Cours C++ eric lecolinet enst

Introduction au langage C++

Eric Lecolinet

Ecole Nationale Supérieure des Télécommunications (ENST/INFRES)

Janvier 2006

Contexte

Avant

C++ : extension objet du langage C (pré-compilateur de C a l'origine)

Bjarne Stroustrup, années 80

Après

Java inspiré de la partie objet de C++

Mais avec des differences importantes

Vision "anachronique"

C++ : sorte de mélange de C et de Java

C++ versus C

Principal avantage : compatibilité C/C++

même syntaxe de base

code C "propre" directement compilable en C++

facilité d'intégration de fichiers C++ et C dans un même programme

Principal 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 !)

Premières remarques

C++ = langage objet **ET** procédural

- contrairement à Java (purement orienté objet)
- -> langage riche et (relativement) complexe

C++ = meilleure **ET** pire des choses

- le meilleur : OO + efficacité du C
- le pire : richesse du langage souvent utilisée à mauvais escient
  - -> programmes inutilement complexes, "usines à gaz"

"Things should be made as simple as possible but not any simpler"

Prérequis et références

Connaissance préalable de C, Java, des concepts OO

cours non exhaustif !

porte essentiellement sur partie objet de C++

de C++ "presque Java" vers spécificités C/C++

Liens et références

travaux pratiques et liens utiles : http://www.enst.fr/~elc/cpp/TP.html

tutoriel et documentation du toolkit graphique Qt : http://www.enst.fr/~elc/qt

tutoriel: de Java à C++ http://www.enst.fr/~elc/C++

Le langage C++, Bjarne Stroustrup, Campus Press/Pearson Education

- écrit par l'auteur du langage, un peu "touffu" mais très complet

En pratique ...

Versions de C++

normalisation tardive (97/98...)

-> grandes variations entre compilateurs existants !

On utilisera

g++ (version 2.95 ou 3.3 : attention aux différences !)

alternative: compilateur CC de Sun avec Forte/Workshop (pas compatible!)

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.hh, xxx.hpp ...

définitions dans fichiers d'implémentation : xxx.cc, xxx.cpp ...

à chaque xxx.cc doit correspondre :

- un xxx.hh : déclaration de l'API "publique"
- (éventuellement) un xxx_impl.hh : types et données "privés"

Déclaration de classe

```
// fichier circle.hh : 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 float 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 {                                  // fichier circle.hh
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 float getArea() const;
};
```

Implémentation (ne pas oublier d'inclure le header !)

```
#include "circle.hh"                            // fichier circle.cc

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

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

unsigned float Circle::getArea() const {
  return 3.14 * radius * radius;
}
```

Instanciation

```
// fichier test.cc

#include "circle.hh"                    // ne pas oublier!

int main() {
  Circle* c = new Circle();             // creation d'un objet
  .....
}
```

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

même principe qu'en Java

allocation mémoire puis appel du constructeur

- constructeur par défaut: ne fait rien (variables pas initialisées !)

c est une variable locale qui pointe sur le nouvel objet

Instanciation (2)

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

Comparaison avec Java

Java : c serait une "référence" (au sens de Java !)

C++ : c est un pointeur (ne pas oublier l' *)

Java detruit les objets s'ils n'ont plus de référent (garbage collector)

! C++ nécessite une destruction explicite par l'opérateur delete

- si l'objet a été créé par new ...

Accès aux variables d'instance

```
#include "circle.hh"

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 VIs

noter l'utilisation de la -> (. en Java)

encapsulation -> restreindre l'accès aux VIs

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 float a = c2->getArea();
}
```

toujours appliquées à un objet

ont accès à toutes les variables de cet objet

propriété fondamentale de l'orienté objet !

```
// rappel de circle.cc

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

Constructeurs

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

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

Appel implicite à l'instanciation

permet d'initialiser les VIs (indéfinies sinon !)

Chaînage des constructeurs

appel implicite des constructeurs des super-classes

dans l'ordre descendant

Destructeur

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

Circle* c = new Circle(100, 200, 35);
delete c;                                  // destruction de l'objet
c = 0;                                     // c "pointe" sur null
```

Appel implicite à la destruction

pour libérer la mémoire (pointeurs), fermer les fichiers, sockets, etc.

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

Chaînage des destructeurs

dans l'ordre ascendant (inverse des constructeurs)

Surcharge (overloading)

Plusieurs méthodes :

ayant le même nom

mais des signatures différentes

pour une même classe

```
class Circle {
  Circle();                              // met les VIs à 0
  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

attention aux 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.hh
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

notion similaire aux variables "statiques" du C (d'où le mot-clé)

const (optionnel) indique que la valeur est constante

Définition des variables de classe

```
// déclaration: fichier circle.hh

class Circle {
public:
  static const float PI;
  ...
};

// définition: fichier circle.cc

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

Doivent être définies et initialisées dans un .cc

et dans un seul .cc !

ne pas répéter static

Cas particulier

```
class Circle {                          // fichier circle.hh
public:
  static const float PI = 3.1415926535;
  ...
};
```

possible uniquement pour types de base const

Méthodes de classe

```
// déclaration: fichier circle.hh

  class Circle {
  public:
    static const float PI = 1415926535;
    static float getPI();
    ...
  };

// définition: fichier circle.cc

  float Circle::getPI() {return PI;}

// appel: fichier test.cc

  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 !

Restrictions d'accès

Trois niveaux

private (le défaut) : accès réservé à cette classe

protected : idem + sous-classes

public

Remarques

le mot-clé porte sur tout ce qui suit (!= Java)

struct C++ == class entièrement publique

```
class Circle {
public:
  static const float PI = 3.1415926535;
  int x, y;

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

Restrictions d'accès (2)

```
class Circle {
private:                                    // NB: private par défaut
  int x, y;
  unsigned int radius;

public:
  static const float PI = 3.1415926535;    // publique car const
  Circle();
  Circle(int x, int y, unsigned int r);
  ....
};
```

Règles usuelles (sauf cas particuliers)

variables: private (ou protected)

méthodes: public (ou protected)

constructeurs: public

Friends

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

class Manager {
   ...              // a accès aux variables des instances de Circle
};

bool equals(const Circle* c1, const Circle* c2) {
   return (c1->x == c2->x && c1->y == c2->y && c1->radius == c2->radius);
}
```

friend donne accès aux variables de la classe

à une fonction ou à une autre classe

NB: pas de protection "package"

Namespaces

namespace = espace de nommage

notion +/- similaire aux packages de Java

using namespace xxx

évite d'avoir à prefixer les classes du namespace xxx (-> import de Java)

std : namespace des bibliothèques standard du C++

```
#include <iostream>                    // E/S du C++
#include "circle.hh"

using namespace std;

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


Bibliothèque standard d'E/S

```
#include <iostream>                          // E/S du C++
#include "circle.hh"

using namespace std;

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

   cout << "radius= " << c->getRadius()     // necessite using template std
        << "area= "   << c->getArea();
        << endl;

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

Concaténation des arguments via << ou >>

std::cout : sortie standard

std::cerr : sortie des erreurs

std::cin : entrée standard (utiliser >> au lieu de <<)

Inclusion des headers

#include

```
#include <iostream>
#include "circle.hh"
```

inclusion textuelle

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

- option -I du compilateur: -I/usr/X11R6/include

Empêcher les inclusions multiples

```
#ifndef _shape_hh_
#define _shape_hh_

  class Shape {
     ...
  };

#endif
```


Inlines

Méthodes implémentées dans les headers

```
class Circle {
  ...
public:
  Circle() {x = y = radius = 0;}                // inline implicite
  virtual unsigned int getRadius() const {return radius;}
  ....
};

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

Avantages :

pas d'appel fonctionnel (en général) -> rapidité

remplace les MACROS (#define) du C

Inconvénients :

augmente taille du code binaire généré

headers peu lisibles

contraire au principe d'encapsulation -> à utiliser avec discernement !

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

Méthodes d'instance : le retour

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

  virtual unsigned int getRadius() const;
  virtual unsigned float getArea() const;
  ....
};
```

virtual

optionnel

souvent nécessaire (à suivre...)

const

la méthode ne modifie pas les variables d'instance

n'a de sens que pour les méthodes d'instance !

très important ! (à suivre...)

Où est la magie ?

Méthodes d'instance toujours appliquées à un objet

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

int main() {
  Circle c = new Circle(100, 200, 35);
  unsigned int r = c->getRadius();      // OK
  unsigned float a = getArea();          // INCORRECT: POURQUOI?
}
```

Et pourtant :

```
unsigned float Circle::getArea() const {
  return PI * getRadius() * getRadius();   // CORRECT
}

unsigned int Circle::getRadius() const {
  return radius;     // ou est le lien avec la VI de l'objet ?
}
```


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 float Circle::getArea() const {
  return PI * radius * getRadius();
}

Circle c = new Circle(100, 200, 35);
unsigned float a = c->getArea();
```

Transformé par le compilateur en l'équivalent de :

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

unsigned float a = Circle::getArea(c);
```

Terminologie

Méthode versus fonction

méthodes d'instance == fonctions membres

méthodes de classe == fonctions statiques

fonctions classiques == fonctions globales

etc.

Termes interchangeables selon auteurs

Exemple de header : circle.hh

```
class Circle {
  static const float PI;
  int x, y;
  unsigned int radius;

public:
  Circle();
  Circle(int x, int y, unsigned int r = 10);

  static float getPI();

  virtual int getX() const;
  virtual int getY() const;
  virtual unsigned int getRadius() const;
  virtual unsigned float getArea() const;

  virtual void setX(int);
  virtual void setY(int);
  virtual void setRadius(unsigned int);
};                              // !NE PAS OUBLIER LE ;
```

Exemple d'implémentation : circle.cc

```
#include <iostream>
#include "circle.hh"

using namespace std;

const float Circle::PI = 3.1415926535;

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

  float pi = Circle::getPI();
  Circle* sh = new Circle(50, 200);
  sh->setX(15);
  int y = sh->getY();
  sh->setRadius(sh->getRadius() + 35);
  cout << "position: x=" << sh->getX() << " y=" << sh->getY() << endl;
  exit(0);
}

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

Circle::Circle() {
  x = y = 0;
  radius = 0;
}

float Circle::getPI() {return PI;}
```



```
float Circle::getPI() {return PI;}

int Circle::getX() const {return x;}
int Circle::getY() const {return y;}
unsigned int Circle::getRadius() const {return radius;}
unsigned float Circle::getArea() const {return PI * radius * radius;}

void Circle::setX(int _x) {x = _x;}
void Circle::setY(int _y) {y = _y;}
void Circle::setRadius(unsigned int _r)  {radius = _r;}
```

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

```
// HEADERS - - - - - - - - - - - - - - - - - - - - - - - - - - -
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 méthodes
  virtual void setHeight(unsigned int);
};

// IMPLEMENTATION - - - - - - - - - - - - - - - - - - - - - - - -

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

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



```
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

Constructeur par défaut

Constructeur par défaut

implicite si AUCUN constructeur n'est défini

ne fait rien (variables non initialisées !)

! à éviter

Constructeur sans argument

```
Rect::Rect() {
  x = y = 0;
  width = height = 0;
}

Square::Square() {}
```

pour initialiser les variables à des valeurs par défaut

! fortement conseillé

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 ?

Liaison dynamique

la méthode liée à l'objet (= à l'instance) est appelée

le choix de la méthode se fait à l'exécution

liaison "dynamique" (= "tardive")

Mécanisme essentiel de l'OO !

Méthodes virtuelles

Deux cas possibles en C++ :

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

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

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

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 (2)

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

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

Conséquences

les méthodes d'instance doivent généralement être virtuelles

surtout pour les classes de base

- car les redéfinitions de MV sont automatiquement virtuelles
- (même si virtual est omis)

! sérieux risques d'erreurs dans le cas contraire !

Méthodes virtuelles (3)

Remarques

les redéfinitions de MVs doivent avoir la même signature

si une MV est surchargée il faut redéfinir toutes les variantes

```
class Rect {
public:
  virtual void setWidth(unsigned int w);
  virtual void setWidth(unsigned float w);
  ...
};

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


Méthodes virtuelles (4)

Cas de Java

les méthodes sont virtuelles par défaut (convention inverse du C++)

Pourquoi les méthodes sont non virtuelles par défaut ?

par compatibilité avec le C

à quoi ca peut servir ?

- à redéfinir des méthodes avec des signatures différentes (à éviter)
- à optimiser l'exécution dans les cas extrêmes (rares !!!)

Comment ca marche ?

via des pointeurs de fonctions (virtual table)

Méthodes abstraites

Ces méthodes :

ne peuvent pas être implémentées

doivent être redéfinies et implémentées dans les sous-classes

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

syntaxe: virtual et = 0

"pure virtual functions" en jargon C++

Classes abstraites

Classes de spécification :

contiennent au moins une méthode abstraite

-> ne peuvent pas être instanciées

Classes héritées instanciables :

-> doivent implémenter toutes les méthodes abstraites

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

class Rect : public Shape {   // nouvelle version: herite de Shape
public:
  virtual void setWidth(unsigned int w);
  ...
};
```

Classes abstraites (2)

Objectifs

"commonaliser" les déclarations de méthodes

- -> 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 {                                   // abstract class
  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;   // abstract
  virtual unsigned int getHeight() const = 0;   // abstract
  virtual unsigned float getArea() const = 0;   // abstract
  ...etc...
};


class Circle {
  static const float PI = 3.14;
  unsigned int radius;

public:
  Circle();
  Circle(int x, int y, unsigned int r = 10);

  // l'implémentation de getX() et getY() est héritée
  virtual unsigned int getRadius() const {return radius;}
  virtual unsigned int getWidth()  const {return 2 * radius;}
  virtual unsigned int getHeight() const {return 2 * radius;}
  virtual unsigned float getArea() const {return PI * radius * radius;}
  ...etc...
}
```

Généricité

```
#include <iostream>
#include "shape.hh"
#include "rect.hh"
#include "square.hh"
#include "circle.hh"

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

  Shape** tab = new Shape* [10];             // vecteur 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;
  }
}
```

Généricité (2)

Gestion unifiée

des classes dérivant la classe abstraite

sans avoir besoin de connaître leur type !

- ! contrairement à la programmation "classique" ...

Modularisation et évolutivité

indépendance des implémentations des divers "modules"

rajout de nouvelles classes sans modification de l'existant

Spécification indépendante de l'implémentation

les classes se conforment à une spécification commune

-> développement en parallèle par plusieurs équipes

Interfaces

Classes totalement abstraites

toutes les méthodes sont abstraites

aucune implémentation

-> pure spécification d'API

(en C++) cas particulier de classe abstraite

pas de mot-clé "interface" comme en Java

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 float getArea() const = 0;    // abstract
  ...etc...
};

class Circle {
  static const float PI = 3.14;
  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 des méthodes

- -> 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
};

void NamedRect::draw() {
  Rect::draw();             // trace le rectangle
  /* code pour afficher le nom */
}
```

Classes imbriquées (inner classes)

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

    Point p1, p2;
    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) {}

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

Technique de composition très utile

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

Remarques

syntaxe d"appel des constructeurs des classes imbriquées

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

Méthodes virtuelles: où est la magie ?

Liaison dynamique / tardive / polymorphisme

mécanisme essentiel

par défaut en Java

coût légèrement plus élévé

Comment ca marche ?

tableau de pointeurs de fonctions (vtable)

- 1 vtable par classe
- 1 pointeur par objet --> vtable de sa classe
- double indirection à l'appel

Chapitre 3 : Mémoire

C++ permet d'allouer les objets

1. en mémoire dynamique (new)
2. dans la pile (variables locales)
3. en mémoire statique (variables static)

```
void foo() {
    Square * s1 = new Square(10, 20, 300);      // 1. dynamique
    Square s2(10, 20, 300);                     // 2. pile
    static Square s3(10, 20, 300);              // 3. statique
}
```

la variable s1 est un pointeur

- qui pointe sur un objet

les variables s2 et s3

- "contiennent" l'objet

Dans tous les cas

```
void foo() {
    Square * s1 = new Square(10, 20, 300);      // 1. dynamique
    Square s2(10, 20, 300);                     // 2. pile
    static Square s3(10, 20, 300);              // 3. statique
}
```

Constructeur appelé quand l'objet est créé

ainsi que ceux des superclasses (chaînage des constructeurs)

éxécutés dans l'ordre "descendant"

Destructeur appelé quand l'objet est détruit

ainsi que ceux des superclasses (chaînage des destructeurs)

éxécutés dans l'ordre "ascendant"

Ce qui change

```
void foo() {
    Square * s1 = new Square(10, 20, 300);      // 1. dynamique
    Square s2(10, 20, 300);                     // 2. pile
    static Square s3(10, 20, 300);              // 3. statique
}
```

La durée de vie de l'objet :

1. dynamique : de new à delete
2. pile : définition de la variable -> sortie de la fonction
3. statique : définition de la variable -> sortie du programme

Remarques

attention, ici le pointeur s1 est détruit mais pas ce qu'il pointe !

Cas de la mémoire dynamique

Pas de ramasse miette !

penser à appeler delete quand c'est nécessaire

! RAPPEL : delete ne met pas le pointeur à 0 !

```
Square * s1 = new Square(10, 20, 300);
....
delete s1;
s1 = 0;           // NE PAS OUBLIER sinon s1 est indefini !
```

Si on oublie delete

fuites mémoire ...

mais mémoire récupérée en fin de programnme

- mais destructeurs pas appelés !

Destruction automatique ?

possible avec librairies optionnelles (sous certaines conditions)

ou par comptage de références (handle classes, smart pointers...)

Cas de la pile et de la mémoire statique

Destruction automatique

jamais de delete dans ces 2 cas !

Remarques

ces 2 possibilités n'existent pas en Java

syntaxe d'accès aux variables et fonctions membres :

```
void foo() {
    Square * s1 = new Square(10, 20, 300);      // 1. dynamique
    Square s2(10, 20, 300);                     // 2. pile
    static Square s3(10, 20, 300);              // 3. statique

    int x = s1->getX();
    int y = s2.getY();
    unsigned int w = s3.getWidth();
}
```

notation -> fléchée uniquement pour les pointeurs . sinon

Variables de classe et d'instance

Même principe :

```
class Dessin {
    Square * s1;        // var. d'instance (pointeur vers objet)
    Square s2;          // var. d'instance (contient l'objet)
    static Square s3;   // var. de classe
};
```

s1 est un pointeur qui pointe sur un objet

s2 et s3 "contiennent" l'objet

Durée de vie de l'objet

en mémoire dynamique (*s1) ou statique (s3) : comme précédemment

"contenu" dans variable d'instance (s2) : comme l'objet contenant

Comment initialiser s1, s2, s3 ?

Initialisation

```
class Dessin {
    Square * s1;
    Square s2;
    static Square s3;
public:
   Dessin(int x, int y, unsigned int w);
};
```

variables de classe :

```
Square Dessin::s3(10, 20, 300);  // on ne repete pas "static"
```

variables d'instance :

```
Dessin::Dessin(int x, int y, unsigned int w) :
    s1(new Square(x, y, w)),   // new : s1 est un pointeur
    s2(x, y, w) {              // pas de new : s2 contient l'objet
};
```

on peut aussi écrire :

```
Dessin::Dessin(int x, int y, unsigned int w) : s2(x, y, w) {
   s1 = new Square(x, y, w);
};
```

Qu'est-ce qui manque ?

Destruction

Il faut un destructeur !

chaque fois qu'un constructeur fait new (memoire dynamique)

sinon fuites mémoires...

```
class Dessin {
    Square * s1;
    Square s2;
public:
   Dessin(int x, int y, unsigned int w);
   virtual ~Dessin();
};


Dessin::Dessin(int x, int y, unsigned int w) :
    s1(new Square(x, y, w)),
    s2(x, y, w) {
};


Dessin::~Dessin() {
    delete s1;            // s2 est detruit automatiquement
}
```

Qu'est-ce qui manque ?

Initialisation et affectation

```
class Dessin {
    Square * s1;
    Square s2;
public:
   Dessin(int x, int y, unsigned int w);
   virtual ~Dessin() {delete s1;}
};


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 * s1;
    Square s2;
public:
   Dessin(int x, int y, unsigned int w);
   virtual ~Dessin() {delete s1;}
};

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 dans d2, d3 et d4

- toutes les variables s1 pointent sur la même instance de Square
- BOUM! cette instance est détruite 4 fois quand on sort de foo( ) !!!
- ... et les 3 autres ne sont jamais détruites

La copie d'objets est dangereuse !

S'ils contiennent des pointeurs ou des références !

Interdire ou redéfinir :

l'opérateur d'initialisation (Copy Constructor) :

```
Dessin(const Dessin&);     // définit: Dessin d3 = d1;
```

l'operateur d'affectation :

```
Dessin& operator=(const Dessin&);  // définit: d2 = d1;
```

Attention

! ce n'est pas le même opérateur !

1e solution: interdire la copie

```
class Dessin {
    Square* s1;
    Square s2;

public:
   Dessin(int x, int y, unsigned int w);
   Dessin() : s1(0) {}
   ~Dessin() {delete s1;}

private:
   Dessin(const Dessin&);                // initialisation
   Dessin& operator=(const Dessin&);     // affectation
};
```

Interdit la copie, y compris pour les sous-classes

implémentation inutile

2e solution: redéfinir la copie

```
class Dessin {
    Square* s1;
    Square s2;

public:
   Dessin(int x, int y, unsigned int w);
   Dessin() : s1(0) {}
   ~Dessin() {delete s1;}

   Dessin(const Dessin&);                // initialisation
   Dessin& operator=(const Dessin&);     // affectation
};

};
```

Implémentation :

```
Dessin::Dessin(const Dessin& d) {
    if (d.s1) s1 = new Square(*d.s1); else s1 = 0;
    s2 = d.s2;
}

Dessin& Dessin::operator=(const Dessin& d) {
   delete s1;                           // ! ne pas oublier !
   if (d.s1) s1 = new Square(*d.s1); else s1 = 0;
   s2 = d.s2;
   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 (ie. "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
cin  >> s2->setX();             // INTERDIT!
```


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 références

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

cas par défaut en C++

seule possibilité en C et en Java

Passage par valeur (2)

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) {
    // recupere i depuis la socket
    i = ...;
}

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

l'argument N'EST PAS recopié dans le paramètre de la fonction

mais c'est son adresse mémoire qui est transmise

- => a et i référencent la même entité
- => donc a est bien modifié !

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 : les champs risquent de pointer sur des copies des noeuds (et non sur les noeuds eux-mêmes)

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);                   // valeur
    void receive(int& i);               // reference
    void send(const string& s);         // const reference
    ....
};
```

Passage par valeur

- l'argument est recopié dans le paramètre
- => l'argument n'est jamais modifié

Passage par référence

- l'argument et le paramètre référencent la même entité
- => l'argument peut être modifié

Passage par const référence

- l'argument et le paramètre référencent la même entité
- mais le paramètre ne peut pas être modifié
- => l'argument n'est jamais modifié

Compléments (1)

Opérateurs d'initialisation et d'affectation (rappel)

```
// d pas recopié, pas modifié
Dessin(const Dessin& d);

Dessin& operator=(const Dessin&);
```

Valeur de retour des fonctions

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

class User {
    string name;

public:
    // recopie name (coûteux!)
    string getName1() const {return name;}

    // ne recopie PAS name (mais interdit les modifs)
    const string& getName2() const {return name;}
    ....
};
```


Compléments (2)

```
class User {
    string name;

public:
    void setName(const string& s) {name = s;}
    ....
};
```

Conversions implicites des const références :

```
User u;
u.setName("Napoleon");    // char* -> string
```

- possible car la classe std::string definit le constructeur:

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

Initialisation des variables d'instances qui sont des références

- comme les variables const (car ce sont des constantes !)

Et les pointeurs ?

Pas de passage par référence en C ni en Java !

=> il faut utiliser des pointeurs

Remarque

les "références" de Java sont en fait des pointeurs

... et n'ont pas le même sens que les références du C++ !

Passage "par pointeurs"

```
class MySocket {
    void send(int i);                  // valeur
    void receive(int* i);              // pointeur
    void send(const string* s);        // const pointer
};

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

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

void foo() {
   int a1;
   sock.receive(&a1);     // ne pas oublier le &
   string a2 = "une chaine ...";
   sock.send(&a2);        // ne pas oublier le &
}
```

Même résultat que les références via une indirection :

la valeur du pointeur est recopiée (passage par valeur)

mais pas ce qu'il pointe !

Pointeurs versus Références

Pointeur

variable qui pointe vers une entité

```
Circle c;
Circle* p1 = &c;                    // ne pas oublier &
Circle* p2 = new Circle();
p1->setX(15);
```

Références C++

constante qui référence une entité (= son adresse mémoire)

concept général en C++ (pas limité au passage des arguments)

```
Circle c;
Circle& r1= c;
Circle& r2= *new Circle();          // ne pas oublier *
r1.setX(15);
```

r1 peut être vue comme une sorte d'"alias" de c

- comme pour c on utilise . et non ->

Pourquoi utiliser les références

Parce que c'est plus simple

entre autres pour le passage par référence

Parce que c'est plus sûr

pas d'arithmétique des références (source notoire d'erreurs)

toujours initialisées (ne peuvent pas pointer sur 0)

référencent TOUJOURS la même entité

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

attention: si r1 et r2 étaient des pointeurs

- r1 = r2 ferait pointer r1 sur c2 !!!

Chapitre 6

Compléménts sur les types et les déclarations

Transtypage

Implicite vers les super-classes

```
class Object {
    ...
};

class Button : Object {
    ...
};

Object* o;
Button* b;

o = b;            // OK: transtypage implicite
b = o;            // ERREUR de compilation!
```

Transtypage : static_cast

Ne jamais utiliser l'opérateur de cast du C

```
Object* o;
Button* b;

b = (Button*) o;            // JAMAIS!
```

static_cast

change impérativement le type

verif syntaxique minimaliste à la compilation

à éviter

```
b = static_cast<Button*>(o);     // A PEINE MIEUX!
```


Transtypage : dynamic_cast

Contrôle dynamique à l'exécution

```
Object* o = ...;

Button* b = dynamic_cast<Button*>(o);

if (b) {
  ....;           // si l'instance derive de Button
else {
  ....;           // sinon b est nul
}
```

Autres opérateurs

const_cast

reinterpret_cast

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 (type const char*)

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)
    ....
}
```

Rappel: protection des headers

```
// header shape.hh

#ifndef _shape_hh_
#define _shape_hh_

  class Shape {
     ...
  };

#endif
```

Evite les redéclarations

si inclusions multiples des headers

généralement dues à des headers qui incluent d'autres headers

```
// fichier test.cc

#include "shape.hh"
#include "shape.hh"   // 2eme inclusion sans effet
```


Types partiellement définis

```
// header circle.hh

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

Ces déclarations partielles

ne nécessitent pas de connaître `CircleImpl` ni `Rect`

propriété des types pointeurs ou références

Types partiellement définis (2)

```
// header circle.hh

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

Propriété éssentielle pour

limiter les dépendances entre headers

- inutile d'inclure `rect.hh`

cacher l'implémentation

- `CircleImpl` définie dans header privé non fourni au client

Handle classes

Ne contiennent

qu'un pointeur vers la classe d'implémentation

et l'API publique

Possibilité: gestion auto de la mémoire

par comptage du nombre de handles associées à chaque objet d'implémentation

-> Smart Pointers ...

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);
```


Doxygen

Système de documentation automatique

similaire à JavaDoc

```
/**
 * racine de la hierarchie des classes d'objets graphiques
 */
class Shape {

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

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

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

http://www.doxygen.org

Chapitre 7 : Surcharge des opérateurs

Possible pour presque tous les opérateurs

en particulier: = == < > + - * / ++ -- += -= -> () [] new delete

exceptions: :: . .* ?

la priorité est inchangée

A utiliser avec discernement !

Exemple typique:

```
class String {
  /// s += s2  et  s += "abcd"
  String& operator+=(const String&);
  String& operator+=(const char*);

  /// s = s2 + s3  et  s = s2 + "abcd"
  friend String operator+(const String&, const String&)
  friend String operator+(const String&, const char*)

  /// s1 == s2  et  s1 == "abcd"
  friend bool operator==(const String&, const String&)
  friend bool operator==(const String&, const char*)
};
```


Surcharge des opérateurs (2)

operator++

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

operator[ ]

l'"indice" n'est pas forcement un int -> mémoires associatives, etc.

```
class String {
    char operator[](int);           // retourne ieme char
    int  operator[](const char*);   // position d'une sous-chaine
};
```

operator()

"objets fonctionnels" (STL) ou accès via indices

Surcharge des opérateurs (3)

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

redéfinition de l'allocation mémoire

conversions de types

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

operator-> (smart pointer)

```
class Ref {
    Obj* obj;
public:
    Ref(Obj *o) : obj(o) {}
    Obj* operator->() const {return obj;}
}

Ref r = new Obj();
r->foo();      // exécute:  obj->foo();
```


Chapitre 8 : Traitement des erreurs

Exceptions

```
class MathErr {};
class Overflow : public MathErr {};

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

try {
   ....
   throw Zerodivide(x);         // si on divise x par 0
}
catch (Zerodivide& e) { cerr << e.x << "divise par 0 \n"; }
catch (MathErr)       { cerr << "erreur de calcul \n"; }
catch (...)           { cerr << "erreur quelconque \n"; }
```

Exceptions (2)

Organisation

généralement regroupées en hiérarchies de classes

héritage multiple également possible

Spécifications d'exceptions

```
void foo() throw (Overflow, Zerodivide);
void zoo() throw ();
```

optionnel mais préférable

throw non indiqué : peut lancer n'importe quelle exception

reféfinitions dans sous-classes : idem 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)

Attention aux fuites mémoire !

seules les variables dans la pile seront desallouées !

-> prévoir traitement approprié pour les autres

Exceptions standard

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

Handlers

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

Chapitre 9 : 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 bases

Constructeurs

headers

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

class 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);
};
```

fichier d'implementation

```
NamedRect::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();
};

void NamedRect::draw() {
  Rect::draw();
  Name::draw();
}
```

pas obligatoire mais préférable

même principe pour variables

using déclarations

```
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

Chapitre 10 : 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);
```


Exemple: auto pointer

prédéfini dans <memory> :

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()     {delete ptr;}
    T* operator->() {return ptr;}
    T& operator*()  {return *ptr;}
    .....
};
```

utilisation

```
#include <memory>

void foo()
{
    auto_ptr p = new Circle();
    p->setWidth(300);
}
```

l'objet pointé est détruit automatiquement quand on sort de foo()

problème : un seul auto_ptr peut pointer sur un objet donné !

voir solution avec comptage de références à la fin

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];
```

Algorithmes

manipulent les données des conteneurs

génériques

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


STL (2)

Itérateurs

sortes de pointeurs généralisés

exemple: v.begin() et v.end()

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

doc en ligne

http://www.sgi.com/tech/stl/

Conteneurs

vector

list

map

deque

queue

stack

set

bitset

Exemple de vecteur

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

struct Point {
    int x, y;
    Point(int x, int y);
};

  vector<Point> 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*> points;

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

  int xx = 50, yy == 50;
  Point* found = null;

  for (list<Point*>::iterator k = points.begin();
       k != points.end();
       k++) {
     if ((*k)->x == xx && (*k)->y == yy) {
         found = *k;
         break;
     }
  }
```

"points" est une liste de pointeurs d'objets

pas d'accès direct aux éléments

coût d'insertion / suppression faible

la liste est doublement chaînée

Exemple avec destruction

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

PointList point_list;
int xx = 200;

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

  if ((*k)->x < xx) k++;
  else {
      PointList::iterator k2 = k; k2++;
      delete *k;
      point_list.erase(k);
      k = k2;
   }
}
```

Attention

l'itérateur est invalide après erase()

l'objet pointé par l'itérateur doit être détruit (par delete)

Exemple d'utilisation d'un "algorithme"

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

class Entry {
public:
  Entry(const std::string& name);
  bool isDir()  const;
  const std::string& getName() const;
  friend bool compareEntries(const Entry*, const Entry*);
};

std::vector<Entry*> entries;
// ...

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

bool compareEntries(const Entry* e1, const Entry* e2) {
  if (e1->isDir() && !e2->isDir())
    return true;
  else if (!e1->isDir() && e2->isDir())
    return false;
  else
    return e1->getName() < e2->getName()):
}
```


Smart pointer avec comptage de références

Idée

```
void foo() {

    sptr<Circle> p1 = new Circle(0, 0, 50);        // objet A
    sptr<Circle> p2 = new Circle(100, 100, 255);   // objet B
    sptr<Circle> p3;                               // pointe sur 0

    p1 = 0;    // p1 pointe sur 0, A est detruit automatiquement

    p3 = p2;   // p2 et p3 pointent sur B

    p2 = 0;    // p2 pointe sur 0, B pas detruit car pointé par p3

}  // p3 est detruit (var. locale) => B est detruit automatiquement
```

gestion automatique de la mémoire (pas de delete)

plusieurs (smart) pointeurs peuvent pointer sur le même objet

- contrairement à auto_ptr de la STL

principe : comptage de références

Implémentation

```
/** Une classe de base pour le comptage de references.
 */
class Obj {
  unsigned int refcount;
public:
  Obj() : refcount(0) {}
  void addRef()    {refcount++;}
  void removeRef() {refcount--; if (refcount == 0) delete this;}
};

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

class Circle : public Shape {..... };
```

Principe

p = objet incrémente le compteur de références de cet objet

p = null (ou destruction de p) décrémente le compteur

p1 = p2 incrémente objet2 et décrémente objet1

l'objet s'auto-détruit qunad le compteur arrive à 0

Implémentation (suite)

```
template <class CC> class sptr {
  CC* obj;

public:
  sptr(CC* o = 0) : obj(o) {if (obj) obj->addRef();}
  ///< initialisation:  sptr<Obj> p = object;

  sptr(const sptr& p2) : obj(p2.obj) {if (obj) obj->addRef();}
  ///< initialisation:  sptr<Obj> p = p2;

  ~sptr() {if (obj) obj->removeRef();}
  ///< destruction.

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  sptr& operator=(CC* o)
  {if (obj) obj->removeRef(); obj = o; if (obj) obj->addRef(); return *this;}
  ///< affectation:  p = object;

  sptr& operator=(const sptr<CC>& p2)
  {if (obj) obj->removeRef(); obj = p2.obj; if (obj) obj->addRef(); return *this;}
  ///< affectation:  p = p2;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  CC* operator->() {if (!obj) throw NullPointer(); return obj;}
  ///< dereference:  p->foo() effectue obj->foo()

  CC& operator*() {if (!obj) throw NullPointer(); return *obj;}
  ///< dereference:  *p renvoie *obj
```

```
};
```


Smart pointer

Avantages

mémoire libérée automatiquement sans jamais faire delete

sptr est compatible avec toute classe possédant addRef() et removeRef()

Contraintes

nécessite un compteur de références pour chaque objet

ne marche pas s'il y a des cycles

n'est censé pointer qu'un objet créé avec new

Plus d'infos

implémentation complète

un site intéressant sur les smart pointers

site traitant des garbage collectors en C++