

Le Langage C++

**Support de cours
(Copie des transparents)**

C.Ernst

Novembre 2014

Le Langage C++ (1)

Contenu :

- **Présentation de la norme C++03**
- **Introduction à la STL**
- **Cours non orienté bibliothèques spécifiques (MFC, ...)**

Le Langage C++ (2)

Plan :

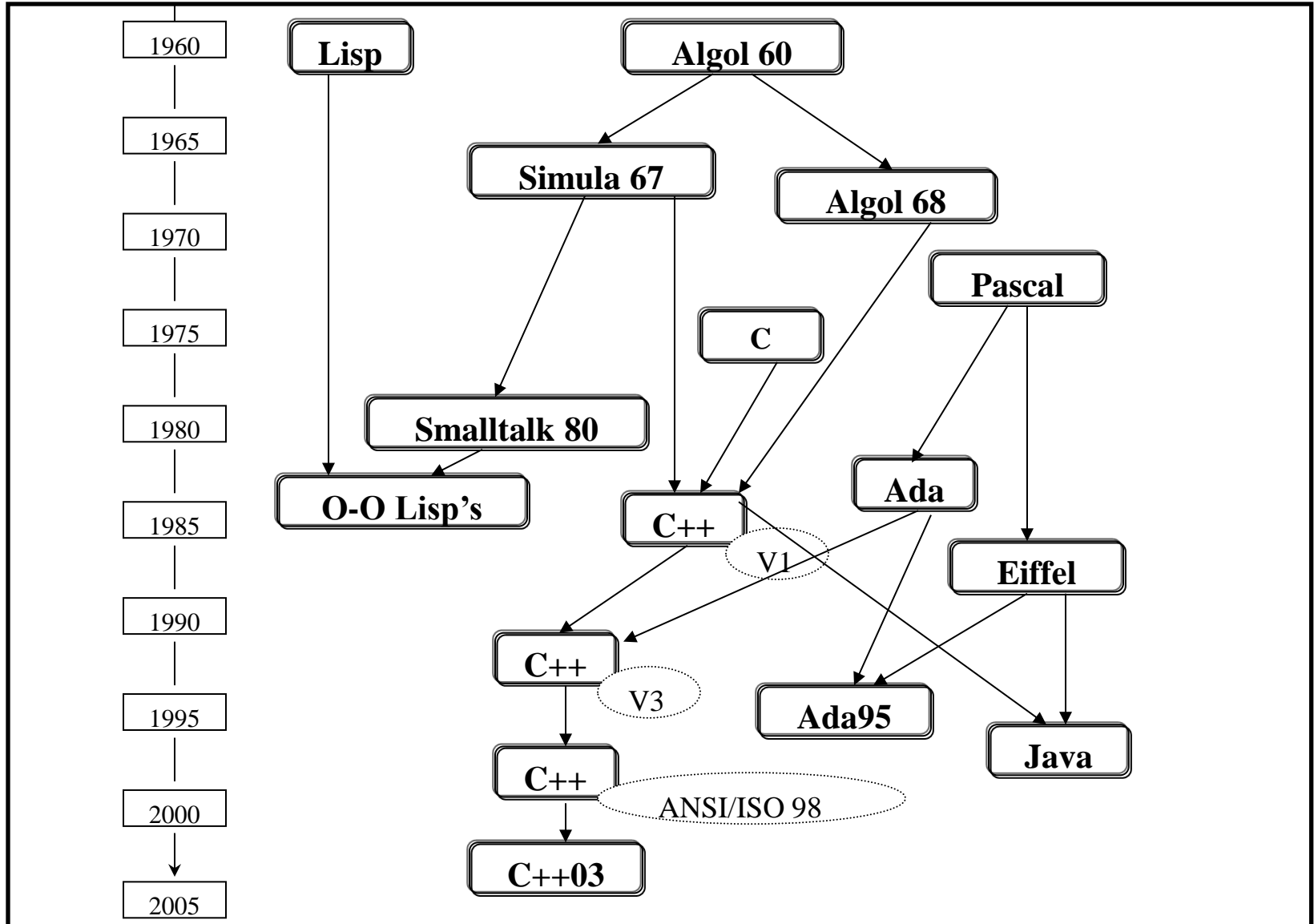
- **S1 : Généralités**
- **S2-S3 : Types utilisateurs (Classes)**
- **S4 : Surcharge d'opérateurs**
- **S5-S6 : Héritage (simple, multiple)**
- **S7 : Espaces de noms – Entrées/Sorties**
- **S8 : Patrons – Exceptions – Système de typage**
- **S9 : C++ & la STL**
- **S10 : C++ & UML**

+ Test final (3h)

SEANCE 1 : GENERALITES

- Historique
- Concepts du langage
- Constructions C++ devenues C
- Constructions C revisitées
- Apports non orientés objets de C++ à C
- Entrées / Sorties basiques

HISTORIQUE (1)



HISTORIQUE (2)

- Auteur : B. STROUSTRUP, Bell-AT&T (1983)
- Caractéristiques de C++ :
 - sur-ensemble de C, typage fort de l'information
 - permet la définition de concepts non facilement représentables en C
 - gère la complexité en structurant des concepts dépendants en graphes
 - autorise la généralisation (familles de concepts)

PROGRAMMATION PAR OBJETS

C++ supporte deux styles de programmation :

- la programmation procédurale, et
- la programmation orientée objet

La POO repose sur deux concepts :

- les **abstractions de données**
- l'**héritage**

ABSTRACTIONS DE DONNEES

TAD = type défini par l'utilisateur + opérations

- Classe : construction de C++ associée
une classe est formée d'une liste de membres qui peuvent être des données (les propriétés) et / ou des fonctions (les opérations)
- Objet : instance d'une classe
- Encapsulation : mécanisme de contrôle des accès aux membres d'une classe

HERITAGE

Classe dérivée : construction C++ autorisant l'ajout de membres à une classe existante (appelée classe de base)

Héritage : mécanisme qui fait qu'une classe dérivée dispose des membres de la classe de base comme s'ils lui étaient propres

POLYMORPHISME (1)

Propriété connexe des langages OO

Définition :

Caractéristique d'un langage qui fait que certaines de ses constructions ont un sens différent selon leur contexte d'utilisation

POLYMORPHISME (2)

Polymorphisme de fonction : mécanisme qui autorise une classe dérivée à redéfinir le sens de fonctions membres de la classe de base

➤ construction C++ : **fonction virtuelle**

Polymorphisme des opérateurs : mécanisme permettant la redéfinition des modalités d'utilisation de la plupart des opérateurs (arithmétiques, logiques, ...) du C

➤ construction C++ : **surcharge d'opérateur**

AUTRES FONCTIONNALITES

- **Patron** : permet de créer des familles de classes ou de fonctions de façon paramétrée (pour des types d'éléments individuels à préciser ensuite)
 - **Gestion des exceptions** : mécanisme de contrôle "d'évènements exceptionnels" pouvant se produire à l'exécution
- + **Espaces de noms** : la « modularité » selon C++

CONSTRUCTIONS C++ DEVENUES C

- Types **void** et **bool**
- Spécificateurs **const** et **volatile**
- Commentaires (`// ...`), spécificateur **asm**
- Déclarations / définitions de fonctions
- Déclarations d'objets
- Fonctions **inline**

DECLARATIONS "D'OBJETS"

N'introduire les variables que là où utilisées :

```
void f ()
{
    // ...
    double db = sqrt(23);
    for (int i = 0; i < 100; i++)
        for (int j = 2; j < 80; j++)
        {
            char tab[12];
            // utiliser tab ...
        }
}
```

SPECIFICATEUR INLINE

Placé devant une déclaration de fonction, il s'agit d'une alternative aux macros du C :

```
inline fact (int i)
{
    return i == 0 ? 1 : i * fact (i - 1);
}
```

→ Génère du code « optimisé »

CONSTRUCTIONS C REVISITEES (1)

Enumérations (1) : alternative aux **const**

```
enum { PUBLIC, PRIVATE, PROTECTED};
```

Équivalent à

```
const PUBLIC = 0;
```

```
const PRIVATE = 1;
```

```
const PROTECTED = 2;
```


CONSTRUCTIONS C REVISITEES (2)

Enumérations (2) nommées → nouveaux types

```
enum mot_cle { PUBLIC, PRIVATE, PROTECTED};
```

```
mot_cle mc = PUBLIC;
```

```
int i = PROTECTED;
```

```
mc = i;                /* erreur */
```

```
mc = mot_cle (i);
```

```
enum val { v1, v2, v3 = 6, v4}; // renumérotation
```

CONSTRUCTIONS C REVISITEES (3)

Typedef & Portée

```
typedef double db;
typedef struct { /* ... */ } db;    // ko en C++

struct st
{
    db x;
    double db;    // ko en C++, ok en C99
    struct T { int a; };
};

typedef struct str { /* ... */ } str; // ok

struct T t;    // ko en C++, ok en C99
```

MOTS CLES SPECIFIQUES A C++

catch	friend	new	public	try
class	inline	operator	template	typeid
delete	mutable	private	this	using
explicit	namespace	protected	throw	virtual

+ les opérateurs étendus de “**cast**”

OPERATEUR ::

Opérateur global (Ex.) ou qualificatif :

- permet de lever certaines ambiguïtés

```
int x;
```

```
void f ()
```

```
{
```

```
    int x = 1; // masque le 'x' global
```

```
    ::x = 2; // affecte au 'x' global
```

```
}
```

OPérateurs NEW & DELETE

Remplacent les fonctions *malloc()* et *free()* du C

```
int* p = new int;
```

```
char* ptab = new char[10];
```

```
...
```

```
delete p;
```

```
delete [] ptab;
```

NB : Ecrire ... `if (p)` au lieu de `if (p != NULL)`

REFERENCES (1)

Une **référence** est un nom alternatif pour un objet, et se note *nom_type*&

Une référence doit être initialisée avec l'objet auquel elle fait référence

```
int i = 1;
int& r = i;    // 'r' et 'i' désignent le
               // même entier
r++;           // incrémente 'i' de 1
```

L'identifiant référencé ne peut être changé après initialisation

REFERENCES (2)

`int& r = i;` est équivalent à `int* r = &i;`
mais `r++` incrémente 'i' et non une adresse

Intérêt : permet d'éviter de passer des pointeurs en argument des fonctions dont le rôle est de modifier la valeur des objets pointés

```
void incr (int& aa)
{
    aa++;
}
```

```
void f()
{
    int x = 1;
    incr (x); // x == 2
}
```

ARGUMENTS PAR DEFAUT

On peut initialiser par défaut les derniers arguments d'une fonction

Un appel à la fonction peut alors se faire sans préciser l'un ou l'autre de ces arguments

```
void affiche (int val, int base = 10);
```

```
void f ()  
{  
    affiche (31);           // affiche 31  
    affiche (31, 10);       // affiche 31  
    affiche (31, 16);       // affiche 1F  
}
```


SURCHARGE DE FONCTIONS

C++ permet d'utiliser un même nom de fonction pour des opérations sur des types différents

Chacune de ces opérations doit donner lieu à définition d'une version spécifique de la fonction

```
void affiche (double);  
void affiche (long);  
  
void f ()  
{  
    affiche (1.0);    // affiche (double)  
    affiche (1L);     // affiche (long)  
}
```

ENTREES / SORTIES DE BASE (1)

La bibliothèque *iostream*

- définit les périphériques standard de sortie, d'entrée et d'erreur sous forme de flux (des objets) nommés **cout**, **cin** et **cerr**
- gère ces flux au travers d'opérateurs surchargés du langage (applicables à tous les types de base) :
 - << : opérateur d'insertion
 - >> : opérateur d'extraction

ENTREES / SORTIES DE BASE (2)

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

void saisie ()
{
    int n;

    cout << "Entrer un nombre : ";
    cin >> n;
    cout << "Valeur saisie : " << n << '\n';
    // ...
}
```

FICHIERS D'EN-TETE (1)

Les extensions «.h» des fichiers d'en-tête système ont disparu. La librairie C++ standard, ou STL, est définie dans un **namespace** unique appelé *std*

```
#include <iostream>    // Option 1
void foo () { std :: cout << "Salam, Aleikoum!\n"; }
```

```
#include <iostream>    // Option 2
using std :: cout;
void foo () { cout << "Salam, Aleikoum!\n"; }
```

```
#include <iostream>    // Option 3
using namespace std;
void foo () { cout << "Salam, Aleikoum!\n"; }
```

FICHIERS D'EN-TETE (2)

BonExempleInclude.C

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

void foo ()
{
    srand((unsigned) time(NULL));
    int i = (rand() % 30);

    cout << " Valeur : " << i << "\n";
}
```

SEANCE 2 : TYPES UTILISATEURS (1)

- Pourquoi les TAD's
- Classes & objets
- Fonctions membres
- Constructeurs & Destructeur
- Auto-référence

INTERET DES TADs (1)

(Pb1) : Il n'existe aucun lien entre données et opérations qui les manipulent dans les langages procéduraux

```
struct date    // en C ...
{
    int  jour, mois, annee;
};

void maj_date (struct date*, int, int, int);
void affiche_date (struct date*);
```

INTERET DES TADs (2)

(Sol1) : C++ permet de déclarer des fonctions comme **membres** d'une structure

```
struct date
{
    int        jour, mois, annee;
    void        maj (int, int, int);
    void        affiche ();
} d;
```

Définition et invocation d'une fonction membre :

```
void date :: affiche () // définition
{
    cout << jour << '/' << mois << '/' << annee;
}
// ...
d.affiche ();           // invocation
```


INTERET DES TADs (3)

(Pb2) : Les langages procéduraux ne permettent pas de contrôler l'accès aux données :

```
int degre = 32;

void f(struct date *refd)          // en C ...
{
    refd->jour = degre;
    //
}
```

➤ Les données d'une structure de type *date* peuvent être manipulées (en C) par « qui le veut »

INTERET DES TADs (4)

(Sol2) : C++ fournit un mécanisme d'encapsulation des données pour en restreindre l'accès :

```
class date
{
    int    jour, mois, annee;
public :
    void maj (int, int, int);
    void cour (int*, int*, int*);
    void affiche ();
};
```

Les membres privés ne peuvent être utilisés que par les autres membres de la classe. Seuls les membres publics sont accessibles de « l'extérieur »

DECLARATION D'UNE CLASSE

```
class nom_classe    // en attendant mieux ...  
{  
  private :          // label facultatif  
    // déclaration de données (privées)  
    // déclaration de fonctions (privées)  
  public :  
    // déclaration de données  
    // déclaration de fonctions  
};
```

Les **struct** et **union** sont des **class** dont les membres sont **public** par définition

OBJETS D'UNE CLASSE

Objets : instances d'une classe (ex- "variables")

- Déclarations "statiques" :

```
date xobj;  
date tobj[10];  
date& robj = tobj[2];  
date* pobj = &xobj;
```

- Objets alloués dynamiquement :

```
date* ptr = new date;  
date* ptab = new date[20];
```

FONCTIONS MEMBRES (1)

Définition :

- dans la déclaration de classe (fonctions **inline**)
- ou après la déclaration de la classe

```
class date {  
    int jour, mois, an;  
public :  
    void maj (int, int, int);    // déf. à venir  
    void affiche () {  
        cout << jour << '/' << mois << '/' << an;  
    }  
};
```

FONCTIONS MEMBRES (2)

```
void date :: maj (int j, int m, int a)
{
    // définition de la fonction
    jour = j;
    mois = m;
    an = a;
    // ...
}
```

Invocation d'une fonction : sur des objets de sa classe

```
date anniv_julie;

anniv_julie.maj (10, 12, 1964);
anniv_julie.affiche ();
```

CONSTRUCTEUR & DESTRUCTEUR DE CLASSE (1)

Un constructeur d'une classe X est une fonction membre de X permettant "d'initialiser" les autres membres de X lors de la création d'objets de type X

Syntaxe : X :: X (/* ... */) { /* ... */ }

Une fonction destructeur est une méthode symétrique appliquée lors de la "disparition" d'un objet

Syntaxe : X :: ~X () { /* ... */ }

CONSTRUCTEUR & DESTRUCTEUR DE CLASSE (2)

- Une classe peut disposer de n constructeurs ($n \geq 0$), mais d'un seul destructeur au plus; ces fonctions ne peuvent être typées, ni exécuter d'instruction *return*;
- Toute déclaration d'objet d'une classe disposant d'au moins un constructeur doit être réalisée au moyen de l'un de ces constructeurs
- Le constructeur adéquat (resp. le destructeur) de la classe d'un objet est automatiquement invoqué lors de la création (resp. de la destruction) de cet objet

CONSTRUCTEUR & DESTRUCTEUR DE CLASSE (3)

```
class pile_car
{
    int taille;
    char* cour;
    char* base;
public :
    pile_car (int tai) {
        cour = base = new char[taille = tai];
    }
    ~pile_car () { delete[] base; }
    void empiler (char c) { *cour++ = c; }
    char depiler () { return *--cour; }
};

void f ()
{
    pile_car pc1 (100), pc2 (200);

    pc1.empiler ('a');
    pc2.empiler (pc1.depiler ());
    char c = pc2.depiler ();
}
```

AUTO-REFERENCE

Une fonction membre peut référencer les membres de l'objet sur lequel elle est invoquée via le pointeur **this** (un pointeur sur cet objet)

```
class X
{
    int m;
public :
    int readm () const { return m; }
};

class X      // déclaration équivalente
{
    int m;
public :
    int readm () const { return this->m; }
};
```

La déclaration implicite de **this** est **X* const this;**

AUTO-REFERENCE (Ex 1)

```
class dchaine {
    dchaine* pre;        // maillon précédent
    dchaine* sui;        // maillon suivant
public :
    void insere (dchaine*);
};

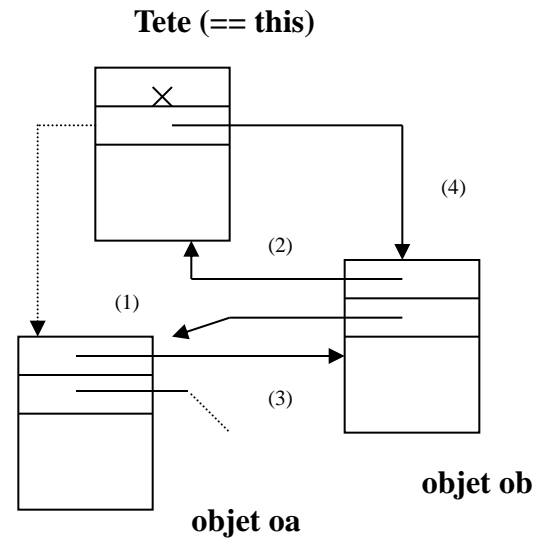
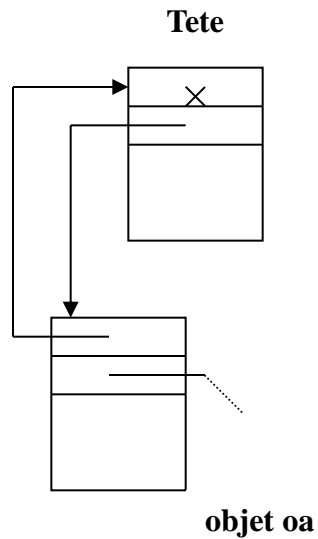
void dchaine :: insere (dchaine* p)
{
    p->sui = sui;        // p->sui = this->sui    (1)
    p->pre = this;      // appel explicite        (2)
    sui->pre = p;        // this->sui->pre = p      (3)
    sui = p;            // this->sui = p          (4)
}

dchaine* tete;

void f (dchaine* oa, dchaine* ob)
{
    tete->insere (oa);
    tete->insere (ob);
}
```

AUTO-REFERENCE (Ex 2)

Résultat :



SEANCE 3 : TYPES-UTILISATEURS (2)

- Fonctions amies
- Membres statiques
- Objets membres
- Constructeur par copie

FONCTIONS AMIES (1)

Les fonctions amies d'une classe sont autorisées à accéder aux membres privés des objets de cette classe

```
class nombre {  
    int n;  
public :  
    nombre (int num = 0) { n = num;}  
    friend void affiche (nombre);  
};  
  
void affiche (nombre num) {  
    cout << num.n << '\n';    // ok  
}  
  
int main () {  
    nombre y (10);  
  
    affiche (y);        // affiche 10  
}
```

FONCTIONS AMIES (2)

Une fonction membre d'une classe peut être l'amie d'une autre classe :

```
class X {          |   class Y {  
    void f ();      |       friend void X :: f ();  
};                 |   };  
  
+  
  
class X  
{  
    friend class Y;  
    // si toutes les fcts de Y sont amies de X  
};
```

Une fonction (non membre) **friend** ne dispose pas de pointeur **this**

Une déclaration **friend** peut être placée n'importe où

FONCTIONS AMIES (3)

```
class Matrice;           // déclaration "forward"

class Vecteur {
    float Vec[3];
    friend Vecteur multiplie (const Matrice&, const Vecteur&);
};

class Matrice {
    Vecteur Vec[3];
    friend Vecteur multiplie (const Matrice&, const Vecteur&);
};

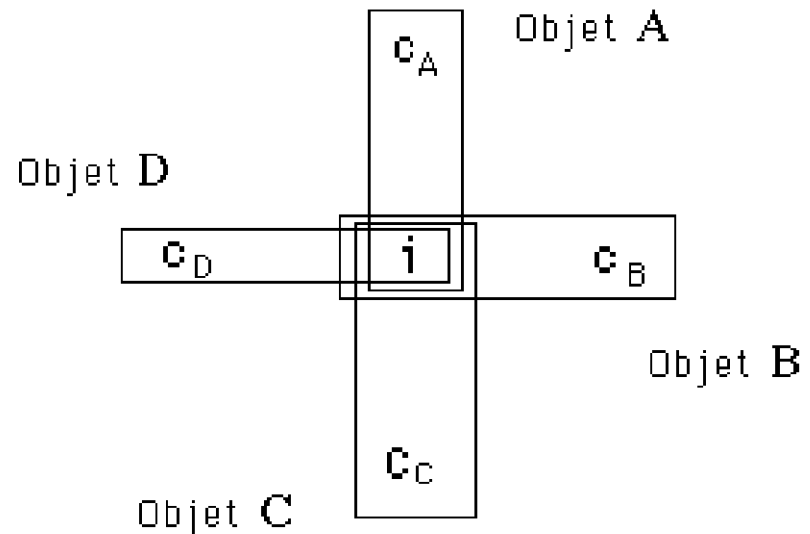
Vecteur multiplie (const Matrice& M, const Vecteur& V)
{
    Vecteur Res;
    for (int i = 0; i < 3; i++)
    {
        Res.Vec[i] = 0;
        for (int j = 0; j < 3; j++)
            Res.Vec[i] += M.Vec[i].Vec[j] * V.Vec[j];
    }
    return Res;
}
```


MEMBRES STATIQUES (1)

Chaque instance d'une classe X possède sa propre copie de toutes les données définies dans X

Pour qu'un membre de X soit unique et partagé par tous les objets de X, il faut le "préfixer" avec **static**

```
class abcd {  
    char c;  
public :  
    static int i;  
} A, B, C, D;
```



MEMBRES STATIQUES (2)

Une donnée membre statique d'une classe (objet global) doit être initialisée avant de pouvoir être manipulée :

```
int abcd :: i = 1;  // création et initialisation
```

L'accès aux membres statiques publics d'une classe X ne peut se faire qu'en les qualifiant avec X :

```
if (abcd :: i == 0)
    // faire quelque chose
```

OBJETS MEMBRES (1)

Soient

```
class Point
{
    int abs, ord;
public :
    Point (int a, int o) { abs = a; ord = o; }
};

class Cercle
{
    Point centre;           // 'centre' est un objet
    double rayon;
public :
    Cercle (int, int, double);
};
```

PB : comment initialiser la donnée *centre* ?

OBJETS MEMBRES (2)

Solution : ... avec un Point

```
Cercle :: Cercle (int a, int o, double r)
    : centre (a, o)
{
    rayon = r;
}
```

Cette construction / syntaxe est la seule possible, et peut être étendue à un ensemble de membres :

```
Cercle :: Cercle (int a, int o, double r)
    : centre (a, o), rayon (r)
{
}
```

CONSTRUCTEUR PAR COPIE (1)

L'affectation est une copie membre à membre :

```
Cercle c1 (2, 4, 7.1), c2 (8, 5, 2.4);  
c1 = c2;
```

L'initialisation est une opération différente, s'appliquant à des objets déjà définis via un constructeur dit de copie :

```
Cercle c3 = c2;  ⇔  Cercle c3 (c2);
```

Le constructeur par copie Cercle (**const** Cercle&) est ajouté par le compilateur à la classe sauf si une définition explicite de ce constructeur est fournie par la classe

CONSTRUCTEUR PAR COPIE (2)

Le constructeur de copie par défaut effectue une copie membre à membre des objets destination / source

Autre version du constructeur de *Cercle* possible :

```
Cercle :: Cercle (const Point& P, double r)
           : centre (P), rayon (r)
{
}
```

car *Point* dispose du constructeur *Point* (**const** *Point*&)

L'utilisation de l'affectation et de l'initialisation peut être source d'erreurs (cf. S4 : surcharge)

SEANCE 4 : SURCHARGE D'OPERATEUR

- Principe(s)
- Exemples
 - Gestion de complexes
 - Conversions explicites et implicites
 - Affectation et initialisation
 - Une classe *string*

SURCHARGE D'OPÉRATEUR : PRINCIPE

La surcharge d'un opérateur @ de C++ pour une classe X permet de donner un sens particulier à @ lorsqu'il est appliqué à des objets de X, via une fonction **operator@** :

```
class X {  
    // ...  
public :  
    X operator@ (X);    // ex. de déclaration  
} a, b, c;  
  
X X :: operator@ (X x) { /* ... */ } // définition  
  
c = a.operator@ (b);    // appel classique (sans intérêt)  
c = a @ b;              // appel conventionnel
```


MISE EN ŒUVRE DE LA SURCHARGE

Une fonction **operator** de X peut être une fonction

- amie de X; sa liste d'argument(s) doit alors comporter au moins un objet de X
- membre de X; son premier argument est alors implicitement une instance de X (un objet de type X)

Elle ne peut changer

- ni les règles de précedence existantes ($b + c * a$ vaudra toujours $b + (c * a)$)
- ni la syntaxe des expressions formées ('!' ne deviendra jamais un opérateur binaire)!

GESTION DE COMPLEXES

```
class Cx {  
    double re, im;  
public :  
    Cx (double r, double i) { re = r; im = i;}  
    friend Cx operator+ (Cx, Cx);  
    Cx operator* (Cx);  
};  
  
Cx operator+ (Cx a, Cx b) {  
    return Cx (a.re + b.re, a.im + b.im);  
}  
  
inline Cx Cx :: operator* (Cx c) {  
    return Cx (re * c.re - im * c.im, re * c.im + im * c.re);  
}  
  
int main () {  
    Cx a (12, 13.1); Cx b = Cx (8.7, 12.45);  
  
    b = b + a;  
    Cx c = a * b + Cx (1, 2);  
}
```

CONVERSIONS EXPLICITES (1)

Une fonction membre $S :: \text{operator } D ()$, où D est un nom de type, définit une conversion pour le type source S en type D

Cette conversion est réalisée automatiquement à chaque apparition d'objets de type S dans les expressions où un type D est nécessaire

CONVERSIONS EXPLICITES (2)

```
class petiti
{
    unsigned char v;
    void assigne (int i) { v = (i > 63) ? 0 : i;}
public :
    petiti (int i) { assigne (i);}
    petiti& operator= (int i) {
        assigne (i); return *this;}
    operator int () { return v;}
};

int main ()
{
    petiti c1 = 2;      // ou petiti c1 = petiti (2);
    petiti c2 (62);
    petiti c3 = c1 + c2;    // c3.v == 0
    int i = c1 + c2;        // i == 64
    petiti& c4 = c3.operator= (12); // c4.v == c3.v == 12
}
```

CONVERSIONS IMPLICITES

```
class Cx {
    double re, im;
public :
    Cx (double r, double i = 0) { re = r; im = i;}
    friend Cx operator* (Cx, Cx);
};

void g (double d1, double d2)
{
    Cx c1 = 23.2;          // Cx c1 = Cx (23.2)

    Cx c2 = c1 * 2;
    // Cx c2 = operator* (c1, Cx (double(2), double(0)));

    Cx c3 = d1 * d2;
    // multiplication en double précision

    Cx c4 = Cx (d1) * d2;
    // force « l'arithmétique » de 'Cx'
}
```

AFFECTATION & INITIALISATION (1)

Soit

```
class string
{
    char *p;
    int t;    // taille de la chaîne
public :
    string (int tai) { p = new char[t = tai]; }
    ~string () { delete [] p; }
};
```

Pb1 : libération de mémoire

```
void f ()
{
    string s1 (10);    // utilisation de
    string s2 (10);    // string :: string (int)

    s1 = s2;           // → perte de s1.p
}    // destruction de s2 puis s1 → Pb1
```

AFFECTATION & INITIALISATION (2)

Sol1 (à Pb1) : redéfinition de l'affectation

```
class string {  
    char *p;  
    int t;  
public :  
    string (int tai) { p = new char[t = tai];}  
    ~string () { delete [] p;}  
    string& operator= (const string&);  
};  
  
string& string::operator= (const string& s)  
{  
    if (this != &s)           // attention à s = s;  
    {  
        delete [] p;  
        p = new char[t = s.t];  
        strcpy(p, s.p);  
    }  
    return *this;  
}
```

AFFECTATION & INITIALISATION (3)

Pb2 : utilisation de l'opérateur d'initialisation

```
void f () {
    string s1 (10);
    string s2 = s1;
} // destruction de 's2' puis de 's1'
```

's2' est créé à partir du constructeur de copie par défaut
(**operator=** ne s'applique qu'à des objets définis)

Sol2 (à Pb2) : redéfinition du constructeur par copie

```
class string {
    char *p;    int t;
public :
    string (const string& s) {
        p = new char[t = s.t]; strcpy(p, s.p);
    };
};
```

```
string s2 = s1;    est interprété par  string s2 = string (s1);
                   soit                string s2 (s1);
```


class *string* (2)

```
string& string :: operator= (const char* s)
{
    if (b)
        delete [] ch;
    else
        b = true;
    ch = new char[strlen(s) + 1];
    strcpy(ch, s);
    return *this;
}

string& string::operator= (const string& x)
{
    if (this != &x && b) {
        delete [] ch;
        b = false;
    }
    ch = x.ch;
    return *this;
}
```

class *string* (3)

```
istream& operator>> (istream& st, string& x)
{
    char buf[256];

    st >> buf;    // entrée de caractères
    x = buf;      // operator= (const char*)
    return st;
}
```

```
char& string :: operator[] (int i)
{
    if (i < 0 || strlen (ch) < i)
        // faire quelque chose
    return ch[i];
}
```

class *string* (4)

```
int main ()
{
    string x, tab[10];
    string y = "fin";

    cin >> x;    // operator>>
    for (int i = 0; i < 3 && i < 9; i++)
    {
        tab[i] = x;
                // operator= (const string&)
        if (x[i] == y[i]) continue;
                // operator[] et
                // comparaison de caractères
        cout << "Chaînes distinctes\n";
        break;
    }
    if (tab[i] == y)    // operator==
        cout << x;
    // ...
}
```

SEANCE 5 : HERITAGE SIMPLE

- Classes dérivées & héritage
- Héritage simple, fonctions membres
- Héritage & constructeurs
- Fonctions virtuelles
- Fonctions virtuelles pures & classes abstraites

CLASSES DERIVEES & HERITAGE (1)

Une classe dérivée D d'une classe existante B permet d'ajouter des membres à B sans modifier B !

```
class Base {  
    /* partie privée */  
    // ...  
public :  
    int m, n, p;  
};
```

```
class Derivee : public Base {  
    /* partie privée */  
    // ...  
public :  
    int p, q, r;  
};
```



Dérivation publique

CLASSES DERIVEES & HERITAGE (2)

L'héritage est le mécanisme OO qui permet d'utiliser dans D certains membres de B comme ... s'ils étaient définis dans D

Les membres publics d'une classe de base peuvent être référencés comme des membres publics de la classe dérivée (sauf en cas de redéfinition)

```
Derivee Der;
```

```
Der.m = 1;           // 'm' de 'Base'
```

```
Der.p = 4;           // 'p' de 'Derivee'
```

```
Der.Base :: p = 3;    // 'p' de 'Base'
```

HERITAGE, POINTEURS & CONVERSIONS (1)

Un objet d'une classe dérivée peut être manipulé comme un objet de la classe de base; l'inverse nécessite une conversion explicite de type

Ex :

```
Derivee Der;  
Base* pBas = &Der;           // ok  
Derivee* pDer1 = pBas;       // erreur  
Derivee* pDer2 = (Derivee*) pBas;  
                // ok, mais dangereux ...
```


HERITAGE SIMPLE

Une classe dérivée peut devenir une classe de base :

```
class Employe { /*...*/ };
```

```
class Manager : public Employe { /*...*/ };
```

```
class Directeur : public Manager { /*...*/ };
```

La hiérarchie de classes obtenue représente un arbre, voire un graphe

L'**héritage** entre classes est dit **simple** lorsque toute classe dérivée est issue d'exactement une classe de base

CLASSES DERIVEES & FCT^s MEMBRES (1)

Les fonctions membres d'une classe dérivée se comportent de façon usuelle :

```
class Employe {  
    char* nom;  
    // ...  
public :  
    Employe* suivant;  
    void affiche () const {  
        cout << "nom : " << nom << "\\n";  
    }  
};  
  
class Manager : public Employe {  
    // ...  
public :  
    void affiche () const;  
};
```

CLASSES DERIVEES & FCT^s MEMBRES (2)

2 fonctions déclarées de façon identique dans 2 classes différentes doivent donner lieu à 2 définitions distinctes :

```
void Manager :: affiche () const
{
    cout << "nom : " << nom << '\n';
    // erreur : 'nom' est privé dans Employe
}
```

mauvais

```
void Manager :: affiche () const
{
    Employe :: affiche ();
    // et non pas affiche ();
    // c.à.d. Manager :: affiche ();
    ...
    // afficher les infos propres au Manager
}
```

bon

HERITAGE, POINTEURS & CONVERSIONS (2)

Des erreurs d'utilisation de fonctions disposant de plusieurs versions peuvent surgir lors de manipulations de pointeurs :

```
Manager M;  
Employe* PtrE = &M;
```

L'invocation de `PtrE->affiche ()` est interprétée comme un appel à `Employe :: affiche ()` → souci potentiel

HERITAGE & CONSTRUCTEURS (1)

Une classe dérivée doit fournir au moins un constructeur lorsque sa classe de base possède → plus d'un constructeur, ou
→ un constructeur ayant besoin d'argument(s)

```
class Employe {  
    char* nom;  
    int departement;  
public :  
    Employe (char*, int);  
};  
  
Employe :: Employe (char* n, int d)  
    : nom (n), departement (d)  
{ /* ... */ }
```

Syntaxe identique que
pour les objets membres

```
class Manager : public Employe {  
    int qualification;  
public :  
    Manager (char*, int, int);  
};  
  
Manager :: Manager (char* n, int q, int d)  
    : Employe (n, d), qualification (q)  
{ /* ... */ }
```

FONCTIONS VIRTUELLES : PRINCIPE

Le mécanisme des **fonctions virtuelles** permet de redéfinir dans une classe dérivée une fonction déjà définie dans la classe de base

```
class Employe
{
    // ...
public :
    // ...
    virtual void affiche ();
};
```

Il garantit avant tout une correspondance correcte entre fonctions et objets utilisés

Le choix de la version de la fonction appelée est réalisé à l'exécution ("*late-binding*")

FONCTIONS VIRTUELLES : REGLES D'UTILISATION

- Le type d'une fonction virtuelle ne peut être modifié dans la classe dérivée
- Une fonction virtuelle doit être définie pour la classe dans laquelle elle est déclarée
- Une classe dérivée peut ne pas fournir de version spécifique d'une fonction virtuelle de la classe de base
- L'emploi de **virtual** devant une fonction membre est possible même si les classes dérivées susceptibles de la redéfinir n'existent pas (... encore)

FONCTIONS VIRTUELLES : EXEMPLE (1)

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

class Employe {
    char* nom;
    int departement;
    Employe* suivant;
    static Employe* list;           // déclaration
public :
    Employe (char*, int);
    static void aff liste ();
    virtual void affiche () const;
};

Employe* Employe :: list = 0;      // définition

class Manager : public Employe {
    int qualification;
public :
    Manager (char*, int, int);
    void affiche () const;
};
```

(rappel)

FONCTIONS VIRTUELLES : EXEMPLE (2)

```
Employe :: Employe (char* n, int d)
    : nom (n), departement (d)
{
    suivant = list;
    list = this;
}

void Employe :: aff_liste ()
{
    for (Employe* p = list; p; p = p->suivant)
        p->affiche ();
}

Manager :: Manager (char* n, int q, int d)
    : Employe (n, d), qualification (q) {}

int main () {
    Employe e ("Job", 4);
    Manager m ("Cresus", 7, 4);

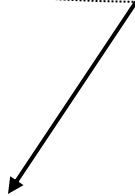
    Employe :: aff_liste ();
    return 0;
}
```

FONCTIONS VIRTUELLES PURES & CLASSES ABSTRAITES

Certaines classes ne représentent que des concepts pour lesquels des objets ne peuvent directement exister

Pour éviter de définir des objets pour de telles classes, C++ permet la déclaration de *fonctions virtuelles pures* :

```
class forme
{
    // ...
public :
    virtual void dessine () = 0;
};
```



Une classe ayant au moins une fonction virtuelle pure est dite abstraite. Aucun objet ne peut être créé pour cette classe; une classe dérivée qui ne redéfinit pas ces fonctions reste abstraite

SEANCE 6 : HERITAGE MULTIPLE++

- Héritage multiple : définition, exemple, utilisation
- Classes virtuelles. Résolution d'ambiguïtés
- Héritage multiple & Constructeurs
- Contrôle d'accès & Visibilité


HERITAGE MULTIPLE : DEFINITION

Une classe peut dériver de plus d'une classe de base

Dans ce cas, l'**héritage** entre classes est dit **multiple**

Syntaxe

```
class nom_classe :  
    public cbase1, ..., public cbase n  
{  
    // ...  
};
```



Héritage public

Ce type d'héritage ne fonctionne que sur des structures de graphes acycliques, et doit être utilisé avec précaution

EXAMPLE (1)

```
class rectangle {
    int px, py, larg, haut;
public :
    rectangle (int x, int y, int l = 5, int h = 5);
    int posx (void) { return px;}
    int posy (void) { return py;}
    void affiche (int x, int y);
};

class texte {
    char* contenu;
public :
    texte (char* chaine) { contenu = chaine;}
    void affiche (int x, int y);
};

class bouton : public rectangle, public texte {
public :
    bouton (char*, int, int);
    void affiche ();
};
```

EXAMPLE (2)

```
rectangle :: rectangle (int x, int y, int l, int h)
{
    px = x;      py = y;      larg = l;      haut = h;
}

bouton :: bouton (char* ch, int x, int y)
        : rectangle (x, y), texte (ch) {}

void bouton :: affiche ()
{
    rectangle :: affiche (rectangle :: posx (),
                          rectangle :: posy ());
    texte :: affiche (this->posx () + 2,
                      this->posy () + 2);
}

int main () {
    bouton b ("ISMEA", 5, 8);
    b.affiche ();
    // ...
}
```

REGLES D'UTILISATION

Quand une classe dérive directement d'un certain nombre d'autres classes, celles-ci doivent toutes être distinctes :

```
class A { /* ... */ };  
class B : public A, public A { /* ... */ }; // illégal
```

Une même classe peut toutefois être une classe de base indirecte plus d'une fois :

```
class C : public A { /* ... */ };  
class D : public A { /* ... */ };  
class E : public C, public D { /* ... */ };
```

Un objet de E contiendra deux sous-objets de A (utiliser **virtual** si cette conséquence n'est pas désirée)

CLASSES VIRTUELLES (1)

Deux classes dérivées ayant une classe de base en commun peuvent préfixer cette déclaration avec **virtual** afin de n'utiliser qu'une seule copie de la classe commune

Ex

```
class F : virtual public A { /* ... */ };  
class G : virtual public A { /* ... */ };  
class H : public F, public G { /* ... */ };
```

Un objet de H dérive ici d'un seul objet de A

CLASSES VIRTUELLES (2)

Une classe peut dériver de classes de base virtuelles et non virtuelles :

```
class I : public C, public F, public G {};
```

Un objet de I comportera deux objets de A : l'un de C et le second, virtuel, partagé par F et G

Le spécificateur **virtual** peut également être placé juste devant le nom du type qualifié :

```
class II : public virtual A {};
```

RESOLUTION D'AMBIGUITES (1)

Accès à un membre d'un objet unique (ou non) :

```
class J {public : int v; void f (int);};  
class K {public : int a;};  
class L : public K, public virtual J {};  
class M : public K, public virtual J {};  
  
class N : public L, public M  
{  
  public :  
    void f (int i) {  
      v = i + 2;    // ok, un 'v' dans 'N'  
      J :: f (v);  // ok  
      a++;         // non; 2 'a' dans 'N'  
    }  
};
```

RESOLUTION D'AMBIGUITES (2)

Règle de dominance \rightarrow B :: f domine A :: f
si A est une classe de base virtuelle de B

```
class V { public : int x;};  
class W : public virtual V {  
    public : int x;};  
class X : public virtual V {};  
  
class Y : public W, public X  
{  
    void g ()  
    {  
        x++; // ok, W :: x domine V :: x  
    }  
};
```

HERITAGE MULTIPLE & CONSTRUCTEURS (1)

```
class W {  
    int* pi;  
public :  
    W (int j) { pi = new int[j]; }  
};
```

```
class X : public virtual W {  
public :  
    X (int k)  
        : W (k + 1) {}  
};
```

```
class Y : public virtual W {  
public :  
    Y (int k)  
        : W (k - 2) {}  
};
```

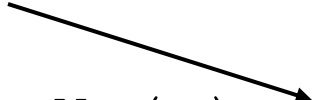
```
class Z : public X, public Y {  
public :  
    Z (int m) : X (m), Y (m) {}  
};
```

PB : un objet de Z dérive ici d'un seul objet de W initialisé deux fois suite aux appels du constructeur de X puis de celui de Y !

HERITAGE MULTIPLE & CONSTRUCTEURS (2)

Sol : les appels au constructeur de W depuis ceux de X et de Y sont supprimés, et le constructeur de Z doit explicitement appeler celui de W :

```
Z :: Z (int m) : X (m), Y (m), W (m)
{
    // ...
}
```



⇒ Le constructeur de la classe la plus dérivée est responsable de l'appel de ceux de toutes les classes de base virtuelles ancêtres dans le graphe d'héritage

CONTRÔLE D'ACCES & VISIBILITE (1)

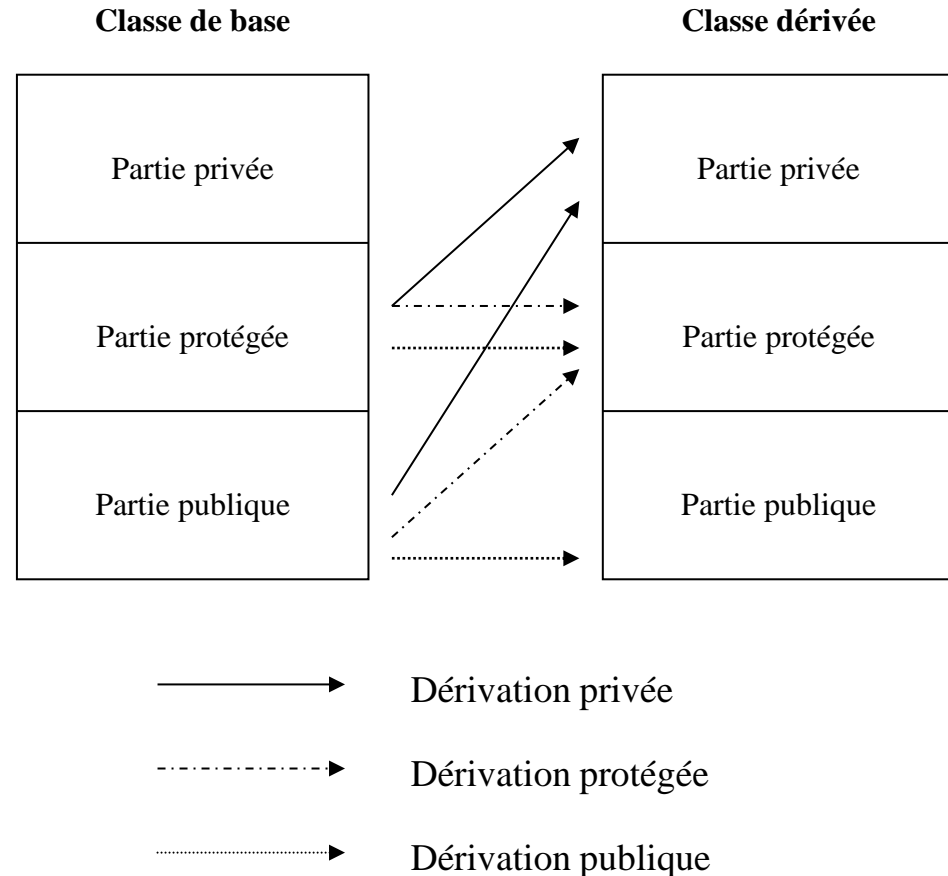
```
class nom_classe // définition formelle
{
private :
    // partie privée
protected :
    // partie protégée
public :
    // partie publique
};
```

Un nom **privé** ne peut être utilisé que par les fonctions membres et amies de la classe d'appartenance

Un nom **protégé** peut être utilisé comme un nom privé ainsi que par les fonctions membres et amies des classes dérivées

CONTRÔLE D'ACCES & VISIBILITE (2)

Règles d'accès lors
d'une dérivation :



DESTRUCTEUR VIRTUEL

L'utilisation d'un **destructeur virtuel** dans la classe de base d'une hiérarchie de classes fournissant chacune un destructeur est vivement conseillée pour permettre une "destruction" correcte des objets :

```
class Employe {  
    // ...  
public :  
    Employe (char*, int);  
    virtual ~Employe () { /* ... */ }  
    // ...  
};
```


SEANCE 7 :

ESPACES DE NOMS - ENTREES/SORTIES

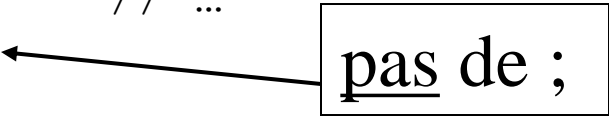
- Espaces de noms : Principes. Mise en œuvre
- Entrées / Sorties : Bibliothèque *iostream*
 - Classes & objets d'*iostream*
 - Éléments d'utilisation
 - Flux : état, format, ...
 - Fichiers & flots
 - Flots & chaînes de caractères en mémoire

NAMESPACES : BASES

Un namespace définit une portée pour les noms qui y sont déclarés

```
namespace monpackage          // déclaration++
{
    class MaClasse
    {
    private:
        int valeur;
    public:
        MaClasse (int v) { valeur = v;}
        int lireValeur( ) { return valeur;}
    };
    // ...
}

void f ()                      // utilisation
{
    monpackage :: MaClasse mc1 (5);
    int i = mc1.lireValeur();
}
```



pas de ;

NAMESPACES : IMPLEMENTATION

Namespace = "package" \neq module C++

MaClasse.h	MaClasse.C
<pre>#ifndef __MaClasse_h__ #define __MaClasse_h__ namespace monpackage { class MaClasse { public : MaClasse (int); int lireValeur (); private : int valeur; }; } #endif</pre>	<pre>#include "MaClasse.h" // autres inclusions de fichiers namespace monpackage { MaClasse :: MaClasse (int i) { valeur = i; } int MaClasse :: lireValeur () { return valeur; } }</pre>

NAMESPACES : UTILISATION (1)

Ex1.C

```
#include "MaClasse.h"
monpackage::MaClasse* uneC = new monpackage::MaClasse(...);
```

Ex2.C

```
#include "MaClasse.h"
using namespace monpackage;

MaClasse* autreC = new MaClasse(...);
```

MAUVAIS MaClasse3.h

```
#include "MaClasse.h"
using namespace monpackage; // NON

namespace xyz
{
    class MaClasse3 : public MaClasse
    {
        ...
    };
}
```

BON MaClasse3.h

```
#include "MaClasse.h"

namespace xyz
{
    class MaClasse3 : // OK
        public monpackage::MaClasse
    {
        ...
    };
}
```

NAMESPACES : UTILISATION (2)

Imbrication de *namespaces* :

Ex4.C

```
using namespace monpackage :: monsouspackage;  
  
MaClasse4* obj = new MaClasse4( ... );
```

Collision de *namespaces* :

Ex5.C

```
...  
using namespace utils; // contenant une classe Liste ...  
class Liste { /* ... */};  
  
utils :: Liste* utilListe = new utils :: Liste( ... );  
:: Liste* appliste = new :: Liste( ... );
```

INCLUSION DE FICHIERS D'EN-TETE

Librairie système : ... **using namespace *std***;

Modules users : ".h" + **using <namespace ...>**

UnBonExemple.C

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
using namespace std;

#include "MaClasse.h"
using monpackage :: MaClasse;

int main()
{
    srand((unsigned) time(NULL));
    MaClasse mc (rand() % 30);

    cout << setw(4) << setfill('0') << mc.lireValeur() << endl;
    return 0;
}
```

LA BIBLIOTHEQUE IOSTREAM

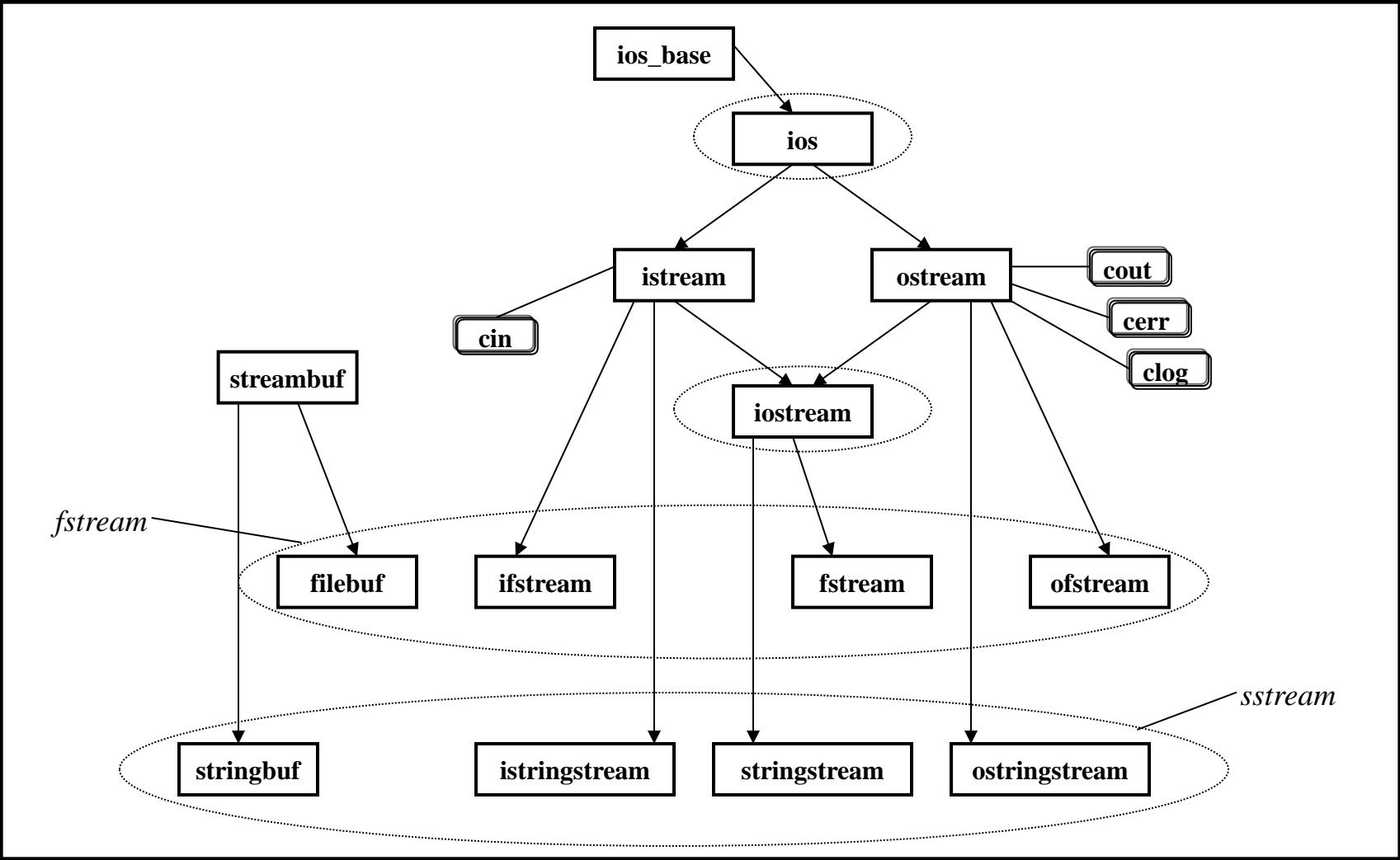
Les Entrées / Sorties C++ sont construites autour de la notion de **flux**

Un flux (ou flot) est un objet qui peut représenter

- la source des opérations de lecture,
- la destination des opérations d'écriture,
- ou les deux à la fois

Un flux peut être attaché à des périphériques, à des fichiers, ou à des buffers en mémoire; il peut être « tamponné » ou non

CLASSES D'E/S & DEPENDANCES



CLASSES D'E/S : UTILISATION

Ex : `<fstream>` contient les déclarations des classes **[filebuf,] ifstream, ofstream** et **fstream** nécessaires à la manipulation de fichiers

Toutes les classes ont redéfini la signification d'un certain nombre d'opérateurs du langage; '<<' et '>>' ont ainsi été surchargés pour autoriser l'envoi (vers un flux) ou la réception (depuis un flux) de données de n'importe quel type fondamental (E/S formatées) :

```
cout << "Machain" << 'e';  
cout.flush();
```

ETAT D'ERREUR D'UN FLUX (1)

L'énumération **enum** `io_state` {goodbit, eofbit, failbit, badbit, hardfail}; définie dans la classe **ios** fournit les différents états possibles pour un flux

La fonction `ios :: rdstate ()` lit l'état d'un flux :

```
switch (cin.rdstate ())
{
    case ios :: goodbit : // tout va bien
        break;

    case ios :: eofbit : // plus de données
        break;

    default : // problème
}
```

Toute opération sur un flux dont l'état n'est pas `ios :: goodbit` retourne immédiatement sans effectuer d'action

ETAT D'ERREUR D'UN FLUX (2)

Autre méthode de test de l'état d'un flux :

```
if (cin)
    // gérer l'erreur sur 'cin'
```

Les différentes classes d'*iostream* définissent en effet une fonction surchargée de conversion de la forme

```
operator void* () { /* ... */ }
```

qui renvoie une valeur non nulle si une erreur d'E/S se produit sur le flot considéré

ETAT DE FORMAT & MANIPULATEURS (1)

L'état de format donne des informations relatives

- à la largeur du champ de sortie,
- au caractère de "remplissage" utilisé,
- à la justification à l'intérieur d'un champ,
- à la base d'affichage des nombres entiers

L'accès ou la modification de cet état se fait au travers soit de fonctions membres de la classe **ios**, soit de manipulateurs

Un manipulateur est un objet particulier qu'il suffit d'insérer dans un flux de sortie pour en modifier l'état

ETAT DE FORMAT & MANIPULATEURS (2)

- `oct`, `dec` et `hex` permettent l'affichage des entiers en base 8, 10 et 16

```
int n = 242;  
cout << "0x" << hex << n << endl; // 0xf2
```

La dernière base définie devient la base par défaut

- le manipulateur `setprecision()` prend pour argument la précision d'affichage pour les valeurs en virgule flottante (6 par défaut)

ETAT DE FORMAT & MANIPULATEURS (3)

Les manipulateurs `setw()` et `setfill()` prennent chacun un argument de type *int*

L'argument de `setw()` indique la largeur de la zone d'affichage allouée à chaque donnée; la valeur passée s'applique uniquement à la prochaine donnée affichée

Celui de `setfill()` précise le caractère utilisé pour remplir la zone d'affichage définie; le caractère spécifié reste "actif" tant qu'il n'est pas à nouveau modifié

```
cout << setw(5) << setfill('0') << n << endl; // 000f2
```

FICHIERS & FLOTS (1)

Un fichier est défini par une déclaration de flot de type **ofstream**, **ifstream** ou **fstream**

Ex : `ofstream fichier;`

Un fichier peut être ouvert

→ en invoquant la fonction membre `open()`

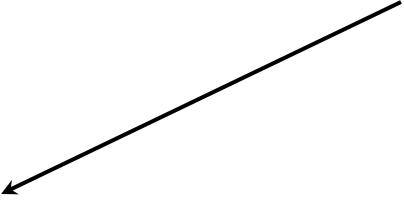
→ par une déclaration utilisant un constructeur de la classe

Ex : `fichier.open("monfichier");`
`ifstream monfich("fichier", ios :: in);`

FICHIERS & FLOTS (2)

Les arguments des deux fonctions sont le nom du fichier et ses modes d'ouverture et d'accès; les modes d'ouverture sont

```
class ios {  
public :  
    enum openmode  
    {  
        in,           // ouverture en lecture  
        out,          // ouverture en écriture  
        ate,          // positionnement initial  
                      // en fin de fichier  
        app,          // ajout en fin de fichier  
        trunc,        // écrase le fichier s'il existe  
        binary         // fichier "binaire"  
    };  
    // ...  
};
```



Ex : `fstream fic("monfich", ios :: in | ios :: out);`

FICHIERS & FLOTS (3)

La fermeture d'un fichier peut être réalisée soit explicitement (fonction membre `close()`), soit implicitement via le destructeur

`seekp()` (resp. `seekg()`) permet de se positionner dans un **ostream** (resp. un **istream**), et possède 2 versions

Un **iostream** a des positions en lecture et en écriture séparées

```
typedef long streampos, streamoff;

class ios
{
public :
    enum seek_dir {beg = 0, cur = 1, end = 2};
    // ...
};

istream& seekg(streampos sp);
istream& seekg(streamoff so, ios :: seek_dir dir);
```

FICHIERS & FLOTS (4)

Les fonctions `tellp()` et `tellg()` renvoient un 'streampos' qui correspond à la position courante du "curseur" dans le fichier

Les lectures et écritures non formatées dans des flux respectivement de type **istream** et **ostream** peuvent être réalisées soit par `get()` et `put()`, soit par `read()` et `write()` :

```
istream& read(char*, int);  
istream& read(signed char*, int);  
istream& read(unsigned char*, int);
```

Les fonctions `gcount()` et `pcount()` renvoient un entier indiquant le nombre d'octets effectivement lus ou écrits lors de la dernière invocation de `read()` ou de `write()`

FLOTS DE CHAINES DE CARACTERES EN MEMOIRE (1)

Le fichier `<sstream>` donne accès aux fonctions d'E/S en mémoire (à la manière de `sscanf()` et `sprintf()` en C)

On peut attacher à un flot de sortie une chaîne de caractères pour mise en forme avant affichage :

```
char* p = new char[taille_message];  
ostream ost;
```

```
mettre_en_forme (arguments, ost);  
strcpy(p, ost.str().c_str());  
afficher (p);
```

La fonction `mettre_en_forme()` peut écrire sur le flot *ost*, passer *ost* à ses sous-opérations, ..., via les opérations standard de sortie

FLOTS DE CHAINES DE CARACTERES EN MEMOIRE (2)

La classe *istream* permet de définir des flots de chaînes de caractères en entrée « lisant » une chaîne de caractères terminée par `\0` :

```
void un_mot_par_ligne (char v[], int t)
{
    istream ist;    // crée un istream pour 'v'
    ...            // à compléter, cf. poly
    char b2[MAX];   // plus grand que le plus grand mot

    // affiche 'v' de taille 't', un mot par ligne
    while (ist >> b2)
        cout << b2 << "\n";
}
```

Le caractère `\0` terminal est interprété comme la fin du fichier

SEANCE 8 :

PATRONS – EXCEPTIONS – TYPAGE

- Patrons de classes :
Définitions. Exemples
- Patrons de fonctions :
Définitions. Contraintes & Exemples
- Exceptions :
Principes. Règles d'utilisation (x 9). Le module <exception>
- C++ & typage

PATRONS DE CLASSES (1) : Définitions

Un patron de classe spécifie comment des classes individuelles peuvent être construites; la construction de celles-ci est ensuite réalisée en fonction des besoins

Syntaxe :

```
template<class T>
class nom_classe
{
    T *p;           // exemples ...
    T f ();         // de membres
    // ...
};
```

Cette déclaration signifie qu'un argument de type T sera utilisé dans une déclaration de classe 'nom_classe'

PATRONS DE CLASSES (2) : Exemple 1

```
template<class T>
class pile
{
    T* sommet;
    T* base;
    int tai;
public :
    pile (int t) {
        sommet = base = new T[tai = t];}
    ~pile () { delete[] base;}
    void empile (T v) { *sommet++ = v;}
    T depile () { return *--sommet;}
    int taille () const { return sommet - base;}
};
```

Un nom de classe suivi d'un type entouré de <> est le nom d'une classe, utilisable comme tout autre nom de classe :

```
pile<char> pch (200); // pile de caractères
```

PATRONS DE CLASSES (3) : Exemple 2

Les définitions de fonctions membres suivent les contraintes syntaxiques usuelles :

```
template<class T>
    void pile<T> :: empile (T v) {
        // ...
    }
```

Des fonctions membres patrons sont automatiquement générées pour chaque type d'argument du patron; pour `pile<complexe> p(10)` :

```
void pile<complexe> :: empile (complexe v) {
    *sommet++ = v;
}
```

On peut fournir plusieurs arguments dans la déclaration d'un patron :

```
template<class T, int t>    class vecteur {
    T v[t];                // ...
};
```


PATRONS DE CLASSES (4) : Terminologie

Dans une définition de patron de classe telle que

```
template<class a1, class a2, ..., class an> class cl
```

→ cl est le nom d'un patron de classe et donc un nom de classe

→ **class** a₁, **class** a₂, ..., **class** a_n constitue la liste des arguments du patron

Et une déclaration

```
cl<type1, type2, ..., typen>
```

correspond à une définition de classe patron générée à partir de son patron de classe

PATRONS DE FONCTIONS : Définitions

Un patron de fonction définit une famille de fonctions :

```
template<liste_arguments_patron>  
    type_retour_nom_fonction (liste_arguments_fonction)
```

Règle Chaque *argument_patron* spécifié dans la *liste_arguments_patron* doit être utilisé au moins une fois comme type d'un argument présent dans la *liste_arguments_fonction* du patron de fonction :

```
template<class T> T* f1 (T a, T b);           // ok  
template<class T> void f2 (nom_classe<T>& c); // ok  
template<class T> T& f3 ();                   // erreur
```

PATRONS DE FONCTIONS : Exemple (1)

```
// Un patron de vecteur ...  
  
template<class T> class Vect {  
    T* pv;  
    int t;  
public :  
    T& operator[] (const int i) {  
        return pv[i];  
    }  
    int taille () const { return t; }  
};
```

PATRONS DE FONCTIONS : Exemple (2)

```
template<class T> void tri (Vect<T>& v)
{
    int n = v.taille ();

    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (v[j] < v[j - 1]) {
                T temp = v[j];
                v[j] = v[j - 1];
                v[j - 1] = temp;
            }
}

void f (Vect<int>& vi, Vect<String>& vs, Vect<char*>& vpc)
{
    tri (vi);           // appelle tri (Vect<int>& v);
    tri (vs);           // appelle tri (Vect<String>& v);
    tri (vpc);          // appelle tri (Vect<char*>& v);
}
```

PATRONS DE FONCTIONS : Exemple (3)

PB : l'opérateur '<' n'existe pas toujours (cas des complexes), ou est inadapté au contexte (cas des char*)

SOL : on peut toujours spécifier explicitement une version appropriée de tri () pour un type particulier :

```
void tri (Vect<char*>& v)  // version à utiliser
{
    int n = v.taille ();

    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (strcmp (v[j], v[j - 1]) < 0)
                { /* échanger v[j] et v[j - 1] */ }
}
```

REGLES DE RESOLUTION DE LA SURCHARGE

(1) une correspondance exacte sur les fonctions est tout d'abord cherchée (mise en correspondance des arguments); si une fonction est trouvée, elle est appelée

(2) un patron de fonction à partir duquel une fonction patron (pouvant être appelée avec une correspondance exacte) peut être générée est recherché; si un tel patron existe, la fonction adéquate est générée puis appelée

(3) enfin, une résolution de surcharge ordinaire pour les fonctions est essayée

EXCEPTIONS : Principes (1)

C++ offre un mécanisme permettant de gérer les erreurs

→ provoquées par l'utilisateur

→ logiques ou système rencontrées à l'exécution

PRINCIPE : une fonction qui détecte une erreur a la possibilité de lancer une exception via une instruction

```
throw [qqchose];
```

Lancer une exception consiste à transférer le contrôle de l'exécution à une tâche spécifique, dont la construction C++ est

```
catch (/*...*/)
{
    // traiter l'exception
}
```

EXCEPTIONS : Principes (2)

Le transfert du contrôle n'est effectif que si

- (1) la construction **catch** est définie, et si
- (2) la fonction à l'origine du lancement de l'exception autorise son traitement (lancement depuis un bloc **try**)

```
class Zero {};  
void VerZero (int i) {if (i == 0) throw Zero ();}  
  
void f () {  
    cout << "Entrer un nombre\n";  
    double a; cin >> a;  
    try {  
        VerZero (a);  
        cout << "Inverse = " << 1.0 / a << '\n';  
    }  
    catch (Zero) {  
        cout << "0 ne peut être inversé\n";  
        exit (99);  
    }  
}
```


MISE EN ŒUVRE DES EXCEPTIONS (1/9)

La construction ***catch*** (*/* ... */*) { */* ... */* } est appelée un gestionnaire d'exception

Elle ne peut être utilisée qu'après un bloc préfixé avec le mot-clé **try** ou immédiatement après un autre gestionnaire d'exception (gestion d'exceptions multiples)

Les parenthèses contiennent une déclaration qui précise le type des objets avec lesquels le gestionnaire peut être appelé

Le gestionnaire d'exception ***catch*** (...) { */* ... */* } peut intercepter n'importe quelle exception

MISE EN ŒUVRE DES EXCEPTIONS (2/9)

Un bloc *try* { /* ... */ } peut être vu comme une déclaration de portée à l'intérieur de laquelle le lancement d'exceptions est autorisé

Lorsqu'un gestionnaire d'exception ne peut être activé (exécution interdite, absence d'un tel gestionnaire, ...), le programme s'arrête automatiquement après avoir libéré toutes les ressources utilisées

MISE EN ŒUVRE DES EXCEPTIONS (3/9)

Important : Un gestionnaire d'exception ne peut comporter d'instruction *return*;

Lorsqu'un tel gestionnaire a terminé son travail, l'exécution du programme reprend juste après le bloc **try** auquel il est associé (en général, juste après l'appel de la fonction auquel il appartient)

MISE EN ŒUVRE DES EXCEPTIONS (4/9)

Lorsqu'une exception a été traitée par un gestionnaire, d'autres gestionnaires également en mesure de l'intercepter ne seront plus pris en compte

C'est donc le gestionnaire actif rencontré le plus récemment qui est invoqué, et lui seul :

```
int ff (int b) {  
    try  
    {  
        f (b) ;  
    }  
    catch (Zero)  
    {  
        // on n'arrivera jamais jusqu'ici  
    }  
}
```

MISE EN ŒUVRE DES EXCEPTIONS (5/9)

Une fonction n'a pas besoin d'intercepter toutes les exceptions possibles :

```
class E1 { /* ... */ };  
class E2 { /* ... */ };  
  
void g () {  
    try { h (); }  
    catch (E1) { /* ... */ }  
}  
  
void h () {  
    try  
    {  
        // ... throw E1 (); ... throw E2 (); ...  
    }  
    catch (E2) { /* ... */ }  
}
```

→ h () intercepte les erreurs de type E2 et transmet celles de type E1 à g ()

MISE EN ŒUVRE DES EXCEPTIONS (6/9)

Des gestionnaires d'exception peuvent s'imbriquer, et un gestionnaire d'exception peut lui-même lancer une exception :

```
class E3 { /* ... */ };

void k () {
    try { /* ... */ }
    catch (E1)
    {
        try
        {
            // qq chose de compliqué
        }
        catch (E2)
        {
            // le code du gestionnaire
            // compliqué a échoué
            throw E3 ();
            // attention au point de retour!
        }
    }
}
```

MISE EN ŒUVRE DES EXCEPTIONS (7a/9)

Une exception est interceptée en précisant son type, mais ce qui est lancé n'est pas un type mais un objet → on peut donc transmettre de l'information au gestionnaire d'exception en insérant des données dans l'objet lancé :

```
class ErrMath {
public :
    char nomerr[30];
}

class DivparZero : public ErrMath {
public :
    DivparZero () {
        strcpy (nomerr, "Div par zero\n");
    };
};

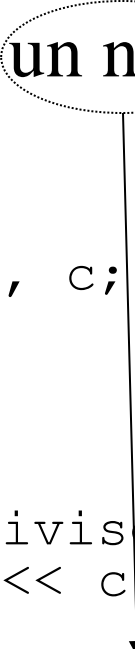
double diviser (double a, double b) {
    if (b == 0) throw DivparZero ();
    return a/b;
}
```

MISE EN ŒUVRE DES EXCEPTIONS (7b/9)

Pour accéder à l'information ainsi mémorisée, le gestionnaire doit donner un nom à "l'objet exception" :

```
void foo ()
{
    double a, b, c;

    cin >> a;
    cin >> b;
    try {
        c = diviser (a, b);
        cout << c << '\n';
    }
    catch (ErrMath e) {
        cout << "Type d'erreur : ";
        cout << e.nomerr << endl;
    }
}
```



MISE EN ŒUVRE DES EXCEPTIONS (8a/9)

Un gestionnaire d'exception peut décider de ne pas traiter une exception → il a alors la possibilité de relancer l'exception au moyen de l'instruction **throw**; (invocation sans précision d'objet)

```
void l ()
{
    try { /* qq chose */ }
    catch (E1)
    {
        if (traitement_possible)
            // gérer l'exception
        else
            throw;          // relancer
    }
}
```

MISE EN ŒUVRE DES EXCEPTIONS (8b/9)

L'appelant de `l ()` peut encore intercepter un objet de type `E1` que `l ()` a intercepté sans pouvoir le traiter :

```
void m ()
{
    try
    {
        l ();
    }
    catch (E1) { /* ... */ }
}
```

Intérêt : utilisation conjointe avec des classes dérivées d'exceptions

MISE EN ŒUVRE DES EXCEPTIONS (9/9)

```
void n (char& c, int i) throw (x1, x2, x3)
{
    // code
}
```

Définitions équivalentes

```
void n (char& c, int i)
{
    try { /* code */ }
    catch (x1) { throw; }           // relance x1
    catch (x2) { throw; }           // relance x2
    catch (x3) { throw; }           // relance x3
    catch (...) { unexpected(); }    // arrêt du pgm
}
```

Fct système

`int p () { /* ... */ }` peut lancer n'importe quelle exception

Redéfinir `unexpected()` si arrêt du pgm non bienvenu ...

LA BIBLIOTHEQUE *exception*

Les exceptions « standard » sont gérées *via* une hiérarchie de classes :

- `exception`
 - `ios_base::failure`
 - `bad_cast`
 - `bad_typeid`
 - `bad_alloc`
 - `bad_exception`
 - `logic_error`
 - `length_error`
 - `domain_error`
 - `out_of_range`
 - `invalid_argument`
 - `runtime_error`
 - `range_error`
 - `overflow_error`
 - `underflow_error`

<exception> : Exemple (*bad_alloc*)

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

int main()
{
    try
    {
        int* monTableau;

        for (int i = 0; ; i++)
            monTableau = new int[1000];      // !!
        // ...
    }
    catch (exception& e)
    {
        cout << "Exception standard : ";
        cout << e.what() << endl;
    }
    return 0;
}
```

RTTI : Run Time Type Information (Verification)

- fonctions virtuelles

- **dynamic cast**

```
class Base { ... } oB, *pbas = &oB;  
class Derivee : public Base { ... } *pder;  
pder = dynamic_cast<Derivee*>(pbas);           // invalide
```

- **opérateur typeid & classe typeid**

```
Derivee oD;  
Base* p = &oD;
```

```
const typeid_info& infos = typeid(*p);  
cout << "type de *p " << infos.name() << "\\n" ; // Derivee
```

SEANCE 9 : la STL de C++

- Généralités
- Fonctionnalités offertes
- Principaux apports
 - Conteneurs
 - Itérateurs
 - Algorithmes

Contenu général de la STL

La STL (**S**tandard **T**emplate **L**ibraries) comporte un ensemble de fonctions, constantes, classes, objets et *templates* qui étendent le langage pour réaliser de nombreuses tâches répondant aux besoins usuels des programmeurs; leurs déclarations sont disséminées dans différents *headers* à inclure

algorithm	complex	exception	list	stack
bitset	csetjmp	fstream	locale	stdexcept
cassert	csignal	functional	map	strstream
cctype	cstdarg	iomanip	memory	streambuf
cerrno	cstddef	ios	new	string
cfloat	cstdid	iosfwd	numeric	typeinfo
ciso646	cstdlib	iostream	ostream	utility
climits	cstring	istream	queue	valarray
clocale	ctime	iterator	set	vector
cmath	deque	limits	sstream	

Les principales fonctionnalités offertes sont au nombre de 3

STL (1/3) : la librairie du langage C

<u>cassert (assert.h)</u>	C Diagnostics Library (header)
<u>cctype (ctype.h)</u>	Character handling functions (header)
<u>cerrno (errno.h)</u>	C Errors (header)
<u>cfloat (float.h)</u>	Characteristics of floating-point types (header)
<u>ciso646 (iso646.h)</u>	ISO 646 Alternative operator spellings (header)
<u>climits (limits.h)</u>	Sizes of integral types (header)
<u>locale (locale.h)</u>	C localization library (header)
<u>cmath (math.h)</u>	C mathematical library (header)
<u>csetjmp (setjmp.h)</u>	Non local jumps (header)
<u>csignal (signal.h)</u>	C library to handle signals (header)
<u>cstdarg (stdarg.h)</u>	Variable arguments handling (header)
<u>cstddef (stddef.h)</u>	C Standard definitions (header)
<u>cstdio (stdio.h)</u>	C library to perform Input/Output operations (header)
<u>cstdlib (stdlib.h)</u>	C Standard General Utilities Library (header)
<u>cstring (string.h)</u>	C Strings (header)
<u>ctime</u>	C Time Library (header)

STL (2/3) : gestion de concepts spécifiques

Language support library:

<u>limits</u>	Numeric limits (header)
<u>new</u>	Dynamic memory (header)
<u>typeinfo</u>	Type information (header)
<u>exception</u>	Standard exception class (class)

Diagnostics library:

<u>stdexcept</u>	Exception classes (header)
----------------------------------	----------------------------

General utilities library:

<u>utility</u>	Utility components (header)
<u>functional</u>	Function objects (header)
<u>memory</u>	Memory elements (header)

Localizations library:

<u>locale</u>	Localization library (header)
-------------------------------	-------------------------------

Strings library:

<u>string</u>	C++ Strings library (library)
-------------------------------	-------------------------------

Numeric library:

<u>complex</u>	Complex numbers library (header)
<u>valarray</u>	Library for arrays of numeric values (header)
<u>numeric</u>	Generalized numeric operations (header)

Input/Output Stream library:

Cf. séance 7

STL (3/3) : *Containers* et outils associés

Containers library:

<u>bitset</u>	Bitset (class template)
<u>deque</u>	Double ended queue (class template)
<u>list</u>	List (class template)
<u>map</u>	Map (class template)
<u>multimap</u>	Multiple-key map (class template)
<u>multiset</u>	Multiple-key set (class template)
<u>priority_queue</u>	Priority queue (class template)
<u>queue</u>	FIFO queue (class template)
<u>set</u>	Set (class template)
<u>stack</u>	LIFO stack (class template)
<u>vector</u>	Vector (class template)

Iterators library:

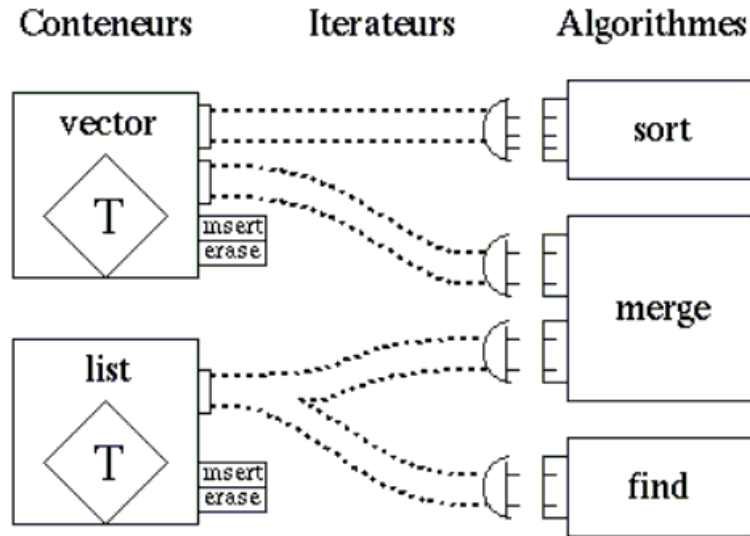
<u>iterator</u>	Iterator definitions (header)
---------------------------------	-------------------------------

Algorithms library:

<u>algorithm</u>	Standard Template Library: Algorithms (library)
----------------------------------	---

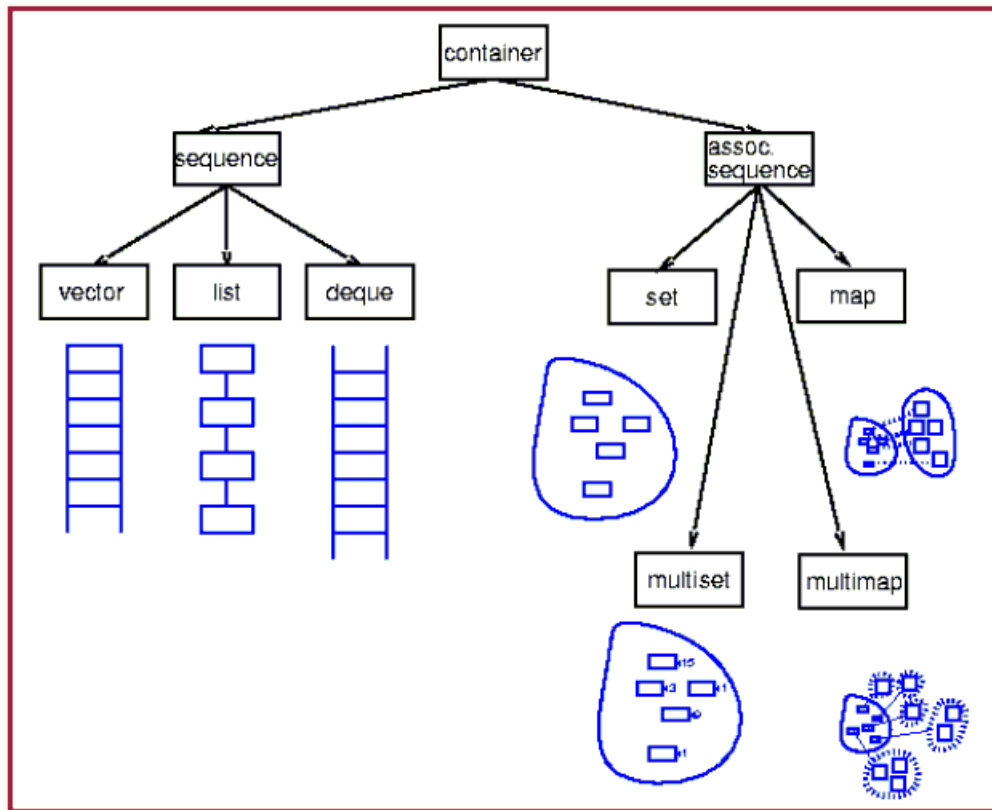
C++11 y a ajouté quelques conteneurs et librairies

Indépendance Algorithmes / Conteneurs



- Les *conteneurs* (tableaux dynamiques, listes, piles, ensembles, ...) sont dotés de fonctions pour les manipuler et fournissent des *itérateurs*
- Les *itérateurs* sont des "pointeurs" sur les éléments des conteneurs, permettant de faire abstraction des structures de données concrètes utilisées pour la réalisation
- Les *algorithmes* (e.g., copier, rechercher, remplacer, compter, ...) prennent ou pas en paramètres des *itérateurs*, mais sont entièrement définis avec eux sans autre référence à la structure concrète

Les conteneurs standards



- Conteneurs *séquentiels*
l'ordre d'insertion dans le conteneur donne l'ordre dans lequel les éléments sont accessibles
- Conteneurs *associatifs*
on fait correspondre une valeur à une clef, l'ordre d'insertion n'est pas respecté lors de l'accès

+ Adaptateurs de conteneurs séquentiels (pile, file, file avec priorité)

Règles générales concernant les conteneurs

- A chaque conteneur le `#include` qui lui correspond!
- Le « contenu » d'un conteneur s'appelle une collection
- Pour savoir ce qui est disponible pour un conteneur, e.g., *set* voir <http://www.cplusplus.com/reference/stl/set>
- Pour simplifier la gestion de la mémoire, privilégier autant que possible :
 - les chaînes de caractères *string* aux `char*`
 - les *références* (&) aux pointeurs (*)
 - les *conteneurs* aux structures « faites maison »

Services de base d'un conteneur

Les conteneurs de la STL définissent tous des itérateurs, des types spécifiques, des opérateurs et des fonctions standard

Ex.

```
set(); // un ensemble vide  
set(const set<T>& x); // un ensemble obtenu par copie  
  
insert(const T& x); // ajoute un nouvel élément  
erase(const T& x); // supprime l'élément x s'il existe  
size(); // le nombre d'éléments dans l'ensemble  
empty(); // l'ensemble est-il vide?
```

La classe *Vector*

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

int main(void)
{
    typedef vector<int> vi;
    vi v(10);           // crée un vecteur de 10 éléments
    v.at(2) = 2;        // modifie un élément
    v.at(5) = 7;
    v.resize(11);       // redimensionne le vecteur
    v.at(10) = 5;
    v.push_back(13);    // ajoute un élément à la fin

    // affiche le vecteur en utilisant l'opérateur []
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << endl;
    // ...
}
```


La classe *List*

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

typedef list<int> li;

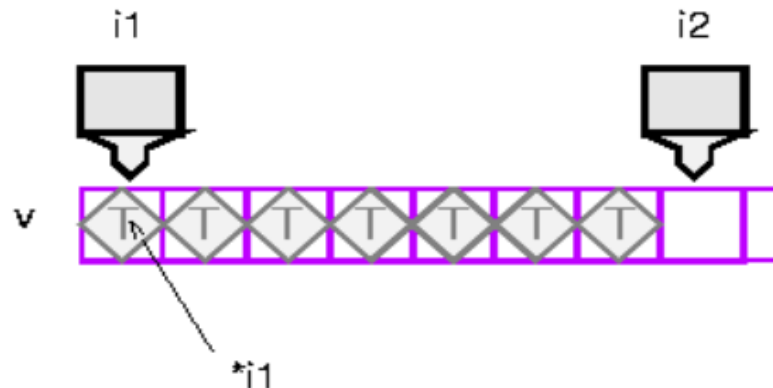
int main(void) {
    li l1;
    l1.push_back(2);    // insertion a la fin
    l1.push_back(5);
    l1.push_front(7);  // insertion au debut
    l1.sort();          // trie la liste
    l1.push_back(5);
    print(l1);
    l1.remove(5);       // supprime tous les 5
    print(l1);
}

void print(li &l) {
    li::iterator i = l.begin();
    while (i != l.end()) {
        cout << *i << " ";
        i++;
    }
    cout << endl;
    return ;
}
```

Itérateurs : principe (1/2)

Soit une collection v dont les éléments sont de type T :

- $i1$ pointe le début de la collection (e.g., obtenu avec `v.begin()`)
- $i2$ pointe la fin de la collection (e.g., obtenu avec `v.end()`)
- $*i1$ est l'élément de type T pointé par $i1$



- **itérateur** = « **pointeur** » (sur les éléments d'un conteneur)
- un pointeur *normal* peut être considéré comme un itérateur (sur un tableau C)!

Itérateurs : principe (2/2)

```
// Obtient un itérateur sur le premier élément
Conteneur::itérateur i = instance.begin();

// Boucle sur toutes les valeurs de l'itérateur
// jusqu'à la dernière
while (i != instance.end())
{
    // f() travaille sur l'élément référencé par i
    f(*i);
    // Passe à l'élément suivant
    ++i;
}
```

où Conteneur est la classe du conteneur et instance en est une instance (un objet)

La classe *Map* (1/2)

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

void display (map <string,char> grades)
{
    cout << "\tTotal size: " << grades.size() << endl;
    // crée un itérateur
    map <string, char>::iterator it;

    for (it = grades.begin(); it != grades.end(); ++it)
        // affiche le string (first) et le char (second)
        cout << setw(10) << it->first << setw(5) << it->second << endl;

    cout << endl;
}
```

La classe *Map* (2/2)

```
int main ()
{
    map <string, char> grades;

    grades["Danny"] = 'A';    grades["Angelina"] = 'A';
    grades["Joe"] = 'F';      grades["Brad"] = 'C';
    display(grades);

    // erase() supprime le membre de grades passé en paramètre
    // find() trouve la donnée associée à la clé, et
    // renvoie la position de l'itérateur correspondant
    grades.erase (grades.find("Joe"));
    display(grades);

    if (grades.count("Joe")) // Joe a été supprimé!
        cout << "Joe est membre de grades" << endl;
    else cout << "grades ne contient pas Joe" << endl;

    grades.clear (); // efface toutes les données
    display(grades);
    // ...
}
```

Itérateurs : écritures équivalentes (1/2)

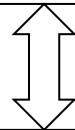
```
vector<int> v = ...;    // initialiser v
int sum = 0;
for (int i = 0; i < v.size(); i++)
    sum += v[i];
```

```
...
int sum = 0;
for (vector::iterator i = v.begin(); i < v.end(); i++)
    sum += *i;
```

```
...
int sum = 0;
for (int x : v) sum += x;    // C++11
```

Itérateurs : écritures équivalentes (2/2)

```
int sum = 0;  
for (vector::iterator i = v.begin(); i < v.end(); i++)  
    sum += *i;
```



```
int sum = for_each (v.begin(), v.end(), Adder());
```

Un *functor* : une classe qui surcharge l'opérateur ()

```
Class Adder {  
    private : int _sum;  
    public : Adder() : _sum(0) { }  
             void operator() {int& i) {_sum += i;}  
             operator int() {return _sum;}  
};
```

La librairie *Algorithm* (1/2)

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

int main(void)
{
    vector<int> intVec = {56, 32, -43, 23, 12, 93, 132, -154};
    sort(intVec.begin(), intVec.end());
    // ...
    vector<string> sVec = {"John", "Bob", "Joe", "Zack", "Randy"};
    sort(sVec.begin(), sVec.end());
    // ...
}
```


La librairie *Algorithm* (2/2)

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

bool monTri(int i, int j) { return i > j; }

int main(void)
{
    vector<int> intVec = {56, 32, -43, 23, 12, 93, 132, -154};

    // ne pas mettre les () lors de l'appel a monTri
    // il faut passer un pointeur de fonction / un objet fonction
    sort(intVec.begin(), intVec.end(), monTri);

    // ...
}
```

SEANCE 10 : C++ & UML

- Généralités :
Vue d'une application. Les diagrammes de modélisation
- Diagrammes structurels :
Classes. Associations
- Diagrammes de comportement :
Cas d'utilisation. Diagrammes de séquence / *scenarii*

UML (Généralités 1/5)

UML (**U**nified **M**odeling **L**anguage) est un outil de modélisation d'applications construites à l'aide d'objets

Ce n'est **pas**

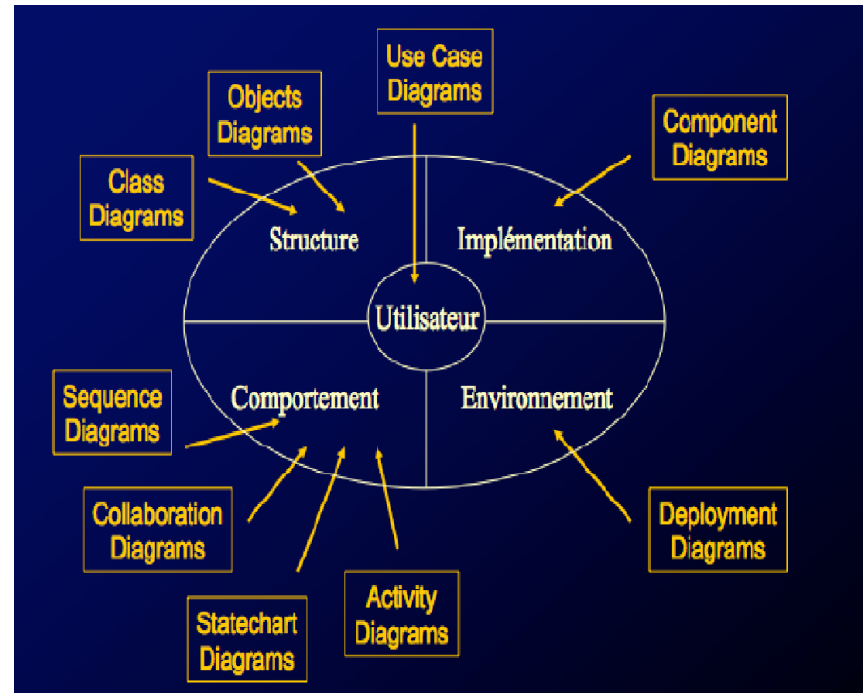
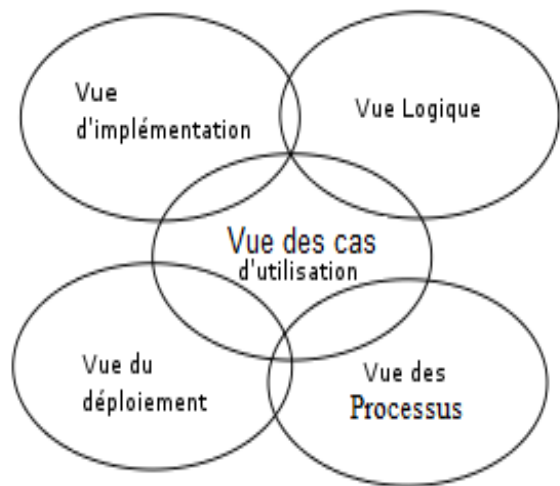
- un processus de développement
- un langage de programmation

UML en est le complément : un langage "visuel"

- adapté à toutes les phases de développement d'un logiciel, et à toutes les méthod(ologi)es associées
- compatible avec toutes les techniques de réalisation

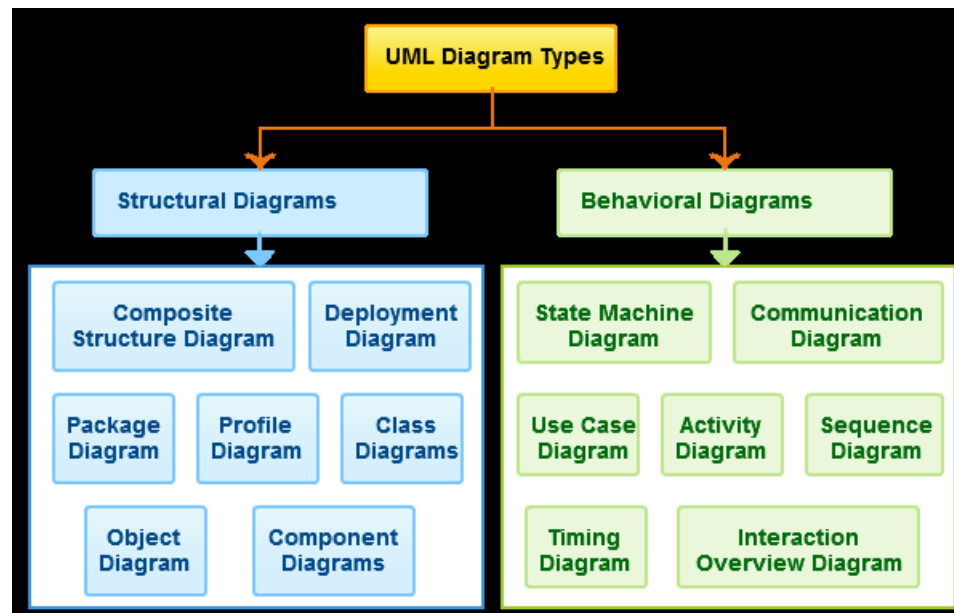
UML (Généralités 2/5)

UML propose 4 (+ 1) vues pour modéliser une application



UML (Généralités 3/5)

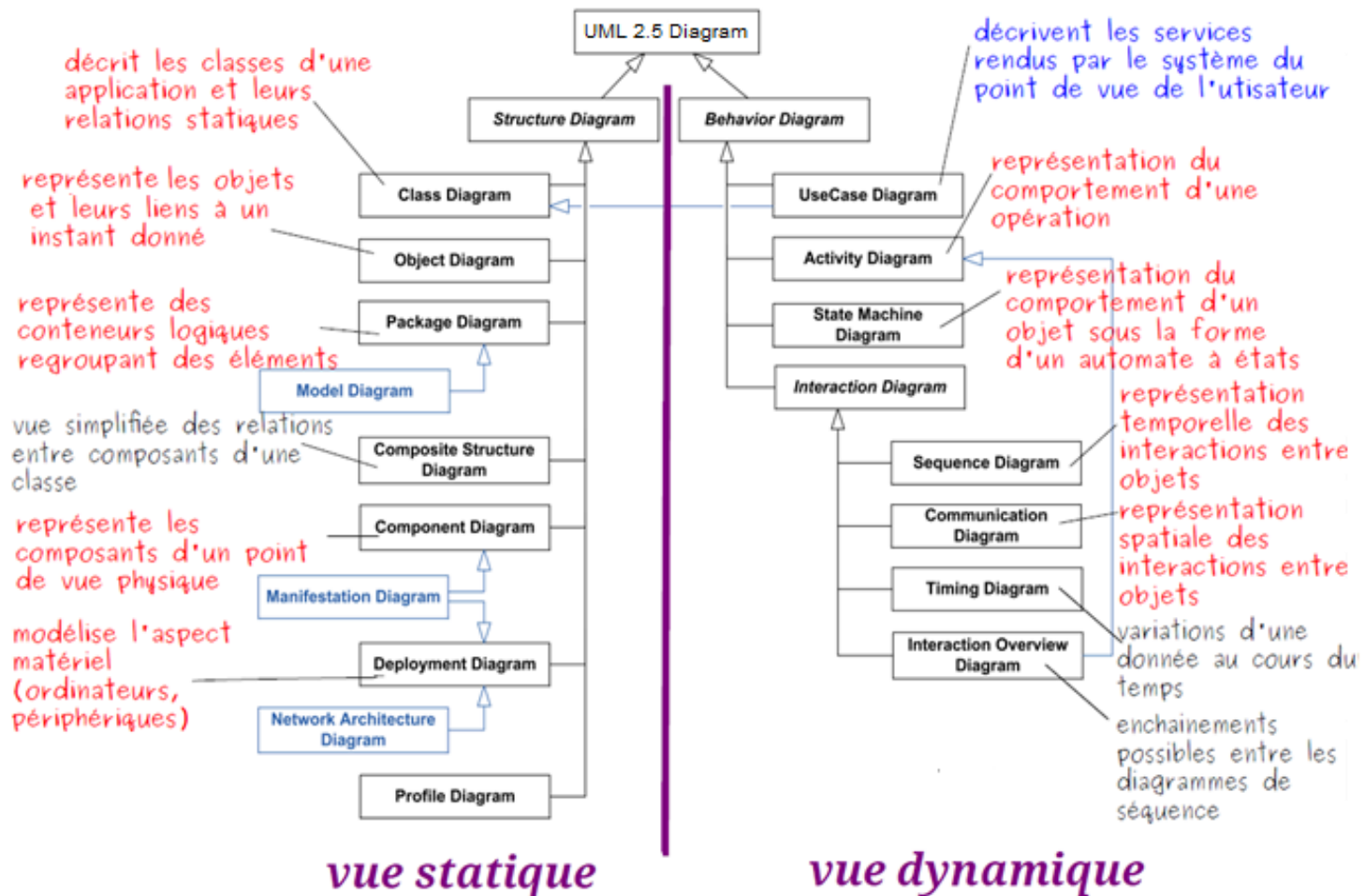
UML (V2.5, 2012) définit 14 types de diagrammes, que l'on peut classer en diagrammes structurels (ou statiques) et diagrammes comportementaux (dynamiques) :



Certains diagrammes "peuvent" être redondants ou peu utiles

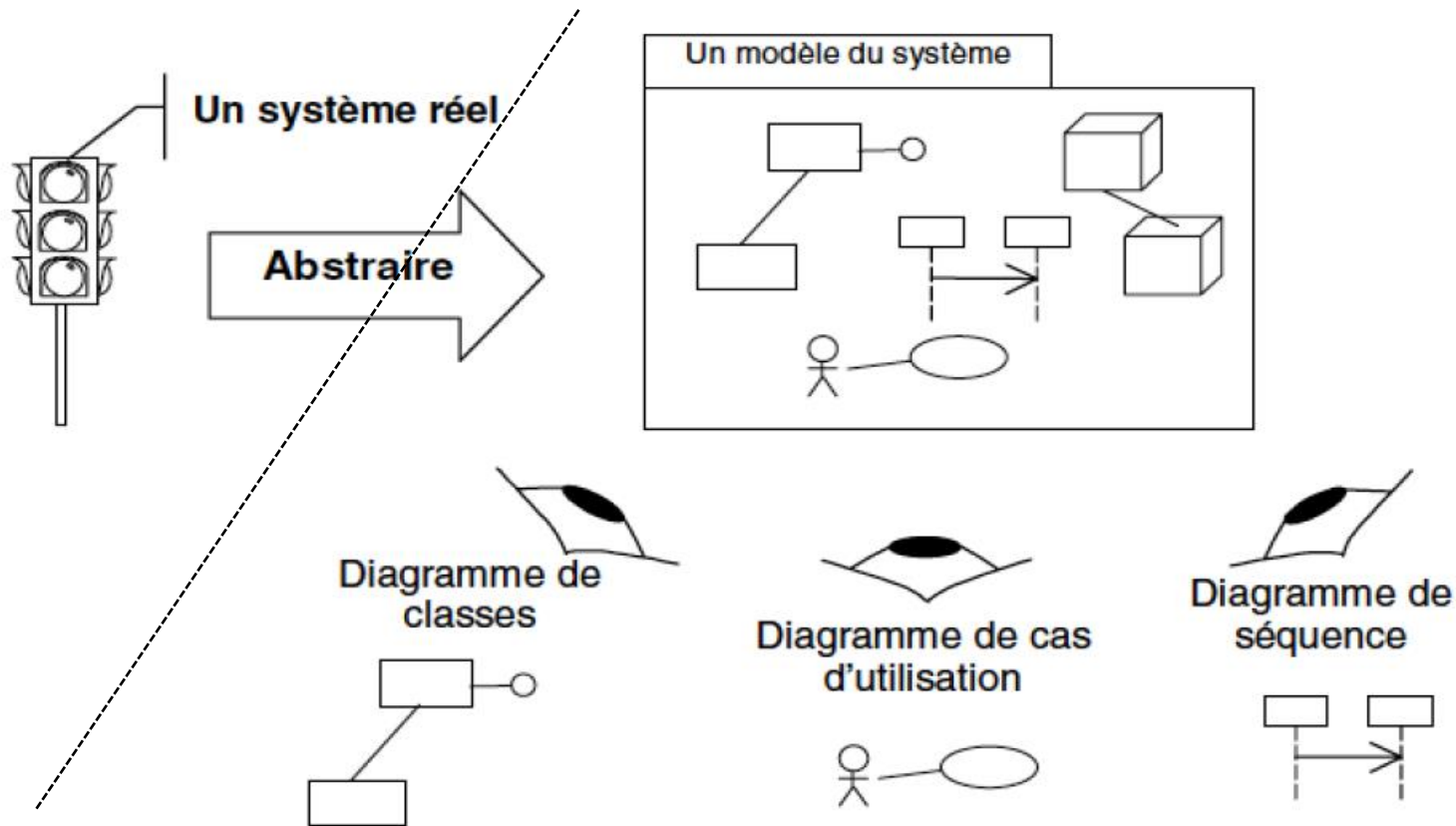
UML (Généralités 4/5)

Les diagrammes plus en détail :



UML (Généralités 5/5)

Quelques éléments de syntaxe (ou ... le point de vue du système modélisé)



+ Diagrammes d'activité / d'états-transitions

UML : Modélisation de la structure

- Représentation des classes
- Diagrammes d'objets
- Diagrammes de classes et associations :
Représentation des associations.
Navigabilité. Associations particulières
- Diagrammes de paquetages

Diagramme de classe (représentation)

Rectangle composé de compartiments

Compartiment 1 : Nom de la classe (commence par une majuscule, en gras)

Compartiment 2 : Attributs (ou Données)

Compartiment 3 : Opérations (ou Méthodes)

+ Possibilité d'ajouter des compartiments (exceptions, ...)

Possibilité de travailler à différents niveaux de détail

On peut omettre des attributs et/ou des opérations

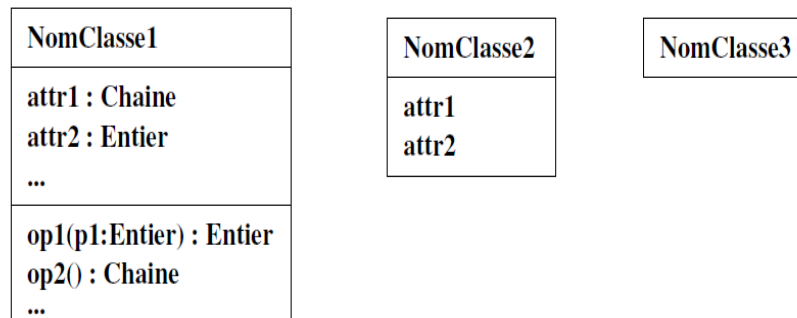


Diagramme de classe (données)

Format de description d'un attribut :

[Vis] Nom [Mult] [":" TypeAtt] ["=" Val] [Prop]

- Vis : + (public), - (privé), # (protégé), ~ (package)
- Mult : "[" nbElt "]" ou "[" Min .. Max "]"
- TypeAtt : type primitif (Entier, Chaîne, ...)
- Val : valeur initiale à la création de l'objet
- Prop : {gelé}, {variable}, {ajoutUniquement}, ...
- Attributs statiques soulignés
- Attributs dérivés précédés de "/"

Ex

onOff : Bouton

- x : Réel

coord[3] : Réel

+ pi : Réel = 3.14 {frozen}

inscrits[2..8] : Personne

/age : Entier

Diagramme de classe (méthodes)

Format de description d'une opération :

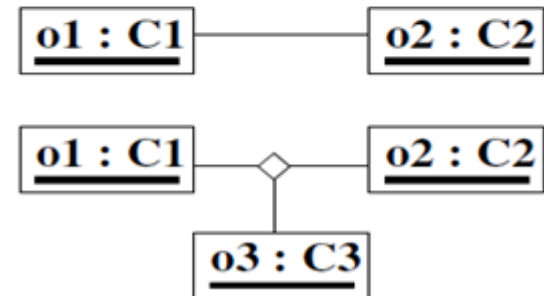
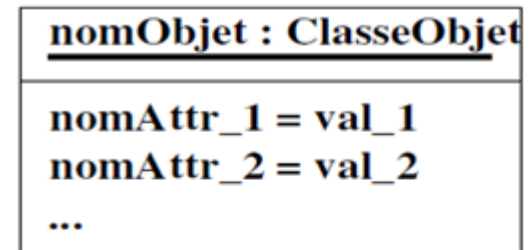
[Vis] Nom ["(" Arg ")"] [":" Type]

- Vis : + (public), - (privé), # (protégé)
- Arg : liste des arguments selon le format
[Dir] NomArgument : TypArgument
où Dir = in (par défaut), out, ou inout
- Opérations abstraites/virtuelles non implémentées en italique
- Opérations de classe statiques soulignées
- Possibilité d'annoter avec des contraintes stéréotypées de type *préconditions* et *postconditions*
- Stéréotypes d'opérations *create* et *destroy*

Diagrammes d'objets

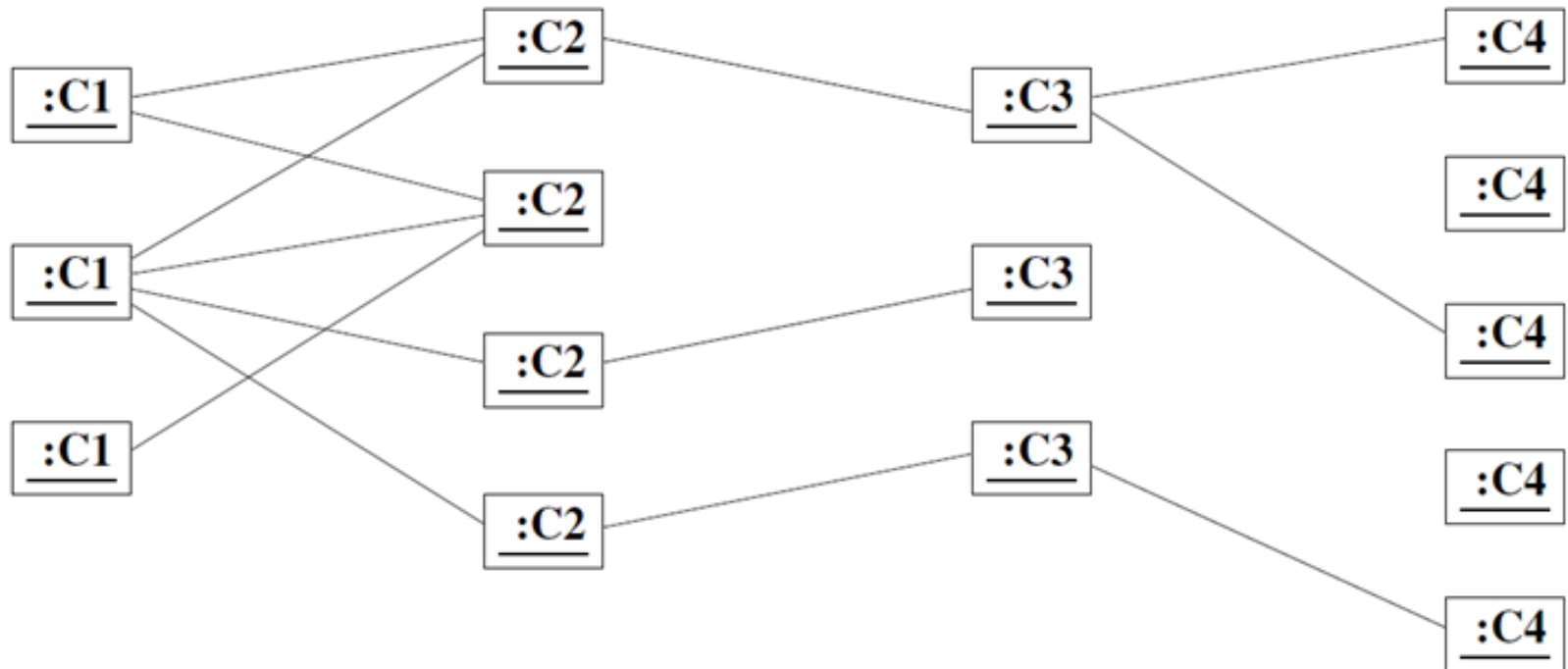
Représentent les objets et leurs liens à un instant donné sous forme d'un graphe non orienté :

- Nœuds du graphe = objets
 - Possibilité de ne pas représenter le nom, la classe et/ou les attributs
- Arêtes du graphe = liens entre objets
 - Lien binaire : entre 2 objets
 - Lien n-aire : entre n objets
 - Possibilité de nommer les liens / rôles



Associations entre classes (1/3)

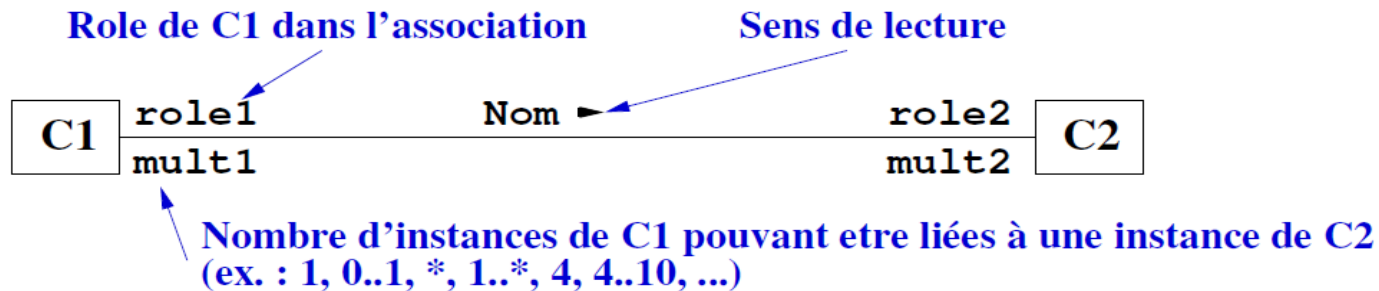
Liens entre objets :



Associations entre classes d'objets :



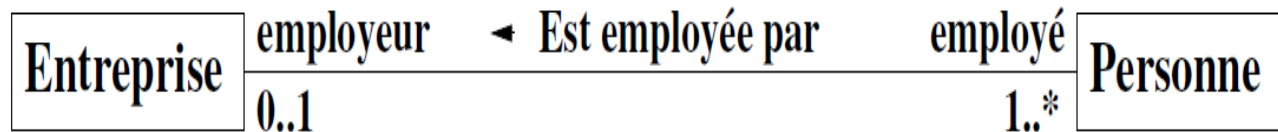
Associations entre classes (2/3)



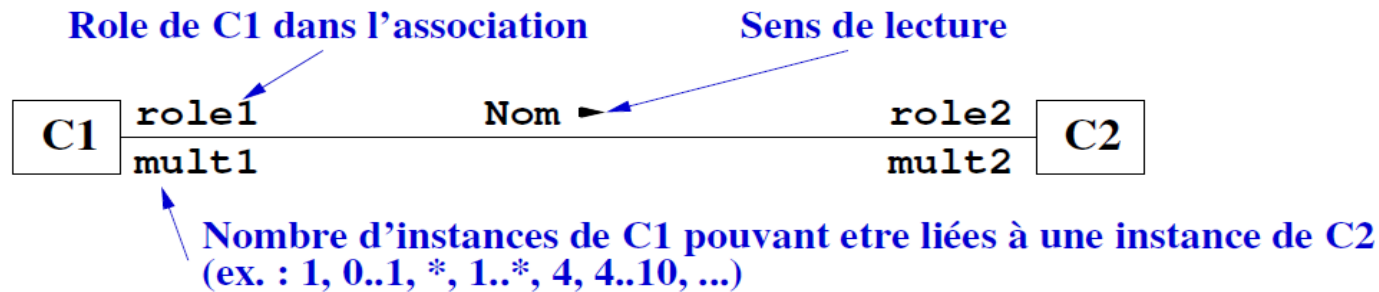
Interprétation en français :

- Un C1 *Nom* un C2 (ou un C2 *Nom* un C1 si sens de lecture inverse)
- Un C1 est *rôle1* d'un C2 et *mult1* C1 peuvent jouer ce rôle pour un C2
- Un C2 est *rôle2* d'un C1 et *mult2* C2 peuvent jouer ce rôle pour un C1

Exemple



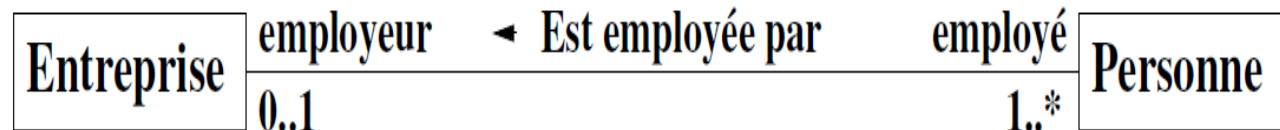
Associations entre classes (3/3)



Interprétation en langage OO :

- La classe C1 a un attribut de nom *rôle2*
Son type est C2 si *mult2* $\in \{1, 0..1\}$, ou collection de C2 sinon
- La classe C2 a un attribut de nom *rôle1*
Son type est C1 si *mult1* $\in \{1, 0..1\}$, ou collection de C1 sinon

Exemple



Navigabilité d'une association

Capacité d'une instance C_i à accéder aux instances de C_j ; une association peut être

- navigable dans les 2 sens (défaut)
- orientée

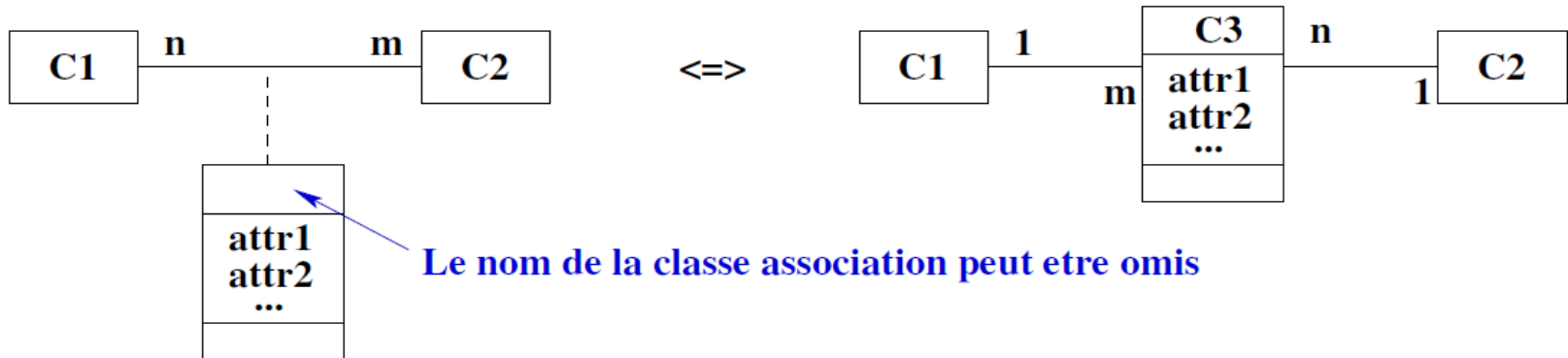


Propriétés sur les extrémités des associations

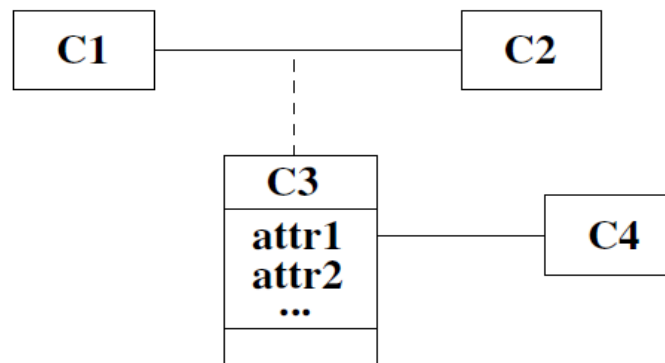
- $\{variable\}$: instance modifiable (défaut)
- $\{frozen\}$: instance non modifiable
- $\{addOnly\}$: instances ajoutables mais non retirables ($mult_i > 1$)

Classes-associations

Association attribuée :

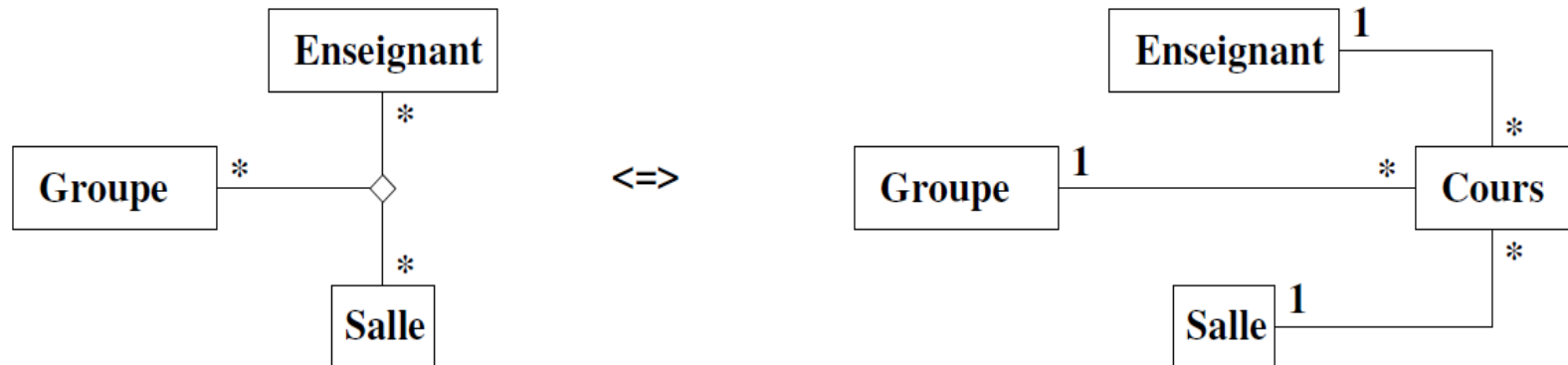


Une classe association peut participer à d'autres associations :

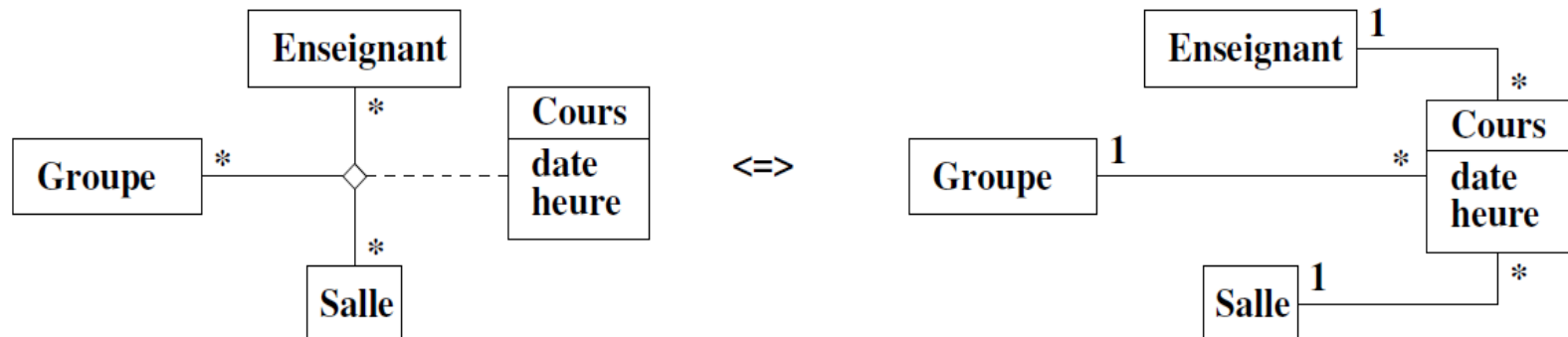


Associations n-aires

Associations entre n classes (avec $n > 2$)



Classes-Associations n-aire



Associations particulières (1/3) :

Composition et Agrégation

Composition :



- Relation transitive et antisymétrique
 - La création (copie, destruction) du composite (container) implique la création (copie, destruction) de ses composants
 - Un composant appartient à au plus un composite
-

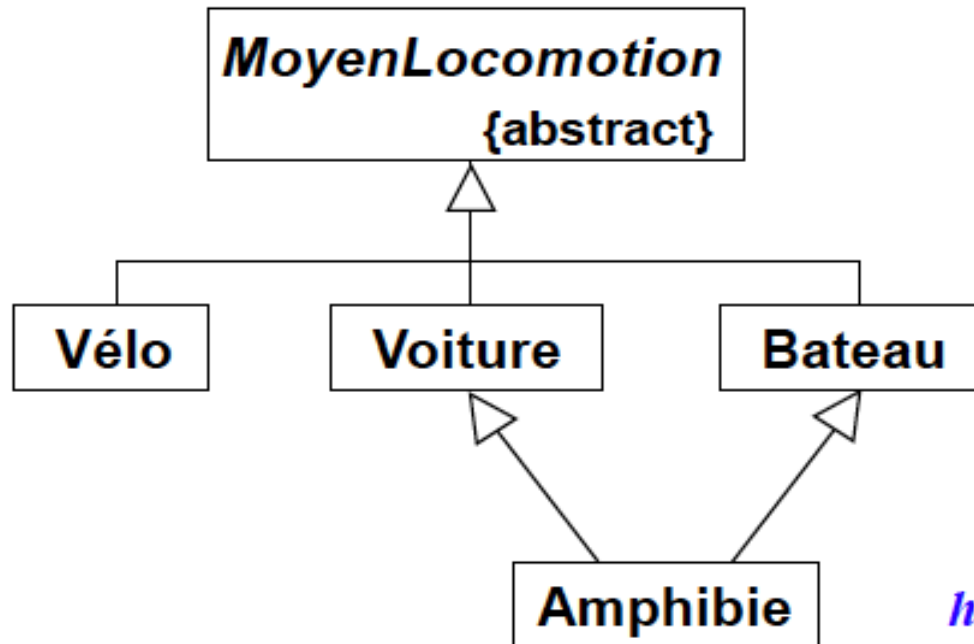
Agrégation :



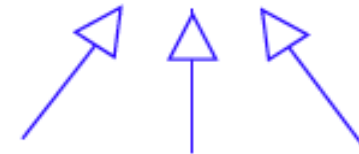
- Simple regroupement de parties dans un tout

Associations particulières (2/3) : Généralisation / Spécialisation (Héritage)

- relation transitive, non réflexive, et non symétrique
- la sous-classe “est-une-sort-de” la super classe; toute instance de la sous-classe est instance de la super classe
- implémenté *via* le mécanisme d’héritage dans les langages OO



notation équivalente



héritage multiple

Associations particulières (3/3) : Problématique

- Pb1 (héritage *vs* composition) : modéliser le fait qu'il existe des voitures rouges, des voitures vertes et des voitures bleues

Héritage : - créer une classe abstraite Voiture
 - créer 3 classes VR, VV et VB héritant de Voiture

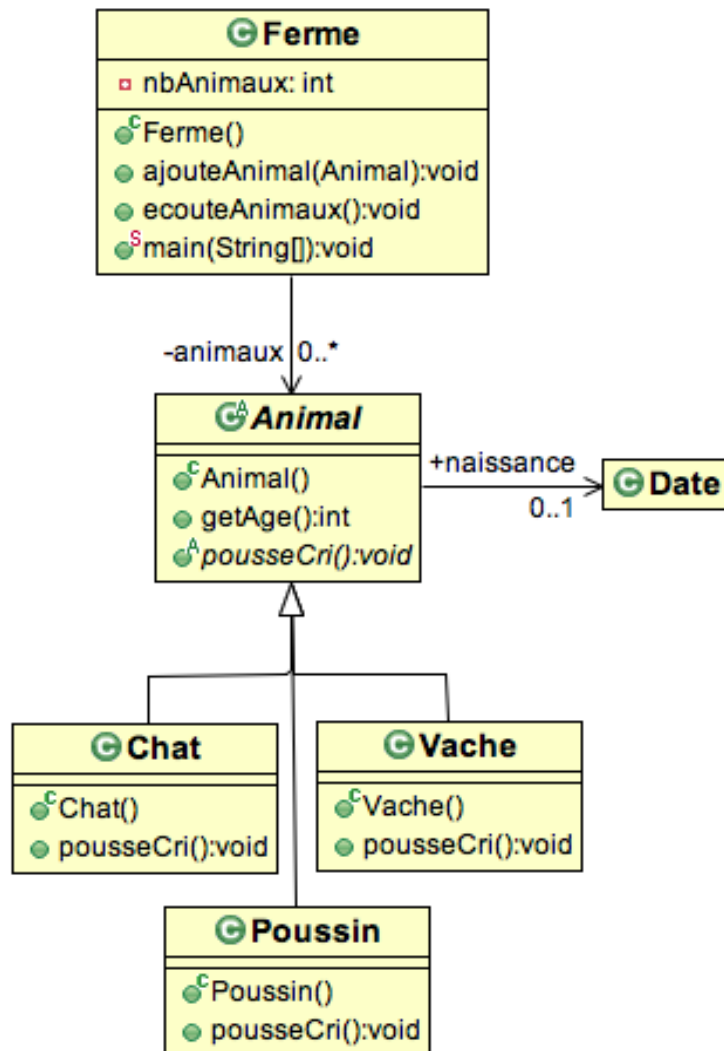
Composition : - créer 2 classes Voiture et Couleur
 - créer une association entre ces classes

- Pb2 (héritage *vs* délégation) : appeler dans une classe Y une opération $op()$ d'une classe X

Héritage : - faire hériter Y de X
 - $op()$ peut être appelée depuis toute instance de Y

Délégation : - ajouter dans Y un attribut x de type X (association)
 - $x.op()$ peut être appelé depuis tout objet de Y

Classes abstraites et polymorphisme (en C++)



```

class Animal{
public:
    Date naissance;
    int getAge(){...}
    virtual void pousseCri() = 0;
};
class Poussin: public Animal{
public:
    void pousseCri(){cout << "Piou" << endl;}
};
class Ferme{
private:
    Animal* animaux[];
    int nbAnimaux;
public:
    Ferme(){...}
    void ajouteAnimal(Animal* a){
        animaux[nbAnimaux++] = a;
    }
    void ecouteAnimaux(){
        for (int i=0; i<nbAnimaux; i++)
            animaux[i]->pousseCri();
    }
};
int main(){
    Ferme f;
    f.ajouteAnimal(new Chat());
    f.ajouteAnimal(new Poussin());
    f.ajouteAnimal(new Vache());
    f.ecouteAnimaux();
}
  
```

Diagramme de classes : exemple de la STL

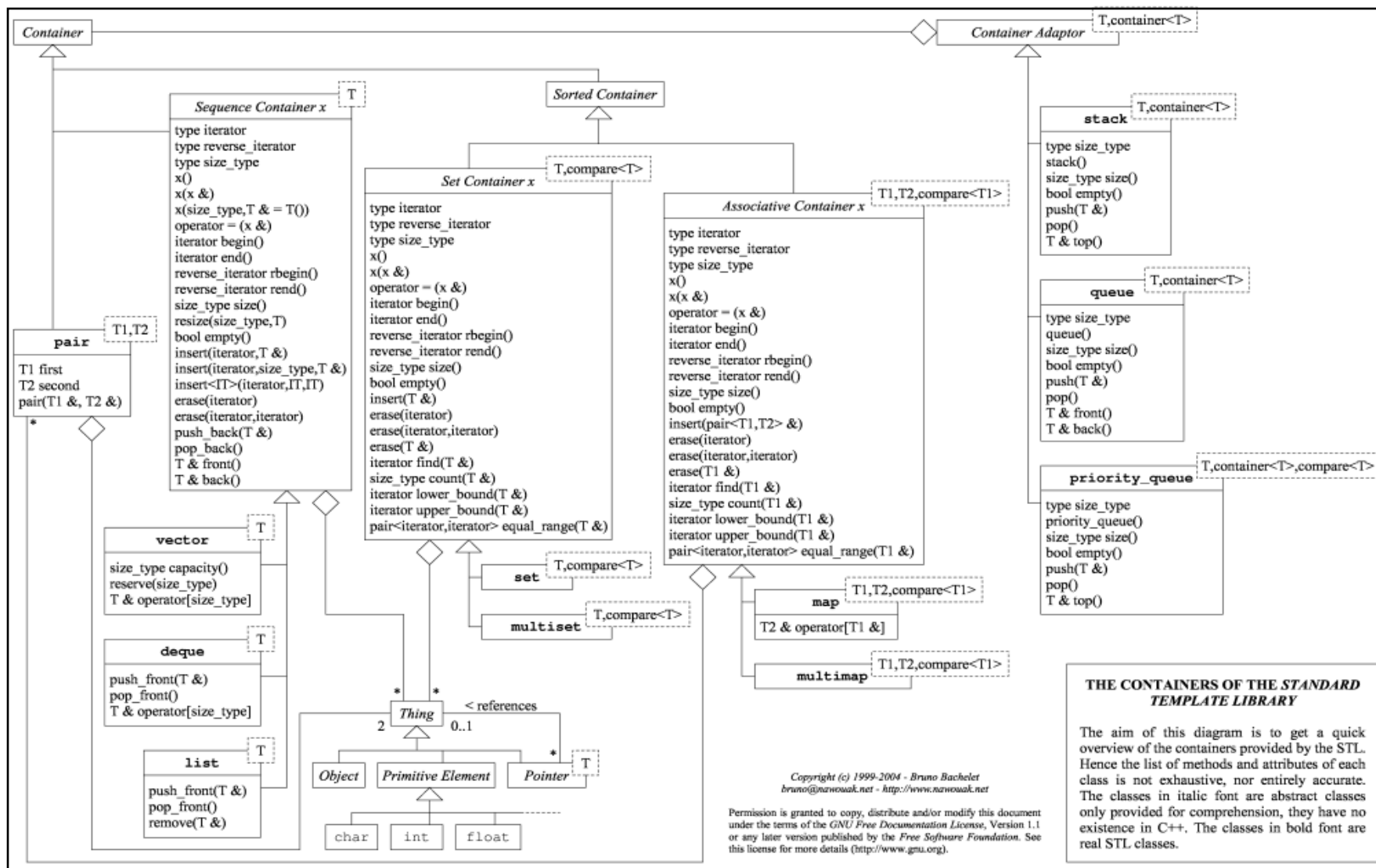


Diagramme de paquets (1/2)

C'est un élément de modélisation qui

- contient d'autres éléments de modélisation (classes, autres paquets, ...)
- peut contenir des dépendances intra-paquetage (associations entre éléments du paquetage) ou inter-paquetages

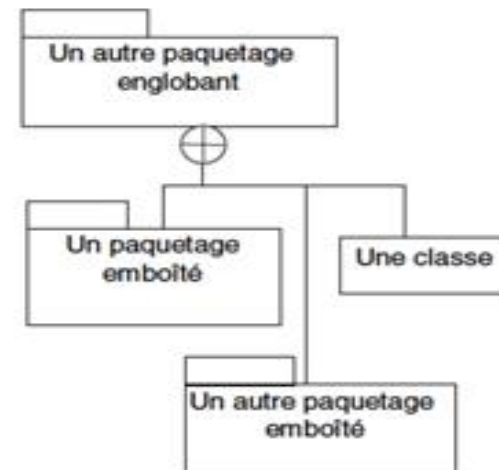
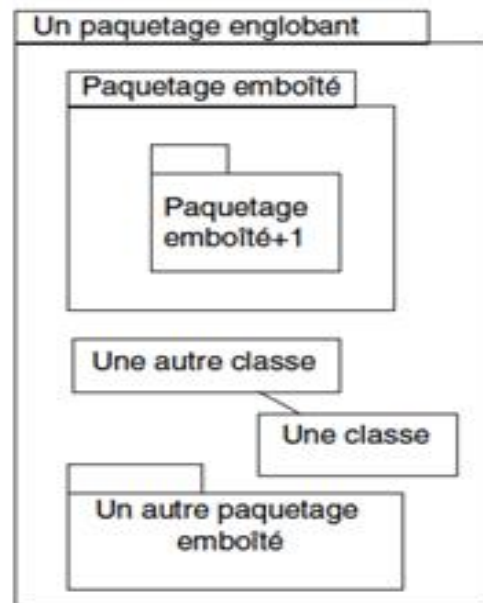
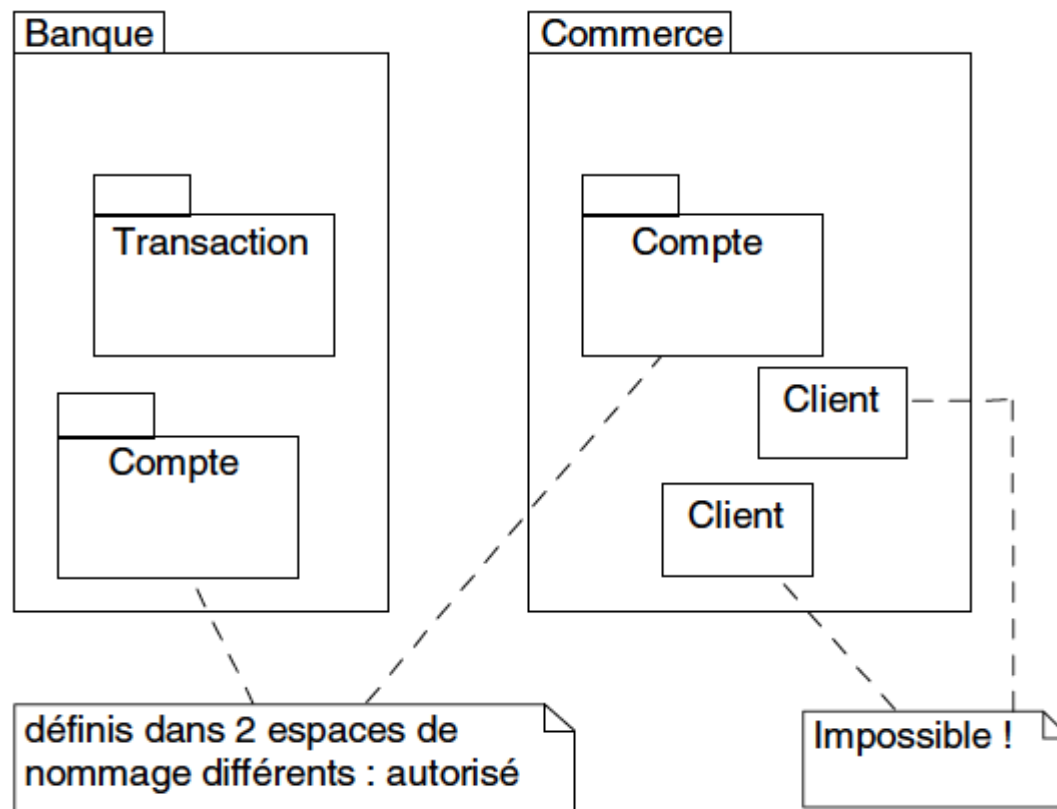


Diagramme de paquetsages (2/2)

Un paquetage correspond à un *namespace* en C++; 2 éléments dans 2 paquetages différents sont différents, quel que soit leur nom ($\text{Banque::Compte} \neq \text{Client::Compte}$)



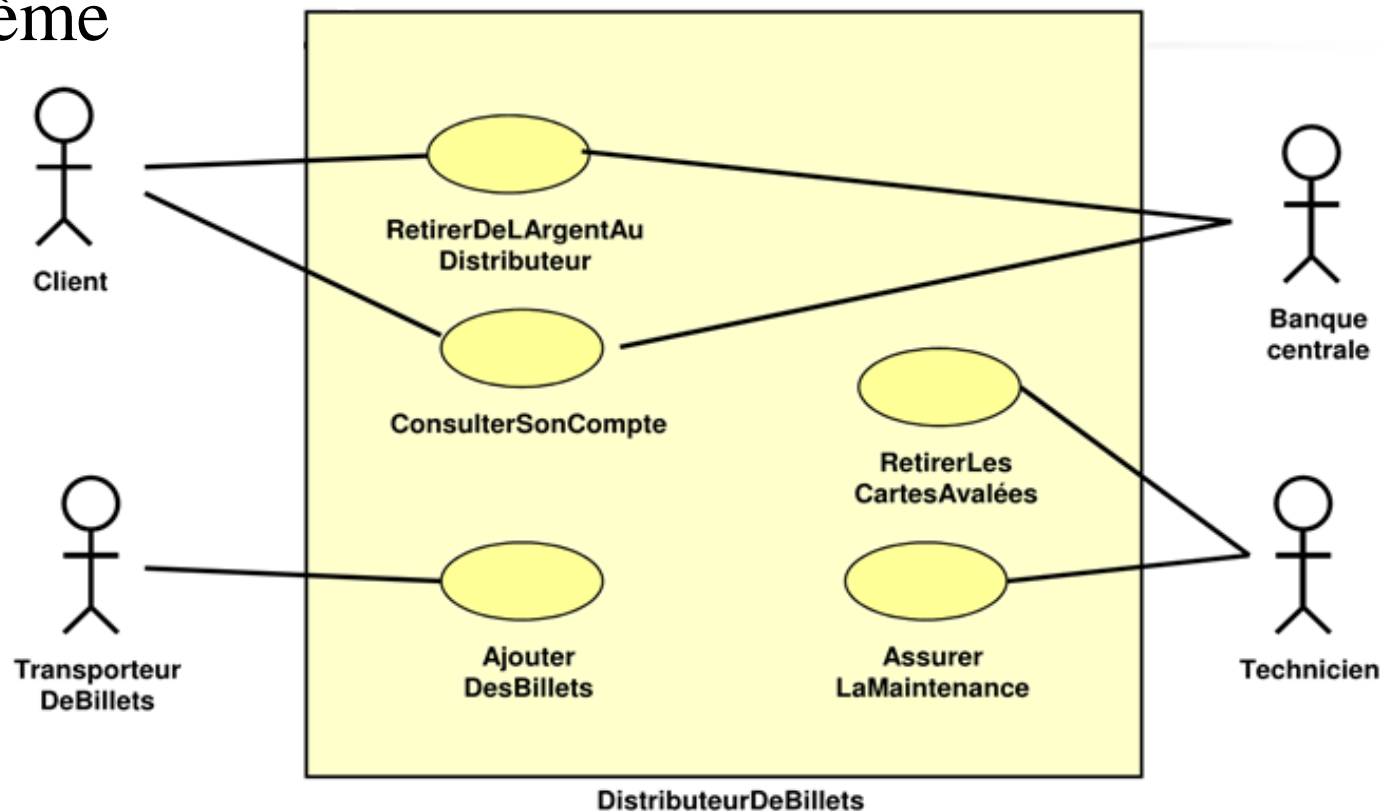
UML : Modélisation du Comportement

- Diagramme de cas d'utilisation (CU) :
Syntaxe. Démarche
- Description d'un CU *via* des diagrammes de séquence. Validation par des *scenarii*
- Diagrammes comportementaux liés aux objets (non traité)

Diagramme de cas d'utilisation (CU)

- les acteurs
- les cas d'utilisation
- le système

+ relations entre ces éléments



Démarche pour l'élaboration des CU

(I) Définir le modèle de cas d'utilisation

- (1) Introduire le système
- (2) Trouver les acteurs (\neq des utilisateurs)
- (3) Décrire brièvement chaque acteur
- (4) Trouver les cas d'utilisation et exprimer les relations
- (5) Décrire le modèle comme un tout

(II) Définir les priorités entres CU

(III) Détailler chaque CU (en fonction des priorités)

Description d'un cas d'utilisation

Description détaillée via des diagrammes de séquence :
Préconditions, débuts, postconditions, fins alternatives,
contraintes non fonctionnelles, ...

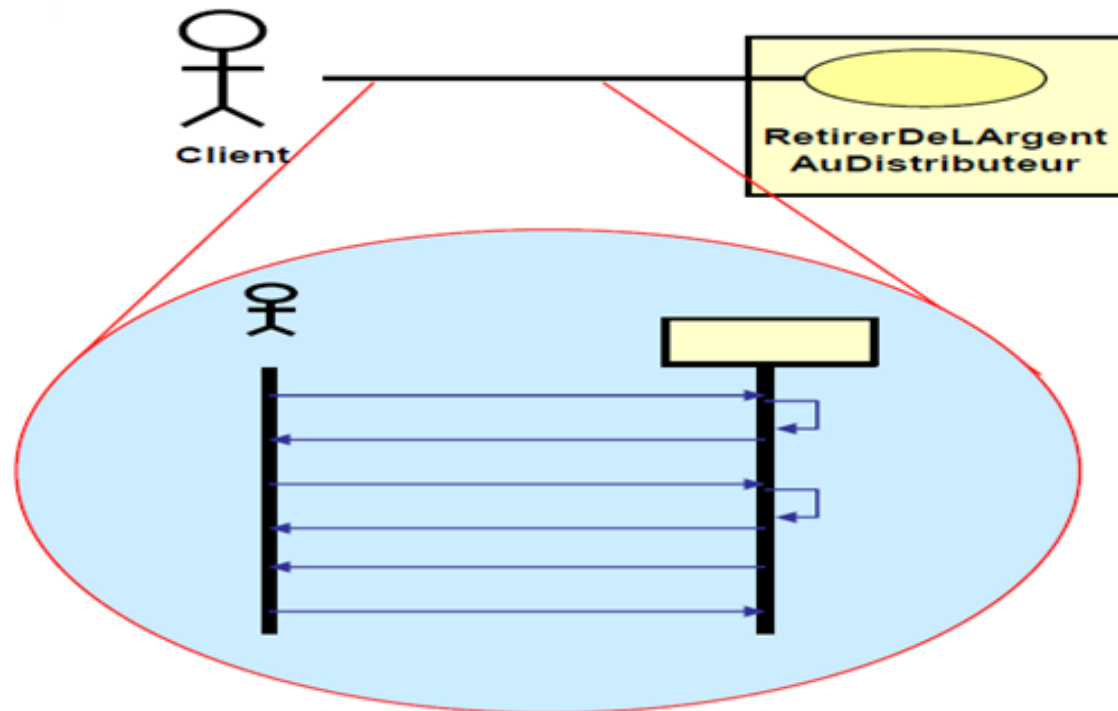


Diagramme de séquence textuel (1/3)

Cas général (1/2)



Retirer
DeLArgent
AuDistributeur

Précondition :

Le distributeur contient des billets, il est en attente d'une opération, il n'est ni en panne, ni en maintenance

Début : lorsqu'un client introduit sa carte bancaire dans le distributeur.


Fin : lorsque la carte bancaire et les billets sont sortis.

Postcondition :

Si de l'argent a pu être retiré la somme d'argent sur le compte est égale à la somme d'argent qu'il y avait avant, moins le montant du retrait. Sinon la somme d'argent sur le compte est la même qu'avant.

Diagramme de séquence textuel (2/3)

Cas général (2/2)



Retirer
DeLArgent
AuDistributeur

Déroulement normal :

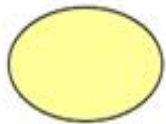
- (1) le *client* introduit sa carte bancaire
- (2) le *système* lit la carte et vérifie si la carte est valide
- (3) le *système* demande au client de taper son code
- (4) le *client* tape son code confidentiel
- (5) le *système* vérifie que le code correspond à la carte
- (6) le *client* choisi une opération de retrait
- (7) le *système* demande le montant à retirer

...

Variantes :

- (A) *Carte invalide* : au cours de l'étape (2) si la carte est jugée invalide, le système affiche un message d'erreur, rejète la carte et le cas d'utilisation se termine.
- (B) *Code erroné* : au cours de l'étape (5) ...

Diagramme de séquence textuel (3/3)



Retirer
DeLArgent
AuDistributeur

Contraintes non fonctionnelles :

(A) *Performance* : le système doit réagir dans un délai inférieur à 4 secondes, quelque soit l'action de l'utilisateur.

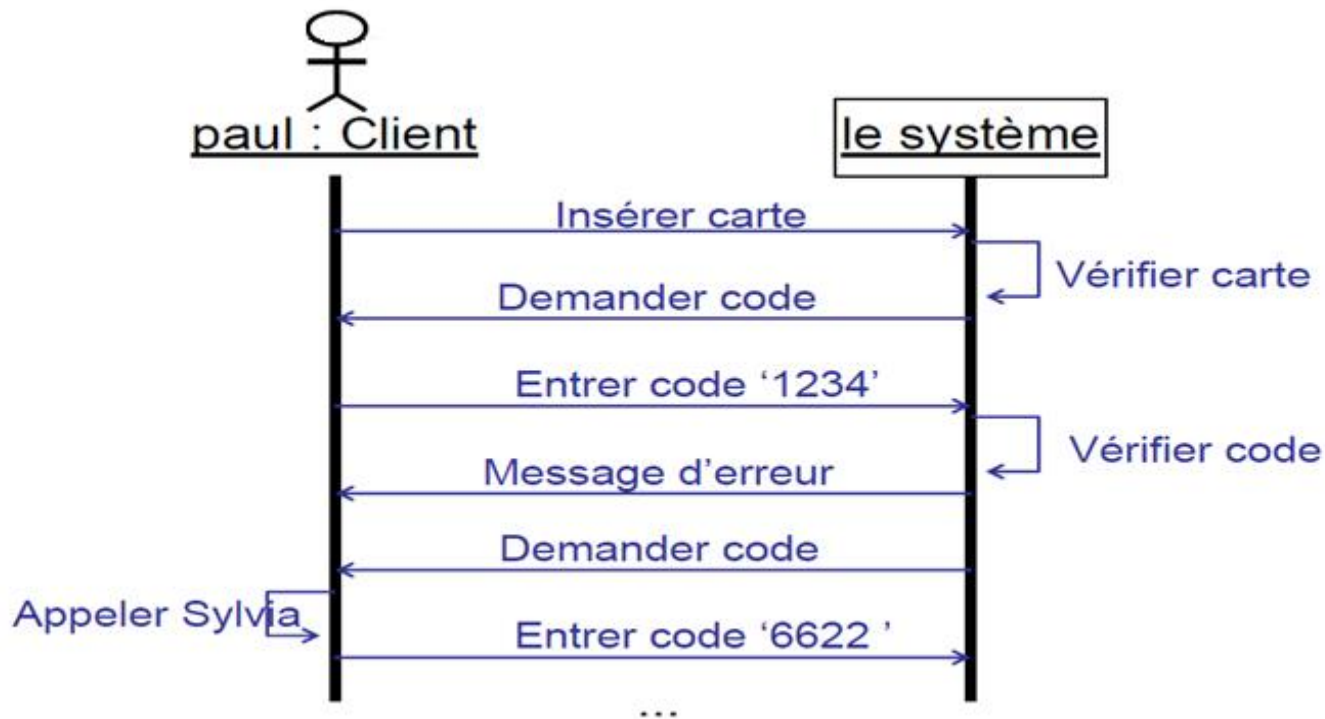
(B) *Résistance aux pannes* : si une coupure de courant ou une autre défaillance survient au cours du cas d'utilisation, la transaction sera annulée, l'argent ne sera pas distribué. Le système doit pouvoir redémarrer automatiquement dans un état cohérent et sans intervention humaine.

(C) *Résistance à la charge* : le système doit pouvoir gérer plus de 1000 retraits d'argent par jour

...

Diagramme de séquence : Scénario

Scénario : niveau instance, permet de valider un CU; un CU = un ensemble de scénarii



CU et Diagrammes de séquence : en résumé

- Décrire les buts et les besoins des acteurs, les interactions, mais pas l'interface (le pourquoi, pour qui, mais pas le comment)
- Se limiter aux cas d'utilisation
« essentiels »
- Intégrer les « responsabilités du système »
- Définir le « style UML » adapté à ses besoins