qt** : www.enst.fr/~elc/qt
- **Extensions Boost** : www.boost.org

Livres, tutoriaux, manuels

- livre : **Le langage C++** de **Bjarne Stroustrup** (auteur du C++)
- **manuels de référence**
<http://cppreference.com> - www.cplusplus.com/reference
- **faqs, aide**
<https://stackoverflow.com> - <https://isocpp.org/faq>
- **Cours C++ de Christian Casteyde**
<http://casteyde.christian.free.fr/>

Premier chapitre :

Programme, classes, objets

Programme C++

Constitué

- de **classes** comme en **Java**
- éventuellement, de **fonctions** et **variables** « **non-membres** » comme en **C**

Bonne pratique : une classe principale par fichier

- pas de contraintes syntaxiques comme en **Java**

Car.cpp

```
#include "Car.h"

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

Truck.cpp

```
#include "Truck.h"

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

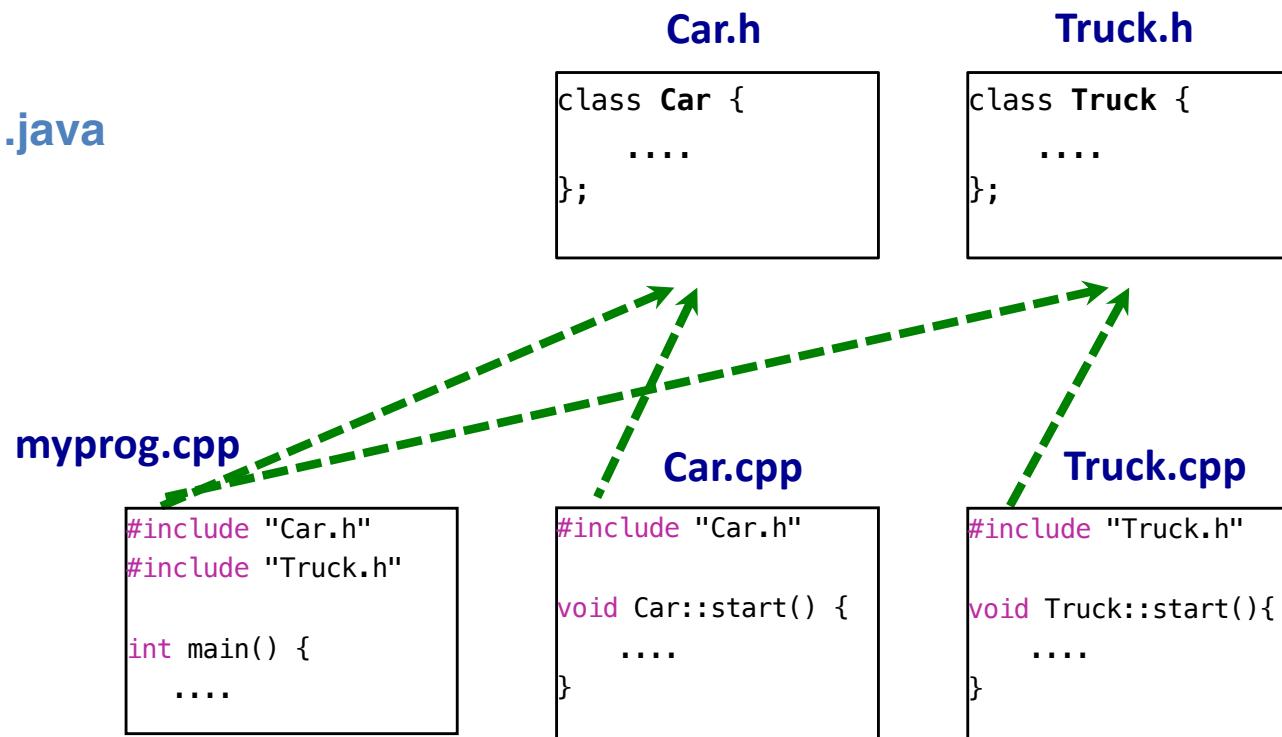
Déclarations et définitions

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**

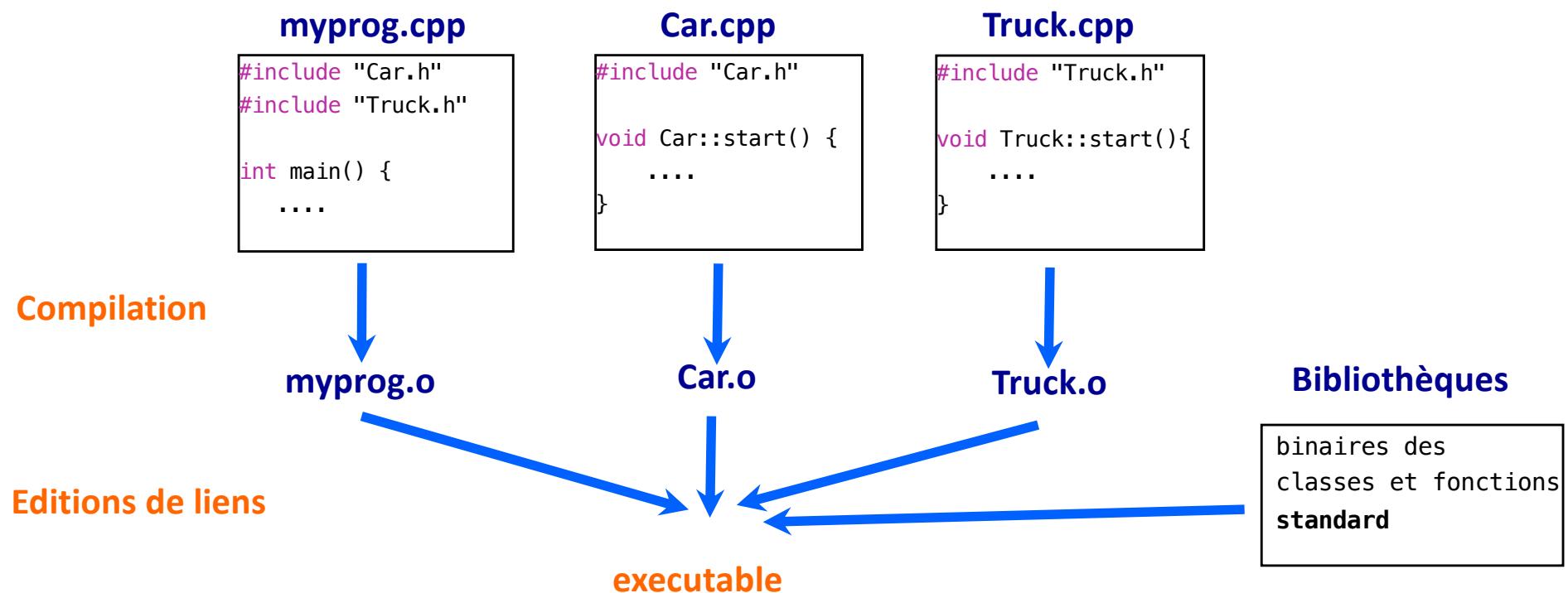
Java

- **tout dans les .java**



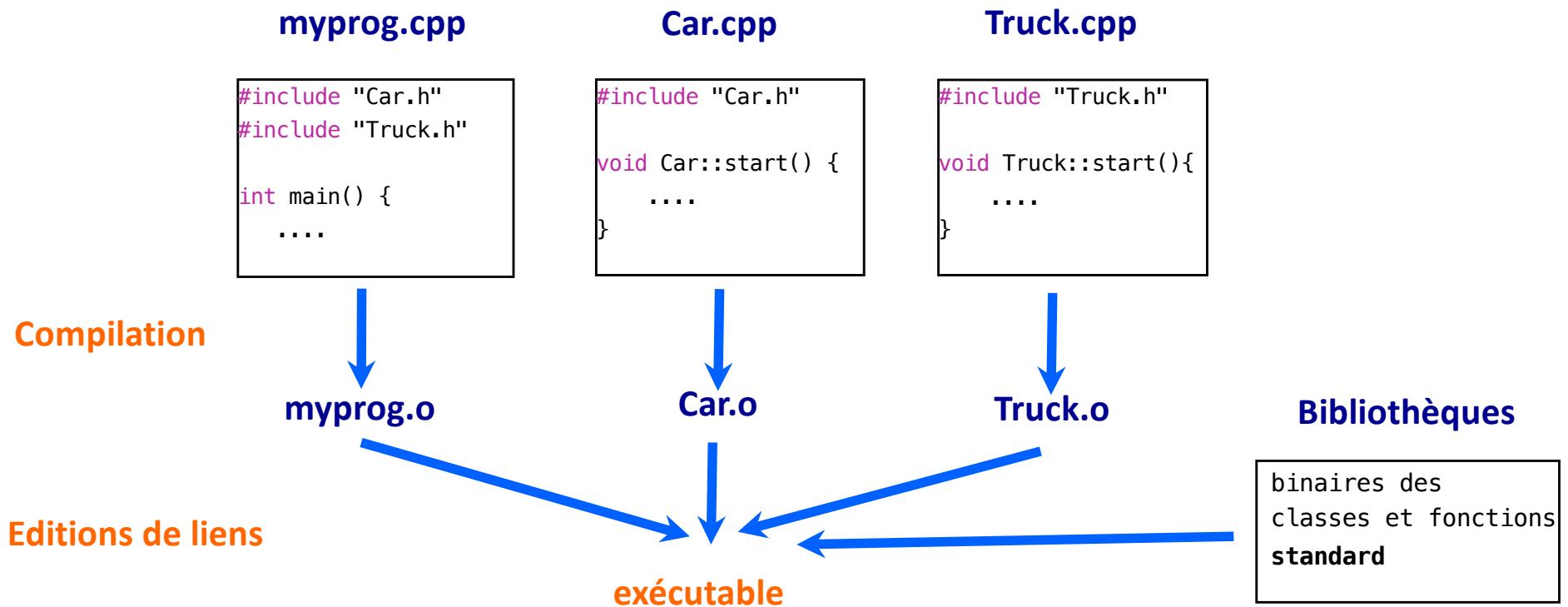
Compilation et édition de liens

- C/C++ : compilé en code machine natif
- Java : compilé (byte code), puis interprété par la JVM (ou compilé "JIT" = à la volée)



Compilation et édition de liens

En **C/C++** ce processus doit être **explicite** (e.g. par un **Makefile**) contrairement à **Java**



Quelques options de **g++** et **clang**

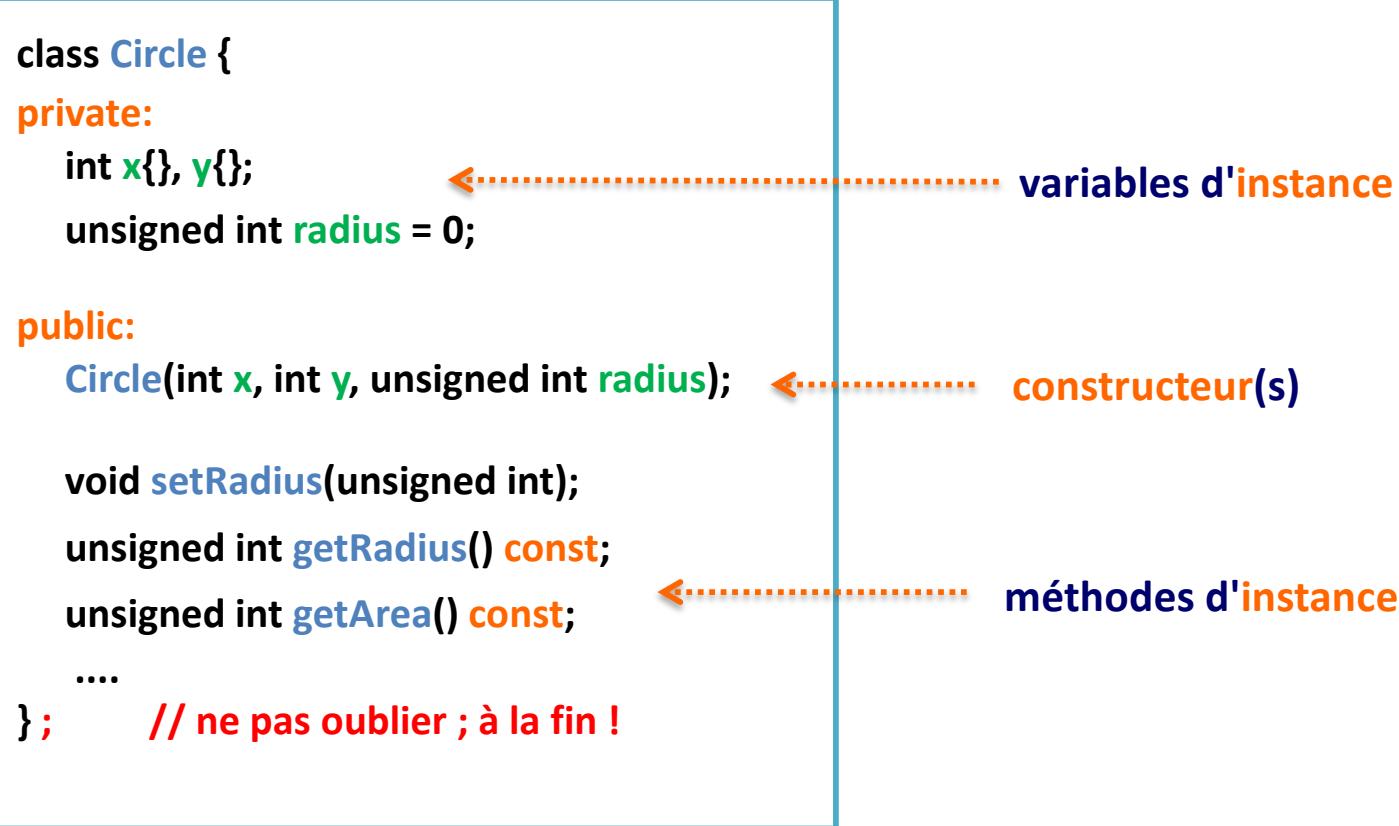
- **version** du langage : **-std=c++11**, etc.
- **erreurs** : **-Wall**
- **débogueur** : **-g**
- **optimisation** : **-O1 -O2 -O3 -Os**, etc.

Attention aux versions de C++
=> avoir un **compilateur** et des **bibliothèques** à jour !

Déclaration de classe

Dans le header **Circle.h** :

```
class Circle {  
private:  
    int x{}, y{};  
    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 !
```



Remarques

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

Variables d'instance

```
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;  
....  
};
```

initialiser les pointeurs et types de base !!!!

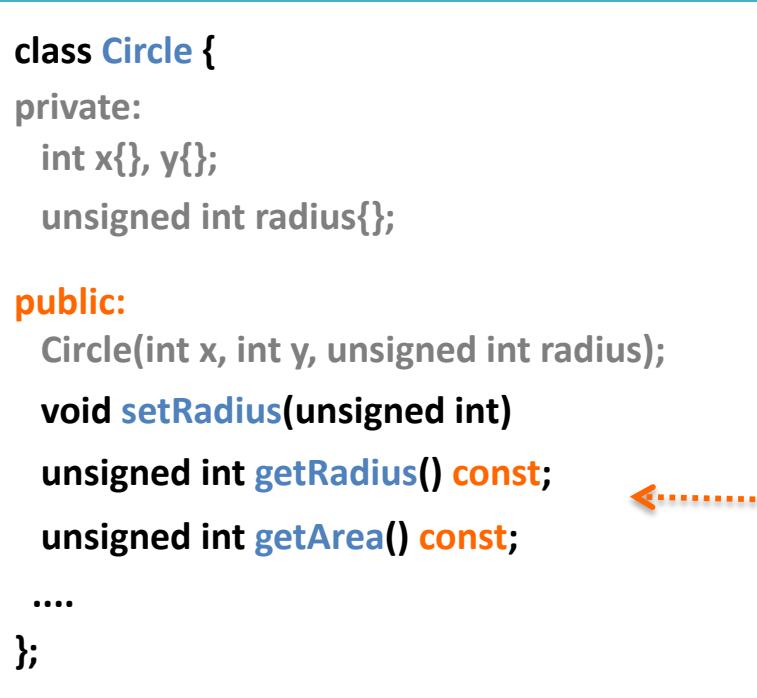
{ } est l'"initialisateur universel"

Variables d'instance

- chaque objet possède **sa propre copie** de la variable
- normalement **private** ou **protected**
- **doivent être initialisées** si c'est des **pointeurs** ou **types de base**

Méthodes d'instance

```
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;  
....  
};
```



← méthodes d'instance

1^{er} concept fondamental de l'OO : lien fonctions / données

- ces méthodes **ont accès** aux **variables d'instance** (et de classe)

Notes :

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

Définition des méthodes

Dans fichier d'implémentation `Circle.cpp`

```
#include "Circle.h"

Circle::Circle(int x, int y, unsigned int r) {
    this->x = x;
    this->y = y;
    this->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

Alternative : déclarer et définir en même temps

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

◀ ··· méthodes inline

Méthodes inline

- exécution (en théorie) **plus rapide** mais programme **plus gros**
- à utiliser avec **modération**
- mot-clé **inline inutile** dans ce cas (c'est implicite)

Constructeurs

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

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

Constructeurs

Trois formes

```
#include "Circle.h"

Circle::Circle(int x , int y) {
    this->x = x;
    this->y = y;
    this->radius = 0;
}

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

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

comme Java

après le :
x(x) est correct

plus général (OK avec conteneurs)

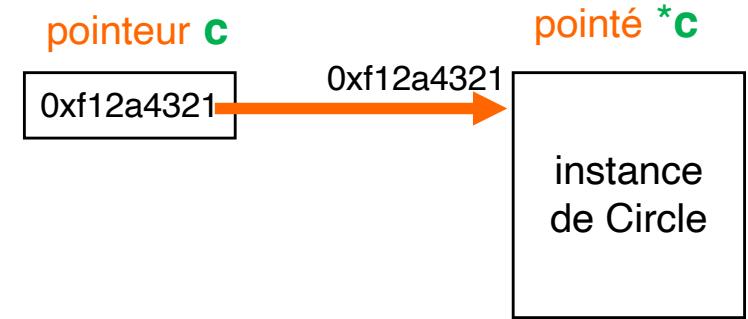
Dans header

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

Instanciation

```
#include "Circle.h"

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

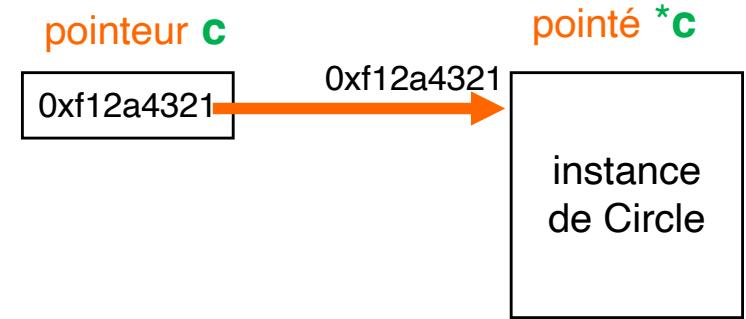


dans un **autre** fichier .cpp !

Instanciation

```
#include "Circle.h"

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



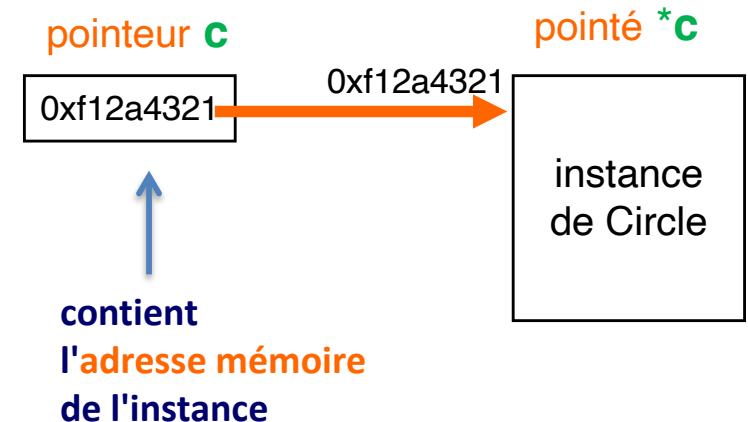
new crée un objet (= une nouvelle instance de la classe)

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

Instanciation

```
#include "Circle.h"

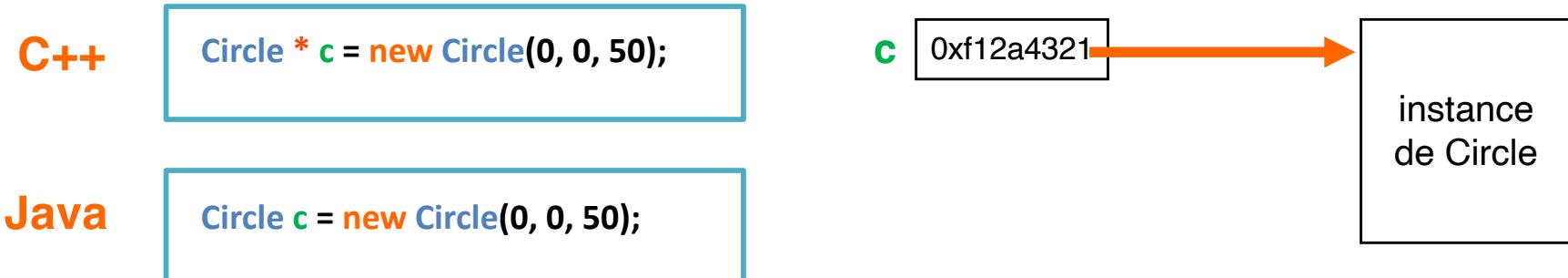
int main() {
    Circle * c = new Circle(0, 0, 50);
    ...
}
```



La variable **c** référence l'objet

- **c** est un **pointeur** (d'où l' *****) qui contient l'**adresse mémoire** de l'instance
- ici **c** est une **variable locale** (variable de la fonction)

Pointeurs C/C++ vs. références Java

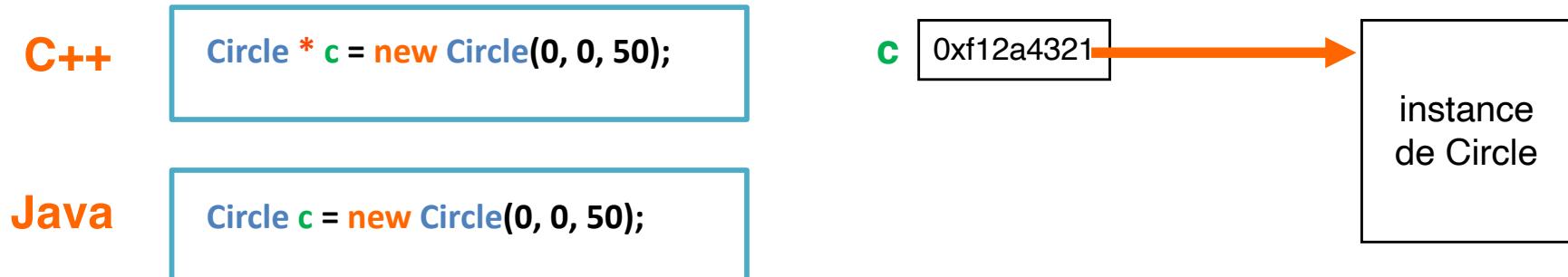


Pointeur C/C++

- variable qui contient une **adresse** **similaire**
- valeur **accessible** **valeur cachée**
- **arithmétique** des pointeurs **pas** d'arithmétique
 - calcul d'adresses bas niveau
 - source d'**erreurs** et de **vulnérabilités** !
- **pas** de ramasse-miettes **ramasse-miettes**

Référence Java

Pointeurs C/C++ vs. références Java



LES REFERENCES JAVA
SONT SIMILAIRES AUX POINTEURS

- les **pointeurs** du **C/C++** sont un cas particulier de **références**
- il n'y a pas de **ramasse-miettes** en **C/C++**

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 `->` déréférence 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;    NE COMPILE PAS !  
  
    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

Séparer la spécification de l'implémentation (concept de "boîte noire")



Déclaration des méthodes

- **interface** avec l'extérieur (API)
- on **interagit** via les **méthodes**

Variables et implémentation

- **propres** à l'objet
- **seul l'objet** accède à ses **variables**



l'interface et la méthode "toc toc toc"



l'objet et ses variables

Encapsulation

Séparer la spécification de l'implémentation (concept de "boîte noire")



Déclaration des méthodes

- interface avec l'extérieur (API)
- on interagit via les méthodes

Variables et implémentation

- propres à l'objet
- seul l'objet accède à ses variables



Abstraire

- exhiber les concepts, cacher les détails

Modulariser

- limiter les dépendances entre composants

Protéger l'intégrité de l'objet

- ne peut être modifié à son insu

L'objet assure la validité de ses données

- il est le mieux placé pour le faire !

Encapsulation : droits d'accès

Droits d'accès C++

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

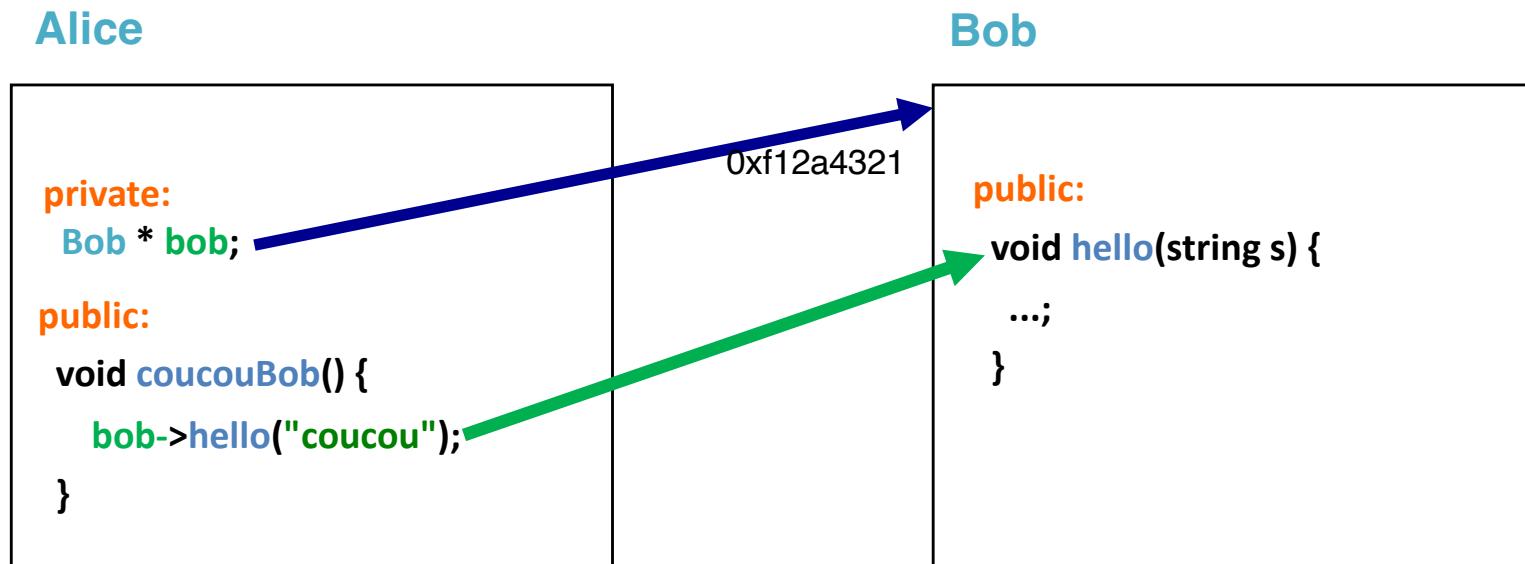
```
class Circle {  
    friend class ShapeManager; <----> cette classe  
    friend bool isInside(const Circle&, int x, int y); <----> cette fonction  
    ...  
};
```

a droit d'accès
a droit d'accès

Droits d'accès Java

- similaires sauf que :
 - **par défaut** : toutes les classes du **package**
 - **friend** n'existe pas

Accès vs. droits d'accès



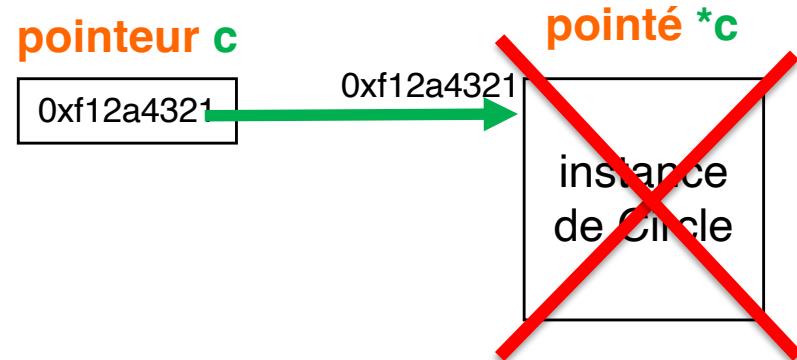
Pour "envoyer un message" à un objet il faut :

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

Il ne suffit pas d'avoir la **clé** il faut aussi savoir **où** il se trouve !

Destruction des objets

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



delete détruit le pointé

- 1) appelle le **destructeur**
- 2) libère la mémoire du **pointé *c** (et non celle du **pointeur C** !)

Pas de ramasse miettes !

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

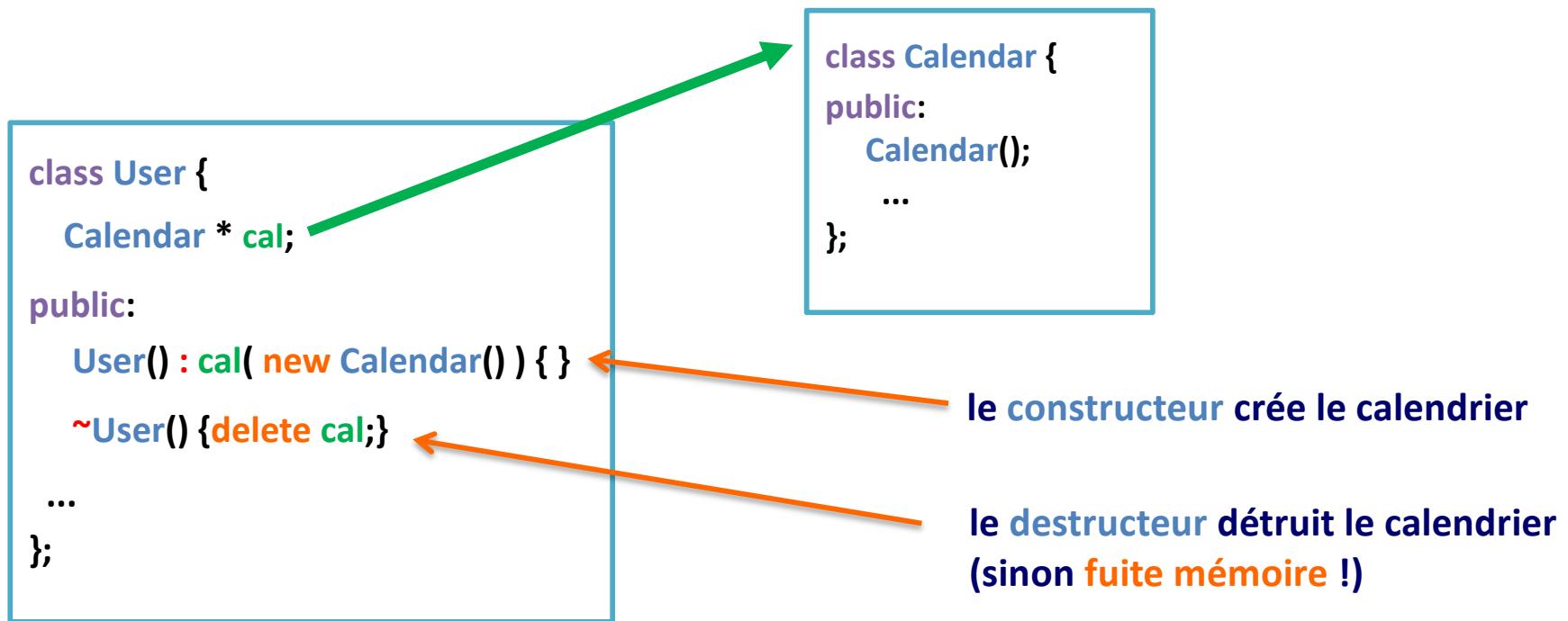
Destructeur

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

Méthode appelée quand l'objet est détruit

- **signale** à l'objet qu'il **va** être détruit
- appelée **automatiquement pas par votre code !!!**

Destructeur



Quand faut-il définir un destructeur ?

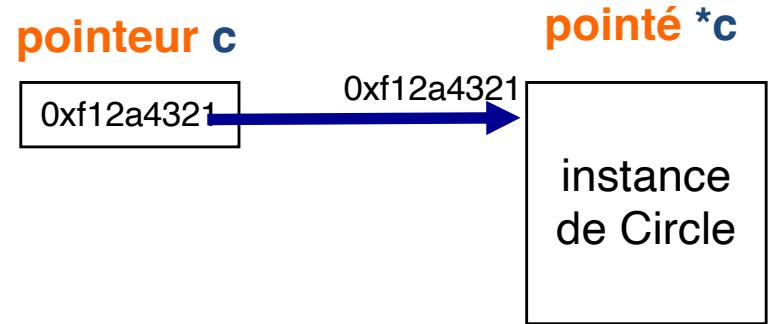
- rarement car il y en a un **par défaut**
- si l'objet doit "**faire le ménage**"
- et pour les **classes de base polymorphes** (à suivre)

En Java la méthode **finalise()** joue le même rôle (peu utilisée)

Pointeurs nuls et pendants

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

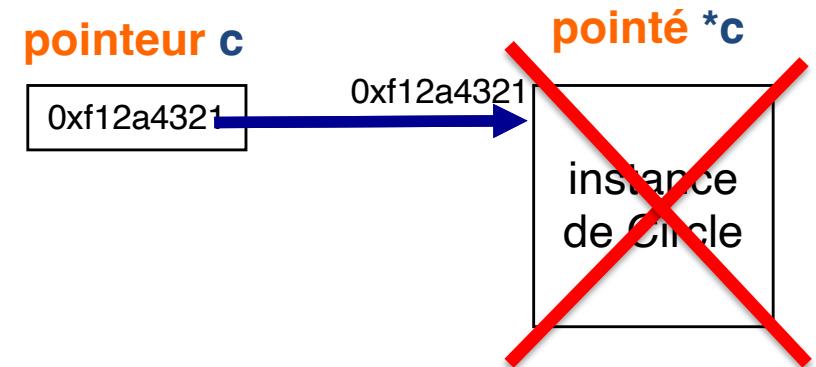
```
void foo(Circle * c) {  
    unsigned int area = c->getArea();  
}
```



Correct ?

Pointeurs nuls et pendants

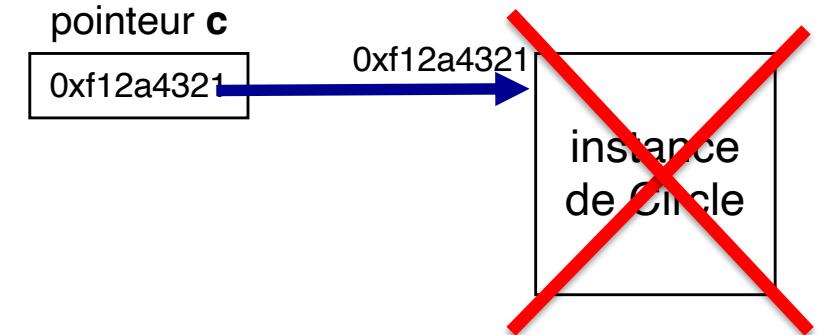
```
void bar() {  
    Circle * c = new Circle(10, 20, 30);  
    foo(c);  
  
    delete c;   ← pointeur pendant (= indéterminé)  
    foo(c);  
    delete c;   ←  
}  
  
void foo(Circle * c) {  
    unsigned int area = c->getArea();  
}
```



BUG !!! peut planter (ou pas !)

Pointeurs nuls et pendants

```
void bar() {  
    Circle * c = new Circle(10, 20, 30);  
    foo(c);  
    delete c;  
    c = nullptr; // pointeur nul (pointe sur rien)  
    foo(c);  
    delete c; // OK ne fait rien  
}
```



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

Initialisation des variables

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

Correct ?

```
void foo() {  
    Circle * c1;  
    Circle * c2 = nullptr;  
    Circle * c3{};  
    Circle * c4 = new Circle;  
}
```

Initialisation des variables

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

x, y, radius indéterminés !

radius indéterminé !

```
void foo() {  
    Circle * c1;   
    Circle * c2 = nullptr;  
    Circle * c3{};  
    Circle * c4 = new Circle();  
}
```

c est pendant !!!!

c est nul

c pointe sur un objet

Initialisation des variables

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

INITIALISER les pointeurs
et types de base

```
void foo() {  
    Circle * c2 = nullptr; // Initialisé à nullptr  
    Circle * c3{}; // Initialisé à 0  
    Circle * c4 = new Circle(); // Initialisé à l'adresse de la nouvelle instance  
}
```

tout est initialisé

Paramètres par défaut

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

- n'existent pas en **Java**
- les valeurs par défaut doivent être **à partir de la fin** :

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

Surcharge (overloading)

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

Même nom mais signatures différentes

- pour des méthodes d'une **même classe**
- possible aussi pour des fonctions **non-membres**

Ne pas confondre !

- avec la **redéfinition de méthodes** dans une **hiérarchie** de classes

Variables de classe

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

variables de classe

pour y accéder:

```
int i = Circle::count;
```

Représentation unique en mémoire

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

Note

- doivent être **définies** dans un (seul) fichier .cpp (sauf en C++17 si elles sont **inline** ou **constexpr**)

Méthodes de classe

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

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;  
  
int main() {  
    Circle * mc = new Circle();  
    graph::Circle * gc = new graph::Circle();  
}
```

math accessible par défaut

équivaut à `math::Circle`

using namespace

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

Entrées / sorties standard

```
#include "Circle.h"
#include <iostream>           ← flux d'entrées/sorties

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

header pour 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

String Buffers

```
#include "Circle.h"
#include <iostream>
#include <sstream>          ← buffers de texte

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);           ← écrit dans un buffer

    unsigned int r = 0, a = 0;
    ss >> r >> a;        ← lit depuis un buffer
    cout << ss.str() << endl;
}
```

Flux pour buffers

stringstream input/output buffer

istringstream input buffer

ostringstream output buffer

Chapitre 2 : Héritage et polymorphisme

Héritage

2^e Concept fondamental de l'OO

- les sous-classes **héritent** les **méthodes** et **variables** de leurs **superclasses**

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

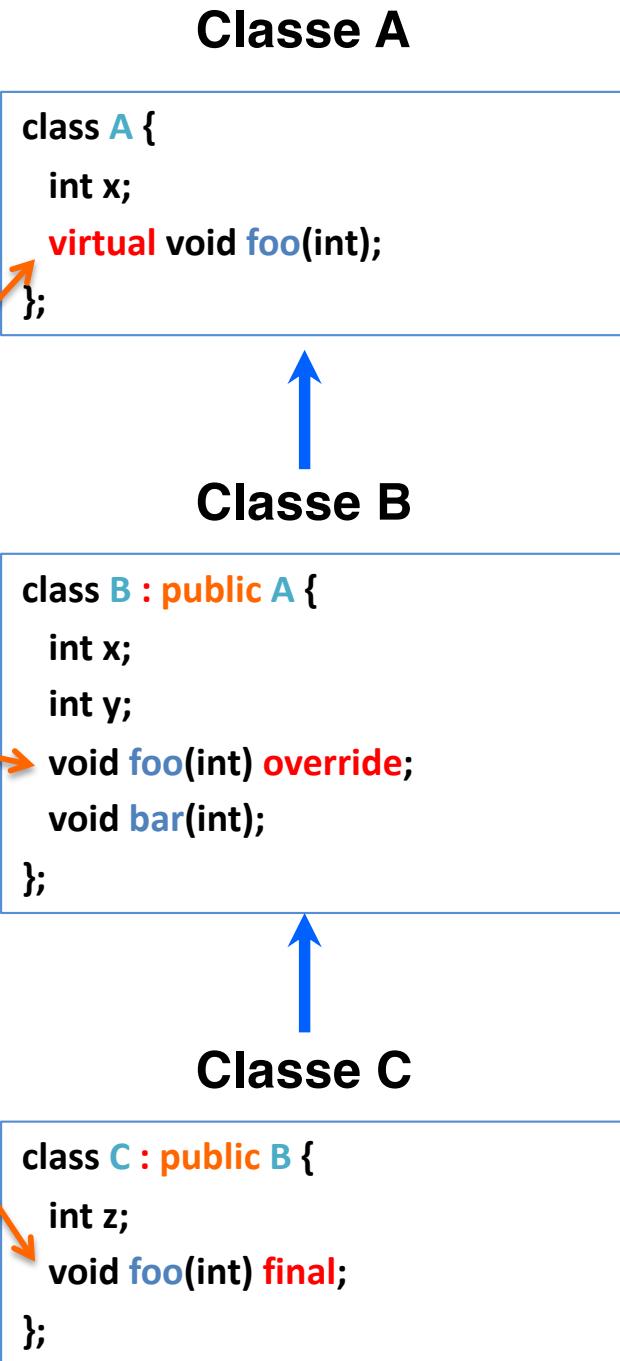
Constructeurs / destructeurs

- pas hérités car chaînés !

Méthodes

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

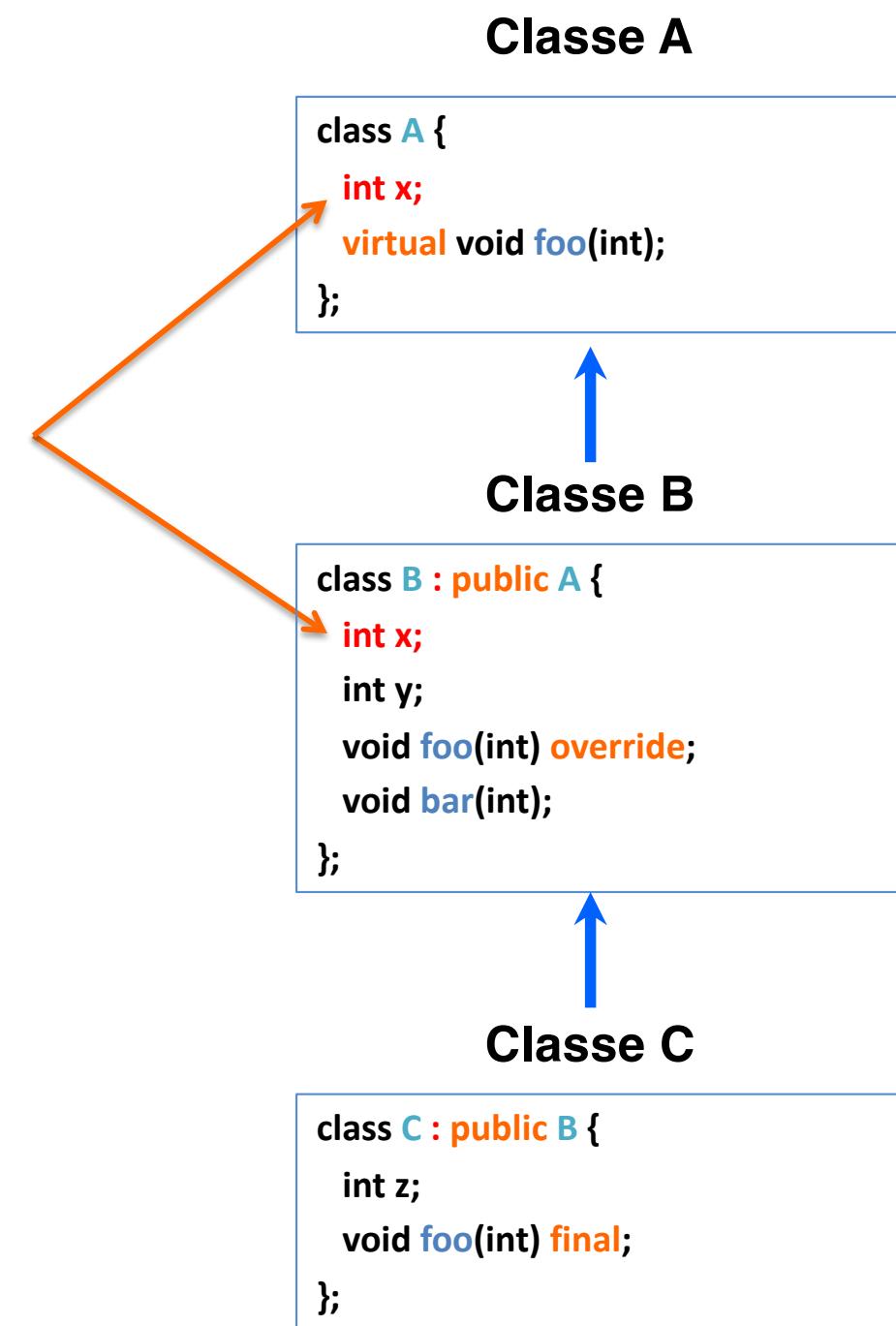
: public : héritage (comme extends de Java)
virtual : définition de plus haut niveau
override : redéfinition
final : ne peut être redéfinie



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: déclarations

```
class Rect {  
protected:  
    int x{}, y{};  
    unsigned int w{}, h{};  
  
public:  
    Rect() {}  
    Rect(int x, int y, unsigned int w, unsigned int h);  
    unsigned int getWidth() const {return width;}  
    unsigned int getHeight() const {return height;}  
    virtual void setWidth(unsigned int w) {width = w;}  
    virtual void setHeight(unsigned int h) {height = h;}  
    //...etc...  
};
```

Initialiser les variables

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

Square

```
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;  
};
```

Dérivation de classe: comme extends en Java

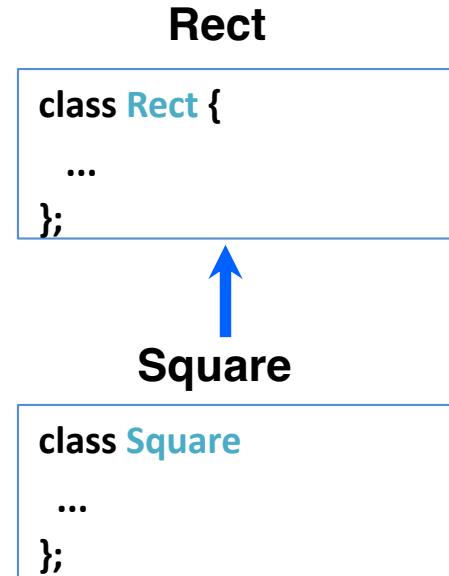
Redéfinition des méthodes
Pourquoi faut-il redéfinir ces deux méthodes ?

virtual car redéfinies

Exemple: redéfinitions

```
class Rect {  
protected:  
    int x{}, y{};  
    unsigned int w{}, h{};  
  
public:  
    Rect() {}  
    Rect(int x, int y, unsigned int w, unsigned int h);  
    unsigned int getWidth() const {return width;}  
    unsigned int getHeight() const {return height;}  
    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;}  
};
```

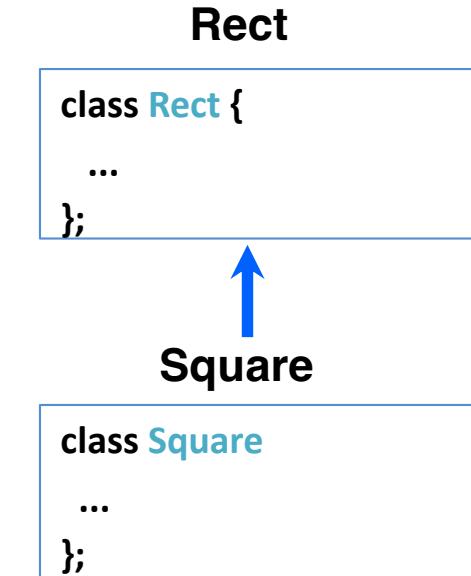


*sinon ce n'est
plus un carré !*

Exemple: redéfinitions

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



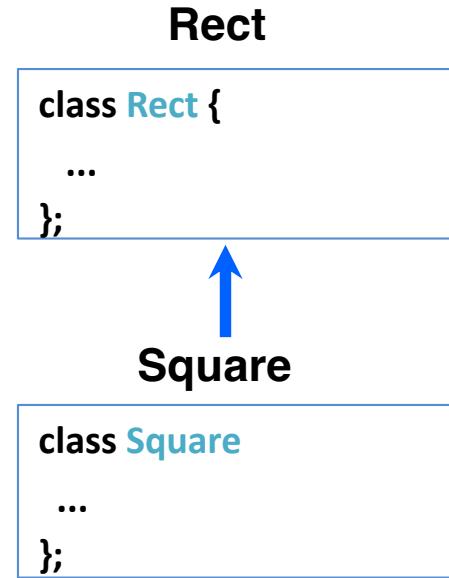
Chaînage implicite des constructeurs
équivaut à: `Square() : Rect() {}`

Chaînage explicite
des constructeurs
en Java:
`super(x,y,size,size)`

Exemple: redéfinitions

```
class Rect {  
protected:  
    int x{}, y{};  
    unsigned int w{}, h{};  
public:  
    Rect() {}  
    Rect(int x, int y, unsigned int w, unsigned int h);  
    unsigned int getWidth();  
    unsigned int getHeight() const;  
    virtual void setWidth(unsigned int w);  
    virtual void setHeight(unsigned int h);  
    //...etc...  
};
```

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

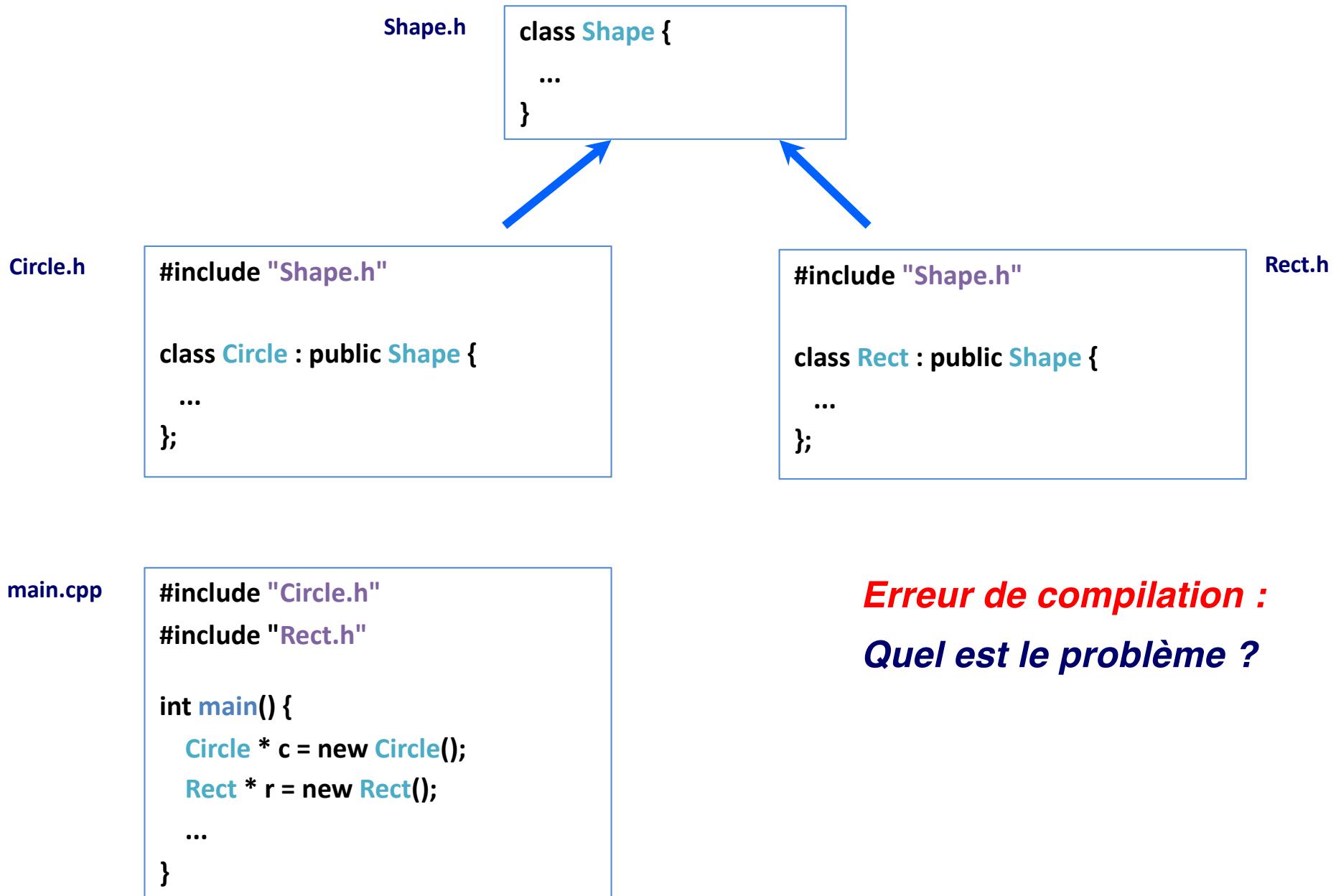


!!! ATTENTION

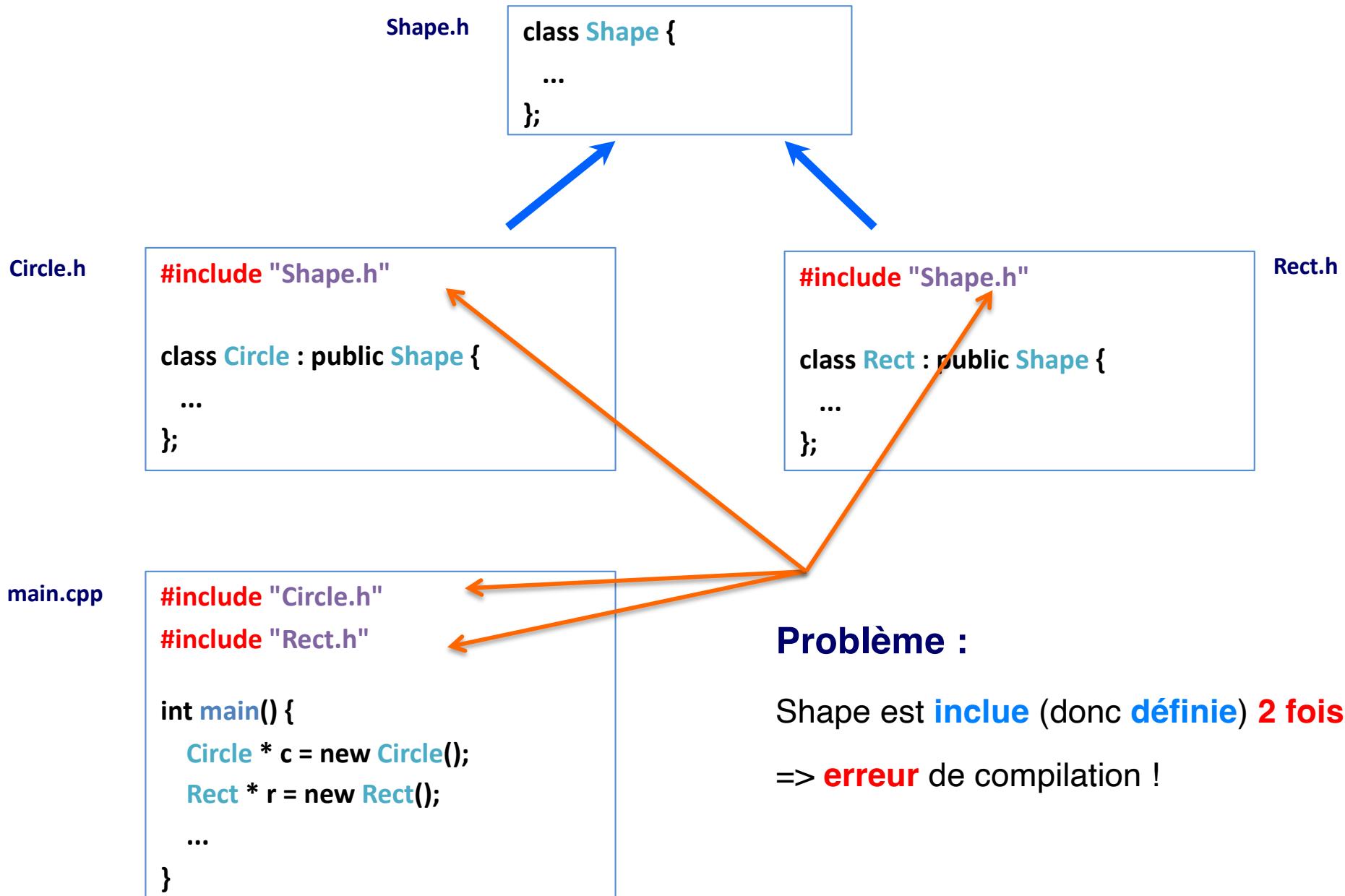
CORRECT

FAUX !!!!

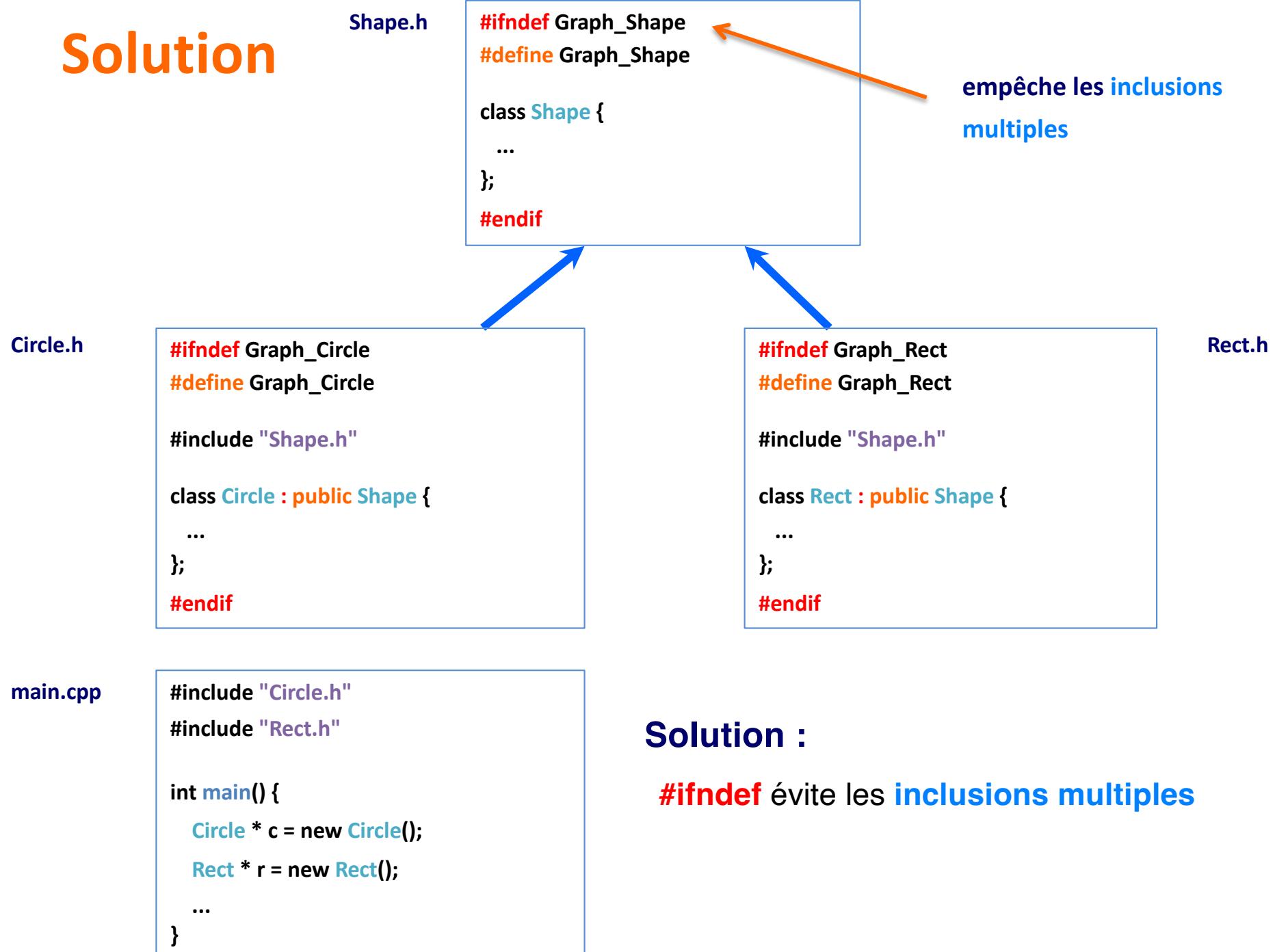
Inclusion des headers



Inclusion des headers



Solution



Solution :

#ifndef évite les **inclusions multiples**

Directives du préprocesseur

Header
Truc.h

```
#ifndef Truc
#define Truc
class Truc {
...
};
#endif
```

inclus 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^e 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();
    Rect * r = s; // OK ?
    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 : OK car upcasting implicite
    Square * s2 = new Rect();       // erreur de compilation! car downcasting interdit !
    Square * s3 = r;               // erreur de compilation!
}
```

Rect

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

Square

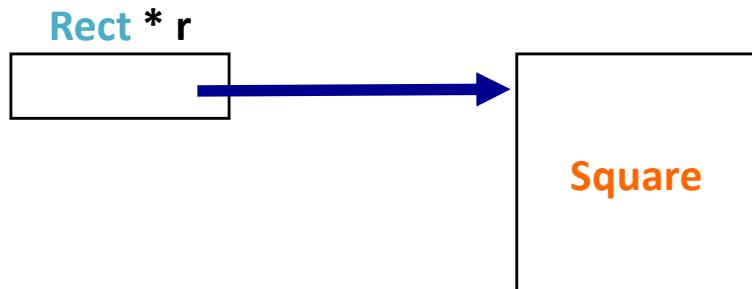
```
class Square : public Rect {  
    ...  
};
```

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);
```



Rect

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

Square

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



Polymorphisme : Java

Question à \$1000

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

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

```
class Rect {
```

```
...
```

```
virtual void setWidth(int);
```

```
...
```

```
};
```

Rect

Square

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

Java : liaison dynamique / tardive

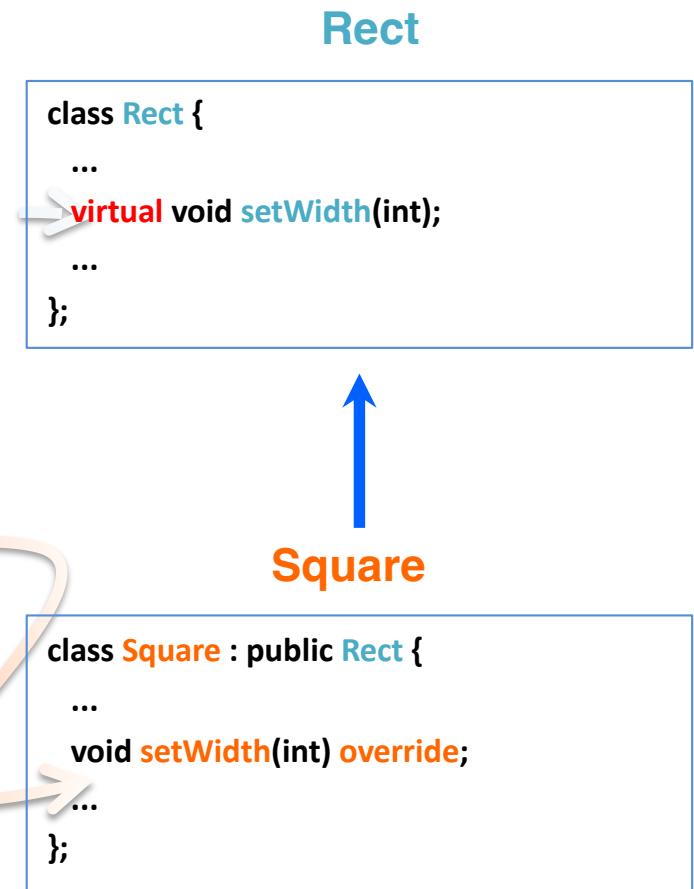
- choix de la méthode à l'**exécution** => méthode du **pointé**
- sinon le **carré** deviendrait un **rectangle** !

Polymorphisme : C++

Question à \$1000

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

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



C++

- avec **virtual** : **liaison dynamique / tardive** => méthode du **pointé**
- sans **virtual** : **liaison statique** => méthode du **pointeur**

Règles à suivre

Dans la classe de base

- mettre **virtual** si la méthode est **redéfinie**

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

Redéfinitions

- une **redéfinition** de méthode **virtual** est **virtual**
- mettre **override** ou **final**

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

Notes

- les **paramètres** doivent être **identiques** (sinon c'est de la **surcharge** !)
- le **type de retour** peut être une **sous-classe** (**covariance** des types de retour)

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

Règles à suivre

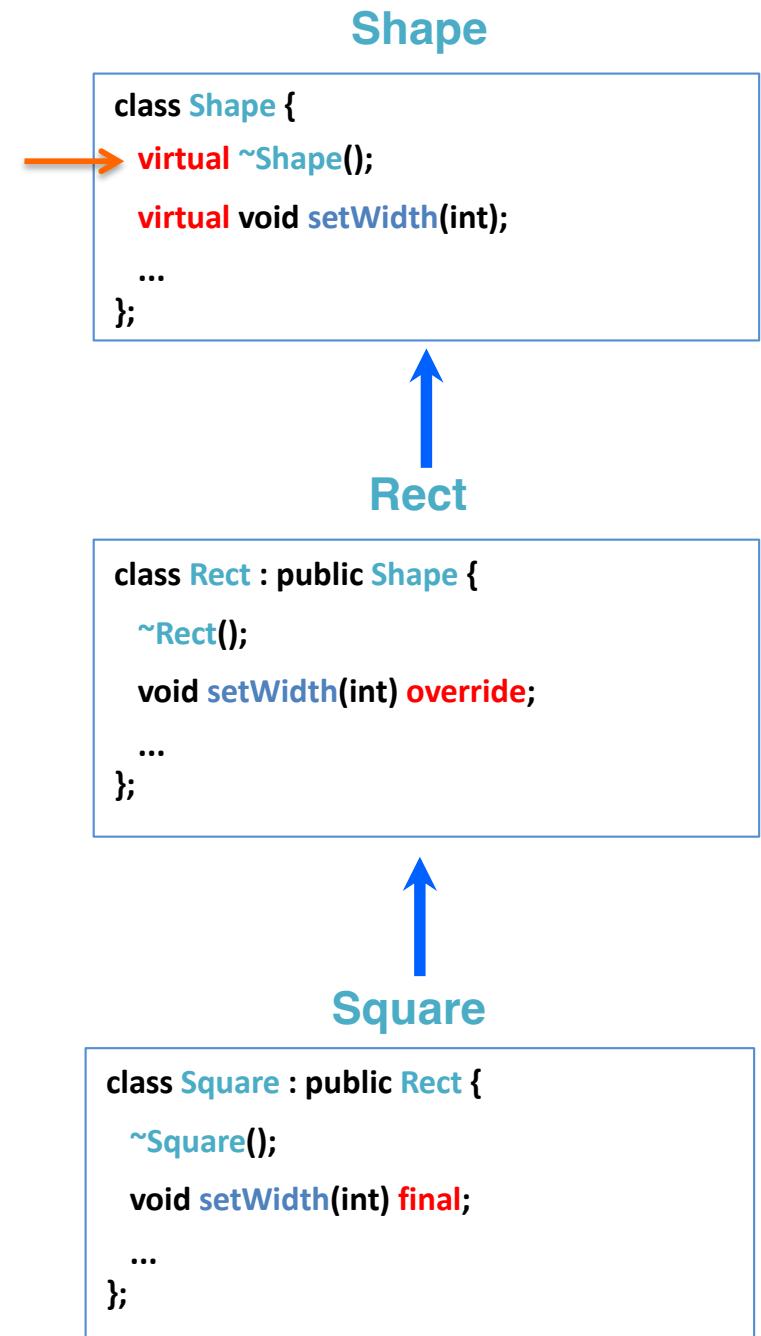
Destructeurs

- doivent être **virtual** s'il y a des méthodes **virtual**
- sinon **seul** le destructeur du **pointeur** est appelé
- exemple :

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

si **~Shape()** pas **virtual**

destructeurs ~Rect() et ~Square() pas appelées !



Méthodes non virtuelles

Shape

- **pas** de polymorphisme !
- le résultat risque donc d'être **faux** !

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

A utiliser seulement si :

- la méthode n'est **jamais redefinie**
- la classe n'est **jamais héritée**
- pour des raisons de **performance**
en faisant attention aux possibles **erreurs** !

Rect

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

Square

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

Les méthodes virtuelles ont un coût

- un peu plus **lentes**
- objets un peu plus **gros** (cf. plus loin)

Méthodes abstraites

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

Méthode abstraite = méthode sans implémentation

- cas particulier de **méthode virtuelle**
- spécification d'un **concept**
- doit être **redéfinie** et **implémentée** dans les sous-classes (pas instantiables sinon)
- **Java** : pareil mais mot clé **abstract**

Méthodes et classes abstraites

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

But des méthodes abstraites

- exposer des **concepts communs** à un ensemble de classes
- pour pouvoir les traiter de **manière uniforme**

Classe abstraite

- classe dont **au moins** une méthode est **abstraite**
- **Java** : pareil mais mot clé **abstract**

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;  
    ....  
};
```

```
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;}  
    unsigned int getWidth() const override {return 2 * radius;}  
    unsigned int getHeight() const override {return 2 * radius;}  
    unsigned int getArea() const override {return PI * radius * radius;}  
    ....  
}
```

implémentations communes
à toutes les sous-classes

méthodes abstraites:
l'implémentation dépend
des sous-classes

doivent être redefinies
et 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

- cas particulier de **classe abstraite**
- pure **spécification** : toutes les méthodes sont **abstraites**
- **Java / C#** : mot clé **interface** qui permet une forme dégradée d'**héritage multiple**

Traitements uniformes

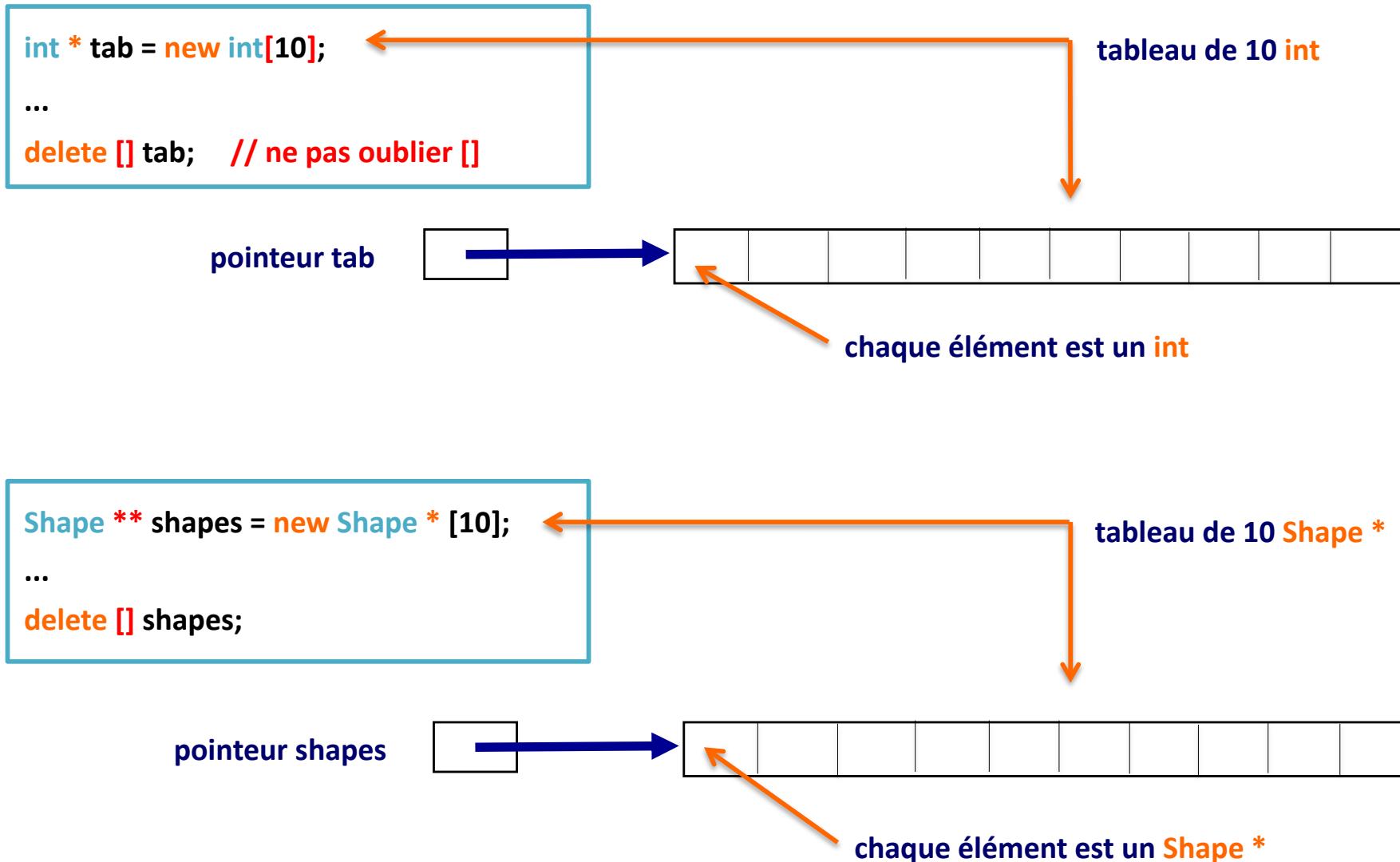
```
#include "Rect.h"
#include "Circle.h"

void foo() {
    Shape ** shapes = new Shape * [10]; ← tableau de 10 Shape *
    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);
}
```

```
#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

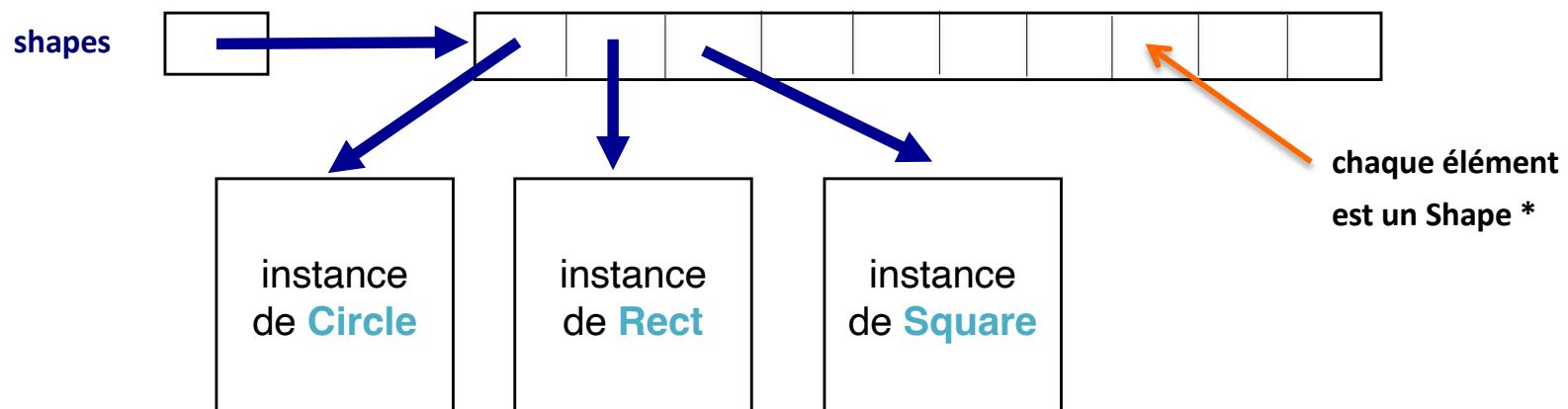


Traitements uniformes (2)

```
#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);
}
```

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



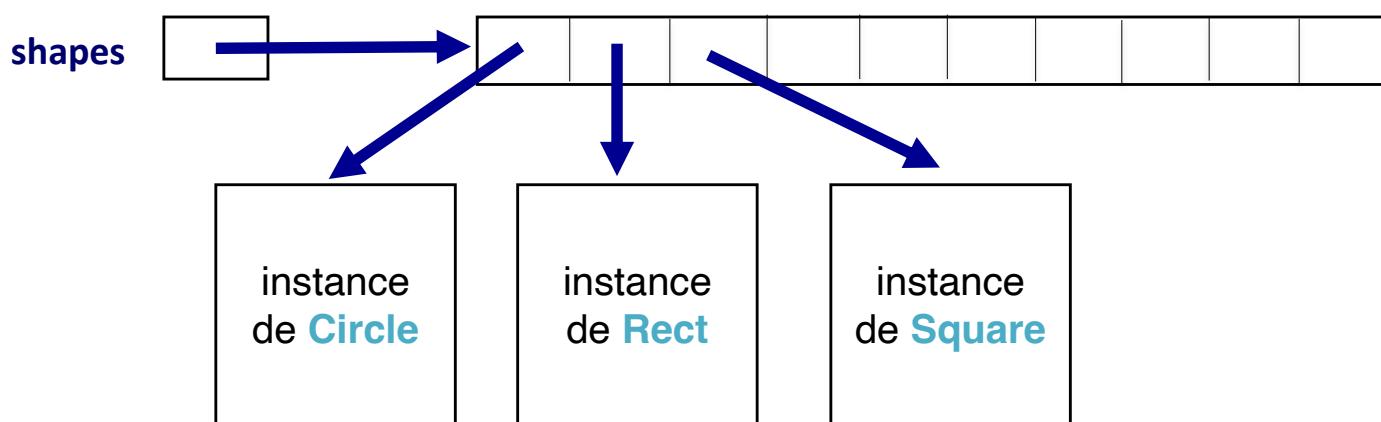
Magie du polymorphisme

```
#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;
    }
}
```

nombre d'éléments
pas d'autre moyen de le 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;
    }
}
```

Noter que

- cette fonction **ignore** l'existence des classes **Circle, Rect, Square !**
- elle pourra donc traiter de **nouvelles classes sans modification !**

Polymorphisme

permet de traiter des classes **differentes** de manière **uniforme**
sans connaître leur implémentation

Chaînage des méthodes

Règle générale : éviter les duplications de code

- à plus ou moins long terme ca **diverge** !
⇒ code difficile à **comprendre** et à **maintenir =>** risque de **bugs** !

Solutions

- 1) utiliser l'**héritage**
- 2) le cas échéant, **chaîner** les méthodes des superclasses

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

Concepts fondamentaux de l'orienté objet

En résumé : 4 fondamentaux

- 1) **méthodes d'instance** : lien entre les fonctions et les données
- 2) **encapsulation** : crucial en OO (mais possible avec des langages non OO)
- 3) **héritage** : simple, multiple ou entre les deux
- 4) **polymorphisme d'héritage** : toute la puissance de l'OO !
 - aussi appelé polymorphisme **dynamique**, de **sous-types**, de **sous-classes**, etc.
 - note : il y a d'autres formes de polymorphisme !

Implémentation des méthodes d'instance

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();  
}
```

```
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;  
....  
};
```

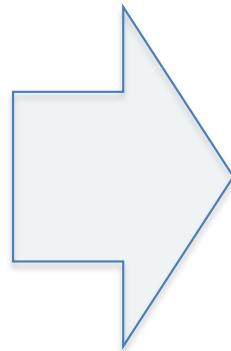
Comment la méthode accède à radius ?

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

Implémentation 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();  
}
```



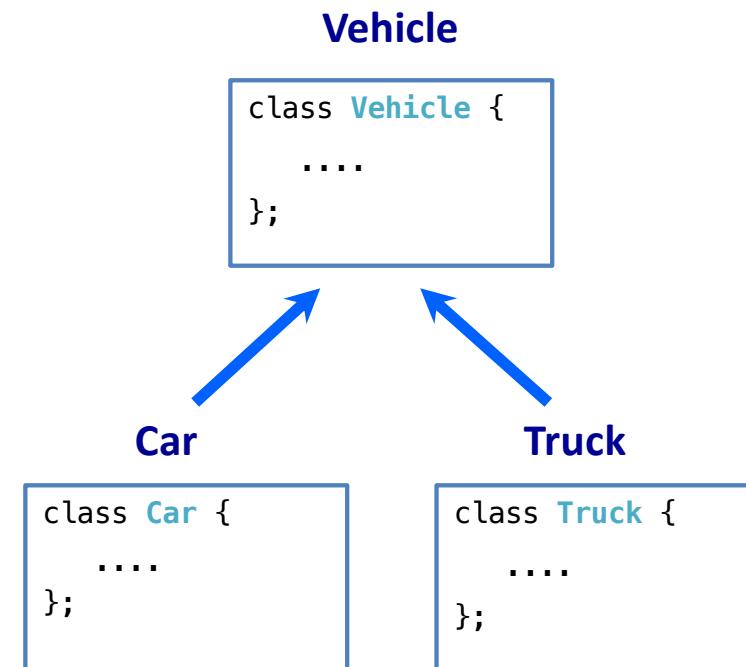
```
unsigned int a = getRadius(c);  
  
unsigned int getRadius(Circle * this) const {  
    return this->radius;  
}  
  
unsigned int getArea(Circle * this) const {  
    return PI * getRadius(this) * getRadius(this);  
}
```

Le paramètre caché **this** permet :

- d'accéder aux **variables** d'instance
- d'appeler les **méthodes** d'instance

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);  
    ...  
};
```

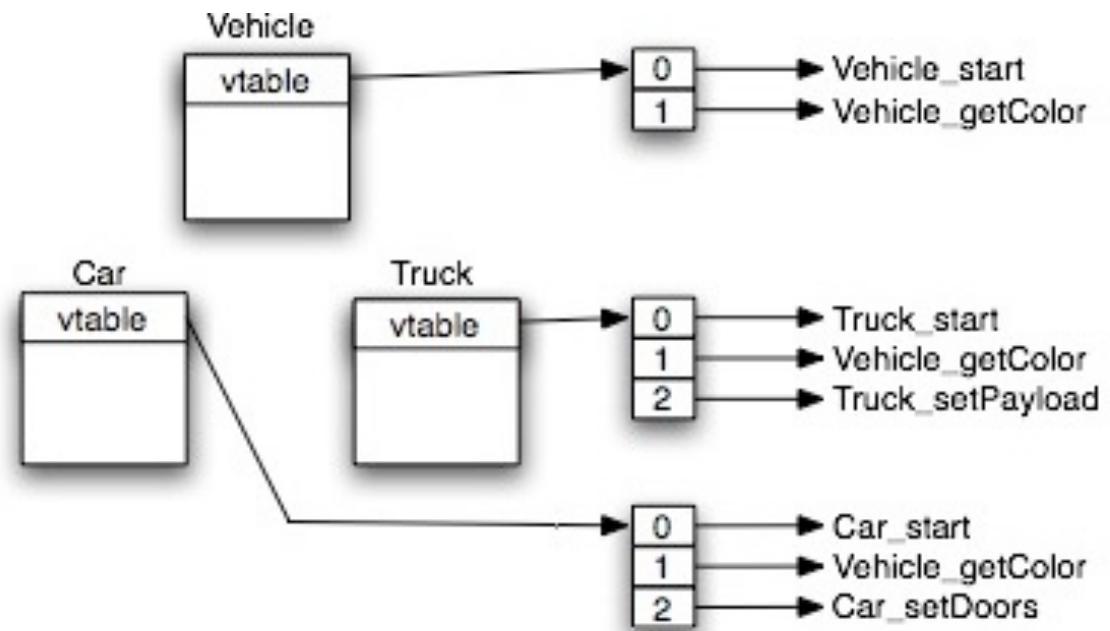


Implémentation des méthodes virtuelles

```
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 **peu important**
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

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().  
     */  
};
```

Doxxygen : documentation automatique

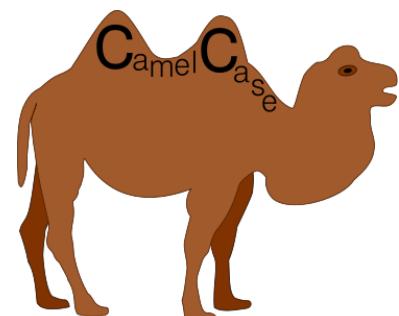
- similaire à **JavaDoc** mais fonctionne avec de **nombreux langages**
- documentation : www.doxygen.org

Style et commentaires

Conseils

- 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

```
/** @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().  
 */  
};
```

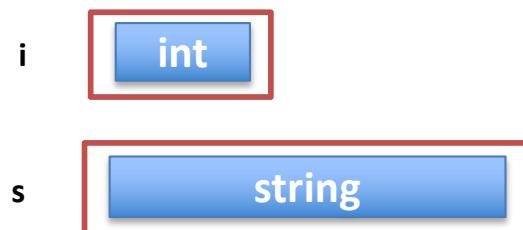


Chapitre 3 : Mémoire

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";  
    string s2 = string("Bonjour");  
    s += " World";  
    s.erase(4, 1);  
}
```

- accède aux champs de l'objet

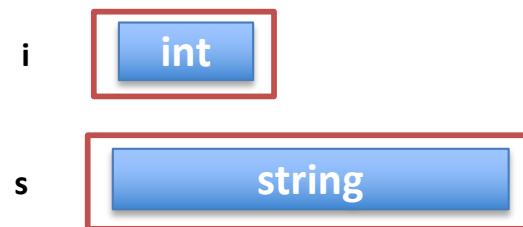
Différences

- **C++** : types de base et **objets** (structures en C)
- **Java** : que **types de base** (et **références**) !

Allocation mémoire

Mémoire globale/static

- variables **globales** ou **static** dont **variables de classe**
- **créées** au lancement du programme **détruites** à la fin du programme
- la variable **contient** la donnée



```
int glob = 0; // variable globale
static int stat = 0; // propre à ce fichier

void foo() {
    static int i = 0; // initialisée une seule fois !
    i += 10;

    static string s = "Hello";
    s += "World";
    s.erase(4, 1);
}
```

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

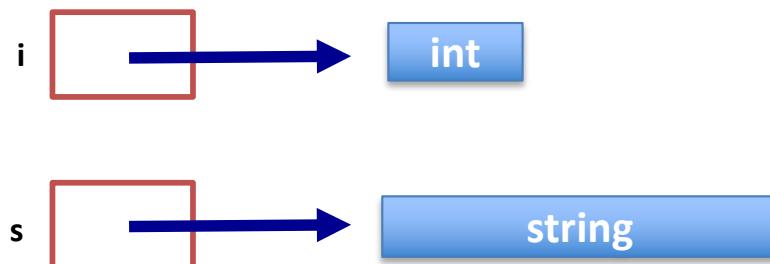
Différences

- **C++** : types de base et **objets**
- **Java** : que pour **variables de classe** (types de base ou références)

Allocation mémoire

Mémoire dynamique (tas/heap)

- pointés **créés** par **new**
détruits par **delete**
- la variable (**pointeur**) **pointe** sur la donnée (**pointé**)



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

Différences

- **C++ : objets et types de base**
penser à **détruire les pointés** via **delete** !
- **Java : que pour les objets**

***s** est le **pointé**

→ accède aux champs de l'objet :
`a->x == (*a).x`

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

sous-objet : contient l'objet (pas possible en Java)

leftDoor

Door

pointeur : pointe l'objet (comme Java)

Variables d'instance contenant un objet

- créés / détruites **en même temps** (et allouées dans même mémoire) que le **contenant**
- les **constructeurs / destructeurs** sont appelés
- n'existe pas en **Java**
- cf. `std::string` dans le TP !

C++ vs. Java

C++

- traite **objets comme types de base**
- idem en C avec les **struct**

```
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++

Java

- **objets toujours** créés avec **new**
- **types de base jamais** créés avec **new**
- **static** que pour **variables de classe**
- **pas** de **sous-objets**

```
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");

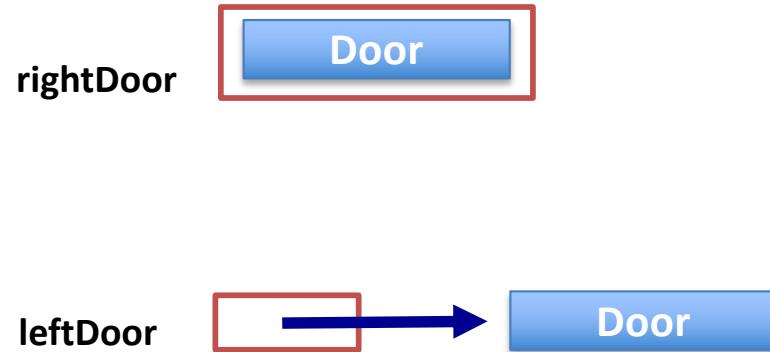
    ...
    delete i;
    delete s;
}
```

Java

Pointeurs vs sous-objets

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

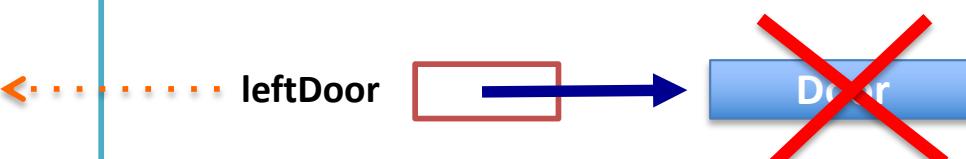
```
class Door {  
public:  
    Door(Car *);  
    ...  
};
```



Qu'est ce qui manque ?

Pointeurs vs sous-objets

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



Il faut un destructeur !

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

Copie d'objets

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

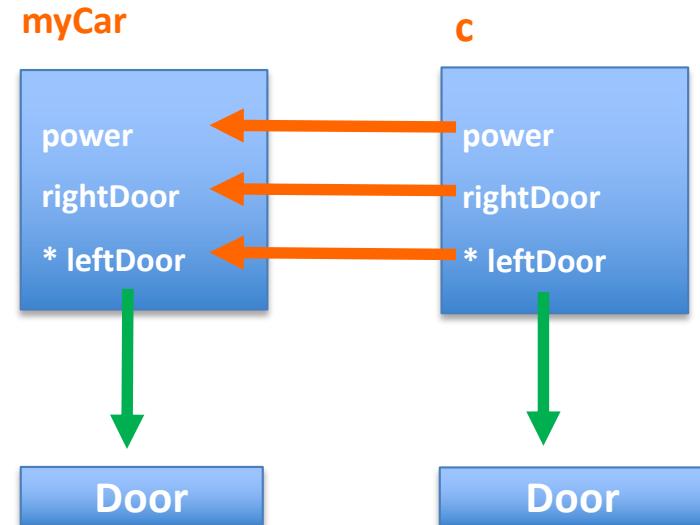
```
class Car : public Vehicle {  
    int power;  
    Door rightDoor;  
    Door * leftDoor;  
    ...  
};
```

affection (après la création)

initialisation (lors de la création)

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

Problème ?



Copie d'objets

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

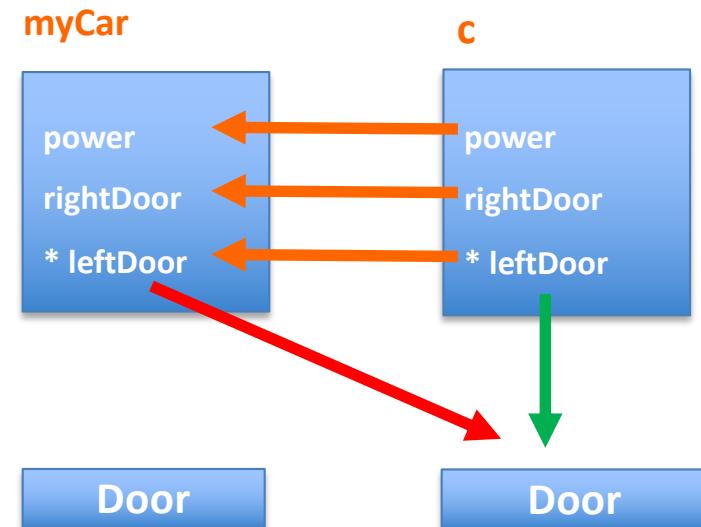
```
class Car : public Vehicle {  
    int power;  
    Door rightDoor;  
    Door * leftDoor;  
    ...  
};
```

myCar et *p ont la même porte gauche !
même chose !

Les pointeurs pointent sur le même objet !

- pas de sens dans ce cas !

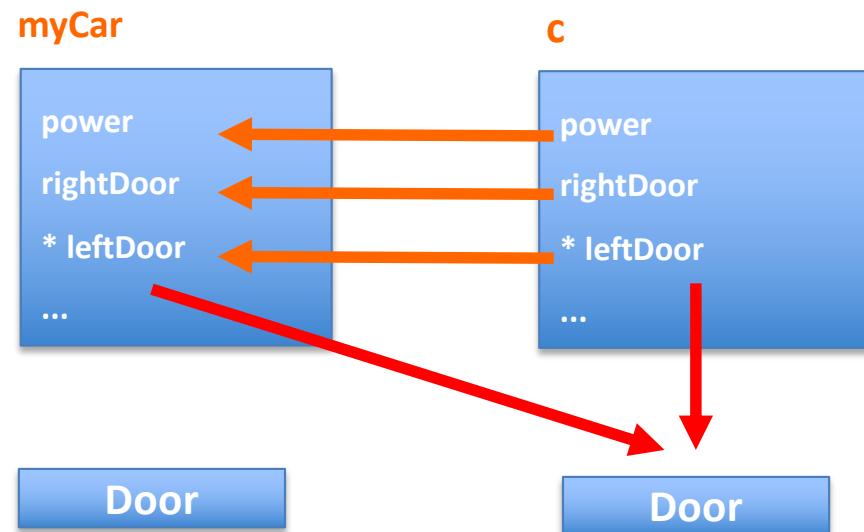
De plus l'objet Door est détruit 2 fois !



Copie superficielle et copie profonde

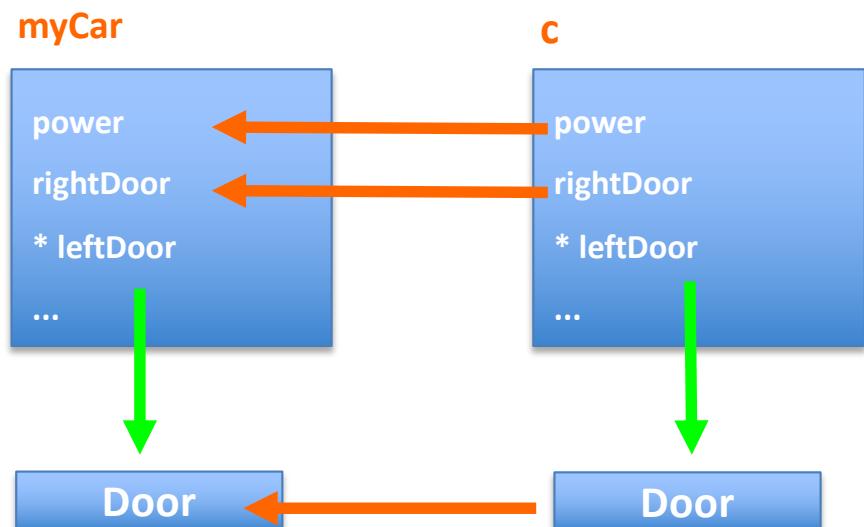
Copie superficielle (shallow)

- copie **champ à champ**
=> copie les **pointeurs**
- généralement **problématique**
si l'objet contient des **pointeurs**



Copie profonde (deep)

- copie les **pointés** récursivement



Solution : opérateurs de copie

On peut les **redéfinir** pour faire de la **copie profonde** :

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

Attention !

- si on change l'un il faut changer l'autre !

Copy constructor

- appelé quand on **crée** un objet
- existe en **Java**

```
Car c;  
Car myThirdCar(c);
```

Opérateur d'affectation

- appelé quand on **affecte**
- n'existe pas en **Java**

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

Opérateurs de copie

On peut aussi les **interdire** pour éviter des erreurs

- ca les interdit aussi dans les **sous-classes** (sauf s'ils sont redéfinis)

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

Opérateurs de copie

Exemple

```
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;  
}
```

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

ne pas oublier de copier
les champs de Vehicle !

Copie en Java

Même problème

- si l'objet contient des **références** (qui se comportent comme des pointeurs)

Différence

- en **Java** = ne permet de copier que les **références** (pas les **pointés**)

C/C++

```
Car * a = new Car();
Car * b = new Car();
Car * c = new Car(*b); <----- copy constructor ----->
a = b; <----- copie le pointeur ----->
*a = *b; <----- copie le pointé ----->
```

Java

```
Car a = new Car();
Car b = new Car();
Car c = new Car(b);
a = b;
a.copy(b);    si ces fonctions
a = b.clone(); sont définies
```

Car x(...); <----- n'existe pas en Java
Car y(...);
x = y; <----- copie l'objet

Coût de l'allocation mémoire

Coût négligeable

Mémoire **globale/static**

- fait à la **compilation**

Mémoire **automatique (pile)**

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

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

Sous-objets

- en général seul l'objet **contenant** est alloué

Coût de l'allocation mémoire

Coût non négligeable

Mémoire **dynamique** (**tas**) :

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

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

Peut être problématique

- **si très grand nombre d'allocations / destructions**
- **en C/C++** : **new** doit trouver de la place mémoire
 - varie selon l'état de la mémoire
- **en Java** : le **ramasse-miettes** "stops the world"
 - il existe **différent types** de ramasse miettes (cf. applications temps réel)
 - dans certains cas **new** alloue dans la **pile** pour optimiser !

Compléments: 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;

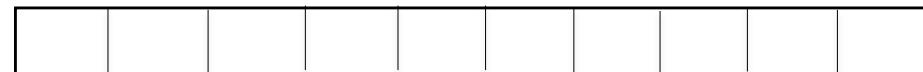
    double * p1 = new double[count];
    double * p2 = new double[count]();
    double * p3 = new double[count]{0., 1., 2., 3., 4., 5.};
    cout << p1[i] << " " << p2[i] << " " << p3[i] << endl;

    delete [] p1;
    delete [] p2;
    delete [] p3;
}
```

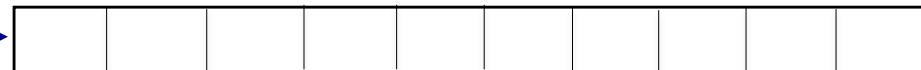
initialise à 0
C++11 seulement

ne pas oublier []

tab



p



Compléments: autres types de mémoire

Mémoire constante

- parfois appelée "**statique**" (mais rien à voir avec **static** !)
 - mot-clé **const** et littéraux: "Hello Word"
 - modification **interdite** !

Variabes volatile

- empêchent des **optimisations** du compilateur
 - utiles pour **threads** en **Java**
 - utiles dans certains cas (signaux, embarqué...) en **C++**

Variabes atomiques

- utiles pour **threads** : synchronisation correcte sans **mutex**
 - voir **std::atomic** en **C++**
 - voir **java.util.concurrent.atomic** en **Java**

Chapitre 4 : Types, constance & smart pointers

Types de base

Types standards

bool
char
short ←----- peuvent être
int signed ou unsigned
long
long long
wchar_t, char16_t, char32_t
float
double
long double

Types normalisés

int8_t
int16_t
int32_t
int64_t
intmax_t
uint8_t
uint16_t
uint32_t
uint64_t
uintmax_t
etc.

Pas portables !

- la taille dépend de la **plateforme** !
- **char** entre [0, 255] **ou bien** [-128, 127]
- tailles définies dans `<climits>` et `<cfloat>`

Portables

- même taille partout
- Définis dans `<cstdint>`

Alias de types

using crée un nouveau nom de type

- comme **typedef** en C mais plus puissant

```
using ShapePtr = Shape *;
```

```
typedef Shape * ShapePtr;
```

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

==

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

Inférence de types

auto => type inféré par le compilateur

auto count = 10;	int count = 10;
==	
auto PI = 3.1416;	double PI = 3.1416;

ShapeList shapes;	using ShapeList = std::list<Shape * >;
--------------------------	---

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

decltype => même type qu'une autre variable

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

Point * p = new Point();

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

Données constantes ou immuables

A quoi ça sert ?

- à éviter les **bugs** !
- à partager des données **sans risques**

Variables et paramètres **const**

- leur valeur **ne peut pas changer**

```
const int SIZE = 100;  
const char * HOST = "localhost";
```

final en Java

Enumérations

- pour définir des **constantes**
- existent en **Java**

```
enum {SIZE = 100};  
  
enum Status {OK, BAD, UNKNOWN};  
  
enum class Status {OK, BAD, UNKNOWN};
```

0, 1, 2 ... par défaut

Autres

- **macros** du langage **C** (obsolètes)
- **constexpr** : expressions constantes calculables à la **compilation**

Eviter les bugs

```
class Clients {  
    int maxSize = 100;           // taille max  
    int usedSize = 0;            // taille utilisée  
    Client * clients = new Client [maxSize];  
  
public:  
    void get(Client *);        // retourne le dernier  
    void add(Client);          // ajoute à la fin  
};
```

Ce code compile,
est-il correct ?

```
void Clients::get(Client * c) {  
    clients[maxSize++] = *c;  
}  
  
void Clients::add(Client c) {  
    c = clients[usedSize - 1];  
};
```

Eviter les bugs

```
class Clients {  
    int maxSize = 100;           // taille max  
    int usedSize = 0;            // taille utilisée  
    Client * clients = new Client [maxSize];  
  
public:  
    void get(Client *);        // retourne le dernier  
    void add(Client);          // ajoute à la fin  
    ...  
};
```



```
void Clients::get(Client * c) {  
    clients[maxSize++] = *c; <  
}; <  
  
void Clients::add(Client * c) {  
    c = clients[usedSize - 1]; <  
}; <
```

get() fait add() !!!
devrait être usedSize !!!
add() fait get() !!!

Eviter les bugs

```
class Clients {  
    const int maxSize = 100;  
    int usedSize = 0;  
    Client * clients = new Client [maxSize];  
  
public:  
    void get(Client *) const;  
    void add(const Client);  
    ...  
};
```

```
void Clients::get(Client * c) const { <.....  
    clients[maxSize++] = *c;  
};  
  
void Clients::add(const Client c) {  
    c = clients[usedSize - 1]; <.....  
};
```

3 erreurs de compilation :

get() const => erreur (ne peut changer clients)

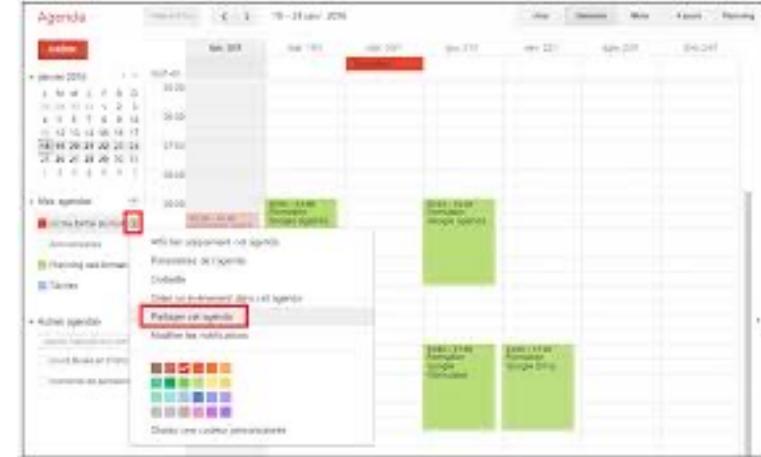
maxSize const => erreur

c const => erreur

Points de vue sur des données partagées

Exemple

- Alice a un calendrier partagé
- ses collègues peuvent le **lire**, pas le **modifier**
 - ⇒ Alice et ses collègues n'ont pas le même **point de vue** sur l'objet !
- cf. aussi tableaux du TP



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** ?

Points de vue sur des données partagées

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

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

Problème ?

Points de vue sur des données partagées

```
void pote(User* alice) {  
  
    Cal * c = alice->getCal(); // Pointe à l'objet  
  
    ...  
};
```

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

*pote peut modifier *c !*

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:  
    Cal* getCal() {return cal;}  
};
```

Solution 1 : objet immutable

- principe : aucune méthode de la classe **Cal** ne modifie l'objet
⇒ exemple : **String**, **Integer** en Java
- pas bon dans ce cas : **Alice** ne peut pas modifier son calendrier 😞

Const

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

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

Alice peut modifier (pas de const)

Autrui ne peut pas modifier (const)

const obligatoire car getCal() retourne const *

Solution 2 : pointé constant

const * => le pointé ne peut pas être modifié

on peut écrire:

const Cal* c ou: Cal const* c

Créer une copie

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

copie le calendrier

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

Solution 3 : créer une copie

- **limitation** : les deux objets vont diverger !
- **avantage** : la **copie** reste valide si l'**original** est **détruit** !
 - *cf. tableaux du TP : l'original peut être détruit ou modifié à tout moment !*

Pointeurs et pointés

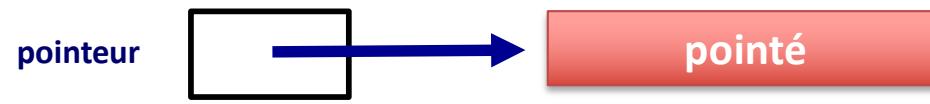
Qu'est-ce qui est constant : le pointeur ou le pointé ?

const porte sur « ce qui suit »

// le pointé est constant:

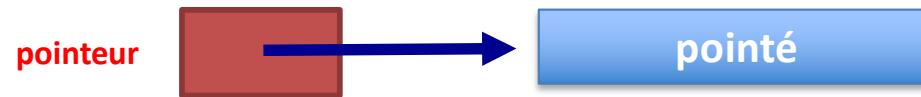
const char * s

char const * s



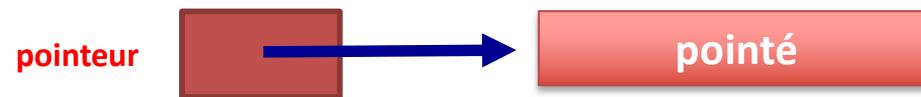
// le pointeur est constant:

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 de l'extérieur comme immuable

- pas de méthode permettant de le modifier : **constance logique**

Mais qui peut modifier son état interne

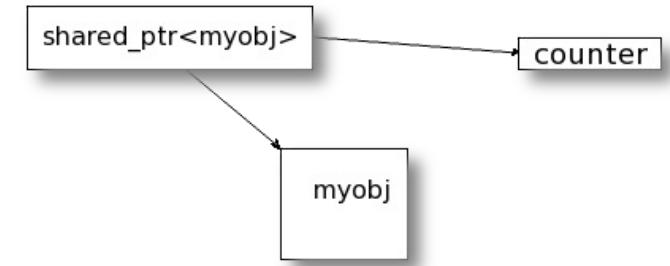
- **print()** peut allouer une ressource interne : **non-constance physique**

Smart pointers

```
#include <memory>

void foo() {
    shared_ptr<Circle> p( new Circle(0, 0, 50) );      // count=1
    shared_ptr<Circle> p2;
    p->setWith(20);
    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
```

smart pointer avec
compteur de références



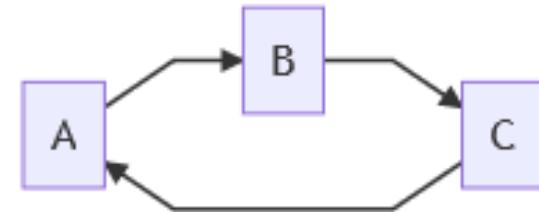
shared_ptr

- s'utilisent comme des "raw pointers" (pointeurs de base)
- l'objet s'**auto-détruit** quand le **compteur** arrive à 0
=> plus de **delete** ni de **pointeurs pendants** !

Smart pointers

Limitations

- objet **jamais détruit** si **dépendance circulaire**
=> utiliser **weak_ptr**
- doivent pointer sur des objets créés avec **new**
- **pas convertibles** en **raw pointers** (perd le compteur)



weak_ptr

- ne "possède" pas l'objet, permet de tester s'il **existe encore**

unique_ptr

- smart pointer **sans** comptage de références
- utiles si l'objet n'a qu'**un seul pointeur**,
en particulier pour les
tableaux et conteneurs

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

Chapitre 5 : Bases des Templates et STL

Programmation générique (templates)

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

i = mymax(4, 10);      <----- T vaut int
x = mymax(66., 77.);  <----- T vaut double
y = mymax<float>(66., 77.); <----- T spécifié, vaut float

string s1 = "aaa", s2 = "bbb";
string s = mymax(s1, s2);  <----- T vaut string
```

Les types sont des paramètres

- **mymax()** instanciée à la compilation comme si on avait défini **4 fonctions différentes**
- => algorithmes et types **génériques**
- existe en **Java (Generics)** mais avec des différences importantes

Classes génériques

```
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;
```

T peut être ce qu'on veut

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

définit: **operator+(a,b)**

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;
        }
    }
};

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;
}
```

passage par const référence
(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** mais aussi **types de base** (contrairement à **Java**)
- ex : **array, vector, list, map, set, deque, queue, stack ...**

Itérateurs

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

Algorithmes

- **manipulent** les éléments 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 :

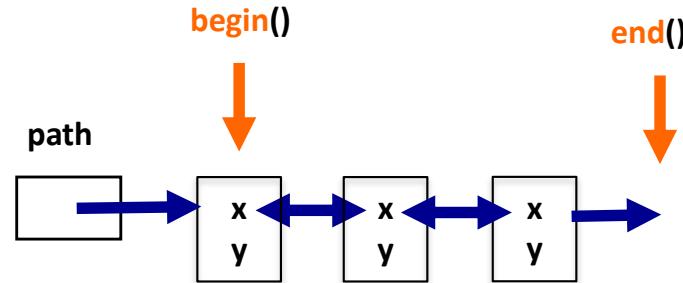
- **path[i]**
- **path.at(i)** : pareil mais vérifie l'indice (**exception** sinon)

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();
}
```



& est optionnel (généralement plus rapide)

équivaut à :

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

Pas d'accès direct aux éléments

- => utiliser des **itérateurs**

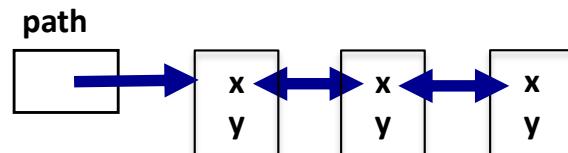
Insertion/suppression d'éléments

- en **temps constant**

Conteneurs et pointeurs

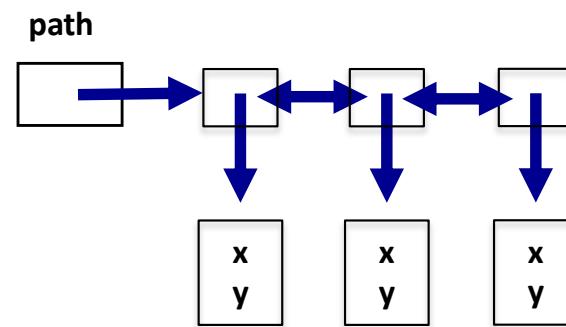
```
#include <list>
```

```
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();
}
```



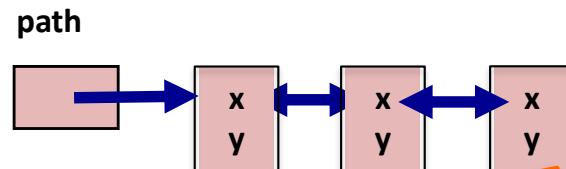
- A gauche : la **liste contient** les éléments
- A droite: la **liste pointe** sur les éléments

Problème ?

Conteneurs et pointeurs

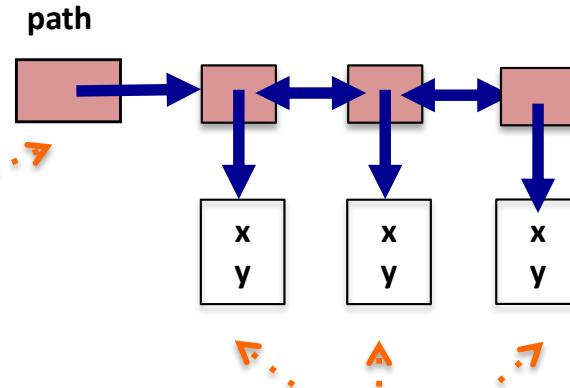
```
#include <list>
```

```
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();
}
```



Problème à droite !

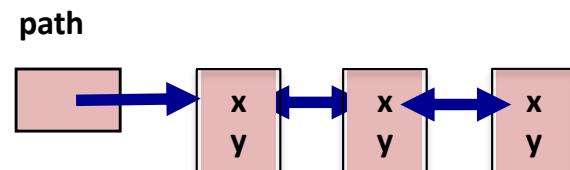
- la liste est bien **détruite** (car path est dans la **pile**)
- mais **pas les pointés** !

Solution ?

Conteneurs et pointeurs

```
#include <list>

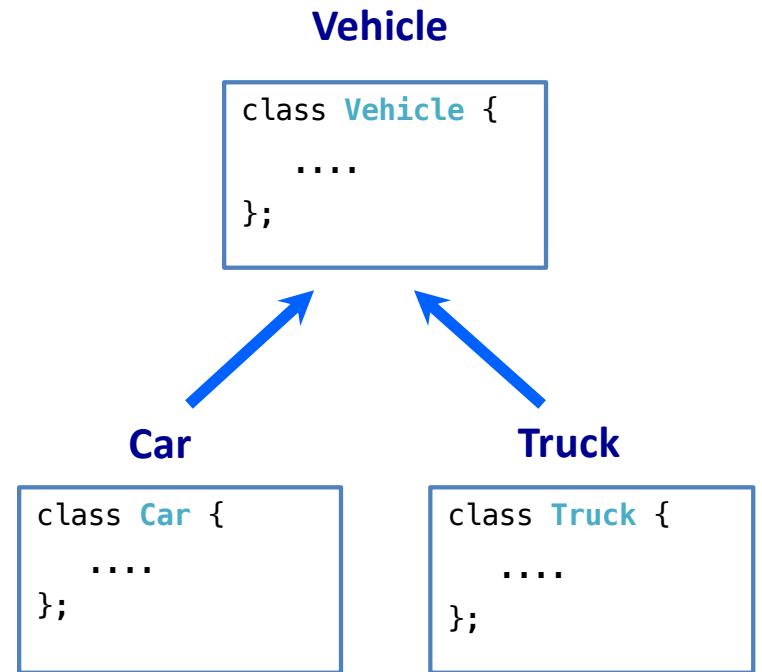
void foo() {
    std::list<Point> path;
    path.push_back(Point(20,20));
    path.push_back(Point(50,50));
    path.push_back(Point(70,70));
    for (auto & it : path) it.print();
}
```



Solution 1 : pas de pointeurs

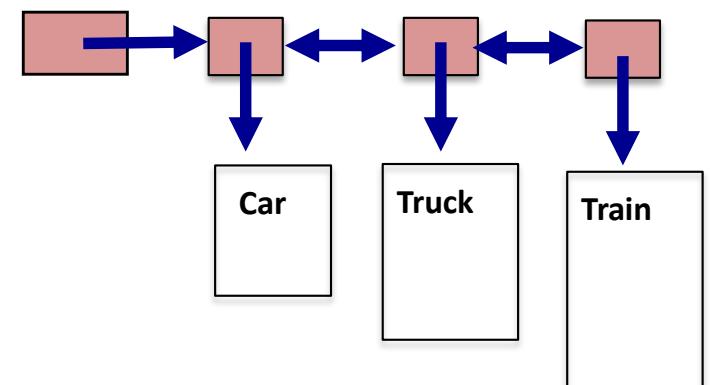
- **simple et efficace en mémoire**
- *limitation ?*

Conteneurs et pointeurs



Limitation Solution 1 (pas de pointeurs)

- **tous** les éléments doivent avoir **le même type**
- => **pointeurs** nécessaires si **polymorphisme**
(en **Java** c'est toujours des pointeurs !)



Conteneurs et pointeurs

Solution 2 : pointeurs + delete

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

équivaut à :

```
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
```

Enlever plusieurs éléments dans une liste

```
std::list<Point> path;  
int value = 200;           // détruire tous 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;             // => il faut deux itérateurs !  
    }  
}
```

Attention !

- l'itérateur **it** est **invalide** après **erase()**
=> il faut un **second itérateur** !

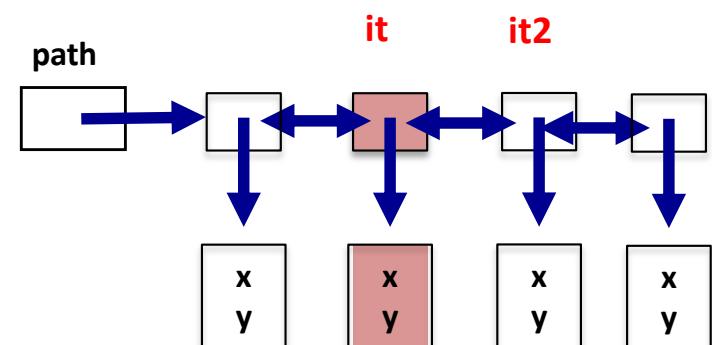


Table associative (map)

```
#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
        << "name: " << it->first
        << "id: " << it->second->getID()
        << std::endl;
    }
}
```

clé élément associé

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

it->**first** est la clé
it->**second** est l'élément associé

Note : on pourrait aussi utiliser le **conteneur set**

Array et Deque

Array : vecteur de taille fixe

- super efficace mais **taille fixe**

Deque ("deck") : hybride entre liste et vecteur

- accès direct aux éléments comme les listes
- insertion / suppression comme les listes
- peu utilisés (**vector** généralement plus efficace en mémoire et rapidité)

```
#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();
}
```

Algorithmes exemple : trier les éléments d'un conteneur

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

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

};

// inline 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 );
    ...
}
```

fonction de comparaison
de 2 éléments

Performance

xxx = **vector** ou **list**

on suppose que le conteneur contient **N** éléments
valant de 0 à **N-1** rangés au hasard

```
void foo(std::xxx<double>& c) {  
    for (int k = 0; k < 1000; ++k) {  
        auto it = std::find(c.begin(), c.end(), k);   
        c.insert(it, k);   
    }  
}
```

on cherche l'élément valant **k**

on insère **k** avant l'élément trouvé

- 1000 recherches
- 1000 insertions

qu'est-ce qui est plus rapide
xxx = vector ou xxx = list ?

Performance

xxx = **vector** ou **list**

on suppose que le conteneur contient **N** éléments valant de 0 à **N-1** rangés au hasard

```
void foo(std::xxx<double>& c) {  
    for (int k = 0; k < 1000; ++k) {  
        auto it = std::find(c.begin(), c.end(), k);   
        c.insert(it, k);   
    }  
}
```

on cherche l'élément valant **k**

on insère **k** avant l'élément trouvé

- 1000 recherches
- 1000 insertions

Théorie

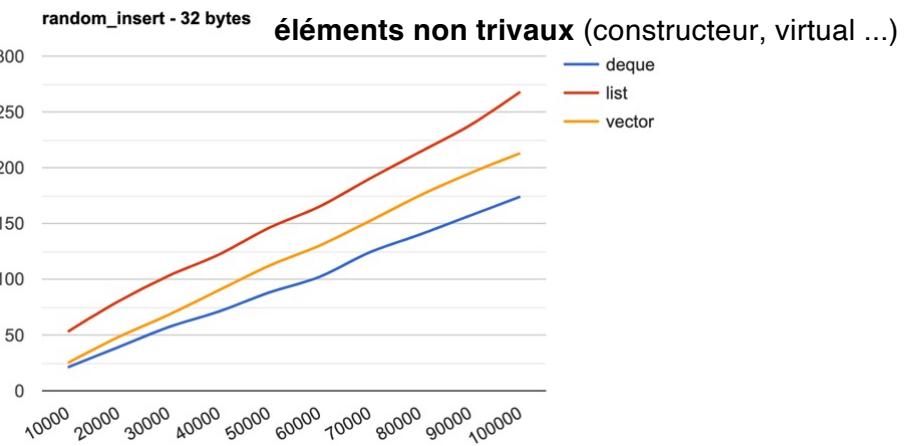
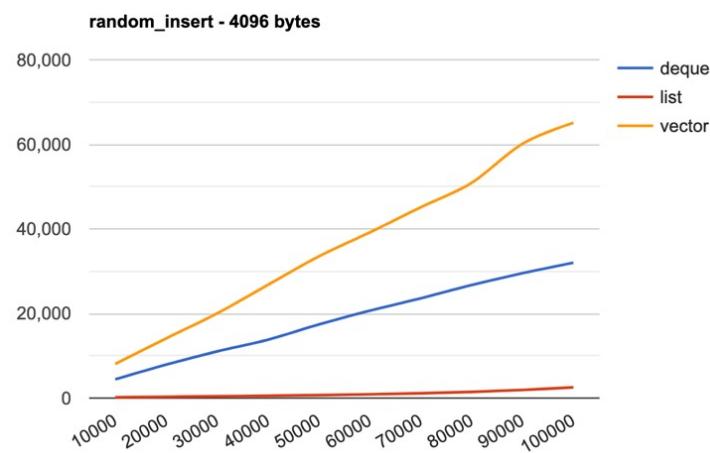
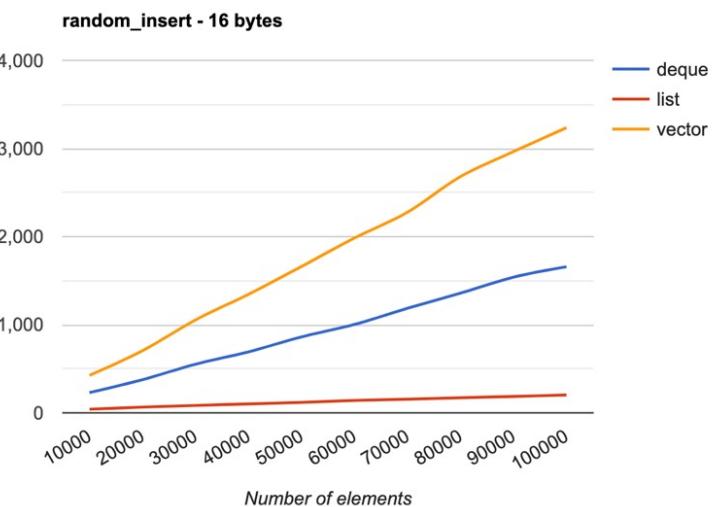
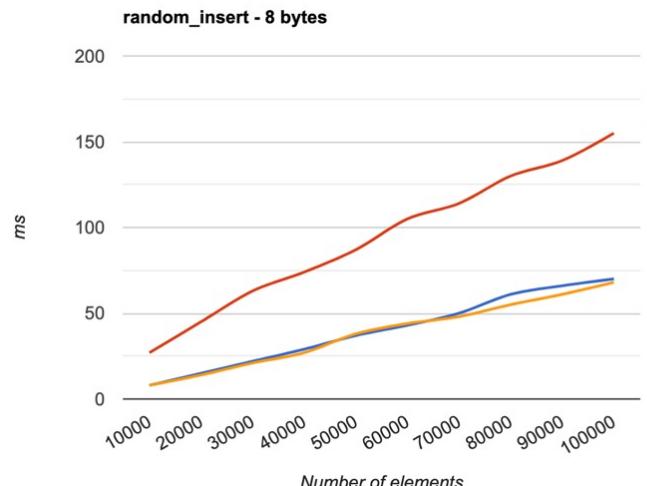
- recherche : $O(N)$ pour **list** et **vector**
- insertion:
 - $O(N)$ pour **vector**
 - $O(1)$ pour **list**

=> **list** plus rapide que **vector**

Performance: réalité

vector nettement plus rapide pour petits éléments triviaux

- à cause du **cache** (contiguïté des données)



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 spécifiques** (ici l'appel terminal)
- les **paramètres** peuvent être **des types de base**

Polymorphisme paramétré

Comment avoir une fonction `print()` générique ?

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

```
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);  
}
```

Polymorphisme paramétré

Comment avoir une fonction print() générique ?

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

```
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);  
}
```

Déjà vu : Polymorphisme de classes

- nécessite une **classe de base**
- pas possible pour **types de base**
- **coût** non négligeable si **beaucoup** d'objets (ex: jeux vidéo)

Autre solution : Polymorphisme paramétré

- **autre forme** de polymorphisme
- effectuée à la **compilation**

Polymorphisme paramétré

```
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);  
}
```

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); // print() est une méthode de Point  
}  
  
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);  
}
```

Amélioration

- 1 définition **générique** pour **tous** les **conteneurs** ?
- => pouvoir **déceler** que c'est des **conteneurs** !

Traits

```
is_array<T>           is_object<T>
is_class<T>            is_abstract<T>
is_enum<T>             is_polymorphic<T>
is_floating_point<T>   is_base_of<Base,Derived>
is_function<T>          is_same<T,V>
is_integral<T>          etc.
is_pointer<T>
is_arithmetic<T>
```

```
#include <type_traits>
```

Permettent des 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
}
```

Problème dans notre cas !

- `is_container<>` n'existe pas !
- comment le créer ?

Définir un trait

```
template <typename T> struct is_container { <----- cas général
    static const bool value = false;
};

template <typename T> struct is_container<std::vector<T>> { <----- spécialisations
    static const bool value = true;
};

template <typename T> struct is_container<std::list<T>> { <----- partielles
    static const bool value = true;
};
```

La définition correspondant au type est instanciée

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

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;
}
```



si **T** n'est pas un conteneur

```
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);
}
```



si **T** est un conteneur

```
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);
}
```

Seule la définition valide est instanciée

idiome **SFINAE**: "Substitution failure is not an error"

print() remplacé par définition valide pour **T**

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**
 - => **optimisations**
prenant compte des **types**
 - => **calcul sur les types**
à la compilation
- **puissants** (Turing complets)
....mais pas très lisibles ☺

Generics Java

Sémantique et implémentation différentes :

- pas pour les **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 {  
    void print(int n, const string * p) {  
        cout << n << " " << *p << endl;  
    }  
  
    void foo() {  
        int i = 10;  
        string * s = new string("YES");  
        print(i, s);  
    }  
};
```

C++

```
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);  
    }  
};
```

Java

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

```
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);  
    }  
};
```

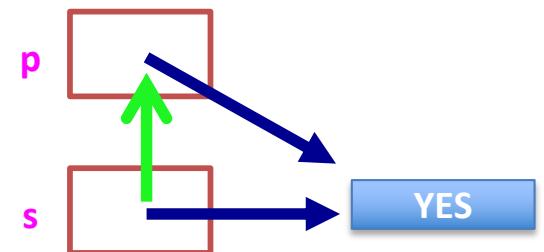
C++

```
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);  
    }  
};
```

Java

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** = pareil



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**

En général **const** n'est utile que s'il y a des **pointeurs** ou des **références**

Récupérer des valeurs d'une fonction

```
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;  
    }  
};
```

C++

```
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);  
    }  
};
```

Java

Résultat

- 10 YES
- 20 NO



Récupérer des valeurs d'une fonction

```
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;  
    }  
};
```

C++

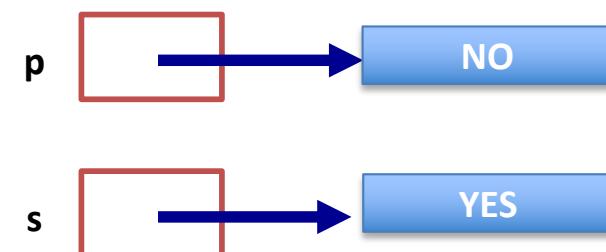
```
class Truc {  
    void get(int n, String p) {  
        n = 20;  
        p = new String("NO");  
    }  
  
    void foo() {  
        int i = 10;  
        String s = new String("YES");  
        get(i, s);  
        System.out.println(i + " " + s);  
    }  
};
```

Java

Résultat : 10 YES

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

Solution ?



Passage par référence

```
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

s

NO

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

Et en Java ?

Passage par référence

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

Java : types de base ET références passés par VALEUR

Le **passage par référence** (ou similaire) existe avec **C++, C#, Pascal, Ada ...**

Solution ?

Passage par (valeur de) pointeur

```
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;  
    }  
};
```

C++

modifie les pointés

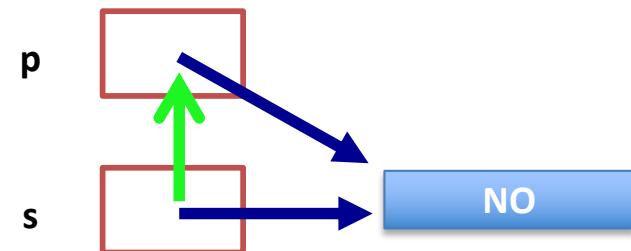
```
class Truc {  
    void get(StringBuffer p) {  
        p.replace(0, p.length(), "NO");  
    }  
  
    void foo() {  
        StringBuffer s = new StringBufer("YES");  
        get(s);  
        System.out.println(i + " " + s);  
    }  
}
```

Java

Solution : modifier les pointés

En Java :

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



Passage des objets

```
class Truc {  
    void print(vector<string> values) {  
        ...;  
    }  
  
    void foo() {  
        vector<string> v{"one", "two", "three", "four"};  
        print(v);  
    }  
};
```

C++

Problème ?

Passage par const référence

```
C++  
class Truc {  
    void print(vector<string> values) {  
        ...;  
    }  
  
    void foo() {  
        vector<string> v{"one", "two", "three", "four"};  
        print(v);  
    }  
};
```

```
C++  
class Truc {  
    void print(vector<string> const & values) {  
        ...;  
    }  
  
    void foo() {  
        vector<string> v{"one", "two", "three", "four"};  
        print(v);  
    }  
};
```

Problème :

v est entièrement recopié !

- pas efficace pour les gros objets
(e.g. les conteneurs)

Solution :
passage par const référence

- pas de copie
- pas de risque de modifier values par erreur

Retour par const référence

```
class Truc {  
    string name;  
  
    string getName() const {return name;}  
  
    void foo() {  
        string s = getName();  
        ...  
    }  
};
```

C++

```
class Truc {  
    string name;  
  
    string const & getName() const {return name;}  
  
    void foo() {  
        string s = getName();  
        ...  
    }  
};
```

C++

Même problème :

- **name** est **recopié** par **getName()**
- cette copie est **recopiée** dans **s**

=> pas efficace !

(en pratique c'est souvent optimisé)

Solution :

retour par **const référence**

- **pas de copie**
- **accès en lecture seule**

Références vs. pointeurs

Une référence est toujours initialisée et non nulle

- => évite des **crashes** !
- ou les **NullPointerException** de Java

```
void changeSize(Square * obj, unsigned int size) {  
    obj->setWidth(size); <----- il faudrait tester que obj n'est pas nul !  
}
```

```
void changeSize(Square & obj, unsigned int size) {  
    obj.setWidth(size); <----- rien à tester !  
}
```

Dans certains langages (e.g. Swift) pointeurs ou références "non-nullables"

Références vs. pointeurs

Une référence se comporte comme un alias
qui ne peut être changé

```
Circle c;
```

```
Circle & ref = c;
```

ref sera toujours un alias de cet objet

```
Circle c2;
```

```
Circle & ref2 = c2;
```

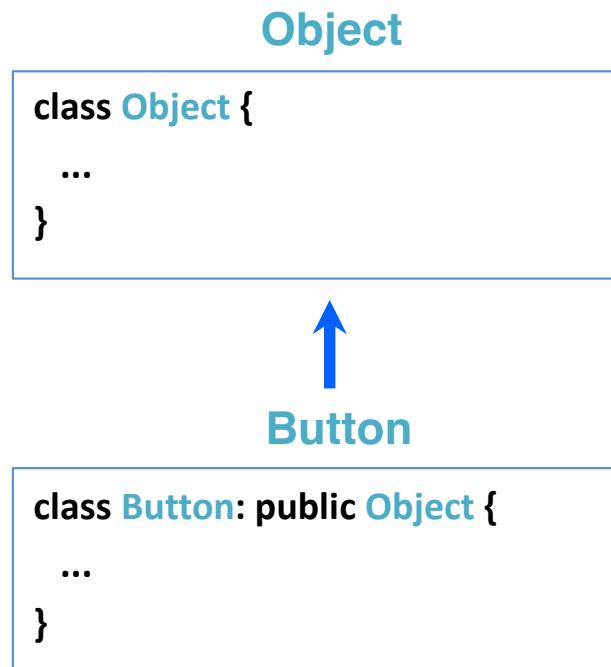
```
ref2 = ref;
```

même effet que : c2 = c; (copie d'objets)

Chapitre 7 : Compléments

Transtypage vers les superclasses

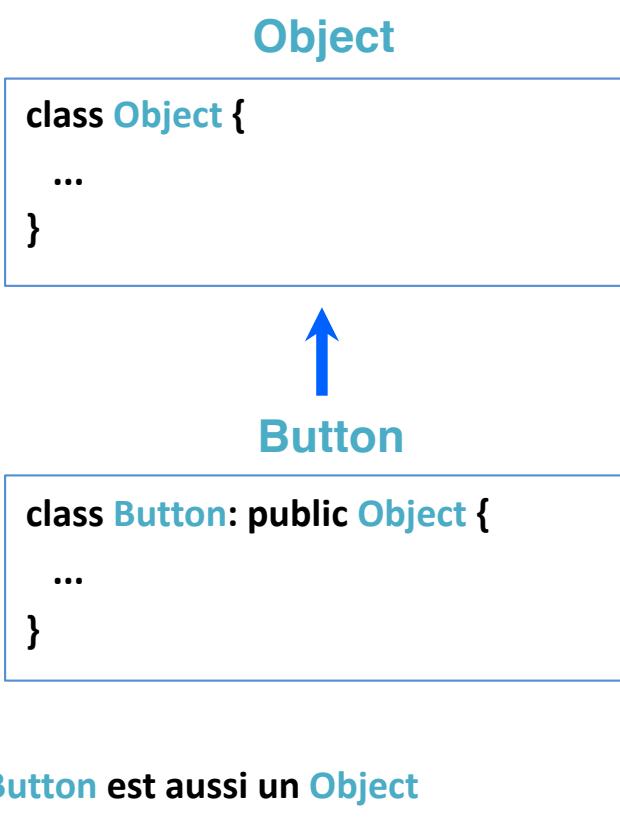
```
class Object {  
    ...  
};  
  
class Button : public Object {  
    ...  
};  
  
void foo() {  
    Object * obj = new Object();  
    Button * but = new Button();  
    obj = but;      // correct?  
    but = obj;      // correct?  
}
```



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;    <-----  
}
```



OK: upcasting : un `Button` est aussi un `Object`
ERREUR: downcasting
un `Object` n'est pas nécessairement un `Button`

Rappel: polymorphisme

- **transtypage implicite** vers les **super-classes** (**upcasting**)
- **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();   
}  
  
void bar() {  
    foo(new Button());  
}
```

Correct ?

Object

```
class Object {  
    ...  
}
```

Button

```
class Button: public Object {  
    virtual void draw();  
  
    ...  
}
```

Transtypage vers les sous-classes

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

Object

```
class Object {  
    ...  
};
```

Button

```
class Button: public Object {  
    virtual void draw();  
    ...  
};
```

erreur de compilation:
draw() n'est pas une méthode de Object

Solution ?

Transtypage vers les sous-classes

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

si on ne peut pas ajouter draw() ici
(ex. : cette classe est imposée)

mauvaise solution !
compile ...
... mais le compilateur ne vérifie plus rien !

```
void bar() {  
    foo(new Object());  
}
```



Transtypage vers les sous-classes

```
class Object {  
    // pas de méthode draw()  
  
    ...  
};  
  
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());  
}
```

bonne solution !
vérifie que c'est un Button
renvoie nullptr sinon

En Java le cast lancerait une exception
• tester avec instanceof()
• ou "catcher" ClassCastException

Opérateurs de transtypage

`dynamic_cast<Type>(b)`

- vérification du type à l'**exécution** : opérateur sûr

`const_cast<Type>(b)`

- pour enlever ou rajouter **const**

`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 !

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

Note : il y a des opérateurs spécifiques pour les `shared_ptr` (voir doc)

Types incomplets et handle classes

```
#include <Widget>

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

Correct ?

Références à des types inconnus

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

Handle classes pour cacher l'implémentation

- implémentation interne **cachée** dans `ButtonImpl` (pas donnée au client)

Types incomplets et handle classes

```
#include <Widget>

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

erreur de compilation:
types inconnus !

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;
};
```

mauvaise solution:

- dépendances croisées entre headers
- l'implémentation n'est plus cachée

Types incomplets

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

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

bonne solution: types incomplets

- déclarent l'**existence** d'une classe

OK avec **pointeurs** ou **références**

RTTI (typeid)

```
#include <typeinfo>

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

Retourne de l'information sur le type

- le nom est généralement **encodé** (mangled), exemples:
 - __ZTIN4guit7GButtonE
 - __ZNK4guit7GButton7getTypeEv

Pointeurs de fonctions

```
class Data {  
public:  
    std::string firstname, lastname;  
    int id{}, age{};  
};  
  
class DataBase {  
public:  
    Data search( std::function< bool(const Data&) > test ) const;  
    ....  
};
```

pointeur de fonction :

- prend un **Data&** en argument
- renvoie un **bool**

```
bool testAge10(const Data& d) {  
    return d.age > 10;  
}
```

base de données (noter le **&**)

```
void foo(const DataBase & base) {  
    Data found = base.search(testAge10);  
    ....  
}
```

renvoie premier **Data** vérifiant **test**

Limitation ?

Pointeurs de fonctions

```
class Data {  
public:  
    std::string firstname, lastname;  
    int id{}, age{};  
};  
  
class DataBase {  
public:  
    Data search( std::function< bool(const Data&) > test ) const;  
    ...  
};
```

pointeur de fonction :

- prend un `Data&` en argument
- renvoie un `bool`

```
bool testAge10(const Data& d) {  
    return d.age > 10;  
}  
  
void foo(const DataBase & base) {  
    Data found = base.search(testAge10);  
    ...  
}
```

Limitation :

il faudrait écrire une fonction
pour **chaque age** et pour **chaque critère** !

Lambdas

```
class Data {  
public:  
    std::string firstname, lastname;  
    int id{}, age{};  
};  
  
class DataBase {  
public:  
    Data search( std::function< bool(const Data&) > test ) const;  
    ...  
};
```

```
void foo(const DataBase& base) {  
    int age = 10; <----- age est capturé  
    Data found = base.search( [=] (const Data& d) { return d.age > age; } );  
}
```

Lambda : fonction anonyme qui capture les variables

- possède une **copie** des **variables locales**
- existent aussi en **Python, Java 8, etc.**

Lambdas et capture de variables

```
Data found = base.search( [=] (const Data& d) { return d.age > age; } );
```

Type de retour **implicite** mais on peut le **spécifier** :

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

Types de capture

- **[]** : capture **rien**
- **[this]** : capture **this**
- **[=]** : capture **this** et **variables locales** par **valeur** (copie)
- **[&]** : capture **this** et **variables locales** par **référence** (alias)
=> permet de **modifier les variables locales** (potentiellement **dangereux** !)
- **[age, &toto]** : capture **age** par **valeur** et **toto** par **référence**

Une possible implémentation de search

```
class Data {  
public:  
    std::string firstname, lastname;  
    int id{}, age{};  
};  
  
class DataBase {  
    std::list<Data> dataList;  
public:  
    Data search( std::function< bool(const Data&) > test ) const {  
        for (auto& it : dataList) {  
            if ( test(it) ) return it;  
        }  
        return Data(); // retourne Data vide  
    }  
};
```

Un autre exemple (callbacks)

```
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 );  
    btn->addCallback( [](Event&) {cout << "Done!" << endl;} );  
}
```

- quand le bouton est cliqué :
- callCallback est appelée :
 - ce qui exécute le callback

Compléments : Pointeurs de fonctions du C

pointeur de fonction du C
(noter l'*)

```
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);  
    }  
};
```

Compléments : Pointeurs de méthodes du C++

pointeur de méthode du C++
(noter l'*)

```
class Truc {  
    string result;  
public:  
    void dolt(Event& e) {  
        cout << "Result:" << result << endl;  
    }  
};
```

```
void foo() {  
    Truc * truc = new Truc();  
    Button * btn = new Button("OK");  
    btn->addCallback(truc, &Truc::dolt);  
}
```

```
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);  
    }  
};
```

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

Compléments : Foncteurs

- l'**objet** est considéré comme une **fonction**
- 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);  
}
```

```
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 !

on ne passe que l'objet** en argument**

Surcharge des opérateurs

```
#include <string>
```

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

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

Possible pour presque tous les opérateurs

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

operator[]

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

Exemples

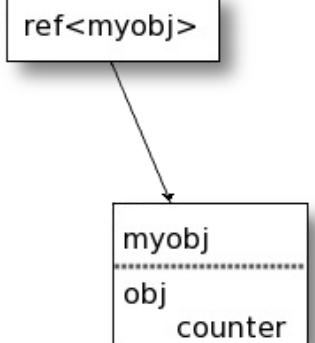
- `operator=`, `operator++`, `operator[]`, `operator()`
- `operator*`, `operator->`
- opérateurs `new` et `delete`
- **conversions** de types

operator++

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

Exemple : smart pointers intrusifs

```
void foo() {
    sptr<Shape> p (new Circle(0, 0, 50));
    p -> setX(20); // appelle operator->
} // p détruit (car dans la pile) => appelle ~sptr()
```



```
template <class T> class sptr {
    T* p;
public:
    sptr(T* obj) : p(obj) {add_ref(p);}
    ~sptr() {release_ref(p);}
    sptr& operator=(T* obj) {...}
    T* operator->() const {return p;}
    T& operator*() const {return *p;}
    ....
};
```

```
class Shape {
    unsigned long counter = 0;
    friend void add_ref(Shape* p);
    friend void release_ref(Shape* p);
};

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

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

- la classe de base `Shape` a un **compteur de références**
- les **smart pointers** redéfinissent la **copie** et le **déréférencement** pour **modifier le compteur**

Exceptions

```
class MathErr {};

class Overflow : public MathErr {};

struct Zerodivide : public MathErr {
    int x;
    Zerodivide(int x) : x(x) {}
};

void foo() {
    try {
        int z = compute(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 compute(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

Facilitent le traitement des erreurs

- remontent dans la pile des appels de fonctions
- jusqu'au **try / catch** correspondant
- **throw** renvoie l'exception au prochain **try / catch**

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

Avantages

- gestion **centralisée** et **systématique** des erreurs
- alternative aux **codes d'erreurs**

Inconvénients

- peuvent rendre le flux d'exécution **difficile à comprendre**
- ne pas en **abuser** et ne s'en servir que pour **gérer les erreurs**

Particularités de Java

- en **Java** les méthodes doivent **spécifier les exceptions** qu'elles peuvent envoyer
- pas en **C++**, ni dans la plupart des **autres langages**

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

- Cette particularité de **Java** pose autant (plus ?) de problèmes qu'elle n'en résout !
- Entre autres les **méthodes redéfinies**
 - ne peuvent pas envoyer **davantage d'exceptions**
 - sauf pour la classe **RuntimeException**

Particularités de C++

- **en C++ les exceptions ne sont pas forcément des objets**
 - ex : on peut renvoyer un **int**
- en général, **sous-classes** de **std::exception** ou **std::runtime_error**
 - header : <**exception**>
- autres exceptions standards:
 - **bad_alloc, bad_cast, bad_typeid, bad_exception, out_of_range ...**
- **handlers d'exceptions** : **set_terminate()**, **set_unexpected()**
 - ce qui se passe en dernier recours

Assertions

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

void changeSize(Square * obj, unsigned int size) {
    assert(obj);           ← .....     précondition
    obj->setWidth(size);
    assert(obj-&gtgetWidth() == obj-&gtgetHeight());   ← .....     postcondition
}
```

Tests en mode debug

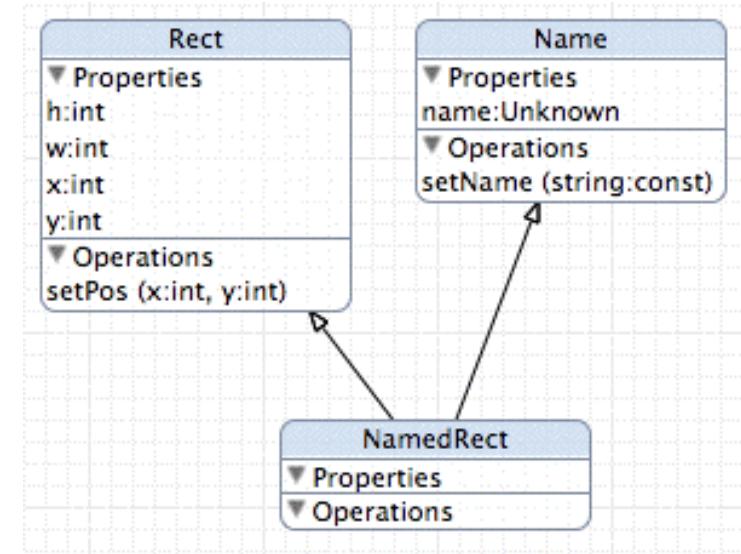
- en mode **debug** : **assert()** **aborte** le programme si **argument nul**
- en mode **production** : **assert()** ne fait rien
- mécanisme **basique** :
 - **attention aux plantages** si on le laisse en production !
 - **plus aucun test** si on l'enlève !
- voir aussi:
 - **tests unitaires** (ex : GoogleTest, CppTest, CppUnit)
 - **exceptions**
 - **contracts** (C++23)

dépend de la macro **NDEBUG**

- option de compilation **-DNDEBUG**
- ou : **#define NDEBUG**
avant : **#include <assert.h>**

Héritage multiple

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

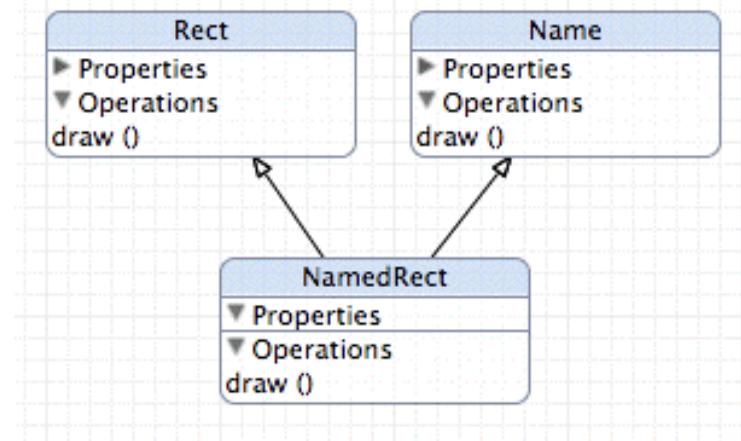


La classe **hérite** des variables et méthodes de **toutes ses superclasses**

Collisions de noms

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

Il faut les préfixer pour les distinguer



variables ou méthodes avec le même nom dans les superclasses

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

Collisions de noms

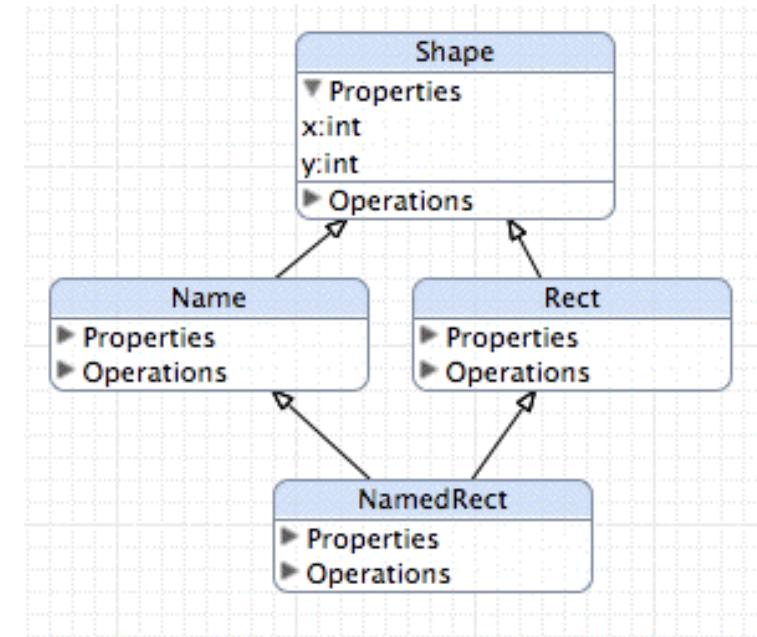
```
class Rect {  
    int x, y, w, h;  
public:  
    virtual void draw();  
};  
  
class Name {  
    std::string wname;  
public:  
    virtual void draw();  
};  
  
class NamedRect : public Rect, public Name {  
public:  
    void draw() override {  
        Rect::draw();  
        Name::draw();  
    }  
  
    // ou bien  
    using Rect::draw();  
};
```

Solutions

- redéfinir les méthodes
- ou choisir la méthode héritée avec using
- éviter les collisions de noms

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:  
    ....  
};
```



les variables sont dupliquées car
Shape est héritée des deux côtés !!!

en général ce n'est pas
ce qu'on veut !

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 avec les default methods des interfaces (depuis Java 8)

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 virtuel

- **pas de duplication** des variables
- 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 les **collisions**
- évites des **dépendances complexes** dans les hiérarchies de classes

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 contenante**

Pas en C++ !

Classes imbriquées (3)

```
class Car : public Vehicle {  
    class Door {  
        Car* car;    <----- pointe l'objet contenant  
        public:  
            Door(Car* car) : car(car) {}  
            virtual void paint();  
            ....  
    };  
  
    Door leftDoor, rightDoor;  
    string model, color;    <----- OK : paint() a accès à car->color  
    public:  
        Car(string model, string color)  
            : leftDoor(this), rightDoor(this) {}  
            ....  
    };
```

pointe l'objet contenant

OK : `paint()` a accès à `car->color`

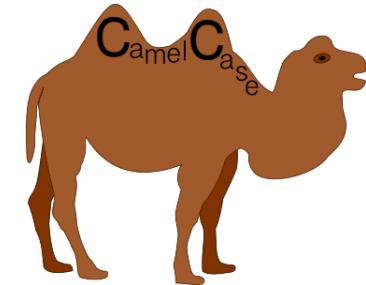
Solution (rappel)

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

Guides de style

Conseils généraux

- 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

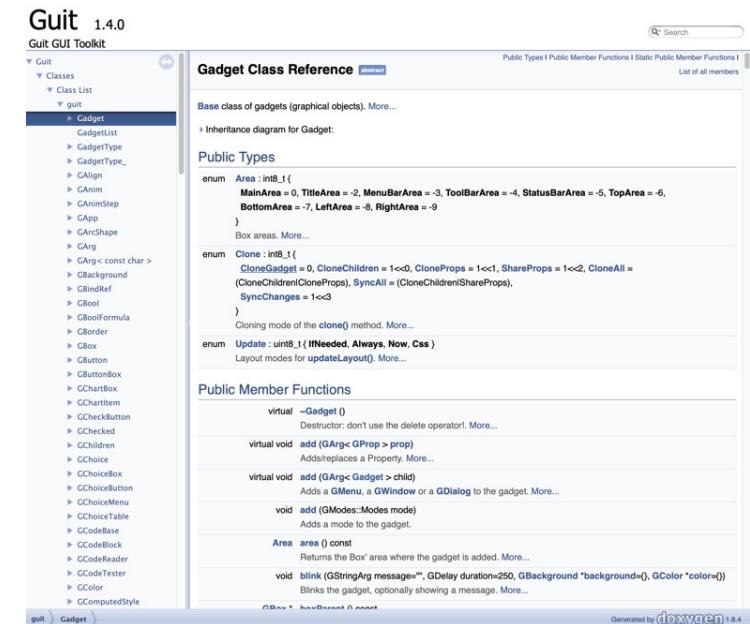


Guides pour la programmation C++

- **Google C++ Style Guide :**
<https://google.github.io/styleguide/cppguide.html>
- **C++ Core Guidelines :**
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

Documentation

```
/** @brief modélise un cercle.  
 * Bla bla bla (description détaillée).  
 */  
  
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 fonctionne avec de **nombreux langages**
- www.doxygen.org

Sérialisation

But

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

Cas d'usage

- **persistante** : sauvegarde / relecture sur/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 des 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)

```
#include <iostream>

class Vehicle {
public:
    virtual void write(std::ostream & f);
    virtual void read(std::istream & f);
    ...
};

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

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

Principe

- définir des fonctions d'écriture **polymorphiques**

ne pas oublier **virtual** !

chaîner les méthodes

Fichier:

```
whatever\n
whatever\n
Ferrari599GTO\n
670\n
whatever\n
whatever\n
Smart Fortwo\n
71\n
```

écrit par
Véhicule

écrit
par Car

Lecture avec espaces

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

Attention

- **>>** s'arrête au **premier espace** (' ', '\n', '\r', '\t', '\v', '\f')
- **getline()** lit **toute la ligne** (ou jusqu'à un caractère donné)

Fichier:

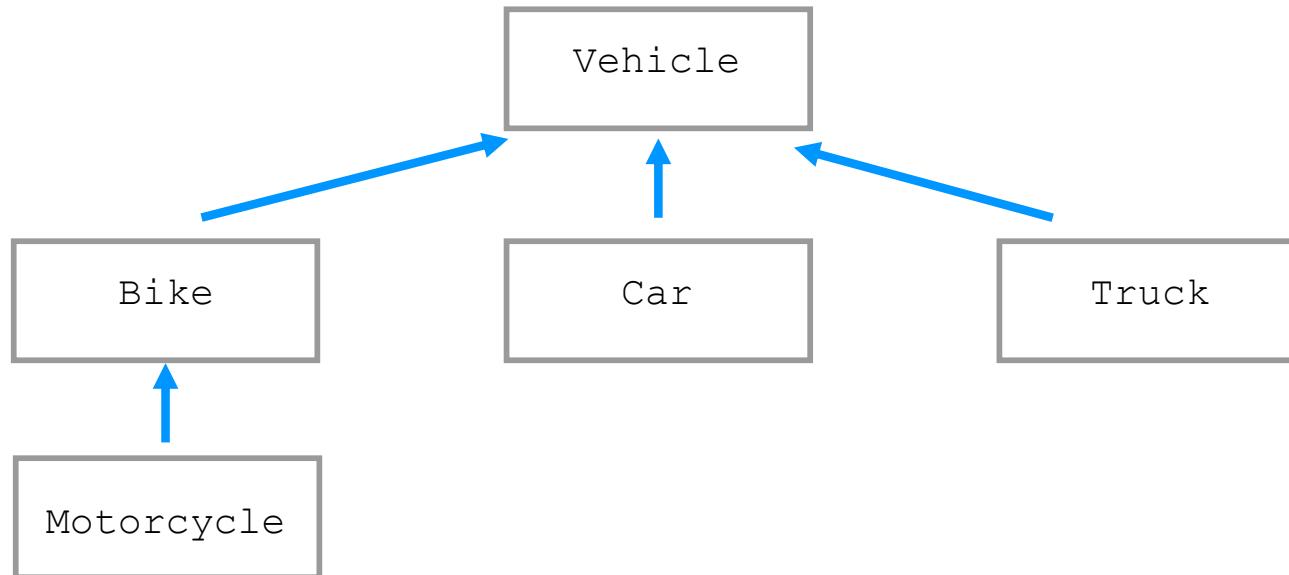
```
whatever\n  
whatever\n  
Ferrari 599 GTO\n670\n  
whatever\n  
whatever\n  
Smart Fortwo\n71\n
```

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

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 **type**



Classes polymorphes

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

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

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

```
bool saveAll(const std::string & filename, const std::vector<Vehicle *> & objects) {
    std::ofstream f(filename);
    if (!f) {           <----- vérifier que le fichier est ouvert
        cerr << "Can't open file " << filename << endl;
        return false;
    }

    for (auto it : objects) {
        f << it->classname(); <----- écrire la classe puis les attributs
        it->write(f);
        if (f.fail()) {
            cerr << "Write error in " << filename << endl; <----- erreur d'écriture
            return false;
        }
    }
    return true;
}
```



Lire des objets

passer par référence

```
bool readAll(const std::string & filename, std::vector<Vehicle *> & objects) {  
    std::ifstream f(filename);  
    if (!f) {  
        cerr << "Can't open file " << filename << endl;  
        return false;  
    }  
  
    while (f) { <----- tant que pas fin de fichier et pas d'erreur  
        std::string classname;  
        getline(f, classname);  
        Vehicle * obj = createVehicle(classname); <----- factory qui crée les objets  
        obj->read(f);  
        if (f.fail()) { <----- erreur de lecture  
            cerr << "Read error in " << filename << endl;  
            delete obj;  
            return false;  
        }  
        else objects.push_back(obj);  
    }  
    return true;  
}
```

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-GTO");
```

Compléments

Améliorations

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

Formats typiques

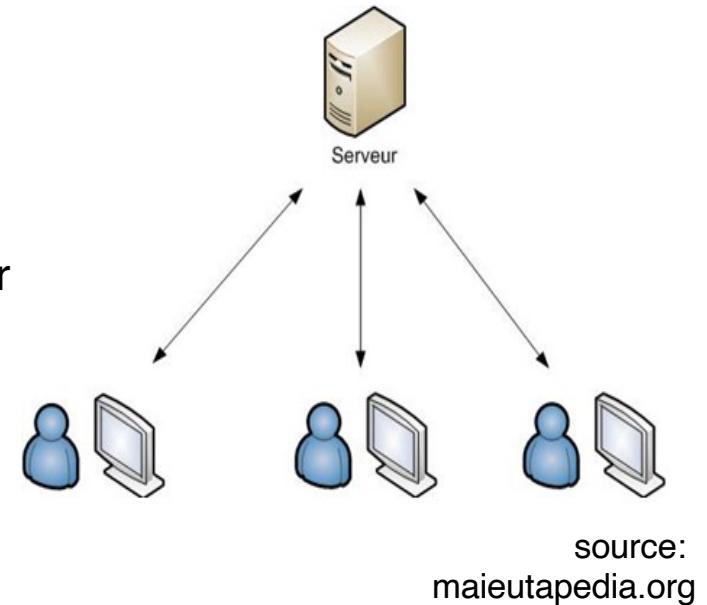
- pour sauver ou échanger sous forme textuelle
- **JSON** : JavaScript Object Notation
- **XML**
- **YAML**

```
{  
    "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



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

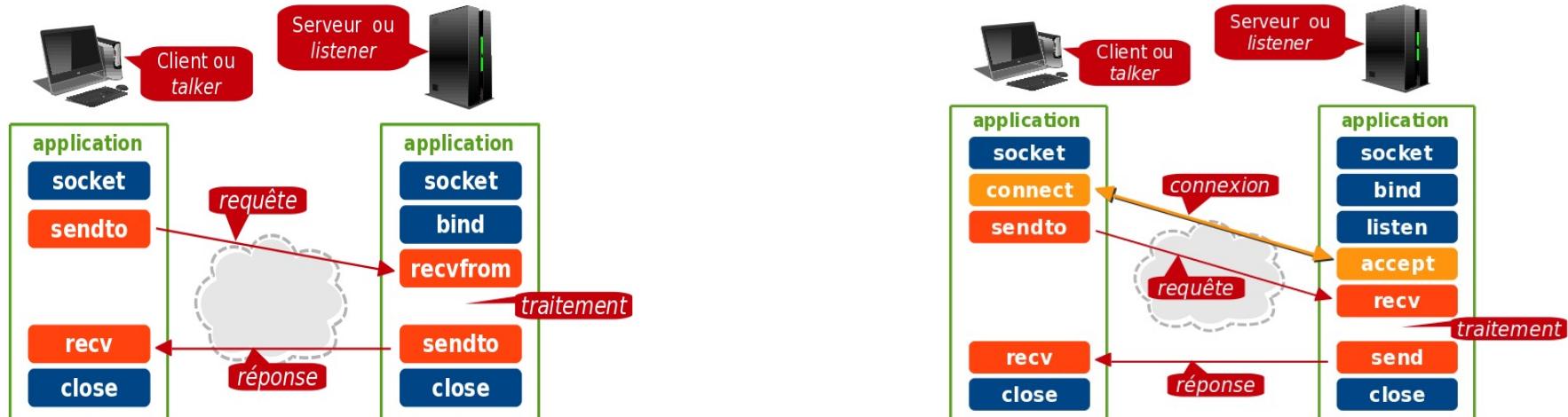
- le client **vague à ses occupations** après l'émission de la requête
- quand la réponse arrive une **fonction de callback** est activée
- exemples :
 - **thread** qui attend la réponse
 - **XMLHttpRequest** de **JavaScript**



Sockets

Principe

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



source: inetdoc.net

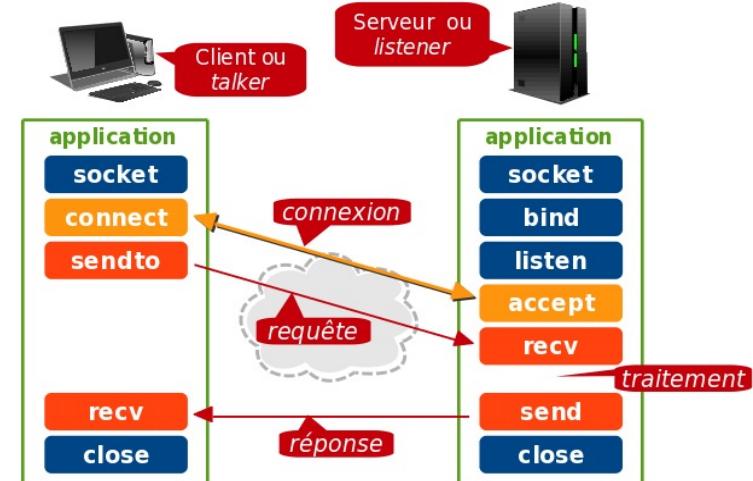
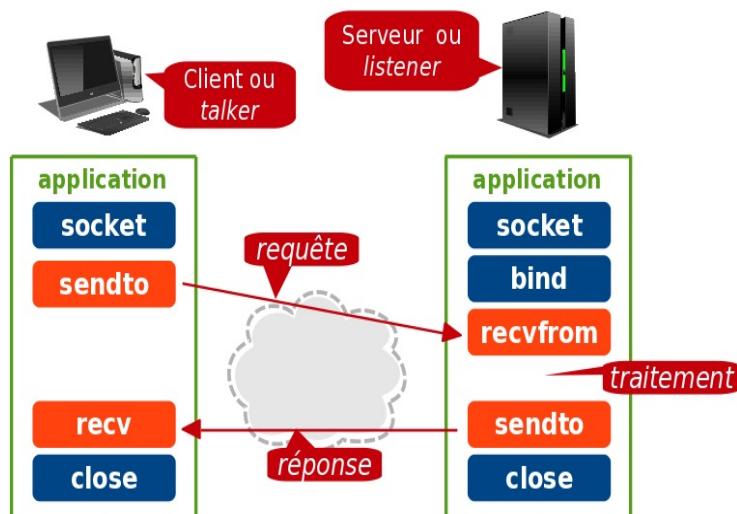
Sockets

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
- **pas de paquets perdus** et toujours dans l'**ordre**
 - ex : HTTP, TP INF224



source: inetdoc.net

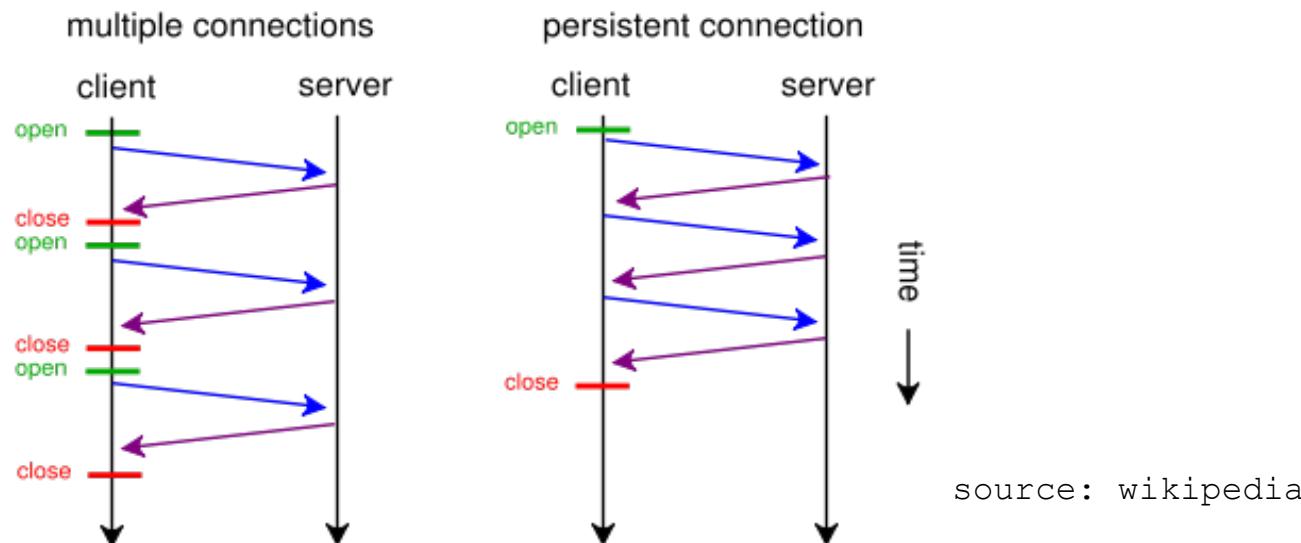
Sockets

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 de **flexibilité**, un peu **moins rapide**
- 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>
#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: 11111
Welcome dear customer ;-)

Adresses:

0x7fff5fbff98a 0x7fff5fbff98f
0x7fff5fbff998 0x7fff5fbff900

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

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; <----- string au lieu de char*

    cout << "Enter password: ";
    cin >> code; <----- pas de débordement :
                           taille allouée automatiquement

    if (code == CODE_SECRET) is_valid = true;
    else is_valid = false; <----- rajouter une clause else
                               peut éviter des erreurs

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

    return 0;
}
```

Mélanger C et C++

Règles

- **tout compiler** (y compris les fichiers C) avec **compilateur C++**
- **édition de liens** avec **compilateur C++**
- **main()** doit être dans un **fichier C++**
- **déclarer** les fonctions C comme suit pour pouvoir les appeler dans une fonction 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);

#endif
```

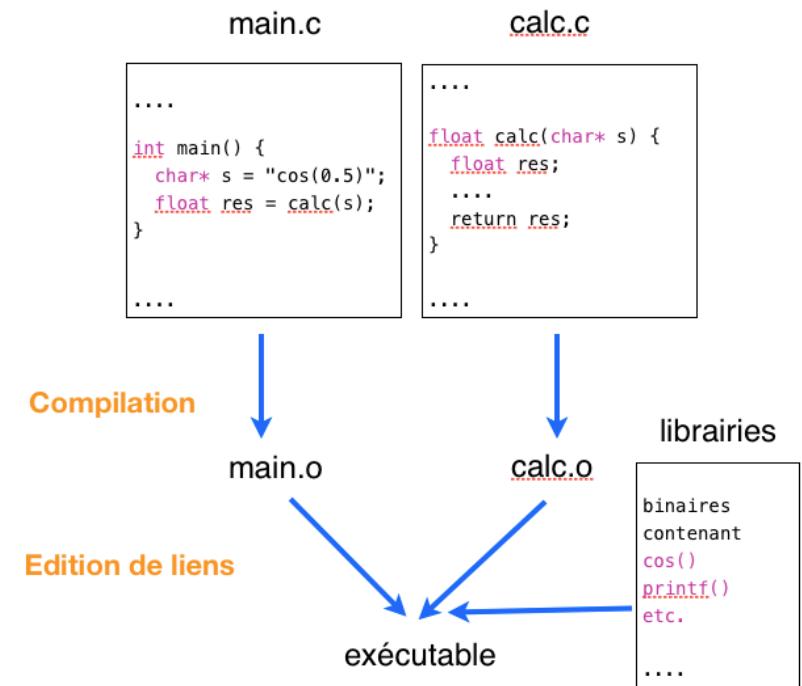
Librairies statiques et dynamiques

Librairies statiques

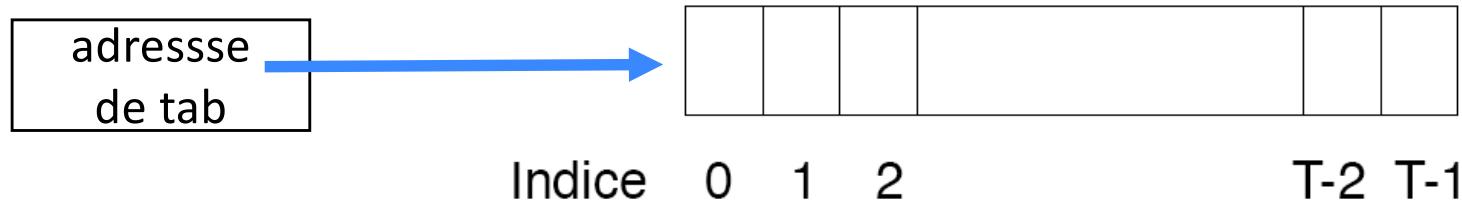
- code binaire **inséré dans l'exécutable**
- extension **.a**

Librairies dynamiques

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



Arithmétique des pointeurs



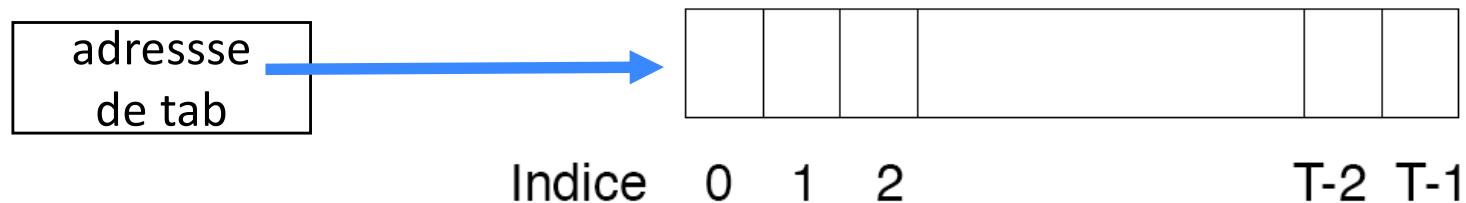
Tableaux

```
int tab[10];
tab[k] == *(tab + k)           // valeur du kième élément du tableau
&tab[k] == tab + k            // adresse du kième élément du tableau
```

Pointeurs : même notation !

```
int* p = tab;                  // équivaut à : p = &tab[0];
p[k] == *(p + k)              // valeur du kième élément à partir de p
&p[k] == p + k                // 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)