Programmation Objet,
Le langage C++

Adel KHALFALLAH
Institut Supérieur d'Informatique – El Manar
Adel.Khalfallah@fst.rnu.tn
Adel_Khalfallah@yahoo.fr

C++
Préliminaires

Adel KHALFALLAH
Institut Supérieur d'Informatique – El Manar
Adel.Khalfallah@fst.rnu.tn
Adel_Khalfallah@yahoo.fr

Standardisation

- ANSI C++
- Standard ISO/IEC 14882

Directives au pré processeur

- Macros

```
#define Symbole Définition
#define MAX 100
#define Begin {
#define Min(a,b) ((a<b)?(a):(b))
```

  - **Symbole** est remplacé par **Définition** par le pré processeur avant la compilation.
  - Ne sont pas « comprises » par les outils tels que débuggeur, générateur de références croisées,…
  - Une macro ne peut pas être surchargée et ne peut pas être récursive
  - Remplacées par des constructions plus appropriées en C++ (généricité, fonctions inline, constantes…)

Directives au pré processeur

- `##` désigne l'opérateur de concaténation

```
#define concat(a,b)    a##b
int concat(x,123);// déclare int x123
```

- `#undef Symbole` Permet de rendre Symbole indéfini, la macro éventuelle correspondante est « annulée ».
- Compilation conditionnelle :
  - `#ifdef Symbole...#endif` : le code inclut avant la directive `#endif` ne sera transmis au compilateur que si la macro `Symbole` est définie
  - `#ifndef Symbole...#endif` : le code inclut avant la directive `#endif` ne sera transmis au compilateur que si la macro `Symbole` n'est pas définie

Types fondamentaux

- Booléen : **`bool`**
- Caractères : **`char`**
- Entier : **`short - int - long`** (**`signed`** / **`unsigned`**)

} Integral types

- Réels : **`float - double`**

} Floating point types

Integral Types
Floating point types

} Arithmetic types (built-in types)

Autres types

- Enumération : **`enum`**

  `enum` **`{`**`Lun, Mar, Mer`**`}`** `Jour;`
- Classes

} Types utilisateurs

- Absence de type : **`void`**

Types composés

- Pointeur : **`int *`**
- Tableaux : **`int[]`**
- Enregistrement (record, data structure) et classes

Booléens

Conversions:

- true $\rightarrow$ 1
- false $\rightarrow$ 0
- Entier $\neq 0$ $\rightarrow$ true
- 0 $\rightarrow$ false

Exemple :

```
bool a=true, b=true;
int i=a+b;
cout << i
```

$\Rightarrow$ Affiche 2

Caractères

- On peut supposer que :
  - Les chiffres 0..9, les caractères latins ‘a’..’z’ et ‘A’..’Z’ et les caractères de ponctuation usuels sont supportés
- On ne peut pas supposer que :
  - Il n’y a que 127 caractères
  - Les codes des caractères sont successifs (EBDIC trou entre ‘i’ et ‘j’)
  - Les caractères de la syntaxe c++ i.e. \ [ ] { } sont toujours disponibles
- char → int ∈ [0..255] ou [-127..127] ?
  - Dépend de l’implantation
  - **`signed char et unsigned char`** lèvent l’ambiguïté
  - **`wchar_t`** caractères UNICODE

Tailles

Il est garanti que :

1 ≡ sizeof**(char)** ≤ sizeof**(short)** ≤ sizeof**(int)** ≤ sizeof**(long)**

```
1 ≤ sizeof(bool) ≤ sizeof(long)
sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
sizeof(N) ≡ sizeof(signed N) ≡ sizeof(unsigned N)
```

Où N peut être **`char`**, **`short int`**, **`int`**, ou **`long int`**.

Tailles

- Il est aussi garanti que :
  - **`char`** a au moins 8 bits,
  - **`short`** a au moins 16 bits,
  - **`long`** a au moins 32 bits.
- **`<limits>`** permet de retrouver les caractéristiques d'une implantation
  - **`numeric_limits<float>::max()`**
  - **`numeric_limits<char>::is_signed`**
  - …

Le type void

- procedure :

```
void Proc()
```

- Pointeur sur un objet de type indéfini :

```
void *ptr;
```

- On peut pas déclarer de variables de type **`void`**

Les énumérations

- Le domaine (range) d'un énumération :

  Soit n, $2^n \leq$ Max(Enumération)

  – valeurs positives uniquement, Domaine : $0..2^n - 1$

  – valeurs positives et négatives, Domaine : $-2^n..2^n - 1$

  Exemples :

```
enum T1  {Noir, Blanc}          ⇒ 0..1                    (n=1)
enum T2 {x=4,y=9}               ⇒ 0..15                   (n=4)
enum T3 {n=-10,m=1 000 000}     ⇒ -1 048 576..1 048 575   (n=20)
```

Les énumérations

- Un type intégral peut être explicitement convertit en un type énuméré :
  - `enum flag { x=1, y=2, z=4, e=8 };` / / range 0..15
  - `flag f1 = 5;` / / Erreur : 5 n'est pas de type flag
  - `flag f2 = flag(5);` / / ok : 5 $\in$ 0..15
  - `flag f4 = flag(99);` / / Ok mais indéfini : 99 $\notin$ 0..15
- Lorsque le domaine d'une énumération E est inclut dans celui de **`int`**, alors :

`Sizeof(E)` $\leq$ `sizeof(int)`

Structure des déclarations

[Spécificateur] Type Déclaration [Initialisation]

Spécificateur : **`static, const, virtual, extern`**,…

Déclaration : Identificateur [Opérateur déclaratif]

Opérateurs déclaratifs :

Symbole	Sémantique	Position
`*`	pointeur	Préfixe
`*const`	pointeur constant	Préfixe
`&`	référence	Préfixe
`[]`	Tableau	Post-fixe
`()`	fonction	Post-fixe

Post-fixe sont plus « prioritaires » que Préfixe

Initialisations

- Les variables globales et statiques sont initialisées par défaut au 0 de leurs types
- Initialiseurs
  - `int a[] ={1,2};` // tableau de taille 2
  - `int a[2] ={1,2};` //Idem
  - `int a[2] ={1,2,3};` //Erreur
  - `int a[5] ={1,2};` //Tab[2] est indéfini
  - `Point P(1,2);` //constructeur
  - `Point P();` // Attention : ceci est une fonction et non un // initialiseur
- `T t=T();` //désigne la valeur par défaut du type
  - `int i=int()`
  - `complex z=complex();`

Initialisations

Structure

```
struct Addresse {
    char* Nom;
    long int Numero;
    char *Rue;
    char Ville[2];
    long CodePostal;
};

                Nom  No   Rue        Ville    CP
Address UneAddr={"Ali",61,"Carthage",{'T','U'},1002};
```

Constantes

```
const int MAX = 100;
const char Directions[]={'S','N','E','O'}
```

- Une constante doit être initialisée
- La valeur d'une constante ne peut plus changer
- Pointeur sur objet constant : **`const T *ptr`**
- Pointeur constant : **`T * const ptr`**
- Pointeur constant sur objet constant :

  `const T *const Ptr`
- Paramètre constant : **`F(const int *X)`**

lvalue

- A l'origine : tout ce qui peut se trouver à gauche d'une affectation :
  - Variable : x=…
  - Cellule de tableau T[I]=…
  - Membre d'enregistrement Addr.Nom=…
  - Objet pointé : *ptr=…
- Actuellement : toute entité qui dispose d'une adresse mémoire

```
const int x=20; // x dispose d'une adresse mémoire (sauf
                // optimisation) donc c'est une lvalue
                //mais x ne peut pas être à gauche d'une affectation
```

Références

- Alias
  - `int i; int &j=i;` // doit toujours être initialisé
  - `double &i=1;` //Erreur
  - `const double &i=1;` //Ok ⇔ `double tmp =double(1);`
    `//  const double &i=tmp;`
    // à éviter !
- Passage de paramètres

```
void Echange(int &x,int &y){
    int tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

appel : `int u; int v; … Echange(u,v);` // lisibilité ? (Localité)

- Signification de : `F(const T &t)` ?

Références

- Retour par référence

```
const int Max;
int Tableau[max];
int &Acces(int i){
  if (i<Max)
     return Tableau[i];
  else throw "indice invalide";
}

main(){
  Acces(1)=999; //Acces devient une lvalue
}
```

Références

Attention à :

```
int &Incr(int x){
      int y;
      y = 2*x+1;
      if (y <= Max)
         return y;//Mauvais usage
     else return x;  //Mauvais usage
}
```

Liste des opérateurs

Priorité 0

Portée :

Nomclasse : : membre

NomNamespace : : Identifiant

: : Nom //global

: : Nom-qualifié //global

Liste des opérateurs

Priorité 1

Sélection de membre :

objet . membre

pointeur->membre

Indexation : pointeur [expression]

Appel de fonction : expression (liste expression)

Construction de valeur : type (liste expression)

Post incrémentation : lvalue ++

Post décrémentation : lvalue --

Identification de type : typeid (type | expression) //Statique | Dynamique

Conversion contrôlée dynamique : dynamic_cast <type> (expression)

Conversion contrôlée statique : static_ cast <type> (expression)

Conversion non contrôlée : reinterpret_cast <type> (expression)

Conversion à constant : const_cast < type> (expression)

Liste des opérateurs

Priorité 2

Taille : `sizeof`(expression | type) //Taille d'un objet ou d'un type

Pré incrementation : `++` lvalue

Pré décrémentation : `--`lvalue

Complément : `~` expression

Non : `!` expression

Moins unaire : `-` expression

Plus unaire : `+` expression

Adresse : `&` lvalue

Déréférenciation : `*`expression

Liste des opérateurs

Priorité 2 - Suite

Allocation :

`new` type

`new` type (liste expression)

`new` (liste expression) type

`new` (liste expression) type (liste expression)

Destruction :

`delete` pointeur

`delete []` pointeur

Conversion de type : `(` type`)` expression

Liste des opérateurs

Priorité 3

Sélection de membre:

objet `.*` Pointeur_Sur_Membre

pointeur`->*` Pointeur_Sur_Membre

Priorité 4

Multiplication : expression `*` expression

Division : expression `/` expression

Modulo : expression `%` expression

Liste des opérateurs

Priorité 5

Addition : expression + expression
Soustraction : expression − expression

Priorité 6

Shift gauche : expression << expression
Shift droit : expression >> expression

Priorité 7

Strictement inférieur : expression < expression
Inférieur ou égal : expression <= expression
Strictement supérieur : expression > expression
Supérieur ou égal : expression >= expression

Liste des opérateurs

Egal : expression == expression
Différent : expression != expression — Priorité 8

Et bit à bit : expression & expression — Priorité 9

Ou exclusif bit à bit expression ^ expression — Priorité 10

Ou bit à bit : expression | expression — Priorité 11

Et logique : expression && expression — Priorité 12

Ou logique : expression || expression — Priorité 13

Liste des opérateurs

Priorité 14

Affectation : lvalue **=** expression

Multiplication et affectation : lvalue ***=** expression

Division et affectation : lvalue **/=** expression

Modulo et affectation : lvalue **%=** expression

Addition et affectation : lvalue **+=** expression

Soustraction et affectation : lvalue**-=**expression

Shift gauche et affectation : lvalue **<<=** expression

Shift droit et affectation : lvalue **>>=** expression

Et bit à bit et affectation : lvalue **&=** expression

Ou bit à bit et affectation : lvalue **|=** expression

Ou exclusif bit à bit et affectation : lvalue **^=** expression

Liste des opérateurs

Expression conditionnelle : expr `?` expr `:` expression — Priorité 15

Soulèvement d'exception : `throw` expression — Priorité 16

Séquence d'expressions : expression `,` expression — Priorité 17

Opérateurs

- Les opérateurs unaires et l'affectation ont une associativité droite tous les autres ont une associativité gauche :

```
a=b=c          a+b+c          *p++     ?
a=(b=c)        (a+b)+c        *(p++)
```

- L'ordre d'évaluation des sous expressions est indéfini

`f(2) + g(3)` pour certains compilateurs `g` peut-être appelé avant `f`

```
int i=1
v[i]=i++   //Ambigu: selon le compilateur signifie
           // v[1]=2 ou v[2]=1
```

Opérateurs

- Les opérateurs '**&&**' et '**||**' garantissent que l'opérande gauche est évalué avant l'opérande droit :
- Pour **&&** l'opérande droit n'est évalué qui si l'opérande gauche est vrai
- Pour **||** l'opérande droit n'est évalué qui si l'opérande gauche est faux

Il en résulte que

```
if (x!=0) && ((y/x) > 1) …
```

est correct dans tous les compilateurs

Opérateurs

- L'opérateur '`,`' garantit que l'opérande gauche est évalué avant l'opérande droit.

```
F1((i++,v[i]))  // 1 argument, i++ est évalué avant v[i]
F2(i++,v[i])  // 2 arguments, l'ordre d'évaluation est indéfini.
```

- `2<=x<=3` //Légal mais toujours vrai.
  // Ne signifie pas `(2<=x && x<=3)`

Cast (Forçage)

```
void* malloc(size_  t);
void f(){
  int* p = static_cast<int*>(malloc(100));
            // Le pointeur retourné par malloc est interprété
            // comme pointeur sur entier
IO_device* d1;
  d1 = reinterpret_cast<IO_device*>(0Xff00);
  //0Xff00 est interprété (sans contrôle) comme l'adresse
  // d'un IO_Device
...
}
```

Déclarations

Standard ISO/IEC 14882

- `if (T t=Expression) {`

  `. . .` // t est visible

  `}`

  `else {`

  `. . .` // t est visible

  `}` // t n'est plus visible
- `for (T t=Expression,Condition,Incrémentation) {`

  `. . .` //t est visible

  `}` // t n'est plus visible

Déclarations

Microsoft Visual C++ 6.0

```
for (T t=Expression,Condition,Incrémentation){
...
}
```

```
T  t=Expression
while (Condition){
...//t est visible
Incrémentation
}
//t est visible
```

Déclarations

ANSI ??

inline

- **`inline`** indique au compilateur qu'on souhaite que l'appel d'une fonction soit remplacée par le corps de la fonction :

```
inline int Max(int a,int b) { return (a>b) ? a : b;}

Main(){
   int i,j,k;
   ...
   k=Max(i,j);// lors de la génération de code sera remplacé par :
                  //  k= (a<b) ? a : b  et non par le traditionnel empilement de
                  //  l'adresse de retour; empilement des paramètres
                  //  branchement à l'adresse de la fonction, …
}
```

inline

- inline n'est pas garanti, exemple:

```
inline int fac(int n){
  return (n<2) ? 1 : n*fac(n1);
}
```

- MS Visual C++ 6.0 :

```
#pragma inline_depth( [0... 255] )
#pragma inline_recursion( [{on | off}] )
```

- inline # macro

```
#define Max(a,b) ((a > b) ? a : b)
int i=1,j=2;
printf("%d",Max(i,j++))
```
//affiche 4, fonction inline affiche 3

static

- variable qui conserve sa valeur comme une globale mais qui n'est visible que dans une certaine portée comme une locale

```
char *Mot= "Bonjour";
void lettre(){
   int Indice=0;//Créée et initalisée 7 fois
   static int IndiceStat=0;  //Créée et initalisée 1 fois
   printf("%c%c",Mot[IndiceStat++],Mot[Indice++])
}//Indice est détruite 7 fois
main(){
for (int i=0;i<=6;i++) lettre();//IndiceStat n'est pas visible ici
//Affiche BBoBnBjBoBuBrB
}//fin du programme : IndiceStat est détruite
```

Valeurs par défaut des paramètres

```
void Imprime(int Val, int Base=10){
...
}

main(){
   Imprime(2005);//Imprime en base 10
   Imprime(2005,2);//Imprime en base 2
}
```

- Tous les paramètres ayant une valeur par défaut doivent être regroupés à la fin du profil

Valeurs par défaut des paramètres

- La valeur par défaut peut être redéfinie dans des portées différentes.

```
{// Dans ce bloc imprimer en base 16
  void Imprime(int Val, int Base=16);
  Imprime(2005);//Imprime en base 16
  Imprime(2005,2);//Imprime en base 2
}
...
{// Dans ce bloc imprimer en base 8
  void Imprime(int Val, int Base=8);
  Imprime(2005);//Imprime en base 8
  Imprime(2005,2);//Imprime en base 2
}
```

Fonctions à arguments variables

- Lorsque aussi bien le nombre que le type des arguments n'est pas connu à priori, il est utile d'introduire une fonction à arguments variables.

```
int printf(const char* ...);
```

- Sous Windows, il doit exister au moins un paramètre formel. Sous Unix, cela n'est pas nécessaire.
- Les pramètres se manipulent avec 3 macros et un type définit dans **<cstdarg>** (et **<varargs.h>** pour Unix)
  - **va_list :** type de la liste des paramètres.
  - **va_start** : macro qui initialise la liste des paramètres.
  - **va_arg** : macro qui permet d'interpréter le paramètre courant dans un type donné.
  - **va_end** : macro qui permet de libérer la liste des paramètres

Fonctions à arguments variables

```
#ifdef ANSI /* Compatibilité ANSI */
#include <stdarg.h>
int Moyenne( int Premier, ... );
#else /* Compatibilité UNIX */
#include <varargs.h>
int Moyenne( va_list );
#endif
void main( void ) {
  printf("Moy 3 : %d", Moyenne( 2, 3, 4, -1 ) );
  printf("Moy 4 : %d", Moyenne( 5, 7, 9, 11, -1 ) );
  printf("Moy 0 : %d", Moyenne( -1 ) );
}
```

Fonctions à arguments variables

```
#ifdef ANSI /* Version ANSI */
int Moyenne( int Premier, ... ) {
  int Cpt = 0, Total = 0, i = Premier;
  va_list ListePar; // Liste des paramètres
  va_start(ListePar,Premier ); //Initialise la liste des paramètres
  while( i != -1 ) {// -1 dernière valeur
     Total += i; Cpt++;
     i = va_arg( ListePar, int);
          // Récupère le paramètre courant et l'interprète comme un int
  }
  va_end( marker );// Libère la liste
return( Cpt ? (Total / Cpt) : 0 ); }
```

Fonctions à arguments variables

```
#else // Version UNIX
int Moyenne( va_alist ) va_dcl {// Doit utiliser l'ancien style
  int i, Cpt=0, Total=0;
  va_list ListePar; // Déclaration de la liste des paramètres
  va_start(ListePar ); // Initialisation liste paramètres
  for(;(i = va_arg(ListePar, int)) != -1; Cpt++ )
      Total += i;
  va_end(ListePar ); //Libère la liste
  return(Cpt ? (Total / Cpt) : 0 ); }
#endif
```

Pointeur sur fonction

- Une fonction peut être appelée à travers un pointeur

```
void UneFonction(string s){
…
}

void (*PtrFct)(string s);
```

// PtrFct : pointeur sur fonction qui prend une string en paramètre et qui ne
// retourne pas de résultat

```
main(){
PtrFct=&UneFonction;
```
// On peut aussi écrire `PtrFct=UneFonction`

```
(*PtrFct)("Un Message");
```
// Ou aussi `PtrFct("Un Message")`

Pointeur sur fonction

- Lorsque 2 pointeurs pointent sur des fonctions avec des profils identiques, ils peuvent être affectés l'un à l'autre
- L'appel doit correspondre au profil exact, il n'y a pas de conversions implicites
- Tableau de pointeurs sur fonction

```
typedef void (*TPtrFct)();
TPtrFct TabFctEdit[3]={&Couper,&Coller,&Copier};
TPtrFct TabFctFich[]={&Ouvrir,&Fermer,&Save,&New};
TPtrFct *Souris=TabFctEdit;// La souris est sur le menu Edition
Souris[2];// Appel à Copier
```

Pointeur sur fonction

- Fonction en paramètre

```
typedef int (*TFctCmp)(const void*,const void*);
```

```
void Trier(void* Tab, size_t NbrElt, size_t
   TailleElt, TFctCmp Ordre);
```

//Trier le tableau **`Tab`** d'éléments de type non spécifiés en utilisant

//la relation d'ordre **`Ordre`**. **`NbrElt`** : Nombre d'éléments à trier dans le tableau. **`TailleElt`** : Taille d'un élément du tableau.

Compilation séparée

- Une variable peut être *déclarée* plusieurs fois dans un programme mais elle ne peut être *définie* qu'une seule fois

Fich1. cpp

```
int x=10;// Définition de x
int y=20;// Définition de y
void F();// Déclaration de F()
void G() {F();}// Définition de G()
```

Fich2.cpp

```
extern int x;// Déclaration de x
void F(){...}// Définition de F
void G();// Déclaration de G()
int y;// Erreur : Redéfinition de y , bien que y ne soit pas visible ici
main(){
  printf("x=%d",x);
  G();}
```

Compilation séparée

- Un variable définie **`static`** ne peut plus être redéclarée avec **`extern`**
- **`extern`** combiné avec une initialisation correspond à une définition et non pas une déclaration.
- Un symbole qui peut être déclarée dans une unité différente de celle où il est défini est dit à liaison externe (external linkage) alors qu'un symbol qui ne peut être utilisé que dans l'unité où il est défini est dit à liaison interne.
- Les fonctions inline, les constantes et les définitions de types sont à liaisons internes : ils ne peuvent donc pas être redéclarés, mais ils peuvent être redéfinis, toutefois, ce qui suit permet de redéclarer une constante

  Fich1.cpp : **`extern const int a=7`**
  Fich2.cpp : **`extern const int`**

Compilation séparée

- **`extern`** permet d'introduire une séparation entre l'interface et l'implantation pour les données :

Fich1.h :

// Fichier d'interface ne contient que des déclarations

```
extern int X; // X est déclaré mais non défini
...//Autres déclarations faisant partie de l'interface
```

Fich1.cpp

```
int X; //X est défini (implanté)
```

L'unité qui fait **`#include "Fich1.h"`** ne verra pas l'implantation de x

Compilation séparée

- Le mécanisme d'inclusion est l'un des outils de la compilation séparée, il s'appuie sur la directive **`#include`**
  - **`#include`** `"`Fich`"` : Fich se trouve dans le même répertoire que le fichier courant
  - **`#include`** <Fich> : Fich se trouve dans l'un des répertoires prédéfinis de l'environnement de développement
- Recompiler à chaque fois le fichier d'inclusion peut sembler aberrant pour cela certains environnements offrent des fonctions de pré compilation.

Compilation séparée

- Le besoin de structurer le programme en modules autonomes peut entrainer qu'un module se retrouve inclut 2 fois dans la même unité ce qui risque de provoquer des erreurs, pour éviter cela on utilise des gardes :

```
#ifndef MODULE1_H
#define MODULE1_H
#include "Module1.h"
...// définition du module client
#endif
```

Compilation séparée

- Il est habituel de trouver dans des fichiers d'entête les structures suivantes :
  - Namespaces non anonymes **`namespace N { /* ... */ }`**
  - Definitions de types **`struct Point { int x, y; };`**
  - Déclarations de génériques **`template<class T> class Z;`**
  - Définitions de génériques **`template<class T> class V {...};`**
  - Déclarations de fonctions **`extern int strlen(const char*);`**
  - Définitions de fonctions inline **`inline char get(char* p){...}`**
  - Déclarations de données **`extern int a;`**
  - Définitions de constantes **`const float pi = 3.141593;`**
  - Enumérations **`enum Feux {Rouge, Orange, Vert};`**
  - Déclarations de noms **`class Matrice;`**
  - Directives d'inclusions **`#include <algorithm>`**
  - Définitions de macros **`#define VERSION 12`**
  - Directives de compilations conditionnelles **`#ifdef __cplusplus`**
  - Commentaires **`/* Module Pile ... */`**

Compilation séparée

- On ne devrait pas trouver dans un fichier d'entête les structures suivantes :
  - Définitions habituelles de fonctions **`char get(char* p) { return *p++; }`**
  - Définitions de données **`int a;`**
  - Définitions d'agrégats **`short tbl[] = { 1, 2, 3};`**
  - namespaces anonymes **`namespace { /* ... */ }`**
  - Définitions de génériques exportées **`export template<class T> f(T t) { /* ... */ }`**

Compilation séparée

- A chaque fichier d'entête standard du langage C <X.h> il correspond un fichier d'entête C++ <cX.h>

- Règle de définition unique (ODR : One Definition Rule) :
  Deux définitions d'une classe, d'un générique ou d'une fonction inline sont acceptées comme 2 facettes de la même et unique définition si et seulement si :
  - Elles sont dans 2 unités différentes
  - Elles sont identiques lexème à lexème
  - La sémantique des lexèmes est identique dans les 2 unités

  Cette règle facilite l'inclusion de fichiers de définition de classe

Compilation séparée

Commun.h

```
struct S {int a; char b;};
void F(S*);
```

Fich1.cpp

```
#include "commun.h"
```

// Utilisation de F

Fich2.cpp

```
#include "commun.h"
```

//S est introduit pour la 2éme fois

```
Void F(S* s) {...}
```

Compilation séparée

- A éviter !

Fich1.cpp

```
typedef int X
struct S {X a; int b;};
```

Fich2.cpp

```
typedef char X
struct S {X a;int b;};
```

- A éviter !

Fich1.cpp

```
struct S {Point a; int b;};// Point devrait être défini dans Fich1
```

Fich2.cpp

```
#define Point int
#include "Fich1.cpp"
```

Compilation séparée

- Spécification des conventions de liaison pour l'interfaçage avec d'autres compilateurs ou d'autres langages :

```
extern "C" int Fact(int N) // Utilisation des conventions du C

extern "C" {
   int Fact(int N);
   float Limites(Point *P);
   bool compare(char *s1,char *s2);
}
```

Fondements de l'approche objet

Adel KHALFALLAH
Institut Supérieur D'informatique – El Manar
Adel.Khalfallah@fst.rnu.tn
Adel_Khalfallah@yahoo.fr

Motivations

Quels problèmes

- Gérer la complexité
- Augmenter la réutilisation
- Faciliter la maintenance

Complexité du logiciel

Crise du logiciel (Fin Années 60) :

Etude sur 9 projets de l'administration américaine d'une valeurs de 6.8 Millions de $

- 29 % Payés mais jamais livrés
- 47 %Livrés mais jamais utilisés avec succès
- 23 % Logiciel utilisé mais avec modifications
- 1 % Utilisés tels quels

Complexité du logiciel

- Taille > 1 Million de LOC

$\Rightarrow$ Impossible à appréhender par une seule personne

- Complexité des problèmes à résoudre
  - Diversité et complexité des systèmes (Systèmes d'information, Systèmes Temps Réel, Système Experts,..)
  - Exigences contradictoires
  - Problèmes de communication (Développeur$\rightarrow$Utilisateur et Développeur $\rightarrow$ Développeur)
  - Le problème change « en cours de résolution » (l'expression de la solution modifie le problème)
  - Technologies changeantes en cours d'utilisation

Complexité du logiciel

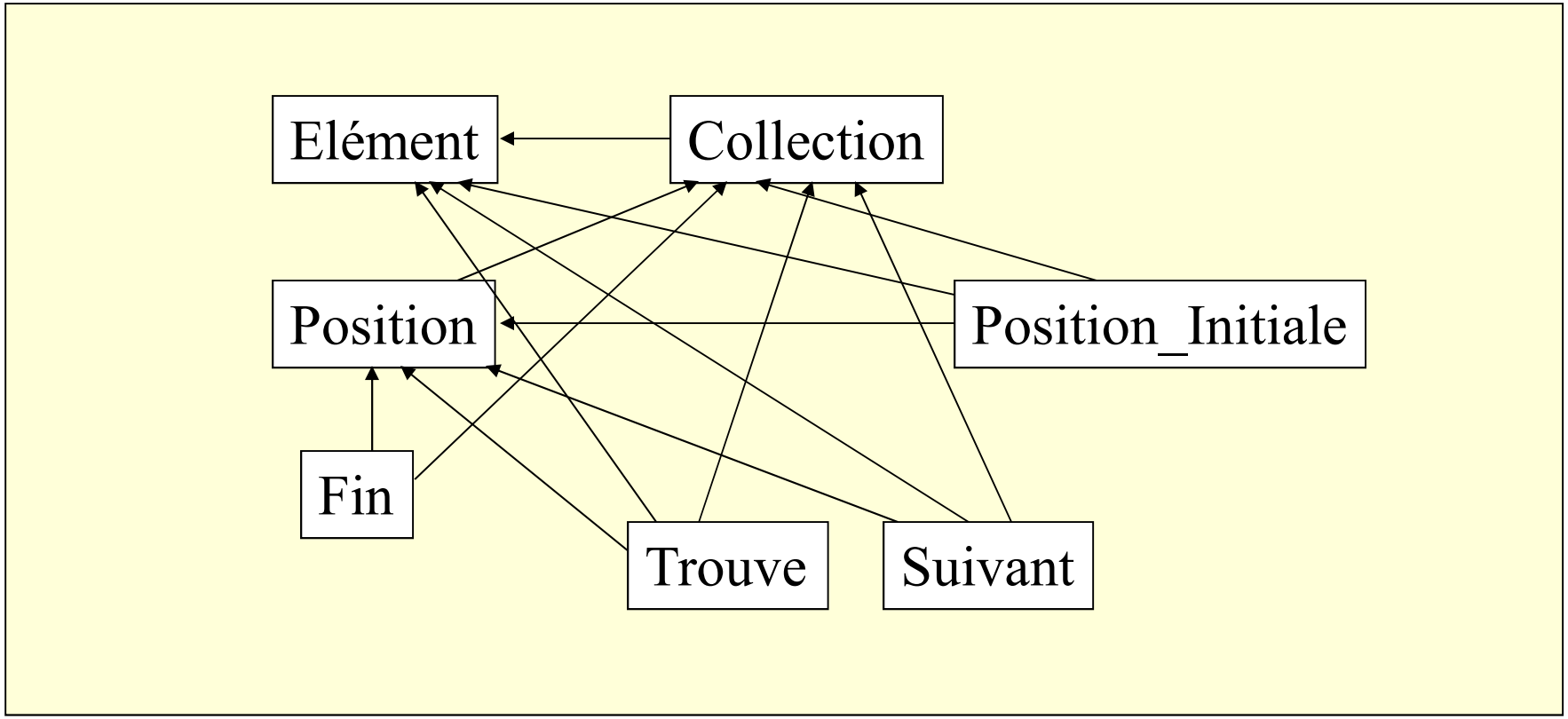
Interdépendances : Logiciel ≈ MIKADO

Complexité du logiciel

Exemple : Recherche d'un élément dans une collection:

```
Recherche (X : Elément; C : Collection): Booleen
  Pos : Position
Début
  Pos :=Position_Initiale(X,C)
  tant que non Fin(Pos,C) et non Trouve(Pos,X,C) faire
    Pos:=Suivant(Pos,X,C)
  ftque
  retourner non Fin(Pos,C)
fin
```

Complexité du logiciel



Réutilisation

- Impression de « déjà vu » (Exemple Tri)
- Les projets qui atteignent un haut niveau de réutilisation sont l'exception plutôt que la règle
- Intérêts de la réutilisation
  - Augmentation de la productivité
  - Baisse des coûts
  - Amélioration de la qualité

Réutilisation

La réutilisation concerne :

- Les produits
  - Code source
  - Données et pilotes de test
  - Modèles de spécification et de conception
  - …
- Les processus
  - Tâches répétitives ($\Rightarrow$ Nécessité de définitions formelles des activités)
  - Personnes et leurs connaissances
  - …

Réutilisation

Autres Aspects limitatifs :

- Economiques : pas de formule juridique satisfaisante
- Outils : absence d'outils de recherche et de techniques de classification appropriées

Maintenance

Importance

Enquête (1986) auprès de 55 sociétés de service en Informatique :

Environ 53 % des coûts du logiciel sont liés à la maintenance

Maintenance

Types de maintenance :

- Corrective : correction d'erreurs
- Adaptative : adapter le logiciel aux évolutions de son environnement
- Perfective : Améliorer le logiciel ou prendre en compte de nouveaux besoins
- Préventive : améliorer la maintenabilité du logiciel

Concepts de base

- Abstraction
- Encapsulation
- Modularité
- Types Abstrait de données

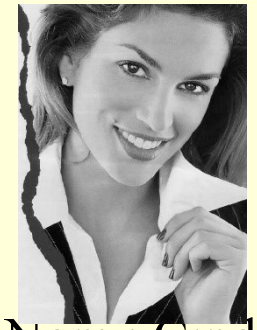
Abstraction

Définition : Une abstraction fait ressortir les *caractéristiques essentielles* d'une structure qui la distinguent de tous les autres types de structures du domaine et donc procure des *frontières conceptuelles* rigoureusement définies par rapport au *point de vue* de l'observateur

Abstraction

Caractéristiques :

Nom : Tom Cruise
Couleur des yeux : Noir
Atomes de carbones : $9.987\ 10^6$
Nombre de poils : 10 752 634 434
Taille : 1.70
Globules rouges : 7 840 656
Groupe sanguin : A +
Sexe : Masculin
Globules blancs : 9 540 786
Date de naissance : 3 Juillet 1962



Nom : Cindy Crawford
Couleur des yeux : Marron
Atomes de carbones : $9.684\ 10^6$
Nombre de poils : 11 233 122 110
Taille : 1.76
Globules rouges : 7 541 553
Groupe sanguin : B+
Sexe : Féminin
Globules blancs : 9 760 956
Date de naissance : 20 Février 1966

Le fait d'avoir les mêmes caractéristiques est le fruit du hasard

Abstraction

Caractéristiques essentielles :

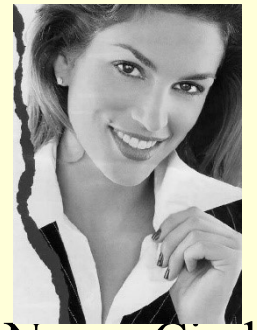
Nom : Tom Cruise
Couleur des yeux : Noir
Atomes de carbones : $9.987\ 10^6$
Nombre de poils : 10 752 634 434
Taille : 1.70
Globules rouges : 7 840 656
Groupe sanguin : A +
Sexe : Masculin
Globules blancs : 9 540 786
Date de naissance : 3 Juillet 1962

Nom : Cindy Crawford
Couleur des yeux : Marron
Atomes de carbones : $9.684\ 10^6$
Nombre de poils : 11 233 122 110
Taille : 1.76
Globules rouges : 7 541 553
Groupe sanguin : B+
Sexe : Féminin
Globules blancs : 9 760 956
Date de naissance : 20 Février 1966

Abstraction

Caractéristiques essentielles (Point de vue de l'hématologue) :

Nom : Tom Cruise
Couleur des yeux : Noir
Atomes de carbones : $9.987\ 10^6$
Nombre de poils : 10 752 634 434
Taille : 1.70
Globules rouges : 7 840 656
Groupe sanguin : A +
Sexe : Masculin
Globules blancs : 9 540 786
Date de naissance : 3 Juillet 1962



Nom : Cindy Crawford
Couleur des yeux : Marron
Atomes de carbones : $9.684\ 10^6$
Nombre de poils : 11 233 122 110
Taille : 1.76
Globules rouges : 7 541 553
Groupe sanguin : B+
Sexe : Féminin
Globules blancs : 9 760 956
Date de naissance : 20 Février 1966

Abstraction

Caractéristiques essentielles porte sur :

- Des données
- Des comportements

Exemple Voiture Immatriculée 96 TU 5475

- Données :
  - Marque : Peugeot 106
  - Couleur : Rouge
  - …
- Comportements
  - Lorsqu'on tourne le volant à la gauche la voiture tourne à gauche
  - Lorsqu'on appuie sur le frein la voiture s'arrête
  - …

Abstraction

L'abstraction du comportement doit être sans surprise

Exemples de surprises :

- *Comportement contre intuitif* : On tourne le volant à gauche la voiture tourne à droite
- *Comportement avec effet de bord* : On tourne le volant à gauche la voiture tourne à gauche et le clignotant gauche s'allume
- *Comportement masquant un cheval de Troyes* : On tourne le volant à gauche la voiture tourne à gauche mais une partie de l'huile moteur est vidangée

Abstraction

Un exemple de bonne abstraction

la télécommande

- Système qui présente des propriétés *émergentes*
- Interface explicite : le spectateur manipule un ensemble de boutons situé sur la face avant
- Chaque bouton à un rôle unique
- Chaque bouton garantit le bon fonctionnement du système même en cas de distractions de l'utilisateur

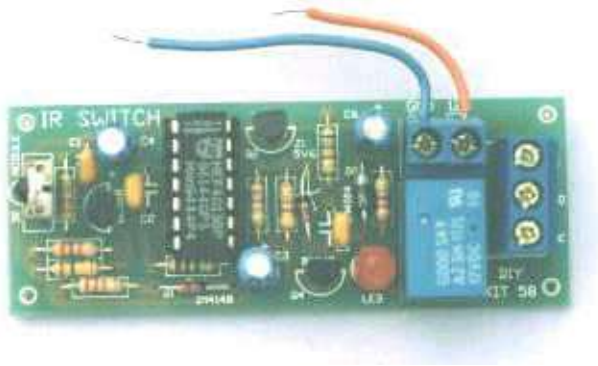
Concepts de base

- Abstraction
- Encapsulation
- Modularité
- Types Abstrait de données

Encapsulation (Data Hiding)

Définition : L'encapsulation est le procédé de *séparation* des éléments d'une abstraction qui constituent sa structure et son comportement. Elle permet de dissocier *l'interface* contractuelle de la *mise en œuvre* d'une structure

Encapsulation (Data Hiding)



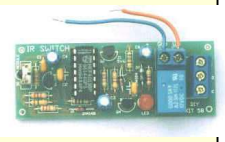
Interface	Réalisation

Encapsulation (Data Hiding)

2 Rôles :

- Utilisateur : manipule les éléments de l'abstraction qui constituent l'interface

- Implanteur : réalise ce qui est encapsulé

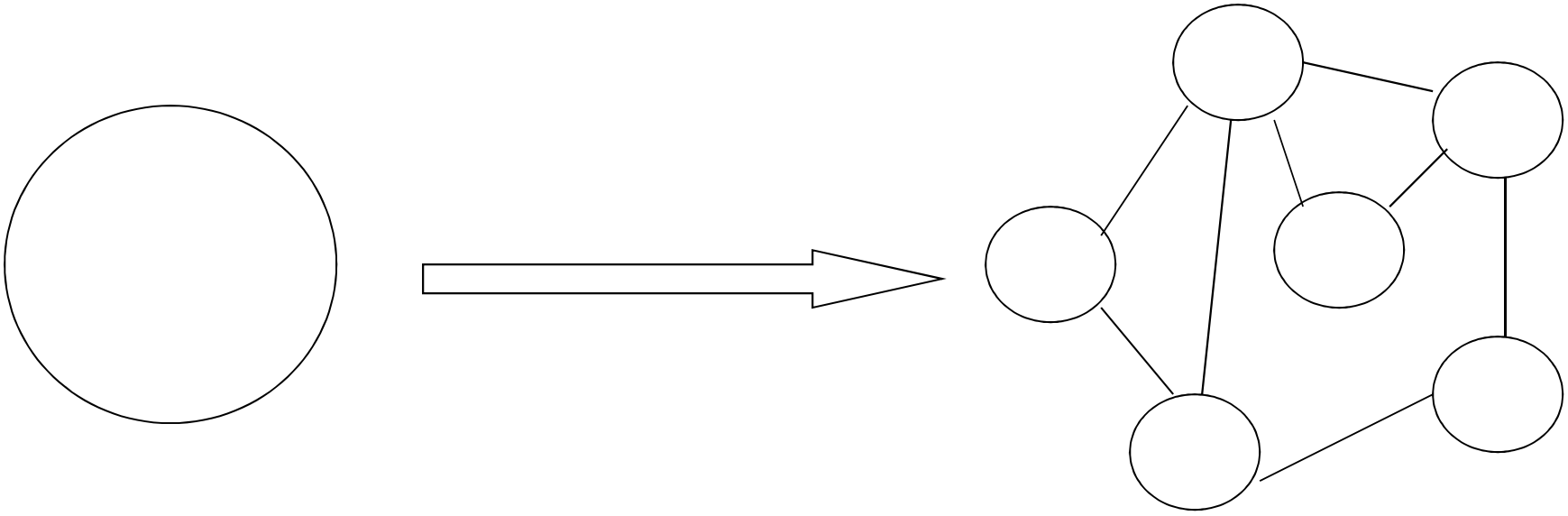


Concepts de base

- Abstraction
- Encapsulation
- Modularité
- Types Abstrait de données

Modularité

Module : Brique de base à l'échelle architecturale



Système

Modules

Modularité

- Décomposition (Top down) : Réduction itérative du problème en sous problèmes de moindre complexité.
- Composition (Bottom up) : Création itérative de la solution par composition de modules.

Modularité

Autonomie

- Un module peut être appréhendé en isolation
- Induit une localité des modifications
- Compartimentalisation : Une condition anormale lors de l'exécution reste confinée au module

Modularité

Module = Interface + Implantation

- Interface
  - Point de communication obéissant à des règles de repérage
  - Décrit les rapports entre le module et le reste du système
  - Définit les responsabilités du module

- Implantation
  - Inaccessible au reste du système
  - Réalise les responsabilités définies dans l'interface
  - Encapsule les informations sensibles liés à la réalisation

Modularité

Caractéristiques de l'interface

- Abstraction des détails. Exemple : Télécommande
- Permettre l'indépendance de l'utilisateur vis à vis de la réalisation. Exemple style de conduite d'une voiture
- Fournir une représentation fidèle. Exemple : Compteur

Modularité

Caractéristiques de l'interface

Indépendance par rapport à l'implantation :

⇒La modification de l'implantation ne change ni l'interface ni ses utilisations

⇒ Promouvoir l'interchangeabilité des implantations

Modularité

Caractéristiques de l'interface

L'interface doit être suffisante

- Offrir un ensemble de services suffisants pour être exploitable

- Non suffisance $\Rightarrow$ Incitation à la violation de l'encapsulation par les utilisateurs

Modularité

Caractéristiques de l'implantation

L'implantation doit être protégée pour maintenir l'intégralité du module. Exemple : Machine à laver

$\Rightarrow$ L'encapsulation sert à prévenir les accidents pas les fraudes

Modularité

Interface # Implantation

- Une interface pourra correspondre à plusieurs implantations
- Plusieurs interfaces pourront avoir la même implantation

Modularité

Qualités d'une structuration modulaire

- Minimalité des interfaces
  - Minimiser les communications inter-modules
  - Minimiser les échanges au sein d'une communication

$\Rightarrow$ **Couplage Faible**

- Maximalité de l'unité
  - Les informations au sein d'un module ont une forte unité logique

$\Rightarrow$ **Cohésion Forte**

Modularité

Quelles informations doivent être encapsulée ?

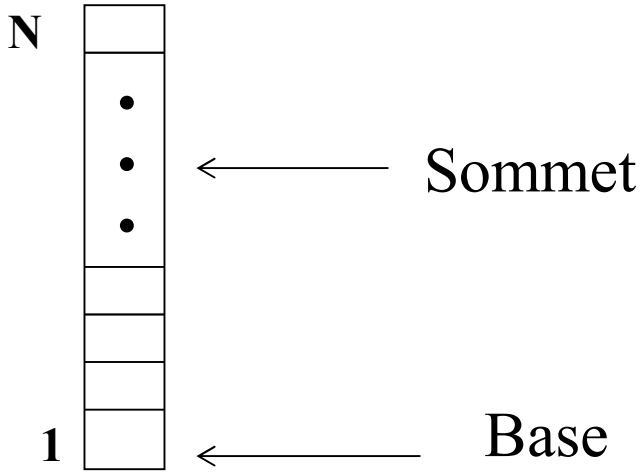
- Tous ce qui n'est pas utile au client : données et fonctions de service,…
- Tous ce qui présente un aspect sécurité : Numéro de compte,…
- Tous ce qui présente des contraintes d'intégrité qui doivent être renforcées (Numéro de téléphone,…)
- Tous ce qui est susceptible de changer dans le futur

Concepts de base

- Abstraction
- Encapsulation
- Modularité
- Types Abstrait de données

Types Abstraits de donnée

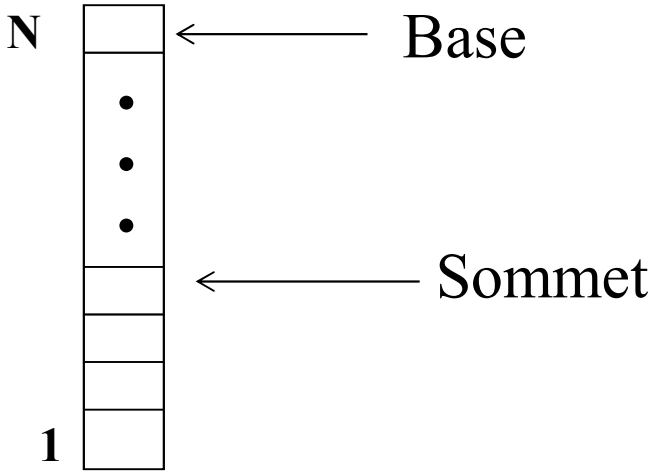
Piles : variations sur un thème




```
Empile( P : Pile; E : Elément)
Début
  Si P.Sommet < N Alors
     Sommet:=Sommet+1
     P.LesElements[Sommet]:=E
  sinon Erreur
Fin
```

Types Abstraits de donnée

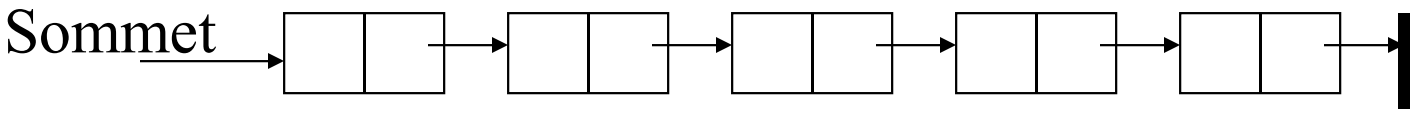
Piles : variations sur un thème




```
Empile( P : Pile; E : Elément)
Début
  Si P.Sommet > 0 Alors
     Sommet:=Sommet-1
     P.LesElements[Sommet]:=E
  sinon Erreur
Fin
```

Types Abstraits de donnée

Piles : variations sur un thème




```
Empile( P : Pile; E : Elément)
Début
    Tmp=Créer(Elément)
    Tmp.Valeur:=E
    Tmp.Suivant=P.Sommet
    P.Sommet:=Tmp
Fin
```

Définir la pile par son implantation n'est pas approprié

Types Abstraits de donnée

- Disposer d'un mécanisme pour étendre les types existants du langage de programmation utilisé
- Les types ne sont pas défini par leurs structures mais par les opérations qu'ils offrent

Spécifications algébriques des TAD

Types

Pile[X]

Fonctions

Vide : Pile[X] $\longrightarrow$ Booleen

Créer : $\longrightarrow$ Pile[X]

Empiler : X $\times$ Pile[X] $\longrightarrow$ Pile[X]

Dépiler : Pile[X] $\not\longrightarrow$ Pile[X]

Sommet : Pile[X] $\not\longrightarrow$ X

Spécifications algébriques des TAD

Préconditions

Pré Dépiler(P : Pile[X]) = **non** Vide(P)

Pré Sommet(P : Pile[X]) = **non** Vide(P)

Axiomes

∀ x : X; P: Pile[X]

Vide(Nouveau)

non Vide(Empiler(x, P)

Sommet(Empiler(x,P)= x

Dépiler(Empiler(x,P)= P

Spécifications algébriques des TAD

Types

- Enumère les types manipulés par la définition
- Un type peut être paramétré par un autre

Fonctions

- Fonction au sens mathématique ($\not\longrightarrow$ Fonction partielle)
- 3 catégories :
  - Constructeurs
  - Accesseurs (ou Sélecteurs)
  - Transformateurs

Spécifications algébriques des TAD

`Préconditions`

Restreignent le domaine des fonctions partielles

`Axiomes`

Définissent les propriétés sémantiques des fonctions

Module # TAD

- Réalisation de TAD nécessitent :
  1. Exportations de types
  2. Définitions d'opérations sur les instances de type
  3. Encapsulation des données
  4. Création de multiples instances du type
- Modules nécessitent
  1. Définitions d'opérations sur les instances de type
  2. Encapsulation des données

$$\Rightarrow \text{TAD} \supset \text{Modules}$$

Langage Procéduraux

Pascal

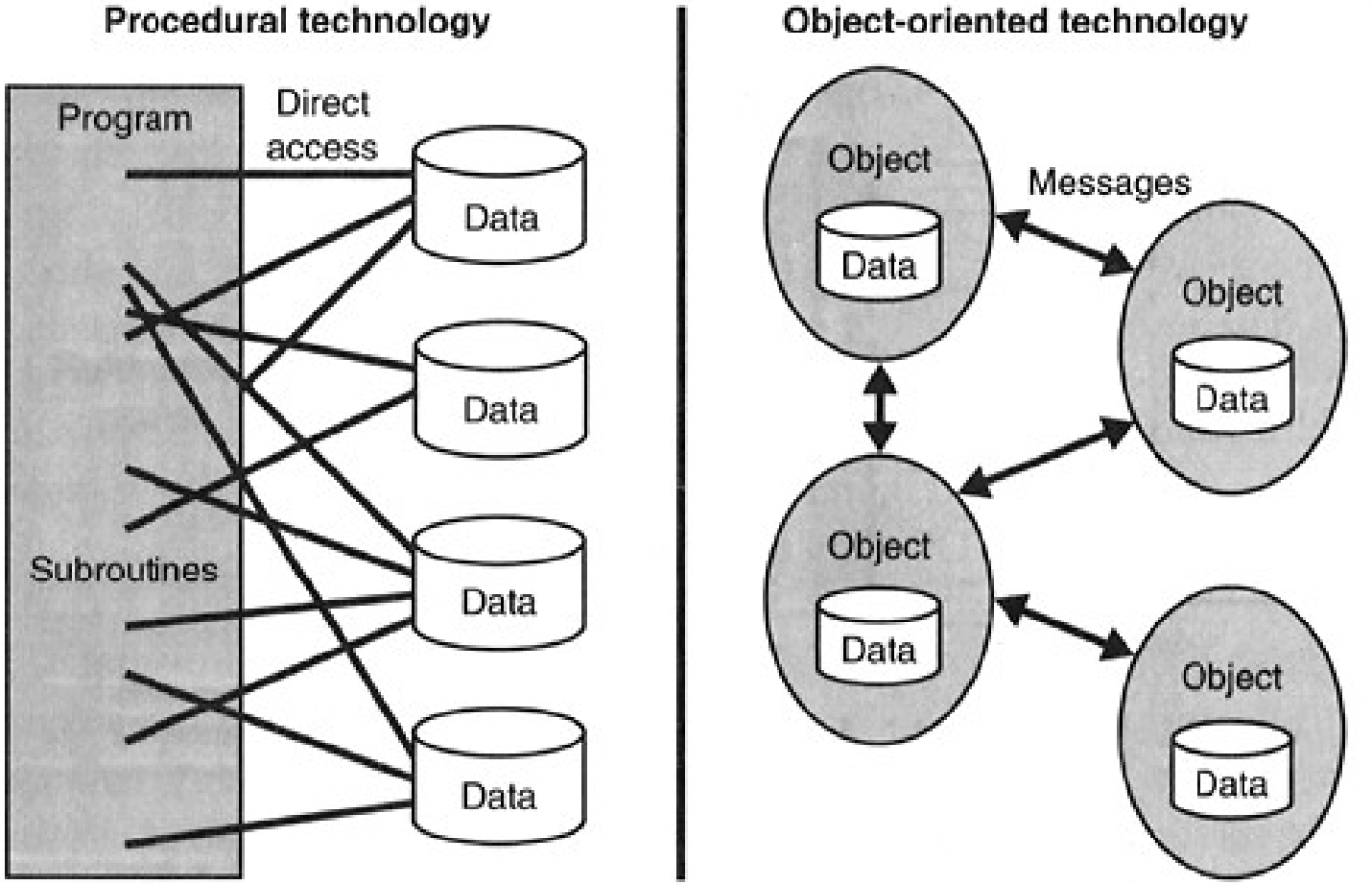
- Module = Fonction/procédure
- Unique mécanisme : Localité des déclarations
- => Détournement de l'imbrication qui supporte à l'origine l'approche de décomposition top down
- Sur utilisation de données globales
- Notion de 'unit' introduite ultérieurement

Langage Procéduraux

Langage C

- Module = Fichier
- **`static`** limite la visibilité au fichier
- => La notion de module est implicite
- => le mécanisme de masquage est rudimentaire: pas d'importation/exportation, pas de visibilité sélective

Langage Procéduraux # Paradigme Objets



Paradigme Objets

- Objets et classe
- Classification et héritage
- Polymorphisme
  - Surcharge
  - Généricité
  - Liaison dynamique

Paradigme Objets

Objets et Classes

- Transcription du réel
- Au cœur de l'abstraction et des TAD
- Unification des données et des traitements
- En relation directe avec :
  - Gérer la complexité
  - Faciliter la maintenance

Paradigme Objets

Hiérarchisation et héritage

- Idée répandue depuis le XIIX^{ème} siècle
- Classification des espèces vivantes selon le genre et l'espèce :

  Principe : factoriser des informations communes dans une catégories et introduire les spécificités dans des sous catégories
- Les sous catégories *héritent* des propriétés des catégories parentes

Paradigme Objets

Polymorphisme

- Poly = Multiple et Morphisme = Structure
- Relatif à un traitement en mesure de s'adapter aux données qu'il manipule
- Exemple
  - A := 'Bonjour'
  - B := 123
  - C := 1.23

Paradigme Objets

Surcharge :

Donner plusieurs significations à un même Symbole

- Symbole ⇔ Nom de traitement
- Symbole ⇔ Nom de variable

Paradigme Objets

Généricité

Paramétrer des traitements et des définitions de structure par des types

Liaison dynamique

Etablir le lien entre le nom d'une opération et l'algorithme de l'opération lors de l'exécution

Principaux langages Objet

- SmallTalk - Centre de recherche de Xerox - 1970
- Object Pascal - Apple 1986
- C++ - B. Stroustrup – Bell – 1986
- Eiffel – B. Meyer - 1987
- CLOS - ANSI – D. Bobrow –1988
- Java – J. Gosling - Sun Microsystems 1995
- C# - Microsoft - 2001

Objets et Classes

Objet

- Etat
- Comportement
- Identité

Objets

Etat

- Mémorise le passé influence le futur
- Caractérisation : Aspects Statiques
- Valeur des caractérisation : Aspects statiques ou dynamiques

Exemple : Chaise

Caractérisation statique

- Nombre de pieds : 4
- Matière : ‘Bois’
- Couleur : ‘Marron’

Valeurs de caractérisation statiques

- Position_X : 1,5
- Position_Y : 1,5

Valeurs de caractérisation dynamique

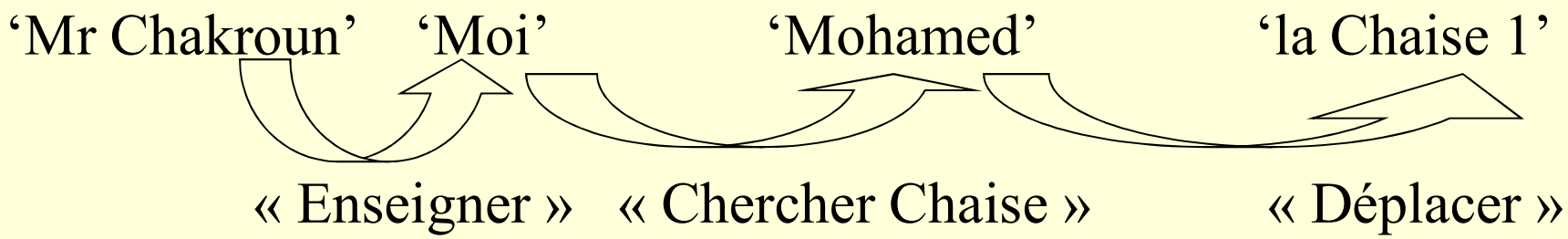
- Nombre Total : 10

valeur commune

Objets

Comportement

- Un objet agit sur d'autres objets en leur demandant des services
- Un objet réagit a une demande de service par un comportement spécifique
- Communication par message



Objets

Comportement

- Interroge l'état pour déterminer les demandes admissibles
- Agit sur l'état
- Publie l'état
- L'état mémorise les conséquences cumulées du comportement
- Peut être indépendant de l'état. Exemple : Incrémenter le nombre total de chaises

Objets

Communication par message

- Spécification de l'objet destinataire
- Spécification du service requis
- Spécification des informations transférées dans les deux sens de la communication

Identité

Etat :
Nom : ‘Ben Salah’
Prénom : ‘Ali’
Date de Naissance : 5/2/48
Comportement
Etudier
SeMarier
travailler
Mourir

≠

Etat :
Nom : ‘Ben Salah’
Prénom : ‘Ali’
Date de Naissance : 5/2/48
Comportement
Etudier
SeMarier
travailler
Mourir

- Identité permet de distinguer systématiquement deux objets
- Est souvent confondu avec l’adresse d’un objet dans les langages de programmation
- Oracle : identité d’un objet = OID (Proche de la notion de clé)

Terminologie

Etat	Opération
Propriété	Message
Attribut	Méthode
Donnée Membre	Fonction membre
	Service

Protocole : ensemble de services

Classe

- Abstraction qui permet de « condenser » un ensemble d'objets ayant des caractéristiques communes en un seul concept
- Abstraction relatives à la caractérisation d'un ensemble d'objets
  - Fixer des informations de caractérisations communes (Etat/Comportement)
  - Omettre les valeurs de caractérisation

Classe

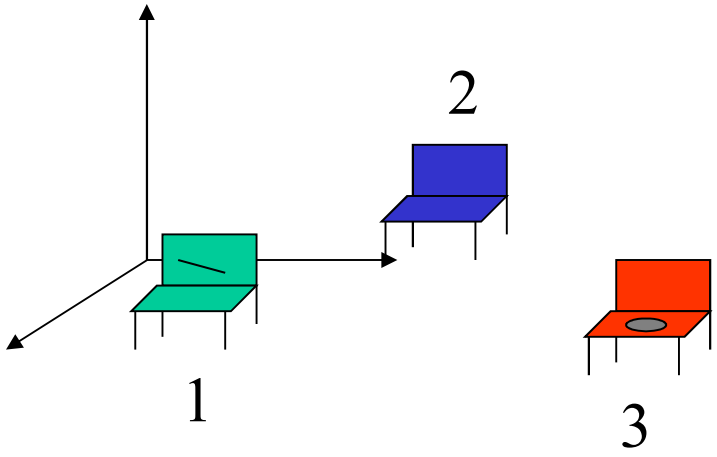
Caractéristiques :

Le fait d'avoir les mêmes caractéristiques était le fruit du hasard

Nom : Tom Cruise
Couleur des yeux : Noir
Atomes de carbones : $9.987\ 10^{12}$
Nombre de poils : 10 752 634 434
Taille : 1.70
Globules rouges : 7 840 656
Groupe sanguin : A +
Sexe : Masculin
Globules blancs : 9 540 786
Date de naissance : 3 Juillet 1962

Nom : Cindy Crawford
Couleur des yeux : Marron
Atomes de carbones : $9.684\ 10^{12}$
Nombre de poils : 11 233 122 110
Taille : 1.76
Globules rouges : 7 541 553
Groupe sanguin : B+
Sexe : Féminin
Globules blancs : 9 760 956
Date de naissance : 20 Février 1966

Classe



Chaise 1

Etat
- Nombre de pieds : 4
- Matière : 'Bois
- Couleur : 'Vert'
- Position_X : 1,5
- Position_Y : 1,5

Comportement
- Déplacer

Chaise 2

Etat
- Nombre de pieds : 4
- Matière : 'Bois
- Couleur : 'Bleu'
- Position_X : 3,5
- Position_Y : 2,5

Comportement
- Déplacer

Première abstraction : Identifier un ensemble de caractéristiques communes à une population d'objets

Classe

Chaise 1

Etat
- Nombre de pieds : 4
- Matière : ‘Bois
- Couleur :‘Vert’
- Position_X : 1,5
- Position_Y : 1,5

Comportement
- Déplacer

Chaise 2

Etat
- Nombre de pieds : 4
- Matière : ‘Bois
- Couleur :‘Bleu’
- Position_X : 3,5
- Position_Y : 2,5

Comportement
- Déplacer

Classe Chaise

Etat
- Nombre de pieds
- Matière
- Couleur
- Position_X
- Position_Y

Comportement
- Déplacer

Deuxième abstraction : Omettre les valeurs de caractérisation

Classe

- Objet instance de classe
- Classe moule (usine) à objets
- Classe = Interface + Implantation
- Module = classe ou ensemble de classes

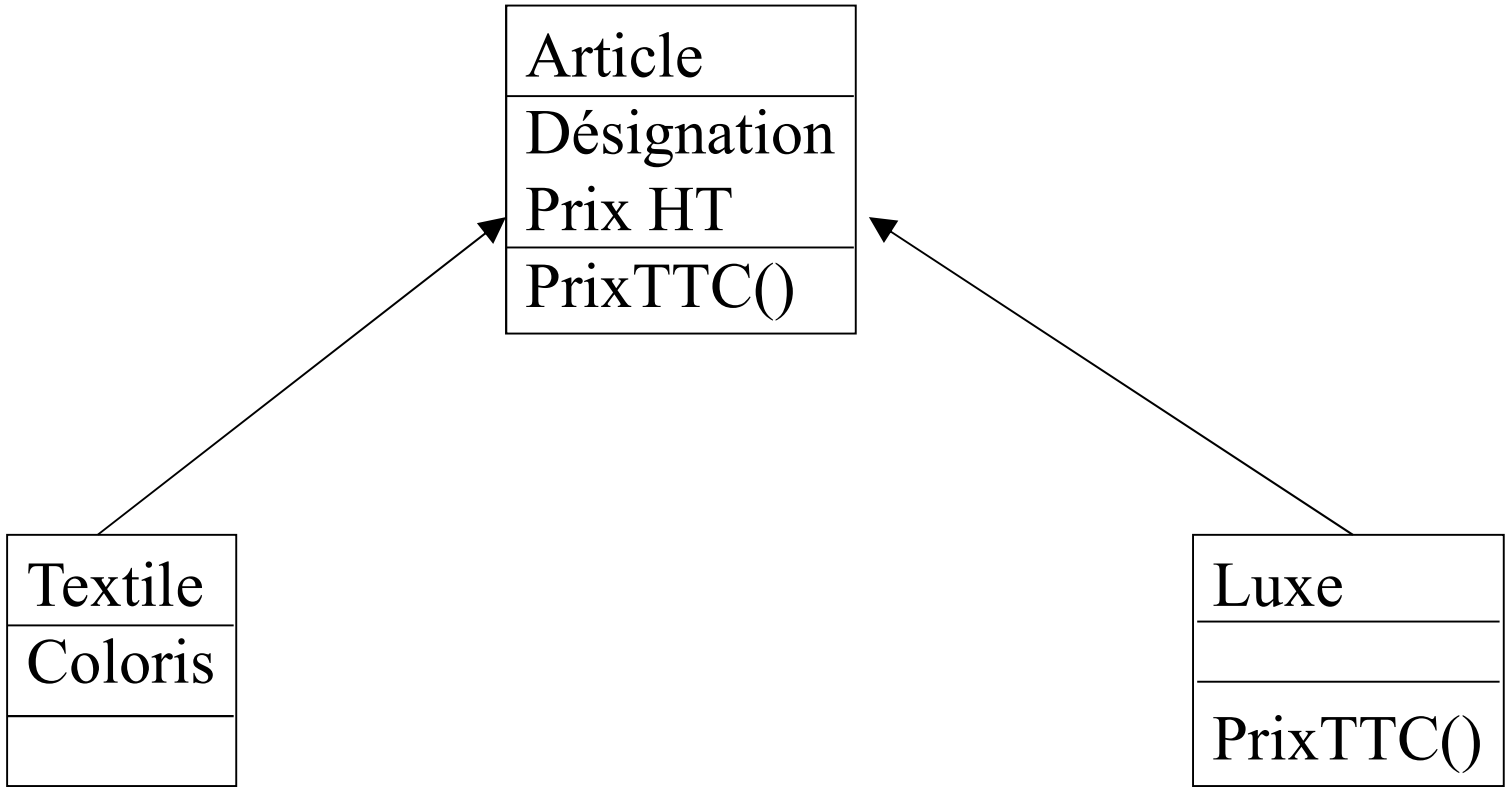
Généricité

- Deux phases :
  - Instanciation
  - Exploitation

Classification et héritage

- 3 Classes Articles/Textiles/Alimentaires
- Redondance
- Textiles et Alimentaires sont des *articles*
- Différentes solutions en particulier dans Textile un article

Classification et héritage



Héritage et liaison dynamique

- Principe de l'ouverture/fermeture

Les langage C++ et Java

Adel KHALFALLAH
Institut Supérieur D'informatique – El Manar
Adel.Khalfallah@fst.rnu.tn
Adel_Khalfallah@yahoo.fr

Objet et classe

- Une classe est un type utilisateur
- Pour C++ la définition d'une classe peut être dupliquée plusieurs fois (notamment via **`#include`**) sans violer la règle de définition unique
- Pour java la machine virtuelle charge la classe

Objets et classes en C++

```
const int MAX = 50
class Pile
{
public:
   BOOL EstPleine();
   BOOL EstVide();
   int Sommet();
   void Depiler();
   void Empiler(int Elt);
private:
   int LesElements[MAX];
   int Tete;
};
```

Sélecteurs

Objets et classes en C++

```
void Pile :: Empiler(int Elt){
  if (!EstPleine()) LesElements[++Tete]=Elt;
}

int Pile :: Depiler(){
  if (!EstVide()) Tete--;
}

BOOL Pile :: EstVide(){
  return Tete==-1;
}
```

Objets et classes en C++

```
BOOL Pile::EstPleine(){
   return Tete==MAX;
}

int Pile::Sommet(){
   if (!EstVide()) return LesElements[Tete];
}
```

Objets et classes en C++

```
#include Pile.h
int main(){
   int x;
   Pile P1;
   P1.Empiler(10);
   Pile LesPiles[20];
   LesPiles[15].Empiler(2);
   Pile *Ptr_P1,*Ptr_P;
   Ptr_P1=&P1;
   Ptr_P=new Pile;
   Ptr_P->Empiler(234);
   Pile &Ref_P1=P1;
   Ref_P1.Empiler(678);
   x=P1.Sommet(); // x ⇒ 678
```

```
   delete Ptr_P;
   Ptr_P = new Pile[10];
   delete [] Ptr_P;
}
```

Durées de vie

```
{
…
  //Aucun Objet n'existe
    {TClasse O; //O existe
     TClasse *Ptr;
    …
     Ptr=new TClasse //L'objet pointé par Ptr Existe
    …
    } //O n'existe plus; l'objet pointé par Ptr existe encore
…
delete « l'objet pointé par Ptr » //Aucun Objet n'existe
}
```

Objets et classes en java

```
package main.java;

public class Pile {
   //Sélecteurs
  public boolean EstPleine(){
     return Tete==MAX;
   }
   public boolean EstVide(){
     return Tete==-1;
   }
   public int Sommet(){
     if (!EstVide())
       return
   LesElements[Tete];
     else return -123;
   }
   public void Depiler(){
     if (!EstVide()) Tete--;
   }
   public void Empiler(int
                          Elt)
   {
     if (!EstPleine())
         LesElements[++Tete]
                          =Elt;
   }
   private final static int
                       MAX=10;
   private int[]
   LesElements=new int[MAX];
   private int Tete=-1;
}
```

Objets et classes en java

```
package main.java;
public class Main {
public static void main(String[] args) {
 Pile P1=new Pile();
 P1.Empiler(10);
 int i=0;
 for (Pile P : LesPiles){LesPiles[i++]=P=new Pile();
   P.Empiler(i);}
 for (Pile P:LesPiles)System.out.println(P.Sommet());
 }
  LesPiles[15].Empiler(123);
  System.out.println(LesPiles[15].Sommet());
}
```

Durée de vie (java)

- Un objet existe tant qu'il est référencé
- Le garbage collector (ramasse-miette) se charge de libérer les objets non référencés, son déclenchement est – généralement – imprévisible

Modes de passage de paramètres

- C++ : par valeur et par référence et paramètres de type pointeurs

- Java : uniquement par valeur, pour les objets la référence est passée par valeur, l'objet référencé est passé par référence

Règles d'accès C++

- C++ introduit 3 niveaux de visibilité :
  - La classe (**`private`**)
  - La classe et ses descendants (**`protected`**)
  - Tout: la classe, les autres classe et traitements (**`public`**)

- C ++ introduit un mécanisme d'exception : une classe peut offrir la visibilité sur ses éléments encapsulés (**`friend`**)

Règles d'accès C++

```
class Laclasse
{
public:
   …
   friend F1();   // La fonction F1() verra les parties privées et
                  // protégées de LaClasse
   friend UneClasse.M1();// La methode M1() de UneClasse verra les
                         // parties privées et protégées de LaClasse
   friend  UneClasse; // toutes les methodes de UneClasse verront les
                      // parties privées et protégées de LaClasse
   private:
…
}
```

friend peut apparaître de façon identique dans la partie privée ou publique

`struct` versus `class` (C++)

`struct` équivalent à `class` sauf que

- `struct` par défaut publique
- `class` par défaut privé

Règles d'accès java

- Java introduit 4 niveaux de visibilité :
  - La classe (**`private`**)
  - Le paquetage (**par défaut**) s'applique à toute la classe ou à un élément de l'état ou du comportement qui sera accessible à toutes les classes du paquetage
  - La classe et ses descendants et le paquetage (**`protected`**)
  - Tout (**`public`**) la classe, les autres classe et paquetages

Règles d'accès

L'encapsulation fait que de préférence :

```
class Toto {
public:
   // méthodes du comportement uniquement
private:
  //Informations sur l'état et éventuellement
    fonctions de service
}
```

Règles d'accès

Getters et Setters

```
class Toto {
public:
    int GetX()  //Lecture contrôlée de X
  void SetX(int Val) //Ecriture contrôlée de X
private:
   int X;
}
```

C++ Attention aux brèches

- Elément de l'état qui est une référence
- Elément de l'état qui est un pointeur et qui adopte les valeurs
- Getter qui retourne un pointeur sur un élément de l'état
- Un Getter qui retourne une référence à un élément de l'état

this

```
BOOL Pile::EstPleine(){
   return Tete==MAX;
}
```

- Quelle valeur pour Tete ?

⇒ Celle de l'objet sur lequel à été invoqué la méthode (P1);

On a implicitement

```
BOOL Pile::EstPleine(Pile *this){
   return this->Tete==MAX;
}
```

Attention à la tentation d'écrire

~~`BOOL EstPleine(Pile P) {return P.Tete==MAX;}`~~

Préciser la portée

C++

```
class Toto
{
public:
   void M_Publique();
private:
   int   Donnée;  //1
};
void Toto::M_Publique(){
int Donnée; //2
                //Cache 1
Donnée=2; //accès à 2
Toto::Donnée=1 // accès à 1}
```

java

```
class Toto
{
Public void M_Publique(){
 int Donnée; //2
                //Cache 1
   Donnée=2; //accès à 2
 Toto.this.Donnée=1 // accès à 1
}
Private int   Donnée;  //1
};
```

`const` dans les paramètres (C++)

```
Fonction_Ou_Methode(const UnType &X)
```

- Efficacité : seule une addresse est passée en paramètre
- Fiabilité : une tentative de modification accidentelle de X sera détectée à la compilation

`const` sélecteur (C++)

```
class Pile
{
public:
   BOOL EstPleine() const;
   BOOL EstVide();
…
};

main() {
const Pile P;
P.EstPleine() //Ok
P.EstVide() //Erreur
}
```

`const` sélecteur (C++)

- Etends le domaine d'application du sélecteur
- Le compilateur vérifie que le sélecteur ne tente pas de modifier l'état

Attribut **`final`** (java)

- NB : **`const`** est un mot clé de java qui n'est pas utilisé
- Un attribut **`final`** devra être initialisée au plus tard lors de la construction
- Un attribut de type primitif **`final`** est une constante
- Un attribut référence à un objet ou un tableau ne pourra pas être réaffecté mais l'objet référencé pourra être modifié

```
public class Voiture {
…
Public M2() {… MonMoteur.AugmenterPuissance();… //Ok}
Public M1() {… MonMoteur= new Moteur();… //Erreur}
private final int NoChassis;
private final Moteur MonMoteur;
…
```

Paramètre/locale **`final`** (java)

- Un paramètre (ou une variable locale) de type primitif **`final`** ne pourra pas être modifié dans le corps du traitement
- Un paramètre (ou une variable locale) référence à un objet ou un tableau ne pourra pas être réaffecté mais l'objet référencé pourra changer dans le corps du traitement

```
public void Proc(final int I, final Voiture V){
  final int tmp;
  I=123; //Erreur
  tmp=456 //Ok
  tmp=789 //Erreur
  V.Accelerer() //Ok
}
```

Objets immuables (immutable)

- Objets dont l'état ne change plus après avoir été crées
- Ils matérialisent des valeurs
- Ils garantissent un comportement cohérent dans un contexte multitâches sans nécessiter de synchronisation

Objets immuables

- C++
  - Tous les attributs et toutes les méthodes sont constantes
  - L'affectation est interdite
- Java
  - Tous les attributs sont final
  - Il ne faut pas fournir de méthodes qui changent l'état
  - Il faut interdire l'héritage (afin d'éviter d'introduire de méthodes qui changent l'état)
  - String est un exemple de classe dont les objets sont immuables

Objets immuables

- Que faire lorsqu'un objet immuable doit changer d'état. Par exemple, on veut rajouter un caractère dans un objet String

=> On crée un nouvel objet contenant le nouvel état

Données mutables (C++)

- Une méthode peut être constante d'un point de vue logique mais nécessiter quand même la modification de l'état

```
class Data {
public:
   TData GetData() const;
...
private:
   TData TheData;
   bool CacheValide;
   TCache Cache;
   TCache CalculCache() const;
}
TData Data::GetData() const {
   if (!CacheValide) Cache=CalculCache();// Problème !
   ...
}
```

Données mutables (C++)

- Première solution : utiliser le forçage

```
Data* PData = const_cast<Data*>(this);
```

dans **`GetData()`**. L'appel de la méthode sur un objet constant sera alors indéfini.

- Deuxième solution : utiliser les données mutables

```
class Data {
public:
   TData GetData() const;
...
private:
   TData TheData;
   mutable bool CacheValide;// CacheValide et cache ne pourront
   mutable TCache Cache;      //  jamais être constants
   TCache CalculCache() const;
}
```

Attributs et méthodes de classe

Chaise

- Nombre Total
- Incrémenter Total

```
class Chaise {
public:
   static void Incrementer()
      …
private:
   static int NombreTotal;
…
}
```

Attributs et méthodes de classe

- **`static int NbrTotal`** pour C++ est une déclaration, la définition correspondante doit apparaître au niveau de l'implantation, elle pourra contenir une initialisation :

```
int chaise::NbrTotal=0
```

- NombreTotal est partagé par tous les objets chaise (il est global à la classe)
- Il n'est pas nécessaire de se référer à un objet chaise pour appeler Incrementer ou acceder à NombreTotal (la où il est visible) on peut ecrire:

C++	java
`Chaise::Incrementer()`	`Chaise.Incrementer`
`Chaise::NombreTotal`	`Chaise.NombreTotal`

Attributs et méthodes de classe

- **`Incrementer()`** n'a pas visibilité sur les attributs non statique (qu'ils soient privés ou publique), il ne peut manipuler que les attributs statiques
- **`Incrementer()`** ne dispose pas du pointeur **`this`**
- **`Incrementer()`** s'applique sur les objets chaise constants (C++)

Constructeurs

Problème: initialisation de la tête de la pile ?

- Premier tentative

```
…
private:
  int Tete=-1; //Erreur de syntaxe (C++) Ok java
…
```

- Deuxième tentative

  Un setter : **`void InitTete() {Tete=-1;}`**

  Peut-on faire confiance au client ? Possibilité d'oublis ou d'initialisations multiples

Constructeurs

```
class Pile  {
public:
        Pile();
   bool EstPleine();
   bool EstVide();
   int Sommet();
   void Depiler();
   void Empiler(int Elt);
private:
   int LesElements[MAX];
   int Tete;
};
```

```
class Pile  {
public
  public Pile(){…}
   public bool EstPleine()…
   public bool EstVide()…
   public int Sommet()…
   public void Depiler()…
   public void Empiler(int
                        Elt)…
   private
      int[] LesElements;
   private int Tete;
};
```

Constructeurs

Constructeur :

- Même nom que la classe
- Pas de type résultat
- Invoqué automatiquement
- Constructeur par défaut
- Autant de constructeurs que de façons d'initialiser (les paramètres doivent permettre de les distinguer)

Constructeur (C++)

Constructeur implanté comme une méthode

```
class Pile   {
public:
        Pile();
…
}

Pile::Pile() {Tete=-1;}
```

Constructeur (C++)

Constructeur inline : première forme

```
class Pile   {
public:
        Pile();
…
}

inline Pile::Pile() {Tete=-1;}
```

Constructeur

Constructeur inline : deuxième forme

```
class Pile   {
public:
        Pile() {Tete=-1;} ;
…
}
```

Constructeur

Liste d'initialisation

```
class Pile   {
public:
        Pile() : Tete(-1) {} ;
…
}
```

Nécessaire lorsqu'un des champs de la classe est une référence ou est constant

Constructeur

- Passage de paramètres à la construction : Exemple, Pile dont la taille maximum est fixée à la construction (C++ et java)

```
class Pile  {
public:
  Pile(int Max):Tete(-1),Elts(new int[LeMax=Max]) {}
...
private:
   int Tete,LeMax;
   int *Elts
};

main(){
   Pile P(10);
}
```

Constructeur

- Un classe peut disposer de plusieurs constructeurs *surchargés*

```
class Date {
  Date(int UnJour, int UnMois, int UnAn);
  // La date initiale est fournie sous la forme jour / mois / An. Par exemple :
  // Date  D(1,1,2004);
  Date(const char* UneDate);
  // La Date est fournie sous forme de chaîne de caractère. Par exemple :
  // Date  D("Jeudi 1er Janvier 2004")
...
}
```

Constructeur par défaut

- Constructeur sans paramètres
- constructeur avec paramètres ayant tous des valeurs par défaut (C++)
- Lorsqu'une classe ne possède aucun constructeur, le compilateur lui en génère un automatiquement
- Ce constructeur est particulièrement utile pour l'initialisation de tableaux d'objets, d'objets construits à partir d'autres objets, etc...(C++)
- Les classes ayant des champs constants ou références ou objet sans constructeur par défaut ne peuvent pas être construits par défaut (C++).

Constructeur par copie (C++)

- Dans différents cas de figure un objet doit être construit à partir d'un autre, par exemple :
  - Lors d'une déclaration

```
Pile P1;
Pile P2(P1);// ou Pile P2=P1;
```

  - Lors d'un appel ou d'un retour par valeur

```
Pile F(Pile P) {
  Pile Res;
  ...
  return Res;
}
main() {
      Pile P1,P2;     P2=F(P1);
}
```

Constructeur par copie (C++)

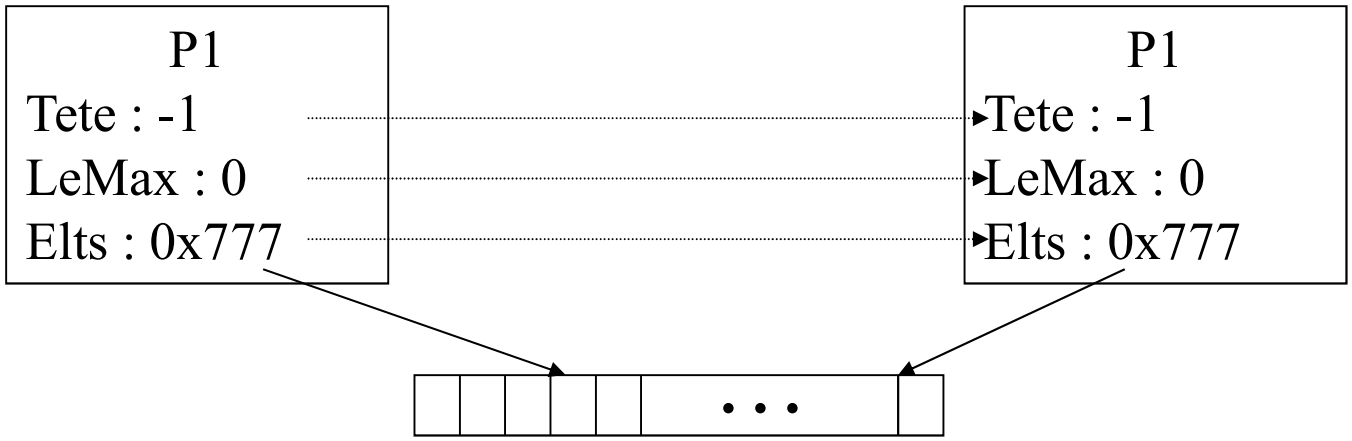
- Chaque classe dispose d'un constructeur par copie généré par le compilateur. Celui-ci *copie* les membres de la classe un à un.

- Même lorsqu'un des champs est une constante ou une référence la classe dispose d'un constructeur par copie généré par le compilateur

Constructeur par copie (C++)

- Dans certains cas les comportement du constructeur par copie généré par le compilateur n'est pas acceptable. Exemple copie de Pile

```
main(){
   Pile *P1=new Pile(20);
   Pile P2(P1);
}
```



Constructeur par copie (C++)

- Lorsque le constructeur par copie généré par le compilateur n'est pas satisfaisant, il est possible d'introduire un constructeur par copie spécifique

```
class Pile {
public:
  Pile(int Max):Tete(-1),Elts(new int[LeMax=Max]) {}
  Pile(const Pile &P);
...
}
Pile:: Pile(const Pile &P){
  Tete=P.Tete;
  Elts=new int[Max=P.Max];
  for(int i=0;i<=Tete;i++) Elts[i]=P.Elts[i];
}
```

Affectation (C++)

- Chaque classe dispose d'un opérateur d'affectation généré par le compilateur
- Lorsqu'un des champs est une constante ou une référence la classe ne dispose plus d'un opérateur d'affectation généré par le compilateur
- Le problème de la construction par copie dans le cas de classes gérant dynamiquement de la mémoire se pose de façon similaire pour l'affectation

Destructeur (C++)

- Le constructeur permet de garantir que certaines initialisations sont effectuées (comme la réservation de ressources) avant la création de tout objet.
- Le destructeur permet de garantir qu'un certaine fonction est réalisée (comme la libération de ressource) avant la destruction d'un objet.
- En règle générale, un constructeur n'est jamais appelé explicitement, il est appelé implicitement lorsqu'un objet va être détruit soit parce que l'exécution quitte le bloc dans lequel il est défini soit suite à un appel à **`delete`**

Destructeur (C++)

```
class Pile {
public:
   Pile(int LaTaille);
   ~Pile();
   ...
private:
   int Max;
   int Tete;
   int *LesElements;
}
Pile::Pile(int LaTaille) {Tete=-1; Max=LaTaille;
   LesElements=new int[Max];
}
Pile::~Pile() {delete [] LesElements;}
```

Construction et destruction (C++)

Contexte	Appel du constructeur	Appel du destructeur
Objet local	− L'exécution arrive sur la déclaration − Dans l'ordre de déclaration	− L'exécution quitte le bloc de déclaration − Dans l'ordre inverse de la déclaration
Objet dynamique	L'exécution arrive sur **`new`**	L'exécution arrive sur **`delete`**
Objet donnée membre	− L'exécution arrive sur la déclaration de l'objet parent et ***avant*** l'appel au constructeur de celui-ci − Dans l'ordre de déclaration des données membres	− L'exécution arrive sur la destruction de l'objet parent ***après*** l'appel au destructeur de celui-ci − Dans l'ordre inverse de la déclaration

Construction et destruction (C++)

Contexte	Appel du constructeur	Appel du destructeur
Tableau d'objets	– L'exécution arrive sur la création du tableau – L'ordre est indéfini	– Le Tableau va être détruit – L'ordre est indéfini
Objet Statique	L'exécution arrive sur la déclaration *pour la première fois*	A la fin du programme Dans l'ordre inverse de la construction
Objet global à un namespace ou au programme et variables de classe	– Avant l'appel à **`main()`** – Dans l'ordre de leurs *définitions,* indéfini pour des unités différentes **`class x {`** **`static int z;}`** //déclaration et non définition **`int y;`** //1 **`int x::z;`**//2	– Après la fin de **`main()`** – Dans l'ordre inverse de la construction, indéfini pour des unités différentes

Construction et destruction (C++)

Contexte	Appel du constructeur	Appel du destructeur
Objet temporaire (passage et retour par valeur, valeur par défaut)	− Le compilateur en a besoin − L'ordre est indéfini	− Le compilateur n'en a plus besoin − L'ordre est indéfini

Initialisations (C++)

- Un champ intégral statique et constant peut avoir une initialisation explicite

```
class XXX {
private:
  static const int xxx=24;
}
```

- Les énumérations peuvent aussi avoir une initialisation explicite

```
class XXX {
public:
  enum {MIN=0,MAX=100};
...
}
```

Bloc d'initialisations (java)

- Les blocs d'initialisation permettent de factoriser le code commun à tous les constructeurs (ils sont donc exécutés avant les constructeurs), lorsqu'ils sont statique ils ne sont exécutés qu'une seule fois lors de la création du 1er objet ou lors de 1er accès à un champ statique i.e. lorsque la classe est chargée

```
class Image {
 private Bit[][] Bitmap=new Bit[N][M];
 { for(int i=0;i<N; i++)
     for(int i=0;i<M; i++) Bitmap[i][jy]=…
 }
 static {…}
 Image(){…}
 Image(format f){…}
}
```

Espace de nommage

```
namespace Nom {
... //définition des ressources
}
```

- Un namespace est une construction offerte par le langage pour matérialiser un module.
- Un namespace permet de déclarer un ensemble de ressources (types, variables, fonctions,…).
- Un namespace n'impose pas de séparation entre une interface et une implantation, toutefois c'est une bonne habitude de maintenir une telle séparation.

Espace de nommage

- Deux ressources distinctes peuvent porter le même nom lorsqu'elles sont dans des namespace différents.
- Pour utiliser une ressource d'un namespace donné dans un autre, il suffit de la *qualifier:*
  `NomNamespace::NomRessource`
- Un namespace peut-être défini soit dans une zone globale soit imbriqué dans un autre namespace
- Un namespace peut être éclaté en plusieurs parties, dans un même fichier ou sur plusieurs fichiers. Il peut donc être défini de façon incrémentale

Espace de nommage

- Un namespace peut être anonyme, ses ressources ne sont accessibles que dans l'unité où il est défini. Ceci permet d'éviter des collisions de noms.
- Les namespaces des arguments d'une fonction sont recherchés implicitement lors de la manipulation de ces arguments. Les régles de résolution habituelles (Surcharge) sont appliquées.

```
namespace Y {
  class XXX;
  T F(XXX x);
}
G(Y::XXX x,int i){
   F(x);  // Ok !
   F(i); //Erreur !
}
```

Espace de nommage

- Le nom d'un namespace peut avoir un alias. Utile lorsque le nom est trop long ou lorsque on manipule des versions de bibliothèques.

```
namespace Poste_Telegraphes_Telephone {
…
}
namespace PTT = Poste_Telegraphes_Telephone;

namespace Lib = Bibliothque_Graphique_V2R11
//Changer de bibliothèque revient à changer la ligne précédente.
Lib::CFenetre F;
...
```

Espace de nommage

```
namespace AnaLex {
enum Token {
  VAR, CHIFFRE, FIN, PLUS='+', MOINS='-' MUL='*',
  DIV='/', ParG='(', ParD=')'
};
Token Courant;
Token Lire();
double Valeur;
string Expression;
class expression{...};
}
Token AnaLex::Lire(){
...//Implantation de lire
}
```

Espace de nommage

- Pour une ressource d'un espace de nommage donné, utilisé plusieurs fois dans un autre espace de nommage, on peut éviter la qualification par une déclaration d'utilisation

```
Calculatrice::Evaluer(){
using AnaLex::Lire;  // Toutes les surcharges de Lire deviennent alors
                                                  // visibles
using AnaLex::Expression;
...
AnaLex::Token t;
t=Lire();
...
}
```

- `using namespace NomNamespace` rend visible toutes les ressources du namespace.

Espace de nommage

```
namespace MaBib {
using namespace Bib1;
using namespace Bib2;
// Bib1 et Bib2 contiennent toutes les deux une ressource nommée R
using Bib1::R
// Résout la collision de nom au profit de Bib1::R
...
}
```

Le Polymorphisme

Adel KHALFALLAH
Institut Supérieur d'Informatique – El Manar
Adel.Khalfallah@fst.rnu.tn
Adel_Khalfallah@yahoo.fr

C++ Surcharge

Echange de réels

```
void EchangeR(float &x, float &y) {
    float tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

Echange d'entiers

```
void EchangeI(int &x, int &y) {
    float tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

```
void main() {
int i,j;
…
EchangeI(i,j);
float u,v;
…
EchangeR(u,v)
…
}
```

Pas très pratique et peu polymorphe

C++ Surcharge

La surcharge consiste à donner des sémantiques différentes à un même symbole

- Donner le même nom de fonction/procédure pour des implantations (i.e. algorithmes) différents.
- Etendre les opérateurs habituels à de nouveaux types

C++ Surcharge

Echange de réels

```
void Echange(float &x, float &y) {
    float tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

Echange d'entiers

```
void Echange(int &x, int &y) {
    float tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

```
void main() {
int i,j;
…
Echange(i,j);
float u,v;
…
Echange(u,v)
…
}
```

Surcharge

- Correspond à un polymorphisme « de façade »
- Particulièrement utile lorsque le symbole surchargé a une « forte identité visuelle »
- Evite la prolifération des noms

C++ Surcharge

```
F(long,long){printf("1\n");}
F(float,float){printf(" 2\n");}
main(){
   int i,j;
   float x,y;
   long u,v;
   double r,s;
   F(i,j);//1
   F(x,y);//2
   F(u,v);//1
   F(r,s);//Ambigu (double,double) → (long,long)  ou (double,double) → (float,float)
   F(i,x);//Ambigu (int,float) →(long,long) ou (int,float) →(float,float)
}
```

Le cast explicite permet de lever les ambiguités: **`F(float(r),s)`** → 2

`F(i,int(x))` → 1

C++ Surcharge

- Promotions : Passer d'un « sous-type » au type

  `short` → `int` → `long`

  `float` → `double` → `long double`

- Conversions : Peuvent introduire une perte d'informations

  `int` → `double`

  `double` → `int`

- Conversions utilisateurs : un classe pourra surcharger des opérateurs de conversions

  Complexe → Point

Surcharge

Règles de recherche de correspondance :

1. Rechercher une correspondance exacte

   0 : Passer à la règle suivant ; 1 : La correspondance est établie ; > 1 : ambiguïté

2. Rechercher une correspondance avec promotion:

   0 : Passer à la règle suivant ; 1 : La correspondance est établie ; > 1 : ambiguïté

3. Rechercher une correspondance avec conversion

   0 : Passer à la règle suivant ; 1 : La correspondance est établie ; > 1 : ambiguïté

4. Rechercher une correspondance avec conversion utilisateur

   0 : Appel non résolu ; 1 : La correspondance est établie ; > 1 ambiguïté

Surcharge

- Le type de résultat ne rentre pas en ligne de compte
- Surcharge et paramètres par défaut peuvent être équivalents (c'est aussi une source d'ambiguïté)

`F(X,Y=10)` {Un Traitement}

⇕

`F(X,Y)` {Le même traitement}

```
F(X) {F(X,10);}
```

C++ Surcharge

- Un symbole peut-être surchargé à travers des espaces de nommage différents.

```
namespace Bib1 {
void F(int);
...
}
namespace Bib2 {
void F(int);
...
}
namespace MaBib {
using namespace Bib1
using namespace Bib2  //2 versions de F sont visibles
```

C++ Surcharge des opérateurs

- Mécanisme permettant d'étendre les opérateurs aux types utilisateurs

```
class Complexe {
public:
 Complexe(float UnRe, float UnIm):Re(UnRe),Im(UnIm){}
 Complexe operator +(Complexe z);
 Complexe operator +(float x);
 Complexe operator *(Complexe z);
 Complexe operator *(float x)
private:
 float Re, Im;
}
```

- Peut-on écrire **`2+z`** ?

C++ Surcharge des opérateurs

- Un opérateur peut être surchargé par une fonction

```
class Complexe {
public:
 Complexe(float UnRe, float UnIm):Re(UnRe),Im(UnIm){}
 float GetRe();
 void SetRe(float);
 float GetIm()
 void SetIm(float)// Ce qui suit n'est pas obligatoire
 friend Complexe operator+(Complexe z1,Complexe z2);
 friend Complexe operator+(float x,Complexe z);
 friend Complexe operator*(Complexe z1,Complexe z2);
 friend Complexe operator*(float x, Complexe z)
private:
 float Re, Im;
}
```

C++ Surcharge des opérateurs

```
Complexe operator +(Complexe z1, Complexe z2){
  Complexe Res;
  Res.Re=z1.Re+z2.Re;
  Res.Im=z1.Im+z2.Im;
  return Res;
}
Complexe operator +(float x, Complexe z) {...}
Complexe operator *(Complexe z1, Complexe z2) {...}
Complexe operator *(float x, Complexe z) {...}
```

C++ Surcharge des opérateurs

- Les opérateurs suivants peuvent être surchargés :

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

- Les opérateurs suivants ne peuvent pas être surchargés :

```
::   .      .*
```

- Les opérateurs suivants ne peuvent être redéfinis que par des méthodes non statiques:

```
=    []     ()     ->
```

C++ Surcharge des opérateurs

- Il n'est pas possible de redéfinir l'arité, l'associativité ou la priorité d'un opérateur
- Il n'est pas possible de définir de nouveaux opérateurs
- Lorsque un opérateur est redéfini par une méthode celle-ci ne doit pas être statique
- Lorsque un opérateur est redéfini par une fonction, au moins l'un des deux paramètres doit être un type utilisateur (sauf pour **`new`** et **`delete`**)

Surcharge des opérateurs

- L'appel à un opérateur peut se faire en utilisant la notation habituelle ou infixée :

```
complexe z1,z2,z3;
z3=z1+z2;
z3=z1.operator+(z2) // + est une méthode
z3=operator+(z1,z2) // + est une fonction
```

C++ Surcharge des opérateurs

- Pour un opérateur binaire `@` et 2 variables **`x, y`** de 2 types **`X,Y`** l'écriture **`x@y`** se résout comme suit:
  1. si **`X`** est une classe et si `@` est défini dans cette classe alors il est utilisé
  2. sinon
     - Rechercher `@` dans le contexte de l'expression **`x@y`**
     - si **`X`** est défini dans le namespace N, rechercher la définition de `@` dans N
     - si **`Y`** est défini dans le namespace M, rechercher la définition de `@` dans M

C++ Surcharge des opérateurs

- Surcharge de l'opérateur **=**
  - Tester **x=x**
  - Libérer éventuellement
  - Affecter
  - retourner ***this**

C++ Surcharge des opérateurs

```
class Pile {
  ...
  Pile & operator =(const Pile &P);
  ...
}
Pile& Pile::operator=(const Pile &P){
  if (this!=&P){
      delete [] P.Elts;
      Tete=P.Tete;
      Elts=new int[Max=P.Max];
      for(int i=0;i<=Tete;i++) Elts[i]=P.Elts[i];
  }
  return *this;
}
```

C++ Surcharge des opérateurs

- Un constructeur d'une classe **X** avec un seul paramètre de type **T**, introduit une conversion implicite **T** → **X**

```
class X {
  X(T t) {...}
  ...
};
void F(X x) {...}
// Le passage doit être par valeur ou F(const X &x)
main(){
  T t1,t2;
  X x=t1; // Conversion implicite
  F(t2);  // Conversion implicite
}
```

C++ Surcharge des opérateurs

- Lorsque la conversion implicite n'est pas souhaitable on peut rendre le constructeur explicite :

```
class String {
public:
  String(const char *s);
  explicit String(int n);
...
}
main(){
  String S=10;//Erreur : pas de conversion int → String
  String S(10);//Ok
```

C++ Surcharge des opérateurs

- Conversion **`X`** → **`T`** où T est soit un type de base soit un type utilisateur ?

```
classe X {
   ...
   operator T()
   // Ni paramètres ni type résultat
  ...
}
```

Java surcharge

- Conversion implicites:
  - Types prédéfinis :
    - Widening : byte → int
    - Narrowing : float → int
    - Boxing/Unboxing: int ↔ Integer
  - Références :
    - Widening reference : sous type de T → T
    - Narrowing reference : T → Sous type de T, valide seulement si l'objet référencé a pour type le sous type
  - La conversion vers String est toujours possible
- Pas de surcharge des opérateurs

C++ Généricité

```
template <class TElt>
class Pile {
public:
  Pile();
  void Empiler(TElt x);
  void Depiler();
  TElt Sommet();
  bool EstVide();
  bool EstPleine();
private:
  int Tete;
  TElt LesElements[Max];
};
```

C++ Généricité

- Les méthodes d'une classe générique sont définies comme des fonctions générique, donc il faut réécrire **`template ...`**

```
template <class TElt>
void Pile<TElt>::Empiler(TElt x){...}

template <class TElt>
void Pile<TElt>::Depiler(){...}

template <class TElt>
TElt Pile<TElt>::Sommet(){...}
...
main(){
   Pile<int> P1;
   Pile<Complexe> P2;
   Pile< Pile<char> > P3;// Attention à l'espace (sinon opérateur >>)
}
```

- En dehors de son interface, une classe générique ne peut apparaître qu'instanciée **`Pile<TElt>`**

C++ Généricité

- Il ne peut y avoir qu'une fonction générique qui définit une méthode d'une classe générique, toutefois on peut surcharger et *spécialiser* des méthodes de classes génériques
- On ne peut pas surcharger une classe générique mais on peut *spécialiser* une classe générique
- Le type utilisé pour instancier une classe générique doit offrir toutes les opérations requises par celle-ci. C'est-à-dire l'utilisation du type générique impose des contraintes sur les types qui peuvent être utilisés pour l'instancier
- Chaque nouvelle instanciation entraîne que le compilateur génère un ensemble de déclarations

C++ Généricité

- Il est possible d'introduire des variables de type ordinaire dans les paramètres génériques

```
template<class TElt, int Max> class Pile {
...
private:
  int LesElements[Max];
...
};

template <class TElt, int Max>
void Pile<TElt,Max>::Empiler(TElt x){}
```

C++ Généricité

- Le paramètre effectif d'une variable dans les paramètres génériques ne pourra être que l'un des éléments suivants :
  - une expression constante de type intégral ou énuméré
  - L'adresse d'un objet ou d'une fonction ou d'une donnée membre
- Une variable paramètre générique est une constante pour l'unité générique qui l'a définie

C++ Généricité

- 2 types obtenus à partir d'unités génériques sont considérés identiques s'ils ont été obtenus par instanciation pour exactement les mêmes types et éventuellement les mêmes valeurs

```
typedef  int Entier;//typedef n'introduit pas de nouveaux types
typedef Pile<unsigned char,3> Pile3UCar;
typedef Pile<char,3> Pile3Car;
typedef Pile<char,4> Pile4Car;
typedef Pile<int,4> Pile4Int;
typedef Pile<Entier,11-7> Pile4Entier;
```

- Seuls `Pile4Int` et `Pile4Entier` sont identiques

C++ Généricité

- Fonctions génériques

```
template <class TVal>
TVal Min(TVal v1, TVal v2) {
  if (v1<v2)
      return v1;
  else return v2;
}
```

- L'instanciation n'apparaît pas

```
main() {
  int i1,i2,i3; i3=Min(i1,i2);
  String s1,s2,s3;s3=Min(s1,s2);
}
```

C++ Généricité

- Lors de l'appel, les types (et éventuellement les valeurs) sont déduits, il y a une instanciation implicite
- Lorsque le type ne peut pas être déduit, on peut instancier explicitement

```
template<class T1,class T2,class T3>
T1 Min(T2 a, T3 b) {a<b ? a:b;}
main(){
  float x,y;
  int i;
  i=Min(x,y) // Erreur le type T1 ne peut pas être déduit
  i=Min<int,float,float>(x,y) // Ok !
```

- Dans cet exemple l'instanciation peut s'interpréter comme calculer le min de 2 float et le convertir en int

Java Généricité

```
public class Pile<T> {
 public Pile(int taille) {
 Max=taille;
 Elts=new ArrayList<T>();
}
public void depiler() {if (Tete > -1) Elts.remove(Tete--);}
public T sommet() {
 if (Tete > -1)
    return Elts.get(Tete);
 else return null;
}
private ArrayList<T> Elts;
private int Tete=-1;
private int Max;}
```

Java Généricité

- Instanciation

```
Pile<Integer> P=new Pile<Integer>(10)
```

- On ne peut pas instancier avec des types primitifs
- On peut spécifier des contraintes sur les types effectifs

```
class UneClasse<T extends UneSuperclasse> {…}
```

=>Seuls les descendants de UneSuperclasse pourront être utilisés pour instancier

- On peut spécifier une contrainte en combinant des interfaces

```
class UneClasse<T extends C1 & I1 & I2>
```

Java Généricité - wildcard

- UneMethode n'accepte que les objets instanciés avec le même type effectif

```
class UneClasse<T> {
public UneMethode(UneClasse<T> x){…}

UneClasse<Integer> o1=new …
UneClasse<Float> o2=new …
o1.UneMethode(o2) //Erreur
```

- Le caractère joker permet de généraliser

```
class UneClasse<T> {
public UneMethode(UneClasse<?> x){…}
```

- Le caractère joker peut être contraint

```
class UneClasse<T> {
public UneMethode(UneClasse<? extends … > x){…}
```

Java méthode générique

- On peut limiter la généricité à des méthodes ou des constructeurs

```
class X{
 public X(){…}
 public <T> X(T t){…}
 public <V> M(V v) {…}
```

- Instanciation

```
X x=new X(123)
X x=new X(3,14)
String ch=new String('Hello')
x.M(ch)
```

Java Généricité - restrictions

- Le type formel ne pas être utilisé pour créer des objets ou des tableaux
- On ne peut pas créer de tableaux même si on fournit le type effectif

```
Pile<Integer> [] desPil=new Pile<Integer> //Erreur
```

- On peut pas créer des classes génériques pour les exceptions

Classes dérivées

- Terminologie
  - Relation d'héritage, ISA,
  - Classe de base et classe dérivée
  - Superclasse et sous-classe
- La classe dérivée est « plus grande » (contient plus d'informations que la classe de base)
  - A chaque fois qu'une classe est attendue une sous-classe est acceptée, l'inverse n'est pas vrai

C++ Classes dérivées

```
classe Article {
public:
   float PrixTTC();
   void Print();
...
protected:
   char *Ref;
   float PrixHT;
...
private:
   int NbrImp;
}
```

```
classe ArtTextile: public Article
   {
public:
   int GetColoris() const;
   void Print() const;
   // Print() est redéfinie ici
   // Est-elle surchargée ?
...
private:
   int Coloris;
...
}
```

C++ Classes dérivées

- La sous classe accède aux parties *publiques et protégées* de sa superclasse mais pas aux parties privées;

```
void Article:: Print() const{
 cout <<"Article Référence : " << Ref;
 cout << "Prix : " << PrixTTC();NbrImp++;
}
void ArtTextile::Print() const{
 cout <<"Article Textile Référence : " << Ref;
// Réutiliser Print() de Article ?
 cout <<" Coloris: " << Coloris;
 cout << "Prix : " << PrixTTC();// NbrImp++ ⇒ Erreur !
}
```

- Seules les classes dérivées ont accès aux parties protégées de la classe de base

C++ Classes dérivées

- La construction d'un objet dérivée sous entend la construction d'un objet de base :
  - Lorsque la classe de base possède un constructeur par défaut (sans paramètres) implicite ou explicite alors il n'y a pas de problèmes particulier, ce constructeur pourra être utilisé
  - Lorsque la classe de base ne possède pas de constructeur par défaut ou que l'on souhaite utiliser un de ses constructeurs en particulier alors il faut passer par la liste d'initialisation
- La destruction d'un objet dérivé sous entend la construction d'un objet de base
- Ordre des constructeurs : base, membres, objet
- Ordre des destructeurs : objet, membres, base

C++ Classes dérivées

- Le constructeur par copie et l'opérateur d'affectation de la classe de base ne sont pas invoqués implicitement lors de la copie ou de l'affectation d'un objet de la classe dérivée

```
class Base {...}
class Derivee : public Base {...}
main(){
 Derivee D;Base B(D);// Le constructeur par copie de Base est invoqué
 Derivee D1,D2(D1);
// Le constructeur par copie de Base n'est pas appelé implictement, le cas
// échéant, il doit être appelé explicitement (Liste D'initialisation) dans le
// constructeur par copie de Derivee :
        Derivee(const Derivee &D) : Base(D) {...}
```

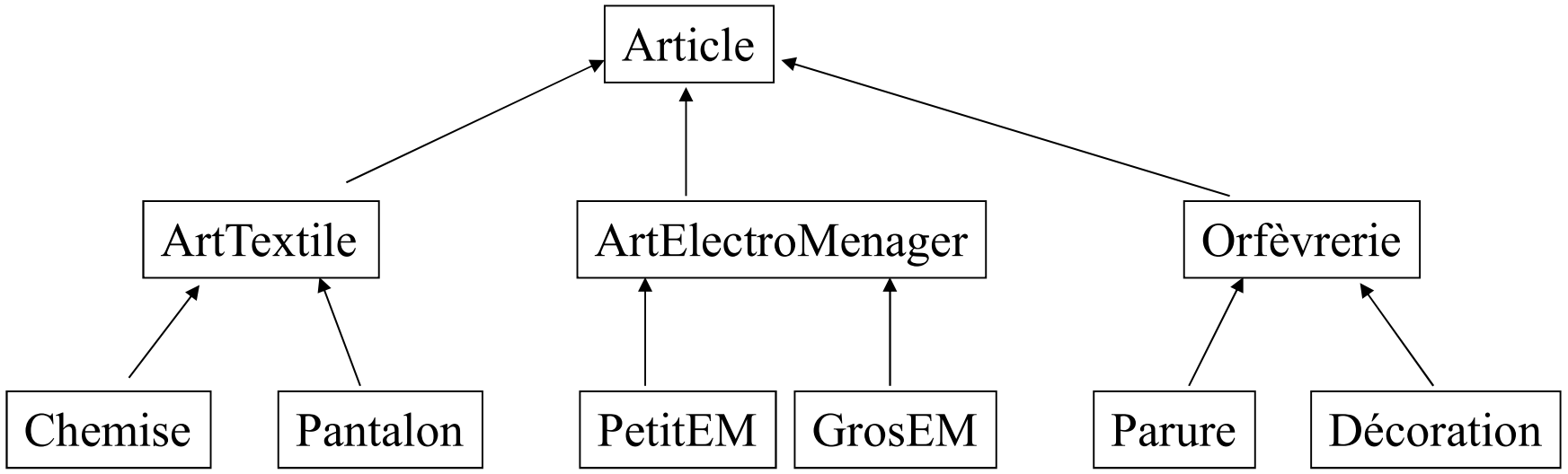
C++ Classes dérivées

```
classe Article {
public:
 Article(char *LaRef, float LePrix) {...}
...
}
```

```
classe ArtTextile public Article {
public:
 ArtTextile(char *UneRef,float UnPrix,int UnColoris) :
   Article(UneRef,UnPrix), Coloris(UnColoris) {}
...
}
```

Classes dérivées

- Hiérarchie de classes : une classe dérivée peut à son tour être une classe de base pour plusieurs autre classes



C++ Classes dérivées

- Liaison statique

```
main() {
 Article A("A001",0.200),*PtrA;
 ArtTextile AT("A002",0.900,1);
 A.Print();// Article Référence : A001 Prix 0.220
 AT.Print(); // Article Textile Référence : A002 Coloris : 1 Prix 0.990
 PtrA=@A;
 PtrA->Print(); // Article Référence : A001 Prix 0.220
 PtrA=@AT;
 PtrA->Print(); // Article Référence : A002 Prix 0.900 !!!
}
```

La Liaison s'est faite statiquement : l'adresse de la méthode à appeler est établie au moment de la compilation et ne dépend que du type de la déclaration

C++ Classes dérivées

- Liaison dynamique

```
classe Article {
public:
  float PrixTTC();
  virtual void Print();
...
}
main() {
 Article *PtrA;
 ArtTextile AT("A002",0.900,1);
 PtrA->Print();// Article Textile Référence : A002 Coloris : 1 Prix 0.900
}
```

La méthode invoquée est déterminée dynamiquement sur la base du type de l'objet pointé

C++ Classes dérivées

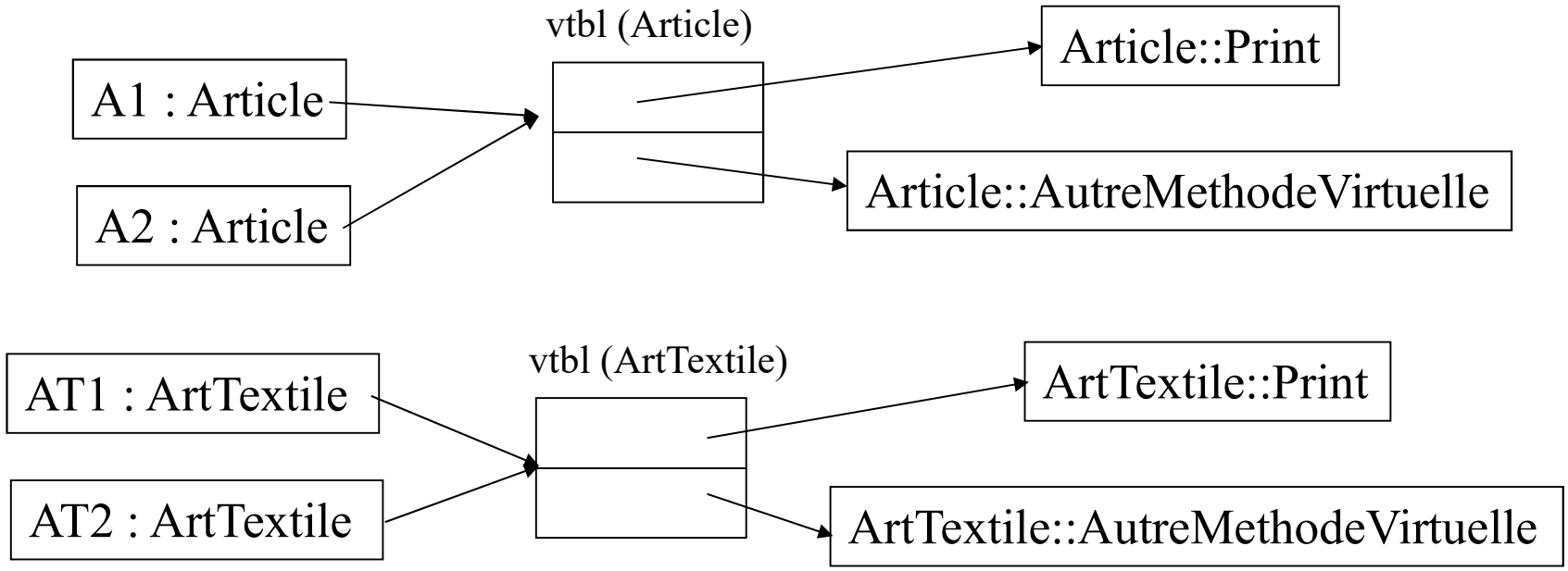
- Une classe peut introduire des méthodes virtuelles même si aucune classe d'hérite d'elle
- Une méthode virtuelle n'est pas obligatoirement redéfinie dans les descendants
- Grâce aux méthodes virtuelles, le compilateur gère automatiquement des constructions

```
Selon Type_Objet faire
    cas TypeA : …, cas TypeB  : …
FSelon
```

- Il suffit qu'une méthode soit définie une fois virtuelle pour être virtuelle dans tous les descendants

C++ Classes dérivées

- Chaque classe ayant des méthodes virtuelles introduit une table des méthodes virtuelles (vtbl). Chaque objet de ces classes a un pointeur sur sa vtbl, l'appel à une méthode nécessite une indirection



C++ Classes dérivées

- Dans certains cas, il n'y a pas d'implantation naturelle pour une méthodes virtuelle au niveau de la classe de base.

  classe de base : **`Figure`**

  classe dérivées : **`Cercle, Triangle`**

  Quelle est l'implantation de **`Affiche`** dans **`Figure`** ?

- Une méthode abstraite (ou virtuelle pure) est une méthode qui n'a pas d'implantation

```
class Figure {
  virtual void Affiche()=0;
...
};
```

Classes dérivées

- Une classe abstraite est une classe dont au moins l'une des méthodes est abstraite
- Une classe qui dérive d'une classe abstraite est soit elle-même aussi abstraite, soit elle implante toutes les méthodes abstraites de la classe dont elle dérive
- Une classe interface est une classe qui n'a pas d'état et dont toutes les méthodes sont abstraites
- Une classe abstraite ne peut pas s'instancier, elle est manipulable à travers des pointeurs

Java héritage

```
public classe Article {
public float PrixTTC(){…}
public  void Print(){…}
...
protected String Ref;
protected  float PrixHT;
...
private int NbrImp;
}
```

```
public classe ArtTextile extends Article  {
public int GetColoris(){…}
private int Coloris;
...
}
```

Java Classes dérivées

- La construction d'un objet dérivé sous entend la construction d'un objet de base : **super** permet d'invoquer le constructeur de la classe de base

```
ArtTextile(String UneRef,float UnPrix,int
UnColoris){super(UneRef,UnPrix),Coloris=UnColoris;}
```

- **super** doit être la première instruction du constructeur de la sous classe.
- L'ordre de construction suit l'ordre de dérivation : la superclasse est construite avant la sous-classe
- La déclaration d'une classe **final** interdit de dériver des sous classes à partir de celle-ci

Java redéfinition

- L'annotation @override avant l'écriture d'une méthode permet de faire vérifier au compilateur qu'il s'agit d'une redéfinition
- **`final`** avant la définition d'une méthode interdit sa redéfinition dans les classes dérivée
- La méthode redéfinie ne peut pas limiter la visibilité de la méthode de la classe mère
- **`super`** dans la classe dérivée permet d'accéder aux ressources de la classe mère, en particulier il permet d'accéder à la version originale d'une méthode redéfinie

Java classe abstraite

```
public abstract class UneClasseAbstraite {
  public abstract void UneMethodeAbstaite(…);
```

- Une classe abstraite contient au moins une méthode abstraite
- Une classe abstraite ne peut pas être instanciée
- Une classe dérivée d'une classe abstraite doit soit implanter toutes les méthodes abstraites soit être elle aussi déclarée abstraite

Java interfaces

```
public interface UneInterface {…}
```

- Une classe implémente une ou plusieurs interfaces

```
public class MaClasse implements I1, I2,…{…}
```

- Une classe qui ne fournit pas une implémentation pour toutes les méthodes des interfaces qu'elle implémente doit être déclarée abstraite
- Une interface peut hériter d'une ou plusieurs interfaces

```
interface I extends I1, I2,  I3,… {…}
```

- Une interface peut introduire des constantes et des attributs finaux.
- Une interface peut fournir une implantation par défaut, son implémentation n'est plus obligatoire dans les descendants

```
public interface I {
  default public void M() {…}
```

Inversion du contrôle

```
class Parametrable {
  public:
   void Perform() {
       ...
      aHook->Service();
      ...
   }
  private:Hook *aHook;
}
```

```
class Hook {
  virtual void Service()=0
}
```

```
class HookImpl : public Hook {
  void Service(){
     ...
  }
}
```

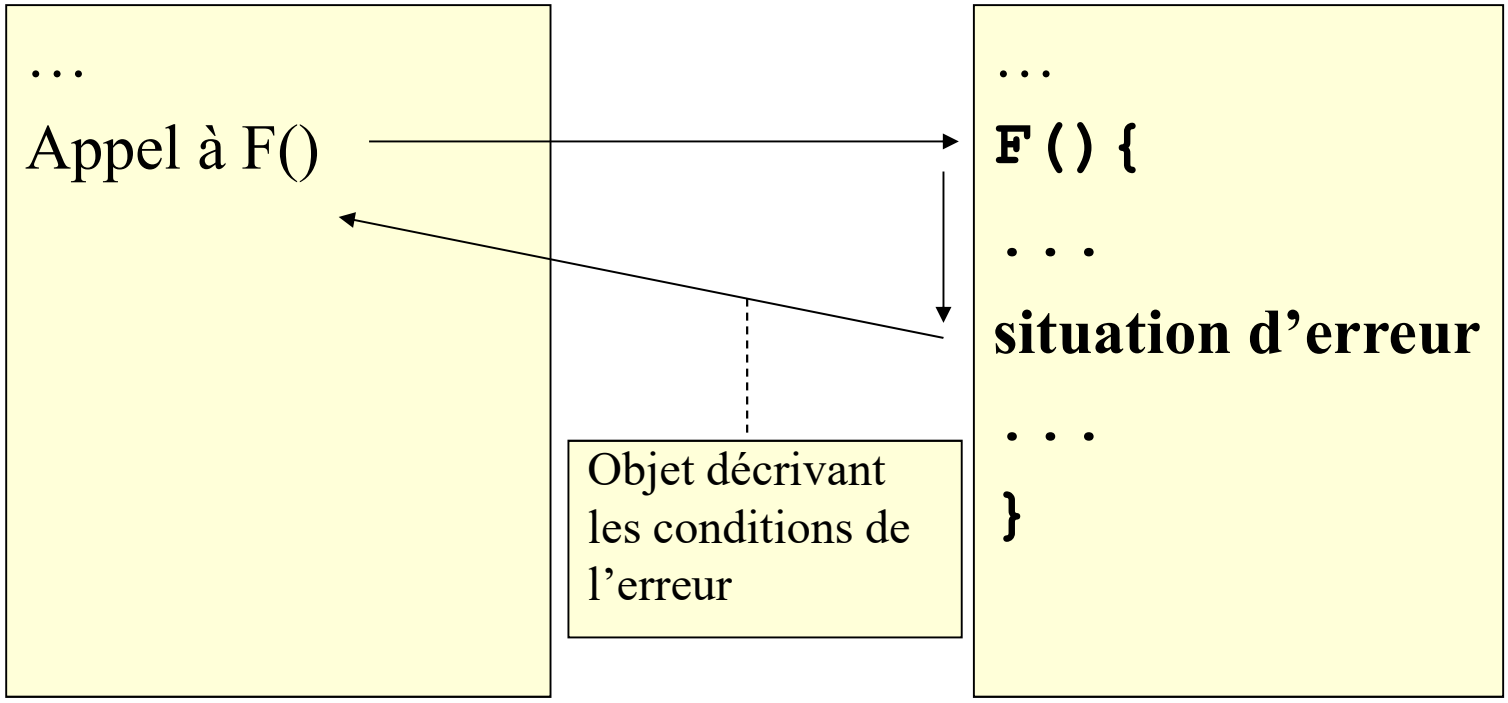
Les exceptions

Adel KHALFALLAH
Institut Supérieur d'Informatique – El Manar
Adel.Khalfallah@fst.rnu.tn
Adel_Khalfallah@yahoo.fr

Les exceptions

- Il est souhaitable d'établir une distinction entre le code qui concerne le mode de fonctionnement normal et celui qui concerne les erreurs
- Le module qui détecte une erreur ne sait pas nécessairement la traiter.
- Un module qui rencontre une situation d'erreur et qui ne sait pas la traiter *soulève* une exception
- Un module qui se trouve en présence d'une exception peut soit la traiter soit la propager
- Une exception est un objet d'un type prédéfini ou utilisateur, soulever une exception revient à transmettre un objet

Les exceptions



Les exceptions

```
struct Exc_Domain {
 Exc_Domain(int Valeur_Err) : Err_Val(Valeur_Err)) {}
 ~Exc_Domain() {}
   int Err_Val;
};

int Fact(int N) {
  if (N<0 && N>=10){
      // Situation d'erreur
      Exc_Domain Exc(N);
      throw Exc;//  ou directement  throw Exc_Domain(N)
  }
  else {// code Normal
         ...
```

Les exceptions

- Le *gestionnaire d'exception* (exception handler) permet de spécifier le code de traitement de l'exception et d'éviter sa propagation

- Exemple 3 modules :
  - Calcul_Structure : introduit un gestionnaire d'exceptions
  - Limite : module intermédiaire qui propage l'exception
  - Fact : module qui soulève l'exception

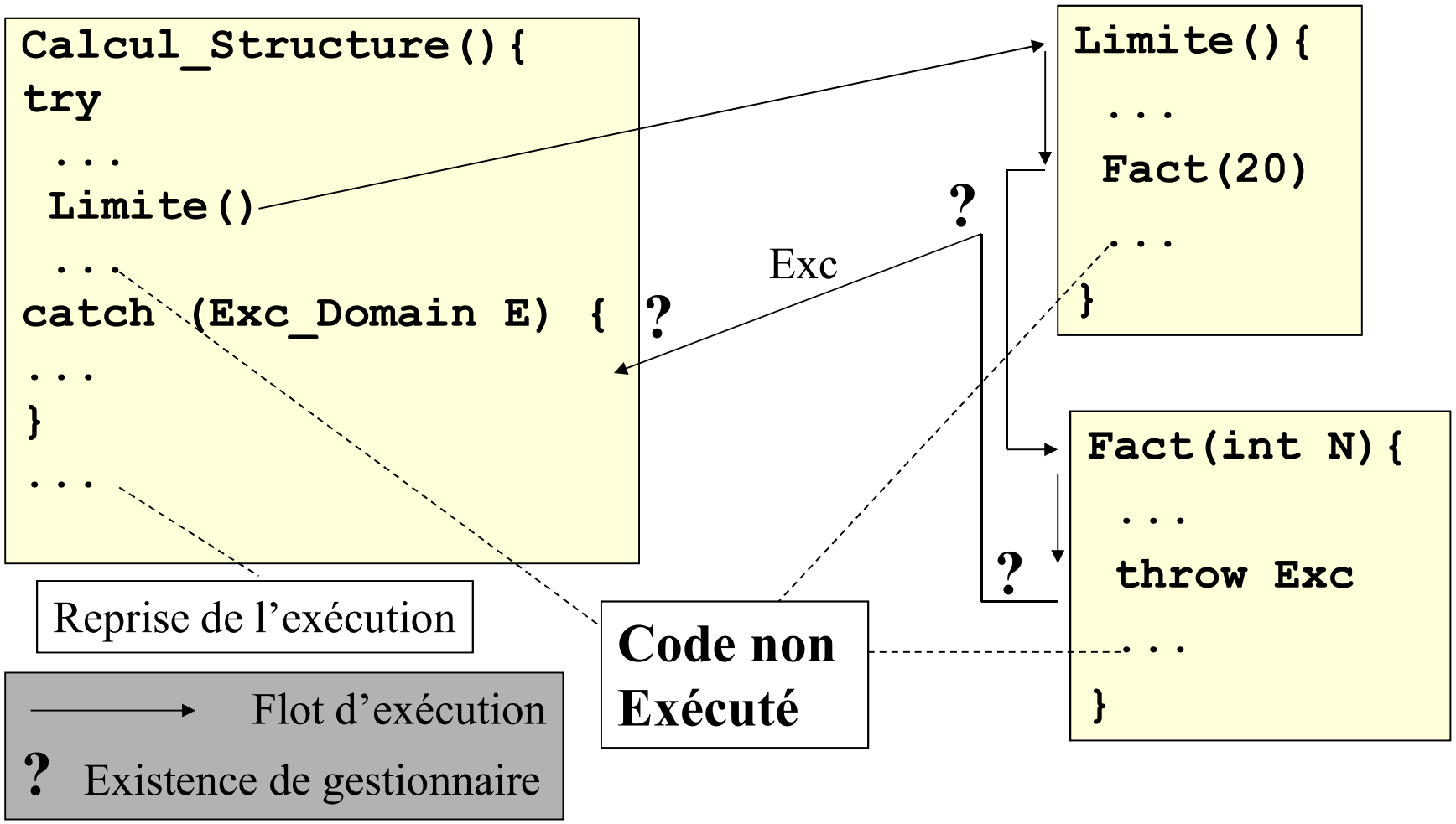
  Les 3 modules sont en relation suite à l'imbrication des appels

Les exceptions

```
double calcul_structure(){
...
try
   ...
   Limite()
   ...
catch (Exc_Domain E){// traitement de l'erreur
   cout<< E.Err_Val << "Hors limites"
}

void Limite(){
...
Fact(20);
...
}
```

Les exceptions



Les exceptions

- Dans un gestionnaire on peut trouver autant de branches **`catch`** que de type d'exceptions traitées
- Une exception est traitée dès qu'elle rencontre son gestionnaire: il n'y a pas de récursivité

```
catch(Exc E){
        throw Exc()
}
```

- Les gestionnaires d'exception peuvent être imbriqués (A éviter)