

Programmation orientée objet et autres concepts en C++

Eric Lecolinet

Télécom ParisTech / Dpt. Infres

Octobre 2012

License:



- [Début du cours](#) (utilisez les flèches en haut à gauche pour naviguer)
- [Index](#)
- Tout dans un seul fichier: [HTML](#) / [PDF](#)

Sommaire

- [Index](#)
- Chapitre 0 : Introduction
- Chapitre 1 : Des objets et des classes
- Chapitre 2 : Héritage
- Chapitre 3 : Mémoire
- Chapitre 4 : Constance
- Chapitre 5 : Templates et STL
- Chapitre 6 : Passage par valeur et par référence
- Chapitre 7 : Surcharge des opérateurs et smart pointers
- Chapitre 8 : Compléments sur les types
- Chapitre 9 : Traitement des erreurs
- Chapitre 10 : Héritage multiple

Liens et références

- Les [travaux pratiques](#) associés à ce cours
- Un petit tutoriel: [de Java à C++](#)
- Une intro au [toolkit graphique Qt](#)
- Le site de [cplusplus.com](#) et son [manuel de référence](#)
- Ainsi que ceux de [C++ Reference](#) et [STL SGI](#)
- La documentation automatique avec [Doxygen](#)
- Les extensions [Boost](#)
- La foire Aux Questions: [C++ FAQ](#)
- Le cours C++ de [Christian Casteyde](#)
- Le site de [Bjarne Stroustrup](#) l'auteur du C++

Brève historique

Langage C

C++

- Bjarne Stroustrup, AT&T Bell Labs
- initialement une **extension objet du C** (pré-compileur)
- plusieurs versions: de 1985 à normalisation ANSI / ISO (1998, 2003)
- **C++11** : nouvelle version (août 2011)

Java

- inspiré de la partie objet de C++ (et d'ADA, Smalltalk ...)

C#

- inspiré de Java, de C++, etc.

Brève historique (2)

Objective C

- une autre **extension objet du C** promue par NeXt puis Apple (MacOSX, iOS)
- simple mais puissant et compatible avec **C et C++**
- syntaxe inhabituelle inspirée de Smalltalk

Python, Ruby

- visent la simplicité d'écriture et la flexibilité
- interprétés et basés sur le **typage dynamique** (comme Objective C)

Evolutions récentes

- progression de Objective C, C# et Python aux dépens de Java
- C/C++ reste stable et (largement) majoritaire

C++ versus C

Avantage : compatibilité C/C++

- même syntaxe de base
- code C facilement compilable en C++ (faibles différences syntaxiques)
- un programme peut combiner des fichiers C et C++
 - => idéal pour rendre un programme C orienté objet

Inconvénient : compatibilité C/C++

- C++ hérite de certains choix malencontreux du langage C !

C++ versus Java

Ressemblances

- syntaxe en partie similaire
- fonctionnalités objet similaires (à part l'héritage multiple)

Différences

- gestion mémoire (pas de garbage collecting, etc.)
- héritage multiple
- redéfinition des opérateurs
- templates et STL
- pas de threads dans le langage (mais bibliothèques ad hoc)
- langage compilé (et ... plus rapide !)

En résumé

C++ = langage objet **ET** procédural

- contrairement à Java, purement OO
- vision anachronique: C++ = "sorte de mélange de C et de Java"

Bon côtés

- orienté objet avec l'efficacité du C (et compatible avec C)
- richesse du langage...

Moins bon côtés

- difficultés héritées du C plus quelques ambiguïtés
- richesse du langage: programmes parfois inutilement complexes
 - *Things should be made as simple as possible but not any simpler* (A. Einstein)

Références et liens

Références

- Le langage C++, Bjarne Stroustrup (Pearson)
- www.cplusplus.com/reference
- www.cppreference.com

Liens utiles

- Travaux Pratiques: www.enst.fr/~elc/cpp/TP.html
- Introduction au toolkit graphique Qt: www.enst.fr/~elc/qt
- Boost C++ Libraries: www.boost.org
- voir aussi liens en 1ere page...

Compilateurs

Attention aux **incompatibilités** entre les versions de C++

- syntaxiques
- binaires
 - le programme et les **bibliothèques** doivent être compilés avec des versions **compatibles** de C++

Salles Unix à Télécom

- **g++** version 4.*.*

Premier chapitre

Des objets et des classes ...

Programme C++

Un programme C++ est constitué :

- de classes réparties dans plusieurs fichiers (comme en Java)
- (éventuellement) de fonctions et variables globales (comme en C)

Chaque fichier peut comprendre :

- un nombre arbitraire de classes (si ça a un sens ...)

Déclarations et définitions

Comme en langage C :

- **déclarations** dans fichiers headers : xxx.h (ou .hpp ou .hh)
- **définitions** dans fichiers d'implémentation : xxx.cpp (ou .cc ou .C)

Règles de base :

- à chaque .cpp (**définitions**) correspond un .h (**déclarations**)
- le .h déclare l'**API publique**, c'est une interface avec le monde extérieur

Déclarations et définitions

Comme en langage C :

- **déclarations** dans fichiers headers : xxx.h (ou .hpp ou .hh)
- **définitions** dans fichiers d'implémentation : xxx.cpp (ou .cc ou .C)

Règles de base :

- à chaque .cpp (**définitions**) correspond un .h (**déclarations**)
- le .h déclare l'**API publique**, c'est une interface avec le monde extérieur

Remarque

- on peut aussi "cacher" des variables ou des fonctions en les déclarant :
 - directement dans le .cpp
 - dans un header privé (exemple: xxx_impl.h) => surtout pour les librairies

Déclaration de classe

```
// fichier Circle.h : header contenant les déclarations

class Circle {
public:
    int x, y;                // variables d'instance
    unsigned int radius;

    virtual void setRadius(unsigned int); // méthodes d'instance
    virtual unsigned int getRadius() const;
    virtual unsigned int getArea() const;
    ....
};                          // !NE PAS OUBLIER LE ;
```

Remarques

- le ; final est obligatoire après la }
- même sémantique que Java, syntaxe similaire mais ...
- l'implémentation est (de préférence) séparée des déclarations

Implémentation de classe

```
// Rappel des déclarations

class Circle {
public:
    int x, y;                // variables d'instance
    unsigned int radius;

    virtual void setRadius(unsigned int);    // méthodes d'instance
    virtual unsigned int getRadius() const;
    virtual unsigned int getArea() const;
};
```

Implémentation

```
// fichier Circle.cpp : contient l'implémentation

#include "Circle.h"        // ne pas oublier d'inclure le header !

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

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

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

Instanciation

```
// fichier main.cpp : main() est le point d'entrée du programme

#include "Circle.h"        // ne pas oublier d'inclure le header

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

Ne pas mettre main() dans Circle.cpp

- sinon la classe Circle ne sera pas réutilisable dans un autre programme !

Instanciation

```
// fichier main.cpp

#include "Circle.h"

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

new cree un objet (= nouvelle instance de la classe)

- allocation mémoire
- puis appel du **constructeur** (à suivre)

c = variable locale qui **pointe** sur le nouvel objet

- **c** est un pointeur

Comparaison avec Java

Pointeur C++ vs. référence Java

```
C++: Circle* c = new Circle();           // pointeur C++
Java: Circle c = new Circle();           // référence Java
```

- dans les 2 cas: une **variable** qui pointe sur un objet
 - attention: "**référence**" a un autre sens en C++ (à suivre...)

Gestion mémoire

- Java détruit les objets qui n'ont plus de référent (garbage collector)
- C++ nécessite une **destruction explicite** par l'opérateur **delete**

Accès aux variables d'instance

```
#include "Circle.h"

int main() {
    Circle* c1 = new Circle();

    c1->x = 100; // noter la ->
    c1->y = 200;
    c1->radius = 35;

    Circle *c2 = new Circle();
    c2->x = c1->x;
}
```

Chaque objet possède sa propre copie des variables d'instance

- noter l'utilisation de la `->` (comme en C, mais `.` en Java)
 - `c->x` équivaut à: `(*c).x`
- **encapsulation** => restreindre l'accès aux variables d'instance

Appel des méthodes d'instance

```
int main() {
    Circle* c1 = new Circle();
    Circle* c2 = new Circle();

    // attention: c->x, c->y, c->radius pas initialisés !

    unsigned int r = c1->getRadius();
    unsigned int a = c2->getArea();
}
```

Toujours appliquées à un objet

- ont accès à **toutes** les variables de **cet** objet
- propriété fondamentale de l'orienté objet !

```
unsigned int Circle::getRadius() const { // dans Circle.cpp
    return radius;
}
```

Constructeurs

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

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

Circle* c = new Circle(100, 200, 35);           // instantiation
```

Appelés à l'instanciation de l'objet

- pour initialiser les variables d'instance

Constructeurs (suite)

Deux syntaxes (quasi) équivalentes

```
Circle::Circle(int _x, int _y, unsigned int _r) { // comme en Java
    x = _x; y = _y; radius = _r;
}

Circle::Circle(int _x, int _y, unsigned int _r) // propre à C++
    : x(_x), y(_y), radius(_r) {
}
```

- la 2eme forme est préférable (vérifie l'ordre des déclarations)

Chaînage des constructeurs

- appel implicite des constructeurs des super-classes
- dans l'ordre descendant

Constructeur par défaut

Si aucun constructeur dans la classe

- on peut cependant écrire :

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

- car C++ crée un constructeur par défaut **mais qui ne fait rien !**
 - => variables **pas initialisées** (contrairement à Java)

Conseils

- toujours définir **au moins un** constructeur et toujours **initialiser** les variables
- de plus, c'est une bonne idée de définir un **constructeur sans argument** :

```
class Circle {
    int x, y;
    unsigned int radius;
public:
    Circle(int x, int y, unsigned int r);
    Circle() : x(0), y(0), radius(0) { }
    ....
};
```

Destruction

```
Circle* c = new Circle(100, 200, 35);
...
delete c;           // destruction de l'objet
c = NULL;          // c pointe sur aucun object
}
```

delete détruit un objet créé par new

- pas de garbage collector (ramasse miettes) comme en Java !
 - NB: on verra plus tard comment se passer de delete
- Attention: **delete** ne met pas **c** à **NULL** !

Remarque

- **NULL** est une macro qui vaut **0** (ce n'est pas un mot-clé)

Destructeur

```
class Circle {
public:
    virtual ~Circle();           // destructeur
    ...
};

Circle* c = new Circle(100, 200, 35);
...
delete c;                       // destruction de l'objet
c = NULL;
```

Fonction appelée à la destruction de l'objet

- un seul destructeur par classe (pas d'argument)

Chaînage des destructeurs

- dans l'ordre **ascendant** (inverse des constructeurs)

delete & destructeur

Attention

- c'est **delete** qui détruit l'objet (qu'il y ait ou non un destructeur)
- le destructeur (s'il existe) est juste une fonction appelée avant la destruction

Quand faut-il un destructeur ?

- si l'objet a des vars d'instance qui pointent vers des objets à détruire
- si l'objet a ouvert des fichiers, sockets... qu'il faut fermer
- pour la **classe de base** d'une hiérarchie de classes

Et en Java ...?

delete & destructeur

Et en Java ?

- `delete` n'existe pas car GC (garbage collector)
- la méthode `finalize()` joue le même rôle que le destructeur, mais:
 - pas de chaînage des "finaliseurs"
 - appel **non déterministe** par le GC (on ne sait pas quand l'objet est détruit)

Surcharge (overloading)

Plusieurs méthodes

- ayant le même nom
- mais des **signatures** différentes
- pour une **même** classe

```
class Circle {  
    Circle();  
    Circle(int x, int y, unsigned int r);  
    ....  
};
```

Remarques

- la valeur de retour ne suffit pas à distinguer les signatures
- applicable aux fonctions "classiques" (hors classes)

Paramètres par défaut

```
class Circle {
    Circle(int x, int y, unsigned int r = 10);
    ....
};

Circle* c1 = new Circle(100, 200, 35);
Circle* c2 = new Circle(100, 200);           // radius vaudra 10
```

Remarques

- en nombre quelconque mais toujours en dernier
- erreur de compilation s'il y a des ambiguïtés :

```
class Circle {
    Circle();
    Circle(int x, int y, unsigned int r = 10);           // OK
    Circle(int x = 0, int y = 0, unsigned int r = 10);  // AMBIGU!
    ....
};
```

Variables de classe

```
class Circle {                                     // fichier Circle.h
public:
    static const float PI;                         // variable de classe
    int x, y;                                       // variables d'instance
    unsigned int radius;
    ...
};
```

Représentation unique en mémoire

- mot-clé **static**
- existe toujours (même si la classe n'a pas été instanciée)

Remarques

- **const** (optionnel) indique que la valeur est **constante**
- notion similaire aux variables "statiques" du C (d'où le mot-clé)

Définition des variables de classe

Les variables de classe doivent également être définies

- dans un (et **un seul**) .cpp, sans répéter **static**
- ce n'est pas nécessaire en Java ou C#

```
// dans Circle.cpp
const float Circle::PI = 3.1415926535;    // noter le ::
```

Exception

- les variables de classe **const int** peuvent être définies dans les headers

```
// dans Circle.h
static const int TAILLE_MAX = 100;
```

Méthodes de classe

```
// déclaration: fichier Circle.h

class Circle {
public:
    static const float PI;
    static float getPI() {return PI;}
    ...
};

// appel: fichier main.cpp

float x = Circle::getPI();
```

Ne s'appliquent pas à un objet

- mot-clé **static**
- similaire à une fonction "classique" du C (mais évite **collisions de noms**)

N'ont accès qu'aux variables de classe !

Namespaces

namespace = espace de nommage

- solution ultime aux collisions de noms
- existent aussi en C#, similaires aux packages de Java

```
namespace Geom {           // dans Circle.h
    class Circle {
        ...
    };
}
-----
namespace Math {          // dans Math.h
    class Circle {       // une autre classe Circle...
        ...
    };
}
-----
#include "Circle.h"       // dans main.cpp
#include "Math.h"

int main() {
    Geom::Circle* c1 = new Geom::Circle();
    Math::Circle* c2 = new Math::Circle();
    ...
}
```

Namespaces

using namespace

- modifie les règles de portée
 - les symboles déclarés dans ce namespace deviennent directement accessibles
- similaire à `import` en Java

```
namespace Geom {           // dans Circle.h
    class Circle {
        ...
    };
}
-----
#include "Circle.h"       // dans main.cpp
using namespace Geom;

int main() {
    Geom::Circle* c1 = new Geom::Circle();
    Circle* c2 = new Circle();           // OK grace a using namespace...
    ...
}
```

Bibliothèque standard d'E/S

```
#include <iostream> // E/S du C++
#include "Circle.h"

using namespace std;

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

    cout << "radius= " << c->getRadius()
    << "area= " << c->getArea()
    << endl;

    std::cerr << "c = " << c << std::endl;
}
```

Concaténation des arguments via << ou >>

- `std::cout` : sortie standard
- `std::cerr` : sortie des erreurs
- `std::cin` : entrée standard (utiliser `>>` au lieu de `<<`)

Encapsulation / droits d'accès

```
class Circle {
private:
    int x, y;
    unsigned int radius;
public:
    static const float PI;
    Circle();
    Circle(int x, int y, unsigned int r);
};
```

Trois niveaux

- `private` (le `défaut` en C++) : accès réservé à cette classe
- `protected` : idem + sous-classes
- `public`
- NB: Java a un niveau `package` (défaut), C++ a également `friend`

Encapsulation / droits d'accès (2)

```
class Circle {  
  // private:                               // private par défaut  
  int x, y;  
  unsigned int radius;  
  public:  
  static const float PI;                   // PI est public car const  
  Circle();  
  Circle(int x, int y, unsigned int r);  
};
```

Règles usuelles d'encapsulation

- l'API (méthodes pour communiquer avec les autres objets) est **public**
- l'implémentation (variables et méthodes internes) est **private** ou **protected**

Encapsulation / droits d'accès (3)

```
class Circle {  
  friend class Manager;  
  friend bool equals(const Circle*, const Circle*);  
  ...  
};
```

friend donne accès à tous les champs de Circle

- à une autre **classe** : Manager
- à une **fonction** : bool equals(const Circle*, const Circle*)

struct

struct = class + public

```
struct Truc {  
    ...  
};
```

- équivaut à :

```
class Truc {  
public:  
    ...  
};
```

struct

- est équivalent à **class** en C++
- n'existe pas en Java
- existe en C# mais ce n'est pas une **class**
- existe en C mais c'est juste un agrégat de variables

Méthodes d'instance: où est la magie ?

Toujours appliquées à un objet

```
class Circle {  
    unsigned int radius;  
    int x, y;  
public:  
    virtual unsigned int getRadius() const;  
    virtual unsigned int getArea() const;  
};  
  
int main() {  
    Circle* c = new Circle(100, 200, 35);  
    unsigned int r = c->getRadius(); // OK  
    unsigned int a = getArea(); // INCORRECT: POURQUOI?  
}
```

Et pourtant :

```
unsigned int Circle::getArea() const {  
    return PI * getRadius() * getRadius(); // idem  
}  
  
unsigned int Circle::getRadius() const {  
    return radius; // comment getRadius() accede a radius ?  
}
```

Le *this* des méthodes d'instance

Paramètre caché *this*

- pointe sur l'objet qui appelle la méthode
- permet d'accéder aux variables d'instance

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

Circle* c = ...;
unsigned int a = c->getArea();
```

Transformé par le compilateur en :

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

Circle* c = ...;
unsigned int a = Circle::getArea(c);
// __ZNK6Circle7getAreaEv(c) avec g++ 4.2
```

Inline

Indique que la fonction est implémentée dans le header

```
// dans Circle.h

class Circle {
public:
    inline int getX() const {return x;}
    int getY() const {return y;} // pareil: inline est implicite
    ....
};

// inline doit être présent si fonction non-membre
inline Circle* createCircle() {return new Circle();}
```

A utiliser avec discernement

- + : rapidité à l'exécution : peut éviter un appel fonctionnel (code dupliqué)
- - : augmente taille du binaire généré
- - : lisibilité
- - : contraire au principe d'encapsulation

Point d'entrée du programme

```
int main(int argc, char** argv)
```

- même syntaxe qu'en C
- `argc` : nombre d'arguments
- `argv` : valeur des arguments
- `argv[0]` : nom du programme
- `valeur de retour` : normalement 0, indique une erreur sinon

Terminologie

Méthode versus fonction

- méthodes d'instance == fonctions membres
- méthodes de classe == fonctions statiques
- fonctions classiques == fonctions globales
- etc.

Termes interchangeables selon auteurs

Doxygen

```
/** 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.
     * voir aussi setX() et setY().
     */

    ...
};
```

Système de documentation automatique

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

Chapitre 2 : Héritage

Concept essentiel de l'OO

- héritage **simple** (comme Java)
- héritage **multiple** (à manier avec précaution : voir plus loin)

Règles d'héritage

Constructeurs

- jamais hérités

Méthodes

- héritées
- peuvent être **redéfinies** (overriding) :
 - la nouvelle méthode **remplace** celle de la superclasse
 - ! ne pas confondre surcharge et redéfinition !

Variables

- héritées
- peuvent être **surajoutées** (shadowing) :
 - la nouvelle variable **cache** celle de la superclasse
 - ! à éviter : source de confusions !

Exemple (déclarations)

```
// header Rect.h

class Rect {
    int x, y;
    unsigned int width, height;

public:
    Rect();
    Rect(int x, int y, unsigned int width, unsigned int height);

    virtual void setWidth(unsigned int);
    virtual void setHeight(unsigned int);
    virtual unsigned int getWidth() const {return width;}
    virtual unsigned int getHeight() const {return height;}
    /*...etc...*/
};

class Square : public Rect {                // dérivation de classe
public:
    Square();
    Square(int x, int y, unsigned int width);

    virtual void setWidth(unsigned int);    // redéfinition de méthode
    virtual void setHeight(unsigned int);
};
```

Exemple (implémentation)

```
class Rect { // rappel des déclarations
    int x, y;
    unsigned int width, height;
public:
    Rect();
    Rect(int x, int y, unsigned int width, unsigned int height);
    virtual void setWidth(unsigned int);
    virtual void setHeight(unsigned int);
    ...
};

class Square : public Rect {
public:
    Square();
    Square(int x, int y, unsigned int width);
    virtual void setWidth(unsigned int);
    virtual void setHeight(unsigned int);
};

// implémentation: Rect.cpp

void Rect::setWidth(unsigned int w) {width = w;}
void Square::setWidth(unsigned int w) {width = height = w;}

Rect::Rect() : x(0), y(0), width(0), height(0) {}

Square::Square() {}
Square::Square(int x, int y, unsigned int w) : Rect(x, y, w, w) {}

/*...etc...*/
```

Remarques

Dérivation de classe

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

- héritage public des **méthodes** et **variables** de la super-classe
 - = extends de Java
 - peut aussi être private ou protected

Chaînage des constructeurs

```
Square::Square() {}
Square::Square(int x, int y, unsigned int w) : Rect(x, y, w, w) {}
```

- 1er cas : appel implicite de Rect()
- 2e cas : appel explicite de Rect(x, y, w, w)
- Pareil en Java, sauf syntaxe : mot-clé **super()**

Headers et inclusions multiples

Problème ?

```
class Shape { // dans Shape.h
    ...
};

//-----

#include "Shape.h" // dans Circle.h
class Circle : public Shape {
    ...
};

//-----

#include "Shape.h" // dans Rect.h
class Rect : public Shape {
    ...
};

//-----

#include "Circle.h" // dans main.cpp
#include "Rect.h"

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

Headers et inclusions multiples (2)

Problème : transitivité des inclusions

- le header Shape.h est inclus 2 fois dans main.cpp
 - => la classe Shape est déclarée 2 fois => erreur de syntaxe !

Pour empêcher les redéclarations

```
#ifndef _Shape_h_ // dans Shape.h
#define _Shape_h_

class Shape {
    ...
};

#endif
```

- A faire systématiquement en C / C++ pour tous les headers
- Note: #import fait cela automatiquement mais n'est pas standard

Compléments sur les headers

- les " " ou <> précisent l'espace de recherche

```
#include <iostream>           // cherche dans /usr/include ou similaire
#include "Circle.h"           // cherche dans répertoire courant
```

- l'option -I du compilateur ajoute un répertoire de recherche pour <>
 - exemple: -I/usr/X11R6/include

Polymorphisme

3eme caractéristique fondamentale de la POO

```
class Rect {
    int x, y;
    unsigned int width, height;
public:
    virtual void setWidth(unsigned int w) {width = w;}
    ...
};

class Square : public Rect {
public:
    virtual void setWidth(unsigned int w) {width = height = w;}
    ...
};

int main() {
    Rect* obj = new Square();           // obj est un Square ou un Rect ?
    obj->setWidth(100);                 // quelle methode est appelée ?
}
```

Polymorphisme et liaison dynamique

Polymorphisme

- un objet peut être vu sous plusieurs formes

```
Rect* obj = new Square(); // obj est un Square ou un Rect ?
obj->setWidth(100);      // quelle methode est appelée ?
```

Liaison dynamique (ou "tardive")

- la méthode liée à l'instance est appelée
- le choix de la méthode se fait à l'exécution
- mécanisme essentiel de l'OO !

Liaison statique

- le contraire : la méthode liée au pointeur est appelée

Méthodes virtuelles

Deux cas possibles en C++

```
class Rect {
public:
    virtual void setWidth(unsigned int); // methode virtuelle
};

class Square : public Rect {
public:
    virtual void setWidth(unsigned int);
};

int main() {
    Rect* obj = new Square();
    obj->setWidth(100);
}
```

Méthodes virtuelles

- mot clé virtual => liaison dynamique : Square::setWidth() est appelée

Méthodes non virtuelles

- PAS de mot clé virtual => liaison statique : Rect::setWidth() est appelée

Pourquoi des méthodes virtuelles ?

Cohérence logique

- les méthodes d'instance doivent généralement être virtuelles
- pour éviter les incohérences !
- exemple :

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

- setWidth() pas virtuelle => Square pas carré !

Java et C#

- même comportement que méthodes virtuelles

Pourquoi des méthodes NON virtuelles ?

!!! DANGER !!!

- code **erroné** si la fonction est redéfinie plus tard dans une sous-classe !

Exceptionnellement

- pour optimiser l'exécution si on est sûr que la fonction **ne sera pas redéfinie**
 - accesseurs, souvent "inline" dans ce cas
 - cas extrêmes, méthode appelée 10 000 000 fois...
- en Java on déclarerait la méthode **final**
 - n'existe pas en C++ => mettre un **commentaire**

Redéfinition des méthodes virtuelles

```
class Rect {
public:
    virtual void setWidth(unsigned int);    // virtual nécessaire
};

class Square : public Rect {
public:
    /*virtual*/ void setWidth(unsigned int); // virtual implicite
};
```

Les redéfinitions des méthodes virtuelles sont toujours virtuelles

- même si **virtual** est omis
 - => **virtual** particulièrement important dans les **classes de base** !
- elles doivent toutes avoir la même signature
 - sauf pour le **type de retour** (covariance des types)

Surcharge des méthodes virtuelles

Il faut redéfinir **toutes** les variantes

```
class Rect {
public:
    virtual void setWidth(unsigned int);
    virtual void setWidth(double);    // surcharge de setWidth()
};

class Square : public Rect {
public:
    virtual void setWidth(unsigned int); // PAS correct, redéfinir les deux
};

class Square : public Rect {
public:
    virtual void setWidth(unsigned int);
    virtual void setWidth(double);    // correct
};
```

Méthode abstraite

Spécification d'un concept dont la réalisation peut varier

- ne peut **pas** être implémentée
- **doit** être redéfinie (et implémentée) dans les sous-classes ad hoc

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

- en C++ : **virtual** et **= 0** (pure virtual function)
- en Java et en C# : **abstract**

Classe abstraite

Contient au moins une méthode abstraite

- => ne peut **pas** être instanciée

Les classes héritées instanciables :

- doivent implémenter **toutes** les méthodes abstraites

```
class Shape {                                // classe abstraite  
public:  
    virtual void setWidth(unsigned int) = 0;  
    ...  
};  
  
class Rect : public Shape {                 // Rect peut être instanciée  
public:  
    virtual void setWidth(unsigned int);  
    ...  
};
```

Classes abstraites (2)

Objectifs

- "commonaliser" les déclarations de méthodes (généralisation)
 - -> permettre des traitements génériques sur une hiérarchie de classes
- imposer une **spécification**
 - -> que les sous-classes doivent obligatoirement implémenter
- peut aussi servir pour **encapsulation**
 - -> séparer la spécification (API) et l'implémentation
 - -> les implémentations sont dans les sous-classes instanciables

Remarque

- pas de mot-clé **abstract** comme en Java
 - il suffit qu'une méthode soit abstraite

Exemple

```
class Shape { // classe abstraite
    int x, y;
public:
    Shape() : x(0), y(0) {}
    Shape(int _x, int _y) : x(_x), y(_y) {}

    virtual int getX() const {return x;}
    virtual int getY() const {return y;}

    virtual unsigned int getWidth() const = 0; // methodes
    virtual unsigned int getHeight() const = 0; // abstraites
    virtual unsigned int getArea() const = 0;

    // ... idem pour setters
};

class Circle : public Shape {
    unsigned int radius;
public:
    Circle() : radius(0) {}
    Circle(int x, int y, unsigned int r) : Shape(x, y), radius(0) {}

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

    // redefinition et implementation des methodes abstraites
    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;}

    // ... idem pour setters
}
```

Traitements génériques

```
#include <iostream>
#include "Shape.h"
#include "Rect.h"
#include "Square.h"
#include "Circle.h"

int main(int argc, char** argv) {

    Shape** tab = new Shape*[10];          // tableau de Shape*
    unsigned int count = 0;

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

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

- Note:

- traitements génériques != programmation générique (que l'on verra plus tard)
- but (en gros) similaire, approche différente

Bénéfices du polymorphisme (1)

Gestion unifiée

- des classes dérivant de la classe abstraite
- sans avoir besoin de connaître leur type
- contrairement aux langages non objet classiques (par exemple C)

```
// fichier print.cpp

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

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

Evolutivité

- rajout de nouvelles classes sans modification de l'existant

Remarque en passant sur les "tableaux" C/C++

```
// fichier print.cpp

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

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

En fait `tab` n'est pas un tableau mais un pointeur !

- qui pointe sur le premier élément
- => `count` est indispensable pour savoir où le tableau se termine !

Remarque: cette notation est équivalente

```
void printAreas(Shape* tab[], int count) {
    ....
}
```

Bénéfices du polymorphisme (2)

Spécification indépendante de l'implémentation

- les classes se conforment à une spécification commune
- => indépendance des implémentations des divers "modules"
- => développement en parallèle par plusieurs équipes

Interfaces

Classes totalement abstraites

- **toutes** les méthodes sont abstraites
- **aucune** implémentation
- -> pure spécification d'**API** (Application Programming Interface)

En C++: cas particulier de classe abstraite

- pas de mot-clé **interface** comme en Java
- pas indispensable car C++ supporte l'héritage multiple

Exemple d'interface

```
class Shape { // interface
    // pas de variables d'instance ni de constructeur
public:
    virtual int getX() const = 0;           // abstract
    virtual int getY() const = 0;         // abstract
    virtual unsigned int getWidth() const = 0; // abstract
    virtual unsigned int getHeight() const = 0; // abstract
    virtual unsigned int getArea() const = 0; // abstract
};

class Circle : public Shape {
    int x, y;
    unsigned int radius;
public:
    Circle();
    Circle(int x, int y, unsigned int r = 10);

    // getX() et getY() doivent être implémentées
    virtual int getX() const {return x;}
    virtual int getY() const {return y;}
    virtual unsigned int getRadius() const {return radius;}
    ...etc...
}
```

Complément: factorisation du code

Eviter les duplications de code

- gain de temps
- évite des incohérences
- lisibilité par autrui
- maintenance : facilite les évolutions ultérieures

Comment ?

- technique de base : **héritage**
 - -> découpage astucieux des méthodes, méthodes intermédiaires ...
- rappel des méthodes des super-classes :

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

Classes imbriquées (1)

```
class Rect {
    class Point {                // classe imbriquée
        int x, y;
    public:
        Point(x, y);
    };

    Point p1, p2;                // variables d'instance

public:
    Rect(int x1, int y1, int x2, int y2);
};
```

Technique de composition très utile

- souvent préférable à l'héritage multiple (à suivre)

Visibilité des champs depuis la classe imbriquée

- les champs de Rect sont **automatiquement visibles** depuis Point en Java
- mais pas en C++ !
 - il faut explicitement rajouter un pointeur vers la classe contenante

Classes imbriquées (2)

```
class Rect {
    class Point {                // classe imbriquée
        int x, y;
    public:
        Point(x, y);
    };

    Point p1, p2;                // variables d'instance

    public:
        Rect(int x1, int y1, int x2, int y2);
};
```

Implémentation (si pas dans le header)

```
Rect::Rect(int x1, int y1, int x2, int y2)
    : p1(x1,y1), p2(x2,y2) { }    // appel du const. de la classe imbriquée

Rect::Point::Point(int _x, int _y) // Rect::Point::Point !
    : x(_x), y(_y) { }
```

Méthodes virtuelles: comment ça marche ?

Tableau de pointeurs de fonctions (vtable)

- 1 vtable par classe
- chaque objet pointe vers la **vtable** de sa classe
- => coût un peu élevé (double indirection)

```
class Vehicle {
    int color = 0;
    public void start() { /* ... */ }
    public int getColor() { return this.color; }
}

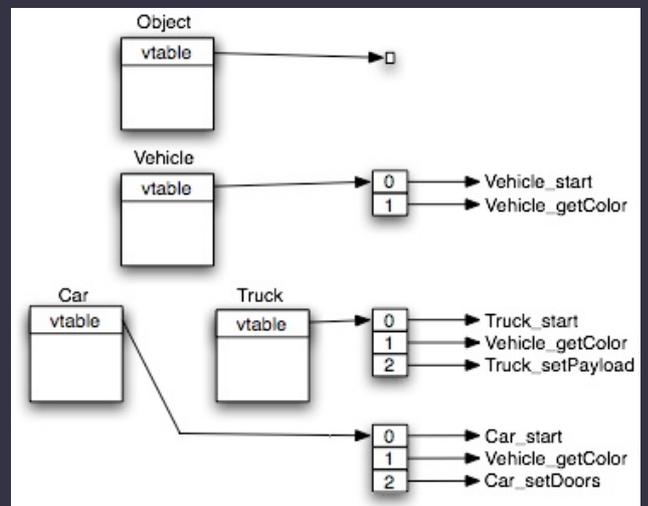
class Car extends Vehicle {
    int doors = 0;
    public void start() { /* ... */ }
    public void setDoors(int doors) { this.doors = doors; }
}

class Truck extends Vehicle {
    int payload = 0;
    public void start() { /* ... */ }
    public void setPayload(int payload) { this.payload = payload; }
}

public class OO {
    public static void main(String[] args) {
        Car car = new Car();
        Truck truck = new Truck();

        int color = car.getColor();
        car.getColor();

        Vehicle vehicle = null;
        vehicle = car; // note: no cast!
        vehicle.start(); // calls Car.start()
        car.start(); // calls Car.start()
        vehicle = truck;
        vehicle.start(); // calls Truck.start()
    }
}
```



<http://www.antlr.org/wiki/display/CS652/Implementing+Polymorphism>

Chapitre 3 : Mémoire

Les différents types de mémoire

- **mémoire statique / globale** : réservée dès la compilation: variables **static** ou globales
- **pile / stack** : variables locales ("automatiques") et paramètres des fonctions
- **mémoire dynamique / tas / heap** : allouée à l'exécution par **new** (**malloc** en C)

```
void foo() {  
    static int count = 0;    // statique  
    count++;  
  
    int i = 0;              // pile  
    i++;  
  
    int* p = new int(0);    // dynamique  
    (*p)++;                // les parenthèses sont nécessaires!  
}
```

- que valent **count**, **i**, ***p** si on appelle **foo()** deux fois ?

Mémoire

Durée de vie

- **mémoire statique / globale** : toute la durée du programme
- **pile** : pendant l'exécution de la fonction
- **mémoire dynamique** : de **new** à **delete** (de **malloc** à **free** en C)

```
void foo() {  
    static int count = 0;    // statique  
    int i = 0;              // pile  
    int* p = new int(0);    // dynamique  
}
```

A la sortie de la fonction

- **count** existe encore (et conserve sa valeur)
- **i** est détruite
- **p** est détruite (elle est dans la pile) mais **pas ce qu'elle pointe**
 - => attention aux fuites mémoire (pas de ramasse miettes en C/C++)

Mémoire : compléments

```
// fichier toto.cpp

bool is_valid = true; // globale
static const char* errmsg = "Valeur invalide"; // statique globale

void foo() {
    static int count = 0; // statique locale
    int i = 0; // pile
    int* p = new int(0); // dynamique
    is_valid = false;
    cerr << errmsg << endl;
}
```

- les variables globales sont **dangereuses** !!!
- il existe un 4e type de mémoire : la **mémoire constante**/read only (parfois appelée statique !)

Java

- pas de variables **globales** ni **static** (sauf dans les classes)
- **new** pas possible sur un type de base

Mémoire et objets

C++ permet d'allouer des objets

- dans les **trois** types de mémoire, contrairement à Java !

```
void foo() {
    static Square a(5,5,20); // statique
    Square b(5,5,20); // pile
    Square* c = new Square(5,5,20); // dynamique, seul cas en Java
}
```

- les variables a et b **contiennent** l'objet
 - impossible en Java : que des types de base ou des références dans la pile
- la variable c **pointe** vers l'objet
 - même chose qu'en Java (sauf qu'il n'y a pas de ramasse miettes en C/C++)

Création et destruction des objets

```
void foo() {  
    static Square a(5,5,20);           // statique  
    Square b(5,5,20);                 // pile  
    Square* c = new Square(5,5,20);   // dynamique  
}
```

Dans tous les cas

- **Constructeur** appelé quand l'objet est créé
 - ainsi que ceux des superclasses (chaînage descendant des constructeurs)
- **Destructeur** appelé quand l'objet est détruit
 - ainsi que ceux des superclasses (chaînage ascendant des destructeurs)

Création et destruction des objets (2)

```
void foo() {  
    static Square a(5,5,20);           // statique  
    Square b(5,5,20);                 // pile  
    Square* c = new Square(5,5,20);   // dynamique  
}
```

new et delete

- à chaque **new** doit correspondre **un** (et un seul) **delete**
 - **delete p** ne fait rien si p vaut **NULL** (ou 0)
- ne pas faire **delete** sur des objets en mémoire statique ou dans la pile
 - ils sont détruits automatiquement

Comment se passer de delete ?

- avec des **smart pointers** (à suivre)
- la mémoire est toujours récupérée en fin de programme
 - aucun **delete** : solution acceptable si peu d'objets pas trop gros

. versus ->

```
void foo() {
    static Square a(5,5,20);           // statique
    Square b(5,5,20);                 // pile
    Square* c = new Square(5,5,20);   // dynamique

    unsigned int w = a.getWidth();
    int y = b.getY();
    int x = c->getX();
}
```

- . pour accéder à un membre d'un objet (ou d'une **struct** en C)
- -> même chose depuis un pointeur (comme en C)
- c->getX() == (*c).getX()

Objets contenant des objets

```
class Dessin {
    static Square a;           // var. de classe qui contient l'objet
    Square b;                 // var. d'instance qui contient l'objet

    Square* c;                // var. d'instance qui pointe vers un objet (comme Java)
    static Square* d;         // var. de classe qui pointe vers un objet (comme Java)
};
```

Durée de vie

- l'objet a est **automatiquement** créé/détruit en même temps que le programme
- l'objet b est **automatiquement** créé/détruit en même temps que l'instance de Dessin
- l'objet pointé par c est typiquement :
 - créé par le constructeur de Dessin
 - détruit par le destructeur de Dessin

Création de l'objet

```
class Dessin {
    static Square a;
    Square b;
    Square* c;

public:
    Dessin(int x, int y, unsigned int w) :
        b(x, y, w), // appelle le constructeur de b
        c(new Square(x, y, w)) { // crée l'objet pointé par c
    }
};

// dans un (et un seul) fichier .cpp
Square Dessin::a(10, 20, 300); // ne pas repeter "static"
```

Qu'est-ce qui manque ?

Destruction de l'objet

Il faut un destructeur !

- chaque fois qu'un constructeur fait un **new** (sinon fuites mémoires)

```
class Dessin {
    Square b;
    Square* c;

public:
    Dessin(int x, int y, unsigned int w) :
        b(x, y, w),
        c(new Square(x, y, w)) {
    }

    virtual ~Dessin() {delete c;} // détruire l'objet créé par le constructeur
};
```

Remarques

- b pas créé avec new => pas de delete
- destructeurs généralement **virtuels** pour avoir le polymorphisme

Qu'est-ce qui manque ?

Initialisation et affectation

```
class Dessin {
    Square b;
    Square* c;
public:
    Dessin(int x, int y, unsigned int w);
    virtual ~Dessin() {delete c;}
};

void foo() {
    Dessin d1(0, 0, 50);    // d1 contient l'objet
    Dessin d2(10, 20, 300);

    d2 = d1;                // affectation (d'un objet existant)

    Dessin d3(d1);         // initialisation (d'un nouvel objet)
    Dessin d4 = d1;        // idem
}
```

Quel est le probleme ?

- quand on sort de foo() ...

Initialisation et affectation

```
void foo() {
    Dessin d1(0, 0, 50);
    Dessin d2(10, 20, 300);
    d2 = d1;                // affectation
    Dessin d3(d1);         // initialisation
    Dessin d4 = d1;        // idem
}
```

Problème

- le contenu de d1 est copié champ à champ dans d2, d3 et d4
 - => tous les Dessins pointent sur la même instance de Square !
 - => elle est détruite 4 fois quand on sort de foo (et les autres jamais) !

Solution

- il faut de la copie profonde, la copie superficielle ne suffit pas
- problème général qui n'est pas propre à C/C++ :
 - quel que soit le langage chaque dessin devrait avoir son propre Square

1ere solution : interdire la copie d'objets

La copie d'objets est dangereuse

- s'ils contiennent des **pointeurs** ou des **références** !

Solution de base : pas de copie, comme en Java

- seuls les types de base peuvent être copiés avec l'opérateur = en Java

```
class Dessin {
    ....
private:
    Dessin(const Dessin&);           // initialisation: Dessin a = b;
    Dessin& operator=(const Dessin&); // affectation: a = b;
};
```

- déclarer privés l'opérateur d'initialisation (**copy constructor**) et d'affectation (**operator=**)
- implémentation inutile
- interdit également la copie pour les sous-classes (sauf si elles redéfinissent ces opérateurs)

2eme solution : redéfinir la copie d'objets

Solution avancée : copie profonde

- en C++: les 2 opérateurs recopient les objets pointés (et non les pointeurs)
- en Java: même chose via une méthode "copy" ou "clone"

```
class Dessin : public Graphique {
    ....
public:
    Dessin(const Dessin&);           // initialisation: Dessin a = b;
    Dessin& operator=(const Dessin&); // affectation: a = b;
    ....
};
```

```
Dessin::Dessin(const Dessin& from) : Graphique(from) {
    b = from.b;

    if (from.c != NULL) c = new Square(*from.c); // copie profonde
    else c = NULL;
}
```

```
Dessin& Dessin::operator=(const Dessin& from) {
    Graphique::operator=(from);

    b = from.b;

    delete c;
    if (from.c != NULL) c = new Square(*from.c); // copie profonde
    else c = NULL;
}
```

```
    return *this;
}
```

Compléments

Tableaux: `new[]` et `delete[]`

```
int* tab = new int[100];
delete [] tab;           // ne pas oublier les []
```

Ne pas mélanger les opérateurs !

```
x = new          => delete x
x = new[]       => delete[] x
x = malloc()    => free(x)   // éviter malloc() et free() en C++
```

Redéfinition de `new` et `delete`

- possible, comme pour presque tous les opérateurs du C++

Méthodes virtuelles

- méthodes virtuelles => destructeur virtuel
- ne le sont plus dans les constructeurs / destructeurs !

Chapitre 4 : Constance

Déclarations de constantes

1) Macros du préprocesseur

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

- substitution textuelle **avant** la compilation
- obsolète et dangereux, à éviter quand c'est possible

2) Enumérations

```
enum {PORT = 3000};
enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}; // Day = nom de l'enum
```

- pour définir des valeurs entières, commencent à 0 par défaut
- existent aussi en Java (plus sophistiquées : classes spéciales)

3) Variables constantes

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

- doivent être initialisées
- comme **final** en Java

Paramètres et champs constants

Paramètres des fonctions

```
char* strcat(char* s1, const char* s2); // fonction de la lib C
```

- `strcat()` ne peut pas modifier le 2e argument

Champs d'une classe

```
class User {  
    const int id;  
    const string name;  
public:  
    User(int i, string n) : id(i), name(n) {}  
};
```

- ces variables d'instance ne peuvent pas changer
- elles doivent être initialisées de cette manière

Pointeur vs. pointé

Qu'est-ce qui est constant ?

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

- la valeur du **pointeur** (= sur quelle chaîne HOST pointe) ?
- ou la valeur de **ce qu'il pointe** (= le contenu de la chaîne "localhost") ?

Par exemple

- peut-on ensuite écrire ?

```
HOST = "www.telecom-paristech.fr";  
HOST[0] = 'x';
```

Pointeur vs. pointé

const porte sur "ce qui suit"

- const avant * => ce qui est pointé est constant :

```
const char * HOST = "localhost";  
HOST = "www.telecom-paristech.fr"; // OK  
HOST[0] = 'x'; // ERREUR (de compilation)
```

- const après * => le pointeur est constant :

```
char * const HOST = "localhost";  
HOST = "www.telecom-paristech.fr"; // ERREUR
```

- tout est constant :

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

Note: les littéraux doivent être constants

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

- interdire de changer le contenu du littéral "localhost" (risque de plantage !)

Méthodes retournant un pointeur

```
class User {  
    char * name;  
public:  
    User(const * _name) : name(_name) {}  
    char* getName() {return name;} // !!! DANGER !!!  
};  
  
User fred("fred");  
char* s = fred.getName();  
s[0] = 'x'; // modifie 'name' à l'insu de fred !
```

Problème propre aux pointeurs:

- l'affectation: `s = fred.getName()`
 - copie les pointeurs, pas les objets pointés => 's' et 'name' pointent sur la même chose
- ceci rompt le principe d'encapsulation

Méthodes retournant un pointeur

```
class User {
    char * name;
public:
    User(const * _name) : name(_name) {}
    char* getName() {return name;} // !!! DANGER !!!
};

User fred("fred");
char* s = fred.getName();
s[0] = 'x'; // modifie 'name' à l'insu de fred !
```

Solution

```
class User {
    .....
    const char* getName() const {return name;}
};

User fred("fred");
const char* s = fred.getName(); // const a droite => const à gauche
s[0] = 'x'; // INTERDIT (erreur de compilation)
```

en Java :

même problème avec les références, c'est pourquoi les String sont "immuables"

Méthodes "constantes"

const appliqué à une méthode

- spécifie que la méthode **ne modifie pas** l'objet
- permet d'appeler cette méthode **si l'objet est constant**

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

Exemple

```
const Square s1(0, 0, 100);
const Square * s2 = new Square(50, 50, 300);

cout << s1.getX() << endl; // OK: getX() est const
s2->setX(50); // ERREUR: setX() n'est pas const
```

Constance logique et constance physique

```
class Doc {
    string text;
    Printer* printer;           // ressource interne

public:
    Doc() : printer(NULL) {}

    void print() const {
        if (printer==NULL) printer = createPrinter(); // ERREUR!
    }
    ....
};
```

Point de vue du client

- print() ne modifie pas le document => methode const

Point de vue de l'implémentation

- print() doit allouer une ressource interne => ne peut pas être const

Contradiction... et erreur de compil !

Constance logique et constance physique

```
class Doc {
    string text;
    mutable Printer* printer; // ressource interne mutable

public:
    Doc() : printer(NULL) {}

    void print() const {
        if (printer==NULL) printer = createPrinter(); // OK
    }
    ....
};
```

Point de vue du client

- print() ne modifie pas le document : constance logique

Point de vue de l'implémentation

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

Solution : mutable

Chapitre 5 : bases des templates et de la STL

Templates = programmation générique

- les types sont des paramètres
- base de la STL (Standard Template Library)

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

int i    = mymax(4, 10);
double x = mymax(6666., 77777.);
float f  = mymax<float>(66., 77.);
```

- *NB: attention: max() existe en standard !*

Templates (2)

Classes templates

```
template <class T>
class vector {
    vector()                { ... }
    void add(T elem)       { ... }
    void add(T elem, int pos) { ... }
    void remove(int pos)   { ... }
};
```

```
template <class T>
void sort(vector<T> v) {
    ....
}
```

```
vector<int> v;
v.add(235);
v.add(1);
v.add(14);
sort(v);
```

Standard Template Library (STL)

Conteneurs

- classes qui contiennent des objets
- gestion automatique de la mémoire

```
vector<int> v(3);      // vecteur de 3 entiers
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];
```

Les plus courants

- vector, list, map

Mais aussi

- deque, queue, stack, set, bitset

STL (2)

Algorithmes

- manipulent les données des conteneurs
- génériques

```
reverse( v.begin(), v.end() );
```

Itérateurs

- sortes de pointeurs généralisés
- exemple: `v.begin()` et `v.end()`

```
reverse(v.begin(), v.end());
```

Documentation

- www.cppreference.com ou www.sgi.com/tech/stl

Exemple de vecteur

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

struct Point {                // struct = class + public
    int x, y;
    Point() : x(0), y(0) {}
    Point(int _x, int _y) : x(_x), y(_y) {}
};

vector<Point> points;        // vecteurs de Points

points.push_back( Point(20, 20) );
points.push_back( Point(50, 50) );
points.push_back( Point(70, 70) );

for (unsigned int i=1; i < points.size(); i++)
    drawLine(points[i-1].x, points[i-1].y, points[i].x, points[i].y);

points.clear();            // vide le vecteur
```

"points" est un vecteur d'objets

- accès direct aux éléments via `[]` ou `at()` (`at()` vérifie la validité de l'index)
- coût d'insertion / suppression élevé

Exemple de liste

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

list<Point *> plist;        // liste de pointeurs

plist.push_back( new Point(20, 20) );
plist.push_back( new Point(50, 50) );
plist.push_back( new Point(70, 70) );

for (list<Point*>::iterator it = plist.begin();
     it != plist.end(); ++it)
{
    (*it)->draw();        // ( ) nécessaires car -> est plus prioritaire que *
}
```

"plist" est une liste de pointeurs d'objets

- pas d'accès direct aux éléments
- coût d'insertion / suppression faible
- la liste est doublement chaînée

Deux problèmes ...

```
void drawAll(list<Point*> plist) {
    for (list<Point*>::iterator it = plist.begin();
        it != plist.end(); ++it)
        (*it)->draw();
}

void foo() {
    list<Point*> plist;

    plist.push_back( new Point(20, 20) );
    plist.push_back( new Point(50, 50) );
    plist.push_back( new Point(70, 70) );

    drawAll(plist);    // PBM 1
} // PBM 2
```

Deux problèmes ...

```
void drawAll(list<Point*> plist) {
    for (list<Point*>::iterator it = plist.begin(); it != plist.end(); ++it)
        (*it)->draw();
}

void foo() {
    list<Point*> plist;

    plist.push_back( new Point(20, 20) );
    plist.push_back( new Point(50, 50) );
    plist.push_back( new Point(70, 70) );

    drawAll(plist);    // PBM 1
} // PBM 2
```

Pbm 1 : La liste est recopiée inutilement (à suivre...)

Pbm 2 : Les objets pointés ne sont pas détruits !!!

- plist est dans la pile => automatiquement détruite mais pas les objets créés par new !

- 1ere solution :

```
for (list<Point*>::iterator it = plist.begin(); it != plist.end(); ++it)
    delete *it;
```

- 2eme solution : smart pointeurs (à suivre...)

Enlever des éléments de `std::list`

Enlever à une position donnée

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

Enlever un élément donné

- void `remove` (const T& value);
- template < class Predicate > void `remove_if` (Predicate pred)

Détruire des éléments tout en parcourant la liste

- Problème : l'itérateur `k` est invalide après `erase()` d'où l'utilité de `k2`
- Remarque : l'objet pointé `*k` est détruit par `delete`

```
typedef std::list<Point*> PointList;

PointList plist;
int val = 200;

for (PointList::iterator k = plist.begin(); k != plist.end(); ) {

    if ((*k)->x != val)
        k++;
    else {
        PointList::iterator k2 = k;
        k2++;
        delete *k;
        plist.erase(k);
        k = k2;
    }
}
```

Table associative (map)

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

class User {
public:
    User(const string& prenom, const string& nom, int id);
    int getID() const;
    ....
};

typedef map<string, User*> UserMap;
UserMap dico; // équivaut à: map<string, User*> dico;

// ajout :
dico["Jean Dupont"] = new User("Jean", "Dupont", 314);
dico["Albert Einstein"] = new User("Albert", "Einstein", 666);

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

- Remarque : si User n'a pas de sous-classe on peut aussi utiliser : `map<string, User>`
 - ce qui simplifie la gestion mémoire (pas de new, pas de delete)

Exemple d'utilisation d'un "algorithme"

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

class Entry {
    string name;
    friend bool compareEntries(const Entry*, const Entry*);
public:
    Entry(const string& n) : name(n) {}
    ....
};

// NB: inline nécessaire si la définition est dans un header
inline bool compareEntries(const Entry* e1, const Entry* e2) {
    return e1->name < e2->name;
}

vector<Entry*> entries;
.....

std::sort( entries.begin(), entries.end(), compareEntries)
```

Chapitre 6 : Passage par valeur et par référence

Passage par valeur

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    ....
};

void MySocket::send(int i) {
    // envoie i sur la socket
}

void foo() {
    MySocket sock("infres", 6666);
    int a = 5;
    sock.send(a);
}
```

- Quelle est la relation entre l'argument `a` et le paramètre `i` ?

Passage par valeur

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    ....
};

void MySocket::send(int i) {
    // envoie i sur la socket
}

void foo() {
    MySocket sock("infres", 6666);
    int a = 5;
    sock.send(a);    // arg a copie dans param i
}
```

- la valeur de l'argument est **recopiée** dans le paramètre de la fonction
 - **sauf** pour les tableaux (l'**adresse** du 1er élément est copiée)
- cas par défaut pour C++ et C# (seule possibilité pour C et Java)

Comment récupérer une valeur ?

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    void receive(int i);
    ....
};

void MySocket::receive(int i) {
    // recupere i depuis la socket
    i = ...;
}

void foo() {
    MySocket sock("infres", 6666);
    int a;
    sock.receive(a);
}
```

Que se passe t'il ?

Passage par référence

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    void receive(int& i);
    ....
};

void MySocket::receive(int& i) {
    i = ...; // recupere i depuis la socket
}

void foo() {
    MySocket sock("infres", 6666);
    int a;
    sock.receive(a);
}
```

Passage par référence

- **pas** de copie : l'argument a et le paramètre i référencent la **même** entité
 - i est un "alias" de a => a est bien modifié au retour de la fonction
 - Attention: **PAS** de passage par référence en Java (contrairement aux apparences) !

Cas des "gros arguments"

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    void receive(int& i);
    void send(string s);
    ....
};

void MySocket::send(string s) {
    // envoie s sur la socket
}

void foo() {
    MySocket sock("infres", 6666);
    string a = "une chaine tres tres tres tres longue....";
    sock.send(a);
}
```

Quel est le probleme ?

Cas des "gros arguments"

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    void receive(int& i);
    void send(string s);
    ....
};

void MySocket::send(string s) {
    // envoie s sur la socket
}

void foo() {
    MySocket sock("infres", 6666);
    string a = "une chaine tres tres tres tres longue....";
    sock.send(a);
}
```

Problèmes

- 1. le contenu de a est recopié inutilement dans s (temps perdu !)
- 2. copie pas souhaitable dans certains cas
 - exemple: noeuds d'un graphe pointant les uns sur les autres

1ere tentative

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    void receive(int& i);
    void send(string& s);
    ....
};

void MySocket::send(string& s) {
    // envoie s sur la socket
}

void foo() {
    MySocket sock("infres", 6666);
    string a = "une chaine tres tres tres tres longue....";
    sock.send(a);
}
```

Pas satisfaisant

- avantage : `a` n'est plus recopié inutilement dans `s`
- inconvénient : `send()` pourrait modifier `a` (ce qui n'a pas de sens)
- amélioration ... ?

Passage par const référence

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);
    void receive(int& i);
    void send(const string& s);    // const reference
    ....
};

void MySocket::send(const string& s) {
    // envoie s sur la socket
}

void foo() {
    MySocket sock("infres", 6666);
    string a = "une chaine tres tres tres tres longue....";
    sock.send(a);
}
```

Passage par référence en lecture seule

- `a` n'est plus recopié inutilement dans `s`
- `send()` ne peut pas modifier `a` ni `s`

Synthèse

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void send(int i);           // par valeur
    void send(const string& s); // par const référence
    void receive(int& i);       // par référence
};
```

- Passage par **valeur**
 - argument **recopié** => **pas** modifié
- Passage par **const référence**
 - argument **pas** recopié, **pas** modifié
 - alternative au cas précédent (gros arguments ou qu'il ne faut pas copier)
- Passage par **référence**
 - argument **pas** recopié, **peut** être modifié
 - cas où on veut récupérer une valeur

Valeur de retour des fonctions

Mêmes règles que pour les paramètres

```
class User {
    string name;

public:
    User(const string& n) : name(n) {}

    const string& getName() const {return name;} // retourne name
    string getNameBAD() const {return name;}     // retourne une copie de name
};

int main() {
    string zname = "Zorglub";
    User z(zname);

    string n1 = z.getName(); // OK: copie 'name' dans n1
    string n2 = z.getNameBAD(); // double copie !
}
```

- getNameBAD() fait une recopie intermédiaire qui ne sert à rien (dans ce cas)

Remarque

Conversions implicites des `const` références

```
class User {
    string name;
public:
    User(string string& n) : name(n) {}
};

int main() {
    User z("Zorgub");    // CORRECT
}
```

- "Zorglub" n'est pas de type `string` (son type est `char *`)
- "Zorglub" est **implicitement** convertie en `string` car constructeur :

```
string::string(const char*);
```

Rappel

Opérateurs d'initialisation et d'affectation

```
Dessin(const Dessin& d2);           // Dessin d = d2;
Dessin& operator=(const Dessin& d2); // d = d2;
```

- `d2` pas copié et pas modifiable

Comparaison avec C et Java

```
class MySocket {
public:
    MySocket(const char* host, int port);
    void receive(int& i);
    ....
};

void MySocket::receive(int& i) {
    // recupere i depuis la socket
    i = ...;
}

void foo() {
    MySocket sock("infres", 6666);
    int a;
    sock.receive(a);
}
```

Pas de passage par référence en C ni en Java : comment faire ?

Comparaison avec C

Pointeurs: solution équivalente mais plus compliquée

```
class MySocket {
public:
    void receive(int& i);           // C++ référence (ref en C#)
    void receive(int* pi);         // C++ ou C : pointeur
};

void MySocket::receive(int& i) {   // i est un alias de a
    i = ...;
}

void MySocket::receive(int* pi) {  // pi pointe sur a
    *pi = ...; // noter l'*
}

void foo() {
    MySocket sock("infres", 6666);
    int a;
    sock.receive(a);               // C++ ou C#
    sock.receive(&a);              // C++ ou C : adresse de a
}
```

Passage par pointeur

- passage par valeur de l'adresse de `a` recopiée dans le pointeur `pi`
- seule possibilité en C (possible en C++ mais préférer les références)

Comparaison avec Java : Types de base

```
class MySocket {
public:
    void receive(int??? i);
};

void foo() {
    MySocket sock = new MySocket("infres", 6666); // Java
    int a;
    sock.receive(a); // Passage par VALEUR: a est recopié
}
```

En Java

- PAS de passage par référence au sens de C++, C#, Pascal ...
- PAS de pointeurs au sens de C ou C++

=> Pas d'équivalent pour les types de base !

Comparaison avec Java : Objets

```
class MySocket {
    public void receive(String s) {
        s = new String("valeur recue"); // il faudrait mettre la valeur recue du socket
    }
}

void foo() {
    MySocket sock = new MySocket("infres", 6666);
    String buf = new String("mon buffer");
    sock.receive(buf);
}
```

buf pointe vers quoi après l'appel ?

- vers "mon buffer" ou "valeur recue" ?

Comparaison avec Java : Objets (Suite)

```
class MySocket {
    public void receive(String s) {
        // ici s pointe sur "mon buffer"
        s = new String("valeur recue");
        // ici s pointe sur "valeur recue"
    }
}

void foo() {
    MySocket sock = new MySocket("infres", 6666);
    String buf = new String("mon buffer");
    sock.receive(buf);
    // ici buf pointe sur "mon buffer"
}
```

Java : passage par valeur des références (= par pointeur)

- la référence (= pointeur) **buf** est copiée dans la référence **s**
 - mais l'objet pointé n'est pas copié
- **s** n'est pas un alias : ce n'est **PAS** du passage par référence !

Comparaison avec Java : Objets (Solution)

```
class MySocket {
    public void receive(StringBuffer s) {
        // s pointe sur le même StringBuffer que buf
        s.append("valeur recue");
    }
}

void foo() {
    MySocket sock = new MySocket("infres", 6666);
    StringBuffer buf = new StringBuffer();
    sock.receive(buf);
}
```

Solution

- modifier le contenu de l'**objet pointé**
- mais pas le pointeur !

Préferer 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)
```

Cas des conteneurs de la STL

```
void drawAll(list<Point*> pl)
{
    for (list<Point*>::iterator it = pl.begin();
        it != pl.end(); ++it)
        (*it)->draw();
}

void foo() {
    list<Point*> plist;

    plist( new Point(20, 20) );
    plist( new Point(50, 50) );
    plist( new Point(70, 70) );

    drawAll(plist);
}
```

Quel est le problème ?

Cas des conteneurs de la STL (2)

```
void drawAll(list<Point*> pl)
{
    for (list<Point*>::iterator it = pl.begin(); it != pl.end(); ++it)
        (*it)->draw();
}

void foo() {
    list<Point*> plist;

    plist( new Point(20, 20) );
    plist( new Point(50, 50) );
    plist( new Point(70, 70) );

    drawAll(plist);
}
```

Passage par valeur

- `plist` est copiée dans `pl` : opération coûteuse si la liste est longue !
- noter que la liste est copiée mais pas les objets pointés

Cas des conteneurs de la STL (3)

```
void drawAll(const list<Point*> & pl)
{
    for (list<Point*>::const_iterator it = pl.begin(); it != pl.end(); ++it)
        (*it)->draw();
}

void foo() {
    list<Point*> plist;

    plist( new Point(20, 20) );
    plist( new Point(50, 50) );
    plist( new Point(70, 70) );

    drawAll(plist);
}
```

Passer les conteneurs par référence ou const référence

- pour éviter de les recopier inutilement
- noter `const_iterator` : itérateur qui ne modifie pas la liste

Chapitre 7 : Surcharge des opérateurs et Smart Pointers

Surcharge des opérateurs

```
#include <string>

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

- `string` est une classe "normale"
- mais les opérateurs `+` et `+=` sont redéfinis

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

Surcharge des opérateurs

Possible pour presque tous les opérateurs

- `== <> + - * / ++ -- += -= -> () [] new delete`
- mais pas pour: `::..*` ?
- la priorité est inchangée

A utiliser avec discernement

- peut rendre le code incompréhensible !

Existe dans de nombreux langages (C#, Python, Ada...)

- mais pas en Java

Cas (particulièrement) intéressants

operator[]

```
template <class T> vector {
    int& operator[](int i) {...}
    ....
};

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

operator()

- "Objets fonctionnels" : le même algorithme peut s'appliquer à des fonctions ou à des objets

operator++

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

operator new , delete , new[], delete[]

- redéfinition de l'allocation mémoire

conversions de types

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

```
};
```

Smart Pointers, comptage de références

Principe

- **compter** le nombre de (smart) pointers qui référencent l'objet
- **détruire** l'objet quand le compteur arrive à 0

```
SharedPtr<Circle> p1 = new Circle(0, 0, 50);    // refcount=1
SharedPtr<Circle> p2;

p2 = p1;    // p2 pointe aussi sur l'objet => refcount=2
p1 = NULL;  // p1 ne pointe plus sur l'objet => refcount=1
p2 = NULL;  // refcount=0 => destruction automatique de l'objet
```

Avantage

- mémoire gérée automatiquement : plus de **delete** !

Smart Pointers "Intrusifs"

Principe

- l'objet pointé possède un **compteur de références**
- les smart pointers **détectent** les affectations et modifient le compteur

Exemple

```
class Shape { // classe de base (Circle dérive de Shape)
    long refcount;
public:
    Shape() : refcount(0) {}
    void addRef() {++refcount;}
    void remRef() {if (--refcount == 0) delete this;} // hara kiri à 0 !
    void setX(int x);
    .....
};

void foo() {
    smptr<Shape> p = new Circle(0, 0, 50); // smart pointer
    p->setX(20);

    vector< smptr<Shape> > vect; // vecteur de smart pointers
    vect.push_back( new Circle(0, 0, 50) );
    vect[0]->setX(20);

} // destruction des variables locales p et vect et de ce qu'elles pointent
```

Ou est la magie ?

Les smart pointers sont des objets qui :

- encapsulent un pointeur standard (raw pointer)
- surchargent le copy constructor et l'opérateur =
- surchargent les opérateurs de déréférencement -> et *

```
template <class T> class smptr {
    T* p;
public:
    smptr(T* obj) : p(obj) {if (p != NULL) p->addRef();}

    ~smptr() {if (p != NULL) p->remRef();}

    smptr& operator=(T* obj) {...}

    .....

    T& operator*() const {return *p;}

    T* operator->() const {return p;} // sptr->setX(20) fait
                                    // sptr.p->setX(20)
};

void foo() {
    smptr<Shape> p = new Circle(0, 0, 50);
    p->setX(20);
}
```

Implémentations et limitations

Il existe plusieurs implémentations

- Smart pointers "intrusifs" (intrusive_ptr)
 - imposent d'avoir un compteur dans l'objet
- Smart pointers "non intrusifs" (shared_ptr)
 - lèvent cette restriction mais incompatibles avec pointeurs standard
- Smart pointers sans comptage de référence (scoped_ptr)
 - un seul pointeur par objet
- Voir [smart pointers de Boost](#) et implémentation donnée en TP

Attention

- ne marchent pas si dépendances circulaires
- rajouter des verrous s'il y a des threads

Exemple d'implémentation

```
template <class T>
class smptr {
    T* p;
public:
    smptr(T* obj = 0) : p(obj) { // smptr<Circle> ptr = object;
        if (p != 0) p->addRef();
    }

    smptr(const smptr& ptr) : p(ptr.p) { // smptr<Circle> ptr = ptr2;
        if (p != 0) p->addRef();
    }

    ~smptr() { // destructeur
        if (p != 0) s->remRef();
    }

    smptr& operator=(T* obj) { // ptr = object;
        if (p != 0) p->remRef();
        p = obj;
        if (p != 0) p->addRef();
    }

    smptr& operator=(const smptr& ptr) { // ptr = ptr2;
        if (p != 0) p->remRef();
        p = ptr.p;
        if (p != 0) p->addRef();
    }

    T* operator->() const {return p;} // ptr->setX(20) fait
                                    // ptr.operator->setX(20)

    T& operator*() const {return *p;}
};
```

Chapitre 8 : Compléments sur les types

transtypage, typage dynamique, types incomplets, RTTI, pointeurs de fonctions et méthodes

Transtypage vers les super-classes

```
class Object {
    ...
};

class Button : public Object {
    ...
};

Object* obj = new Object();
Button* but = new Button();

obj = but;           // correct?
but = obj;           // ???
```

Transtypage vers les super-classes

```
class Object {
    ...
};

class Button : public Object {
    ...
};

Object* obj = new Object();
Button* but = new Button();

obj = but;           // OK: transtypage implicite
but = obj;          // ERREUR de compilation (pareil en Java)
```

Transtypage implicite vers les super-classes ("upcasting")

Transtypage vers les sous-classes

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

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

Object* obj = new Button();

//... ailleurs dans le programme on doit dessiner l'objet
obj->draw();           // correct?
```

Transtypage vers les sous-classes

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

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

Object* obj = new Button();

//... ailleurs dans le programme on doit dessiner l'objet
obj->draw();      // ERREUR: draw() n'est pas une methode de Object
```

Que faire ?

Une solution qui a ses limites

```
class Object {
    virtual void draw() = 0; // rajouter draw() dans la classe de base
};

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

Object* obj = new Button();

//... ailleurs ...

obj->draw();      // COMPILER: draw() est une methode de Object
```

Problèmes

- "Object" peut ne pas être modifiable (exple: classe d'une librairie)
- "Object" finit par contenir tout et n'importe quoi !

Une mauvaise solution

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

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

Object* obj = new Button();

//... ailleurs ...

Button* but = (Button*)obj;    // DANGEREUX !!!
but->draw();
```

Pourquoi ?

Une mauvaise solution

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

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

Object* obj = new Button();

//... ailleurs ...

Button* but = (Button*)obj;    // DANGEREUX !!!
but->draw();
```

Et si on se trompe ?

- comment être sûr que obj pointe sur un Button ? => ne **JAMAIS** utiliser le "cast" du langage C
- et en Java ? Attraper les exceptions !

Bonne solution: Transtypage dynamique

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

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

Object* obj = new Button();

//... ailleurs ...

Button* but = dynamic_cast<Button*>(obj);

if (but != NULL) {
    but->draw();           // obj pointait sur un Button
} else {
    cerr << "Not a Button!" << endl;   // mais pas dans ce cas !
}
```

Contrôle dynamique du type à l'exécution

- => pas de risque d'erreur

Typage statique et typage dynamique

Typage statique

- cas de Java, C++, C#... : les objets sont fortement typés
- exceptionnellement : transtypage dynamique (dynamic_cast)

Typage dynamique

- le type des objets est généralement déterminé à l'exécution

```
@interface Object {           // classe en Objective-C
    // pas de methode draw()
}
@end

@interface Button : Object {
}
+ (void)draw;
@end

Object* obj = [[Button alloc] init];

[obj draw];   // COMPIL OK : on envoie le message draw a obj
              // obj "décide": si obj est un Button draw est executé
```

Autres operateurs de transtypage

static_cast

```
Button* but = static_cast<Button*>(obj);
```

- similaire au cast du C mais detecte quelques absurdites
- à éviter (pas de contrôle à l'exécution)

reinterpret_cast

- meme chose en pire

const_cast

- pour enlever ou rajouter `const` au type

RTTI

Accès dynamique au type d'un objet

```
#include <typeinfo>

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

Principales méthodes de `type_info`

- `name()` retourne le nom de la classe (sous forme encodee)
- `opérateur ==` pour comparer 2 types

RTTI (2)

Ce qu'il ne faut pas faire

```
void drawShape(Shape *p)
{
    if (typeid(*p) == typeid(Rect))
        p->Rect::draw();

    else if (typeid(*p) == typeid(Square))
        p->Square::draw();

    else if (typeid(*p) == typeid(Circle))
        p->Circle::draw();
}
```

Utiliser le polymorphisme (liaison dynamique)

```
class Shape {
    ....
    virtual void draw() const; // éventuellement abstraite (= 0)
    ....
}
```

Types incomplets et handle classes

```
// header Button.h

class Button {
public:
    Button();
    void repaint(Rect &);
    ...
private:
    ButtonImpl * impl;
};
```

Références croisées

- la méthode `repaint()` dépend d'une classe `Rect` déclarée ailleurs
 - à ce stade on n'a pas besoin de savoir ce que fait `Rect`

Cacher l'implémentation

- les variables et méthodes internes sont cachées dans `ButtonImpl`
- `ButtonImpl` est déclarée dans un header privé `ButtonImpl.h` (pas donné au client de la librairie)
- `Button` est une "handle class"

Problème ?

Types incomplets : problème

```
// header Button.h

class Button {
public:
    Button();
    void repaint(Rect &);
    ...
private:
    ButtonImpl * impl;
};
```

Problème :

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

Solution ?

Types incomplets : mauvaise solution

```
// header Button.h

#include "Rect.h"
#include "ButtonImpl.h"

class Button {
public:
    Button();
    void repaint(Rect &);
    ...
private:
    ButtonImpl * impl;
};
```

Références croisées

- sac de noeuds : les headers vont tous s'inclure les uns les autres !

Cacher l'implémentation

- c'est raté : il faut maintenant donner `ButtonImpl.h` au client !

Types incomplets : bonne solution

```
// header Button.h

class Button {
public:
    Button();
    void repaint(class Rect &);    // ne pas oublier "class"
    ...
private:
    class ButtonImpl * impl;      // idem
};
```

Cette syntaxe

- permet de spécifier qu'**qu'une classe existe** sans avoir à la déclarer
- n'est valide que pour les **pointeurs** et les **références**
- est valide en **langage C** pour les pointeurs sur les **struct**

Pointeurs de fonctions et de méthodes

```
class Integer {
    bool isSup(const Integer&);
    bool isInf(const Integer&);
    ...
};

Integer a(5), b(10);
bool test1 = a.isSup(b);

bool (Integer::*f)(const Integer&);

f = &Integer::isSup;

bool test2 = (a.f)(b);
```

Chapitre 9 : Traitement des erreurs

Exceptions

But : faciliter le traitement des erreurs

- permettent de "remonter dans la pile" des appels des fonctions
- jusqu'à un (ou des) endroit(s) bien défini(s)

Avantages

- gestion plus claire, plus centralisée, plus homogène des erreurs
- que les enchaînements de fonctions retournant des codes d'erreurs
 - impliquant une gestion des erreurs souvent déficiente car trop complexe

Exceptions : exemple

```
class MathErr {};          // cf. complément page suivante

class Overflow : public MathErr {};

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

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

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

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

Exceptions : types et organisation

Types

- en Java `throw` doit envoyer un **objet** dérivant de la classe `Exception`
- en C++ `throw` peut envoyer ce qu'on veut (objet, entier, chaîne de caractères...)

En pratique

- en C++ les exceptions sont généralement des classes, comme en Java :
 - organisées en hiérarchies (l'héritage multiple est permis)
 - dérivant de la classe `exception` (mais ce n'est pas obligatoire)

```
#include <exception>

class MathErr : std::exception {};

...etc...
```

Spécification d'exceptions

Principale différence C++ / Java

- en **Java** les méthodes **doivent spécifier** les exceptions qu'elles peuvent envoyer
- mais **pas** en **C++** ni en **C#** (c'était optionnel en C++, c'est maintenant obsolète)

```
// Java
public int divide(int x, int y) throws Zerodivide, Overflow {...} // NB: throws

// C++
int divide(int x, int y); // OK

int divide(int x, int y) throw (Zerodivide, Overflow); // OK mais obsolete
```

Spécification d'exceptions

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

// C++
int divide(int x, int y);
```

Avantages et inconvénients

- les spécifications d'exceptions permettent un meilleur contrôle
- mais elle réduisent la puissance de l'héritage :
 - une méthode redéfinie dans une sous-classe **ne peut pas** spécifier de nouvelles exceptions
- ce qui amène à des complications ou des acrobaties...
 - => choix différents selon les langages

Compléments sur les exceptions

Exceptions standard

- `bad_alloc`, `bad_cast`, `bad_typeid`, `bad_exception`, `out_of_range`, etc.

Handlers

- `std::set_terminate()` et `std::set_unexpected()` dans `<exception>`

Redéclenchement

```
try {
    ..etc..
}

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

- in fine, la fonction `std::terminate` est appelée

Chapitre 10 : Héritage multiple

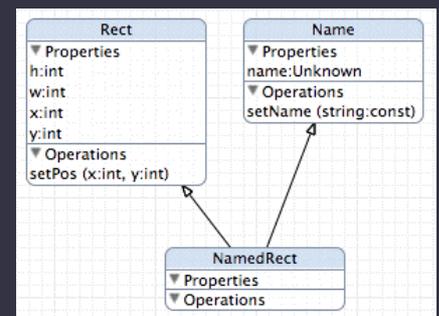
Bases de l'héritage multiple

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

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

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

- `NamedRect` hérite des variables et méthodes des 2 superclasses



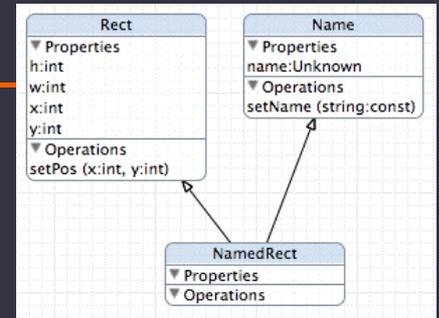
Constructeurs

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

class Name {
    std::string name;
public:
    Name(const std::string&);
    ...
};

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

- respecter l'ordre d'appel des constructeurs



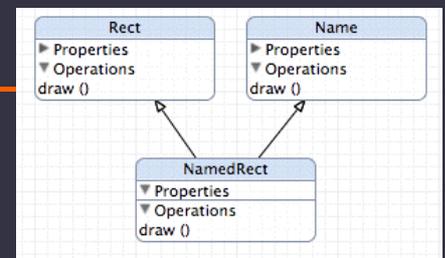
Ambiguïtés

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

class Name {
    int x, y;
public:
    virtual void draw();
};

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

- redéfinition de `NamedRect::draw()` pas obligatoire mais préférable
- même principe pour variables



"using"

```
class A {
public:
    int  foo(int);
    char foo(char);
};

class B {
public:
    double foo(double);
};

class AB : public A, public B {
public:
    using A::foo;
    using B::foo;
    char foo(char);    // redefinit A::foo(char)
};

AB ab;
ab.foo(1);           // A::foo(int)
ab.foo('a');         // AB::foo(char)
ab.foo(2.);          // B::foo(double)
```

- étend la résolution de la surcharge aux sous-classes

Duplication de bases

```
class Shape {
    int x, y;
};

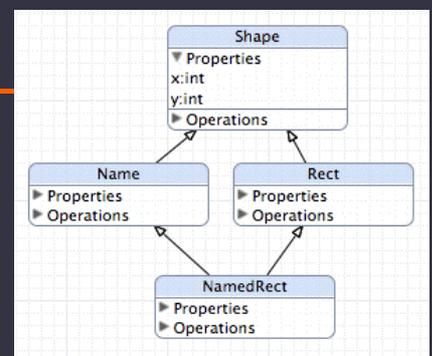
class Rect : public Shape {
    // ...
};

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

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

- la classe Shape est **dupliquée** dans NameRect
- même principe pour accéder aux méthodes et variables

```
float m = (Rect::x + Name::x) / 2.;
```



Bases virtuelles

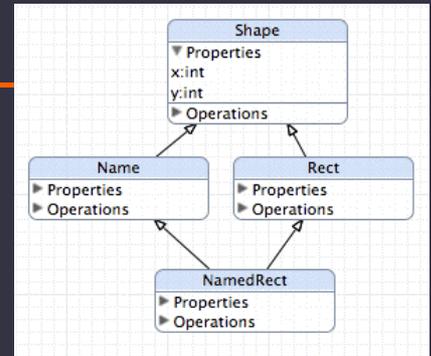
```
class Shape {
    int x, y;
};

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

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

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

- la classe Shape n'est **PAS** dupliquée dans NameRect
- attention: surcharge en traitement et espace mémoire
 - utilisation systématique découragée



Classes imbriquées (inner classes)

```
class NamedRect : public Rect {
    struct Name {          // classe imbriquée
        ...
    }; name
public:
    NamedRect(..etc..);
};
```

Technique de composition très utile

- souvent préférable à l'héritage multiple
 - car moins de dépendances dans le modèle des classes

Remarque

- pas d'accès aux champs de la classe imbriquante (!= Java)

Plus d'infos

- toutes les réponses aux questions possibles et impossibles : [C++ FAQ LITE](#)
- [le site de Boost C++](#)
- [un site intéressant sur les smart pointers](#)
- [un site traitant des garbage collectors en C++](#)