Programmation scientifique orientée objet

UML, C++, make, gnuplot

Damien Olivier

Université du Havre
Master MI 1ère année

Octobre 2009

Sommaire

Introduction

1. Présentation
   - Contexte, objectifs et enjeux
   - Modèles
   - Simulation
   - Exemple

UML

2. UML
   - Introduction
   - Définition et historique
3. UML un langage
   - UML et les domaines d'utilisation
   - Vues en UML
4. Les diagrammes UML
   - Entité, objets, relations et diagrammes
   - Diagrammes de classes
   - Diagramme de cas
   - Diagramme de séquences
   - Diagramme de collaboration
   - Diagramme d'état
   - Diagramme d'activités

C++

5. Présentation du C++
   - Introduction
   - Quelques spécificités
   - LOO
6. Notion de référence
   - Définition
   - Déclaration et initialisation
   - Transmission par référence
   - Les modes de transmission
   - Fonctions renvoyant une référence
   - Surcharge de fonction
7. Allocation dynamique
   - Organisation de la mémoire
   - Les opérateurs

Sommaire

C++ (suite)

8. Langage objet
   - Classe
   - Constructeurs-destructeurs
   - Surcharge d'opérateurs
   - Fonctions et classes amies
   - Surcharge des opérateurs de flux fichiers
   - Ex : classe matrice
9. Héritage
   - Constructeurs et destructeurs
   - Contrôle des accès
   - Constructeur par recopie
10. Virtualité, classes abstraites
    - Polymorphisme
    - Méthodes virtuelles
    - Classes abstraites
    - Héritage multiple

C++ avancé

11. Généricité, template et namespace
    - Template
    - Namespace
12. Exceptions en C++
13. Patron de conception
    - Les principaux patrons de conception
14. La STL
    - L'API STL
15. Qt
    - Généralités
    - Architecture objet de Qt
    - Quelques widgets
    - Placer les widgets

Plan

1 Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

Introduction

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

- Objectifs : Maîtriser l'environnement de programmation pour le calcul scientifique
- Orientations : Programmation orientée objet

Objectifs et enjeux du calcul scientifique

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

On cherche à construire des modèles et à mettre en œuvre des simulations dans de nombreux domaines d'applications :

- Rentrée dans l'atmosphère d'une capsule spatiale ;
- Soufflerie numérique (avions, automobiles, ...) ;
- Aménagement fluvial ;
- Trafic routier ;
- Simulation de réacteurs nucléaires ;
- Évolution climatique . . .

FIG.: Climat

Modèles

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

- Modélisation : Concevoir une représentation d'un objet ou d'un phénomène d'une manière utilisable.

  "Tout ce qui est simple est faux, tout ce qui ne l'est pas est inutilisable" – Paul Valéry

- Différentes natures des modèles :
  - Carte, plan ou maquette physique . . .
  - Algorithme ;
  - Formulation mathématique ;
  - . . .

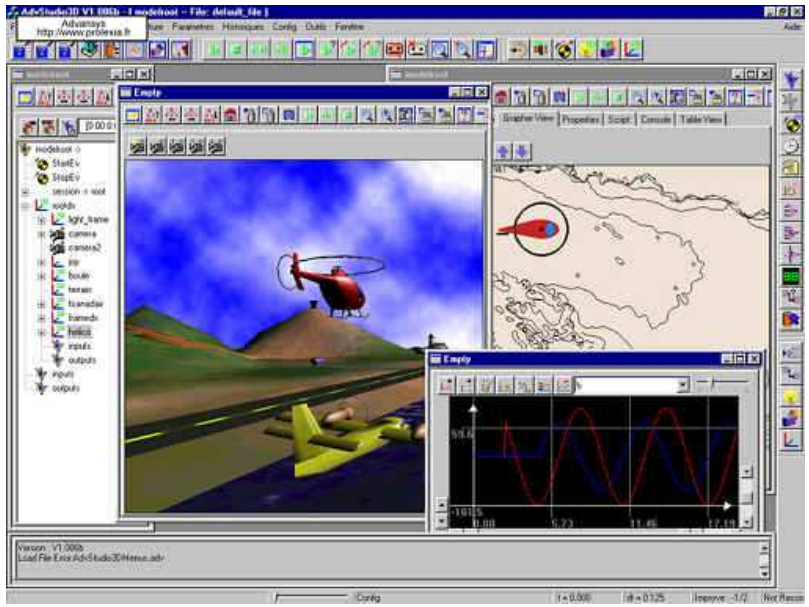


FIG.: Exemple de simulation

Précision de la description

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

- Pour un phénomène ou un objet, il existe plusieurs niveaux de description. Il faut trouver le niveau pertinent
- Modélisation d'un avion ... comment le décrire ?

  - Une seule entité ponctuelle et indivisible qui se déplace : pertinent pour la gestion du trafic aérien
  - Représentation des molécules constituant le gaz du réacteur : pertinent pour l'étude de la combustion et compression interne

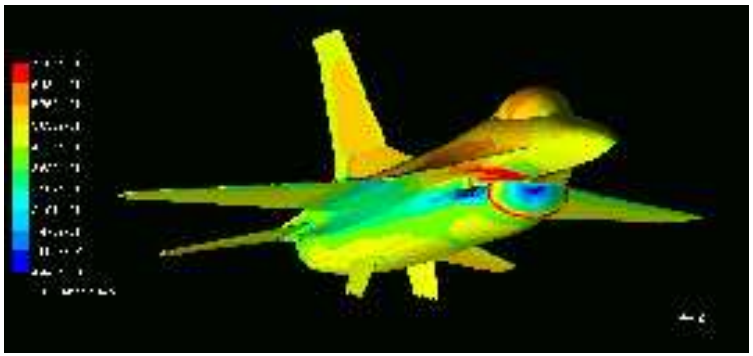


FIG.: Modélisation d'un avion

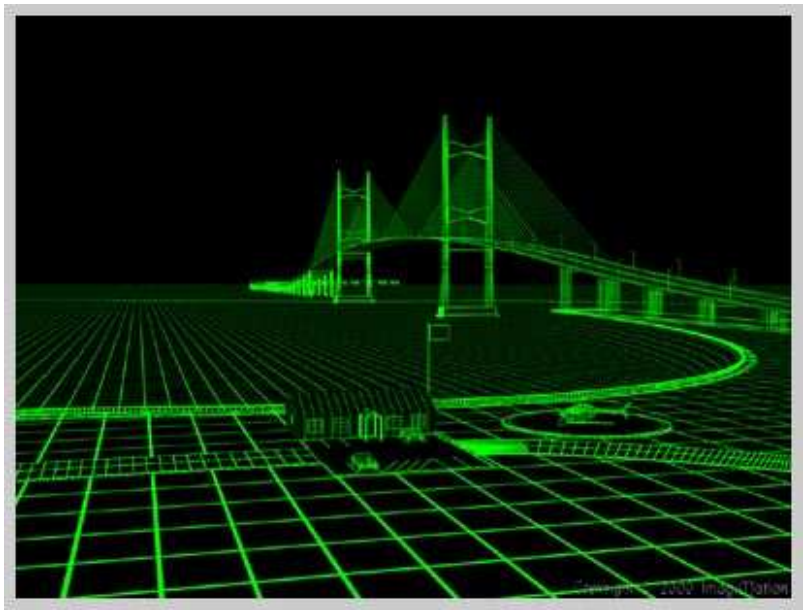
- Et lorsque plusieurs niveaux de description sont simultanément nécessaires (déstabilisation d'un écosystème et ses conséquences à grande échelle) ?

Simulation numérique

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

Transformation d'une expression mathématique en programme informatique traitable par un ordinateur.

- un ordinateur ... nombre finis d'informations
- découpage des objets/équations en un ensemble fini de points ou de relations : discrétisation/ approximation



Précision de l'approximation

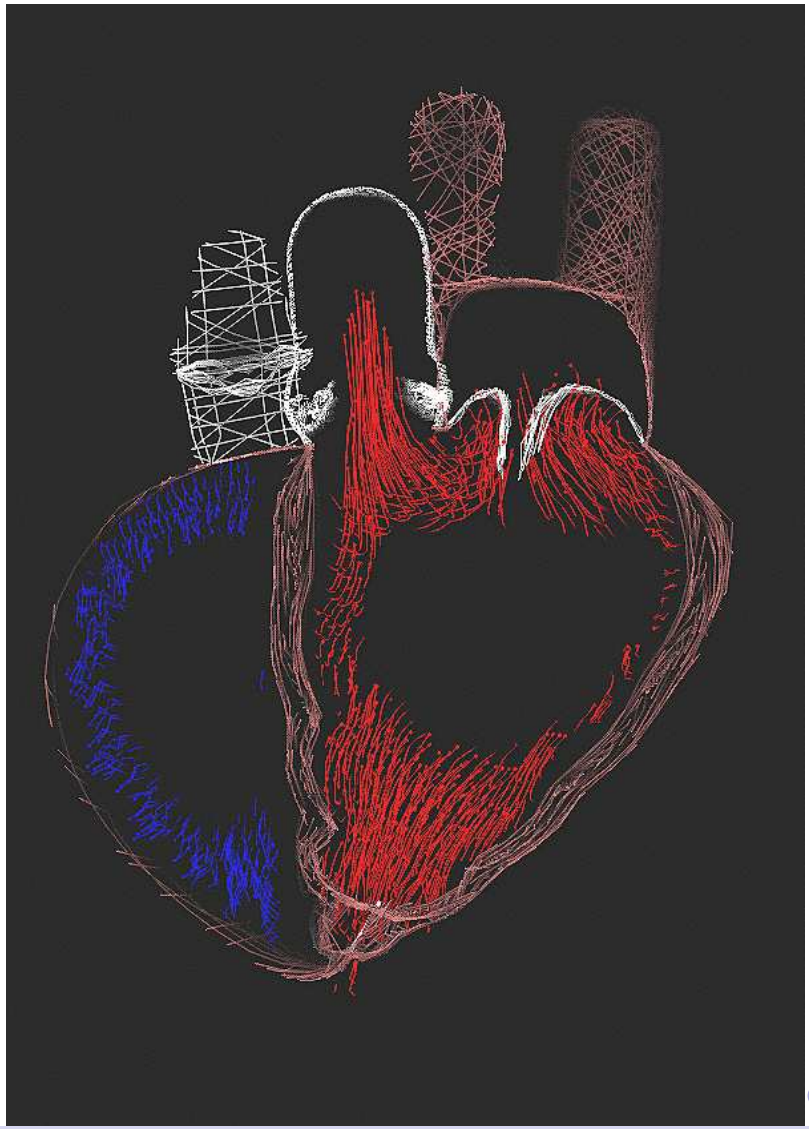
Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

- Problème : Trouver le degré d'approximation
  - Compatible avec le niveau de description du modèle ;
  - Gérable par l'ordinateur en un temps qui ne soit pas infini (ou presque).

Exploitation des résultats

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

- Visualisation ... graphisme 2D ou 3D
- Diagnostic / validation ... expérimentation ?
- Établir le domaine de validité du modèle (du à son niveau de description) et de sa résolution numérique (du à la précision de sa discrétisation)



Équation de la chaleur

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

Le problème

Quelle est la température T d'une barre métallique dont les deux extrémités sont maintenues aux températures T_0, T_M et plongée dans une atmosphère ambiante elle-même à une température donnée T_e. Cette barre est assimilée à un segment de droite $[0, L]$?

Modélisation et analyse numérique

Dans ce problème, il y a une perte de la chaleur due à la convection de l'air, que l'on peut modéliser par une fonction $a(x)$; la température T est alors solution de l'équation différentielle ordinaire :

$$-k\frac{d^2T}{dx^2} + a(x)(T - T_e) = 0 \text{ avec } 0 < x < L \text{ et } k \text{ coefficient de diffusion thermique.}$$

Conditions limites $T(0) = T_0$, $T(L) = T_M$

Discrétisation

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

Une bonne nouvelle

Si k est strictement positif et si a est une fonction à valeurs positives, alors le problème est bien admet en particulier une solution et une seule. On peut donc tenter un calcul approché.

Différences finies

On approche les termes de dérivation par des quotients différentiels, se basant sur la définition de la dérivée qui permet d'écrire, que pour h petit, on a :

$$\frac{df(x)}{dx} \simeq \frac{f(x+h)-f(x)}{h} \text{ et } \frac{d^2f(x)}{dx^2} \simeq \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$$

L'intervalle est divisé en M, la solution T_m est recherchée pour chaque $x_m = m\frac{L}{M}, m = 0..M$

Résolution

Cela revient à résoudre le système :

$$-\frac{k}{h^2}[T_{m+1} - 2T_m + T_{m-1}] + a_m(T_m - T_0) = 0, m = 1..M-1$$

Par Gauss Siedel, par exemple.

Le code C++ I

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include "rgrafpor.h"

const int M = 100;
const float dx = 0.1;
const int jmax = 200;

using namespace std;

typedef enum{ texte,graphique } affiche;

class data {
  public:
   float k, T0, TM, Te;
   float a[M];
   void read();
};

void init( float* T, data& d);
void relax( float* T, data& d);
void showresult( affiche mode, float* T);
```

Le code C++ II

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
void data::read()
{
cout << "enter k:";      cin >> k;
cout << "enter Te:"; cin >> Te;
cout << "enter T0:"; cin >> T0;
cout << "enter TM:"; cin >> TM;
for(int m = 0; m< M ; m++)
     a[m] = float(m)*float(m)/(M-1.)/(M-1.);
}

void init( float* T, data& d)
{
for(int m = 0; m<= M; m++)
    T[m]=d.T0 * float(M- m)/float(M) + d.TM *float(m)/float(M);
}

void relax( float* T, data& d)
{
   const float kdx2 = d.k / dx / dx;
   for(int m = 1; m< M; m++)
    T[m] = (kdx2 * (T[m+1] +T[m-1])
    + d.a[m] * d.Te) / (d.a[m]+ 2 * kdx2);
}
```

Le code C++ III

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
void showresult( affiche mode, float* T)
{ int m;
   if(mode==graphique)
   {
       initgraphique();
       move2( 0, (int)T[0]);
       for( m = 1; m<= M; m++)
        line2( m, (int)T[m]);
       rattente(1);
       closegraphique();
   } else if(mode==texte)
     for( m = 0; m<= M; m++)
       cout<<T[m]<<endl;
}

int main()
{
float* T = new float[M+1];
data d;
d.read();
init(T, d);
  for (int j= 0; j< jmax; j++)
   relax(T, d);
showresult(graphique, T);
```

Le code C++ IV

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
return 0;
}
```

Le code C++ I

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
#ifndef RGRAFPOR_
#define RGRAFPOR_
// D'apres Pironneau
// fichier rgrafpor.h
//=====================
void initgraphique();
void closegraphique();
void rattente(int waitm);
void move2(int x, int y);
void line2(int x, int y);
void reffecran();
#endif /*RGRAFPOR_*/
```

Le code C++ I

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
// Pas vraiment c++ !
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include "rgrafpor.h"

static Display *display;
static Window win;
static XSizeHints size_hints;
static GC gc;
static XFontStruct *font_info;
static int screen, width, height, currx, curry;

void closegraphique()
{
  XUnloadFont(display, font_info->fid);
  XFreeGC(display, gc);
  XCloseDisplay(display);
}

void rattente(int waitm)
```

Le code C++ II

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
{
  char click[] = "Click to continue...";
  char values[256];
  XEvent report;
  if (waitm)
    {
      XDrawString (display,
                   win,
                   gc,
                   5,20,
                   click,
                   strlen(click));
      do XNextEvent(display, &report);
      while (report.type != ButtonPress && report.type != KeyPress);
    }
  XCheckMaskEvent(display, ButtonPressMask,&report);
  if (report.type == ButtonPress)
  XFlush (display);
}

int xerror()
{
  fprintf(stderr, "Probleme avec X-Windows\n");
  return 1;
```

Le code C++ III

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
}

void initgraphique()
{
  XEvent report;
  display = XOpenDisplay(NULL);
  font_info = XLoadQueryFont(display, "7x13");
  XSetErrorHandler((XErrorHandler)xerror);
  XSetIOErrorHandler((XIOErrorHandler)xerror);
  screen = DefaultScreen(display);
  width = DisplayWidth(display, screen) - 100;
  height = DisplayHeight(display, screen) - 160;
  win = XCreateSimpleWindow(display, RootWindow(display, screen),
        50, 80, width, height, 4, BlackPixel(display, screen),
        WhitePixel(display, screen));
   size_hints.flags = PPosition | PSize;
   size_hints.x = 0;
   size_hints.y = 0;
   size_hints.width = width;
   size_hints.height = height;
   XSetStandardProperties(display, win, "plot", NULL, 0, NULL, 0, &size_h
   XSelectInput(display, win, ExposureMask | ButtonPressMask);
   gc = XCreateGC(display, win, 0, NULL);
   XSetFont(display, gc, font_info->fid);
```

Le code C++ IV

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
    XSetForeground(display, gc, BlackPixel(display, screen));
    XMapWindow(display, win);
    do XNextEvent(display, &report); while (report.type != Expose);
}

void move2(int x, int y)
{
    currx = x;
    curry = y;
}

void line2(int x, int y)
{
    int newx = x, newy = y;
    XDrawLine(display, win, gc, currx, curry, newx, newy);
    currx = newx;
    curry = newy;
}

void reffecran()
{
  XClearWindow(display,win);
}
```

Le code C++ I

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
# fichier makefile pour g++
.SUFFIXES: .cpp
CXX = g++-4.2
CXXFLAGS = -Wall
EXEC_NAME = heat
CXXINCLUDES = -I/usr/X11/include
CXXLIBS = -L/usr/X11/lib -lX11 -lm
OBJ_FILES = heatbeam.o rgrafpor.o
INSTALL_DIR = /usr/bin

all : $(EXEC_NAME)

clean :
rm $(EXEC_NAME) $(OBJ_FILES)

$(EXEC_NAME) : $(OBJ_FILES)
$(CXX) -o $(EXEC_NAME) $(OBJ_FILES) $(CXXLIBS)

%.o: %.cpp
$(CXX) $(CXXFLAGS) $(CXXINCLUDES) -o $@ -c $<

%.o: %.cc
$(CXX) $(CXXFLAGS) $(CXXINCLUDES) -o $@ -c $<
```

Le code C++ II

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

```
install :
cp $(EXEC_NAME) $(INSTALL_DIR)

### dependencies
rgrafpor.o: rgrafpor.h
heatbeam.o: rgrafpor.h
```

Références complémentaires

Présentation
Contexte, objectifs et enjeux
Modèles
Simulation
Exemple

- *"Rapport sur le calcul scientifique"*, écrit par un collectif coordonné par C. Bernardi, à la demande du CNRS. http ://www.ann.jussieu.fr/rapportCS/RapportCS.html
- Michel Bernadou *"Le calcul scientifique"*, Que sais-je ?, 2001

Plan

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

2 UML
Introduction
Définition et historique

3 UML un langage
UML et les domaines d'utilisation
Vues en UML

4 Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

Du problème au modèle

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- La modélisation consiste à créer une représentation simplifiée d'un problème.
- Le modèle doit permettre de simuler le comportement du problème.
- 2 étapes :
  1. L'analyse qui étudie problème.
  2. La conception qui simule le problème pour le résoudre.

FIG.: Modélisation des arborescences naturelles : L-System

Pourquoi modéliser ?

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Un modèle est une simplification de la réalité qui permet de mieux comprendre le système à développer.
- Il permet :
  - De visualiser le système comme il est ou comme il devrait être ;
  - De valider le modèle vis à vis des clients ;
  - De spécifier les structures de données et le comportement du système ;
  - De fournir un guide pour la construction du système ;
  - De documenter le système et les décisions prises.

Qu'apporte la modélisation ?

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Plus grande indépendance du modèle par rapport aux fonctionnalités demandées.
- Des fonctionnalités peuvent être rajoutées ou modifiées, le modèle objet ne change pas.
- Plus proche du monde réel.

Concept Objet

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Un objet représente un concept, une idée ou une chose réelle.
- C'est une agrégation d'états et de comportements cohérents (qui vont ensemble).
- Caractérisé par 3 propriétés :
  - une identité qui le distingue des autres objets.
  - un état qui le qualifie, qui peut évoluer.
  - un comportement qui décrit ce qu'il fait, comment son état évolue.

Intérêt de l'approche objet

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Encapsulation des données $\Longrightarrow$ Sécurité :
  - Interface publique : ce que l'on peut faire de l'objet, les services qu'il propose.
  - Interface privée : ses données membres et les méthodes de son fonctionnement interne.
- Héritage $\Longrightarrow$ Réutilisabilité :
  - Réutilisabilité du code.
  - Factorisation de parties communes.
  - Facilite et accélère l'évolution des programmes.
- Généricité $\Longrightarrow$ Fiabilité :
  - Sorte de "contrat de service" : Si ces méthodes sont implémentées dans une nouvelle classe alors celle-ci recevra tels évènements, telles informations.
  - Facilite la lisibilité du code.

Définition et historique

UML
- Introduction
- Définition et historique

UML un langage
- UML et les domaines d'utilisation
- Vues en UML

Les diagrammes UML
- Entité, objets, relations et diagrammes
- Diagrammes de classes
- Diagramme de cas
- Diagramme de séquences
- Diagramme de collaboration
- Diagramme d'état
- Diagramme d'activités

- UML : Unified Modeling Language est un langage unifié de modélisation objets.
- Unification : né (1997) de la fusion de 3 langages
  - OMT (Object Modeling Technique) : par Rumbaugh (Rational Software)
  - OOD (Object-Oriented Design) : par Booch (General Electric)
  - OOSE (Object-Oriented Software Engineering) : par Jacobson (Ericsson)

Définition et historique

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Cycle de vie logiciel : UML couvre toutes les phases du cycle de vie
- Indépendance : UML est indépendant du domaine d'application et des langages d'implantations
- Standard : intégré dans de nombreux outils de modélisation (GDPro, Rational Rose, TogetherSoft, Visio, Éclipse . . .)
- Format d'échange : Utilise XML pour échanger les modèles UML

Plan

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

2 UML
Introduction
Définition et historique

3 UML un langage
UML et les domaines d'utilisation
Vues en UML

4 Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

UML un langage

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- UML n'est pas une méthode ;
- UML est un langage de modélisation objet ;
- UML a été adopté par toutes les méthodes objet ;
- UML est dans le domaine public, c'est une norme.

UML et les domaines d'utilisation

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Systèmes d'information des entreprises ;
- Les Banques et les services financiers ;
- Télécommunications ;
- Transport ;
- Défense et aérospatiale ;
- Scientifique ;
- Applications distribuées par le WEB ;
- . . .

Les Démarches de modélisation

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- UML ne définit pas le processus d'élaboration des modèles
- Mais suggère 3 démarches :
  1. Itérative et incrémentale : L'idée est de concevoir un prototype et de l'améliorer.
  2. Besoins utilisateurs : Dans ce cas ce sont les utilisateurs qui guident la réalisation du modèle. Validation des différents livrables du modèle par les utilisateurs.
  3. Centré sur l'architecture : en utilisant les différentes vues d'UML

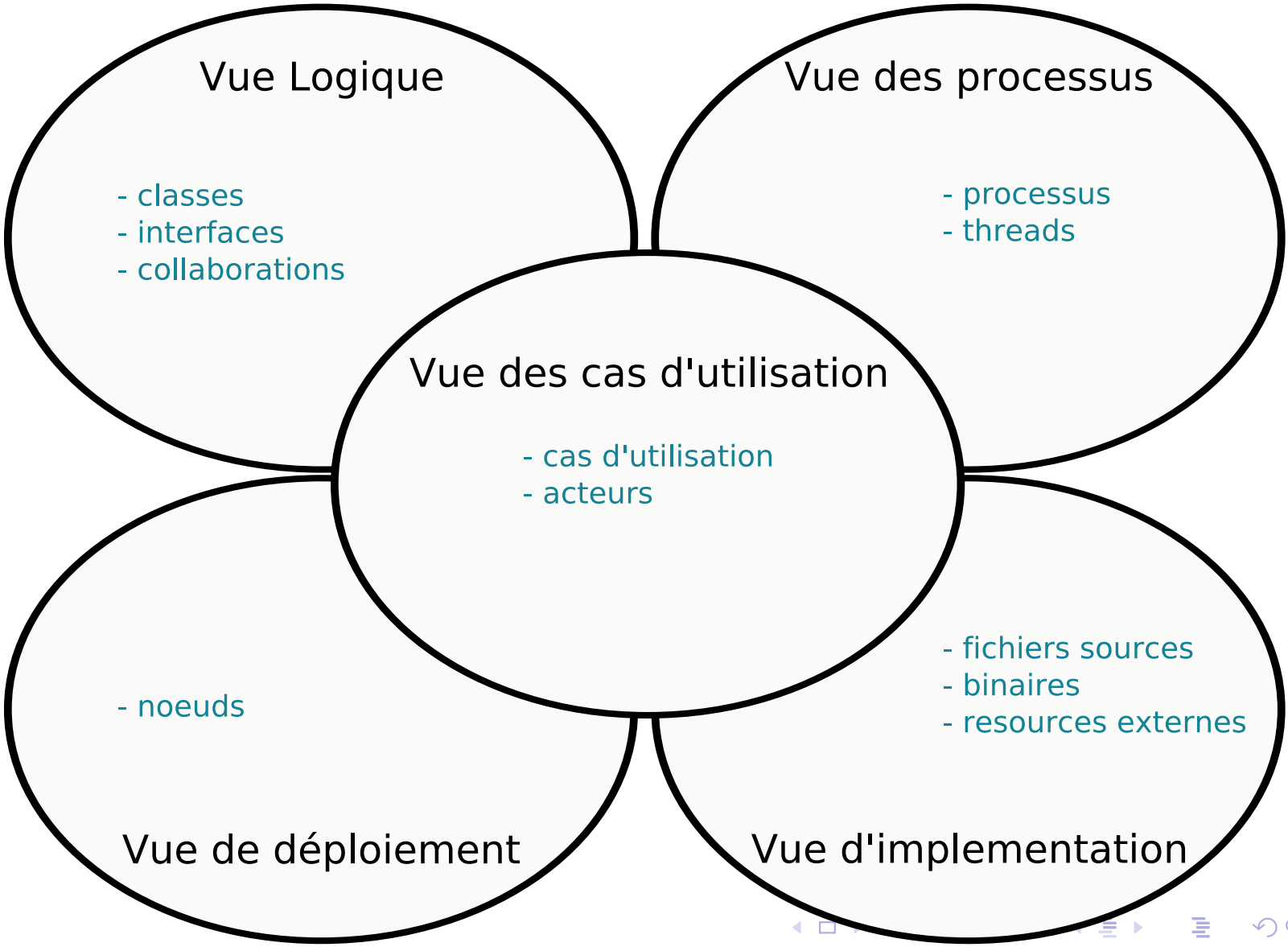
Vues en UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Les vues définissent le système.
- Ce sont des formulations du problème selon un certain point de vue.
- Elles peuvent se chevaucher pour compléter une description
- Leur somme représente le modèle en entier : 4 vues plus 1

Vues en UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités



Vues en UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

Les différentes vues sont :

- Vue des cas d'utilisation (qui, quoi) : Description du système vu par les acteurs du système.
- Vue logique (de conception) : elle modélise les éléments et mécanismes principaux du système. Les élément UML impliqués sont les classes, les interfaces, etc.
- Vue d'implantation : elle liste les différentes ressources du projets, fichiers binaires, bibliothèques, bases de données, etc.

- Elle établie le lien entre les composants.
- Elle permet d'établir des dépendances et de ranger les composants en modules.

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Vue de déploiement : Dans le cas de système distribué, elle définie les composants présents sur chaque noeud du système. C'est la vue spatiale du projet.
- Vue des processus : c'est la vue temporelle et technique, qui manipule les notions de tâches concurrentes, contrôle, synchronisation, processus, threads, etc.

Plan

UML
- Introduction
- Définition et historique

UML un langage
- UML et les domaines d'utilisation
- Vues en UML

Les diagrammes UML
- Entité, objets, relations et diagrammes
- Diagrammes de classes
- Diagramme de cas
- Diagramme de séquences
- Diagramme de collaboration
- Diagramme d'état
- Diagramme d'activités

2 UML
- Introduction
- Définition et historique

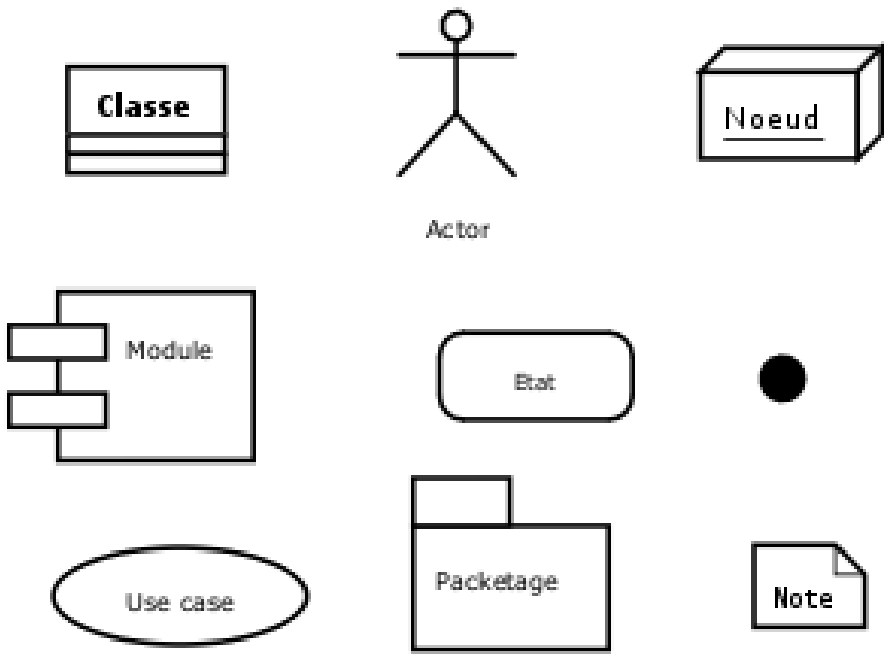
3 UML un langage
- UML et les domaines d'utilisation
- Vues en UML

4 Les diagrammes UML
- Entité, objets, relations et diagrammes
- Diagrammes de classes
- Diagramme de cas
- Diagramme de séquences
- Diagramme de collaboration
- Diagramme d'état
- Diagramme d'activités

Les entités UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Se sont les objets que manipule le formalisme : **Les objets communs :** (classe, objet, cas d'utilisation, paquetage, note, nœud, fourche, acteur, état, activité, état initial, état terminal, interface)



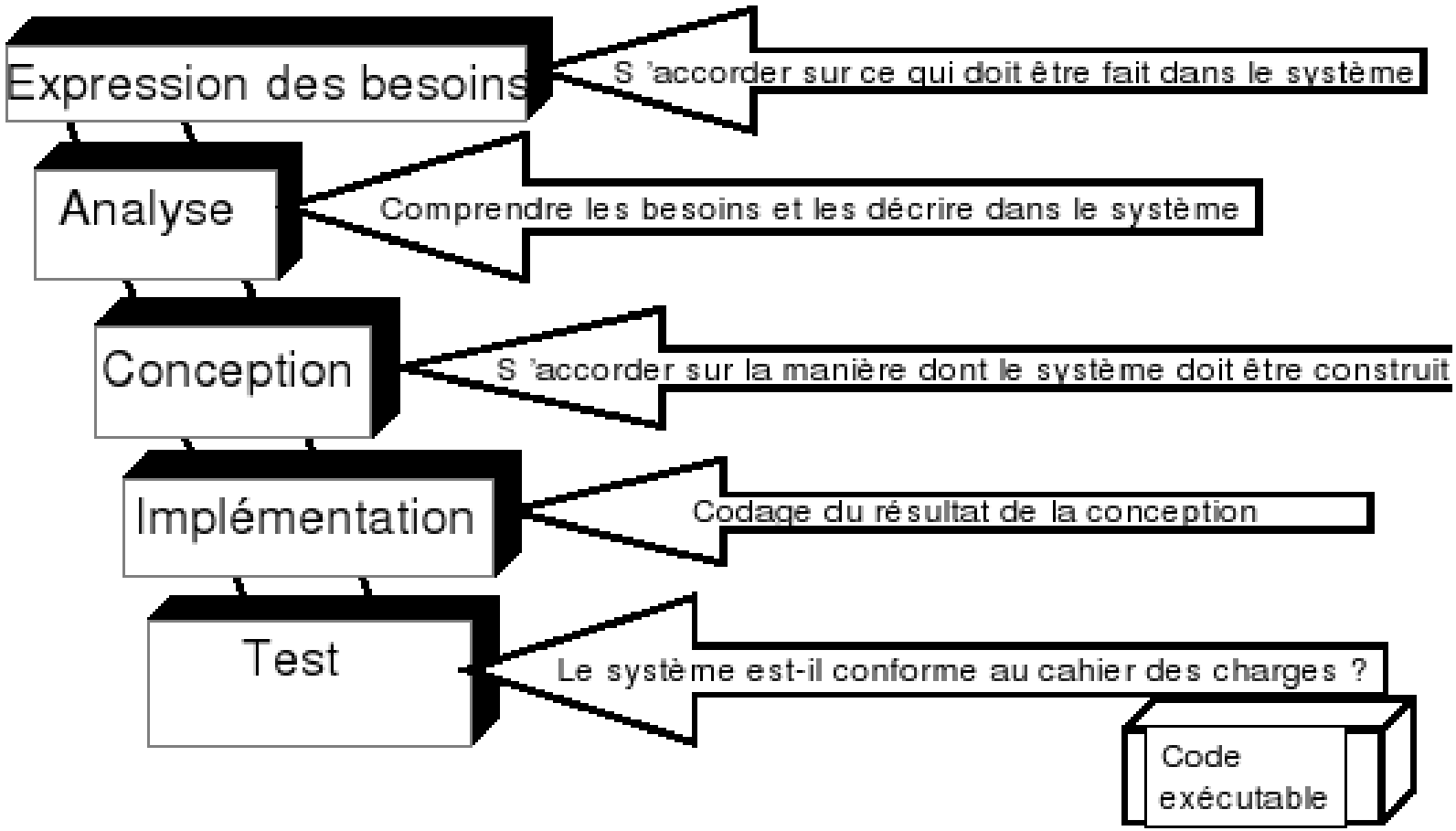
Les trois composantes d'UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Aspect fonctionnel $\Longrightarrow$ que fait le système (diagramme de cas d'utilisation)
- Aspect statique $\Longrightarrow$ sur quoi l'objet agit (diagramme diagramme de classes et d'objet)
- Aspect dynamique $\Longrightarrow$ séquencement des actions dans le système (diagramme de séquences, de collaboration, d'états-transitions et d'activités)

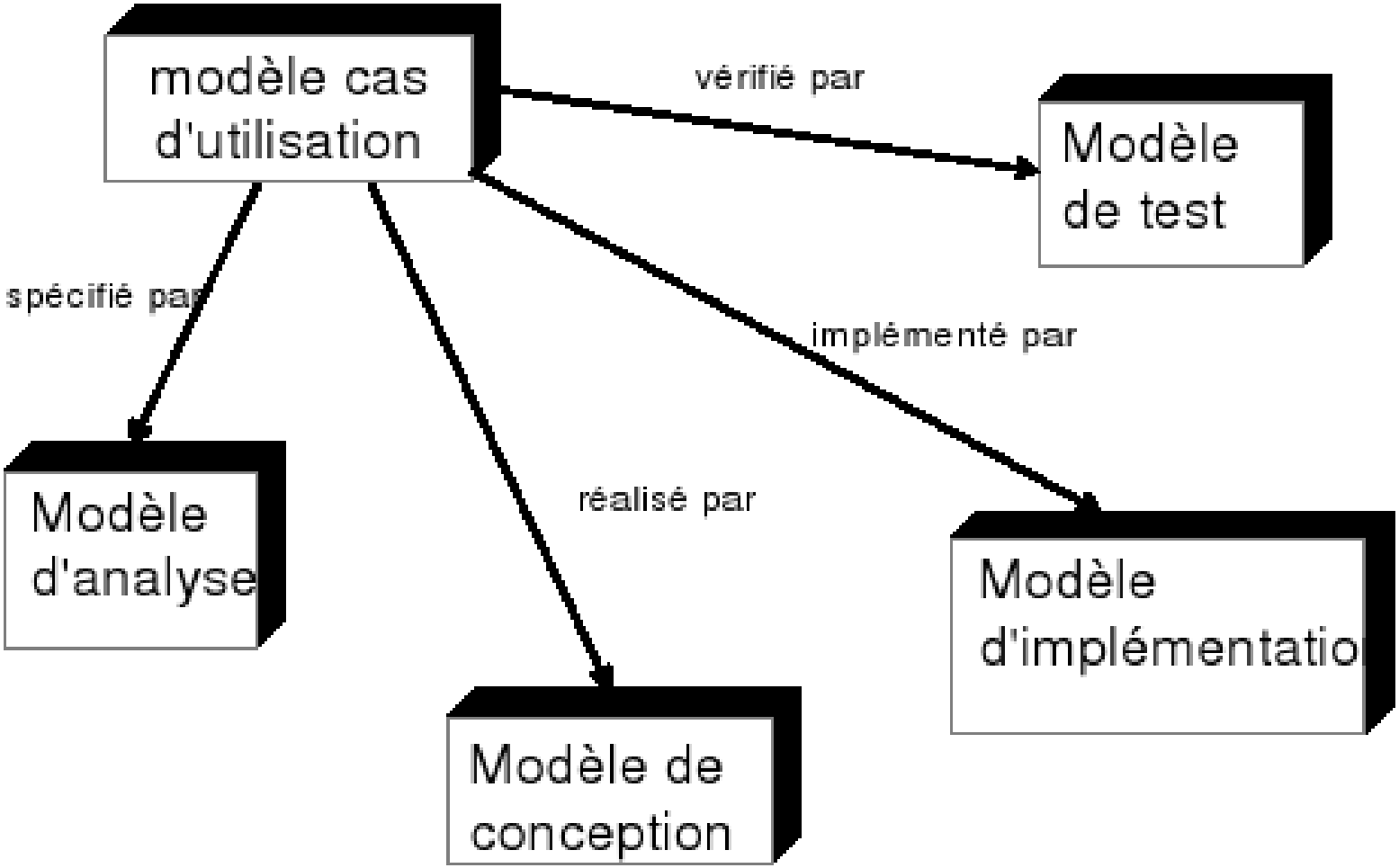
Vues en UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités



Vues en UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités



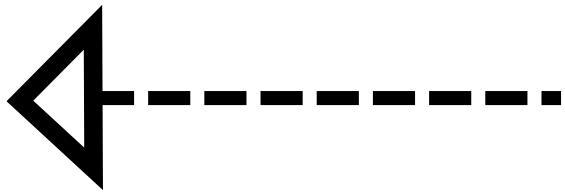
Les objets relationnels

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

Association

Dépendance

Réalisation



Agrégation

Héritage

Les objets relationnels

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Association : des objets interagissent par le biais d'appels à méthodes, d'envoi de message.
- Dépendance : s'exprime en "degré" de dépendance. S'applique aux bibliothèques et éléments logiciels. Un élément B est dépendent de A si tout changement de A peut affecter B.
- Réalisation (ou implantation) : un composant B réalise, implémente les fonctionnalités annoncées par A. A est une interface que B implémente.
- Héritage (ou généralisation) : B hérite de A, elle hérite des données membres et des méthodes de cette dernière.

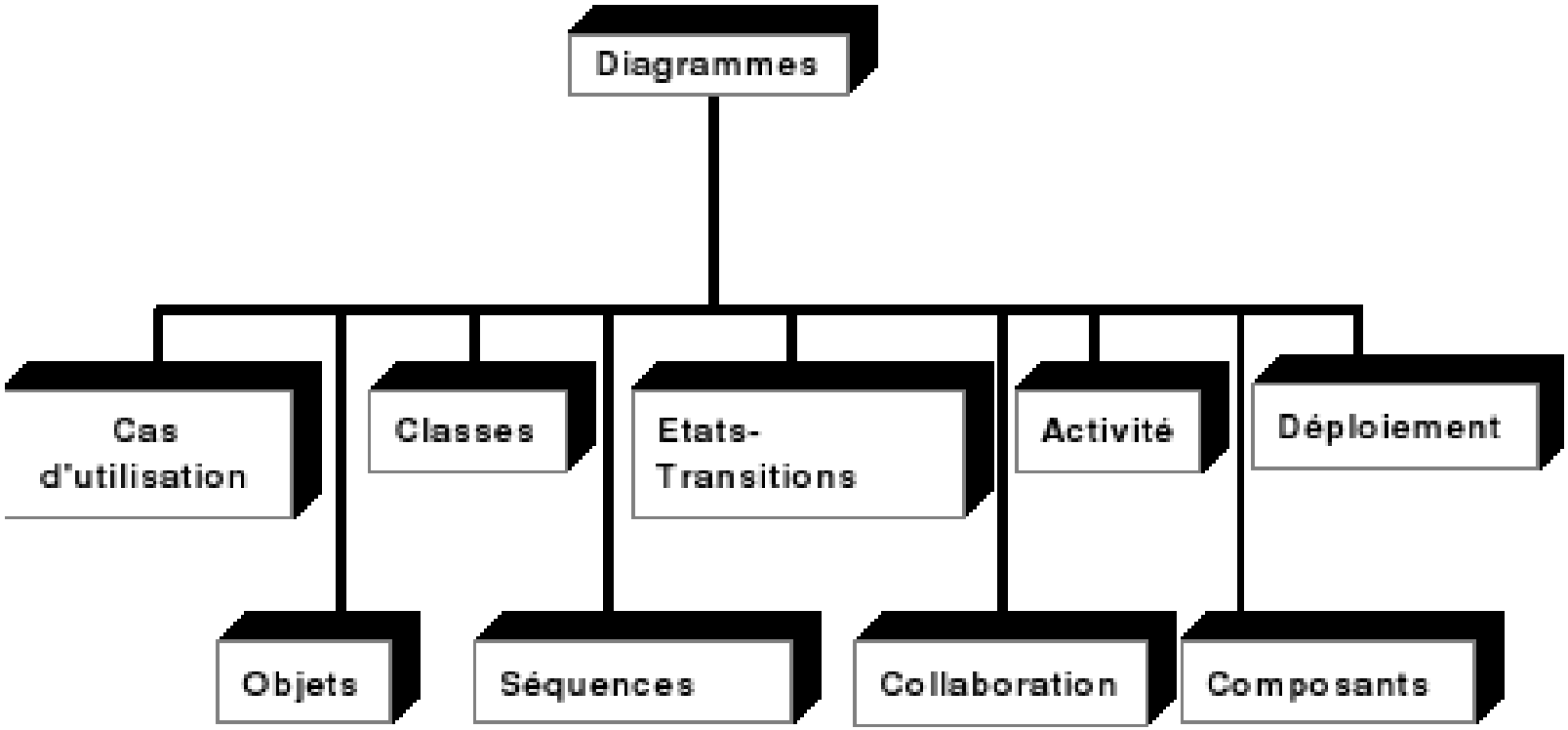
Les objets relationnels

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Composition : B est inclue dans A et en est indissociable. Si A disparait, B disparait aussi. Exemple : Une Université possède des filières (Info, Math, Philo...). Si l'université ferme ses portes les filières disparaissent.
- Agrégation : c'est l'inclusion non exclusive d'un composant dans un autre. Exemple : les filières de l'université ont des étudiants. Si une filière ferme, les étudiants ne "disparaissent" pas, ils changent de filière.

Les 9 diagrammes UML

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités



Diagrammes de classes

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

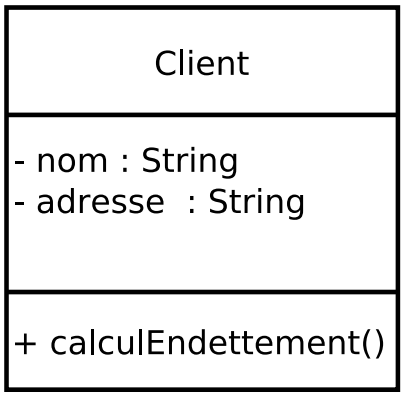
- Il est largement utilisé pour définir les types d'objets utilisés et leurs relations.
- Il modélise la structure et le contenu des classes.
- Il manipule des composants UML de type objet, classe, package.
- Il décrit les 3 vues logique, déploiement et implantation.

Classe

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

L'objet le plus manipulé ici est la classe :

- Une classe est composée des 3 élément identifient, attributs et méthodes.
- Les attributs et méthodes peuvent être + publics, - privés, # protégés, $\sim$ visible pour le paquetage.



Classe

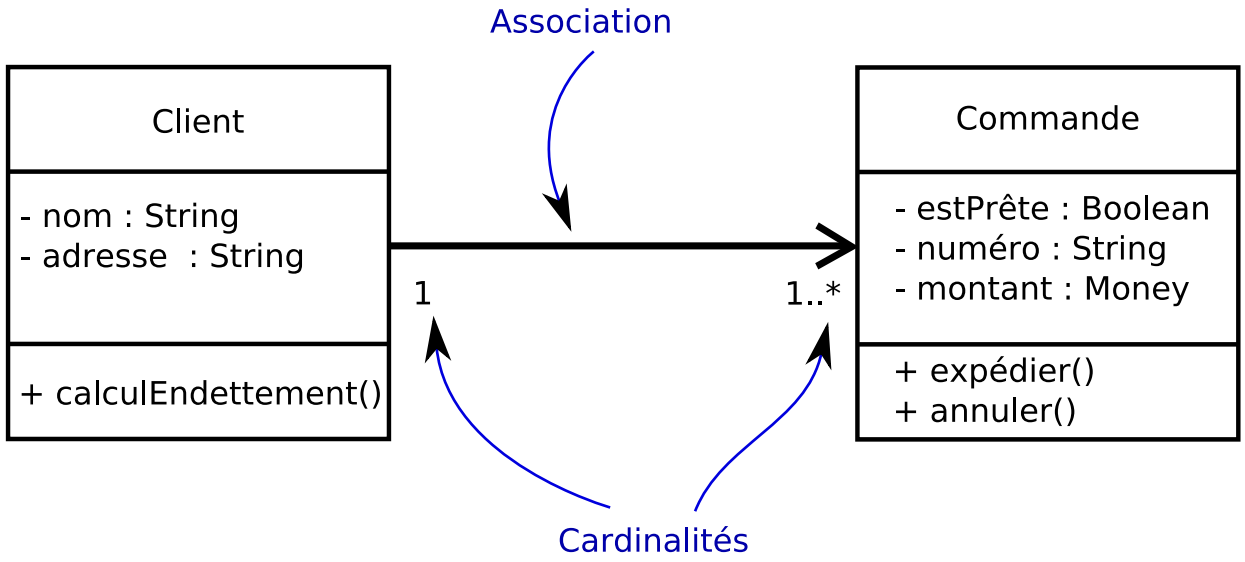
UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Il affiche des liens entre des classes tels que contenance, héritage ou association.
- Les associations peuvent faire interagir plusieurs instances de classes. On indique la cardinalité des instances participant à l'association :

Classe

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

0..1	0 à une instance
1	une instance exactement
*	0 ou plusieurs instances
1..*	une ou plusieurs instances



Héritage

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

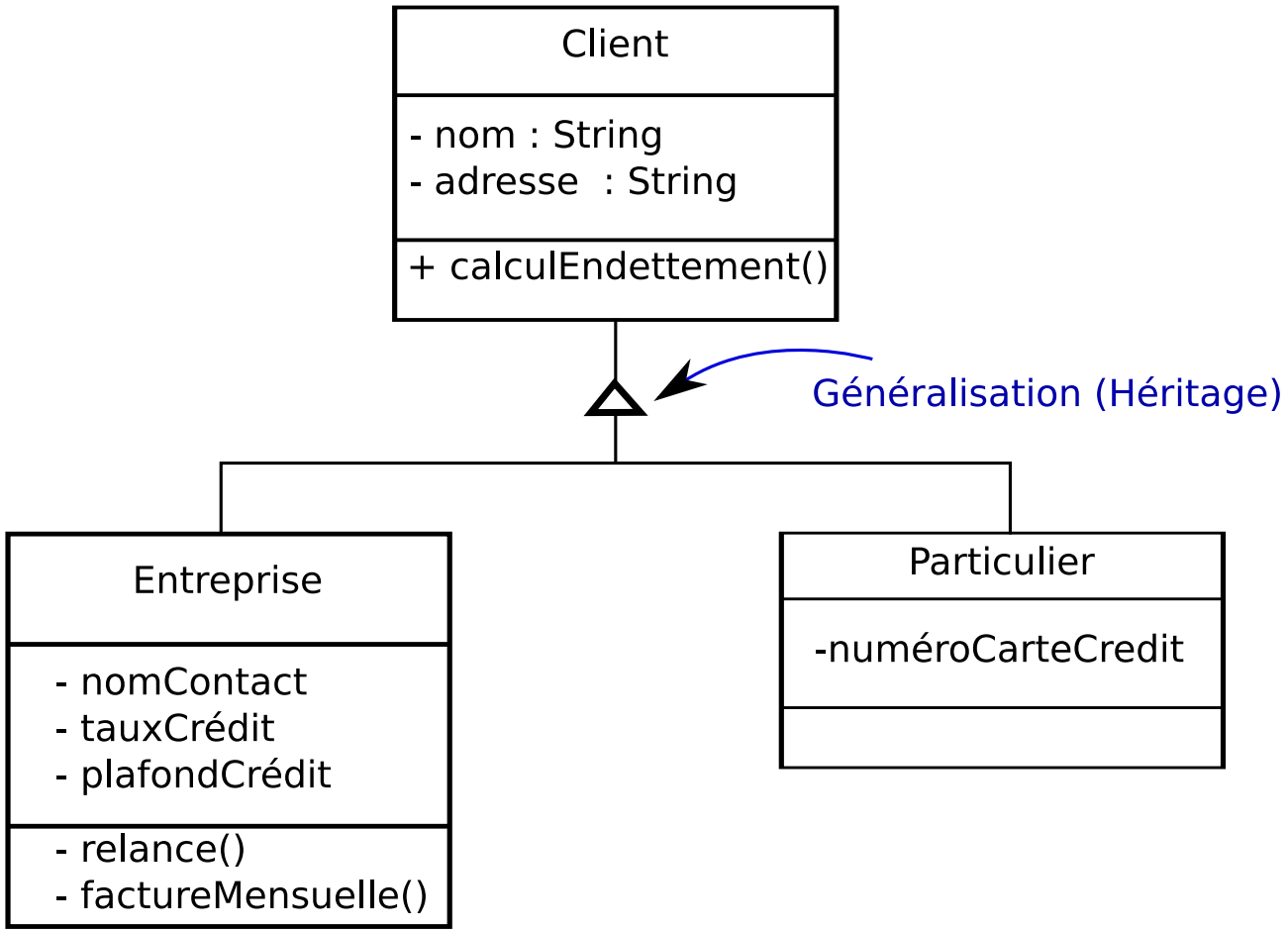


Diagramme de cas d'utilisation

UML
- Introduction
- Définition et historique

UML un langage
- UML et les domaines d'utilisation
- Vues en UML

Les diagrammes UML
- Entité, objets, relations et diagrammes
- Diagrammes de classes
- Diagramme de cas
- Diagramme de séquences
- Diagramme de collaboration
- Diagramme d'état
- Diagramme d'activités

- Les cas d'utilisation représentent les fonctionnalités que le système doit savoir faire.
- Chaque cas d'utilisation peut être complété par un ensemble d'interactions successives d'une entité en dehors du système (l'utilisateur) avec le système lui même.

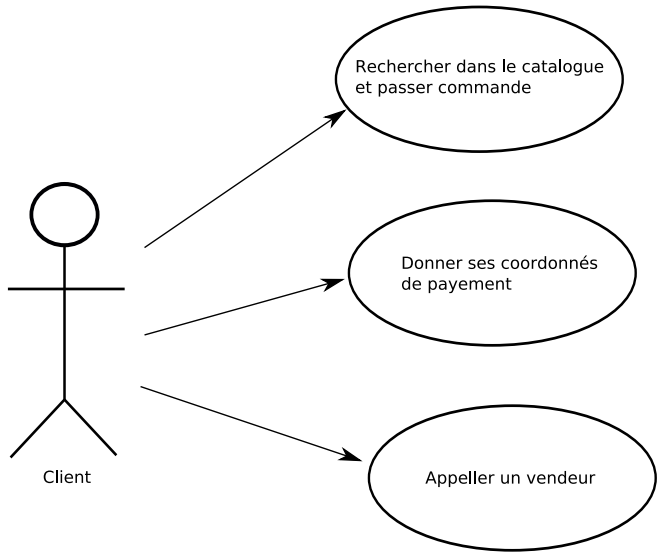


Diagramme de séquences

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- suite aux descriptions textuelles, le scénario peut être représenté en utilisant un diagramme de séquences.
- le diagramme de séquences permet :
  - de visualiser l'aspect temporel des interactions
  - de connaître le sens des interactions (acteur vers système ou contraire)

Diagramme de séquences

UML
- Introduction
- Définition et historique

UML un langage
- UML et les domaines d'utilisation
- Vues en UML

Les diagrammes UML
- Entité, objets, relations et diagrammes
- Diagrammes de classes
- Diagramme de cas
- Diagramme de séquences
- Diagramme de collaboration
- Diagramme d'état
- Diagramme d'activités

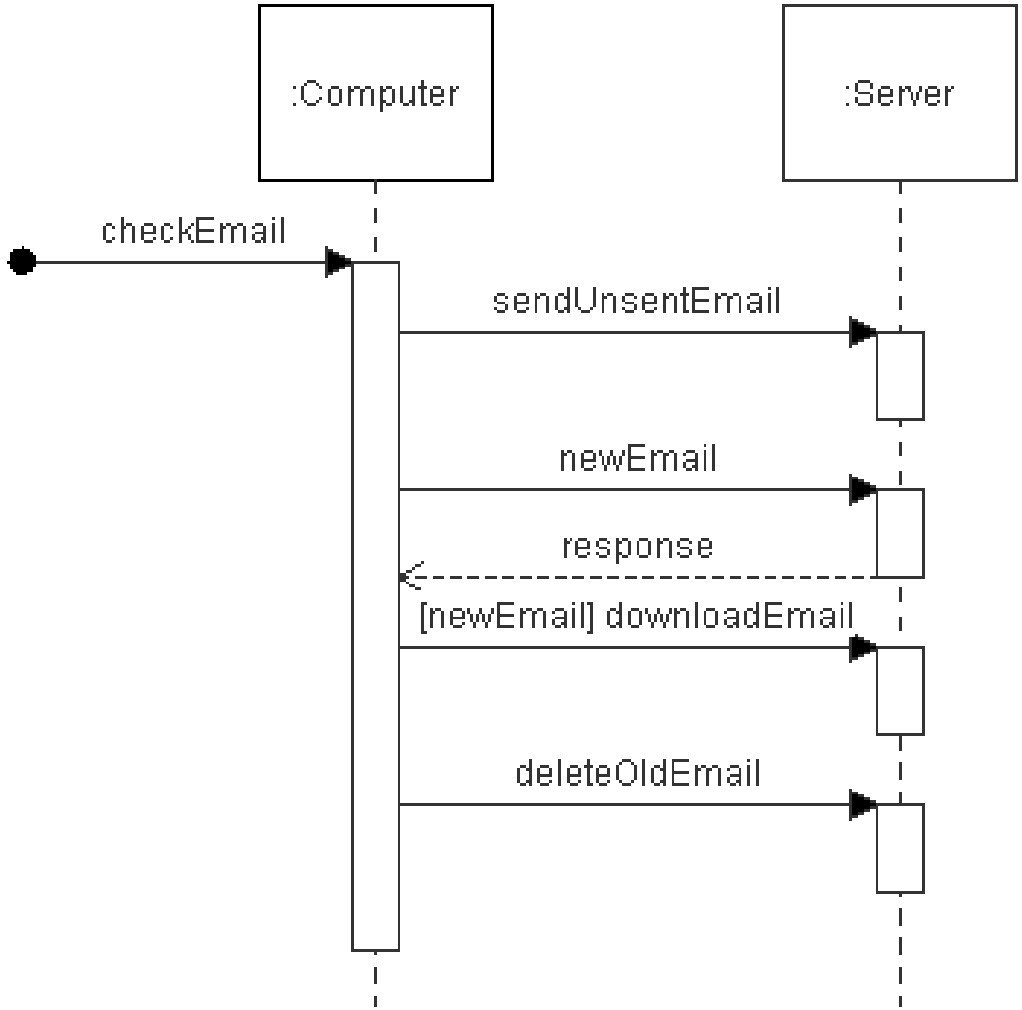


Diagramme de collaboration

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Le diagramme de collaborations sous une forme distincte du diagramme de séquences représente les interactions entre classes en mettant moins en évidence l'aspect temporel.
- Ce modèle explique la coopération entre objets pour la réalisation d'une fonctionnalité, cette coopération implique les différents événements qui se propagent d'un objet à un autre. Un objet doit avoir une méthode appropriée pour traiter chaque événement qu'il reçoit (message).

Diagramme de collaboration

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- L'aspect temporel n'est pas complètement caché car chaque message est numéroté.

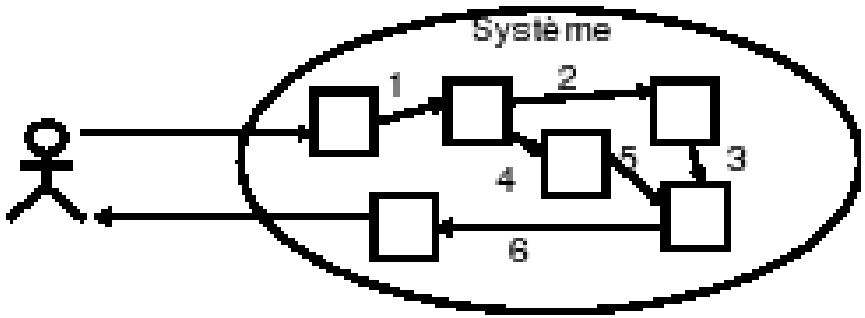


Diagramme d'état

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

- Il trace l'activité du système.
- Un objet à la fois est représenté.
- Chacun de ses états est décrit en fonctions des cas d'utilisation qu'il rencontre.

Diagramme d'état

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

Exemple : états d'un objet Commande

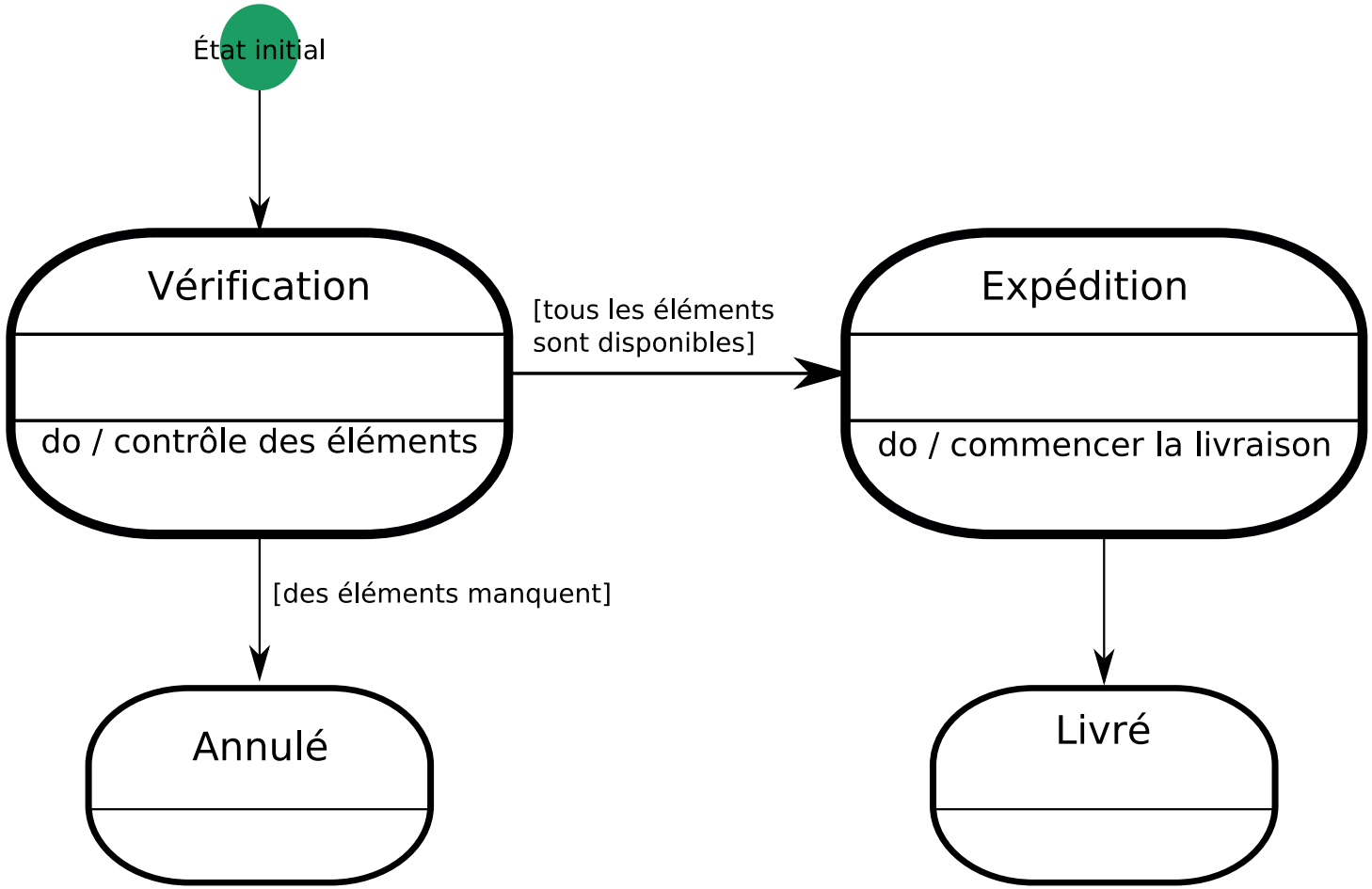


Diagramme d'activités

UML
Introduction
Définition et historique
UML un langage
UML et les domaines d'utilisation
Vues en UML
Les diagrammes UML
Entité, objets, relations et diagrammes
Diagrammes de classes
Diagramme de cas
Diagramme de séquences
Diagramme de collaboration
Diagramme d'état
Diagramme d'activités

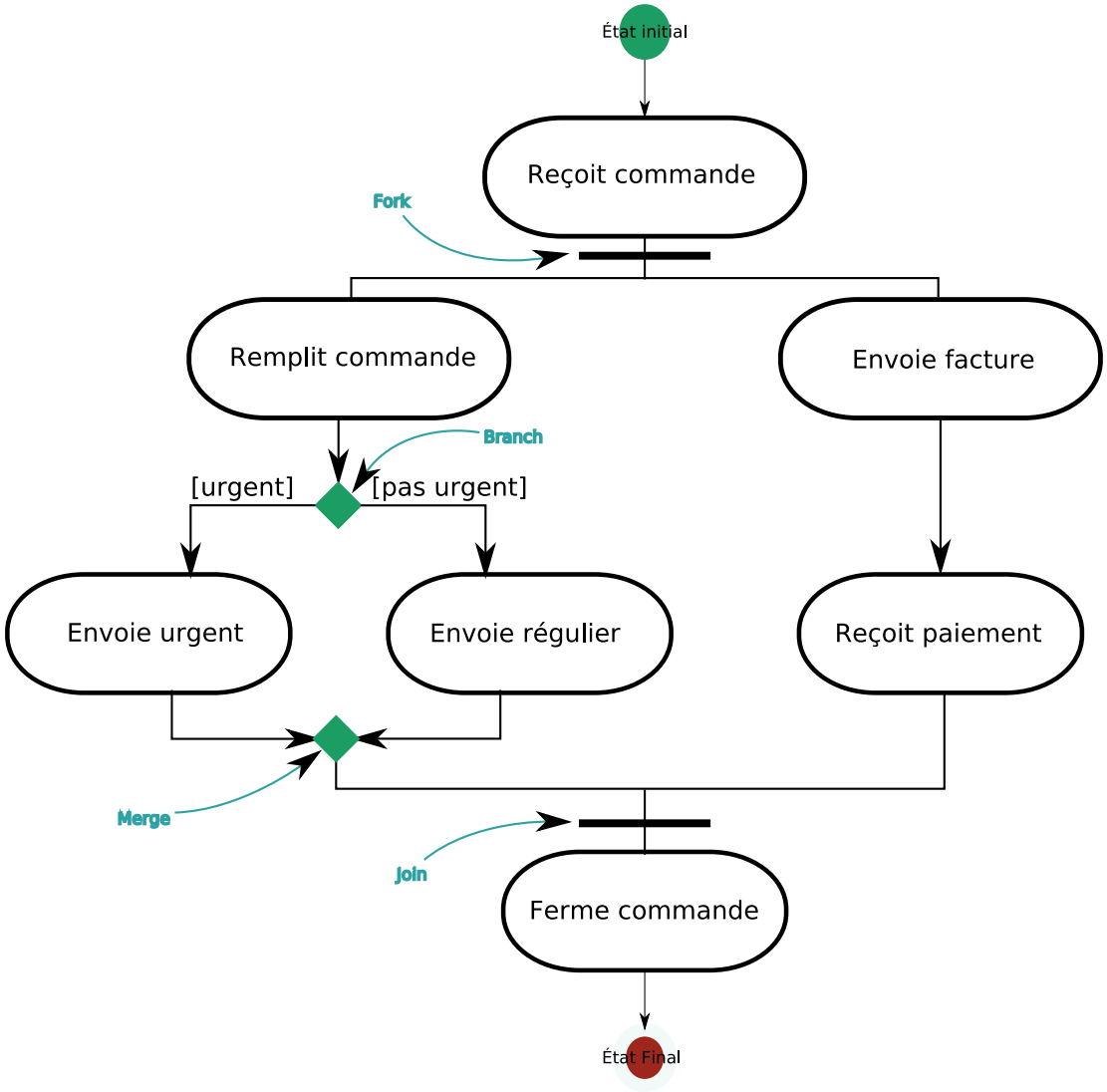
- UML permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation, à l'aide de diagrammes d'activités (une variante des diagrammes d'états-transitions).
- Une activité représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles.
- Le passage d'une activité vers une autre est matérialisé par une transition.
- Les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre (elles sont automatiques).

Digramme d'activités

UML
- Introduction
- Définition et historique

UML un langage
- UML et les domaines d'utilisation
- Vues en UML

Les diagrammes UML
- Entité, objets, relations et diagrammes
- Diagrammes de classes
- Diagramme de cas
- Diagramme de séquences
- Diagramme de collaboration
- Diagramme d'état
- Diagramme d'activités



Plan

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

5 Présentation du C++
Introduction
Quelques spécificités
LOO

6 Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction

7 Allocation dynamique
Organisation de la mémoire
Les opérateurs

Histoire

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- 1982 par Bjarne Stroustrup (AT&T laboratory) ;
- Objectif : greffer sur C les fonctionnalités des langages objets.

Qu'attend-on d'un programme ?

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Exactitude : aptitude d'un programme à fournir un résultat donné dans des conditions spécifiées.
- La robustesse : le programme doit bien réagir lorsqu'on s'éloigne des conditions normales d'utilisation.
- L'extensibilité : le programme doit pouvoir être modifié pour satisfaire aux évolutions des spécifications.
- La réutilisabilité : possibilité d'utiliser des parties d'un programme pour traiter d'autres problèmes.
- La portabilité : Un programme développé pour un processeur et un système d'exploitation doit pouvoir être utilisé sur un autre processeur et/ou un autre système d'exploitation.
- L'efficacité : Le code doit être suffisamment rapide.

Les solutions proposées

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- La programmation structurée
  Algorithmes + Structures de données = Programmes d'après N. Wirth (Pascal).
- La programmation structurée + les types abstraits de données.
- La programmation orientée objet (POO)

Qu'est-ce qu'un objet ?

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

C'est une association de :

- Données stockées dans des attributs ;
- Fonctions appelées méthodes qui agissent sur ces données.

Principe d'encapsulation

- Attributs non directement accessibles ... accès grâce à des méthodes pour consultation ou modification.
- L'appel d'une méthode correspond à un envoi de message à l'objet.
- Ces méthodes jouent un rôle d'interface.

Qu'est-ce qu'une classe ?

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Généralisation de la notion de type.
- Description d'un ensemble d'objets qui possèdent une structure de données identique et qui disposent des mêmes méthodes.
- Un objet est une instance de sa classe.
- Les données sont propres à chaque objet (en général).
- Les méthodes sont communes.
- Une classe peut être définie à l'aide d'une autre classe. Elle hérite des propriétés et des aptitudes de la première.

Incompatibilités C et C++

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Définition de fonctions ;
- Utilisation de `const` ;
- Conversions avec le pointeur générique `void *`.

$\Rightarrow$ Pour le reste, sauf précision contraire, le C++ contient le C.

Définition de fonctions

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- En C, vous pouvez utiliser une fonction qui n'a pas fait l'objet d'une définition ou d'un prototypage. La fonction retourne alors un `int`.
- En C++, toute fonction doit être définie avec la précision du type de ses arguments et de sa valeur de retour.
- Fonction sans argument :
  - En C++ : `float sansarg();`
  - En C : `float sansarg(void);`
- Une fonction qui ne retourne rien en C++ :

```
void sansretour(int x);
```

Utilisation de const

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Portée :
  En C et C++, `const` désigne un identicateur dont la valeur ne peut pas changer.
  - `const` appliqué à une variable locale, la portée est limitée au bloc dans lequel s'est effectuée la déclaration.
  - `const` appliqué à une variable globale, C++ limite la portée au fichier source.
  - Expression constante (expression dont la valeur peut être calculée à la compilation)
    - En C++, le compilateur sait évaluer :

```
const int MAX = 100;
double tab1[2*MAX+1], tab2[2*MAX+1][MAX];
```

    - En C, non ! ... en général. On doit utiliser `#define` :

```
#define MAX 100
double tab1[2*MAX+1], tab2[2*MAX+1][MAX];
```

Le type void * : pointeur générique

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- En C, `void *` est compatible avec tout pointeur, dans les deux sens de convertion.

```
void* gen; int* adi;
gen = adi; adi = gen;
```

- En C++, seule la convertion d'un pointeur quelconque vers `void *` est acceptée sans opérateur de cast :

```
gen = adi; \\acceptee
adi = gen; \\refusee
adi = (int*) gen; \\acceptee
```

Commentaires

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- La notation du C : commentaires entre /* et */ est toujours valide ;
- A celle-ci s'ajoute, les commentaires de fin de ligne qui démarrent par // et vont jusqu'à la fin de ligne.

```
int main()
{ int i=0; // i est une variable entiere
  ...
}
```

Déclaration

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- En C, obligation de regrouper toutes les déclarations au début du programme.
- En C++, ce n'est plus obligatoire. Elles peuvent apparaître n'importe où, avant d'être utilisée.

```
int main()
{ int i;
  i=3;
  ...
  int q=3*i;
  ...
  for (int j=0; j<q; j++) ...
}
```

Entrées/sorties

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

Gestion simplifiée des fonctions d'entrées/sortie grâce à 2 nouvelles fonctions :

- En saisie, cin :
  Exemple :

```
cin >> x;
```

- En affichage, cout :
  Exemple :

```
cout << "coucou";
cout << "voici le nombre : " << x << endl;
```

Remarques

- Les anciennes fonctions sont toujours utilisables ;
- L'opérateur de référence `&` n'est pas nécessaire pour la saisie ;
- Un processus de vérification automatique de type permet de s'affranchir des multiples formats de type très utilisés en C.

Langage Orienté Objet

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

Nouvelles possibilitées non particulières à un LOO

- Nouvelle forme de commentaire ;
- Liberté dans l'emplacement des déclarations ;
- Notion de référence ;
- Surcharge de fonction ;
- Allocation dynamique par les opérateurs `new` et `delete` ;
- Fonction inline.

Nouvelles possibilitées offertes par l'orientation objet

- Notion de classe ;
- Mécanisme d'instanciation (création d'objet) ;
- Constructeur/ Destructeur ;
- Fonction amies ;
- Surcharge d'opérateurs ;
- Redéfinition de conversions ;
- Héritage - héritage multiple ;
- Notion de flot.

Plan

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

5 Présentation du C++
- Introduction
- Quelques spécificités
- LOO

6 Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

7 Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

Définition

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Référence : nouveau type qui permet de manipuler un alias sur une autre variable existante.
- Toute action sur un alias est effectuée sur l'entité à laquelle l'alias fait référence.
- Notion de référence proche de la notion de pointeur
  - Une variable de type pointeur pointe sur une autre entité ;
  - Une variable de type référence fait référence à une autre entité.
- Référence = pointeur caché.

Déclaration et manipulation

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

Déclaration

- Une référence s'effectue toujours sur une entité d'un type donné.
- Une référence s'écrit : `type &` ;
- Exemple :
  - `int &` référence sur un `int` ;
  - `int* &` référence sur un `int*`.

Création, manipulation d'une référence

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Une variable de type référence doit toujours être initialisée lors de sa déclaration ;
- Exemple :

```
//Declaration incorrecte
int & nombre;
//Declaration correcte
int number;
int & nombre = number;
```

- Exemple développé : prog1.cc
- La trace d'exécution : traceprog1.txt

Création, manipulation d'une référence

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

- Adresse d'une information associée à une référence ;
- Exemple :
  - Exemple développé : prog2.cc
  - La trace d'exécution : traceprog2.txt

Création, manipulation d'une référence

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

Références constantes

- Lorsque la référence est constante il n'est pas possible de modifier l'information par l'intermédiaire de la référence.
  - ```
    int nombre;
    const int & number = nombre;

    //Interdit
    number++;
    //Autorise
    nombre++;
    ```
- Une référence sur une constante doit être constante.
  - ```
    //Interdit
    const int x = 0;
    int & ref_x = x;
    //Autorise
    const int x = 0;
    const int & ref_x = x;
    ```

Transmission par référence

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- En C
  - Les arguments et le retour d'une fonction sont transmis par valeur.
  - $\Rightarrow$ Pour modifier un paramètre appelant on transmet la valeur de son adresse. On manipule alors le pointeur au niveau de la fonction.
  - Exemple : prog3.cc
  - La trace d'exécution : traceprog3.txt
- En C++
  - Transmission par valeur ;
  - Transmission par référence. Les modifications portent sur le paramètre d'appel et non sur une copie.
  - Exemple : prog4.cc
  - La trace d'exécution : traceprog4.txt

Transmission par référence

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

- En C il est parfois nécessaire de transmettre un pointeur de pointeur (structure dynamique de type arbre par ex.) ;
- En C++ on utilise une référence sur le pointeur.
- Exemple prog5.cc

Transmission par référence

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

Références constantes

- Passage par référence constante, cas de la fonction `affiche()` et `print()` ;
- Le code : prog6.cc
- `print()` et `affiche()` fournissent le même résultat
  - Dans le cas de `print()` on passe la valeur de `S_tableau t` ce qui nécessite une recopie de `nb_elt` et de `tab`.
  - Dans le cas d'`affiche()` on transmet une référence constante de `t`.
    `t` ne peut pas être modifiée.

Les arguments par défaut d'une fonction

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Il est possible d'attribuer des valeurs par défaut aux arguments d'une fonction ;
- On peut appeler une fonction sans correspondance exacte avec le nombre d'arguments ;
- Ce sont les valeurs par défaut qui sont transmises.

Mise en œuvre

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Les valeurs par défaut sont fixées dans la déclaration (prototype) de la fonction et non dans sa définition (sauf si elles sont conjointes) ;
- Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés sont obligatoirement les derniers de la liste.
- Pour spécifier des arguments par défaut, il faut :
  - le type de l'argument formel ;
  - le nom de l'argument formel (optionnel) ;
  - =
  - Une expression ne contenant pas d'arguments formels.
- Exemples : prog7.cc, traceprog7.txt, prog8.cc, prog9.cc.

Argument par défaut et transmission par référence

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

- Si on utilise une référence comme valeur par défaut, la variable doit être globale.
  - Exemple incorrect : prog10.cc
  - Message d'erreur du compilateur : erreurprog10.txt
  - Une version correcte : prog10bis.cc
- Avec des pointeurs il est possible d'utiliser des variables locales (attention aux adresses "folles")
  - prog11.cc

Utilisation d'une structure comme valeur par défaut

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- On peut utiliser soit une valeur ;
- soit une référence avec la restriction déjà citée ;
- Exemple : prog12.cc ;
- Résultat : traceprog12.txt.

Expressions utilisées comme valeur par défaut

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Les arguments par défaut sont évalués à l'endroit de la déclaration.
- Quel est le résultat de ce programme ? : prog13.cc ;
- Il affiche 2.
- Quel est le résultat de ce programme ? : prog13bis.cc ;
- Il affiche 3.

Expressions utilisées comme valeur par défaut

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Les arguments par défaut sont évalués à l'endroit de la déclaration.
- Quel est le résultat de ce programme ? : prog13.cc ;
- Il affiche 2.
- Quel est le résultat de ce programme ? : prog13bis.cc ;
- Il affiche 3.

Expressions utilisées comme valeur par défaut

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

- Les arguments par défaut sont évalués à l'endroit de la déclaration.
- Quel est le résultat de ce programme ? : prog13.cc ;
- Il affiche 2.
- Quel est le résultat de ce programme ? : prog13bis.cc ;
- Il affiche 3.

Expressions utilisées comme valeur par défaut

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Les arguments par défaut sont évalués à l'endroit de la déclaration.
- Quel est le résultat de ce programme ? : prog13.cc ;
- Il affiche 2.
- Quel est le résultat de ce programme ? : prog13bis.cc ;
- Il affiche 3.

Expressions utilisées comme valeur par défaut

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Les arguments par défaut sont évalués à l'endroit de la déclaration.
- Quel est le résultat de ce programme ? : prog13.cc ;
- Il affiche 2.
- Quel est le résultat de ce programme ? : prog13bis.cc ;
- Il affiche 3.

Pointeur sur une fonction comme paramètre par défaut

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

- Il est possible de passer des pointeurs sur des fonctions ;
- On peut donc définir une fonction par défaut ;
- Exemple : prog14.cc ;
- Résultat : traceprog14.txt.

Fonctions renvoyant une référence

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

Une fonction peut renvoyer une référence sur une variable, comme illustré sur l'exemple suivant :

```
const int L=3, C=2;
int& place (int tab[L][C], int);
void ecrire (int tab[L][C]);
int main()
{ int t[L][C];
  for (int i=0; i<L; i++)
    for (int j=0; j<C; j++)
      t[i][j] = i + 2*j;
  ecrire (t);
  place (t,3) = 0;
  ecrire (t);
  return 0;
}
```

```
int& place (int tab[L][C], int val)
{ for (int i=0; i<L; i++)
    for (int j=0; j<C; j++)
      if (tab[i][j] == val)
        return tab[i][j];
  return tab[0][0];
}
void ecrire (int tab[L][C])
{ for (int i=0; i<L; i++)
  { for (int j=0; j<C; j++)
      cout << tab[i][j] << "/";
    cout << endl;
  }
}
```

Résultats affichés

```
0/2
1/3
2/4

0/2
1/0
2/4
```

Surcharge de fonction

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Plusieurs fonctions peuvent avoir le même identificateur ;
- Le compilateur identifie la fonction appelée grace aux types des arguments ;
- $\Rightarrow$ Les types doivent être discriminants ;
- Exemple : prog15.cc ;
- Résultat : traceprog15.txt.
- On verra par la suite que l'on peut surcharger les opérateurs en C++.

Plan

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

5 Présentation du C++
- Introduction
- Quelques spécificités
- LOO

6 Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

7 Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

Organisation de la mémoire

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

La mémoire attachée à un programme qui s'exécute (processus) est divisée en plusieurs zones qui correspondent à la nature des entités qui y sont stockées.

- Une zone pour stocker le code.
- Une zone de mémoire statique dans laquelle on trouve les variables statiques et les variables globales (durée de vie = celle du processus).
- La pile qui sert à stocker les variables automatiques.Dans cette zone les variables sont constamment créées et détruites en fonction de la durée de vie.
- Le tas c'est dans cette zone que l'on réserve un emplacement mémoire. On connaît cet emplacement par son adresse et non par un identificateur. L'emplacement est réservé tant qu'il n'a pas été désalloué. On utilise `new` et `delete` pour gérer cet emplacement.

L'opérateur `new`

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

- `new type` où `type` représente un type quelconque ;
- Résultat : pointeur de type `type *` sur la zone allouée ;
- ... ou pointeur nul en cas d'échec ;
- On peut allouer de la place mémoire pour `n` éléments
  - `new type[n] n` est une expression entière quelconque ;
  - Le pointeur fourni pointe sur le 1^{er} élément.
- On peut initialiser en même temps que l'allocation

```
int *ptr_int = new int(-56);
```

Quelles est la nature de ces pointeurs ?

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs


```
char *ptr1 = new char;

int *ptr2 = new int;

double *ptr3 = new double;

float **ptr4 = new float *;


int (** ptr5)() = new (int (*)());

typedef int (* T)();
T *ptr6 = new T ;
```

```
char *ptr7 = new char[1];

float *ptr8 = new float[5];

char ** ptr9[5] = new char * [5];


int (** ptr10)() = new(int (* [5])());

T *ptr11 = new T[5];
```

Quelles est la nature de ces pointeurs ?
Les réponses

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs


```
//Un caractere
char *ptr_char = new char;
//Un entier
int *ptr_int = new int;
//Un double
double *ptr_double = new double;
//Un pointeur sur un float
float **ptr_ptr_float = new float *;
//Un pointeur sur une fonction sans argument
// qui retourne un int
int (** ptr_fonc)() = new (int (*)());
// ou encore
typedef int (* T_ptr_surfonc)();
T_ptr_surfonc *ptr_fonc = new T_ptr_surfonc ;
```

```
//Tableau de 1 caractere
char *tabcar = new char[1];
//Tableau de 5 floats
float *tabfloat = new float[5];
//Tableau de 5 pointeurs sur des char
char ** tabptrchar[5] = new char * [5];
//Tableau de 5 pointeurs sur des fcts sans arguments
//qui retournent un int
int (** tabfonc)() = new(int (* [5])());
// ou encore
T_ptr_surfonc *tabfonc = new Tptr_surfonc[5];
```

L'opérateur `delete`

Présentation du C++
- Introduction
- Quelques spécificités
- LOO

Notion de référence
- Définition
- Déclaration et initialisation
- Transmission par référence
- Les modes de transmission
- Fonctions renvoyant une référence
- Surcharge de fonction

Allocation dynamique
- Organisation de la mémoire
- Les opérateurs

- `delete adresse` désalloue la zone mémoire allouée à l'emplacement `adresse`
- Exemple

```
int *ptr  = new int;
//...
delete ptr;
```

- Pour désallouer un tableau :
  `delete adresse`
- Pour désallouer un tableau d'objets :
  `delete [] adresse`
- Exemple

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

Exemple d'utilisation de `new` et `delete`

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Dans cet exemple on construit un tableau de chaînes de caractères ;
- Le tableau est alloué dynamiquement ;
- Les chaînes également ;
- A la fin du programme on désalloue.
- Exemple : prog18.cc ;
- Résultat : traceprog18.txt.

Les tableaux à plusieurs dimensions

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

- Un tableau de dimension `n` c'est un tableau de dimension 1 dont les éléments sont de dimension `n - 1` ;
- Exemple pour un tableau de dimension 2 :

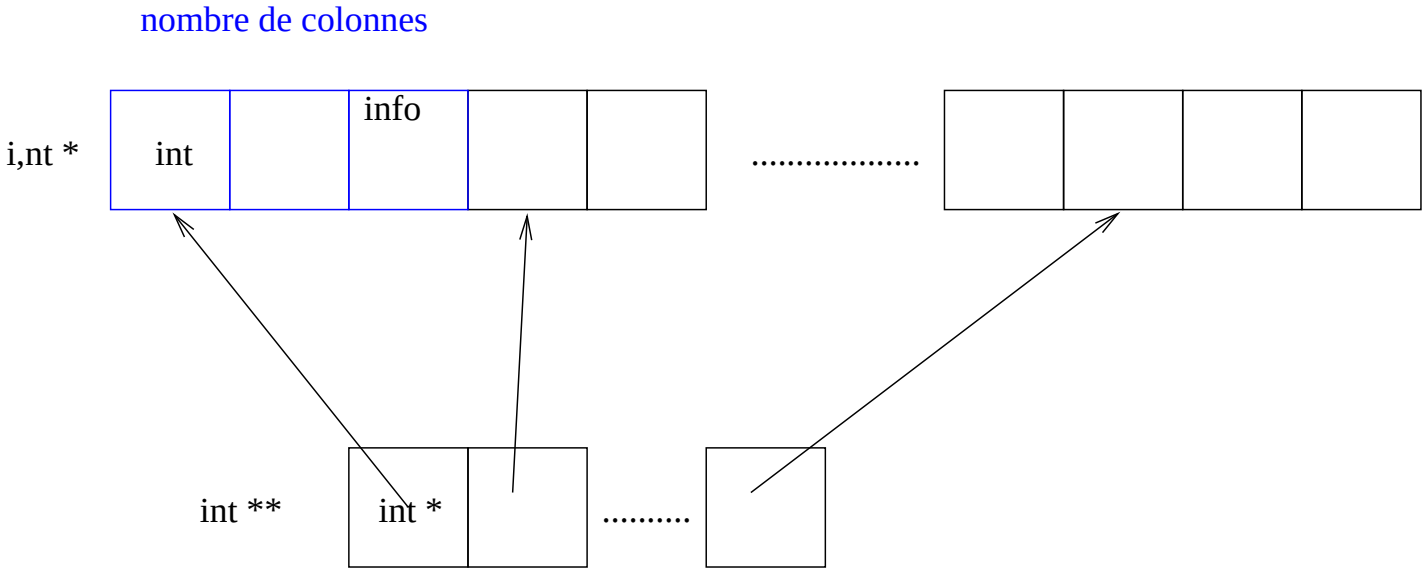
```
int ligne = 5; const int NB_COL = 3;
int (* ptr)[NB_COL] = new int[ligne][NB_COL];
```

  Les éléments sont des tableaux d'entiers, `ptr` est donc un pointeur sur un tableau de `NB_COL` entiers ;
- Toutes les dimensions doivent être connue sauf eventuellement la première.
- Pour libérer la zone : `delete [] ptr;`
- Exemple : prog19.cc ;

Les tableaux à plusieurs dimensions

Présentation du C++
Introduction
Quelques spécificités
LOO
Notion de référence
Définition
Déclaration et initialisation
Transmission par référence
Les modes de transmission
Fonctions renvoyant une référence
Surcharge de fonction
Allocation dynamique
Organisation de la mémoire
Les opérateurs

Pour créer un tableau de dimension 2 dont on donne à l'exécution les deux dimensions. On peut procéder de la façon suivante :



Exemple : prog20.cc.

Plan

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

8 Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice

9 Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie

10 Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Définition d'une classe

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Une classe permet de définir un nouveau type de données et des primitives de traitement associées. Elle contient des membres de 2 natures différentes :

- des attributs (données membres) ;
- des méthodes (fonctions membres).

Visibilité des membres

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Les membres d'une classe peuvent être déclarés avec les qualificatifs suivants :

- private (valeur par défaut) : non accessibles en dehors de la classe
- public : accessibles en dehors de la classe, ces membres constituent l'interface de la classe.
- protected : concerne la notion d'héritage et les classes dérivées (cf. suite)

Exemple

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Voici la première version de notre fichier `Vecteur.h`, en-tête de la classe `Vecteur` :

```
class Vecteur
{
  int taille;
  float* valeur;
 public :
  void initialise (int, float);
  void ajoute (int, float);
  void affiche ();
};
```

Voici maintenant le fichier `Vecteur.cpp` qui contient l'implémentation des méthodes définies dans le fichier d'en-tête (on notera la manière de désigner les méthodes rattachées aux classes) :

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

void Vecteur::initialise (int a, float b)
{ taille = a;
  valeur = new float[taille];
  for (int i=0; i < taille; i++) valeur[i] = b;
}
void Vecteur::affiche()
{ for (int i=0; i < taille; i++)
  cout << valeur[i] << ' ';
  cout << endl;
}

void Vecteur::ajoute(int i, float a)
{ valeur[i] += a; }

int main()
{ Vecteur v;
  v.initialise (10, 3.0); v.affiche();
  v.ajoute (3, 2.0); v.affiche();
}
```

Remarque : Dans la construction finale de la classe Vecteur, ces méthodes n'existeront pas ou seront traitées différemment.

Le pointeur `this`

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Exemple

- Chaque classe possède implicitement (sans que l'on ait besoin de la déclarer) la donnée qui correspond au pointeur `this` qui pointe sur l'objet courant.
- Exemple : on ajoute la méthode suivante à la classe vecteur :

```
int Vecteur::est_egal(Vecteur p)
{ return (this == &p);}
```

Attention :

- Cette méthode ne teste pas si l'objet `p` a les mêmes valeurs de composantes que l'objet courant !
- Cette méthode vérifie que ces deux objets ont la même adresse.

Membres `static`

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Un membre peut être déclaré `static`.
- Il est alors commun à tous les objets de la classe.
- Si un objet le modifie, il le sera donc pour tous les autres objets de la classe.

Exemple : dans la classe `Vecteur`, on ajoute un membre

```
static int nb_objet;
```

qui va compter le nombre d'objets instanciés de la classe :

- On l'initialise en dehors de la déclaration de classe par

  ```
  int Vecteur::nb_objet=0;
  ```

  Remarque : on préfixe l'attribut par le nom de la classe et pas par le nom d'un objet instancié. Par défaut une variable `static` est initialisée à 0.
- On l'incrémente dans la procédure initialise par l'instruction

  ```
  Vecteur::nb_objet++;
  ```

Constructeurs/destructeurs

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- A l'instanciation d'un objet, une fonction membre appelée *constructeur* est invoquée automatiquement.
- Lorsque l'objet n'existe plus, une fonction membre appelée *destructeur* est invoquée automatiquement.
- On peut redéfinir soi-même un ou des constructeurs ou destructeurs.
- Un constructeur (resp. destructeur) est donc une focntion membre `public` qui porte obligatoirement le nom de la classe.

Constructeurs - exemple 1

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Dans la classe `Vecteur`, la fonction `initialise` peut être avantageusement remplacée par un constructeur :

```
class Vecteur
{   int taille;
    float* valeur;
  public:
    Vecteur(int, float);
    void ajoute(int, float);
    void affiche();
}
```

```
Vecteur::Vecteur(int a, float b) // Le constructeur
{ taille = a;
  valeur = new float [taille];
  for (int i=0; i<taille; i++) valeur[i]=b;
}
...
int main()
{ Vecteur v(10, 3.0); // Appel au constructeur
  v.affiche();
  return 0;
}
```

Constructeurs - affectation automatique

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Dans la défintion d'un constructeur, on peut faire suivre l'en-tête par des affectations d'attributs comme ci-dessous :

```
Vecteur::Vecteur(int a, float b):taille(a)
{ ...}
```

Remarque : il peut y avoir plusieurs affectations successives, séparées par des virgules. Cela permet en particulier d'initialiser des attributs constants.

Constructeurs surchargés constructeur sans paramètre

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Il est courant de définir plusieurs constructeurs avec des paramètres différents (en nombre ou en type) : il s'agit de surcharge.
- Par exemple, on pourrait ajouter à la classe `Vecteur` un constructeur sans paramètre. L'instantiation invoquera la classe sans mettre de paranthèse.

```
Vecteur::Vecteur()
{ taille=0;
  valeur = new float;
}
int main()
{ Vecteur p; ...}
```

Destructeur

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

Dans la classe `Vecteur`, on va définir un destructeur qui libère la zone mémoire allouée par le constructeur. Il s'execute à la fin du programme ou d'un bloc où des objets locaux ont été définis.

```
class Vecteur
{  int taille;
   float* valeur;
  public :
   Vecteur (int, float);
   ~Vecteur();
}

Vecteur::~Vecteur()
{ delete valeur; }
```

Constructeur de copie

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Dans les cas suivants :

- l'initialisation d'un objet par un autre,
- la transmission d'un objet en tant que paramètre d'une fonction,

on appelle implicitement une méthode prédéfinie qui est en fait un *constructeur de copie*.

Exemple d'invocations
Implicites

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

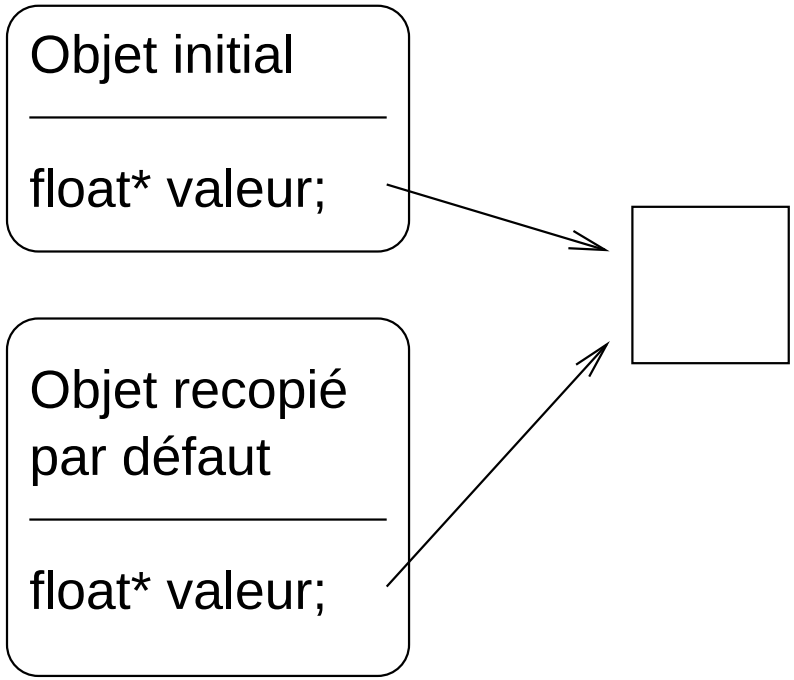

```
void f(Vecteur q)
{ ...}
int main()
{ Vecteur p1, p2(p1), p3=p1;
  ...
  f(p1);
}
```

- p2 et p3 sont des recopies de p1 ;
- q, le vecteur local à l'exécution de la fonction f, est une recopie de p1.

Création du constructeur de copie

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- On peut créer soi-même son propre constructeur de copie.
- C'est nécessaire lorsque l'objet contient des pointeurs et des structures dynamiques.
- Par exemple, pour la classe vecteur, le constructeur de recopie (par défaut) va uniquement recopier l'adresse du tableau pointé par le vecteur (voir figure).



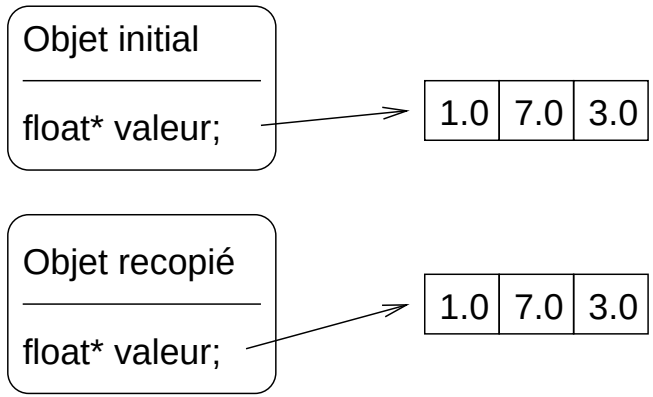
Création du constructeur de copie

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

En raison du destructeur écrit précédemment, on peut rencontrer le problème suivant :

- Si la copie concerne une transmission de paramètre,
- à l'appel de la fonction, un vecteur local est créé,
- son attribut `valeur` pointe sur la même adresse que l'objet initial.
- A la sortie de la fonction, le destructeur de ce vecteur local est invoqué
- et provoque la désallocation de la zone mémoire pointée par `valeur`
- et donc celle qui correspond aussi à l'objet initial !

Il est nécessaire d'écrire soi-même son constructeur de copie qui fera une nouvelle allocation pour le vecteur recopié. On doit alors recopier les valeurs pointées dans le nouvel emplacement (voir figure).



Syntaxe

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Un constructeur de recopie devra obligatoirement être défini par le prototype suivant : `Vecteur (Vecteur &)` ou encore `Vecteur (const Vecteur &)` car il n'y a pas lieu de modifier le paramètre transmis.

```
#include <iostream>
using namespace std;
class Vecteur
{  int taille;
   float* valeur;
  public :
   Vecteur(int n=0) //constructeur ...
            //sous sa forme finale
   { valeur = new float [taille=n];}
   Vecteur (const Vecteur & v) // const. par recopie
   { valeur = new float [taille=v.taille];
     for (int i=0; i<taille; i++)
      valeur [i] = v.valeur[i];
   }

   ~Vecteur() { delete valeur;}
};
```

Remarque : Le corps des fonctions est directement écrit dans la déclaration de la classe. Il s'agit de fonctions *inline*.

Surcharge d'opérateurs

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Les opérateurs classiques (+,-,/,*) peuvent être surchargés pour s'appliquer à des objets d'une classe donnée.
- Dans la classe Vecteur, on va définir l'opérateur d'addition qui permettra d'écrire une expression de la forme `v1 + v2`, où `v1` et `v2` sont des vecteurs.
- Pour réaliser cela, on ajoute une méthode à la classe vecteur qui s'appelle `operator+`.`v1 + v2` doit donc être interprété sous la forme `v1.operator+(v2)`.

```
class Vecteur
{ ...
  Vecteur operator+ (const Vecteur &);
  ...
}
Vecteur Vecteur::operator+ (const Vecteur &v)
// on suppose que la taille de v est
// identique au vecteur courant
{ Vecteur result(taille);
  for (int i=0; i<taille; i++)
    result.valeur[i]=valeur[i]+v.valeur[i];
  return result;
}
```

Remarques

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

1. Les paramètres sont transmis par référence (constante) pour éviter les recopies des tableaux.
2. Attention de ne pas renvoyer une référence de vecteur :
   `Vecteur & Vecteur : :operator+ (const Vecteur &v)`
   car on renvoie ici l'adresse du vecteur local `result` qui est désalloué en fin d'exécution !
3. On peut aussi définir l'opérateur + en dehors de la classe `Vecteur`, pour une même utilisation

```
Vecteur operator+ (const Vecteur &v1,
                   const Vecteur &v2)
// on suppose que les tailles de v1 et
// de v2 identiques
{ Vecteur result(v1.taille);
  for (int i=0; i<v1.taille; i++)
    result.valeur[i]=v1.valeur[i]+v2.valeur[i];
  return result;
}
```

On suppose ici que les attributs des vecteurs sont publiques, sinon on utilise des opérateurs d'accès, comme décrit plus loin.

Surcharge de l'opérateur d'affectation =

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Par défaut, on peut faire des affectations entre 2 objets (de même nature). Un opérateur d'affectation implicite est construit pour chaque classe.
- Comme pour le constructeur de recopie existant par défaut, l'opérateur d'affectation peut poser des problèmes avec les structures dynamiques.
- Dans la classe `Vecteur`, si on écrit

```
Vecteur a(5);
Vecteur b;
b=a;
```

  Alors les attributs `valeur` de `b` et `a` pointent sur la même zone mémoire. Une supression de `b` va désallouer le tableau géré aussi par `a` !

On redéfinit l'opérateur d'affectation comme ci-après :

```
class Vecteur
{ ...
  Vecteur & operator= (const Vecteur &);
  ...
}
Vecteur & Vecteur::operator= (const Vecteur & v)
{ if (this != &v)
  { if (taille != v.taille)
    { delete valeur;
      valeur = new float [taille=v.taille];
    }
    memcpy (valeur, v.valeur, taille*sizeof(float));
  }
  return *this;
}
```

Remarques

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

1. La transmission de l'argument doit se faire par référence car on a besoin de l'adresse dans le test `(this != &v)` qui évite que l'on affecte un objet sur lui-même et éviter notamment de faire `delete valeur;` qui efface tout !
2. Le qualificatif `const` devant le paramètre permet d'affecter aussi des vecteurs constants (sinon c'est impossible).
3. L'opérateur d'affectation renvoie une adresse pour permettre des affectations multiples : `v1 = v2 = v3;`

Forme canonique d'une classe

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

On vient de voir que l'utilisation efficace de structures dynamiques, impliquant des désallocations à la destruction des objets, nécessite une grande prudence et une "discipline" s'impose : on parle de forme canonique qui est basée sur une définition systématique des 4 méthodes :

- Un constructeur de base
- Un destructeur
- Un constructeur de copie
- Un opérateur d'affectation

```
class T
{ ...
  public :
    T(...); //constructeur
    T(const T &); // constructeur de copie
    ~T(); // destructeur
    T & operator= (const T&); // op. affectation
};
```

Surcharge des opérateurs d'accès

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

- Dans la classe `Vecteur`, on va définir un opérateur permettant d'accéder à l'élément de rang i de l'attribut `valeur`.

On utilise la surcharge de l'opérateur [], ainsi `a[i]` va désigner l'élément voulu pour `a` un objet de la classe.

```
class Vecteur
{ ...
  float & operator[] (int);
  ...
}
float & Vecteur::operator[](int i)
{ return valeur[i]; }
```

La méthode renvoie une référence. On peut l'utiliser dans le membre gauche d'une affectation :

```
Vecteur a;
a[6]=3.2;
```

Si on veut utiliser cette fonction sur un objet constant, il faut ajouter `const` à droite de l'en-tête. Dans ce cas, il n'y a pas lieu de faire d'affectation sur une composante de l'objet ! et la fonction n'a pas besoin de renvoyer une référence. On a intérêt à redéfinir une deuxième focntion d'accès pour les objets constants :

```
class Vecteur
{ ...
  float operator[](int) const;
  ...
}
float Vecteur::operator[](int i) const
{ return valeur[i]; }
```

- On peut à l'identique utiliser l'opérateur ().

On peut utiliser l'opérateur d'accès dans la définition des autres méthodes de la classe. Par exemple, l'opérateur + peut se réécrire :

```
Vecteur Vecteur::operator+ (const Vecteur &v)
{ Vecteur result(taille);
  for (int i=0; i<taille; i++)
    result[i]=valeur[i]+v[i];
  return result;
}
```

Fonctions et classes amies

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Une fonction est *amie* d'une classe si elle peut accéder directement à toutes les données privées de cette classe.
- Une classe est *amie* d'une autre classe si toutes ses fonctions lui sont amies.
- La déclaration d'*amitié* doit se faire dans la classe qui autorise les accès à ses données privées.

```
class A
{ friend void fonction_b ();
  friend class C;
  ...
}
```

- La fonction `fonction_b` peut accéder aux données privées de A.
- Toutes les fonctions de la classe `C` peuvent accéder aux données privés de `A`.

Attention : la notion d'amitié est contraire à la notion d'encapsulation. Elle doit être utilisée de manière pertinente et avec parsimonie.

Exemple de fonction amie

Multiplication scalaire/vecteur

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

On veut écrire une fonction qui effectue le produit

$$\alpha \star v$$

Le membre de gauche étant d'un type simple, on pourrait la définir ainsi

```
Vecteur operator* (float alpha, const Vecteur &v)
{ Vecteur result(v.taille);
  for (int i=0; i<v.taille; i++)
    result[i] = alpha * v.valeur[i];
  return result;
}
```

Telle qu'elle est écrite, cette fonction accède aux données privées du vecteur v. Elle doit être déclarée amie de la classe vecteur comme indiqué ci-après :

```
class Vecteur
{ ...
  public :
  ...
  friend Vecteur operator* (float, const Vecteur &);
  ...
}
```

Bien attendu, on peut se dispenser ici de cette déclaration d'amitié, définir et utiliser proprement des opérateurs d'accès.

Surcharge des opérateurs de flux

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

- `cin` (resp. `cout`) est un objet appelé *flot* ou *flux* d'entrée prédéfini. Il est de la classe `istream` (resp. `ostream`) qui contient tous les flots d'entrée (resp. de sortie).
- Lorsque l'on écrit

```
cin >> quelque_chose;
```

cela correspond à

```
operator>> (cin, quelque_chose);
```

- Pour surcharger », il faut décrire la fonction hors-classe suivante :

```
istream & operator >>
        (istream &in, nom_classe & objet);
```

- Cette fonction, une fois définie, permettra de saisir des objets de type tt nom_classe, avec l'instruction :

```
in >> ...
```

- Par ailleurs, ce flot d'entrée peut être soit un flot prédéfini (par exemple, cin), soit un fichier. Il est alors possible de tester ce flot d'entrée (dans l'exemple on regarde si il correspond à `cin`) :

```
if (&in==&cin) ...;
```

- La fonction de surcharge de l'opérateur » se terminera finalement par

```
return in;
```

- On procède de la même manière pour surcharger l'opérateur de sortie «.

Exemple : surcharge du flux d'entrée dans la classe `Vecteur`

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

Dans la version qui suit, la saisie est directe si on utilise un fichier et interactive par l'entrée standard (clavier)

```
istream & operator >> (istream & is, Vecteur &v)
{ int ntaille;
  if (&is == &cin) cout << "taille du vecteur ? ";
  is >> ntaille;
  Vecteur nv(ntaille);
  if (&is == &cin) cout << "coefficients ? " << endl;
  for (int i=0; i < ntaille; i++)
    is >> nv[i];
  v = nv;
  return is;
}
```

Utilisation de fichiers

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

On se contente ici de donner un exemple élémentaire utilisant les flux d'entrée et de sortie (opérateur « à écrire ...) sur la classe `Vecteur` :

```
#include <fstream>
#include "Vecteur.h"
int main()
{ Vecteur x1, x2;
  ifstream in;
  ofstream out;
  in.open("input.dat");
  out.open("output.dat");
  in >> x1; in >> x2;
  Vecteur x3 = 2*x1+x2;
  out << "calcul de 2*x1+x2 :" << endl << x3;
  in.close();  out.close();
  return 0;
}
```

fichier "input.dat" :

```
4
2  3  4  1
4
1  2  3  5
```

fichier "output.dat"

```
calcul de 2*x1+x2 :
taille du vecteur : 4
coefficient du vecteur :
5  8  11  7
```

Construction d'une classe matrice

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

On construit une classe `Matrice` danns laquelle les données privées sont :

- le nombre de lignes ;
- le nombre de colonnes ;
- les coefficients sont gérés par un tableau de `Vecteur` (pointeur de `Vecteur`)

Classe Matrice
Forme canonique

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

On écrit les méthodes qui lui confèrent une `forme canonique`

- Constructeur de base
- Destructeur
- Constructeur de copie
- Opérateur d'affectation

```
#ifndef MATRICE
#define MATRICE

#include "Vecteur.h"
class Matrice
{ int nligne;
  int ncolonne;
  Vecteur * coefligne;
 public :
  Matrice (int=0, int=0);
  Matrice (const Matrice &);
  ~Matrice();
  Matrice & operator= (const Matrice &);
}
```

Classe `Matrice`

Opérateurs d'accès, surcharges des flux d'E/S

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

- On construit un opérateur d'accès [] qui renvoie une ligne de la matrice de type `Vecteur`.
- On construit des surcharges des opérateurs d'entrée/sorties

Classe Matrice
Opérateur de conversion

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Un constructeur qui n'a qu'un seul argument de type `T` spécifie une conversion d'une variable de type `T` vers un objet de la classe du constructeur.
- Ce constructeur est appelé automatiquement chaque fois qu'un objet de type `T` sera rencontré là où il faut un objet de la classe du constructeur.
- Dans la classe `Matrice`, on écrit un constructeur qui convertit un `Vecteur` de taille n en une `Matrice` de n lignes et 1 colonne.

```
class Matrice
{ ...
  Matrice (const Vecteur &);
  ...
}
Matrice::Matrice (const Vecteur & v):
                  nligne(v.taille),ncolonne(1)
{ coefligne = new Vecteur [nligne](ncolonne);
  for (int i=0; i<nligne; i++)
    coefligne[i][0] = v[i];
}
int main()
{ Vecteur v(10);
  cin >> v;
  Matrice m=v;
  cout << m;
  return 0;
}
```

Plan

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

8 Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

9 Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

10 Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

Héritage

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Concept fondamental de la POO
  `class <classe derive> : <etiquette de protection> <classe mere>`
- Une classe dite dérivée peut hériter d'une classe (ou plusieurs) : elle a donc implicitement les propriétés de celle-ci en plus de propriétés caractéristiques propres.

- Exemple : classe point colorée qui hérite d'une classe point.
- Ici, `Pointcol` hérite de manière *publique* de `Point`. Donc les membres de `Pointcol` ont accès aux membres publics de `Point` mais pas aux membres privés

```
class Point
{ int x; int y;
 public :
  Point(int, int);
  void deplace(int, int);
  void affiche ();
};
class Pointcol : public Point
{ char couleur;
 public :
  void colore (char cl) {couleur = cl;}
};
```

- Exemple :

```
void Pointcol::affichecol()
{ affiche() //appel affiche de la classe mere
  cout << "couleur : " << couleur << endl;
}
```

- On peut aussi redéfinir la fonction `affiche`. Il faut alors distinguer les deux méthodes du même nom : celle de la classe mère et celle de la classe dérivée.

```
void Pointcol::affiche()
{ Point::affiche();; //appel affiche de Point
  cout << "couleur : " << couleur << endl;
}
```

Les différentes dérivations

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- `public` : un membre public est accessible dans et hors de la classe ;
- `private` : un membre private est accessible que dans la classe ;
- `protected` : un membre protected est accessible dans la classe et dans les classes dérivées ;
- Règle : On ne peut pas augmenter la visibilité d'un membre par l'intermédiaire de l'héritage.

		Dans la classe de base	Dans la classe dérivée
Mode de dérivation	public	public	public
		protected	protected
		private	inaccessible
	protected	public	protected
		protected	protected
		private	inaccessible
	private	public	private
		protected	private
		private	inaccessible

Constructeurs et destructeurs

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple


```
class Base
{ ...
 public :
  Base(...);
  ~Base();
  ...
}
```

```
Class Derive : public Base
{ ...
 public :
  Derivee(...);
  ~Derive();
  ...
}
```

- La création d'un objet `Derive` nécessite d'abord la création d'un objet `Base`. Le constructeur de `Base` est appelé implicitement *avant toute autre instruction* au tout début de l'éxecution du constructeur de `Derive`.
- On a le processus inverse pour le destructeur. Celui de la classe mère est appelé implicitement *à la fin* de l'exécution de celui de la classe dérivée.

Constructeurs avec arguments

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple


```
class Point
{ ...
 public :
  Point(int , int);
 ...
}
```

```
Class Pointcol:public Point
{ ...
 public :
   Pointcol(int , int , char);
  ...
}
```

La description précédente est incomplète.

- A la construction d'un objet `Pointcol`, son constructeur est appelé et ce dernier appelle alors immédiatement celui de `Point` qui attend 2 arguments.
- Pour transmettre les arguments entre le constructeur de la classe dérivée et celui de la classe mère, on décrit cette transmissions dans l'en-tête de `Pointcal` de la manière suivante :

```
Pointcol(int a, int b, char cl) : Point(a, b)
```

Contrôle des accès

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

La Notion de protection permet de rendre accessibles pour une classe dérivée (et seulement dans ce cas) certains membres d'une classe de base.

```
class Point
{ protected :
    int x, y;
  public :
   Point( ...);
  ...
}
```

```
class Pointcol : public Point
{ char couleur;
 public :
  void affiche()
  { cout << "position " << x << " et " << y << endl;
    cout << "couleur " << couleur << endl;
  }
}
```

- `affiche` peut accéder aux champs protégés de `Point`.
- Ces mêmes champs restent privés pour les utilisateurs de la classe `Point`.

Amitiés et héritage

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Les fonctions amies d'une classe dérivées ont les mêmes autorisation d'accès que les fonctions membres de cette classe (notamment pour l'accès aux membres protégés).
- Les déclarations d'amitié ne s'héritent pas.

Les différents types de dérivations
Dérivation publique

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

On a utilisé jusqu'alors des dérivations publiques indiquées dans l'en-tête des classes dérivées.

```
class Pointcol : public Point { ...}
```

Voici les propriétés de ce type de dérivation :

- Les membres publics de la classe de base sont accessibles partout.
- Les membres protégés de la classe de base sont accessibles aux fonctions membres et aux fonctions amies de la classe dérivée, mais pas aux utilisateurs de la classe dérivée.
- Les membres privés de la classe de base sont inaccessibles en dehors de cette classe de base.

Les différents types de dérivations
Dérivation privée

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

On la signifie dans l'en-tête des classes dérivées.

```
class Pointcol : private Point { ...}
```

- Elle interdit à un utilisateur d'une classe dérivée l'accès aux membres publics de sa classe de base. Par contre, le concepteur de la classe dérivée peut utiliser les membres publics de la classe de base (comme un utilisateur ordinaire de la classe de base).
- Par exemple, si `Pointcol` hérite de manière privée, un objet de type `Pointcol` ne pourra pas appeler `deplace()` qui est un membre public de `Point`.

Les différents types de dérivations
Dérivation protégée

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Introduit plus tardivement dans les spécifications du C++, on la signifie dans l'en-tête des classes dérivées.

```
class Pointcol : protected Point { ...}
```

- Elle permet que les membres publics de la classe de base soient considérés comme protégés pour les dérivations suivantes.

Constructeur par recopie

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Soit une classe B qui hérite d'une classe A

- Si B n'a pas de constructeur de copie,
  - le constructeur de copie par défaut sera appelé,
  - pour la partie héritée de A, le constructeur de copie de A sera appelé s'il existe (sinon c'est encore le constructeur de copie par défaut)
- Si B possède un constructeur de copie,
  - celui-ci sera appelé ... mais pas celui de A.
  - Donc le constructeur de copie d'une classe dérivée doit prendre en charge la totalité de la copie de l'objet et notamment sa partie héritée. Mais ...
  - ... on peut utiliser un constructeur de A dans le constructeur de copie de B et lui transmettre l'objet à copier x lui-même. Alors x est converti au type A avec appel du constructeur de copie.

```
B(B &x) : A(x) //on provoque l'appel
               //du constructeur de copie de A
{ //copie de la partie de x specifique a B}
```

Opérateur d'affectation

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

Soit une classe B qui hérite d'une classe A

- Si B n'a pas de définition de =
  - l'affectation se fait membre à membre,
  - la partie héritée est gérée par l'affectation éventuellement redéfinie dans A.
- Si B a une définition de =
  - Seule l'affectation de B est appelée et pas celle de A.
  - L'opérateur = de B doit gérer tous les membres même ceux hérités

Plan

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

8 Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

9 Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

10 Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

Polymorphisme

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- C++ dans le cadre de l'héritage permet de redéfinir une méthode, *i.e* une méthode d'une classe fille possède le même nom et les mêmes paramètres ;
- Un pointeur/référence de type classe mère peut pointer/désigner un objet de type classe fille ;
- Problème : quelle méthode est appelée si elle est définie à la fois dans la classe mère ou la classe fille ?
- En fonction du type liaison statique ;
- En fonction de la nature de l'objet $\Rightarrow$ polymorphisme ;
- Le polymorphisme est possible en C++ quand il est spécifié par `virtual`.

Sans virtualité

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple


```
class Animal
{
public:
Animal();
        ~Animal();
void cri();
};
class Oiseau: public Animal
{
public :
Oiseau();
virtual ~Oiseau();
void cri();
};
```

Résultat

Je suis un animal

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

Animal::Animal(){}

Animal::~Animal(){}

void Animal::cri()
{
cout << "Je suis un animal";
}


Oiseau::Oiseau(){}

Oiseau::~Oiseau(){}

void Oiseau::cri()
{
cout << "Cuicui";
}

int main()
{
Animal *ptr = new Oiseau;
ptr->cri();
return 0;
}
```

Avec virtualité

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple


```
class Animal
{
public:
Animal();
virtual ~Animal();
virtual void cri();
};
class Oiseau: public Animal
{
public :
Oiseau();
virtual ~Oiseau();
void cri();
};
```

Résultat

Cuicui

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

Animal::Animal(){}

Animal::~Animal(){}

void Animal::cri()
{
cout << "Je suis un animal";
}


Oiseau::Oiseau(){}

Oiseau::~Oiseau(){}

void Oiseau::cri()
{
cout << "Cuicui";
}

int main()
{
Animal *ptr = new Oiseau;
ptr->cri();
return 0;
}
```

Virtualité du destructeur

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Dans le cadre de l'héritage, le destructeur de la classe de base n'est pas capable d'assurer la destruction compléte d'une instance d'une classe dérivée ;
- Il faut alors déclarer le destructeur de la classe de base virtuel ;
- Ce n'est pas nécessaire si votre classe ne sert JAMAIS de classe de base ;

Règle

Dès qu'une classe comporte une méthode virtuelle, il faut obligatoirement rendre le destructeur virtuel.

Virtualité et valeur par défaut

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple


```
class Animal
{
public:
virtual void cri(const char * = "Je suis un animal");
};

class Oiseau: public Animal
{
public :
void cri(const char * = "Cuicui");
};
```

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

void Animal::cri(const char * lecri)
{
cout <<  "Cri de l'animal : " << lecri;
}

void Oiseau::cri(const char * lecri)
{
cout <<  "Cri de l'oiseau : " << lecri;
}

int main()
{
Animal *ptr = new Oiseau;
ptr->cri();
return 0;
}
```

Résultat

Cri de l'oiseau : Je suis un animal

- Il ne faut jamais doter de valeur par défaut ses paramètres quand on utilise le polymorphisme !

Transmission de la virtualité

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Lorsqu'une classe dérivée redéfinit une méthode déclarée virtuelle dans une classe de base cette méthode reste virtuelle ;

Règle

Il est fortement conseillé de conserver la virtualité explicitement !

Méthode virtuelle pure

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Pour déclarer une méthode virtuelle pure dans une classe, il suffit de faire suivre sa déclaration de « =0 ».
- La fonction doit également être déclarée virtuelle.

```
virtual type nom(parametres) =0;
```

Classe abstraite

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Une classe abstraite n'est utilisée que dans le cadre de l'héritage.

$\Rightarrow$ Pas d'instanciation possible

$\Rightarrow$ Pas de constructeur en génèral

- En C++ contient au moins une méthode virtuelle pure.

Rôle

- Permet de définir des comportements (méthodes) qui devront être implémentés dans les classes filles, mais sans implémenter ces comportements.
- Les classes filles respecteront le contrat défini par la classe mère abstraite.
- Interface de programmation.

Classe abstraite
Exemple

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple


```
class SurfaceFermee
{
 public :
  virtual double surface()=0;
  // Fonction virtuelle pure
};

class Disque : public SurfaceFermee
{
 protected :
  double rayon;
 public :
  Disque(double r) : rayon(r) {}
  virtual double surface()
{return 3.14 * rayon * rayon;}
};

class Rectangle : public SurfaceFermee
{
 protected :
  double longueur, largeur;
 public :
  Rectangle(double long, double lar)
    : longueur(long), largeur(larg) {}
  virtual double surface()
  {
   return longueur * largeur;
  }
};
```

```
int main()
{
 //......
 SurfaceFermee construction[3];
 //......
 double surfaceTotale = 0;
 int i;
 for(i = 0; i < 3; i++)
  surfaceTotale += construction[i].surface();
 //......
 return 0;
}
```

Intérêt

- Permet de gérer correctement le polymorphisme
- La classe `SurfaceFermee` crée un rapport syntaxique.
- Les deux classes `Disque` et `Rectangle` respectent un contrat.

Méthode virtuelle pure avec corps

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Il est possible de déclarer une méthode virtuelle pure dans une classe, avec un corps.
- `virtual type nom(parametres) =0 { corps de la methode };`
- La classe reste virtuelle mais les classes qui en hérite n'ont pas obligatoirement besoin de redéfinir la méthode.
- Attention cela doit être manié avec beaucoup de précautions.

Méthode virtuelle pure avec corps

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Il permet de créer des classes dérivées à partir de plusieurs classes de base.
- Pour chaque classe de base, on peut définir le mode d'héritage.

```
class A {
  public:
    void fa() { /* ... */ }
  protected:
    int x;
};

class B {
  public:
    void fb() { /* ... */ }
  protected:
    int x;
};
```

```
class C: public B, public A {
  public:
    void fc();
};

void C::fc() {
  int i;
  fa();
  i = A::x + B::x;
 // resolution de portee pour lever l'ambiguite
}
```

Ordre d'appel des constructeurs et des destructeurs

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

- Dans l'héritage multiple, les constructeurs sont appelés dans l'ordre de déclaration de l'héritage.
- Les destructeurs sont appelés dans l'ordre inverse de celui des constructeurs.

```
class A {
  public:
    A(int n=0) { /* ... */ }
  // ...
};

class B {
  public:
    B(int n=0) { /* ... */ }
  // ...
};

class C: public B, public A {
//       ^^^^^^^^^^^^^^^^^^
// ordre d'appel des constructeurs des classes de base
//
  public:
    C(int i, int j) : A(i) , B(j) { /* ... */ }
  // ...
};
```

```
int main() {
  C objet_c;
  // appel des constructeurs B(), A() et C()
  // ...
}
```

Héritage virtuel

Langage objet
Classe
Constructeurs-destructeurs
Surcharge d'opérateurs
Fonctions et classes amies
Surcharge des opérateurs de flux
fichiers
Ex : classe matrice
Héritage
Constructeurs et destructeurs
Contrôle des accès
Constructeur par recopie
Virtualité, classes abstraites
Polymorphisme
Méthodes virtuelles
Classes abstraites
Héritage multiple

- Utilisé pour régler les problèmes d'héritage multiple.
- Supposons qu'une classe `B` et qu'une classe `C` hérite de `A` et qu'une classe `D` hérite de `B` et `C`, l'héritage virtuel va permettre qu'elle n'hérite pas deux fois de `A`.

Héritage virtuel
Mise œuvre

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple


```
// Classe de base du systeme
class A
{
  public:
    A ()
     { cout << "Constructeur de A" << endl;}
  private:
    int a;
};

// Sous classe directe de A
class B : virtual public A
{
  public:
    // Le constructeur de B appelle celui de A
    // afin d'initialiser correctement les
    // attributs de A presents dans la classe B
    B() : A()
    { cout << "Constructeur de B" << endl;};
  private:
    int b;
};
```

```
// Sous classe directe de A
class C : virtual public A
{
  public:
    // Le constructeur de C appele celui de A
    // afin d'initialiser correctement les attributs
    // presents dans la classe C
    C() : A ()
    { cout << "Constructeur de C" << endl;};
  private:
    int c;
};

// Sous classe de B et C
// Afin de gerer l'heritage a repetition,
// on introduit egalement A dans la liste
// des super classes
class D : virtual public A, public B, public C
{
  public:
    D() : A (), B (), C ()
    { cout << "Constructeur de D" << endl;};
  private:
    int d;
};
```

Héritage virtuel
Mise œuvre

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple


```
int main()
{
  B b1;
  cout << endl;
  C c1;
  cout << endl;
  D d1;
  return 0;
}
```

```
Constructeur de A
// Le constructeur de B (pour l'objet b1)
// appelle bien celui de A
Constructeur de B

Constructeur de A
// Le constructeur de C (pour l'objet c1)
// appelle bien celui de A
Constructeur de C

Constructeur de A
// Le constructeur de D //
 appelle une fois le constructeur de A
Constructeur de B
// une fois le constructeur de B qui n'appelle
// pas celui de A du fait de l'héritage virtuel
Constructeur de C
// une fois le constructeur de C qui n'appelle
// celui de A du fait de l'héritage virtuel
Constructeur de D
// puis finalement son code propre !
```

Héritage virtuel

Langage objet
- Classe
- Constructeurs-destructeurs
- Surcharge d'opérateurs
- Fonctions et classes amies
- Surcharge des opérateurs de flux
- fichiers
- Ex : classe matrice

Héritage
- Constructeurs et destructeurs
- Contrôle des accès
- Constructeur par recopie

Virtualité, classes abstraites
- Polymorphisme
- Méthodes virtuelles
- Classes abstraites
- Héritage multiple

- En plus us de l'héritage double sur les classes `B` et `C`, ce code ajoute explicitement la classe `A` en super classe de `D`.
- `A` doit arriver en première position dans la liste des super classes et l'héritage sur `A` doit de nouveau être virtuel.
- Cette construction garantit que :
  - Les attributs de `A` ne seront présents qu'en un seul exemplaire, et ce, directement depuis `A`.
  - Le constructeur de `A` est explicitement appelé dans celui de `D` assurant ainsi l'initialisation correcte des attributs hérités de `A`. Les appels au constructeur de `A` depuis ceux de `B` et `C` ne seront pas exécutés.
- L'ordre des modificateurs virtual et public dans un héritage virtuel est sans conséquence.

Plan

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

11 Généricité, template et namespace
- Template
- Namespace

12 Exceptions en C++

13 Patron de conception
- Les principaux patrons de conception

14 La STL
- L'API STL

15 Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

Les templates (patrons)

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Permettent de paramétrer les structures, les classes, les fonctions et les méthodes.

Objectif

Les rendre indépendantes du ou des types de données manipulés.

- Le compilateur utilise le patron pour l'instancier sur des types particuliers.

Exemple d'utilisation

Fonction qui calcule le maximum sur un tableau

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- 2 choix possibles :
  1. Surcharger la fonction pour chaque type nécessaire
  2. Créer un patron, le compilateur instanciera alors celui-ci pour chaque type employé dans le programme.

Syntaxe

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Pour créer un template, on préfixe la déclaration de la structure, la classe, la fonction ou la méthode par
  template<typename T1, ..., typename Tn>
- T1, ... Tn représentent les types qui seront employés dans les instanciations.
- Dans le reste de la fonction, la structure, la classe, la méthode, on emploie T1, ... Tn comme un type quelconque.

```
// Sur une structure
template<typename T>
struct Complexe
{
    T re;
    T img;
};
Complexe<int> cpxEntier = {1, 2};
Complexe<double> cpxDouble = {1.0, 2.0};
// Sur une fonction
template<typename T>
T max(const T &a, const T &b)
{
    return (a < b) ? b : a;
}
int a = max<int>(3, 2);
// Type explicite
double b = max(10.0, 12.0);
// Type implicite
```

Exemple

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets


```
template <typename T>
class Complexe {
  private:
        T Reel;
        T Imaginaire;
    public:
        Complexe(T re = 0, T im = 0;) {}
        T GetReel() { return Reel;}
        T GetImaginaire() { return Imaginaire; }
        void SetReel(T re) {  Reel = re;}
        void SetImaginaire(T
im) {Imaginaire = im;}
        double GetModule();
        void AfficheNombre()  {
            cout << GetReel() << "␣+␣"
                 << GetImaginaire() << "i" <<
            endl;}
};
template <typename T>
double Complexe<T>::GetModule(){ // dans fichier .h
   // calcul du module ...
}
```

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

int main()

{
   Complexe <float> fz(2.,3.);
   Complexe <double> dz(4.,5.);
   fz.AfficheNombre();
   cout << fz.GetModule() << endl;
   dz.AfficheNombre();
   cout << dz.GetModule() << endl;
   return 0;
}
```

Pourquoi ?

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

Le problème

- Conflits de nom :
  - Avec des noms comme `Date`,`Object` ou encore `String`, il peut y avoir des conflit avec les classes déjà définies.
  - Une solution, préfixer toutes les classes par un code (exemple avec (C) : `CString`, `CObject` comme dans les MFC.
  - Soit le préfixe est trop court et le risque de conflit existe encore, soit il est trop long et il devient pénible à écrire.

Solution

- Utilisation d'un namespace !

```
namespace NomDuNamespace
{
        //Contenu du namespace
}
```

- Les éléments du namespace ont pour identificateur `NomDuNameSpace::id`

Utilisation

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets


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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

Trace :

```
5
3.1416
```

```cpp
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

Trace

```
5
2.7183
10
3.1416
```

Utilisation

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets


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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

Trace :

```
5
10
3.1416
2.7183
```

```
// using namespace example
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

```
5
3.1416
```

Plan

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

11 Généricité, template et namespace
- Template
- Namespace

12 Exceptions en C++

13 Patron de conception
- Les principaux patrons de conception

14 La STL
- L'API STL

15 Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

Exception

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Interruption de l'exécution du programme à la suite d'un événement particulier ;
- Gestion des erreurs ;
- Réaliser des traitements spécifiques aux événements qui en sont la cause.
- La fonction qui détecte une erreur lance une exception. Cette exception interrompt l'exécution de la fonction, et un gestionnaire d'exception approprié est recherché.

Lancement et récupération d'une exception

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Pour lancer une exception, on créé un objet dont la classe caractérise cette exception que l'on lance par `throw`.

```
throw objet;
```

- Gestion de l'exception à l'aide d'un bloc `try catch`

```
try
{
    // Code susceptible de generer des exceptions ...
}
catch (classe [&][temp])
{
    // Traitement de l'exception associee a la classe
}
```

- Exemple : Erreur.cc ;

Spécification des exceptions lancées par une fonction

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- On peut limiter les exceptions lancées directement ou indirectement par une fonction.

```
float mafonction (char param) throw (int, float);
```

- Les seules exceptions possibles sont de type `int` et `float`.

Syntaxe

```
int mafonction (int param) throw(); // Pas d'exception autorisee
int mafonction (int param);         // Toutes les exceptions autorisees
```

La classe exception

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Classe appartenant à la STL ;
- Définie dans le fichier d'en-tête `exception` ;
- Appartient au namespace `std` ;
- Possède une méthode virtuelle `char *what()` pouvant être redéfinie dans les classes dérivées.

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

class MonException: public exception
{
  virtual const char* what() const throw()
  {
    return "Oups une exception";
  }
} monex;

int main () {
  try
  {
    throw monex;
  }
  catch (exception& e)
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

Plan

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

11 Généricité, template et namespace
- Template
- Namespace

12 Exceptions en C++

13 Patron de conception
- Les principaux patrons de conception

14 La STL
- L'API STL

15 Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

Design pattern ou patron de conception

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

Définition

- Concept destiné à résoudre les problèmes récurrents suivant le paradigme objet.
- Décrivent des solutions standards pour répondre à des problèmes d'architecture et de conception des logiciels.
- Outil de capitalisation de l'expérience appliqué à la conception logicielle.

- Les design patterns sont conçus pour être utilisés dans n'importe quel langage objet.
- Leur description fait généralement usage de diagrammes de classes UML.
- L'héritage et le polymorphisme sont largement utilisés.
- Rien n'empêche de mettre en œuvre les design patterns en utilisant les templates.

Conteneurs et iterateurs

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Une multitude d'objets peuvent être rangés dans divers conteneurs :
  - Listes, Matrices, Dictionnaires . . .
- Comment écrire des algorithmes qui fonctionneront avec tous les conteneurs ?

  $\Rightarrow$ Mécanisme unique pour parcourir les éléments d'un conteneur, quel qu'il soit : c'est le rôle des objets itérateurs.

Stratégie

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Un objet stratégie représente une fonction.
- Un algorithme générique peut être paramétré par des stratégies, exemple :

```
struct OrdreAscendant {
   template <typename T> int comparer
     (const T & a, const T & b) { return a-b; }
};
struct OrdreDescendant {
   template <typename T> int comparer
     (const T & a, const T & b) { return b-a; }
};
//.....
OrdreDescendant desc;
maListe.trier(desc);
```

Adaptateur

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Un adaptateur est un objet qui «entoure»un autre objet pour en modifier l'interface.
- Par exemple :

```
class Vecteur { int * tab; \\... };
class Matrice {
   Vecteur v;
 public:
   void int & element(int x, int y) {
     return v[y*largeur+x];
   }
//....
};
```

Composite

Généricité, template et namespace
- Template
- Namespace

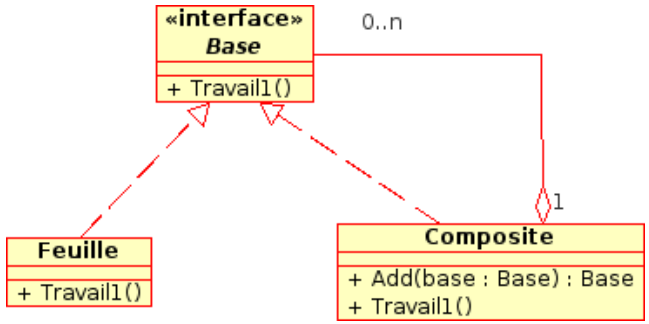
Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Un composite est un objet qui peut contenir d'autres objets, dont éventuellement d'autres composites.
- Permet de manipuler un groupe d'objets de la même façon que s'il s'agissait d'un seul objet. (ex : système de fichiers).
- Ceci permet de représenter des structures arborescentes.
- L'implémentation classique ressemble à :




```
class Element { \\... };
class Composite : public Element {
   Element* tableauElements;
//...};
```

Fabrique

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Un objet fabrique a pour but de créer des instances d'une autre classe.
- Exemple : un programme doit régulièrement créer de nouvelles connexions à une base de données.
- On lui fournit un objet `FabriqueCnx` dont le rôle est de fabriquer des objets `CnxBase`.
- Si l'on change de base de données, il suffira de fournir une nouvelle fabrique au programme.
- Exemple : exempleFabrique.cpp ;

Singleton

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Il arrive fréquemment que certains objets soient uniques au sein d'une application.
  - Connexion base de données, objet de configuration ...
- Pour éviter de définir des variables globales et pour offrir un peu plus de flexibilité au programme, on peut définir une classe qui ne peut avoir qu'une seule instance : il s'agit alors d'un Singleton.
- Exemple :

```
class Config {
  protected: Config() { ... } // constructeur protege
  public:
   static Config & instance() { return instance; }
   static Config instance;
  ...
};
...
Chaine s = Config.instance().parametre("host");
```

Observateur

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Il est fréquent de devoir gérer des évènements dans une application.
- Un observateur est un objet qui doit être « prévenu » lorsqu'un évènement a lieu.
- On l'enregistrer auprès de l'objet qui est susceptible de déclencher l'évènement.

```
class ObservateurDeBouton {
 public:
  virtual void clic(bool estDouble) = 0;
};

class Bouton { ...
  ObservateurDeBouton * tableauObservateurs;
  void declencherEvenement();
 public:
  void enregistrer (const Observateur & obs) {...}
};

class Afficheur : public ObservateurDeBouton {
 public:
  virtual void clic(bool estDouble) {
   if (estDouble) afficherMessage("Clic");
  }
};

int main() {
 Afficheur monAfficheur;
 Bouton monBouton;
 monBouton.enregistrer(monAfficheur);
 return 0;
}
```

Proxy

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Lorsque l'on veut communiquer avec un objet distant, une technique consiste à communiquer avec un objet « proxy » local, qui s'occupe de relayer les requêtes.
- L'objet proxy possède la même interface que l'objet distant, ce qui facilite la programmation du logiciel client.

Visiteur

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Lorsque l'on souhaite appliquer un algorithme à un graphe d'objets (généralement une arborescence), le plus simple est souvent de fournir un objet « visiteur » à l'un des nœuds du graphe.
- Les nœuds doivent être programmés pour propager les visiteurs dans tout le graphe.
- L'objet visiteur applique l'algorithme souhaité au fur et à mesure qu'on lui fait parcourir le graphe.

```
class Fichier;
class VisiteurDeFichier { public:
   virtual void visiter(Fichier & fichier) = 0;
};
class Fichier { ... public:
   virtual void propager(VisiteurDeFichier & v) {
       v.visiter(*this);
   }
};
class Repertoire : public Fichier { ... };
// propager est redefinie pour appeler propager sur
// chacun des fichiers du repertoire
class TrouveFichiers : public VisiteurDeFichier {
   Fichier * tableauFichiersTrouves;
   Chaine critereRecherche;
  public:
   virtual void visiter(Fichier & fichier) {
     if (fichier.filtre(critereRecherche))
                    ajouteFichierTrouve(fichier);
   }
};
...
TrouveFichiers trouveFichiersTexte("*.txt");
Repertoire rep ("/home/machin/");
rep.propager(trouveFichiersTexte);
```

Façade

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Lorsqu'un module d'une application a de multiples fonctionnalités, mises en œuvre par de nombreux objets, il vaut mieux éviter que les autres modules puissent accéder à tous ces objets, surtout s'il s'agit d'une application distribuée !
- On définit un petit nombre d'objets (la façade) pour l'interface avec le reste de l'application.

Plan

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

11 Généricité, template et namespace
- Template
- Namespace

12 Exceptions en C++

13 Patron de conception
- Les principaux patrons de conception

14 La STL
- L'API STL

15 Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

Standard Template Library

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles.
  - des algorithmes de recherches, suppression, de tas.
- Il existe principalement trois types d'éléments :
  - les conteneurs : stockent les données.
  - les itérateurs : parcourent les conteneurs.
  - les algorithmes : travaillent sur les conteneurs à partir des itérateurs.
- http://www.sgi.com/tech/stl

Les conteneurs

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Les conteneurs permettent de contenir des éléments : la généricité paramétrera le type des éléments à utiliser.
- Plusieurs types de conteneurs sont utilisables : les tableaux, classes `vector` et `deque`, les listes `list`, les piles `stack`, les `queue`, les ensembles `set`, les ensembles multiples `multiset`, les dictionnaires associatifs `map` et `multimap`.

Les conteneurs

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un «pointeur »sur un élément d'un conteneur.
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux . . .

Les algorithmes et allocateurs

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
- Les algorithmes utilisent des itérateurs linéaires ou à accès aléatoire $\Rightarrow$ tous les conteneurs peuvent appliquer les algorithmes.
- Il en existe beaucoup :
  - `std::find` : cherche quelque chose ;
  - `std::fill` : rempli un conteneur avec une donnée ;
  - `std::generate` : génére des données à partir d'une fonction.
- Les allocateurs servent à gérer la mémoire dans certains cas.

La classe std : :string

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- La classe `std::string` encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Quelques méthodes ou opérateurs :
  - `operator[](size_t)` : l'accès à un char.
  - `c_str()` et `data()` : la conversion de la chaîne de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - `append`, `operator+`, `operator+=`, `insert`, `replace`, `erase`, `assign`, `operator=` : les fonctions d'ajout, d'insertion, de suppression, d'affectation.
  - `find`, `rfind`, `find_first_of`, `find_first_not_of`, `find_last_of` : les fonctions de recherche.
  - `http://www.sgi.com/tech/stl/basic_string.html`

Les flux de chaînes

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- `std::string` ne possède pas de fonctions pour ajouter des entiers, réels ou booléen directement.
- Les `std::stringstream` permettent une édition des chaînes de caractères par les flux.
- Exemple :

```
std::ostringstream st ;
st << "␣la␣valeur␣de␣x␣est␣:␣" << 10 << "␣pixels␣,␣"
   << "␣alors␣que␣y␣vaut␣:␣" << 0.345 << "␣points␣,␣"
   << "␣x␣==␣10␣?␣:␣" << std::boolalpha << ( x == 10) << "␣.\␣n␣"

std::string chaine ( st.str()); // transforme en std::string
std::cout << chaine ;
printf ( "␣La␣chaine␣:␣%␣s␣" , chaine.c_str());
```

La classe std : :vector

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Gestion automatique de la mémoire, allocation, réallocation et destruction.
- Coût de O(1) pour l'accès aléatoire et la suppression en fin de tableau.
- Coût de O(n) pour l'ajout ou la suppression d'une case.
- Exemple :

```
std::vector<double> tab2(1000);
std::vector<std::vector> tab3(1000);
for(size_t i = 0; i < 1000; ++ i )
    tab3[i].resize(1000);
```

La classe std : :vector

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Les méthodes :
  - `operator[](int)` et `at(int)` les opérateurs d'accès aléatoires aux éléments, le deuxième lève une exception si l'entier passé en paramètre sort du tableau. `size_t size()` et `resize(size_t)` et `reserve(size_t)` les fonctions pour, respectivement, connaître la taille d'un tableau, modifier sa taille ou préallouer une zone mémoire.
  - `push_back(T)`, `pop_back()`, les fonctions d'ajout et de suppression d'élément en fin de tableau (coût O(1)).
- Exemple :

```
std::vector<double> v(1000, 1.0);
v.push_back(2.0);
```

La classe std : :vector

Les itérateurs

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Un parcours de conteneur fonctionne toujours avec un itérateur ;
- Comme pour les pointeurs classiques, il faut faire attention à ne pas utiliser un itérateur qui « pointe » vers un objet qui n'existe plus.
- Dans le cas d'un `vector`, tant qu'on ne dépasse pas la capacité, il n'y a pas de réallocation, les itérateurs restent valides.

```
std::vector<std::string> botte(200);
for (std::vector<int>::iterator it=botte.begin(); it != botte.end(); ++ it )
  std::cout << (*it) << '' '';
std::vector<int>::iterator fd ;
fd = std::find(botte.begin() , botte.end(), ''aiguille'');
```

La classe std : :vector

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets


```
template<typename T, typename A = allocator<T>>
class vector {
public:
  typedef A::reference reference;
  typedef A::pointer iterator;
  explicit vector(const A& al = A());
  explicit vector(size_type n,
                  const T& v = T(),
                  const A& al = A());
  vector(const vector& x);
  void reserve(size_type n);
  size_type capacity() const;
  iterator begin();
  iterator end();
  void resize(size_type n, T x = T());
  size_type size() const;
  size_type max_size() const;
  bool empty() const;
```

```
A get_allocator() const;
reference at(size_type pos);
const_reference at(size_type pos) const;
reference operator[](size_type pos);
const_reference operator[](size_type pos);
reference front();
reference back();
void push_back(const T& x);
void pop_back();
void assign(const_iterator first,
            const_iterator last);
void assign(size_type n, const T& x = T());
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
iterator erase(iterator it);
void clear();
void swap(vector x);
```

La classe std : :list

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Liste doublement chaînée ;
- Coût en O(1) pour l'ajout, la suppression ;
- `push_back(T)`, `push_front(T)`, `pop_back()` et `pop_front()` respectivement ajout en début et fin de liste et suppression en début et fin de liste.

```
std::list<int> l ;
l.push_back(5);
l.push_front(3);
std::cout << l.front() == 3;
std::list<int>::iterator it ;
it = std::find (l.begin(); l.end(); 3);
std::cout << it != l.end();
```

La classe std : :list

Quelques méthodes

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- `splice()` : transférer le contenu d'un conteneur dans la liste ;
- `sort()` : trier la liste en se basant sur l'operateur < des éléments.
- `unique()` : remplace toute séquence d'éléments identiques par un seul élément (en général, on aura trié la liste auparavant).
- `merge()` : fusion ordonnée de deux listes.
- `reverse()` : inverse l'ordre de la liste

```
// list::merge
#include <iostream>
#include <list>
using namespace std;

// this compares equal two doubles if
// their interger equivalents are equal
bool mycomparison (double first, double second)
{ return ( int(first)<int(second) ); }

int main ()
{
  list<double> first, second;

  first.push_back (3.1);
  first.push_back (2.2);
  first.push_back (2.9);
  second.push_back (3.7);
  second.push_back (7.1);
  second.push_back (1.4);
  first.sort();
  second.sort();
  first.merge(second);
  second.push_back (2.1);
  first.merge(second,mycomparison);
  cout << "first␣contains:";
  for (list<double>::iterator it=first.begin();
       it!=first.end(); ++it)
    cout << "␣" << *it;
  cout << endl;  return 0;
}
//first contains: 1.4 2.2 2.9 2.1 3.1 3.7 7.1
```

La classe std : :deque

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Un conteneur `deque` est un intermédiaire entre un `vector` et une `list` : il gère des tableaux d'éléments chaînés de façon bidirectionnelle.
- Insertions, suppressions rapides et accès rapide aux éléments.
- L'opérateur `[]` et la fonction `at()` sont définis.

Les classes std : :map et std : :multimap

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Conteneurs associatifs : il permettent de récupérer une valeur lorsqu'on leur donne une clé (ex : un dictionnaire)
- Les clés doivent pouvoir être comparées avec l'opérateur `<`.
- Un `multimap` est identique à un `map`, à ceci près qu'il autorise le stockage de plusieurs valeurs ayant des clés identiques.

```
typedef map<double, string> DoubleEnString;
DoubleEnString leMap;
leMap.insert(
   DoubleEnString::value_type(3.14, "Pi"));
leMap.insert(
   DoubleEnString::value_type(0.00, "Zero"));
leMap.insert(
   DoubleEnString::value_type(0.50, "Un demi"));
double aChercher;
cout << "Tapez un reel connu : ";
cin >> aChercher;
DoubleEnString::iterator iter =
   leMap.find(aChercher);
if (iter == leMap.end())
   cout << "Inconnu au bataillon" << endl;
else cout << (*iter).second << endl;
```

Plan

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

11 Généricité, template et namespace
- Template
- Namespace

12 Exceptions en C++

13 Patron de conception
- Les principaux patrons de conception

14 La STL
- L'API STL

15 Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

Introduction

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Qt offre aux développeurs toutes les fonctionnalités nécessaires pour construire des interfaces graphiques.
- Qt est portable sous Windows Microsoft, Mac OS, Linux et de nombreuses variantes Unix.
- Qt est orienté objet et s'appuie sur C++.
- Qt est utilisé pour Kde.
- Développé par la société trolltech rachetée par Nokia.

Objets graphiques, évenements et callback

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Qt manipule des objets graphiques *widget*.
- La communication est basée sur les concepts de « Signaux » et « Slots »
- Gestion des évenements.
- Gestion des « Timers ».
- Composition d'objets.
- Pointeurs protégés.

Gestion de la mémoire

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Les objets Qt sont des descendants de `QObject` ;
- Quand on crée un objet il est possible de passer un pointeur vers l'objet parent. Ce qui permet lorque le parent est supprimé de supprimer les descendants.

```
class monObjet : public QObject {
 Public :
       MonObjet(QObject *parent=0, char *name=0) : QObject(parent, name) {
        // ....
       }
};
```

Introspection

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Chaque objet Qt a une description de lui-même ;
- `const QMetaObject * QObject::metaObject () const`
  Retoune un pointeur sur le «méta-objet » de l'objet.
- Un méta-objet contient les informations sur le nom de la classe, la classe mère, les signaux et les slots.
- Exemple :

```
QObject *obj = new QPushButton;
cout << obj->metaObject()->className(); // retourne "QPushButton"
```

Signaux et slots

Généricité, template et namespace
- Template
- Namespace

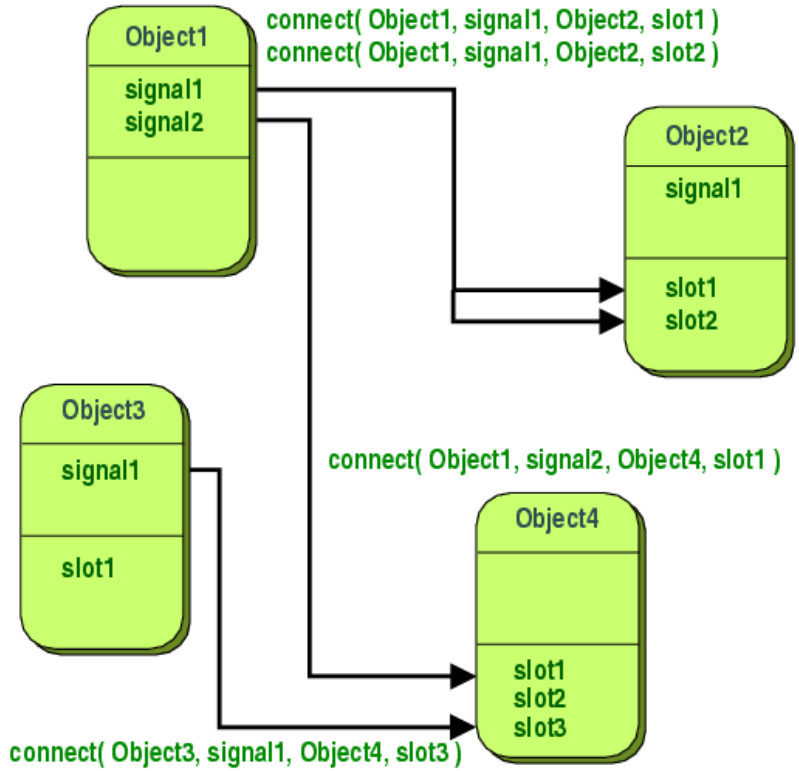
Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Tout objet Qt (descendant de `QObject`) peut émettre des signaux dans l'environnement pour indiquer (en général) un changement d'état pouvant être intéressant pour d'autres objets.
- Tout objet Qt peut réagir à des signaux émis en exécutant des méthodes particulières appelées slots.
- Ce mécanisme favorise la réutilisabilité des composants.



Signaux et slots

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Signal : notification d'un évènement ;
- Slot : méthode pouvant être déclenchée par un signal ;
- Plusieurs signaux peuvent être connectés à un même slot et un même signal peut être connecté à plusieurs slots.
- Les paramètres du signal sont transmis au slot auquel il est connecté.
- Un slot peut avoir moins de paramètres que le signal associé.

- Un slot est une fonction comme une autre.
- Un signal est un élément particulier :
  - Sa définition ressemble à la déclaration d'une fonction.
  - Le type de retour indiqué est toujours `void`.
  - On n'écrit pas le corps de cette fonction/signal.
- Emission d'un signal : `emit(nom_du_signal(params)).`
- Les signaux et slots ne peuvent être déclarés que dans le cadre d'une classe qui étend `QObject`.
- La première ligne de la déclaration de cette classe doit être un appel à la macro `Q_OBJECT`.
- Connexion

```
connect(Emetteur, SIGNAL(nomSignal(...)),
        Receveur, SLOT(nomSlot(...)))
```

Signaux et slots
Exemple

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

```
//signaux.h
#include <qobject.h>
#include <iostream>

class Receveur : public QObject {
  Q_OBJECT // Macro pour utiliser le modele Qt
 public:
   Receveur(QObject *parent=0, char *name=0)
: QObject(parent, name){}
 public slots:
   void get(const char *x )
   {
     std::cout << "Receveur: " << x << std::endl;
   }
};

class EnvoyeurA : public QObject {
  Q_OBJECT
 public:
   EnvoyeurA( QObject *parent=0, char *name=0 )
: QObject( parent, name ){}
   void doSend()
   {
     emit(send("Signal de A"));
   }
signals:
   void send(const char *); };
```

```
//signaux.h (suite)
class EnvoyeurB : public QObject {
  Q_OBJECT
 public:
   EnvoyeurB( QObject *parent=0, char *name=0 )
: QObject( parent, name ){}
   void doStuff()
   {
     emit(transmit("Signal de B"));
   }
 signals:
   void transmit(const char *); };


// main.cpp
#include <qapplication.h>
#include "signaux.h"

int main( int argc, char **argv ){
   QApplication app(argc, argv);
   Receveur r;
   EnvoyeurA sa;
   EnvoyeurB sb;
   QObject::connect(&sa,
                    SIGNAL(send(const char *)),
                    &r, SLOT(get(const char *)));
   QObject::connect(&sb,
                    SIGNAL(transmit(const char *)),
                    &r, SLOT(get(const char *)));
   sa.doSend();
   sb.doStuff();
   app.quit();
   return 0;
}
```

Compilation

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Programme principal : `main.cpp`.
- Classes (composant) : 2 fichiers
  - `nomClasse.h` : définition de la classe ;
  - `nomClasse.cpp` : implémentation des méthodes (éventuellement).
- `moc` traduit le modèle Qt (signaux/slots...) en du C++ standard `moc nomClasse.h -o moc_nomClasse.cpp`
- qmake ensuite :

```
qmake -project
qmake
make
```

Quelques widgets

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Dérivent d'une classe `QWidget` qui elle-même dérive de `QObject`
  - Possèdent des signaux et des slots ;
- Chaque Widget peut disposer d'un parent : fenêtre, menu, ou layout auquel il est ajouté.
- Pour chaque Widget :
  - constructeur ;
  - aspect ;
  - principaux signaux/slots.

QWidget

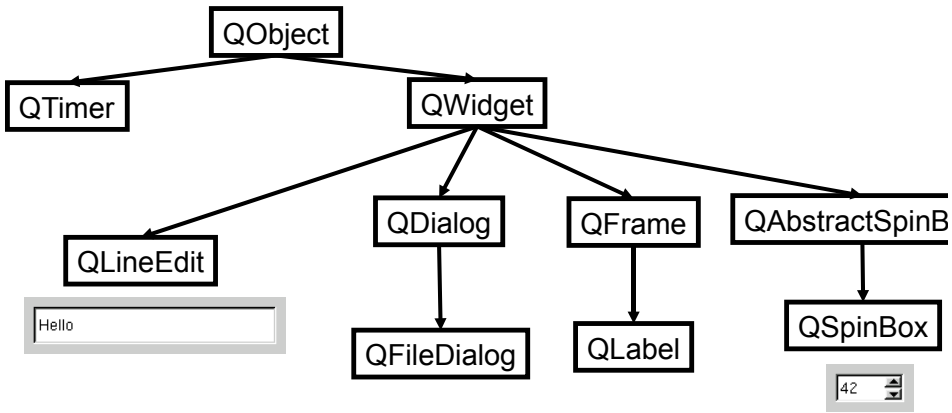
Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- Classe mère

```
QWidget(QWidget* parent); // parent peut etre NULL
```

- Slots :

```
void show();              void hide();
void move(int, int);      void resize(int, int);
void enabled(bool);
```



QLabel

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Affichage de texte

```
QQLabel::QLabel (QString t,
                 Qwidget *p)
```

- Slots :

```
void setText(Qstring);
void clear();
```

```
#include <qapplication.h>
#include <qlabel.h>

int main(int argc, char *argv[])
{
 QApplication app(argc, argv);
 QLabel * l = new QLabel( QString("Vive Qt"), 0);
 l->show();
 return app.exec();
}
```

Les boutons

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- `QPushButton` : bouton classique

  ```
  QPushButton(QString t, QWidget* p);
  ```

  - Signal

    ```
    void clicked();
    ```

- `QCheckBox` : case à cocher

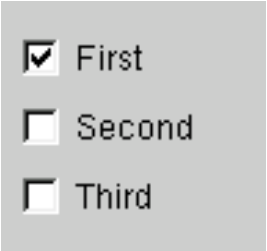
  ```
  QCheckBox(QString t, QWidget* p);
  ```

  - Slot

    ```
    void setChecked(bool);
    ```

  - Signaux

    ```
    void clicked();
    void toggled(bool);
    ```



Les boutons

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- `QButtonGroup` : grouper les boutons radio

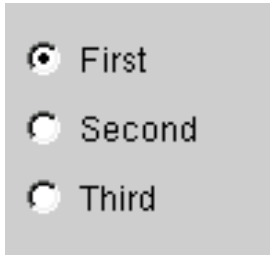
```
QButtonGroup (QString t, QWidget *p);
```

  - Signal

```
void clicked(int id);
```

- `QRadioButton` : bouton radio

```
QRadioButton(QString t, QWidget* p);
```



  - Slot

```
void setChecked(bool);
```

  - Signaux

```
void clicked();
void toggled(bool);
```

Les listes

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- `QListBox` : liste en lecture seule

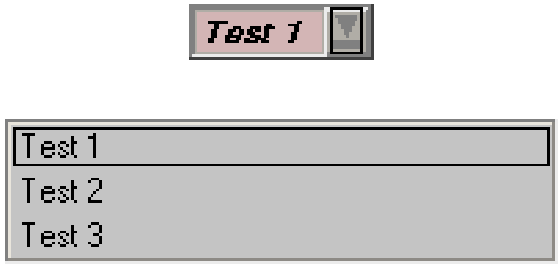
```
QListBox(QWidget *p);
```

  - Slot

```
void clear();
```

  - Signaux

```
void selected(int);
void selected(QString);
```



- `QComboBox` : liste déroulante (éditable)

```
QComboBox(QWidget* p);
```

  - Slot/Signaux (cf. `QListBox`)

La saisie

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- `QLineEdit` : pour du texte ou des valeurs

```
QLineEdit(QString c,QWidget* p);
```

  - Slots :

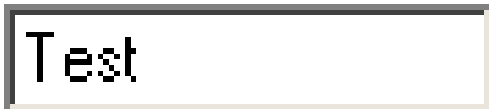
```
void setText(QString);
void clear();
```

  - Signaux :

```
void textChanged(Qstring);
void returnPressed();
```

  - Pour des valeurs entières ou flottantes :

```
void setValidator(QValidator* val);
QIntValidator, QDoubleValidator
```



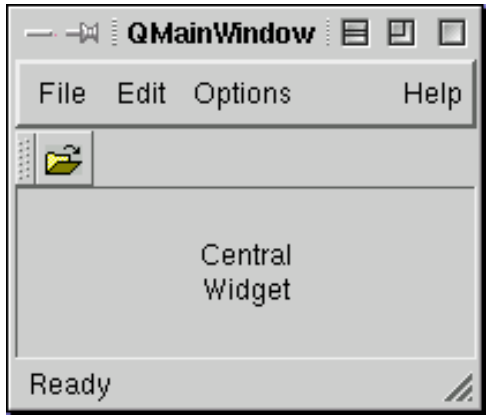
QMainWindow

Généricité, template et namespace
Template
Namespace
Exceptions en C++
Patron de conception
Les principaux patrons de conception
La STL
L'API STL
Qt
Généralités
Architecture objet de Qt
Quelques widgets
Placer les widgets

- `QMainWindow` : fenêtre

  `QMainWindow(QWidget *p);`

- Contient :
  - Une barre de menu : `QMenuBar` ;
  - Une barre d'état : `QStatusBar` ;
  - Une widget centrale : `QWidget`.



Dimensionner un widget

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- A la création d'une widget on indique la widget parente.
- Les widgets sont dessinées à l'intérieur de l'espace attribué à leur parente.
- Les méthodes `setGeometry(const Qrect &)` ou `setGeometry(int x, int y, int w, int h)` permettent de positionner et de dimensionner une widget. La position est relative à celle de la widget parente.

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qlineedit.h>

int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    QWidget contenu(0);
    QLineEdit txt(&contenu);
    QPushButton efface("efface",&contenu);
    QPushButton quitter("quitter",&contenu);

    contenu.setGeometry(20,20,200,200);
    quitter.setGeometry(10,10,100,30);
    efface.setGeometry(10,50,100,30);
    txt.setGeometry(10,100,100,30);

    QObject::connect(&efface,SIGNAL(clicked()),
                     &txt,SLOT(clear()));
    QObject::connect(&quitter,SIGNAL(clicked()),
                     &a,SLOT(quit()));

    contenu.show();
    return a.exec();
}
```

Placer les widgets

Généricité, template et namespace
- Template
- Namespace

Exceptions en C++

Patron de conception
- Les principaux patrons de conception

La STL
- L'API STL

Qt
- Généralités
- Architecture objet de Qt
- Quelques widgets
- Placer les widgets

- Les classes de Layout simplifie ce travail d'organisation.
- On peut ajouter des widgets dans un layout :

```
void QLayout::addWidget(QWidget * w)
```