

Programmation orientée objet et autres concepts illustrés en C++11

Eric Lecolinet - Télécom ParisTech

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

12 janvier 2016

Dans ce cours

Organisation du cours

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

Deux langages support

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

Liens

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

Brève historique

Début 70 : Langage C

Début 80 : C++

Extension du **C** par *Bjarne Stroustrup* aux **Bell Labs**
C++11 : révision majeure en 2011/2014

Début 80 : Objective C

Autre extension du **C** de **NeXT** puis **Apple**
Puissant, syntaxe inhabituelle inspirée de **Smalltalk**

Fin 80... : Python

Vise la simplicité/rapidité d'écriture, créé par *G. van Rossum*
Interprété, typage dynamique

Mi-90 : Java

Simplification du **C++** de **Sun Microsystems** puis **Oracle**
Egalement inspiré de **Smalltalk**, **ADA** ...

Début 2000: C#

A l'origine, le « Java de **Microsoft** »
Egalement inspiré de **Delphi**, **C++** ...

Mi-2000: Swift

Le successeur d'**Objective C**, par **Apple**

C++ versus C et Java

C++ = extension du langage C

- un compilateur C++ peut **compiler du C** (avec qq restrictions)
- un même programme peut **combinaison C, C++ et Objective C** (Apple) ou **C#** (Windows)

C++, Java, C# dérivent de la syntaxe du C

- avec l'**orienté objet** et bien d'autres fonctionnalités en plus

Différences notables entre C++ et Java

- gestion mémoire, héritage multiple, redéfinition des opérateurs, pointeurs de fonctions et de méthodes, passage des arguments, templates ...
- programmes :
 - **Java** : à la fois **compilés (byte code)** puis **interprétés** ou compilés à la volée
 - **C/C++** : **compilés en code natif** (et généralement plus rapides)

Références et liens

Livres, tutoriaux, manuels

- **Le langage C++**, Bjarne Stroustrup (auteur du C++), Pearson
- **cplusplus** : www.cplusplus.com et www.cplusplus.com/reference
- **C++ reference** : <http://cppreference.com>
- **C++ (et C++11) FAQ** : <https://isocpp.org/faq>

Liens

- **Travaux Pratiques** de ce cours : www.enst.fr/~elc/cpp/TP.html
- **Petit tutoriel de Java à C++** (pas maintenu) : <http://www.enst.fr/~elc/C++/>
- **Toolkit graphique Qt** : www.enst.fr/~elc/qt
- **Questions/réponses** : <http://stackoverflow.com>
- **Extensions Boost** : www.boost.org
- **Cours C++ de Christian Casteyde** : <http://casteyde.christian.free.fr/>
- **Site de B. Stroustrup** : <http://www.stroustrup.com>

Premier chapitre :

Des objets et des classes

Programme C++

Constitué

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

Bonne pratique : une classe principale par fichier

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

myprog.cpp

```
#include "Car.h"
#include "Truck.h"

int main() {
    ....
}
```

Car.cpp

```
#include "Car.h"

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

Truck.cpp

```
#include "Truck.h"

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

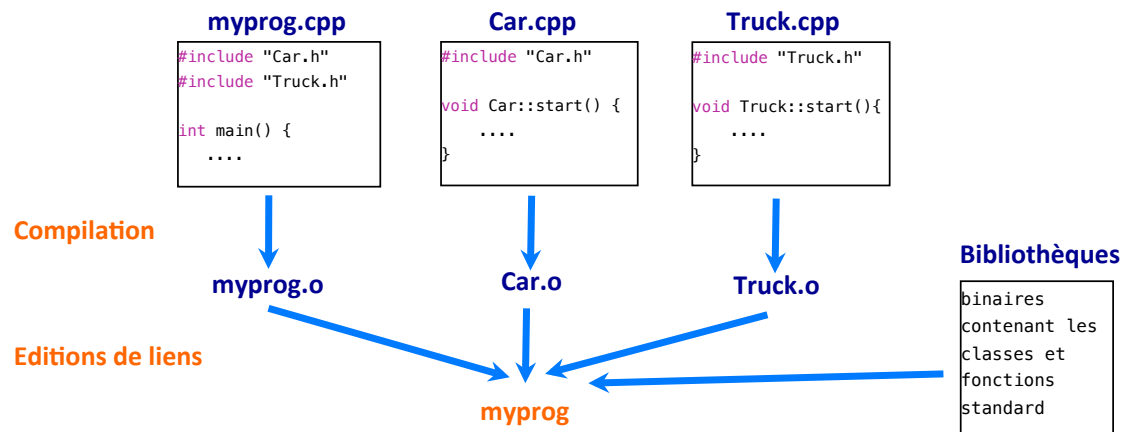
Compilation et édition de liens

Attention aux incompatibilités des compilateurs

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

Options g++

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



Eric Lecolinet - Télécom ParisTech - Programmation orientée objet et autres concepts illustrés en C++11

9

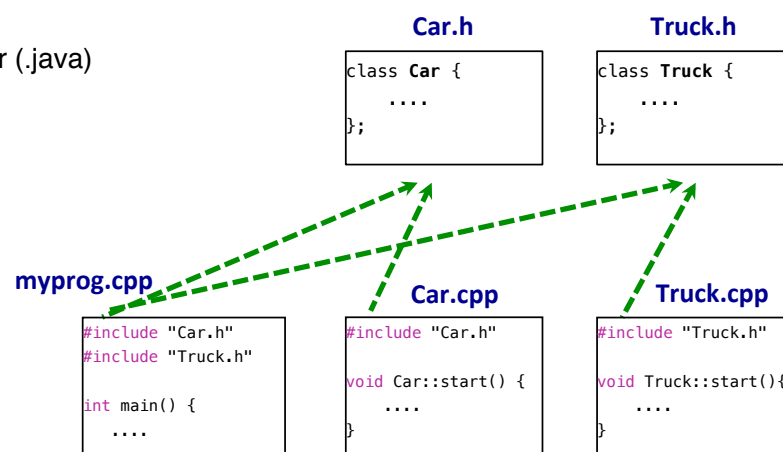
Déclarations et définitions

C/C++ : deux types de fichiers

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

En Java

- un seul fichier (**.java**)



Eric Lecolinet - Télécom ParisTech - Programmation orientée objet et autres concepts illustrés en C++11

10

Déclaration de classe

Dans le header **Circle.h** :

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

← variables d'instance

← constructeur

← méthodes d'instance

Remarques

- même sémantique que **Java** (à part **const**)
- il faut un **;** après la **}**
- **virtual** est généralement nécessaire (à suivre)

Variables et méthodes d'instance

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

← variables d'instance

← méthodes d'instance

Variables d'instance

- chaque objet possède **sa propre copie** de la variable

Méthodes d'instance

- ont automatiquement **accès aux variables d'instance et de classe**
 - **propriété fondamentale de l'orienté objet !**
- méthodes **const** : ne modifient pas les variables d'instance (n'existent pas en **Java**)

Constructeurs

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

Les constructeurs

- sont appelés quand les objets sont créés afin de les initialiser
- sont **toujours chaînés** :
 - les constructeurs des **superclasses** sont appelés dans l'ordre **descendant**
 - pareil en **Java**, **C#**, etc.

Définition des méthodes

Dans le fichier d'implémentation Circle.cpp :

```
#include "Circle.h"  
  
Circle::Circle(int _x, int _y, unsigned int _r) {  
    x = _x;  
    y = _y;  
    radius = _r;  
}  
  
void Circle::setRadius(unsigned int r) {  
    radius = r;  
}  
  
unsigned int Circle::getRadius() const {  
    return radius;  
}  
  
unsigned int Circle::getArea() const {  
    return 3.1416 * radius * radius;  
}
```

Rappel du header Circle.h :

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

insère le contenu de Circle.h

précise la classe :: typique du C++

ne pas répéter virtual

Définitions dans les headers

Déclarations + définitions dans Circle.h

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

← initialisations: C++11 seulement!

← méthodes inline

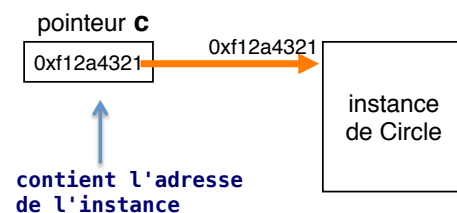
Méthodes inline

- **suggèrent** au compilateur de remplacer l'appel fonctionnel par son **code source**
 - **avantage** : exécution plus rapide
 - **inconvénients**: exécutables volumineux, ralentit la compilation, nuit à l'encapsulation
- **à utiliser avec modération !**
 - => petites méthodes appelées très souvent (accesseurs ...)

Instanciation

Dans un autre fichier .cpp :

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

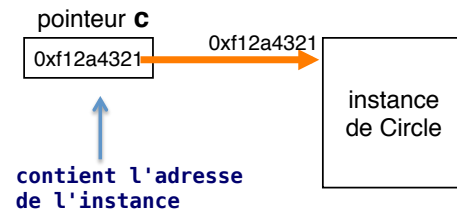


Instanciation

Dans un autre fichier .cpp :

```
#include "Circle.h"

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



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

- 1) alloue la mémoire
- 2) appelle le **constructeur**

c est une variable locale qui pointe sur cet objet

- **c** est un **pointeur** qui contient l'**adresse mémoire** de l'instance

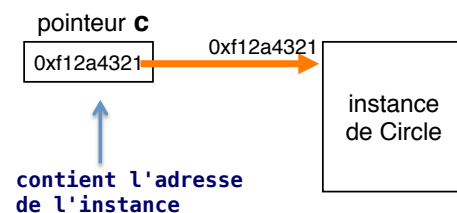
Pointeurs C/C++ vs. références Java

C++

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

Java

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



Référence Java = similaire à un pointeur C/C++ mais :

- pas d'*
- pas d'arithmétique
- valeur pas accessible
- ramasse miettes

Note : **référence** a un autre sens en C++ (à suivre)

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);  
    virtual void setRadius(unsigned int);  
    virtual unsigned int getRadius() const;  
    virtual 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; // interdit!  
    unsigned int area = c->getArea();  
}
```

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

Problème

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

Encapsulation

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

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

Encapsulation

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

Encapsulation : buts

Abstraire

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

Protéger l'intégrité de l'objet

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

Limiter les interdépendances entre composants logiciels

- pouvoir changer l'implémentation d'un objet sans modifier les autres

– **spécification** : déclaration des méthodes

- interface avec l'extérieur (API) => on ne peut interagir que **via les méthodes**

– **implémentation** : variables et définition des méthodes

- interne à l'objet => seul l'objet peut accéder à **ses variables**

Encapsulation : droits d'accès

Droits d'accès

- **private** : seuls les objets de cette classe ont **droit d'accès** (le **défaut** en C++)
- **protected** : également les sous-classes
- **public** : tout le monde
- ...

Encapsulation : droits d'accès

Droits d'accès

- **private** : seuls les objets de cette classe ont **droit d'accès** (le **défaut** en C++)
- **protected** : également les sous-classes
- **public** : tout le monde
- **friend** : droit d'accès pour **certaines classes** ou **certaines fonctions**

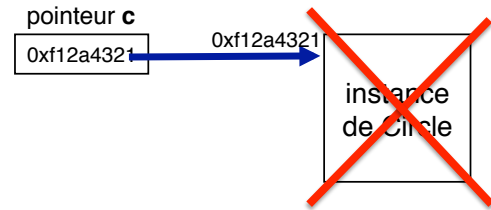
```
class Circle {  
    friend class ShapeManager;  
    friend bool isInside(const Circle&, int x, int y);  
    ...  
};
```

Remarques

- **Java** a aussi le niveau **package** (le défaut), mais pas de **friends**
- on peut encapsuler d'autres manières (par exemple en C)

Destruction des objets

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



delete détruit l'objet pointé par le pointeur c

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

Rappel: pas de ramasse miettes en C/C++ !

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

Destructeur

```
class Circle {  
public:  
    virtual ~Circle() {} ← destructeur  
    ...  
};  
  
void foo() {  
    Circle * c = new Circle(100, 200, 35);  
    ...  
    delete c;  
}
```

Le destructeur sert à "faire le ménage"

- il est appelé **juste avant** la destruction de l'objet
- utile s'il faut **finaliser ou détruire** des données créées par cet objet :
 - si *Circle* avait créé un autre objet devenu inutile
 - si *Circle* devait fermer un fichier ou une socket

En général il n'y en a pas !

- ex.: le destructeur `~Circle()` ne sert à rien !

Destructeur

```
class Circle {  
public:  
    virtual ~Circle() {} ← destructeur  
    ...  
};  
  
void foo() {  
    Circle * c = new Circle(100, 200, 35);  
    ...  
    delete c;  
}
```

Attention " faux ami "

- le destructeur ne détruit pas l'objet : il est appelé **avant** sa destruction !

Remarques

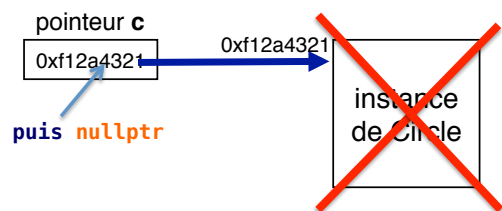
- les destructeurs sont **chaînés** et exécutés dans l'ordre **ascendant**
- les **classes de base** doivent généralement avoir un **destructeur virtuel** (à suivre)

Java

- **finalize()** joue le même rôle
 - mais pas de chaînage et appel non déterministe

Pointeurs nuls, pendants, indéfinis

```
void foo(Circle * c) {  
    unsigned int area = 0;  
    if (c) area = c->getArea();  
    else perror("Null pointer");  
    ...  
}  
  
void bar() {  
    Circle * c = new Circle(10, 20, 30);  
    foo(c);  
    delete c; // l'objet est détruit => c est "pendant"  
    c = nullptr; // c pointe sur rien  
    foo(c); // OK car c est nul sinon possible plantage !  
    delete c; // OK car c est nul  
}
```



Pointeur nul : pointe sur rien

- **nullptr** (C++11) ou **NULL** ou **0**

Penser à :

- **initialiser** les pointeurs
- les **mettre à nul** après **delete**

```
Circle * c; // indéfini DANGER !!!  
Circle * c = nullptr; // OK  
Circle * c = new Circle(); // OK
```

Précisions sur les constructeurs

Initialisation : 3 formes

```
Circle(int _x, int _y) {  
    x = _x; y = _y; radius = 0;    // comme Java (x,y,radius = variables d'instance)  
}
```

```
Circle(int x, int y)  
: x(x), y(y), radius(0) {}        // que C++: vérifie l'ordre, x(x) est OK
```

```
Circle(int x, int y)  
: x{x}, y{y}, radius{0} {}       // que C++11, compatible avec les conteneurs
```

Attention !

initialiser les **pointeurs** et les **types de base** sinon **leur valeur est indéfinie !**

```
Circle(int x, int y) : x(x), y(y) {}    // DANGER: radius est indéfini !
```

Surcharge (overloading)

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

Fonctions ou méthodes

- ayant le même nom mais des **signatures différentes**
- pareil en **Java**

Attention : méthodes d'une même classe !

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

Paramètres par défaut

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

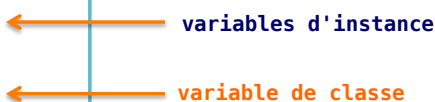
Alternative à la surcharge

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

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

Variables de classe

```
class Circle {  
    int x, y;  
    unsigned int radius;  
    static int count;  
public:  
    ...  
};
```



variables d'instance

variable de classe

Représentation unique en mémoire

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

Variables de classe (définition)

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

← doit être définie dans un .cpp

← pas besoin de la définir (C++11 seulement)

Les variables **static** doivent être définies

- dans un (et un seul) fichier .cpp

```
// dans Circle.cpp
```

```
int Circle::count = 0;
```

Exceptions

- si le type est **const int**
- en utilisant **constexpr** (C++11)

Méthodes de classe

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

← variable de classe

← méthode de classe

← appel de la méthode de classe

Ne s'appliquent pas à un objet

- mot-clé **static** comme en **Java**
- ont accès aux **variables de classe**
- de même nature que les **fonctions classiques** du **C** mais réduisent les **collisions de noms**

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 **package** de Java, existent aussi en **C#**

Using namespace

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; ← modifie la portée  
  
int main() {  
    Circle * mc = new Circle(); ← équivaut à math::Circle  
    graph::Circle * gc = new graph::Circle();  
}
```

using namespace

- modifie les **règles de portée** : symboles de ce **namespace** directement accessibles
- similaire à **import** en Java

Entrées / sorties standard

```
#include "Circle.h"
#include <iostream>
using namespace std;

int foo() {
    Circle * c = new Circle(100, 200, 35);

    cout << "radius: " << c->getRadius() << '\n' << "area: " << c->getArea() << endl;
}
```

entrées/sorties standard

std = bibliothèque standard du C++

↳
passe à la ligne
et vide le buffer

Flux standards

- **std::cout** sortie standard (console par défaut)
- **std::cerr** sortie des **erreurs** (non bufferisé : affichage immédiat)
- **std::cin** entrée standard

Entrées / sorties standard

```
#include "Circle.h"
#include <iostream>
using namespace std;

int foo() {
    Circle * c = new Circle(100, 200, 35);

    cout << "radius: " << c->getRadius() << '\n' << "area: " << c->getArea() << endl;

    string s;
    int id = 0;
    cout << "Entrez votre nom et votre ID: ";
    cin >> s >> id;
}
```

Flux standards

- **std::cout** sortie standard (console par défaut)
- **std::cerr** sortie des **erreurs** (non bufferisé : affichage immédiat)
- **std::cin** entrée standard

Flux d'entrées / sorties (streams)

```
#include "Circle.h"
#include <iostream>    // entrées/sorties
#include <fstream>     // fichiers
using namespace std;

void print(ostream & s, Circle * c) {
    s << c->getX() << ' ' << c->getY() << ' ' << c->getRadius() << ' ' << c->getArea() << endl;
}

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

noter le & (à suivre)

print() peut écrire sur la console ou un fichier

Flux (streams)

- **ostream** output stream (générique)
- **istream** input stream
- **ofstream** output file stream (pour fichiers)
- **ifstream** input file stream

stringstream

```
#include "Circle.h"
#include <iostream>    // entrées/sorties
#include <fstream>     // fichiers
#include <sstream>     // buffers de texte
using namespace std;

void print(ostream & s, Circle * c) {
    s << c->getX() << ' ' << c->getY() << ' ' << c->getRadius() << ' ' << c->getArea() << endl;
}

void foo() {
    Circle * c = new Circle(100, 200, 35);
    stringstream ss;
    print(ss, c);
    cout << ss.str() << endl;
    int x = 0, y = 0;
    ss >> x >> y;
    cout << "x=" << x << " y=" << y << endl;
}
```

print() peut écrire sur la console, un fichier ou un buffer !

Flux (streams)

- **stringstream** : buffer de texte en entrée/sortie
- Et aussi :
- **istringstream**, **ostringstream**

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

Toujours appliquées à un objet :

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

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

Mais pas la pourquoi ?

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

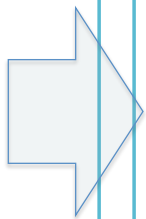
Comment la méthode accède à radius ?

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

Le *this* des méthodes d'instance

Le compilateur fait la transformation :

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



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

Le paramètre caché *this* permet :

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

Documentation

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

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

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

Doxygen : documentation automatique

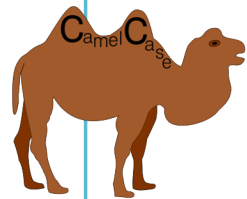
- similaire à **JavaDoc** mais plus général (fonctionne avec de nombreux langages)
- documentation : www.doxygen.org

Style et commentaires

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

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

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



Règles

- être **cohérent**
- **indenter** (utiliser un IDE qui le fait automatiquement : **TAB** ou **Ctrl-I** en général)
- **aérer**, éviter plus de 80 colonnes
- **commenter** quand c'est utile
- **camelCase**

Chapitre 2 : Héritage et polymorphisme

Héritage

Concept fondamental de l'orienté objet

- la sous-classe **hérite** automatiquement des **méthodes** et des **variables** de ses superclasses
 - **B** a une méthode **foo()** et une variable **x**

Héritage simple vs. multiple

- **héritage simple**
 - une classe ne peut hériter que d'**une** superclasse
 - Java, C#, Objective C ...
- **héritage multiple**
 - une classe peut hériter de **plusieurs** classes
 - C++, Python, Eiffel ...

Classe A

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



Classe B

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

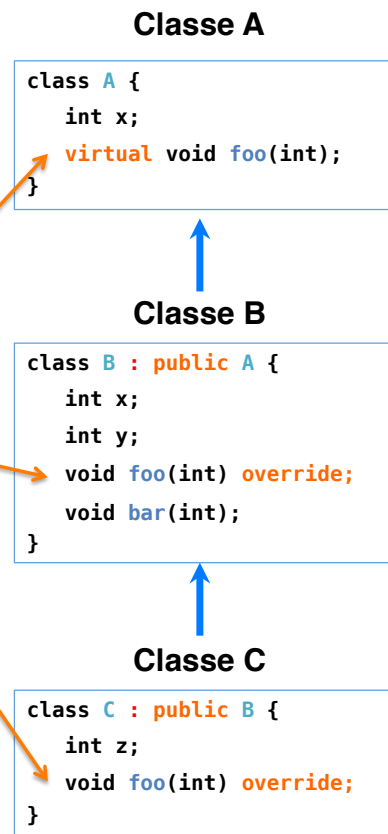
Règles d'héritage

Constructeurs

- pas hérités (mais **chaînés** !)

Méthodes

- **héritées**
- peuvent être **redéfinies** (overriding) :
 - la nouvelle méthode **remplace** celle de la superclasse
- noter :
 - **virtual** (à suivre)
 - **override** : redéfinition (C++11)
- **attention** : **même signature** sinon c'est de la **surcharge** !



Règles d'héritage

Constructeurs

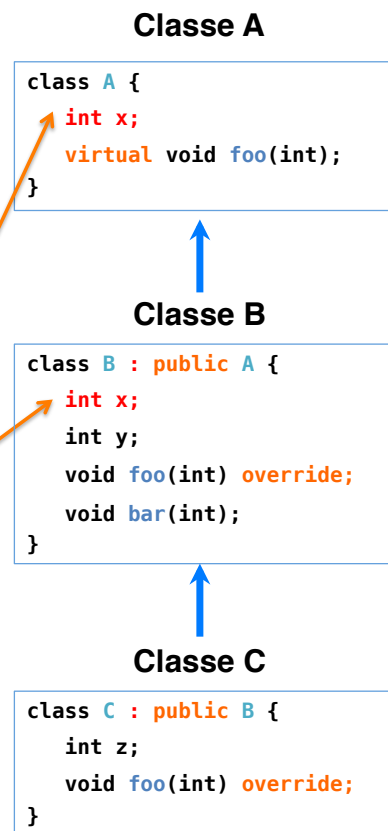
- pas hérités (mais **chaînés** !)

Méthodes

- **héritées**
- peuvent être **redéfinies** :

Variables

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



Exemple

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

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

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

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

Rect

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

Square

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

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

Redéfinition de méthode
=> **override** en C++11

***Pourquoi faut-il
redéfinir ces
deux méthodes ?***

Exemple

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

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

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

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

Rect

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

Square

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

**sinon ce n'est
plus un carré !**

Chaînage des constructeurs

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

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

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

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

Chaînage **implicite** des constructeurs
=> appelle **Rect()**

Chaînage **explicite** des constructeurs
=> comme **super()** de Java

Remarques

Chaînage des constructeurs

<code>Square::Square() : Rect() {}</code>	chaînage explicite du constr. de la superclasse
<code>Square::Square() {}</code>	chaînage implicite : fait la même chose
<code>Square::Square(int x, int y, unsigned int w) : Rect(x, y, w, w) {}</code>	même chose que super() de Java

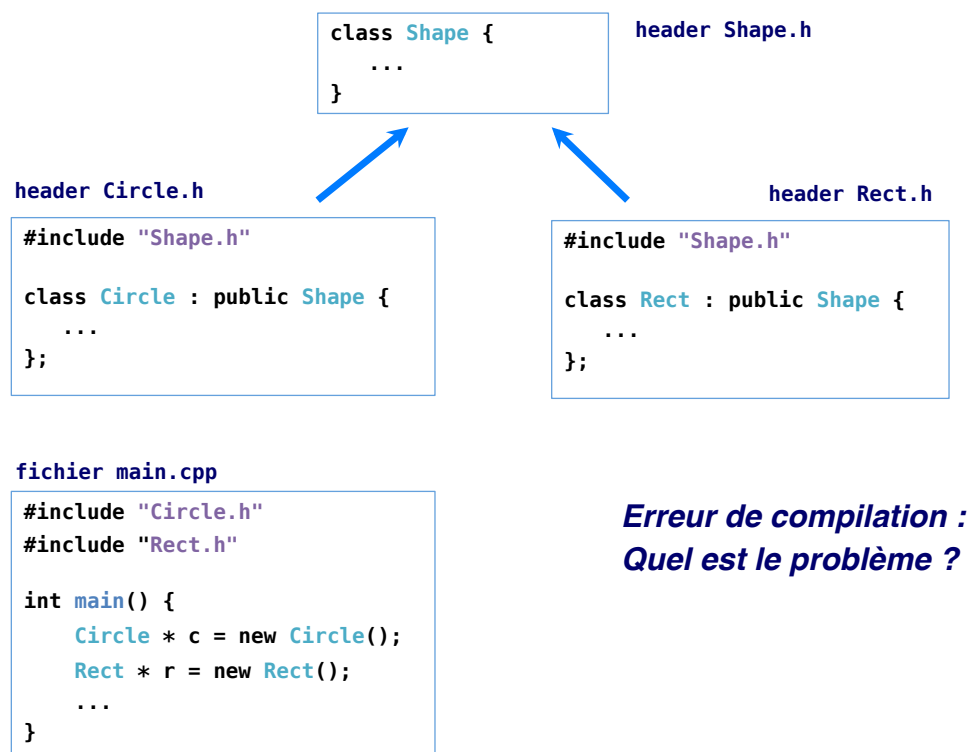
Covariance des types de retour

- redéfinition de méthode => même **signature**
- mais **Muche** peut-être une sous-classe de **Truc**

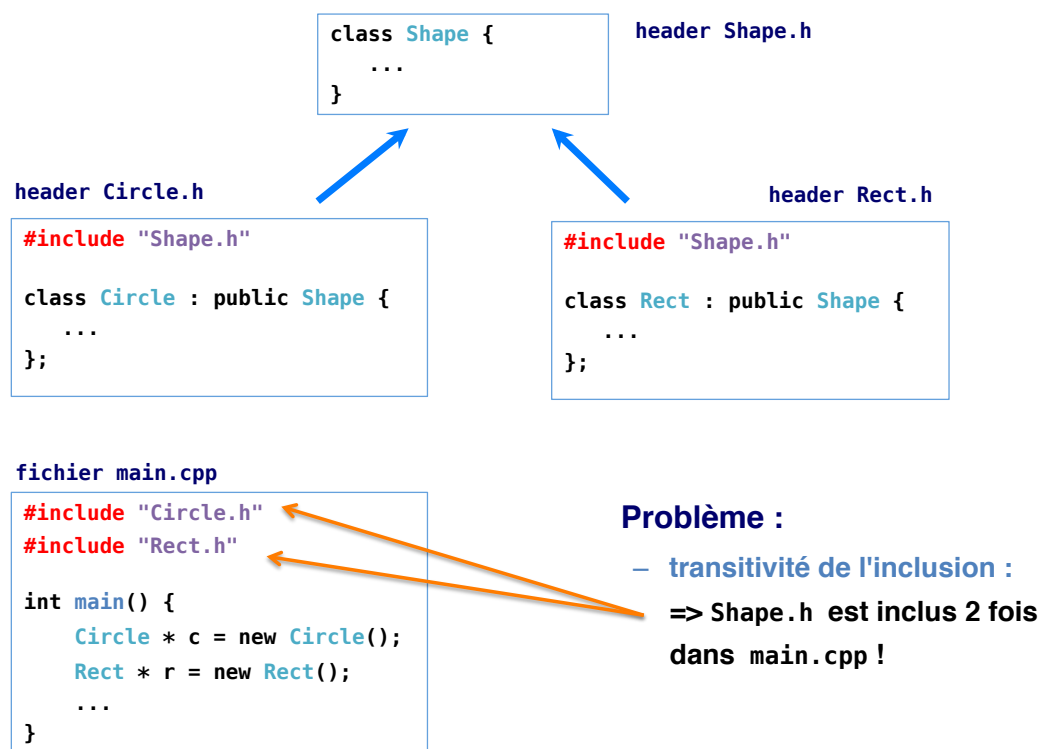
```
class A {
    virtual Truc * makeAux();
    ...
}

class B : public A {
    virtual Muche * makeAux();
    ...
}
```

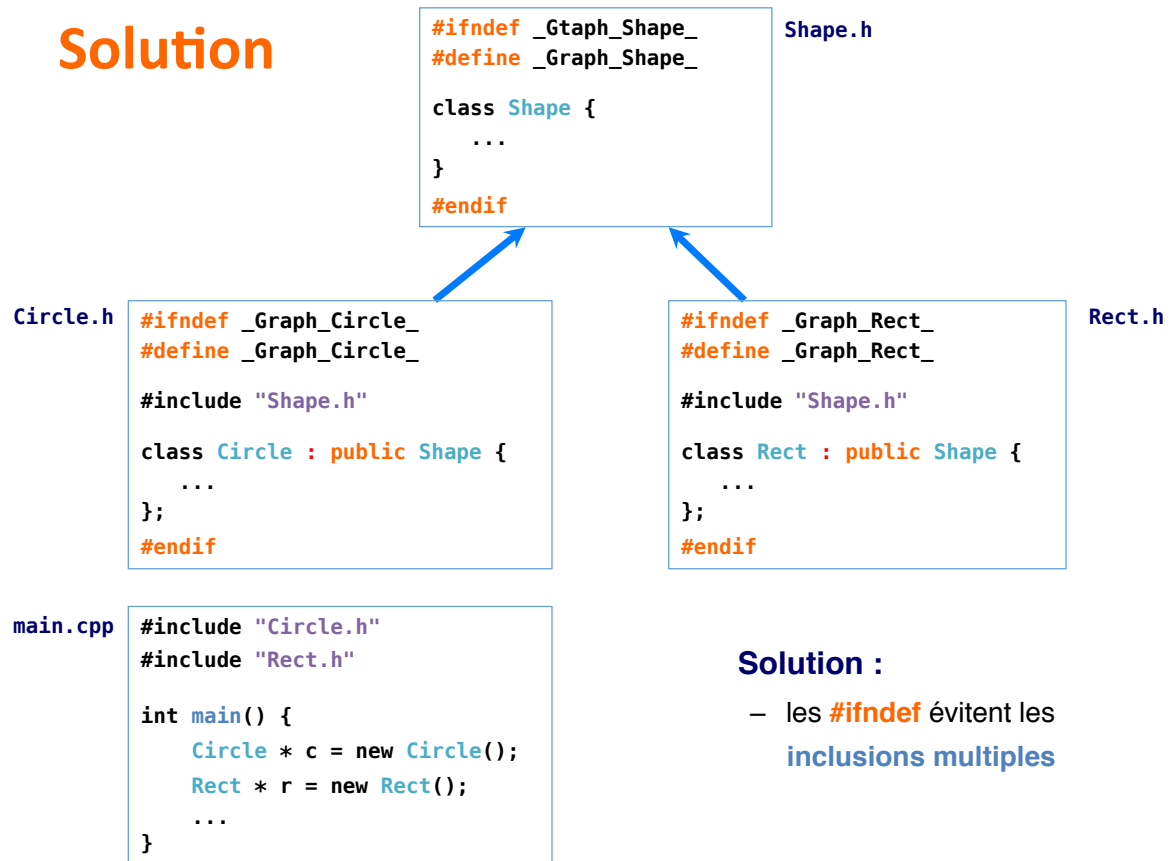
Classes de base



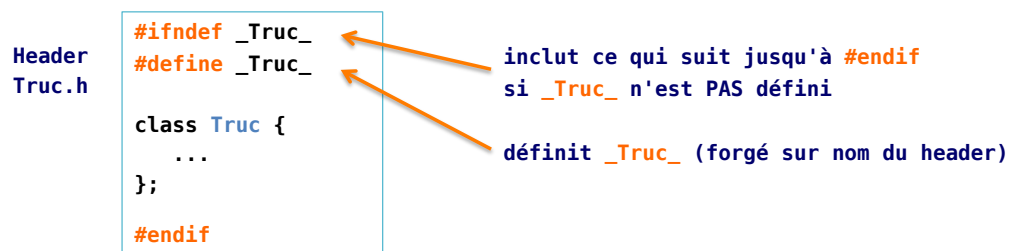
Classes de base (problème)



Solution



Préprocesseur et directives de compilation



- **#if** / **#ifdef** / **#ifndef** permettent la **compilation conditionnelle**
- **#import** empêche l'inclusion multiple (mais pas standard)

Localisation des headers

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

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

Polymorphisme de type

Dernier concept fondamental de l'orienté objet

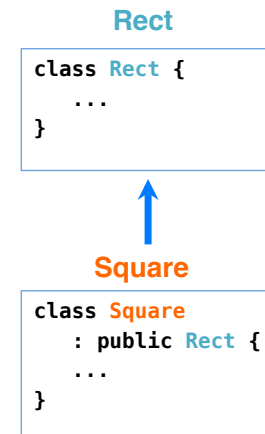
- le plus puissant mais pas toujours le mieux compris !

Un objet peut être vu sous plusieurs formes

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

```
#include "Rect.h"

void foo() {
    Square * s = new Square();    // s voit l'objet comme un Square
    Rect * r = s;                 // r voit objet comme un Rect
    ...
    Square * s2 = new Rect();     // erreur de compilation!
    Square * s3 = r;              // erreur de compilation!
}
```



Buts du polymorphisme

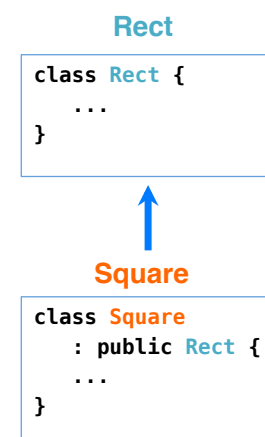
- pouvoir choisir le **point de vue le plus approprié** selon les besoins
- pouvoir traiter un ensemble de classes liées entre elles de **manière uniforme sans considérer leurs détails**

```
#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
    ...
    Square * s2 = new Rect();     // erreur de compilation!
    Square * s3 = r;              // erreur de compilation!
}
```

upcasting implicite

downcasting interdit

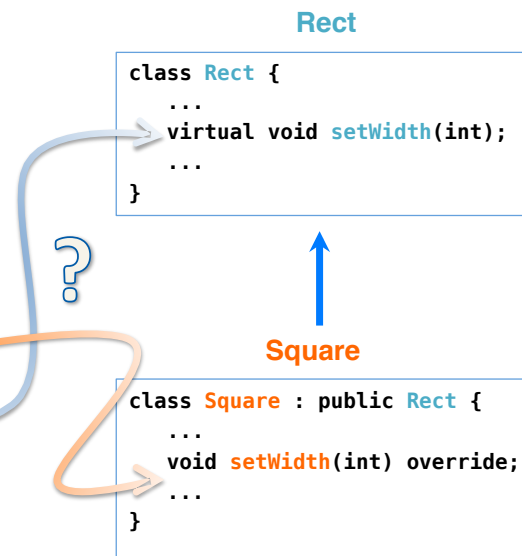
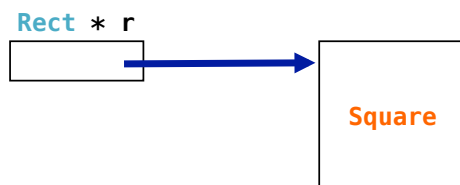


Polymorphisme

Question à \$1000

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

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

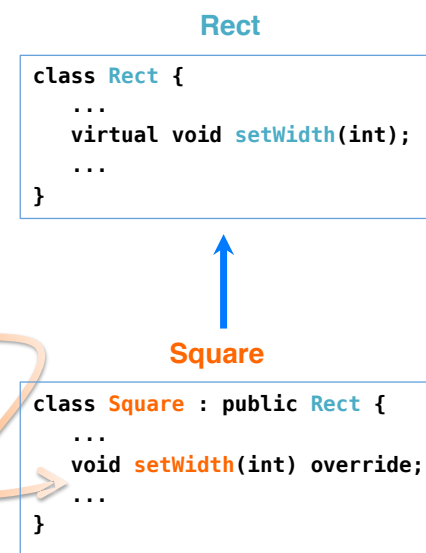


Polymorphisme : Java

Question à \$1000

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

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



Java

- **liaison dynamique / tardive** : choix de la méthode à l'exécution

⇒ appelle toujours la méthode du **pointé**

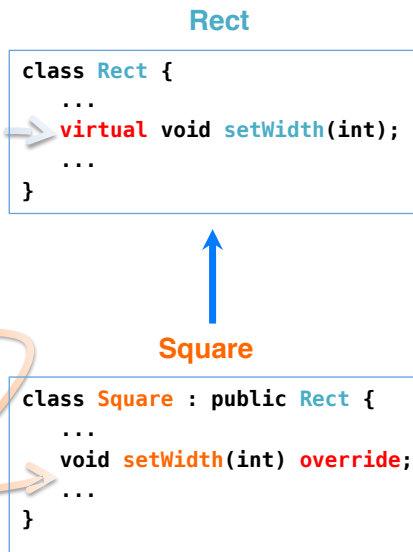
- **heureusement sinon le carré deviendrait un rectangle !**

Polymorphisme : C++

Question à \$1000

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

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



C++ et C#

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

Règles à suivre

Méthode parente

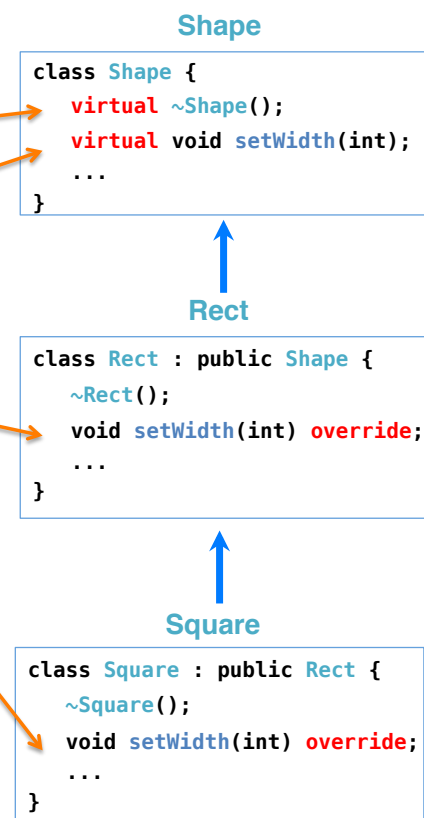
- ⇒ mettre **virtual**
- ⇒ y compris pour les **destructeurs**

Méthode redéfinie

- ⇒ mettre **override** (ou **final**)
(en C++11)

Dans le doute on peut mettre **virtual** partout !

```
Rect * r = new Square();  
  
// appelle ~Square() et ~Rect() car ~Shape() virtual  
delete r;
```



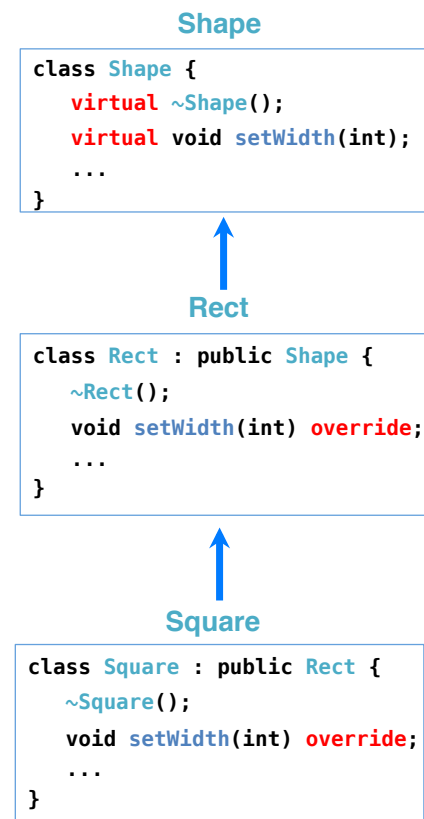
Règles à suivre

Détails

- une **redéfinition** de méthode **virtuelle** est automatiquement **virtuelle**
- **override** vérifie que la méthode **parente** est **virtuelle**
=> sert à détecter les erreurs !
- de plus **final** **interdit** toute redéfinition

Méthodes non virtuelles

- principal intérêt : (un peu) plus rapides
- impact négligeable dans 99% des cas !
- typiquement :
 - *getters et setters*
 - *fonctions appelées très très très souvent*



Méthodes et classes abstraites

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

Méthode abstraite

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

Classe abstraite

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

Java

- pareil mais mot clé **abstract**

Bénéfices des classes abstraites

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

Méthode abstraite

– spécification d'un **concept** dont la réalisation diffère selon les sous-classes

- **pas implémentée**
- doit être **redéfinie et implémentée** dans les sous-classes **instanciables**

Traiter un ensemble de classes liées entre elles :

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

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

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

Exemple de classe abstraite

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

implémentation commune à toutes les sous-classes

méthodes abstraites: l'implémentation dépend des sous-classes

doivent être implémentées dans les sous-classes

Interfaces

Classes totalement abstraites

- **toutes** les méthodes sont **abstraites** : pure spécification
- **C++** : cas particulier de **classe abstraite**
- **Java** : mot clé **interface** (pas nécessaire en **C++** car héritage multiple)

```
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

Traitements uniformes

```
#include "Rect.h"  
#include "Circle.h"  
  
void foo() {  
    Shape ** shapes = new Shape * [10];  
  
    unsigned int count = 0;  
    shapes[count++] = new Circle(0, 0, 100);  
    shapes[count++] = new Rect(10, 10, 35, 40);  
    shapes[count++] = new Square(0, 0, 60);  
  
    printShapes(shapes, count);  
}
```

← tableau de 10 Shape *

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

Tableaux dynamiques

```
int * tab = new int[10];  
...  
delete [] tab;    // ne pas oublier []
```

tableau de 10 int

pointeur tab



chaque élément est un int

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

tableau de 10 Shape *

pointeur shapes



chaque élément est un Shape *

Traitements uniformes (2)

```
#include "Rect.h"  
#include "Circle.h"  
  
void foo() {  
    Shape ** shapes = new Shape * [10];  
  
    unsigned int count = 0;  
    shapes[count++] = new Circle(0, 0, 100);  
    shapes[count++] = new Rect(10, 10, 35, 40);  
    shapes[count++] = new Square(0, 0, 60);  
  
    printShapes(shapes, count);  
}
```

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

shapes



instance
de Circle

instance
de Rect

instance
de Square

chaque élément
est un Shape *

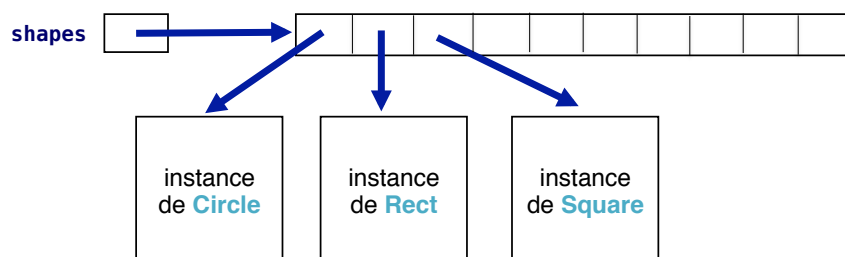
Magie du polymorphisme

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

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

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

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



Magie du polymorphisme

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

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

Remarque

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

Mission accomplie !

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

Chaînage des méthodes

Règle générale : éviter les duplications de code

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

Solutions

- utiliser l'**héritage** !
- le cas échéant : **rappeler les méthodes des superclasses**

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

Concepts fondamentaux de l'orienté objet

En résumé : 4 fondamentaux

- 1) **méthodes** (liaison automatique entre les fonctions et les données)
- 2) **encapsulation** (essentiel en OO mais possible avec des langages non OO)
- 3) **héritage** (simple ou multiple)
- 4) **polymorphisme de type** (toute la puissance de l'OO !)

Implémentation des méthodes virtuelles

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

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

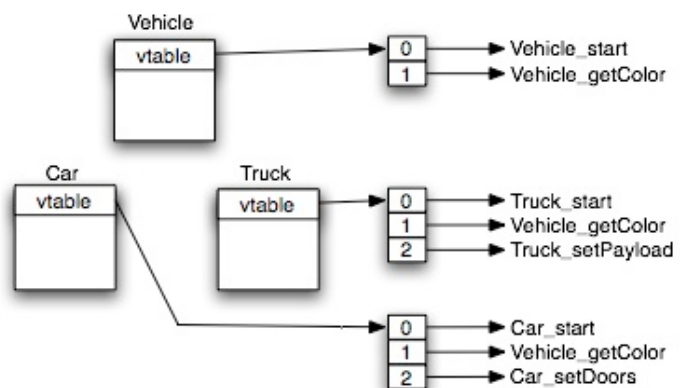
class Truck : public Vehicle {
public:
    virtual void start();
    virtual void setPayload(int payload);
    ...
};
```

Implémentation des méthodes virtuelles

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

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

class Truck : public Vehicle {
    __TruckTable * __vtable;
public:
    virtual void start();
    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[#start])();
```

Coût des méthodes virtuelles

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

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

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

Coût d'exécution

- double indirection
 - coût généralement négligeable
- contre exemple :
 - méthode appelée très très très souvent
 - ⇒ plus rapide si **non virtuelle**
 - ⇒ **gare aux erreurs** si on la redéfinit !

```
Vehicle * p = new Car();
p->start(); == (p->__vtable[#start])();
```

Coût des méthodes virtuelles

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

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

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

Coût mémoire

- un pointeur (**vtable**) par objet
- ⇒ méthodes virtuelles inutiles si :
 - aucune sous-classe
 - ou aucune redéfinition de méthode

Implémentation des méthodes virtuelles

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

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

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

```
0000000100001040 t __ZN3Car5printEv
0000000100000ff0 t __ZN3Car5startEv
0000000100000f40 t __ZN3CarC1Ei
0000000100000f70 t __ZN3CarC2Ei

0000000100001100 t __ZN7Vehicle5printEv
00000001000010b0 t __ZN7Vehicle5startEv
0000000100001c70 t __ZN7Vehicle8getColorEv
0000000100000fc0 t __ZN7VehicleC2Ei

0000000100002150 D __ZTI3Car
0000000100002140 D __ZTI7Vehicle
0000000100001ef4 S __ZTS3Car
0000000100001ef9 S __ZTS7Vehicle
0000000100002120 d __ZTV3Car
0000000100002168 d __ZTV7Vehicle
                    U __ZTVN10__cxxabiv117__class_type_infoE
                    U __ZTVN10__cxxabiv120__si_class_type_infoE

0000000100000ec0 T _main
```

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

Classes imbriquées (2)

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

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

Java

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

Pas en C++ !

Classes imbriquées (3)

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

`car` pointe sur l'objet contenant

`paint()` a accès à `car->color`

Solution générique

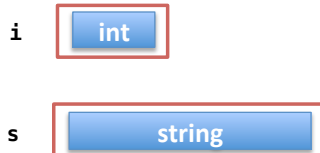
- pour « envoyer un message » à un objet il faut son **adresse**

Chapitre 3 : Gestion mémoire

Allocation mémoire

Mémoire automatique (pile/stack)

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



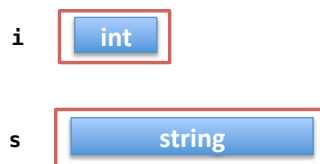
```
void foo(bool option) {  
    int i = 0;  
    i += 10;  
  
    string s = "Hello";  
    s += " World";  
    s.erase(4, 1);  
    ...  
}
```

- accède aux champs de l'objet

Allocation mémoire

Mémoire globale/statique

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



initialisés une seule fois

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

void foo() {
    static int i = 0;
    i += 10;

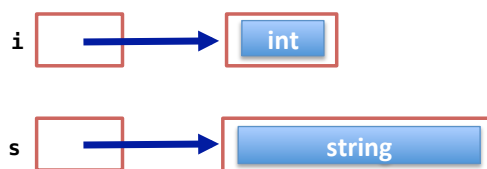
    static string s = "Hello";
    s += "World";
    s.erase(4, 1);
    ...
}
```

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

Allocation mémoire

Mémoire dynamique (tas/heap)

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



```
void foo() {
    int * i = new int(0);
    *i += 10;

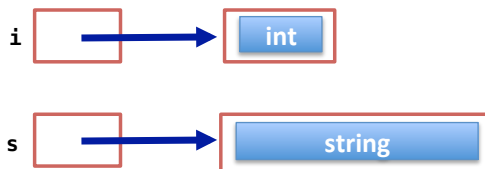
    string * s = new string("Hello");
    *s += " World";
    s->erase(1, 1);
    ...
    delete i;
    delete s;
}
```

-> accède aux champs de l'objet
 $a \rightarrow x == (*a).x$

Allocation mémoire

Mémoire dynamique (tas/heap)

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



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

Penser à détruire les pointés !

- sinon la donnée existe jusqu'à la terminaison du programme
- ! **delete** ne détruit pas la variable (ici dans la pile) mais ce qu'elle **pointe** !

Objets et types de base

C/C++

- C++ traite les **objets** (C les **struct**) de la même manière que les **types de base**
- sauf que les **constructeurs** / **destructeurs** des objets sont appelés

```
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;  
}
```

Objets et types de base

C/C++

- C++ traite les **objets** (C les **struct**) de la même manière que les **types de base**
- sauf que les **constructeurs** / **destructeurs** des objets sont appelés
- **ce n'est pas le cas en Java !**

Java

- **objets** toujours créés avec **new**
- **types de base** jamais créés avec **new**
- pas de variables **globales** ou **static**
 - sauf **variables de classes**

```
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;  
}
```

Coût de l'allocation mémoire

Gratuit ou négligeable

- mémoire **globale/statique**
 - fait à la compilation
- mémoire **automatique/pile**
 - attention : la taille de la pile est limitée !
- **objets dans les objets** (voir ci-après)

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

Coût de l'allocation mémoire

Gratuit ou négligeable

- mémoire **globale/statique**
 - fait à la compilation
- mémoire **automatique/pile**
 - attention : la taille de la pile est limitée !
- **objets dans les objets** (voir ci-après)

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

Coûteux : mémoire dynamique (tas)

- **new** (C/C++) et **ramasse-miettes** (Java)
- **impact important** sur les performances (en particulier en C/C++ où on peut choisir)
- difficilement **prédictible** (en particulier le ramasse miettes en Java)
 - ⇒ peut être problématique pour l'embarqué et le temps réel

Objets dans des objets

```
class Car : public Vehicle {  
    Door * leftDoor;  
    Door rightDoor;  
    ...  
public:  
    Car(string model, string color) :  
        leftDoor(new Door(this)),  
        rightDoor(this) {  
    }  
};
```

leftDoor 

rightDoor 

contient l'objet, pas possible en Java

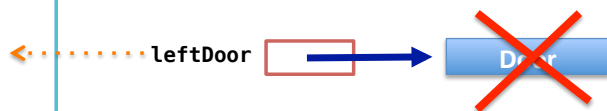
Variables d'instance **contenant** un objet

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

Qu'est-ce qui manque ?

Objets dans des objets

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



Il faut un destructeur !

- pour détruire l'objet **créé par le constructeur**

Par contre

- les objets **contenus dans les variables** sont **autodétruits**

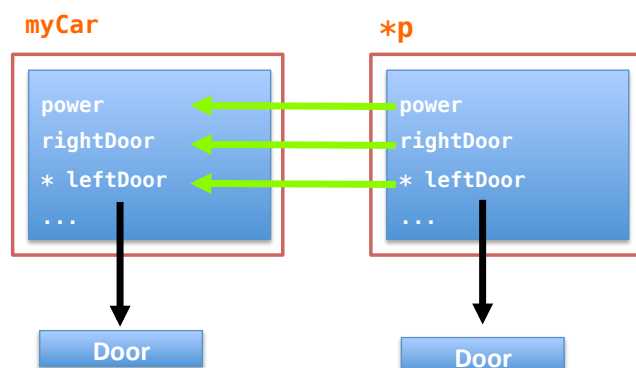
Copie d'objets

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

    myCar = *p; // affectation (après la création)
    Car mySecondCar(c); // initialisation (lors de la création)
}
```

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

Problème ?



Copie d'objets : cas problématique

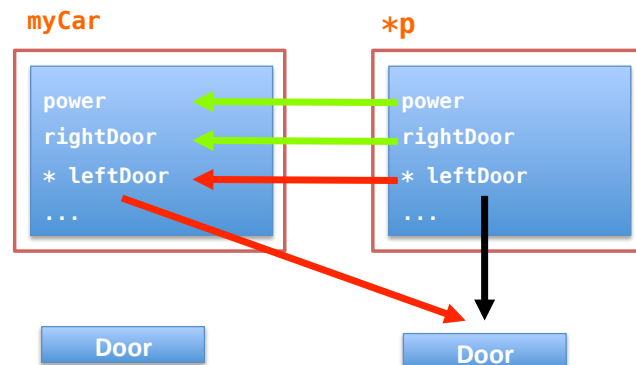
```
void foo() {  
    Car * p = new Car("Ferrari-599-GT0", "red");  
    Car c("Smart-Fortwo", "blue");  
    Car myCar;  
  
    myCar = *p;  ← ces 2 voitures ont la même porte !  
    Car mySecondCar(c);  ← même problème !  
}
```

Problème (général)

- les pointeurs pointent **sur la même chose !**
- pas de sens dans ce cas !

De plus (en C/C++)

- **plantage** lors de la destruction !



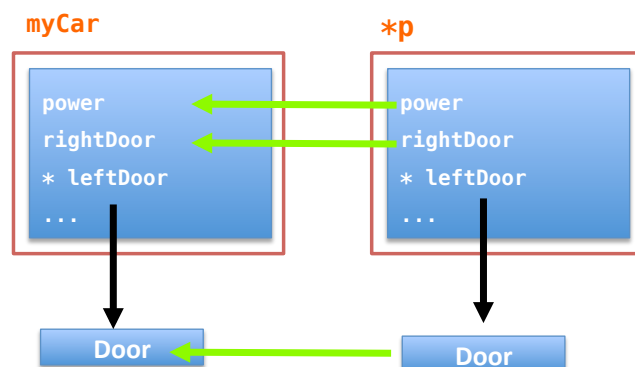
Copie superficielle et copie profonde

Copie superficielle

- copie **champ à champ** (le défaut)
- généralement problématique si l'objet contient des **pointeurs** ou des **références**
- problème pas spécifique à **C/C++**

Copie profonde

- copier les objets **pointés**
pas les pointeurs !
- et ce récursivement



Opérateurs de copie

Copy constructor et operator=

- respectivement appelés à l'**initialisation** et à l'**affectation**
- solution basique :
 - les **interdire** : les rendre **private** (pas besoin de les implémenter)
- solution avancée :
 - les **redéfinir** pour faire de la copie profonde (exemple page suivante)

```
class Car : public Vehicle {  
    ....  
private:  
    Car(const Car& from); <..... initialisation : Car myThirdCar(c);  
    Car& operator=(const Car& from); <..... affectation : myCar = c;  
};
```

Redéfinir la copie d'objets

```
Car::Car(const Car& from) : Vehicle(from) {  
    rightDoor = from.rightDoor;  
    ....  
    if (from.leftDoor) leftDoor = new Door(*from.leftDoor); // crée une copie de leftDoor  
    else leftDoor = nullptr;  
}  
  
Car& Car::operator=(const Car& from) {  
    Vehicle::operator=(from); // ne pas oublier de copier les champs de Vehicle !  
    rightDoor = from.rightDoor;  
    ....  
    if (leftDoor && from.leftDoor) *leftDoor = *from.leftDoor; // copie leftDoor  
    else {  
        delete leftDoor;  
        if (from.leftDoor) leftDoor = new Door(*from.leftDoor);  
        else leftDoor = nullptr;  
    }  
    return *this;  
}
```

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

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;  
}
```

certains compilos
requièrent une
constante

initialise à 0

C++11 seulement

ne pas oublier []

tab 

p 

Compléments

```
bool is_valid = true;  
static const char * errmsg = "Valeur invalide";  
  
void foo() {  
    is_valid = false;  
    cerr << errmsg << endl;  
}
```

variable globale

variable statique de fichier

En C/C++ il y a aussi :

Les variables **globales**

- accessibles dans **toutes les fonctions de tous les fichiers**

=> dangereuses !

Les variables **statiques de fichier**

- accessibles dans les fonctions **de ce fichier**

En C/C++, Java, etc. il y a aussi :

La mémoire **constante** (parfois appelée statique)

- exple : littéraux comme **"Hello Word"**

Les variables **volatiles**

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

Chapitre 4 : Types, constantes et smart pointers

Types de base

- `bool`
- `char16_t`, `char32_T`, `wchar_t`
- `char`
- `short`
- `int`
- `long`
- `long long`
- `float`
- `double`
- `long double`

← peuvent être
signed ou unsigned

Attention la taille dépend de la plateforme !

⇒ éventuels problèmes de **portabilité**

- tailles définies dans `limits.h` et `float.h` (dans `/usr/include` sous **Unix**)

– `char` peut être **signé** ou **non signé** selon les OS

- valeur entre `[0, 255]` ou bien `[-128, 127]` !

Typedef et inférence de types

typedef crée un nouveau nom de type

```
typedef Shape* ShapePtr;  
typedef list<Shape*> ShapeList;  
typedef bool (*compareShapes)(const Shape* s1, const Shape* s2);
```

Typedef et inférence de types

typedef crée un nouveau nom de type

```
typedef Shape* ShapePtr;  
typedef list<Shape*> ShapeList;  
typedef bool (*compareShapes)(const Shape* s1, const Shape* s2);
```

Inférence de types (C++11)

<pre>auto count = 10; auto PI = 3.1416;</pre>	<pre>int count = 10; double PI = 3.1416</pre>
==	
<pre>ShapeList shapes; auto it = shapes.begin();</pre>	<pre>list<Shape*> shapes; list<Shape*>::iterator it = shapes.begin();</pre>

decltype (C++11)

<pre>struct Point {double x, y;}; Point * p = new Point(); decltype(p->x) val = p->x;</pre>	<pre>définit le type à partir de celui d'une autre variable</pre>
---	---

Constantes

Macros du C (obsolète)

- substitution textuelle **avant** la compilation

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

Enumérations

- pour définir des **valeurs intégrales**
- commencent à 0 par défaut
- existent aussi en **Java**

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

Variables **const**

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

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

constexpr (C++11)

- expression calculée à la compilation

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

Pointeurs et pointés

Qu'est-ce qui est constant ?

- le **pointeur** ou le **pointé** ?

const porte sur « ce qui suit »

```
// *s est constant:  
const char * s  
char const * s
```

pointeur



pointé

```
// s est constant:  
char * const s
```

pointeur



pointé

```
// les deux sont constants:  
const char * const s
```

pointeur



pointé

Paramètres et méthodes const

```
char* strcat(char * s1, const char * s2) {  
    ....  
}
```

```
class Square {  
public:  
    int getX() const;  
    void setX(int x);  
    ....  
};
```

Paramètre const

- la fonction **ne peut pas modifier ce paramètre**

Méthode const

- la fonction **ne peut pas modifier l'objet** (i.e. ses variables d'instance)

Dans les deux cas

⇒ *spécifie ce que la fonction a le droit de faire* ⇒ **évite les erreurs !**

Objets immuables

Objets immuables

- objets que l'on ne peut pas modifier
- peuvent être **partagés** sans risque, ce qui évite d'avoir à les **dupliquer**

Deux techniques

- l'objet n'a pas de méthode **permettant de le modifier**
 - exemple : **String** en **Java**
- variable **const**
 - seules les **méthodes const** peuvent être appelées

```
const Square * s = new Square(10, 10, 50);  
  
s->setX(50);                // erreur: setX() pas const  
cout << s->getX() << endl; // OK: getX() est const
```

```
class Square {  
public:  
    int getX() const;  
    void setX(int x);  
    ....  
};
```

Constance logique

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

Objet vu comme immuable

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

Mais qui peut modifier son état interne

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

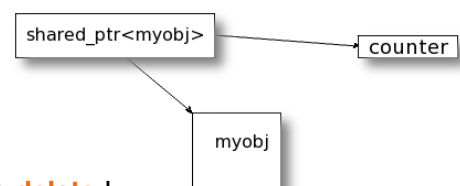
Smart pointers

```
#include <memory>

void foo() {
    shared_ptr<Circle> p(new Circle(0, 0, 50));    // count=1
    shared_ptr<Circle> p2;
    p2 = p;    // p2 pointe aussi sur l'objet => count=2
    p.reset();    // p ne pointe plus sur rien => count=1
}    // p2 est détruit => count=0 => destruction automatique de l'objet
```

shared_ptr

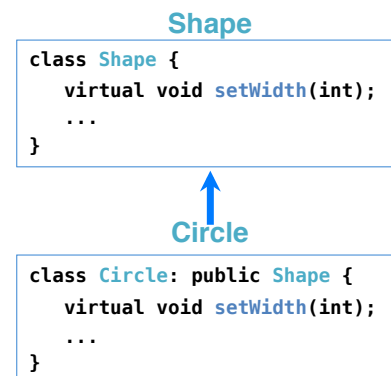
- **smart pointer** avec comptage de références
 - objet détruit quand le compteur arrive à 0
 - => mémoire gérée **automatiquement** : plus de **delete** !
- par défaut en C++11, sinon en option via **Boost**



Smart pointers

```
#include <memory>

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



S'utilisent comme des "raw pointers"

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

Attention !

- ne marchent pas si **dépendances circulaires** entre les objets pointés !
- il est **dangereux** de les convertir en raw pointers (car on perd le compteur !)

Compléments

```
#include <memory>

void foo() {
    unique_ptr<Shape> tab[10];
    tab[10] = unique_ptr<Shape>(new Circle(0, 0, 50));

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

unique_ptr : smart pointer sans comptage de références

- lorsque l'objet est pointé par **un seul** smart pointer
- pas de compteur => pas de coût mémoire
- particulièrement utiles pour les **tableaux** ou **conteneurs** pointant des objets

weak_ptr :

- pointe un objet déjà pointé par un **shared_ptr** sans le "posséder"
- sait **si l'objet existe** encore ou a été détruit (contrairement à un raw pointer)
- permet d'éviter les **dépendances circulaires**

Chapitre 5 : Bases des Templates et STL

Templates et programmation générique

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

i = max(4, 10);           <..... T déduit: int
x = max(66., 77.);        <..... T déduit: double
y = max<float>(66., 77.); <..... T spécifié: float

string s1 = "aaa", s2 = "bbb"; <..... T déduit: string
s = max(s1, s2);
```

Templates = paramétrage de type

- les **types** sont des paramètres
- permet de définir des **algorithmes** ou des **types génériques**

Exemple

- **max()** est **instanciée** à la compilation comme si on avait défini **4 fonctions différentes**
- **Note** : **max()** est définie en standard sous une forme plus optimale

Classes templates

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

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

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

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

appelle: operator+(a,b)

T peut être ce qu'on veut

- pourvu qu'il soit compatible avec les méthodes de `Matrix`

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)

NB: on verra une solution plus performante au chapitre suivant

Metaprogrammation

```
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
}
```

calculer factorielle
sans appel de fonction !

spécialisation
de template

appel récursif

Programme qui manipule un programme

- ici, la valeur est calculée à la **compilation** !
- appel **récursif**
- **spécialisation** pour appel terminal

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++

- instantiation faite à la **compilation** => **optimisation** en fonction des types réels
- puissants (Turing complets !) mais vite complexes !
- base de la **STL**

Generics Java

- sémantique et implémentation différentes :
 - pas de types de base,
 - pas instanciés à la compilation,
 - pas de spécialisation,
 - le type est « effacé » ...

Standard Template Library (STL)

```
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 **regrouper** et **manipuler** une collection d'objets
- compatibles avec les **objets** et les **types de base**
- gèrent **automatiquement** la mémoire nécessaire à leurs éléments
 - exemples : **vector**, **list**, **map**, **set**, **deque**, **queue**, **stack** ...

Itérateurs

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

Algorithmes

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

Vecteur

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

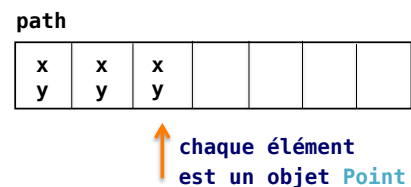
struct Point {
    int x, y;
    Point(int x, int y);
    void print() const;
};

void foo() {
    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();
    path.clear();
}
```

struct == class + public:

clear() vide le vecteur



Vecteur

- **accès direct** aux éléments par **[i]** ou **at(i)** – **Note** : **at()** vérifie l'indice, mais pas **[]**
- **coût élevé** d'insertion / suppression

Liste et itérateurs

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

void foo() {
    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();
}
```

liste de pointeurs
d'où les new

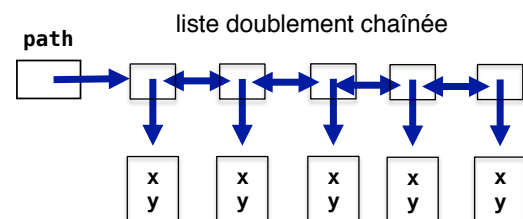
auto et for (:)
C++11 seulement

Liste

- pas d'accès direct aux éléments
- faible coût d'insertion / suppression

Note

- cette liste pointe sur les objets
- elle pourrait aussi les contenir



Liste et itérateurs : ancienne syntaxe

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

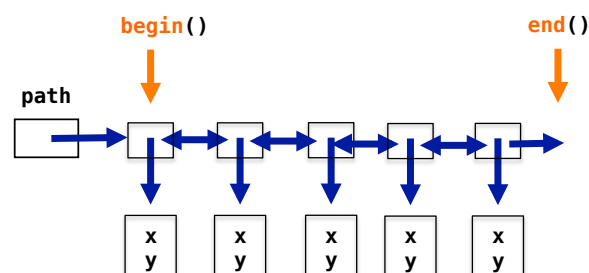
void foo() {
    list<Point*> path;
    path.push_back(new Point(20, 20));
    path.push_back(new Point(50, 50));
    path.push_back(new Point(70, 70));

    for (auto it : path) it->print();

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

C++11 seulement

parenthèses nécessaires



Conteneurs pointant des objets

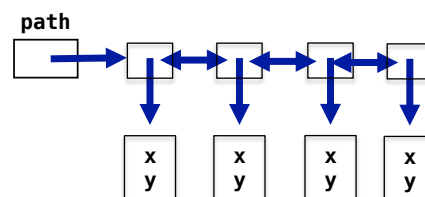
```
#include <list>
using namespace std;

void foo() {
    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();
}
```

Cette liste pointe sur les objets

⇒ *problème !*



Conteneurs pointant des objets

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

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

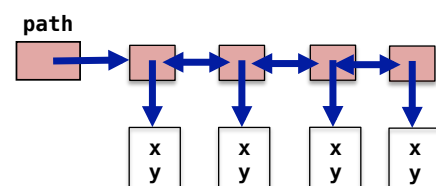
    for (auto it : path) it->print();
    ....
    for (auto it : path) delete it;
}
```

Détruire les objets pointés !

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

Alternatives

- utiliser des **smart pointers**
- contenir les objets (quand c'est possible)



Enlever des éléments dans une liste

Enlever à une position donnée

- iterator `erase(iterator position);`
- iterator `erase(iterator first, iterator last);`

Enlever un élément donné

- void `remove(const T& value);`
- void `remove_if(Predicate)`

Attention !

- ces fonctions invalident l'itérateur courant !

Détruire plusieurs éléments dans une liste

```
typedef std::list<Point> PointList;    // typedef simplifie l'écriture

PointList path;
int val = 200;                        // détruire les points dont x vaut 200

for (PointList::iterator k = path.begin(); k != path.end(); ) {
    if ((*k)->x != val)
        k++;
    else {
        PointList::iterator k2 = k;
        k2++;
        delete *k;                // détruit l'objet pointé par l'itérateur
        path.erase(k);            // k est invalide après erase()
        k = k2;
    }
}
```

Attention

- l'itérateur `k` est invalide après `erase()`
d'où l'utilité d'un second itérateur `k2`

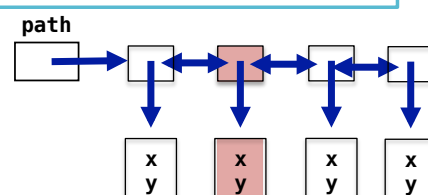


Table associative (map)

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

```
typedef map<string, User*> Dict;    // typedef simplifie l'écriture
```

```
void foo() {
    Dict dict;
    dict["Dupont"] = new User("Dupont", 666);    // ajout
    dict["Einstein"] = new User("Einstein", 314);

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

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

Remarque

- on pourrait utiliser **set** au lieu de **map**

Trier les éléments d'un conteneur

```
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
```

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

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

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


Chapitre 6 :

Passage par valeur et par référence

Passage par valeur

```
class Point {  
    int x, y;  
public:  
    Point(int x, int y);  
    void set(int x, int y);  
    ...  
};  
  
void foo() {  
    Point p(10, 10);  
    int a = 20, b = 30;  
    p.set(a, b);  
}
```

paramètres formels

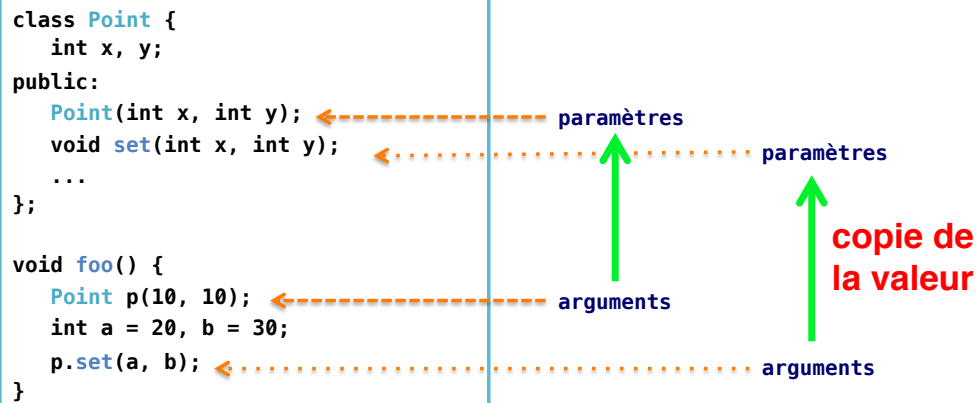
paramètres formels

arguments

arguments

Quelle est la relation entre les paramètres et les arguments ?

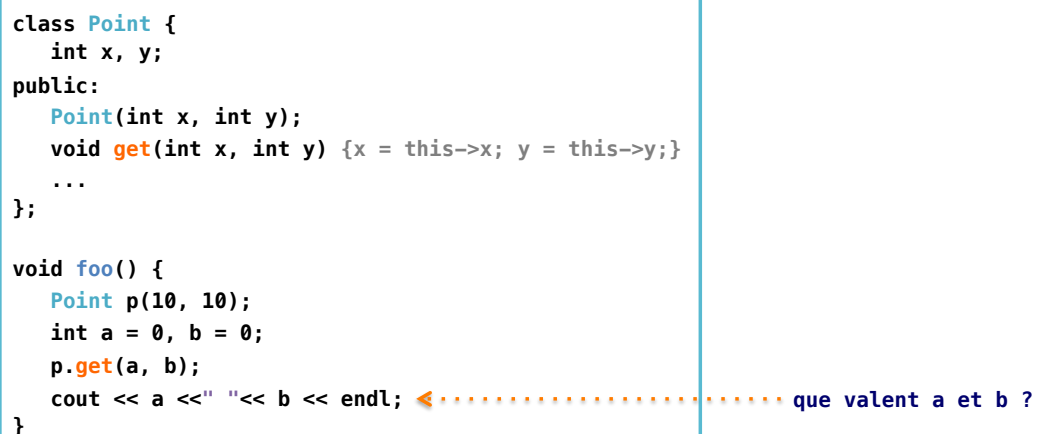
Passage par valeur



La valeur de l'argument est copiée dans le paramètre

- exemple : 20 (valeur de a) est copié dans x
- même chose par défaut avec la plupart des langages
- seule possibilité en C et en Java

Comment récupérer des valeurs ?

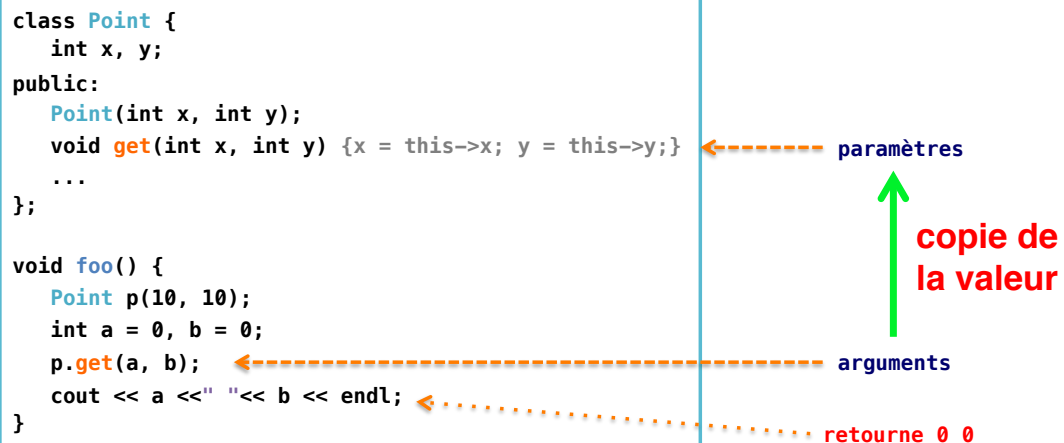


But

- récupérer à la fois x et y
- return ne retourne qu'un élément !

Résultat ?

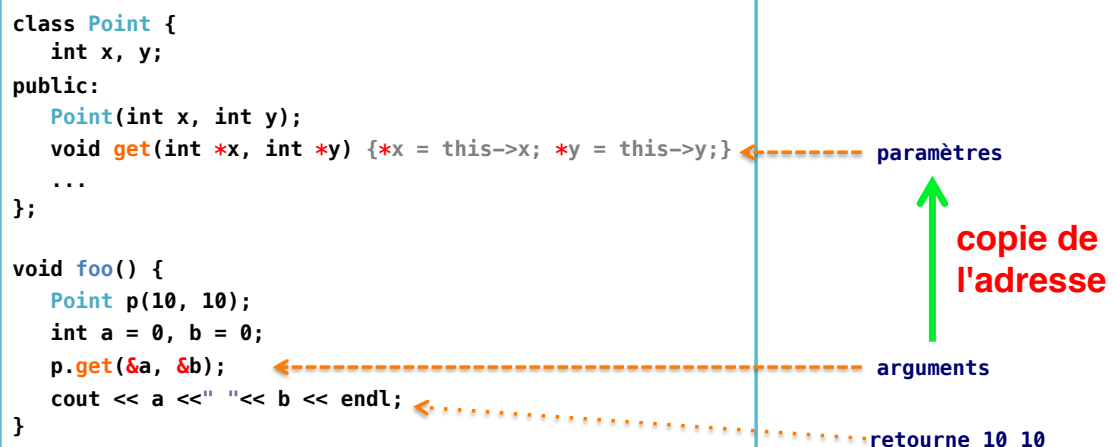
Comment récupérer des valeurs ?



Passage par valeur => arguments inchangés

- les arguments sont **copiés** dans les paramètres mais **pas l'inverse !**
 - = paramètres "in" (en entrée)
 - pareil en **C** et en **Java !!!** (que du passage par valeur)

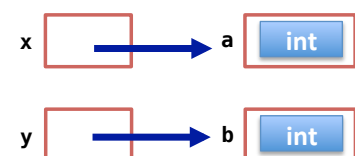
Passage par (valeur de l') adresse



Passage par valeur => arguments inchangés
sauf si c'est l'adresse qu'on passe par valeur

= Passage par adresse (ou par pointeur)

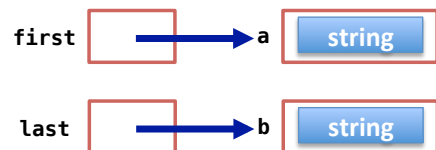
=> get() peut alors modifier le contenu des **pointés** a et b



Passage par adresse des objets

```
class User {  
    string firstname, lastname;  
public:  
    User(string first, string last);  
    void get(string *first, string *last) {  
        first->assign(firstname);  
        last->assign(lastname);  
    }  
};  
  
void foo() {  
    User user("Jean", "Dupont");  
    string a, b;  
    user.get(&a, &b);  
    cout << a <<" " << b << endl;  
}
```

// ou si l'on préfère:
*first = firstname;
*last = lastname;

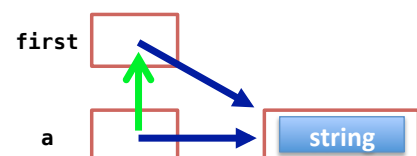


C'est pareil

- get() peut changer le contenu des **pointés** (a, b)
- par contre changer les **pointeurs** (first, last) n'aurait aucun effet !

Variante

```
class User {  
    string *firstname, *lastname;  
public:  
    User(string *first, string *last);  
    void get(string *first, string *last) {  
        first->assign(*firstname);  
        last->assign(*lastname);  
    }  
};  
  
void foo() {  
    User *user = new User("Jean", "Dupont");  
    string *a = new string(), *b = new string();  
    user->get(a, b);  
    cout << *a <<" " << *b << endl;  
}
```



Ce code C++ est équivalent à du Java

- sauf que ***** est implicite en **Java** pour les objets et qu'il faudrait utiliser **StringBuffer**
- rappel : get() doit changer le contenu des **pointés**

En Java

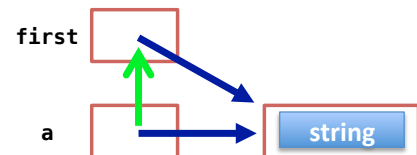
```
class User {  
    private String firstname, lastname;  
    public User(String first, String last);  
    public void get(StringBuffer first, StringBuffer last) {  
        first->replace(0, first->length(), firstname);  
        last->replace(0, last->length(), lastname);  
    }  
};  
  
void foo() {  
    User user = new User("Jean", "Dupont");  
    StringBuffer a = new StringBuffer();  
    StringBuffer b = new StringBuffer();  
    user.get(a, b);  
    System.out.println(a, b);  
}
```

Objets : toujours passés par adresse

- * implicite

Types de base : pas possible

- pas d'opérateurs * et & !



Comment récupérer des valeurs ?

Et si on faisait plus simple ?

- c'est possible grâce au **passage par référence** !
- paramètres "out" ou "in/out"
- existe dans divers langages (C#, Pascal, Ada..) avec des variantes

Passage par référence

```
class Point {
    int x, y;
public:
    Point(int x, int y);
    void get(int & x, int & y) {x = this->x; y = this->y;}
    ...
};

void foo() {
    Point p(10, 10);
    int a = 0, b = 0;
    p.get(a, b);
    cout << a << " " << b << endl;
}
```

paramètres

alias

arguments

résultat: 10 10

Les paramètres sont des **alias** des arguments

- pas de recopie (coût similaire au passage par adresse)
- paramètres "in/out"
- n'existe pas en **C** ni en **Java !!!** (passage par **valeur** des références Java)

Retour sur le passage par valeur

```
class Image {
public:
    void setPixel(int x, int y, Pixel p);
    Pixel getPixel(int x, int y) const;
    ....
};

Image add(Image ima1, Image ima2) {
    ....
}

void foo() {
    Image a(1000,1000), b(1000,1000);
    ....
    Image c = add(a, b);
}
```

```
struct Pixel {
    unsigned char r, g, b;
};
```

paramètres

copie

arguments

Problème ?

Retour sur le passage par valeur

```
class Image {
public:
    void setPixel(int x, int y, Pixel p);
    Pixel getPixel(int x, int y) const;
    ....
};

Image add(Image ima1, Image ima2) {
    ....
}

void foo() {
    Image a(1000,1000), b(1000,1000);
    ....
    Image c = add(a, b);
}
```

```
struct Pixel {
    unsigned char r, g, b;
};
```

paramètres

copie

arguments

Problème

- copier de (gros) objets est inefficace
- exemples : images, **vecteurs**, **listes**, **strings** ...
- solution ?

Passage par const référence

```
class Image {
public:
    void setPixel(int x, int y, const Pixel & p);
    Pixel getPixel(int x, int y) const;
    ....
};

Image add(const Image & ima1, const Image & ima2) {
    ....
}

void foo() {
    Image a(1000,1000), b(1000,1000);
    ....
    Image c = add(a, b);
}
```

```
struct Pixel {
    unsigned char r, g, b;
};
```

paramètres

alias

arguments

Arguments pas copié (alias) et pas modifiables (const)

- ⇒ passer les images, **vecteurs**, **listes**, **strings** ... par **const référence**
- ⇒ aucun intérêt pour les **types de base**

Retour des méthodes


```
class Image {  
    int width, height;  
    Pixel* pixels;  
public:  
    Image(int w, int h) : width(w), height(h), pixels(new Pixel[w * h]) {}  
    Pixel getPixel(int x, int y) const {return pixels[x + y * width];}  
    ....  
};  
  
void foo() {  
    Image a(1000,1000), b(1000,1000);  
    ....  
    Pixel pix = a.getPixel(10,10);  
}
```

```
struct Pixel {  
    unsigned char r, g, b;  
};
```

return fait du passage par valeur

⇒ renvoie une **copie** (sauf si le compilateur optimise)

Retour des méthodes



```
class Image {  
    int width, height;  
    Pixel* pixels;  
public:  
    Image(int w, int h) : width(w), height(h), pixels(new Pixel[w * h]) {}  
    const Pixel & getPixel(int x, int y) const {return pixels[x + y * width];}  
    ....  
};  
  
void foo() {  
    Image a(1000,1000), b(1000,1000);  
    ....  
    Pixel pix = a.getPixel(10,10);  
}
```

```
struct Pixel {  
    unsigned char r, g, b;  
};
```

return par const référence

⇒ évite la copie

Plus subtil !

```
class Image {  
    int width, height;  
    Pixel* pixels;  
public:  
    Image(int w, int h) : width(w), height(h), pixels(new Pixel[w * h]) {}  
    Pixel & getPixel(int x, int y) {return pixels[x + y * width];}  
    ....  
};  
  
void foo() {  
    Image a(1000,1000), b(1000,1000);  
    ....  
    Pixel pix = a.getPixel(10,10);  
    a.getPixel(20,20) = pix;  
}
```

```
struct Pixel {  
    unsigned char r, g, b;  
};
```

return par référence (sans const)

- pas de const => l'appelant peut modifier le pixel !
- dangereux ... sauf si c'est voulu !


Exemple

```
class Image {  
    int width, height;  
    Pixel* pixels;  
public:  
    Image(int w, int h) : width(w), height(h), pixels(new Pixel[w * h]) {}  
    const Pixel & operator()(int x, int y) const {return pixels[x + y * width];}  
    Pixel & operator()(int x, int y) {return pixels[x + y * width];}  
    ....  
};  
  
void foo() {  
    Image a(1000,1000), b(1000,1000);  
    ....  
    Pixel pix = a(10,10);  
    a(20,20) = pix;  
}
```

```
struct Pixel {  
    unsigned char r, g, b;  
};
```

Surcharge de operator()

Retour des fonctions



```
class Image {
    ....
};

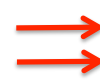
Image add(const Image & ima1, const Image & ima2) {
    Image res;
    for (auto y = 0; y < height; ++y)
        for (auto x = 0; x < width; ++x)
            res(x,y) = ima1(x,y) + ima2(x,y);
    return res;
}

void foo() {
    Image a(1000,1000), b(1000,1000);
    ....
    Image c = add(a, b);
}
```

Problème : return recopie *res*

⇒ pas efficace (sauf si le compilateur optimise)

Retour des fonctions



```
class Image {
    ....
};

const Image & add(const Image & ima1, const Image & ima2) {
    Image res;
    for (auto y = 0; y < height; ++y)
        for (auto x = 0; x < width; ++x)
            res(x,y) = ima1(x,y) + ima2(x,y);
    return res;
}


void foo() {
    Image a(1000,1000), b(1000,1000);
    ....
    Image c = add(a, b);
}
```

Problème : *res* est une variable locale

⇒ détruite quand on sort de la fonction ⇒ dangereux !

Alternative au retour des fonctions

```
class Image {  
    ....  
};  
  
void add(const Image & ima1, const Image & ima2, Image & res) {  
    for (auto y = 0; y < height; ++y)  
        for (auto x = 0; x < width; ++x)  
            res(x,y) = ima1(x,y) + ima2(x,y);  
}  
  
void foo() {  
    Image a(1000,1000), b(1000,1000), c(1000,1000);  
    ....  
    add(a, b, c);  
}
```



Solution optimale

- passer le resultat par référence

Préférer les références aux pointeurs

Parce que c'est plus simple

- en particulier pour le passage par référence

Parce que c'est plus sûr

- pas d'**arithmétique des références** (source d'erreurs)
- toujours **initialisées** (ne peuvent pas pointer sur 0)
- référencent toujours la **même** entité

```
Circle c1;  
Circle& r1 = c1;    // r1 sera toujours un alias de c1
```

Copie : références vs. pointeurs

Référence C++ = **alias** d'un objet (y compris pour la copie)

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

Référence Java ou pointeur C++ = **pointe** un objet

```
Circle* p1 = &c1;  
Circle* p2 = &c2;  
  
p1 = p2;          // copie le pointeur, pas l'objet pointé (comme en Java)
```

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;   
}
```

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

OK: upcasting

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

Rappel

- transtypage **implicite** vers les **super-classes** (**upcasting**)
- mais **pas** vers les **sous-classes** (**downcasting**)
- propriété de l'**héritage**

Transtypage vers les sous-classes

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

Correct ?


Transtypage vers les sous-classes

```
class Object {  
    // pas de methode draw()  
    ...  
};  
  
class Button : public Object {  
    virtual void draw();  
    ...  
};  
  
void foo(Object * obj) {  
    obj->draw();           <..... erreur de compilation :  
                                draw() n'est pas  
                                une méthode de Object !  
}  
  
void bar() {  
    foo(new Button());  
}
```

Que faire ?

- si on ne peut pas modifier **Object** ni la signature de **foo()**
 - cas typique : ils sont imposés par une bibliothèque


Transtypage vers les sous-classes

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

Mauvaise solution

- trompe le compilateur => **plante** si jamais **obj** n'est pas un **Object** !

Transtypage dynamique

```
class Object {  
    // pas de methode draw()  
    ...  
};  
  
class Button : public Object {  
    virtual void draw();  
    ...  
};  
  
void foo(Object * obj) {  
    Button * b = dynamic_cast<Button*>(obj);  renvoie null si ce  
    if (b) b->draw(); n'est pas un Button  
}  
  
void bar() {  
    foo(new Button());  
}
```

Bonne solution

- **contrôle dynamique** du type à l'exécution
- en **Java** : tester avec **instanceof** puis faire un cast

Opérateurs de transtypage

A l'exécution

- **dynamic_cast** : pour le downcasting des classes polymorphes
 - opérateur sûr

A la compilation

- tous ces opérateurs sont dangereux !
- à utiliser avec prudence :
 - **static_cast** : conversions de types "raisonnables"
- à éviter :
 - **const_cast** : pour enlever ou rajouter **const** au type
 - **reinterpret_cast** : conversions de types "radicales"
- à éviter absolument :
 - **cast du C** : ne vérifie rien du tout !

RTTI

```
#include <typeinfo>

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

Retourne de l'information sur le type

- **name()** retourne le nom du type
 - malheureusement il est généralement **encodé** donc peu utilisable !
- **opérateur==** compare si deux types sont égaux

Remarques

- pour tester si un type dérive d'un autre : **dynamic_cast**
- fonctionnalités limitées, c'est rarement utile

Types incomplets et handle classes

```
#include <Widget>

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

header Button.h

Handle class : pour cacher l'implémentation

- l'implémentation est **cachée** dans la classe `ButtonImpl`
 - déclarée dans un **header privé** `ButtonImpl.h` pas donné au client

Références à des objets auxiliaires

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

Types incomplets

```
#include <Widget>

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

header Button.h

Problème

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

Solution ?

Types incomplets

```
#include <Widget>
#include <ButtonImpl>
#include <MouseEvent>
...

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

header Button.h

Mauvaise solution

- Cacher l'implémentation :
 - c'est raté : il faut maintenant donner **ButtonImpl.h** au client !
- Références externes :
 - il faut inclure plein de headers (qui peuvent se référencer les uns les autres) !

Types incomplets

```
#include <Widget>
class ButtonImpl;
class MouseEvent;
...

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

header Button.h

Bonne solution

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

Pointeurs de fonctions

```
class Button : public Widget {
public:
    Button(const string& name);
    void addClickCallback( void(*func)(MouseEvent& event) );
    ....
};

void doIt(MouseEvent& event) {
    ....
}

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

func est un pointeur de fonction

- existe également en C mais pas en Java

Pointeurs de méthodes

```
class Button : public Widget {
public:
    Button(const string& name);
    void addClickCallback(Object* obj, void(Object::*func)(MouseEvent& event) );
    ....
};

class Object {
    void doIt(MouseEvent& event) {
        ....
    }
}

void foo() {
    Button * btn = new Button("OK");
    Object * obj = new Object();
    btn->addClickCallback(obj, &Object::doIt);
}
```

func est un pointeur de méthode

- particulièrement utile avec des **templates**
- alternative : passer des objets (cf. les **Listeners** de Java) ou des "foncteurs"

Fonctions et "foncteurs"

```
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
```

```
void doIt(const User& u) {
    cout << u.getName() < endl;
}
```

```
class Object {
    void operator()(const User& u) {cout << u.getName() < endl;}
}
```

```
void foo() {
    vector<User> entries;
    ....
    for_each(entries.begin(), entries.end(), doIt);

    Object obj();
    for_each(entries.begin(), entries.end(), obj);
    ....
}
```

```
class User {
    string name;
public:
    User(const string & name) : name(name) {}
    const string& getName() const {return name;}
};
```

foncteur

Surcharge des opérateurs

```
#include <string>

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

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

Possible pour presque tous les opérateurs

= == < > + - * / ++ -- += -= -> () [] new delete mais pas :: . .* ?

la priorité est inchangée

A utiliser avec discernement

– peut rendre le code incompréhensible !

Existe dans divers langages (C#, Python, Ada...)

– mais pas en Java

Surcharge des opérateurs

operator[]

operator()

- entre autres pour la **STL** : le même algorithme peut s'appliquer à des fonctions ou à des objets

```
#include <vector>

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

operator-> et operator*

- redéfinissent le déréférencement

operator++

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

Conversion de types

```
class String {
    operator char*() const {return c_s;}
    ....
};
```

Opérateurs new , delete , new[], delete[]

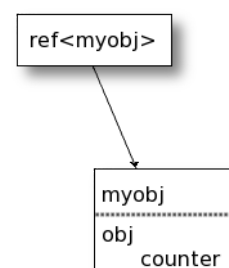
- redéfinissent l'allocation mémoire

Exemple : smart pointers intrusifs

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

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

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



Principe

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

Exemple : smart pointers intrusifs

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

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

Le smart pointer

- encapsule un pointeur standard (raw pointer)
- surcharge le copy constructor, l'opérateur = et les opérateurs -> et *

Exceptions

```
class MathErr {};

class Overflow : public MathErr {};

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

void foo() {
    try {
        int z = calcul(4, 0)
    }
    catch(Zerodivide & e) { cerr << e.x << "divisé par 0" << endl; }
    catch(MathErr)        { cerr << "erreur de calcul" << endl; }
    catch(...)            { cerr << "autre erreur" << endl; }
}

int calcul(int x, int y) {
    return divise(x, y);
}

int divise(int x, int y) {
    if (y == 0) throw Zerodivide(x);    // throw leve l'exception
    else return x / y;
}
```

Exceptions

But : faciliter le traitement des erreurs

- permettent de "remonter dans la pile" des appels des fonctions
- jusqu'à un point de contrôle

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

Avantage

- gestion plus centralisée, plus systématique des erreurs
 - que des enchaînements de fonctions retournant des codes d'erreurs

Inconvénient

- peuvent rendre le flux d'exécution difficile à comprendre si on en abuse !

Exceptions

Différences entre C++ et Java

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

Spécification d'exceptions (Java)

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

- existent mais obsolètes en **C++**, abandonnées en **C#**
- compliquent le code et posent certains problèmes
 - *attention en Java une méthode redéfinie dans une sous-classe ne peut pas spécifier de nouvelles exceptions*
- d'une manière générale les exceptions sont moins "à la mode"

Exceptions

Exceptions standards

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

Handlers

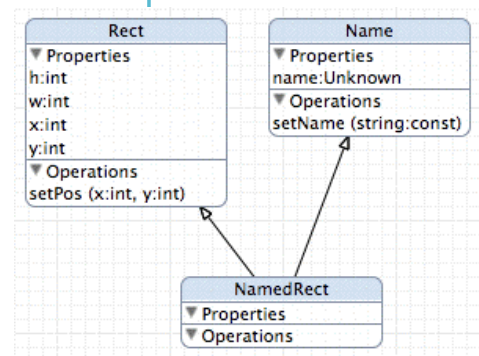
- `set_terminate()` et `set_unexpected()` spécifient ce qui se passe en dernier recours

Redéclenchement d'exceptions

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

Héritage multiple

```
class Rect {  
    int x, y, w, h;  
public:  
    virtual void setPos(int x, int y);  
    ....  
};  
  
class Name {  
    string name;  
public:  
    virtual void setName(const string&);  
    ....  
};  
  
class NamedRect : public Rect, public Name {  
public:  
    NamedRect(const string& s, int x, int y, int w, int h)  
        : Rect(x,y,w,h), Name(s) {}  
};
```



Principe

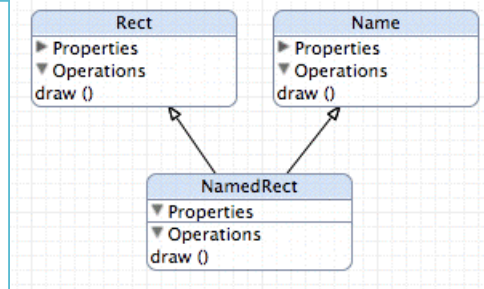
- la classe hérite de **toutes** les variables et méthodes de **ses** superclasses

Collisions de noms

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

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

class NamedRect : public Rect, public Name {
public:
    ....
};
```



Variables ou méthodes ayant le **même nom**
dans les superclasses
=> il faut les **préfixer** pour pouvoir y accéder

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

Collisions de noms

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

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

class NamedRect : public Rect, public Name {
public:
    virtual void draw() {
        Rect::draw();
        Name::draw();
    }
    ....
};
```

Solution typique

- redéfinir les méthodes concernées

Collisions de noms (using)

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

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

class NamedRect : public Rect, public Name {
public:
    using Rect::draw();
    ....
};
```

using

- spécifie quelle fonction on veut utiliser
- utile en particulier s'il y a de la surcharge

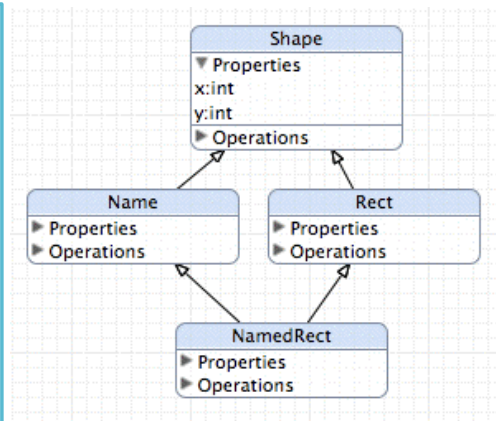
Héritage en diamant

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

class Rect : public Shape {
    ....
};

class Name : public Shape {
    ....
};

class NamedRect : public Rect, public Name {
public:
    ....
};
```

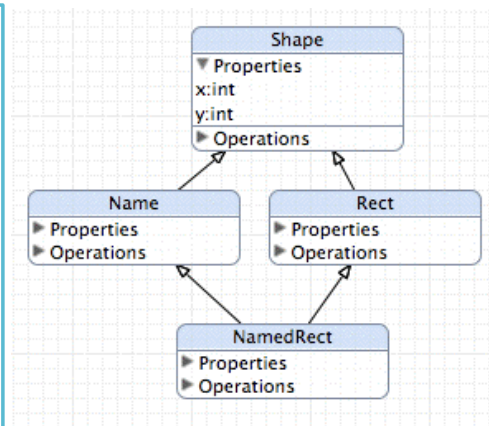


Problème

- la classe partagée (**Shape**) **est dupliquée** car elle est héritée "des deux côtés"
- en général ça n'a pas de sens !

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 : pas de duplication grâce à **virtual**

- un peu plus coûteux en mémoire et en temps que l'héritage simple

Attention : ne pas faire de casts, seulement du **dynamic_cast**

Sérialisation

But

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

Cas d'usage

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

Implémentation

- **pas d'implémentation standard** pour les objets en C++ (ou les **struct** en C)
 - contrairement à Java
- mais des extensions :
 - **Cereal, Boost, Qt, Protocol Buffers** de Google, **OSC**, etc.

Sérialisation

Sérialisation binaire

- objets stockés en binaire
- codage **compact** mais **illisible** par un humain
- a priori **pas compatible** d'un ordinateur à l'autre
 - taille et codage des nombres (little/big endian), alignement des champs
 - sauf si **protocole standardisé** (e.g. Protocol Buffers)

Sérialisation au format texte

- tout est converti en texte (y compris les nombres)
- prend **plus de place** mais **facile à lire** (et à vérifier !)
- un peu plus coûteux en CPU
- **compatible** entre ordinateurs
 - il existe des standards : XML/SOAP, JSON, etc.

Ecriture d'objets (format texte)

Principe : définir une fonction d'écriture (polymorphe) pour chaque classe

```
#include <iostream>

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

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

Fichier:

```
xxx\n
xxx\n
670\n
Ferrari-599-GTO\n
xxx\n
xxx\n
71\n
Smart-Fortwo\n

xxx : écrit par
Vehicule
```

Lecture d'objets

Même principe

```
#include <iostream>

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

class Car : public Vehicle {
public:
    virtual void read(std::istream & f) {
        Vehicule::read(f);
        f >> power >> model;
    }
    ....
};
```

Fichier:

```
xxx\n
xxx\n
670\n
Ferrari-599-GTO\n
xxx\n
xxx\n
71\n
Smart-Fortwo\n

xxx : écrit par
Vehicule
```

Lecture d'objets (problème)

```
class Car : public Vehicle {
public:
    virtual void read(std::istream & f) {
        Vehicule::read(f);
        f >> power >> model;
    }
    ....
};
```

Fichier:

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

présence d'espaces

Problème

- >> **model** s'arrête au premier « espace » (par défaut ' ', '\n', '\r', '\t', '\v', '\f')
- => lecture incorrecte si la **string** contient un de ces caractères !

Lecture d'objets (solution)

```
inline void skipnl(std::istream & f) {
    f.ignore(numeric_limits<streamsize>::max(), '\n');
}

class Car : public Vehicle {
public:
    virtual void read(std::istream & f) {
        Vehicule::read(f);
        f >> power; skipnl(f),
        getline(f, model);
    }
    ....
};
```

Fichier:

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

présence d'espaces

Solution

- **getline()** : lit toute la ligne (et supprime le '\n' final)
- **ignore()** : supprime le '\n' restant après la lecture de **power**
- alternative : tout lire avec **getline()** puis convertir au type voulu

Ecrire sur un fichier

```
using namespace std;
....

bool save(const string & fileName, const vector<Car *> & objects) {
    ostream f(fileName); // f(fileName.c_str()) avant C++11
    if (!f) {
        cerr << "Can't open file " << fileName << endl;
        return false;
    }
    // seulement avec C++11, sinon utiliser forme usuelle avec begin()/end()
    for (auto it : objects) it->write(f);
    return true;
}
```

Lire depuis un fichier

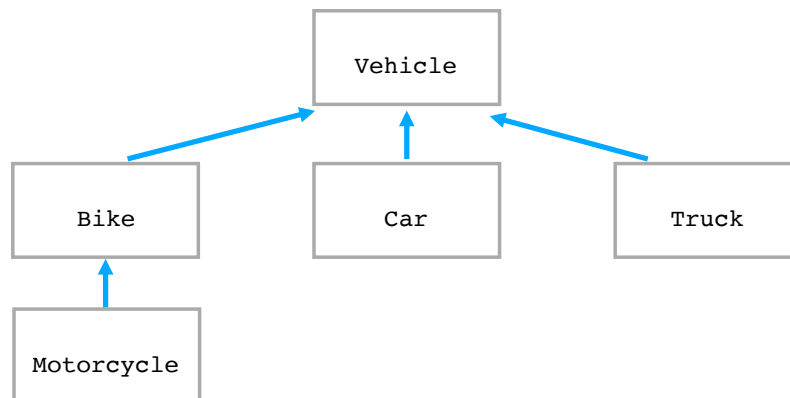
```
bool load(const string & fileName, vector<Car *> & objects) {
    istream f(fileName);
    if (!f) {
        cerr << "Can't open file " << fileName << endl;
        return false;
    }
    while (f.good()) { // pas d'erreur et pas en fin de fichier
        Car * car = new Car();
        car->read(f);
        if (f.fail()) { // erreur de lecture
            cerr << "Read error in " << fileName << endl;
            delete car;
            return false;
        }
        else objects.push_back(car);
    }
    return true;
}
```

Classes polymorphes

Problème

- les objets ne sont pas tous du même type (mais dérivent d'un même type)
 - e.g. Car, Truck, Bike ...

```
bool save(const string & fileName, const vector<Vehicle *> & objects);  
bool load(const string & fileName, vector<Vehicle *> & objects);
```



Classes polymorphes

Problème

- les objets ne sont pas tous du même type (mais dérivent d'un même type)
 - e.g. Car, Truck, Bike ...

```
bool save(const string & fileName, const vector<Vehicle *> & objects);  
bool load(const string & fileName, vector<Vehicle *> & objects);
```

Solution

- **en écriture :**
 - écrire le **nom de la classe** de l'objet, **puis** ses attributs
- **en lecture :**
 - lire le **nom de la classe**
 - **créer l'objet** correspondant
 - lire ses attributs

Classes polymorphes

```
class Vehicle {
public:
    virtual const char* classname() const {return "Vehicle";}
    ....
};

class Car : public Vehicle {
public:
    virtual const char* classname() const {return "Car";}
    ....
};

bool load(const string & fileName, vector<Vehicle *> & objects);
    ....
    while (f.good()) {
        string className;
        f >> className;
        Vehicle * obj = createVehicle(className);    // factory qui sert à
        if (obj) obj->read(f);                       // créer les objets
        ....
    }
    ....
}
```

stringstream

C'est un flux de caractères

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

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

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

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

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

Plus généralement

Améliorations

- meilleur traitement des erreurs
- gérer les pointeurs et les conteneurs
 - => utiliser **Boost** ou équivalent

JSON

- JavaScript Object Notation.
- format normalisé commode pour les échanges textuels

```
{
  "car": [
    { "firstName": "John" , "lastName": "Doe" },
    { "firstName": "Anna" , "lastName": "Smith" },
    { "firstName": "Peter" , "lastName": "Jones" }
  ]
}
```

Communication entre applications

Cas typique

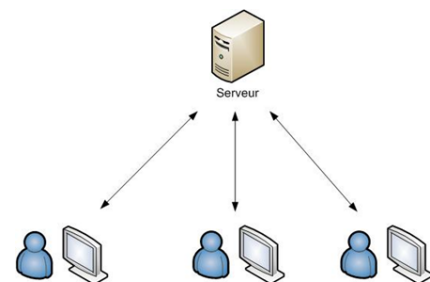
- un serveur de calcul
- des interfaces utilisateur pour interagir avec le serveur

Illustration en TP avec

- un serveur C++
- un client en ligne de commande
- un client implémentant une interface graphique Java Swing

Remarques

- on se limite au rappel des principes et technologies de base
- il existe des technologies plus avancées (middlewares)
 - => voir cours consacrés à ces sujets à l'Ecole



source:
maieutapedia.org

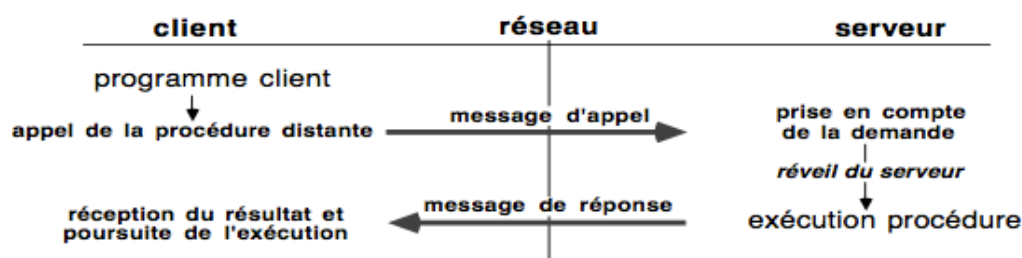
Client / serveur

Principe

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

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



Client / serveur

Principe

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

Dialogue asynchrone

- le client vaque à ses occupations après émission de la requête
- quand la réponse arrive une **fonction de callback** est activée
 - solution classique : le client lance un **thread** qui attend la réponse
 - exemple : **XMLHttpRequest** de JavaScript

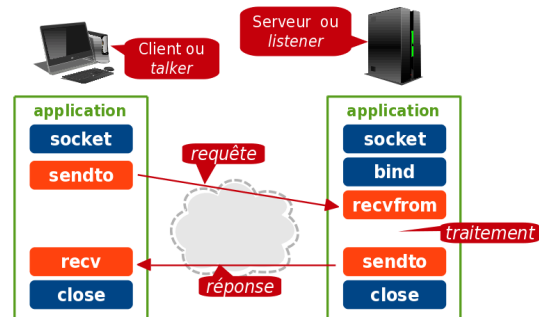
Sockets

Principe

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

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



source: inetdoc.net

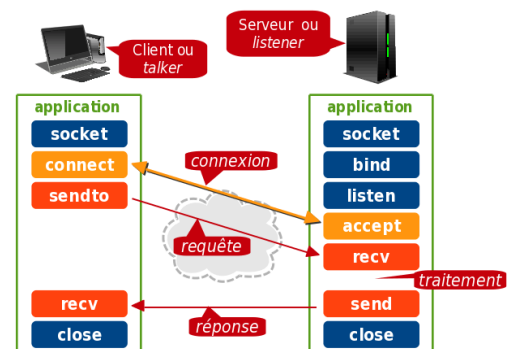
Sockets

Principe

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

Protocole TCP

- **Stream sockets** (type SOCK_STREAM)
- protocole **connecté**
- (un peu) plus coûteux en ressources
- flux d'octets entre 2 programmes, pas de paquets perdus et toujours dans l'ordre
 - exemple : HTTP
 - c'est ce qu'on utilisera en TP



source: inetdoc.net

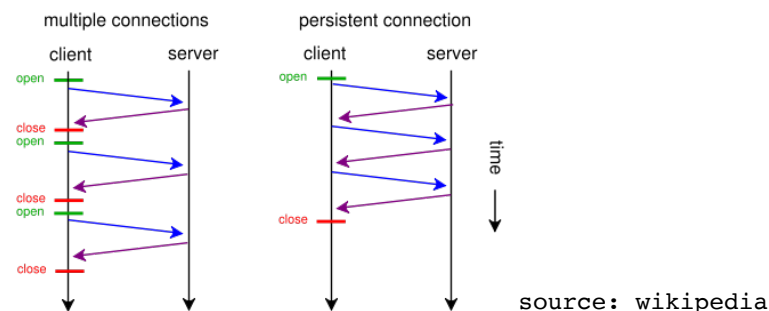
Sockets

Connexion TCP persistante

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

Connexion TCP non persistante

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



Mélanger C et C++

De préférence

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

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

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

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

Mélanger C et C++

Dans un header C

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

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

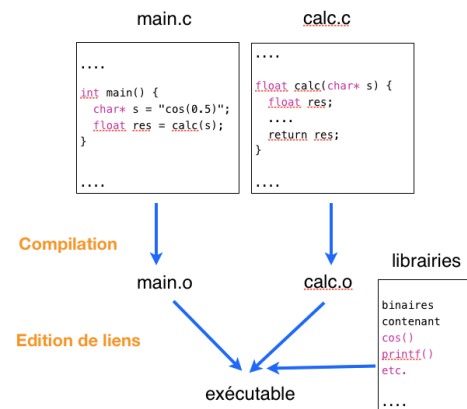
Librairies statiques et dynamiques

Librairies statiques

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

Librairies dynamiques

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



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");

    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;
}
```

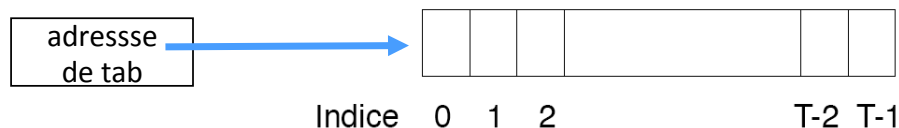
Avec LLVM sous MacOSX 10.7.1 :

Enter password: 111111
Welcome dear customer ;-)

Adresses:
0x7fff5fbff98a 0x7fff5fbff98f
0x7fff5fbff998 0x7fff5fbff900

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

Arithmétique des pointeurs



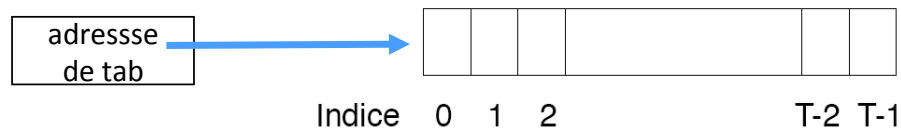
Tableaux

```
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)

Orienté objet en C

C

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

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

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

C++

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

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

Orienté objet en C

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

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

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

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

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

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

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

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

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