Programmation objet et autres concepts en C++

Eric Lecolinet

Télécom ParisTech

Octobre 2011

Sommaire

Liens et références

- Les TPs associés à ce cours
- Un petit tutoriel: de Java à C++
- Une intro au toolkit graphique Qt
- Le site de référence cplusplus.com et le manuel de la STL
- Deux autres sites interessants: C++ Reference et STL SGI
- La documentation automatique avec Doxygen
- Les extensions Boost
- La Méga Foire Aux Questions: C++ FAQ LITE
- Le "Méga 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-compilateur)
- 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 et Smalltalk ...)

C#

- inspiré de Java (et de C++ ...)

Brève historique (2)

Mais aussi...

Objective C (MacOSX, IOS)

- une autre extension objet du C
  - hybridation de C et Smalltalk
  - popuplaire grace à l'iPhone, etc.
  - syntaxe bizarre mais simple et puissant

Python, Ruby

  - visent la simplicité d'écriture et la flexibilité
  - interprétés et basés sur le typage dynamique (comme Objective C)
  - forte progression ces dernières années (au détriment de Java entre autres)

C++ versus C

Avantage : compatibilité C/C++

- même syntaxe de base
- code C "propre" compilable en C++
- 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 de même nature

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 compatibilité avec C)
- richesse du langage...

Moins bon côtés

- difficultés héritées du C
- richesse du langage...
  - parfois mal utilisée : programmes 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, Campus Press/Pearson Education
- www.cplusplus.com/reference
- www.cppreference.com

Liens utiles

- Travaux Pratiques (et d'autres liens utiles): 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

Versions de C++

- normalisation ANSI tardive => variations entre compilateurs !

Salles Sun à Télécom

- g++ version 4.*.*

Attention aux incompabilités !

- syntaxiques
- binaires: vérifier compatibilité entre le programme et les librairies utilisées

Programme C++

Un programme C++ est constitué :

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

Chaque fichier peut comprendre :

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

Pas de packages

- mais des namespaces

Premier chapitre

Des objets et des classes ...

Déclarations et définitions

Même distinction qu'en langage C :

- déclarations dans fichiers headers : xxx.h (ou .hpp ou .hh)
- définitions dans fichiers d'implémentation : xxx.cpp (ou .cc ou .C)
- règle standard : à chaque .cpp correspond un .h
  - qui déclare l'API publique
- éventuellement (pour les librairies) un second .h (typiquement: xxxImpl.h)
  - qui déclare l'API privée interne à la librairie
  - et n'est jamais fourni aux "clients" de la librairie

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 (ne pas oublier d'inclure le header !)

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

#include "Circle.h"

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                    // ne pas mettre main() dans Circle.cpp !

#include "Circle.h"                    // ne pas oublier!

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

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++ !

Gestion mémoire

- Java detruit 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();       // noter la ->
  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);              // instanciation
```

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) {
   x = _x;  y = _y;  radius = _r;
}

Circle::Circle(int _x, int _y, unsigned int _r)
   : 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 on ne définit aucun constructeur dans la classe

- on peut écrire :

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

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

Conseils

- toujours definir au moins un constructeur
- en général, c'est une bonne idée d'avoir 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;                  // signifie: c pointe sur aucun object
}
```

delete détruit un objet crée par new

- pas de garbage collector (ramasse miettes) comme en Java !
- mais ca existe en option (ou techniques alternatives, cf. Smart Pointers)

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;
```

Appelé 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 detruit l'objet (qui'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 detruire
- 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 methode finalize() joue le meme role que le destructeur, mais:
  - pas de chainage des "finaliseurs"
  - appel non deterministe 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 ambiguités :

```
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

```
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();
    ...
}
```

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

using namespace

```
namespace Geom {         // dans Circle.h
    class Circle {
       ...
    };

    class Rect {
       ...
    };
}


#include "Circle.h"      // dans main.cpp
using namespace Geom;

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

- spéficie un chemin d'accès par défaut
  - => évite d'avoir à prefixer les classes et les fonctions
- similaire à import en Java

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()     // necessite using...
  << "area= "   << c->getArea()
  << endl;

  std::cerr << "c = " << c  << std::endl;  // OK sans using...
}
```

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 4e niveau: package (par défaut), C++ a également friend

Encapsulation / droits d'accès (2)

```
class Circle {
// private:                        // private par defaut
   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:
   ...
};
```

- en Java struct n'existe pas, en C# ce n'est pas une classe

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 l'équivalent de

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

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

Inlines

Méthodes implémentées dans les headers

```
class Circle {
public:
  // inline implicite pour les methodes
  unsigned int getRadius() const {return radius;}
  ....
};

inline Circle* createCircle() {return new Circle();}
```

A utiliser avec discernement

- + : rapidité (theoriquement pas d'appel fonctionnel)
- +/- : lisibilité
- - : augmente taille du binaire généré
- - : contraire au principe d'encapsulation

Point d'entrée du programme

int main(int argc, char** argv)

- même syntaxe qu'en C
- arc : 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

Chapitre 2 : Héritage

Concept essentiel de l'OO

- héritage simple (comme Java)
- héritage multiple (à manier avec précaution !)

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 {   // héritage des variables et méthodes
public:
  Square();
  Square(int x, int y, unsigned int width);

  virtual void setWidth(unsigned int);    // redéfinition de 2 méthodes
  virtual void setHeight(unsigned int);
};
```

Exemple (implémentation)

```
class Rect {                // rappel des délarations
  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

Héritage 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 : implicite
- 2e cas : explicite == super() de Java

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() {
    ...
};
```

Transitivité des inclusions

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

Headers et inclusions multiples (2)

Empêcher les redéfinitions

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

  class Shape {
     ...
  };

#endif
```

- A faire systématiquement en C et en C++ pour tous les headers

Remarques

- #import : un #include intelligent (mais pas standard)
- les " " ou < > précisent l'espace de recherche (programme vs. librairies)

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

- 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 !

Laison 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

Méthodes virtuelles: redéfinition

```
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 virtuelles

- même si virtual est omis => virtual important dans les classes de base !

Elles doivent avoir la même signature

- sinon c'est une autre fonction (surcharge) !
- sauf que le type de retour peut être une sous-classe (covariance des types)

Méthodes virtuelles: surcharge

Il faut rédéfinir toutes les variantes

- **si** on redéfinit l'une d'entres-elles

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

- **pas** correct :

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

- correct :

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

Pourquoi des méthodes virtuelles ?

Programmation orientée objet

- 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#

- méthodes virtuelles par défaut

Pourquoi des méthodes NON virtuelles ?

A EVITER !

- principale raison: compatibilité avec le C

Cas particuliers

- optimiser l'exécution
  - accesseurs (getters, setters), et encore !
  - cas extrêmes (méthode appelée 10 000 000 fois...)
- redéfinir des méthodes avec des signatures différentes

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
- principe d'encapsulation
  - -> séparer la spécification et l'implémentation

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;
};


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

  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;}
}
```

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;
  }
}
```

Bénéfices du polymorphisme (1)

Gestion unifiée

- des classes dérivant de la classe abstraite
- sans avoir besoin de connaître leur type !
  - contrairement à la programmation "classique" (en C, etc.)

```
#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

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 imbriquee
       int x, y;
   public:
       Point(x, y);
   };

   Point p1, p2;              // variables d'instance
public:
   Rect(int x1, int y1, int x2, int y2);
};


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)
   : x(_x), y(_y) { }
```

Technique de composition très utile

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

Classes imbriquées (2)

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

   Point p1, p2;              // variables d'instance
public:
   Rect(int x1, int y1, int x2, int y2);
};
```

Visibilité des champs depuis la classe imbriquée

- les champs de Rect sont automatiquement visibles depuis Point en Java
- mais pas en C++ !

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 plus élévé (double indirection)

Chapitre 3 : Mémoire

Les différents types de mémoire

- mémoire statique (ou globale) : réservée dès la compilation, variables static
- pile (stack) : variables locales ("automatiques") 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)++;                   // NB: ne pas oublier les parenthèses!
}
```

- que valent count, i, *p si on appelle foo() deux fois ?
- **Remarque**: il existe un 4e type : la mémoire constante "read only"

Mémoire

Durée de vie

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

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
}
```

- 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++)

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
}
```

Utilisation de new et delete

- à chaque new doit correspondre un (et un seul) delete
- jamais de delete sur des objets statiques ou dans la pile (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

. 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() équivaut à (*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
    static Square* d;   // var. de classe qui pointe vers un objet
};
```

Durée de vie : même principe

- static (cas a et d) : même durée de vie que le programme
- sinon (cas b et c) : même durée de vie que l'objet contenant
- NB : l'objet pointé (cas c et d) est en mémoire dynamique (créé par new / détruit par delete)

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
     }
};
```

- on pourrait aussi écrire :

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

- il faut rajouter dans le fichier .cpp

```
Square Dessin::a(10, 20, 300);     // on ne repete pas "static"
```

Qu'est-ce qui manque ?

Destruction de l'objet

Il faut un destructeur !

- chaque fois qu'un constructeur fait 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;}
};
```

Remarques

- b pas créé avec new => pas de delete
- destructeurs généralement virtuels (pour 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);
    Dessin d2(10, 20, 300);

    d2 = d1;            // affectation

    Dessin d3 = d1;     // initialisation
    Dessin d4(d1);      // idem (syntaxe equivalente)
}
```

Quel est le probleme ?

- quand on sort de foo() ...

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);
    Dessin d2(10, 20, 300);
    d2 = d1;            // affectation
    Dessin d3 = d1;     // initialisation
    Dessin d4(d1);      // idem
}
```

Le contenu de d1 est copié dans d2, d3 et d4 !

- => toutes les variables c pointent sur la même instance de Square
- => cette instance est détruite 4 fois (BOUM!) quand on sort de foo (et les autres jamais)

Il faudrait de la copie profonde

Interdire la copie d'objets

La copie d'objets est dangereuse

- s'ils contiennent des pointeurs ou des références !

1e solution : interdire la copie

- déclarer privés l'opérateur d'initialisation (copy constructor) et d'affectation (operator=)
- implémentation inutile
- interdit également la copie pour les sous-classes

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

Solution similaire à Java

- qui ne permet pas de copier un objet dans un autre en utilisant =
  - (en Java = ne peut copier que des types de base ou des références)

Redéfinir la copie d'objets (copie profonde)

2e solution : redéfinir la copie

- l'opérateur d'initialisation et d'affectation font de la copie profonde

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


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 []
tab = 0;
```

Ne pas mélanger les opérateurs !

```
x = new       ->  delete x
x = new[]     ->  delete[] x
x = malloc()  ->  free(x)    // éviter malloc() et free()
```

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

Variables "const"

- ne peuvent pas changer de valeur
- doivent obligatoirement être initialisées

Exemples

- alternative aux #define :

```
const int MAX_ELEM = 200;
```

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

```
char* strcat(char* s1, const char* s2);
```

- les variables d'instance ne peuvent pas changer :

```
class User {
    const int id;
    const string name;      // name contient l'objet
public:
    // pas optimal (a suivre...)
    User(int i, string n) : id(i), name(n)) {}
};
```

Pointeurs et littéraux

Pointeurs : le const porte sur ce qui suit :

- l'objet pointé est constant

```
const char *s = "abcd";
*s = 'x';           // INTERDIT (par le compilateur)
s = "toto"          // OK
```

- le pointeur est constant

```
char* const s = "abcd"
*s = 'x';          // PERMIS (mais faux dans ce cas !!!)
s = "toto"         // INTERDIT
```

- pointeur et objet pointé constants

```
const char* const s = "abcd";
```

Attention aux littéraux (i.e. "abcd")

- souvent en mémoire "read-only" => plantage si on modifie leur contenu!
  - écrire : `const char* s = "abcd";` pour éviter les erreurs

Méthodes "const"

Les méthodes d'instance const

- ne modifient pas les variables d'instance
- mais sont applicables sur des objets const

```
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);             // INTERDIT! setX() n'est PAS const
```

Remarques

Un bon conseil !

- mettre les const DES LE DEBUT de l'écriture du programme
- changements pénibles a posteriori (modifs. en cascade...)

enum, une alternative a const pour les entiers

```
enum WEEK_END {SAMEDI=6, DIMANCHE=7};
```

Valeur de retour des fonctions

```
class User {
    char * name;          // chaine du C (a eviter!)
public:
    const char* getName() const {return name;}
    char* getName() {return name;}
};
```

- renvoie directement la chaîne stockée par l'objet sans faire de copie
- !!! la 2e version est DANGEREUSE (mais pas la 1ere) !!!

Conversion de constance

Exemple

- on veut interdire la modification d'un objet
- mais cet objet doit allouer une ressource à l'exécution

```
class DrawableSquare {   // s'affiche a l'ecran
    Peer* peer;

public:
    DrawableSquare() : peer(0) {}    // peer inconnu a ce stade
    void draw() const {if (!peer) peer = createPeer();}
};


void foo() {
    const DrawableSquare top_left(0,0,10);  // ne doit pas bouger;
    rect.draw();                            // OK: draw() est const
}
```

Probleme ?

Conversion de constance

```
class DrawableSquare {   // s'affiche a l'ecran
    Peer* peer;

public:
    DrawableSquare() : peer(0) {}
    void draw() const {if (!peer) peer = createPeer();}
};


void foo() {
    const DrawableSquare top_left(0,0,10);
    rect.draw();
}
```

Probleme

- draw( ) ne peut etre const car elle modifie 'peer' !

Solution ?

Cast et const_cast<>

(Très) mauvaise solution

```
void draw() const {         // et si on se trompe de type ?
    if (!peer)
        ((DrawableSquare*)this)->peer = createPeer();
}
```

(Moins) mauvaise solution

```
void draw() const {         // verifie que c'est le meme type
    if (!peer)
        const_cast<DrawableSquare*>(this)->peer = createPeer();
}
```

Risque de plantage dans les 2 cas !

- un objet const peut être stocké en mémoire "read-only"

Constance logique et constance physique

Bonne solution

```
class DrawableSquare {
    mutable Peer* peer;       // toujours modifiable !
public:
    void draw() const {if (!peer) peer = createPeer();}
};
```

- constance logique : point de vue du client
- constance physique : implémentation, inconnue du client

Chapitre 5 : 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 parametre 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 recopiée)
- cas par défaut pour C++ et C# (seule possibilité pour C et Java)

Comment recupérer 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 recopie : 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 reference 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. recopie 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 recopiée dans la référence s
  - mais l'objet pointé n'est pas recopié
- 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 !

Preferer 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 recopiée dans pl : opération coûteuse si la liste est longue !
- noter que la liste est recopiée 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 : iterateur qui ne modifie pas la liste

Chapitre 8 : 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 operateurs + et += sont redefinis

```
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 incomprehensible !

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

```
smptr<Circle> p1 = new Circle(0, 0, 50);     // refcount=1
smptr<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'operateur =
- 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'implementation

```
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;}
};
```

Doxygen

Système de documentation automatique

- similaire à JavaDoc

```
/** Classe de base des objets geometriques.
 * Noter que cette classe est abstraite
 * et ne peut donc pas être instanciee.
 */
class Shape {

  /// retourne la largeur.
  virtual unsigned int getWidth() const = 0;

  virtual unsigned int getWidth() const = 0;
  ///< retourne la hauteur.

  virtual void setPos(int x, int y) = 0;
  /**< change la position.
   * voir aussi setX() et setY().
   */
```

- www.doxygen.org

Chapitre 6 : Templates et 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 {
    int x, y;
    Point(int x, int y);
};

vector<Point> points;     // vecteurs de Points

Point p1(10, 25);
points.push_back(p1);

Point p2(10, 25);
points.push_back(p2);

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();
```

"points" est un vecteur d'objets

- accès direct aux éléments via [ ] ou at()
- coût d'insertion / suppression élévé

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();    // (*it) 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 éventuels ...

```
void drawAll(list<Point*> plist) {
   ... affiche tous les points
}

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
```

1) Passer les conteneurs par référence

```
void drawAll(const list<Point*> & plist)
{
  for (list<Point*>::const_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 corrigé

}  // PBM 2
```

Rappel: par défaut, les arguments sont passés par valeur

- donc recopiés (sauf les tableaux)
- => passer la list par référence pour éviter de la recopier inutilement

2) Détruire les objets pointés

```
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 2
```

Lorsqu'un conteneur est détruit

- il détruit ses éléments mais pas les objets pointés !
- faire :

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

Enlever des élements 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 élements tout en consultant 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;
  }
}
```

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 7 : Compléments

Doxygen, typage dynamique, RTTI, pointeurs...

Doxygen

Système de documentation automatique

- similaire à JavaDoc

```
/** Classe de base des objets geometriques.
 * Noter que cette classe est abstraite
 * et ne peut donc pas être instanciee.
 */
class Shape {

  /// retourne la largeur.
  virtual unsigned int getWidth() const = 0;

  virtual unsigned int getWidth() const = 0;
  ///< retourne la hauteur.

  virtual void setPos(int x, int y) = 0;
  /**< change la position.
   * voir aussi setX() et setY().
   */
```

- www.doxygen.org

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();    // COMPILE: 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;   // eventuellement abstraite (= 0)
    ....
}
```

Types incomplets

```
// header Circle.h

class Circle {            // handle class
  CircleImpl* impl;
public:
  Circle();
  void foo(Rect&);
};
```

Objectifs

- 1. cacher l'implementation de "Circle" dans "CircleImpl"
- 2. la methode "foo()" depend d'une classe definie ailleurs

Problème ?

Types incomplets

```
// header Circle.h

class Circle {
  CircleImpl* impl;
public:
  Circle();
  void foo(Rect&);
};
```

Problème :

- erreur de compilation: "CircleImpl" et "Rect" sont inconnus !

Solution ?

Types incomplets : mauvaise solution

```
// header Circle.h

#include "CircleImpl.h"
#include "Rect.h"

class Circle {
  CircleImpl* impl;
public:
  Circle();
  void foo(Rect&);
  ...
};
```

Inconvenients

- 1. l'implementation de "CircleImpl" devient visible !
- 2. "Circle.h" inclut "Rect.h" qui inclut peut-etre "Circle.h", etc...
  - => dépendances circulaires entre headers

Types incomplets : bonne solution

```
// header Circle.h

class Circle {
  class CircleImpl* impl;
public:
  Circle();
  void foo(class Rect&);
  ...
};
```

Ces déclarations partielles

- compilent sans connaître CircleImpl ni Rect
- propriété des pointeurs et références de classes

Interêt

- cacher l'implémentation (Handle classes)
- moins de dépendances entre headers

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

```
class MathErr {};

class Overflow : public MathErr {};

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

Organisation

- classes
- généralement regroupées en hiérarchies
- héritage multiple également possible

Exceptions

```
class MathErr {};

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 << "divise par 0" << endl; }
catch (MathErr)       { cerr << "erreur de calcul" << endl; }
catch (...)           { cerr << "erreur quelconque" << endl; }


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

Exceptions (2)

Spécifications d'exceptions

```
void foo() throw (Overflow, Zerodivide);   // que ces exceptions

void foo() throw ();   // aucune exception

void foo();            // toutes les exceptions
```

- throw indique les exceptions que la fonction peut lever
- throw est optionnel en C++
- mais pas en Java et le mot clé est throws !
- reféfinitions des méthodes : pareil ou moins d'exceptions

Exceptions (3)

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

Exceptions (4)

Exceptions standard

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

Handlers

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

NB: attention aux fuites mémoire !

- seules les variables dans la pile seront desallouées !
- -> prévoir traitement approprié pour les autres

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 herite des variables et methodes des 2 superclasses

Constructeurs

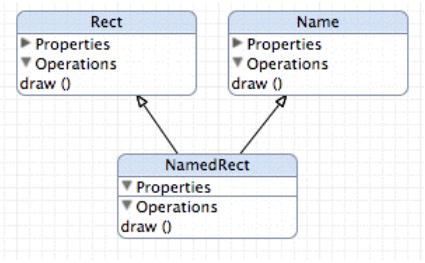
```
class Rect {
    int x, y, w, h;
public:
    Rect(int x, int y, int width, int height);
    ...
};

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

Ambiguités




```
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();
    }
}
```

- redfinition 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 imbriquee
        ...
    }; name

public:
    NamedRect(..etc..);
};
```

Technique de composition très utile

- souvent préférable à l'héritage multiple
  - car moins de dependances dans le modele 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++