

Programmation Orientée Objet – C++



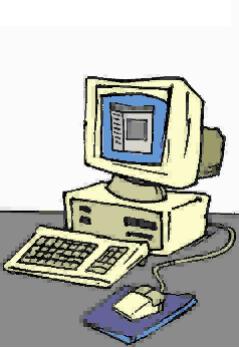
« There are two ways to write error-free programs;
only the third one works »

« You never finish a program,
you just stop working on it »

« L'expérience est une lanterne que l'on porte
sur le dos et qui n'éclaire jamais que le
chemin parcouru »

« La nature nous a dotés de **deux** oreilles et
d'**une** seule bouche...
Il convient donc d'écouter **deux fois plus** que
de parler ! »

« Combien faut-il de programmeurs C++ pour
changer une ampoule ?
Réponse : 6
Un pour la changer et 5 autres, six mois plus
tard, pour comprendre comment il a fait. »




© MM - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON

Mathieu Maranzana

Mathieu.Maranzana@insa-lyon.fr



Programmation Orientée Objet – C++





Naufrage

Sources de ce COURS

- B. Stroustrup... la principale source
 - ◆ Le Créateur du C++
 - ◆ Livre : Le Langage C++
- Différents cours en ligne
- La Toile... Évidemment !
- Et Moi !
- Et tous ceux que j'oublie...



Divin ?



Frisson



Bricolage



© MM - Reproduction interdite sans l'autorisation de l'auteur



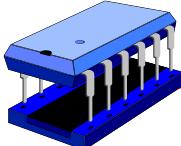
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON

2

Programmation Orientée Objet – C++



Compétences – IFA-3-S1-EC-POO1



- Approche par compétences...
 - ◆ Mettre en œuvre des méthodologies de développement de logiciels
 - ◆ Concevoir l'architecture d'un logiciel orienté objet
 - ◆ Concevoir, réaliser et maintenir des logiciels de qualité
 - ◆ Être capable d'implémenter de bons logiciels (mécanisme des langages orientés objet, choix des algorithmes et structures de données...)







© MM - Reproduction interdite sans l'autorisation de l'auteur

Programmation Orientée Objet – C++



PLAN du COURS




Naufrage

- Programmation Orientée Objet
- Préprocesseur
- Données et Types
- Expressions
- Fonctions
- Instructions
- Classe
- Héritage



Divin ?



Bricolage




© MM - Reproduction interdite sans l'autorisation de l'auteur

Programmation Orientée Objet – C++

PLAN du COURS



Naufrage

- Programmation Orientée Objet
 - ◆ Approche Procédurale vs Objet
 - ◆ Historique et Caractéristiques du C++
 - ◆ Concepts Essentiels du C++
 - ◆ Mots-clés du C++
 - ◆ Un Premier Programme C++
 - ◆ Module, Interface et Réalisation



Divin ?



Bricolage

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON



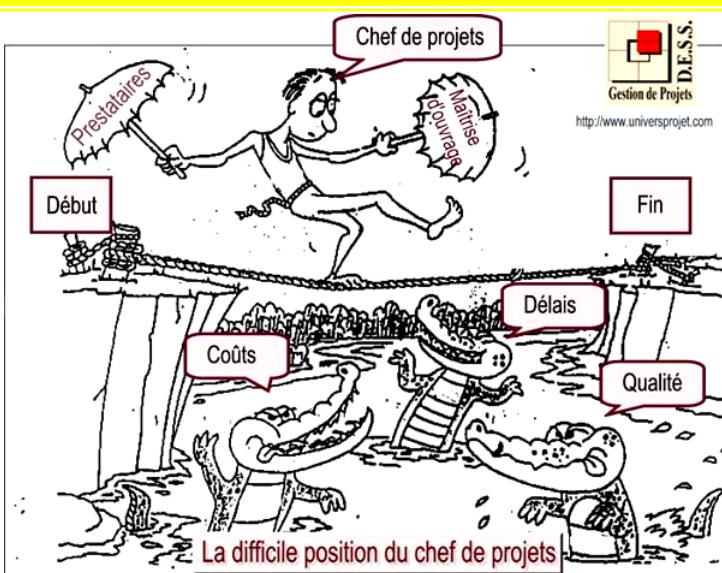
Frisson

- Préprocesseur
- Données et Types
- Expressions



© MM - Reproduction interdite sans l'autorisation de l'auteur

Programmation Orientée Objet – C++



Chef de projets

Début

Prestataires

Maitrise

Contrôle

Fin

Délais

Coûts

Qualité

La difficile position du chef de projets

D.E.S.S.

Gestion de Projets

<http://www.universprojet.com>

<http://www.projetsinformatiques.com>

© MM - Reproduction interdite sans l'autorisation de l'auteur



Approche Procédurale vs Objet

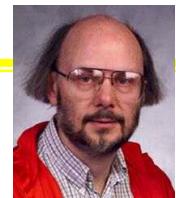
- Un programme informatique, c'est quoi ?
 - ◆ Des données...
 - ◆ Des traitements définis par des algorithmes utilisant un langage de programmation

- Approche procédurale
 - ◆ Accent mis sur les traitements
 - ◆ Découpage en fonctions pour la gestion des traitements
 - ◆ Processus de raffinements successifs
 - ◆ Développement *top-down*

- Approche objet
 - ◆ Accent mis sur les données
 - ◆ Regroupement des traitements (méthodes) selon la sémantique des données (attributs)
 - ◆ Construction de classes
 - ◆ Développement *bottom-up*



Historique du Langage C++



- Langage C++
 - ◆ Première implémentation dans les années 1980
 - ◆ Bjarne Stroustrup – Laboratoire AT&T Bell
 - ◆ BCPL + C + Simula67 + Algol68 = "C with Classes"
 - ◆ Rick Mascitti : nom C++ (prononcé *see plus plus*) (1983)
 - ◆ C++ ≡ C + 1 ☺ ⇒ nature évolutive du langage
 - ◆ Première normalisation ISO / IEC 14882:1998 (en 1998)
 - ◆ Depuis... C++11, C++14, C++17, C++20, C++23, C++26...

- Au départ, simple pré compilateur ⇒ traduction des constructions C++ vers C
 - ◆ Conséquence : c++ est un sur ensemble du c
 - ◆ Pour faire du (mauvais) c++, il suffit de changer l'extension du fichier source (.c vers .cpp)
 - ◆ Approche procédurale ≠ approche orientée objet

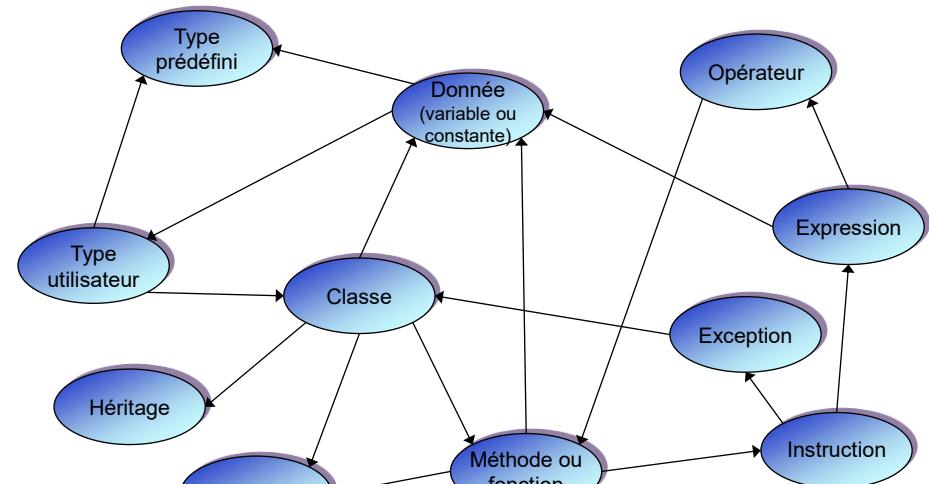


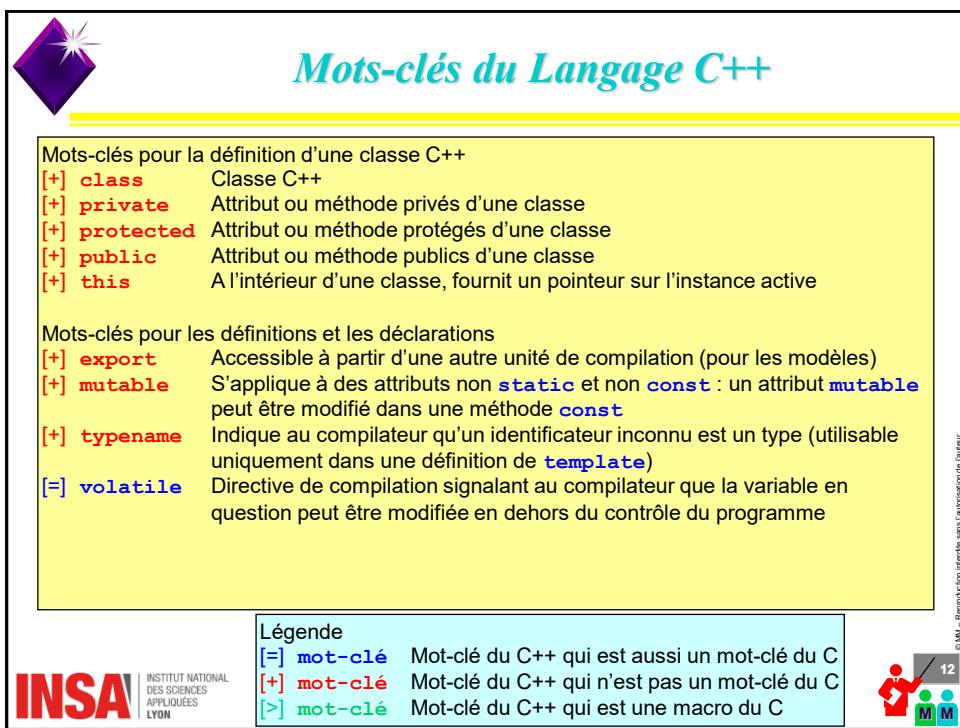
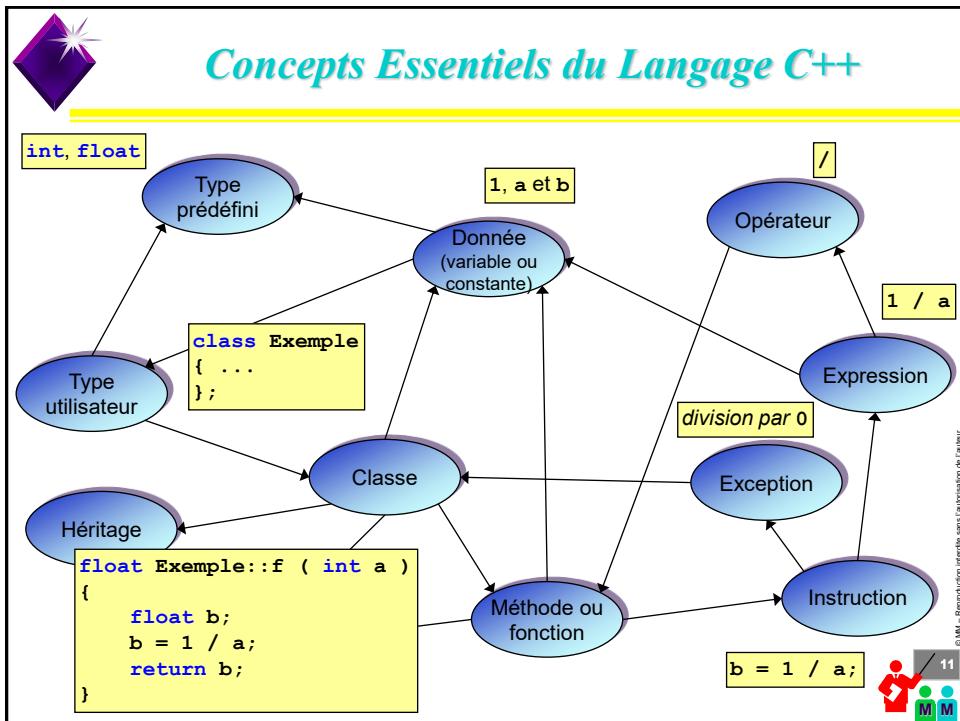
Caractéristiques du langage C++

- Principales extensions par rapport au langage C
 - ◆ Surcharge des fonctions et des opérateurs
 - ◆ Paramètres par défaut des méthodes et fonctions
 - ◆ Référence et passage des paramètres par référence
 - ◆ Généralisation de la notion de constante (mot-clé `const`)
 - ◆ Allocation typée de la mémoire dynamique
 - ◆ Bibliothèques d'entrées / sorties – Librairies standard
 - ◆ Traitement des exceptions
- Ajout des concepts de la programmation objet
 - ◆ Encapsulation des données
 - ◆ Héritage (multiple)
 - ◆ Polymorphisme
 - ◆ Généricité des classes et des fonctions



Concepts Essentiels du Langage C++







Mots-clés du Langage C++

Mots-clés pour les définitions et les déclarations

[=] auto	Directive de compilation pour l'allocation de variables
[=] const	Déclaration d'une valeur constante
[=] extern	Signale une valeur connue à l'extérieur du contexte du fichier courant (.cpp) de compilation
[+] namespace	Définition d'un espace de dénomination des identificateurs
[=] register	Directive de compilation assignant la variable considérée à un registre plutôt qu'à une place mémoire
[=] static	Signale une valeur non connue à l'extérieur du contexte du fichier courant (.cpp) de compilation ou à l'extérieur de la procédure courante
[+] using	Utilisation d'un espace de dénomination
[+] virtual	Membre virtuel pouvant être redéfini par une classe dérivée

© MM - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



13



Mots-clés du Langage C++

Mots-clés pour les types prédéfinis

[+] bool	Booléen
[=] char	Caractère et wchar_t Caractère étendu
[=] char16_t	Caractère étendu (représentation des caractères UTF-16 (depuis C++11))
[=] char32_t	Caractère étendu (représentation des caractères UTF-32 (depuis C++11))
[=] double	Nombre virgule flottante en double précision
[=] enum	Type énuméré
[+] false	Valeur possible pour une donnée (variable ou constante) booléenne
[=] float	Nombre virgule flottante en simple précision
[=] int	Entier
[=] long	Entier long
[=] short	Entier court
[=] signed	Type signé (entier, entier court...)
[=] struct	Définition d'une structure
[+] true	Valeur possible pour une donnée (variable ou constante) booléenne
[=] union	Définition d'une structure avec des variantes
[=] unsigned	Type non signé (entier, entier court...)
[=] void	Non typé

© MM - Reproduction interdite sans l'autorisation de l'auteur

Mot-clé pour les types de l'utilisateur

[=] typedef	Définition d'un synonyme de type
--------------------	----------------------------------



Mots-clés du Langage C++

Mots-clés pour les instructions

[+] asm	Directive de compilation: les instructions comprises dans le bloc asm { } sont en langage assembleur
[=] break	Rupture d'une séquence d'exécution
[=] case	Utilisé en conjonction avec une instruction de sélection multiple (switch)
[=] continue	Continuation dans une séquence d'instructions (boucle)
[=] default	Utilisé en conjonction avec switch , si la valeur ne correspond à aucune étiquette case (cas par défaut)
[=] do	Boucle "répéter ... jusqu'à ..." (itérations 1 à n)
[=] else	Clause sinon d'une instruction conditionnelle "si ... alors ... sinon ..."
[=] for	Boucle "pour ..." (itérations n à m)
[=] goto	Branchemet à une étiquette (instruction inutile)
[=] if	Condition suivie de la clause alors dans une instruction conditionnelle "si ... alors ... sinon ..."
[=] return	Retour de fonction
[=] switch	Condition dans une instruction de sélection multiple
[=] while	Boucle "tant que ... faire ..." (itérations 0 à n) ou "répéter ... jusqu'à ..." (itérations 1 à n)



© MM - Reproduction interdite sans l'autorisation de l'auteur



15



Mots-clés du Langage C++

Mots-clés pour les opérateurs

[>] and	Opérateur && dans une opération logique
[>] and_eq	Opérateur &=
[>] bitand	Opérateur & dans une opération bit à bit
[>] bitor	Opérateur dans une opération bit à bit
[>] compl	Opérateur ~ dans une opération bit à bit
[+] const_cast	Opérateur const_cast < > supprime la caractéristique const d'un objet
[+] delete	Opérateur de libération de place mémoire
[+] dynamic_cast	Opérateur de conversion dynamique d'une expression vers un type donné
[+] new	Opérateur d'allocation de la mémoire
[+] nullptr	Pointeur nul (C++11)
[>] not	Opérateur ! dans une opération logique
[>] not_eq	Opérateur != dans une opération logique
[+] operator	Déclaration d'un opérateur
[>] or	Opérateur dans une opération logique
[>] or_eq	Opérateur !=
[+] reinterpret_cast	Opérateur de conversion de n'importe quel type de pointeur vers un autre type de pointeur
[=] sizeof	Opérateur de calcul de taille de type ou de donnée en octets
[+] static_cast	Opérateur de conversion statique d'une expression vers un type donné
[+] typeid	Opérateur typeid détermine le type d'un objet à l'exécution
[>] xor	Opérateur ^ dans une opération bit à bit
[>] xor_eq	Opérateur ^=

© MM - Reroduction interdite sans l'autorisation de l'auteur



Mots-clés du Langage C++

Mots-clés pour la définition de méthodes ou de fonctions

[+] **explicit** Interdit l'utilisation d'un constructeur **explicit** pour une conversion

implicite

[+] **friend** Fonction "amie" d'une classe

[=] **inline** Fonction en ligne : demande au compilateur de ne pas générer un appel de procédure, mais de recopier le code

Mot-clé pour la généréricité (définition de modèles)

[+] **template** Déclaration d'un modèle de fonction ou de classe

Mots-clés pour les différents modes d'héritage

[+] **private** Mode d'héritage le plus strict

[+] **protected**

[+] **public** Mode d'héritage le plus courant

Mots-clés pour les exceptions

[+] **catch** Déclaration d'un gestionnaire d'exception

[+] **throw** Lancement d'une exception à partir de la séquence d'instructions gardée

[+] **try** Exécution gardée d'une séquence d'instructions



© AMI - Reproduction interdite sans l'autorisation de l'auteur

17



Premier Programme C++

```
/*
 * Texte source (compte.cpp) d'un
 * programme comptant le nombre de lignes
 * d'un fichier texte lu sur l'entrée standard
 */
using namespace std;
#include <iostream>

int main ()
{
    char carLu;
    int nbLignes = 0;
    while ( cin.get ( carLu ) )
    {
        if ( carLu == '\n' )
        {
            nbLignes++;
        }
    }
    cout << nbLignes << " lignes";
    cout << endl;
    return 0;
} //---- Fin du main
```

⇒ Lecture des interfaces des classes d'entrées / sorties (objets **cin** et **cout**)
 ⇒ Utilisation de l'espace de noms **std** pour éviter les écritures **std::cin.get**, **std::cout** OU **std::endl**

⇒ Définition des variables locales (données) avec initialisation à 0 de **nbLignes**

⇒ Boucle (tant que) de lecture des caractères à partir du flot d'entrée standard (**cin**)
 ⇒ Fin de fichier au clavier : **ctrl-d** sous Linux
 ⇒ Redirection possible à l'appel (< sous Linux)

⇒ Affichage du résultat sur le flot de sortie standard (**cout**)
 ⇒ Retour à la ligne en fin d'affichage grâce au manipulateur de flux **endl**
`ostream & endl (ostream &);`



Exécution du Programme C++

- Compilation + Édition des liens (production d'un programme)
 - ◆ `g++ -o compte compte.cpp`
- Exécution
 - ◆ Avec une lecture au clavier
 - ◆ Avec une lecture à partir d'un fichier disque

```
3IF>compte<
Voilà une première ligne
Suisie d'une deuxième
Et la dernière ligne
Ctrl-d
3 lignes
3IF>
```

⇒ Ctrl-d au début d'une nouvelle ligne

```
3IF>compte < compte.cpp<
22 lignes
3IF>
```

⇒ < opérateur de redirection Linux



Aspects Lexicaux en C++

- Présentation libre du texte source ⇒ à exploiter
- Majuscules ≠ minuscules
- Mots-clés du langage réservés
 - ◆ Toujours en minuscules
- Identificateur
 - ◆ Suite de lettres, de chiffres et de '_' ne commençant pas par un chiffre
 - ◆ Longueur quelconque
- Commentaire
 - ◆ `// ...` commentaire style C++, s'étend jusqu'à la fin de la ligne
 - ◆ `/* ... */` commentaire style C, peut occuper plusieurs lignes
 - ◆ **Bien choisis et bien rédigés**, les commentaires contribuent à la qualité d'un programme



Programmation Modulaire en C++

- Module source : composé de 2 fichiers...
 - ◆ Interface : fichier de déclaration ou d'en-tête (header file), d'extension **.h** ⇒ peut contenir...
 - Définition de classes, de constantes, de types...
 - Déclaration de méthodes et de fonctions ordinaires exportées
 - ◆ Réalisation : fichier de définition ou d'implémentation, d'extension **.cpp** ⇒ peut contenir...
 - Définition des méthodes (fonctions membres)
 - Définition des fonctions ordinaires exportées
- Recommandation pour les classes
 - ◆ Une classe = un module source = une interface + une réalisation
 - ◆ Utilisation de squelettes ⇒ aide à l'organisation
- Point d'entrée d'une application = fonction C / C++ main
 - ◆ **int main () { ... } // sans paramètre**
 - ◆ **int main (int nb, char *list []) { ... }**

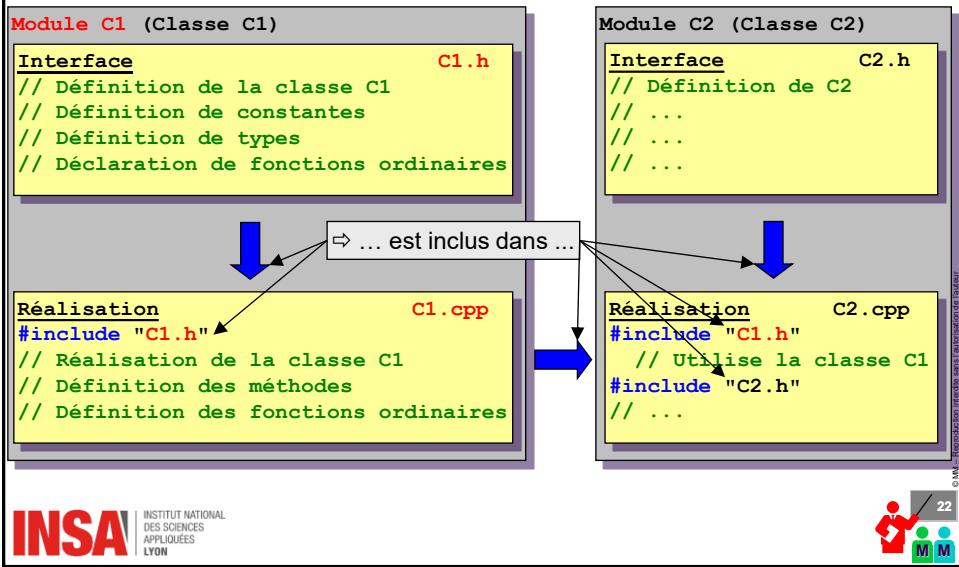
© AMI - Reproduction interdites sans l'autorisation de l'auteur

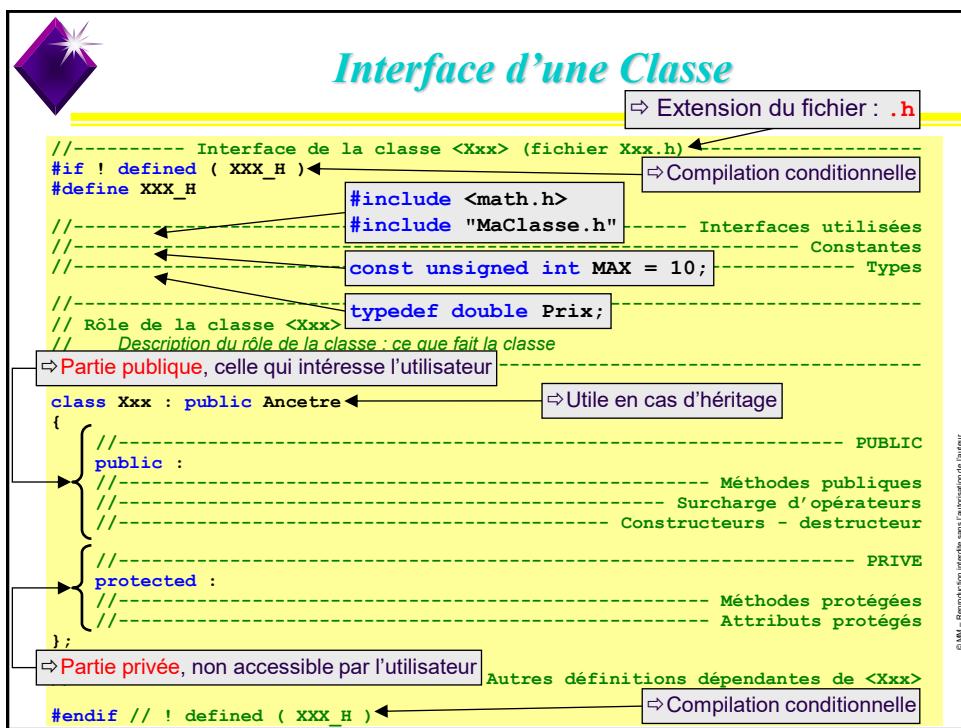
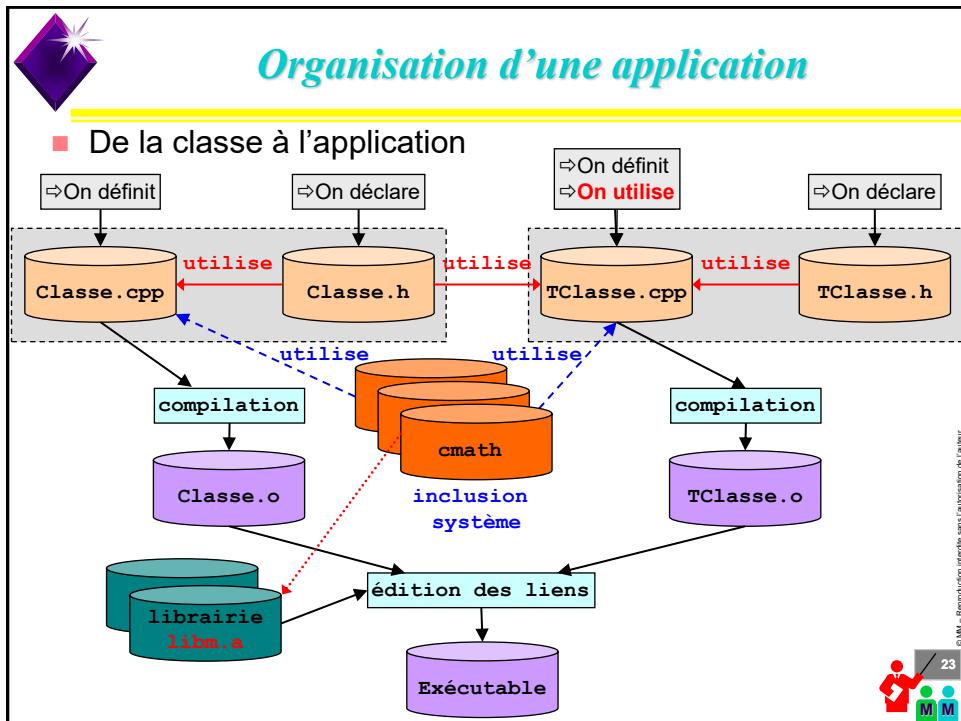


21



Programmation Modulaire en C++





Interface d'une Classe

```

//----- Interface de la classe <XXX> (fichier XXX.h) -----
#ifndef ! defined ( XXX_H )
#define XXX_H

//----- Interfaces utilisées ----- Constantes
//----- Rôle de la classe <XXX>
// Description du rôle de la classe : ce que fait la classe
//----- Contrat :
// Présentation du contrat, si nécessaire

class Xxx : public Ancetre
{
    //----- Méthodes publiques ----- Surcharge d'opérateurs
    //----- Constructeurs - destructeur
    //----- PRIVE
    //----- Protégées
    //----- Protégées
};

//----- Contrat :
// Présentation du contrat, si nécessaire
#endif // ! defined ( XXX_H )

```

Annotations:

- Interfaces utilisées**: `type NomDeLaMéthode (liste des paramètres);`
- Rôle de la classe**: `// Mode d'emploi :
// Description des paramètres
// Ce que fait la méthode`
- Contrat**: `// Contrat :
// Présentation du contrat, si nécessaire`
- Méthodes publiques**: `Xxx & operator = (const Xxx & unXxx);`
- Surcharge d'opérateurs**: `// Mode d'emploi :
// Description du paramètre
// Ce que fait l'opérateur d'affectation`
- Constructeurs - destructeur**: `Xxx (const Xxx & unXxx);
Xxx ();
virtual ~Xxx ();`
- PRIVE**: `Xxx ();`
- Protégées**: `operator = (const Xxx & unXxx);`
- Protégées**: `~Xxx ();`

© MM - Reproduction interdite sans l'autorisation de l'auteur

Interface d'une Classe

```

//----- Interface de la classe <XXX> (fichier XXX.h)
#ifndef ! defined ( XXX_H )
#define XXX_H

//----- Rôle de la classe <XXX>
// Description du rôle de la classe : ce que fait la classe
//----- Contrat :
// Présentation du contrat, si nécessaire

class Xxx : public Ancetre
{
    //----- Méthodes publiques ----- Surcharge d'opérateurs
    //----- Constructeurs - destructeur
    //----- PRIVE
    //----- Protégées
    //----- Protégées
};

//----- Contrat :
// Présentation du contrat, si nécessaire
#endif // ! defined ( XXX_H )

```

Annotations:

- Interfaces utilisées**: `Xxx (const Xxx & unXxx);`
- Rôle de la classe**: `// Mode d'emploi :
// Description du paramètre
// Ce que fait le constructeur de copie`
- Contrat**: `// Contrat :
// Présentation du contrat, si nécessaire`
- Méthodes publiques**: `Xxx ();`
- Surcharge d'opérateurs**: `// Mode d'emploi :
// Ce que fait le constructeur par défaut`
- Constructeurs - destructeur**: `Xxx ();
virtual ~Xxx ();`
- PRIVE**: `Xxx ();`
- Protégées**: `operator = (const Xxx & unXxx);`
- Protégées**: `~Xxx ();`

© MM - Reproduction interdite sans l'autorisation de l'auteur

Interface d'une Classe

```
----- Interface de la classe <Xxx> (fichier Xxx.h) -----
#ifndef ! defined ( XXX_H )
#define XXX_H

//----- Interfaces utilisées
//----- Constantes
//----- Types

//----- Rôle de la classe <Xxx>
// Description du rôle de la classe : ce que fait la classe
//-----



class Xxx : public Ancetre
{
    //----- public :
    //----- Contrat :
    //----- Présentation du contrat, si nécessaire
    type nomDeLaMéthode ( liste des paramètres );
    // Mode d'emploi :
    // Description des paramètres
    // Ce que fait la méthode
    //----- Méthodes protégées
    //----- Attributs protégés
protected :
};

//----- Autres définitions dépendantes de <Xxx>

#endif // ! defined ( XXX_H )
int tailleCourante;
```



Réalisation d'une Classe

```
----- Réalisation de la classe <Xxx> (fichier Xxx.cpp) -----
// Extension du fichier : .cpp
// INCLUDE
// Include système
using namespace std;
#include <iostream>
// Include personnel
#include "Xxx.h"
// Constantes
// PUBLIC
// Méthodes publiques
// Surcharge d'opérateurs
// Constructeurs - destructeur
// PRIVE
// Méthodes protégées

----- Réalisation de la classe <Xxx> (fichier Xxx.cpp) -----
```

Diagramme des éléments d'un fichier C++:

- Extension du fichier : .cpp
- Inclusion de la bibliothèque C++ de manipulation des flots d'E/S
- Définition de l'espace de noms std
- Inclusion par défaut de l'interface de la classe



Réalisation d'une Classe

```
----- Réalisation de la classe <Xxx> (fichier Xxx.cpp) -----
//----- INCLUDE
//----- Include système

using namespace std;
#include <iostream>
//----- XXXXX.h
#include "Xxx.h"

type Xxx::NomDeLaMéthode ( liste des paramètres )
// Inutile de rappeler les commentaires de l'interface
// Algorithme :
// Présentation de l'algorithme (astuces)
// (si la méthode est complexe)
{
    // Code de la méthode avec des commentaires pertinents
} //--- Fin de NomDeLaMéthode

//----- Méthodes publiques

//----- Surcharge d'opérateurs

    <-->
Xxx & Xxx::operator = ( const Xxx & unXxx )
// Inutile de rappeler les commentaires de l'interface
// Algorithme :
// Présentation de l'algorithme (astuces)
// (si la surcharge de l'opérateur est complexe)
{
    // Code de la méthode avec des commentaires pertinents
} //--- Fin de operator =
```

© MM - Reproduction interdite sans l'autorisation de l'auteur

Réalisation d'une Classe

```
----- Réalisation de la classe <Xxx> (fichier Xxx.cpp) -----  
  
//----- INCLUDE -----  
//----- Include système -----  
using namespace std;  
#include "Xxx.h"  
  
Xxx::Xxx ( const Xxx & unXxx )  
// Inutile de rappeler les commentaires de l'interface  
// Algorithme :  
// Présentation de l'algorithme (astuces)  
// (si le constructeur de copie est complexe)  
{  
#if defined ( MAP ) // MAP : Mise Au Point  
    cout << "Appel au constructeur de copie de <Xxx>" << endl;  
#endif  
    // Code du constructeur de copie avec des commentaires pertinents  
} //---- Fin de Xxx (constructeur de copie)  
  
//----- Constructeurs - destructeur -----  
//----- PRIVE -----  
//----- Méthodes protégées -----
```

© MM - Reproduction interdite sans l'autorisation de l'auteur



Réalisation d'une Classe

```

//----- Réalisation de la classe <Xxx> (fichier Xxx.cpp) -----
//----- INCLUDE -----  

//----- Include système -----  

using namespace std;  

#include <iostream>  

Xxx::Xxx ()  

// Inutile de rappeler les commentaires de l'interface  

// Algorithme :  

// Présentation de l'algorithme (astuces)  

// (si le constructeur par défaut est complexe)  

{  

#if defined ( MAP ) // MAP : Mise Au Point  

    cout << "Appel au constructeur par défaut de <Xxx>" << endl;  

#endif  

    // Code du constructeur par défaut avec des commentaires pertinents  

} //--- Fin de Xxx (constructeur par défaut)  

//----- Constructeurs - destructeur -----  

//----- PRIVE -----  

//----- Méthodes protégées -----  


```

© MM - Reproduction interdite sans l'autorisation de l'auteur



Réalisation d'une Classe

```

//----- Réalisation de la classe <Xxx> (fichier Xxx.cpp) -----
//----- INCLUDE -----  

//----- Include système -----  

using namespace std;  

#include <iostream>  

#include "Xxx.h"  

Xxx::~Xxx ()  

// Inutile de rappeler les commentaires de l'interface  

// Algorithme :  

// Présentation de l'algorithme (astuces)  

// (si le destructeur est complexe)  

{  

#if defined ( MAP ) // MAP : Mise Au Point  

    cout << "Appel au destructeur de <Xxx>" << endl;  

#endif  

    // Code du destructeur avec des commentaires pertinents  

} //--- Fin de ~Xxx (destructeur)  

//----- Constructeurs - destructeur -----  

//----- PRIVE -----  

//----- Méthodes protégées -----  


```

© MM - Reproduction interdite sans l'autorisation de l'auteur



Réalisation d'une Classe

```

//----- Réalisation de la classe <Xxx> (fichier Xxx.cpp) -----
//----- INCLUDE -----  

//----- Include système -----  

using namespace std;  

#include <iostream>  

//----- Include personnel -----  

#include "Xxx.h"  

//----- Constantes -----  

//-----  

//----- type Xxx::nomDeLaMéthode ( liste des paramètres )  

//----- Inutile de rappeler les commentaires de l'interface  

//----- Algorithme :  

//----- Présentation de l'algorithme (astuces)  

//----- (si la méthode est complexe)  

//----- {  

//-----     // Code de la méthode avec des commentaires pertinents  

//----- } //---- Fin de nomDeLaMéthode  

//-----  

//-----  

//----- PRIVE -----  

//-----  

//----- Méthodes protégées -----  


```

© INSA - Reproduction interdite sans l'autorisation de l'auteur



Guide de Style INSA-IF – Général

- ◆ Présentation
 - Pour des problèmes d'impression et de facilité de lecture dans une fenêtre, limiter la longueur des lignes (≈ 80 caractères).
- ◆ Commentaires
 - **O-16** Expliquer les passages de l'algorithme au code ne présentant pas un caractère évident. Un commentaire doit **expliquer et non dupliquer**.
La mise en page doit faciliter la lecture du code et des commentaires associés. Les commentaires noyés dans le code sont de moindre intérêt.

© INSA - Reproduction interdite sans l'autorisation de l'auteur

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

34



Guide de Style INSA-IF – Général

- ◆ Longueur du code
 - **C-21** Ne pas s'acharner à écrire du code très court (illisible), mais ne pas le décomposer sans fin non plus et ne pas introduire de code inutile.
 - **C-22** Entre code élaboré et rustique, choisir la compréhensibilité et la simplicité.

Note : le compromis entre code trop long ou trop court, entre rustique et sophistiqué, pourrait s'appeler **élégance**. On s'entend généralement sur le fait que la solution la plus élégante est celle qui est **la plus générale** (englobe tous les cas particuliers), qui est **la plus facile à étendre**, qui demande **le moins de ressources** tout en étant **efficace**, et qui malgré tout cela reste **très lisible** !
- ◆ Attention : Transgression des règles
 - Signaler de façon visible toutes les déviations afin qu'on ne les identifie pas comme des erreurs.



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



35

© AMI - Reproduction interdite sans l'autorisation de l'auteur



Guide de Style INSA-IF – Général

- ◆ Mise au point et Test
 - **M-1** Tester séparément chacune des parties du programme en réalisant si besoin un programme de test dédié, puis tester l'ensemble. Apporter un maximum de soins à la construction du jeu d'essai.
C'est aussi une raison pour laquelle en conception il est intéressant d'éliminer au maximum les cas particuliers en dégageant des règles générales. Les cas particuliers ont l'inconvénient de demander des tests spécifiques.
- ◆ Erreur courante
 - **M-2** Interpréter correctement les messages : un message non élucidé correspond probablement à une erreur.



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



36

© AMI - Reproduction interdite sans l'autorisation de l'auteur

Programmation Orientée Objet – C++



PLAN du COURS



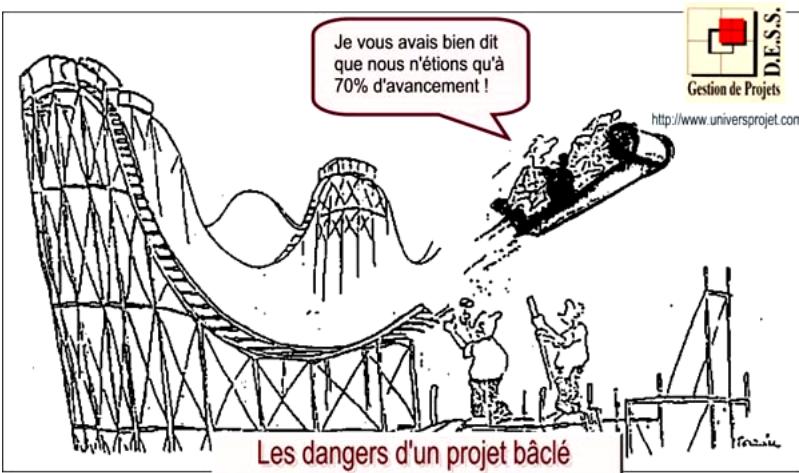
- Naufrage**
 - Programmation Orientée Objet
 - Préprocesseur
 - ◆ Directives #
 - ◆ Directive `#include`
 - ◆ Directive `#define`
 - ◆ Compilation Conditionnelle
 - ◆ Guide de Style
 - Données et Types
 - Expressions
 - ...
- Frisson**
- Bricolage**

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur

37

Préprocesseur



Je vous avais bien dit que nous n'étions qu'à 70% d'avancement !

D.E.S.S.
Gestion de Projets
<http://www.universprojet.com>

Les dangers d'un projet bâclé

<http://www.projetsinformatiques.com>

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur

38



Directives

- Liste (non exhaustive) des directives du préprocesseur
 - ◆ **#include** : inclusion d'un fichier
 - ◆ **#define** : définit une macro ou un identificateur de préprocesseur
 - ◆ **#undef** : annule une définition précédemment faite avec **#define**
 - ◆ **#if** : teste une expression constante entière
 - **#if defined** ou **#ifdef** : teste si un identificateur est défini
 - **#if ! defined** ou **#ifndef** : teste si un identificateur n'est pas défini
 - ◆ **#endif** : fin d'une clause **#if** (ou **#ifdef** ou **#ifndef**)
 - ◆ **#elif** : clause **#else** suivie immédiatement d'une clause **#if**
 - ◆ **#else** : clause sinon dans une clause **#if**
 - ◆ **#error** : permet d'afficher un message (typiquement une erreur) sur **stderr** au moment de la compilation
 - ◆ **#pragma** : définit un comportement spécifique du préprocesseur
 - ◆ ...



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

39



Directive #include

- Principe
 - ◆ Remplacement de la directive par le contenu complet du fichier spécifié
- Rôle essentiel
 - ◆ Importation, depuis un fichier source, dans tout module qui en a besoin, des définitions de classes, de constantes, de types et des déclarations de fonctions, de variables...
- Syntaxe
 - ◆ **#include <CheminDAccès>**
 - À partir des répertoires standard (sous Linux, **/usr/include** ou **/usr/include/sys** ou ...)
 - À partir des répertoires spécifiés derrière l'option **-I** de la commande de compilation **g++**
 - ◆ **#include "CheminDAccès"**
 - À partir du répertoire de travail



© INSA - Reproduction interdite sans l'autorisation de l'auteur

40



Directive #define

■ Rôles essentiels Peu utile en C++

- ◆ Attribution d'un nom à une constante fondamentale
- ◆ Définition d'un identificateur pour la compilation conditionnelle
- ◆ Substitution avec paramètres

■ Syntaxe

- ◆ **#define identificateur [chaîne]**
 - chaîne peut être multi-lignes (utilisation du caractère '\')
- ◆ **#define macro([p₁, p₂, ...]) [chaîne]**
 - chaîne peut (doit) faire référence aux différents paramètres p_i

```
#define Afficher( a , b ) std::cout << " ( " << ( a ) \
<< " , " << ( b ) \
<< " ) " << endl;
```

⇒ **Attention** : l'utilisation de parenthèses pour les paramètres est vivement recommandée dans la chaîne de substitution (priorité des opérateurs)

⇒ **Attention** aux effets de bord

© MM - Reproduction interdite sans l'autorisation de l'auteur



Directive #define

■ Exemples de macros à effets de bord...

```
// ❶ Mauvaise définition de macro : problème de parenthésage (acte 1)
#define MUL(a,b) a * b
MUL ( 2+3, 5+9 ); ⇔ 2+3 * 5+9; ⇔ 2+ (3 * 5) +9;
// d'où, un résultat inattendu de 26 au lieu de 70
#define MUL(a,b) (a) * (b) // réponse au problème

// ❷ Mauvaise définition de macro : problème de parenthésage (acte 2)
#define ADD(a,b) (a) + (b)
ADD ( 2, 3 ) * 5; ⇔ (2) + (3) * 5; ⇔ (2) + ((3) * 5);
// d'où, un résultat inattendu de 17 au lieu de 25
#define ADD(a,b) ((a) + (b)) // réponse au problème

// ❸ Expression à l'appel de la macro
#define MIN(a,b) ( (a) < (b) ? (a) : (b) )
MIN ( f(3), 5 ); ⇔ ( (f(3)) < (5) ? (f(3)) : (5) );
// d'où, un double appel possible de la fonction f, si f(3) < 5
MIN ( ++i, j ); ⇔ ( (++i) < (j) ? (++i) : (j) );
// d'où, une double incrémentation possible de i
```



Compilation Conditionnelle

Rôles essentiels

- ◆ Éviter les inclusions multiples de fichiers
- ◆ Aider à la mise au point
- ◆ Gérer des versions de code

Syntaxe

```

#if expression constante
    Phrases C++ ou directives
    [ #elif expression constante
        Phrases C++ ou directives ] ...
    [ #else ←
        Phrases C++ ou directives ]
#endif

```

⇒ De 0 à n occurrences

⇒ De 0 à 1 occurrence



© INSA - Reproduction interdite sans l'autorisation de l'auteur

43



Guide de Style INSA-IF – Préprocesseur

- ◆ Présentation
 - **T-12** Placer les **#** des directives du préprocesseur en première colonne afin qu'ils soient nettement visibles (pas d'indentation).
- ◆ Commentaires
 - **T-20** Lorsque le **#endif** (ou le **#else**) est éloigné du **#if**, préciser la correspondance par un commentaire.
- ◆ Constantes et macro-définitions
 - **O-19** Pour définir les constantes, préférer au préprocesseur (**#define**) les formes **const** et **enum** dont la portée est plus facile à contrôler (Cf. Données et Types).
 - **E-16** Ne pas utiliser les macro-définitions et les remplacer par des méthodes en ligne (**inline**). Les macro-définitions posent des problèmes lorsque des expressions sont passées en paramètre et leur portée ne peut pas être contrôlée aussi finement que celle des méthodes.



© INSA - Reproduction interdite sans l'autorisation de l'auteur

44

Programmation Orientée Objet – C++

PLAN du COURS



Naufrage

- ...
- Préprocesseur
- Données et Types
 - ◆ Données C++
 - ◆ Types C++
 - ◆ Types Primitifs
 - ◆ Type `void` et énuméré
 - ◆ Types Dérivés
 - ◆ Types Définis par l'Utilisateur
 - ◆ Guide de Style
- Expressions
- ...



Divin ?



Bricolage

© MM - Reproduction interdite sans l'autorisation de l'auteur

45

INSA

Données et Types



MOTOR DESIGNS

Gestion de Projets

http://www.universprojet.com

Mieux vaut bien définir son besoin!

http://www.projetsinformatiques.com

© MM - Reproduction interdite sans l'autorisation de l'auteur

46

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON



Données C++

- Toute valeur manipulée dans un programme C++, variable (objet) ou constante
- Déclaration : 4 parties suivies d'un ';' :
 - ◆ Un descripteur ou classe d'allocation : optionnel
 - `extern, static, auto, register, const...`
 - `virtual...`
 - ◆ Un type
 - ◆ Un déclarateur : composé de...
 - Un nom : le choix des noms est un art ! (définition classique)
 - Un opérateur de déclaration : optionnel
 - ◆ Un initialiseur : optionnel et fonction du type et du déclarateur
- Définition : toujours unique
 - ◆ Déclaration + réservation de mémoire
 - ◆ Choix entre définition / déclaration : fonction du descripteur

Opérateur	Rôle	Position
<code>*</code>	Pointeur	Préfixe
<code>* const</code>	Pointeur constant	Préfixe
<code>&</code>	Référence	Préfixe
<code>[...]</code>	Tableau	Postfixe
<code>(...)</code>	Fonction	Postfixe

© AMI - Reproduction interdite sans l'autorisation de l'auteur



Types C++

- Type
 - ◆ Représentation physique (place mémoire)
 - ◆ Évaluation possible de la taille avec `sizeof`
 - ◆ Ensemble des valeurs possibles
 - ◆ Ensemble des opérations possibles et leur interprétation
- Les différents types
 - ◆ Un type booléen (`bool`)
 - ◆ Des types caractère (tels que `char`)
 - ◆ Des types entier (tels que `int`)
 - ◆ Des types virgule flottante (tels que `double`)
 - ◆ Des types énumérés pour un ensemble de valeurs (`enum`)
 - ◆ Des types construits à partir de ces types : pointeur, tableau, référence, structure, union et classe
 - ◆ Un type `void` pour indiquer l'absence d'information

⇒ `sizeof (type) OU sizeof variable`

© AMI - Reproduction interdite sans l'autorisation de l'auteur





Type Primitif Booléen

- Booléen : **bool** n'existe pas en C
 - ◆ 2 valeurs possibles : **true** et **false**
 - ◆ Résultat des opérations logiques
 - ◆ Par définition, **true** donne **1**, après conversion en entier
 - ◆ Par définition, **false** donne **0**, après conversion en entier
 - ◆ A l'inverse, les entiers non nuls correspondent à **true** et 0 correspond à **false**
- Exemples de données booléennes

```
// Définition / déclaration de données de type primitif booléen
bool fin;
bool nonTrouve = true; // ou bool nonTrouve ( true );
static bool premier = false;
bool *dernier;
extern bool Rechercher ( int );
```



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



49

© INSA - Reproduction interdite sans l'autorisation de l'auteur



Types Primitifs Caractère

- Caractère : **char**
 - ◆ **char** : un caractère appartenant au jeu de caractères de la machine
 - ◆ Chaque caractère constant possède une valeur entière dans un jeu de caractères donné
 - Interprétation -127 à 127 (caractère signé)
 - Interprétation 0 à 255 (caractère non signé)
 - Choix signé / non signé imposé par l'implémentation
 - Problème de portabilité
 - Utilisation possible de **signed** ou **unsigned** devant **char**
- Autre caractère : **wchar_t**
 - ◆ **wchar_t** : un caractère appartenant à un jeu de caractères plus étendu tel qu'Unicode
 - Taille de ce type définie par l'implémentation (souvent 16 bits)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



50

© INSA - Reproduction interdite sans l'autorisation de l'auteur



Types Primitifs Caractère

Constante littérale caractère

- ◆ Encadrée d'apostrophes : 'a', '0' ou '!' !
- ◆ Ne pas confondre 'a' (caractère a) avec "a" (chaîne a)
- ◆ Constantes littérales caractères spéciales
 - Nouvelle ligne : '\n'
 - Tabulation horizontale : '\t'
 - Retour-arrière : '\b'
 - Alerte : '\a'
 - Guillemet simple : '\''
 - Guillemet double : '\"'
 - Nombre octal : '\ooo'
 - Nombre hexadécimal : '\xhhh'

Exemples de données caractère



```
// Définition / déclaration de données de type primitif caractère
const char RC = '\n'; // ou const char RC ( '\n' );
char c = '\60'; // Nombre octal => '0'
char car = '\x05f'; // Nombre hexadécimal => '_'
wchar_t *nom;
static char *SousChaine ( const char * );
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur



51



Types Primitifs Entier

Entier : int, long, short, signed et unsigned

- ◆ 3 formes : int ordinaire, signed int et unsigned int
- ◆ 3 tailles : short int, int ordinaire et long int
- ◆ Équivalence
 - short ≡ short int
 - long ≡ long int
 - unsigned ≡ unsigned int
 - signed ≡ signed int (synonyme de int)

```
// Définition / déclaration de données de type primitif entier
const int nbRepetition = 5;
const int droits = 064; // Nombre octal => 'rw- rw- r--'
long taille = 30L; // ou long taille ( 30L );
long vecteur [ 10 ];
unsigned int *matrice;
static int Maximum ( const int [ ] );
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur

Constantes littérales entier

- ◆ 4 apparences
 - Constante littérale décimale
 - Constante littérale octale : débute par un 0
 - Constante littérale hexadécimale : débute par 0 suivie de x ou X
 - Constante littérale caractère
- ◆ 2 suffixes
 - u ou U : constante littérale unsigned (par exemple, 63U)
 - l ou L : constante littérale long (par exemple, 0L)

⇒ Programmation système (bits)



52



Types Primitifs Virgule Flottante

- Virgule flottante : **double**, **float** et **long double**
 - ◆ 3 tailles : **float** (simple précision), **double** (double précision) et **long double** (précision étendue)
- Constantes littérales *virgule flottante*
 - ◆ Par défaut de type **double**
 - ◆ 2 suffixes
 - **f** ou **F** : constante littérale virgule flottante de type **float**
 - **l** ou **L** : constante littérale virgule flottante de type **long double**
- Exemples de données *virgule flottante*

```
// Définition / déclaration de données de type primitif virgule flottante
const float pi = 3.1415F;
const double prix = 9.99;
extern long double matrice [ 10 ] [ 10 ];
double *intensite;
```



Type void

- **void** : type fondamental du C++
 - ◆ Il n'existe pas de variable de type **void**
 - ◆ 2 utilisations possibles
 - Spécifier qu'une fonction ne renvoie aucune valeur
 - Type de base pour les pointeurs d'objets de type inconnu

```
// Utilisation possible de void
void nombre; // Erreur de compilation
void *pObjet; // Pointeur sur un objet de type inconnu
void Afficher ( const char * ); // Fonction = Procédure
```

⇒ Le moindre détail compte ! (opérateur de déclaration *****)

⇒ Pseudo-type renvoyé par la fonction **Afficher**



Types Énumérés

■ enum

- ◆ Définition d'un nouveau type ne pouvant prendre que des valeurs désignées par des noms (constantes nommées)
- ◆ Sorte d'entier \Rightarrow utilisation très proche
- ◆ Constantes nommées de l'énumération : énumérateurs
- ◆ Une énumération peut être nommée
- ◆ Valeurs attribuées aux énumérateurs
 - Par défaut, rang dans l'énumération en commençant à 0
 - Expression constante d'un type intégral
- ◆ Utilisation des types énumérés

```

⇒ COEUR ≡ -10
⇒ CARREAU ≡ -9
⇒ PIQUE ≡ 5
⇒ TREFLE ≡ 6
  
```

```

// Définition de nouveaux types énumération
enum FeuTricolore { ROUGE, ORANGE, VERT };
enum Carte { COEUR = -10, CARREAU, PIQUE = 5, TREFLE };
  
```

© MM - Reproduction interdite sans l'autorisation de l'auteur



Types Dérivés

■ Tableau

- ◆ Collection ordonnée de **N** (expression constante) éléments...
- ◆ De même type : primitif, dérivé, défini par l'utilisateur...
- ◆ Accessibles par leur rang : de 0 à **N** – 1
- ◆ Opérateur de déclaration : [...]
- ◆ Tableau à plusieurs dimensions : tableaux de tableaux
- ◆ Initialiseur possible : = { ... }

```

// Définition de données de type tableau
const int TAILLE = 10;
int vecteur [ TAILLE ]; // Collection ordonnée de TAILLE éléments int,
                       // accessibles par leur rang : de 0 à TAILLE - 1
float image [ 640 ] [ 480 ];
                       // image est un tableau de 640 tableaux de 480 flottants
char voyelle [ 6 ] = { 'a', 'e', 'i', 'o', 'u', 'y' };
                       // initialisation du tableau voyelle de 6 caractères
  
```

© MM - Reproduction interdite sans l'autorisation de l'auteur



Types Dérivés

■ Tableau

- ◆ Accès à la valeur de l'ensemble des éléments impossible
- ◆ Accès à un élément par l'opérateur d'indexation : [...]
- ◆ Utilisation d'une expression entière pour définir le rang
- ◆ Classe standard **vector** : généralisation du type tableau
⇒ nombre d'éléments du tableau a priori variable
- ◆ Type chaîne de caractères : n'existe pas en C / C++
 - Représentation possible sous forme d'un tableau de type **char**
 - En C++, utilisation préférable de la classe standard **string**

```
// Accès aux données de type tableau
int vecteur [ 10 ]; // Collection ordonnée de 10 éléments int,
int x;
int i = 9; // ou int i ( 9 );
x = vecteur [ 0 ]; // Accès au premier élément du tableau
vecteur [ i ] = x; // Accès au i + 1ère élément du tableau
```



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

57



Tableau et Constante Littérale Chaîne

■ Constante littérale chaîne

- ◆ Séquence de caractères encadrée de guillemets doubles ("...")
- ◆ Un caractère **de plus** : le caractère '\0' dont la valeur est 0
- ◆ Type d'une constante littérale chaîne : *tableau du nombre approprié de caractères const*
- ◆ Cas particulier : la chaîne vide ""
- ◆ Compatibilité
 - Constante littérale chaîne aussi de type **char ***
 - Modification impossible à partir d'un tel pointeur ⇒ passage obligatoire par un tableau de caractères

```
// Constante littérale chaîne de caractères
"Bonjour"
"Bonjour" == Bonjour\0 d'où sizeof ( "Bonjour" ) = 8
Type de "Bonjour" : const char [ 8 ];
```

⇒ Attention : **char * = const char [9]**
⇒ conversion implicite !

```
// Compatibilité avec les anciennes versions du C et du C++
char *message = "Monsieur"; // Initialisation tolérée
message [ 0 ] = 'm'; // Erreur à l'exécution : affectation sur const (indéfini)

char message [ ] = "Monsieur"; // Initialiseur de tableau non const
message [ 0 ] = 'm'; // Affectation correcte
```

© INSA - Reproduction interdite sans l'autorisation de l'auteur

Types Dérivés

- Pointeur
 - Pour un type T , T^* est le type "pointeur de T "
 - Une variable de type T^* peut contenir l'adresse d'un objet de type T

```
// Manipulation de pointeurs et d'adresses
char c = 'A'; // type T = char
char *pt; // pt est un pointeur de char
           // Non initialisé ⇒ pointe n'importe où
pt = &c; // pt reçoit l'adresse de la variable c de type char
```

◆ Généralisation possible : T peut être un type "pointeur de $T1$ "

```
// Généralisation de la notion de pointeurs
char * *p1; // p1 est un pointeur de pointeur de char
             // Non initialisé ⇒ pointe n'importe où
p1 = &pt; // p1 reçoit l'adresse de pt de type pointeur de char
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Types Dérivés

- Opérations sur le pointeur
 - Opération fondamentale : l'**indirection**
 - Opérateur $*$: "prendre la valeur de ce qui est actuellement pointé par le pointeur" (déréférence)

```
// Manipulation de pointeurs et d'adresses
char c1 = 'A'; // =char c1 ( 'A' );
char *pt = &c1; // pt est un pointeur de char
               // Initialisé ⇒ pointe la variable c1 de type char
char c2 = *pt; // c2 reçoit la valeur de ce qui est
               // actuellement pointé par pt, c'est-à-dire c1
```

◆ Opérations arithmétiques : $+$, $-$, $++$ et $--$

- Déplacement de la taille de l'objet pointé
⇒ `sizeof (T)`

◆ $T^* \text{pt} = \text{nullptr}$; indique que le pointeur pt ne fait pas référence à un objet (macro **NULL** du C)

◆ Attention à la signification de l'affectation $pt1 = pt2$;

- Affectation des pointeurs (surface) \neq affectation des objets pointés (profondeur)

© AMI - Reproduction interdite sans l'autorisation de l'auteur

Types Dérivés



Pointeur et tableau

- ◆ Nom du tableau = adresse de son 1^{er} élément
- ◆ Pointeurs et tableaux sont étroitement liés

```
// Pointeurs et Tableaux
int v [ ] = { 1, 2, 3, 4 };
int *pt = v; // pt est un pointeur de int
            // initialisé ⇒ pointe le premier élément de v
            // équivalent à int *pt = &v[0]; car v ≡ &v[0]
*pt++ = 9; // *pt = 9; suivi de pt = pt + 1;
            // est équivalent à v[0] = 9; ou pt[0] = 9;
int x = pt[ -1 ]; // x reçoit la valeur de l'élément
                    // précédent celui qui est actuellement pointé par pt
                    // équivalent à x = *(pt - 1);
// Attention aux différences de taille (architecture 32 bits)
sizeof v = 16 // taille tableau * sizeof (int)
sizeof pt = 4 // sizeof ( int * )
sizeof x = 4 // sizeof ( int )
```

Mémoire

0	1	2	3	N-1	N
9					
@x		@pt		@v	

Diagramme de la mémoire montrant l'attribution d'adresses à un tableau v et à un pointeur pt. L'adresse de v est @v, l'adresse de pt est @pt, et l'adresse de l'élément 9 dans v est @x. Les adresses 0, 1, 2, 3 et N-1 sont également indiquées.

© AMI - Reproduction interdite sans l'autorisation de l'auteur

Types Dérivés



Pointeur et `const`

- ◆ 4 syntaxes différentes

- ❶ `T * pt;`
// pt pointeur non constant sur un objet pointé non constant
// ⇒ le pointeur et l'objet pointé peuvent être modifiés
- ❷ `T * const ptc = ...;`
// ptc pointeur constant sur un objet pointé non constant
// ⇒ le pointeur ne peut pas être modifié et doit être initialisé
// ⇒ l'objet pointé peut être modifié
- ❸ `const T * cpt; ou T const * cpt;`
// cpt pointeur non constant sur un objet pointé constant
// ⇒ le pointeur peut être modifié
// ⇒ l'objet pointé ne peut pas être modifié
- ❹ `const T * const cptc = ...; ou T const * const cptc = ...;`
// cptc pointeur constant sur un objet pointé constant
// ⇒ le pointeur et l'objet pointé ne peuvent pas être modifiés

Types Dérivés

■ Référence

- ◆ N'existe pas en C / ANSI
 - ◆ Opérateur de déclaration : &
 - ◆ Fournit un autre nom pour un objet (synonyme)
 - ◆ Est toujours initialisée à sa définition
 - ◆ **Attention** : aucune opération ne s'applique directement à une référence

■ Usages

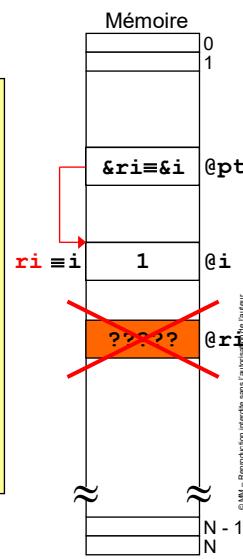
- ◆ Passage de paramètres par référence
 - ◆ Valeur de retour par référence pour les fonctions
 - ◆ Surcharge des opérateurs



Types Dérivés

■ Référence : exemples

```
// Référence – utilisation
int i = 0;
int &ri = i; // définition de la référence ri
              // initialisé ⇒ ri et i font maintenant référence au
              // même int (ri est un autre nom pour i)
ri++; // ce n'est pas la référence ri qui est incrémentée, au
       // contraire, ++ s'applique à un int qui se trouve être i
int *pt = &ri; // permet d'obtenir un pointeur sur l'objet
               // indiqué par la référence ri, c'est-à-dire i
               // adresse de ri = adresse de i (&ri = &i)
int &rx; // erreur : initialiseur manquant
extern int &ry; // ok : déclaration de ry et ry est défini
                 // et initialisé dans une autre partie du code (autre
                 // fichier)
```





Types Définis par l'Utilisateur

■ Structure

- ◆ Agrégat d'éléments de types quelconques, accessibles par leur nom, sans opération associée, ni initialisation
- ◆ Équivalent à une classe où toutes les méthodes sont publiques par défaut
- ◆ Utilisée en C mais peu utilisée en C++ ⇒ classe

```
// Définition d'une structure
struct Etudiant { // définition du type Etudiant
    char nom [ MAX ];
    int numero;
    enum { IF3, IF4, IF5 } annee;
};

Etudiant un3IF = { "Pierre", 2005001, Etudiant::IF3 };
Etudiant *leMeme = &un3IF; // leMeme est un pointeur d'Etudiant
```

⇒ Type quelconque
 ⇒ Champ de la structure
 ⇒ Initialiseur possible = { ... } comme pour le type dérivé tableau
 ⇒ Attention : ne pas oublier le ;
 ⇒ Attention : 3IF, 4IF et 5IF ne seraient pas des constantes nommées licites

© MM - Reproduction interdite sans l'autorisation de l'auteur


Types Définis par l'Utilisateur

■ Utilisation d'une structure

- ◆ Accès au champ
 - . à partir d'une variable de type structure
 - -> à partir d'un pointeur sur un type structure

```
// Utilisation de la structure Etudiant
Etudiant un3IF = { "Pierre", 2005001, Etudiant::IF3 };
Etudiant *leMeme = &un3IF; // leMeme est un pointeur d'Etudiant
Etudiant unAutre;

cout << un3IF.nom[ 0 ] << endl; // initiale du nom
cout << leMeme->annee << endl; // accès au champ année de la
                                // structure actuellement pointée par leMeme
cout << (*leMeme).annee << endl; // notation équivalente
unAutre = un3IF; // affectation de structures licite
```

⇒ Attention aux priorités entre les opérateurs * et .
 ⇒ Utilisation obligatoire des parenthèses



Types Définis par l'Utilisateur : `typedef`

Définition de type

- ◆ Syntaxe : 3 parties suivie d'un ';' :
 - Mot-clé `typedef`
 - Un type existant
 - Un déclarateur : composé de...
 - Un nouveau nom de type
 - Un opérateur de déclaration : optionnel (cf. données)
- ◆ Type synonyme, souvent un raccourci pratique
- ◆ Astuce : de la variable au type...

Syntaxe de base

```
typedef type déclarateur;
```

```
// Type synonyme – raccourci
unsigned char *pt; // définition classique

typedef unsigned char *puc;
puc p1; // raccourci pratique...
// attention au choix du nom du type
p1 = pt; // correct : type synonyme
// puc n'est pas un nouveau type
```

```
// De la variable au type
int vecteur [ 10 ]; // définition de variable

// ajout de typedef en préfixe de la définition
// ⇒ la variable vecteur devient le type vecteur
typedef int vecteur [ 10 ]; // définition de type
vecteur v; // exploitation de la définition de type
```

© INSA – Reproduction interdite sans l'autorisation de l'auteur



67



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



Guide de Style INSA-IF – Donnée

Noms

- **T-1** Tous les objets doivent porter un nom rappelant leur signification (`bool trouve;`). Il n'est pas utile que le nom rappelle le type de l'objet (`bool bool1;`).

Le choix d'un nom significatif est reconnu comme l'un des problèmes les plus difficiles qui soit.

C'est aussi le moment où le développeur concrétise l'idée parfois un peu floue qu'il a du rôle de l'objet.
- **T-2** Ne pas donner à un objet le même nom qu'un autre objet du même espace de nommage (attribut ⇔ paramètre).

© INSA – Reproduction interdite sans l'autorisation de l'auteur



68



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



Guide de Style INSA-IF – Donnée

◆ Typographie

- **T-3** Ne pas commencer de nom par un blanc souligné ("_") (recommandations ANSI).
- **T-4** Dans les noms composés de plusieurs mots, mettre une majuscule en tête de chaque mot pour faciliter la lecture.
- **T-5** Écrire les noms de constantes et les valeurs de types énumérés en lettres majuscules.
- **T-6** Commencer les noms des objets publics et des types par une majuscule, et les noms des objets protégés et privés par une minuscule.
- **T-16** Réserver les variables d'une seule lettre pour de simples indices de boucle n'ayant pas de signification particulière ou pour des variables de manœuvre temporaires.



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

69



Guide de Style INSA-IF – Donnée

◆ Présentation

- **T-8** Déclarer une seule variable par ligne et en profiter pour expliciter son rôle (**Attention** : `char * p, q;`).

◆ Structures

- N'utiliser les structures (**struct**) que pour grouper des données, ce qui est assez rare. Si ces données nécessitent des méthodes, ou si elles doivent être initialisées, définir une classe.

◆ Expressions

- Programmer avec le type booléen et non le type entier pris comme booléen (cas du C).

◆ Globaux

- **O-5** Ne pas utiliser de global commun à plusieurs classes.



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

70



Guide de Style INSA-IF – Donnée

- ◆ Constantes et macro-définitions
 - **O-20** Définir les constantes ayant un rapport entre elles dans un `enum`.
 - **O-3** Limiter la portée des constantes au strict nécessaire : mettre dans la réalisation et non dans l'interface les constantes uniquement utilisées dans la réalisation, et déclarer ces constantes `static`.
 - **T-13** Définir toutes les constantes du programme par des littéraux.
- ◆ Portabilité et représentation des objets
 - **Po-1** Les objets peuvent avoir une représentation différente suivant les machines hôtes ou selon les options de compilation choisies : ne faire aucune hypothèse sur leur mode de représentation ni leur longueur.



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

71



Guide de Style INSA-IF – Donnée

- ◆ Erreurs courantes
 - **E-2** Les indices d'un tableau de dimension `n` varient de 0 à `n-1`.
 - **E-8** La longueur de la zone nécessaire pour stocker la chaîne `Chaine` est de `strlen (Chaine) + 1`.
 - **E-5** Pour comparer deux chaînes de caractères, vous devez comparer les contenus et non les adresses.
 - Pour copier une chaîne de caractères dans une autre, vous devez copier les contenus et non les adresses.
 - Tous les pointeurs et variables doivent être initialisés avant leur utilisation.



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

72



Guide de Style - Conseils

■ Conseils (C++ - B. Stroustrup)

- ◆ Ne faites pas de supposition inutile sur la valeur numérique des caractères.
- ◆ Ne faites pas de supposition inutile concernant la taille des entiers.
- ◆ Ne faites pas de supposition inutile concernant la plage de valeurs des types virgule-flottante.
- ◆ Choisissez de préférence un `int` ordinaire à un type `short int` ou `long int`.
- ◆ Choisissez de préférence un `double` à un `float` ou à un `long double`.
- ◆ Choisissez de préférence un `char` à un `signed char` ou à un `unsigned char`
- ◆ Évitez l'arithmétique non signée.
- ◆ Envisagez avec méfiance les conversions virgule-flottante vers entier, `signed` vers `unsigned` (et réciproque) et vers un type plus petit, comme `int` vers `char`

© INSA - Reproduction interdite sans l'autorisation de l'auteur



Guide de Style - Conseils

■ Conseils (C++ - B. Stroustrup)

- ◆ Évitez toute arithmétique fantaisiste avec les pointeurs.
- ◆ Prenez garde à ne pas écrire au-delà des limites d'un tableau.
- ◆ Utilisez `nullptr` plutôt que 0 ou `NULL`.
- ◆ Préférez la classe `vector` aux tableaux de style C.
- ◆ Préférez la classe `string` aux tableaux de caractères terminés par '\0'.
- ◆ Minimisez l'utilisation des arguments références.
- ◆ Évitez l'emploi de `void *`.
- ◆ Évitez l'emploi de constantes littérales dans le code. Préférez l'utilisation et la définition de constantes symboliques (`const`).

© INSA - Reproduction interdite sans l'autorisation de l'auteur

Programmation Orientée Objet – C++

PLAN du COURS



Naufrage

- ...
- Données et Types
- Expressions
 - ◆ Définition
 - ◆ Opérateurs sur Entiers, sur Virgules-flottantes
 - ◆ Opérateurs sur Tableaux et Pointeurs
 - ◆ Incrémentation / Décrémentation
 - ◆ Expression Conditionnelle
 - ◆ Opérateur `new / delete`
 - ◆ Guide de Style
- Fonctions – Instructions
- ...



Davin ?



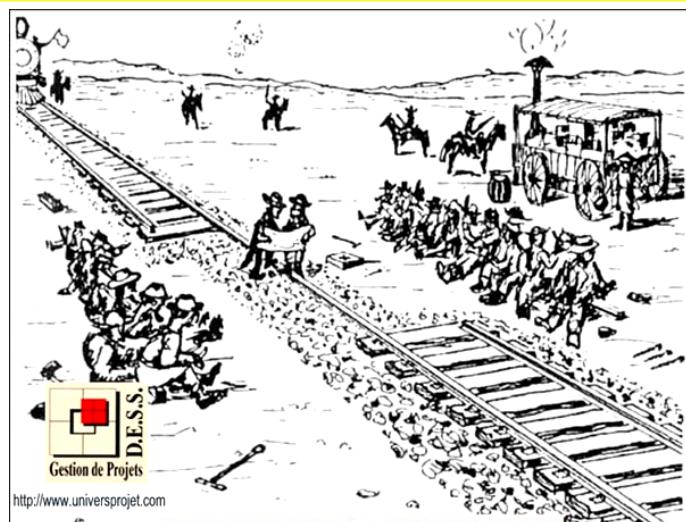
Bricolage

© MM - Reproduction interdite sans l'autorisation de l'auteur

INSA

75

Expressions



DESS. Gestion de Projets
http://www.universprojet.com

Savez vous communiquer dans vos projets ?

http://www.projetsinformatiques.com

© MM - Reproduction interdite sans l'autorisation de l'auteur

INSA

76



Expressions

- Combinaison d'opérandes à l'aide d'opérateurs
- Opérande
 - ◆ Expression (définition récursive)
 - ◆ Variable ou constante nommée ou constante littérale
 - ◆ De type primitif, dérivé ou défini par l'utilisateur
- Opérateur
 - ◆ Hiérarchisé \Rightarrow priorité dans l'ordre d'évaluation
 - En cas d'inégalité, l'opération prioritaire est effectuée en premier
 - Pour casser les priorités, on utilise les (...)
 - ◆ Prédéfini pour les types primitifs
 - ◆ Défini par l'utilisateur
- Conversion
 - ◆ Conversion implicite de certains opérandes dans une expression, lors de l'évaluation

Dans l'expression a/b , l'opérateur / a une signification contextuelle qui dépend du type des objets a et b :
 - si a et b sont des entiers, / désigne la division entière donnant un résultat entier ;
 - si a et b sont des réels, / désigne la division réelle donnant un résultat réel ;
 - si un des objets est de type `T` défini par l'utilisateur, celui-ci peut donner un sens à cette expression en définissant (surchargeant) l'opérateur / pour cette utilisation.

© IMI – Reproduction interdite sans l'autorisation de l'auteur



Opérateurs

- Priorité de l'opérateur (Cf. tableau)
 - ◆ Présentation dans l'ordre décroissant des priorités
- ```
// Exemple de priorité
a + b * c; est évaluée ainsi a + (b * c);
// car * possède une priorité plus forte que +
```
- Associativité de l'opérateur (Cf. tableau)
  - ◆ **Droite** : indique que l'évaluation s'effectue de droite à gauche
  - ◆ **Gauche** : indique que l'évaluation s'effectue de gauche à droite

`// Exemple d'associativité
// ① Associativité à droite – opérateur =
a = b = c; est équivalent à a = ( b = c );
// ② Associativité à gauche – opérateur +
a + b + c; est équivalent à ( a + b ) + c;
// ③ Priorité et associativité à droite – opérateur * et ++
*p++; est équivalent à * (p++) ; et non (*p) ++;`



© IMI – Reproduction interdite sans l'autorisation de l'auteur



## Opérateurs C++

| P  | A      | opérateur        | description                         | exemple                      |
|----|--------|------------------|-------------------------------------|------------------------------|
| 18 | droite | ::               | Sélection de contexte global        | ::nom                        |
| 18 | gauche | ::               | Sélection de contexte de classe     | nomClasse::membre            |
| 17 | gauche | .                | Sélecteurs de membres               | objet.membre                 |
| 17 | gauche | ->               | Sélecteurs de membres               | pointeur->membre             |
| 17 | gauche | [ ]              | Index de tableau                    | pointeur[expr]               |
| 17 | gauche | ( )              | Appel de fonction                   | expr(liste_expr)             |
| 17 | gauche | ( )              | Construction de type                | type(liste_expr)             |
| 17 | droite | ++               | Post incrémentation                 | lvalue++                     |
| 17 | droite | --               | Post décrémentation                 | lvalue--                     |
| 17 | droite | typeid           | Identification du type              | typeid(type)                 |
| 17 | droite | typeid           | Identification du type d'exécution  | typeid(expr)                 |
| 17 | droite | dynamic_cast     | Conversion contrôlée d'exécution    | dynamic_cast<type>(expr)     |
| 17 | droite | static_cast      | Conversion contrôlée de compilation | static_cast<type>(expr)      |
| 17 | droite | reinterpret_cast | Conversion non contrôlée            | reinterpret_cast<type>(expr) |
| 17 | droite | const_cast       | Conversion const                    | const_cast<type>(expr)       |
| 16 | droite | sizeof           | Taille de l'objet                   | sizeof expr                  |
| 16 | droite | sizeof           | Taille du type en octets            | sizeof(type)                 |

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Opérateurs C++

| P  | A      | opérateur  | description                             | exemple                        |
|----|--------|------------|-----------------------------------------|--------------------------------|
| 16 | droite | ++         | Pré incrémentation                      | ++lvalue                       |
| 16 | droite | --         | Pré décrémentation                      | --lvalue                       |
| 16 | droite | ~ ou compl | Complément binaire, bit à bit           | ~expr                          |
| 16 | droite | ! ou not   | Non logique                             | !expr                          |
| 16 | droite | -          | Opérateur moins unaire                  | -expr                          |
| 16 | droite | +          | Opérateur plus unaire (superflu)        | +expr                          |
| 16 | droite | &          | Opérateur adresse                       | &lvalue                        |
| 16 | droite | *          | Déférancement de pointeur, indirection  | *expr                          |
| 16 | droite | new        | Allocation dynamique de mémoire         | new type                       |
| 16 | droite | new        | Allocation et initialisation de mémoire | new type (liste_expr)          |
| 16 | droite | delete     | Dé-allocation de mémoire (destruction)  | delete pointeur                |
| 16 | droite | delete     | Dé-allocation de tableau (destruction)  | delete [ ] pointeur            |
| 16 | droite | ( )        | Conversion de type, casting             | (type)expr                     |
| 15 | gauche | . *        | Sélecteurs de pointeurs de membres      | objet.*pointeur-vers-membre    |
| 15 | gauche | ->*        | Sélecteurs de pointeurs de membres      | pointeur->pointeur-vers-membre |

© INSA - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON





## Opérateurs C++

⇒ Attention à ne pas faire de confusion avec les opérateurs surchargés << et >> de `iostream` (surcharge d'opérateurs)

| P  | A      | opérateur                 | description                                  | exemple                         |
|----|--------|---------------------------|----------------------------------------------|---------------------------------|
| 14 | gauche | *                         | Opérateur arithmétique de multiplication     | <code>expr * expr</code>        |
| 14 | gauche | /                         | Opérateur arithmétique de division           | <code>expr / expr</code>        |
| 14 | gauche | %                         | Opérateur arithmétique de modulo (reste)     | <code>expr % expr</code>        |
| 13 | gauche | +                         | Opérateur arithmétique d'addition            | <code>expr + expr</code>        |
| 13 | gauche | -                         | Opérateur arithmétique de soustraction       | <code>expr - expr</code>        |
| 12 | gauche | <<                        | Opérateur de décalage de bits à gauche       | <code>expr &lt;&lt; expr</code> |
| 12 | gauche | >>                        | Opérateur de décalage de bits à droite       | <code>expr &gt;&gt; expr</code> |
| 11 | gauche | <                         | Opérateur relationnel inférieur à            | <code>expr &lt; expr</code>     |
| 11 | gauche | <=                        | Opérateur relationnel inférieur ou égal à    | <code>expr &lt;= expr</code>    |
| 11 | gauche | >                         | Opérateur relationnel supérieur à            | <code>expr &gt; expr</code>     |
| 11 | gauche | >=                        | Opérateur relationnel supérieur ou égal à    | <code>expr &gt;= expr</code>    |
| 10 | gauche | ==                        | Opérateur relationnel égal à                 | <code>expr == expr</code>       |
| 10 | gauche | != ou <code>not_eq</code> | Opérateur relationnel non égal à (différent) | <code>expr != expr</code>       |
| 9  | gauche | & ou <code>bitand</code>  | Opérateur ET binaire, bit à bit (AND)        | <code>expr &amp; expr</code>    |
| 8  | gauche | ^ ou <code>xor</code>     | Opérateur OU exclusif, bit à bit (XOR)       | <code>expr ^ expr</code>        |
| 7  | gauche | ou <code>bitor</code>     | Opérateur OU inclusif, bit à bit (OR)        | <code>expr   expr</code>        |

© IMI - Reproduction interdite sans l'autorisation de l'auteur



## Opérateurs C++

| P | A      | opérateur                                   | description                                   | exemple                           |
|---|--------|---------------------------------------------|-----------------------------------------------|-----------------------------------|
| 6 | gauche | <code>&amp;&amp;</code> ou <code>and</code> | Opérateur ET logique                          | <code>expr &amp;&amp; expr</code> |
| 5 | gauche | <code>  </code> ou <code>or</code>          | Opérateur OU inclusif logique                 | <code>expr    expr</code>         |
| 4 | gauche | ? :                                         | Expression conditionnelle (IF arithmétique)   | <code>expr ? expr : expr</code>   |
| 3 | droite | =                                           | Affection simple                              | <code>lvalue = expr</code>        |
| 3 | droite | *=                                          | Multiplication suivie d'une affectation       | <code>lvalue *= expr</code>       |
| 3 | droite | /=                                          | Division suivie d'une affectation             | <code>lvalue /= expr</code>       |
| 3 | droite | %=                                          | Modulo suivi d'une affectation                | <code>lvalue %= expr</code>       |
| 3 | droite | +=                                          | Addition suivie d'une affectation             | <code>lvalue += expr</code>       |
| 3 | droite | -=                                          | Soustraction suivie d'une affectation         | <code>lvalue -= expr</code>       |
| 3 | droite | <=                                          | Décalage à gauche suivi d'une affectation     | <code>lvalue &lt;= expr</code>    |
| 3 | droite | >=                                          | Décalage à droite suivi d'une affectation     | <code>lvalue &gt;= expr</code>    |
| 3 | droite | <code>&amp;=</code> ou <code>and_eq</code>  | ET bit à bit suivi d'une affectation          | <code>lvalue &amp;= expr</code>   |
| 3 | droite | <code>!=</code> ou <code>or_eq</code>       | OU inclusif bit à bit suivi d'une affectation | <code>lvalue != expr</code>       |
| 3 | droite | <code>^=</code> ou <code>xor_eq</code>      | OU exclusif bit à bit suivi d'une affectation | <code>lvalue ^= expr</code>       |
| 2 | droite | <code>throw</code>                          | Déclenchement d'une exception                 | <code>throw expr</code>           |
| 1 | gauche | ,                                           | Opérateur de séquence                         | <code>expr , expr</code>          |

© IMI - Reproduction interdite sans l'autorisation de l'auteur



## Incrémentation / Décrémentation

- Utilisable sur des variables...
  - ◆ Entières (`char`, `short`, `int` et `long`),
  - ◆ Virgules-flottantes (`float` et `double`)
  - ◆ Pointeurs
- Incrémente ou décrémente d'une **unité** la valeur d'une variable (attention aux pointeurs)
- Utilisation
  - ◆ `x++` : la variable est utilisée avec sa valeur actuelle puis elle est incrémentée (post incrémantation)
  - ◆ `++x` : la variable est incrémentée puis elle est utilisée avec sa nouvelle valeur (pré incrémantation)
  - ◆ `x--` : la variable est utilisée avec sa valeur actuelle puis elle est décrémentée (post décrémantation)
  - ◆ `--x` : la variable est décrémentée puis elle est utilisée avec sa nouvelle valeur (pré décrémantation)



© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Expression Conditionnelle

### Syntaxe

`expr1 ? expr2 : expr3`

⇒ Le tout est une expression

- ◆ Cette écriture désigne une expression dont la valeur est...
  - `expr2`, si la condition `expr1` est **vraie** (`true` ou valeur non nulle)
  - `expr3`, si la condition `expr1` est **fausse** (`false` ou valeur nulle)
- ◆ Cette expression supprime souvent une instruction conditionnelle `if`, plus lourde

```
// Exemple d'utilisation de l'expression conditionnelle
// ❶ Calcul d'un minimum
minimum = (a < b) ? a : b;
// ❷ Gestion d'un pluriel
cout << "J'ai trouvé " << x << " élément"
 << ((x > 1) ? 's' : ' ') << endl;
// Écriture sur le flot de sortie standard de la chaîne :
// "J'ai trouvé 0 élément" (si x est nul)
// "J'ai trouvé 5 éléments" (si x = 5)
```



© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Opérateur de Séquence

- Opérateur ,

- ◆ Priorité la plus basse
- ◆ Évaluation de gauche à droite pour les expressions d'une séquence
- ◆ Utilisation possible dans l'instruction **for**
- ◆ Valeur de l'expression : valeur de la dernière expression de la séquence

```
// Fonctionnement de l'opérateur de séquence ,
x = expr1, expr2, expr3;
// Revient à évaluer expr1, puis expr2, puis expr3 et à
// affecter la valeur de l'évaluation de expr3 à x

// Exemple : affectation et opérateur de séquence ,
a = b + (c = 4 , d += c , c + d);
// est équivalent à c = 4;
// puis d = d + c;
// puis a = b + c + d;
```



## Opérateur new

- Opérateur **new**

- ◆ Réservation dynamique d'une zone de mémoire
- ◆ **new T** : l'opérateur **new** essaie de créer un objet de type **T**
  - Renvoie un pointeur sur l'objet créé, en cas de succès
  - Déclenche, par défaut, une exception **bad\_alloc**, en cas de mémoire épuisée ou renvoie un pointeur nul
- ◆ Création d'un tableau dynamique
  - new T [ expression-entière ]**
  - Renvoie un pointeur sur le premier élément du tableau créé
- ◆ Par défaut, pas d'initialisation de la mémoire renvoyée pour les types primitifs et les structures
- ◆ Pour les instances de classes C++, le constructeur de ces classes est automatiquement appelé lors de leur allocation (initialisation)
- ◆ **Attention** : destruction explicite obligatoire (opérateur **delete**)

```
// Exemple d'utilisation de l'opérateur new
Pile *nouvelle = new Pile; // Crédit d'une pile
// Crédit d'un tableau de caractères
char *message = new char [lgMsg + 1]; // Tableau de caractères
```

⇒ + 1 pour la marque de fin de chaîne '\0' non comptabilisée



## Opérateur `delete`

### ■ Opérateur `delete`

- ◆ `delete` détruit un élément créé par l'opérateur `new`
- ◆ `delete [ ]` est utilisée pour détruire un tableau de type `T` créé par `new T [ ... ]`
  - **Attention** : syntaxe source d'erreur difficile à déceler
- ◆ `delete` sur un pointeur nul est autorisé et sans effet
- ◆ Appel du destructeur, si l'élément est une instance de classe

```
// Exemple d'utilisation de l'opérateur delete
delete nouvelle; // Destruction de l'objet (instance de Pile)
 // ⇒ Appel du destructeur de la classe Pile

// Destruction d'un tableau de caractères
delete [] message; // Destruction du tableau message
delete message; // Erreur : destruction incomplète et incorrecte
```

© AM - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© AM - Reproduction interdite sans l'autorisation de l'auteur



## Guide de Style INSA-IF – Expression

- ◆ Présentation
  - **T-11** Ne truffez pas vos expressions de parenthèses inutiles qui finissent par polluer le code et le rendre illisible (priorité).
- ◆ Expressions
  - **E-1** Proscrire les expressions trop complexes. Utiliser l'affectation en passant avec discernement.
  - **E-3** Ne pas écrire d'expression dont le résultat dépend de l'ordre dans lequel elle sera évaluée (`rang [ i++ ] = i;`)
  - **C-13** Exploiter l'ordre d'évaluation des expressions (gauche à droite pour `and` et `or`) pour éviter les évaluations inutiles ou interdites.
  - **C-14** Maîtriser l'emploi des fonctions booléennes.
  - **T-14** Calculer la taille des zones allouées pour les structures par `sizeof` et non directement.

© AM - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© AM - Reproduction interdite sans l'autorisation de l'auteur



## Guide de Style INSA-IF – Expression

- ◆ Création et Destruction
  - **C-6** Associer chaque création (opérateur `new`) à une et une seule destruction (opérateur `delete`) et chaque destruction à une seule création afin de vérifier que tous les éléments créés sont détruits et que les éléments détruits ont été créés.
- ◆ Erreurs courantes
  - **E-1** La comparaison se note "==" et non "=" : la présence d'une affectation dans une condition n'étant pas une erreur, ce cas ne peut être détecté par les analyseurs.
  - **E-10** Utiliser `delete [ ]` pour détruire un tableau et non `delete`.



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

89



## Programmation Orientée Objet – C++

### PLAN du COURS

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                             |                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
|  <p><b>Naufrage</b></p> <ul style="list-style-type: none"> <li>■ ...</li> <li>■ Expressions</li> <li>■ Fonctions           <ul style="list-style-type: none"> <li>◆ Définition</li> <li>◆ Déclaration</li> <li>◆ Appel</li> </ul> </li> <li>■ Instructions</li> <li>■ Classe</li> <li>■ Héritage</li> </ul>  <p><b>Frisson</b></p> |  <p><b>Divin ?</b></p> |  <p><b>Bricolage</b></p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

90



## Fonction

- Unité d'exécution dans un programme C++
- 3 composantes pour une fonction
  - ◆ Définition de la fonction
    - Description du traitement réalisé lors de l'exécution (bloc)
    - Définition unique, dans un seul fichier source de l'application
  - ◆ Déclaration de la fonction
    - En-tête de la fonction, sans son bloc de définition
    - Pour les besoins de la compilation (contrainte sémantique)
      - ⇒ dans chaque fichier où on désire utiliser la fonction (inclusion)
  - ◆ Appel de la fonction
    - Nom de la fonction avec la liste des paramètres effectifs
    - Instruction dans un bloc d'un fichier quelconque de l'application

⇒ Le plus souvent, dans un **fichier d'interface** de classe ou de module, inclus par la directive  
**#include**



© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Définition d'une Fonction

- Syntaxe

```
[inline] type nom ([ParamètresFormels]) bloc
```

- ◆ **inline** indique que l'on souhaite que le corps de la fonction soit inséré à la place de l'appel dans le texte source
- ◆ **type** est le type de la valeur renvoyée par l'exécution de la fonction. **type** peut être...
  - Un type primitif : booléen, entier, virgule-flottante ou caractère
  - Un type défini par l'utilisateur : **enum**, **struct**, union ou classe
  - **void** pour définir une fonction sans valeur de retour (procédure)
  - Un pointeur sur un type **T** (type primitif ou défini par l'utilisateur)
  - Une référence à un type **T** (le retour doit être une variable et non pas une valeur)
  - **Attention** : retour type dérivé tableau impossible ⇒ retour d'un pointeur (tableau = pointeur) ou retour d'une référence à un tableau



© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Définition d'une Fonction

### Syntaxe (suite)

```
[inline] type nom ([ParamètresFormels]) bloc
```

◆ nom

- Identificateur au sens du C++
- Attention au choix des noms !

◆ ParamètresFormels

- Liste de définitions de paramètres séparées par une virgule
- Liste vide, si la fonction n'a pas d'argument

◆ bloc

⇒ Les items définis dans le bloc (variables et types) sont locaux au bloc et ne peuvent être utilisés que par les instructions du bloc

```
{
 // liste, éventuellement vide, d'instructions
 // liste, éventuellement vide, de déclarations
 // liste, éventuellement vide, de définitions
}
```



© MM - Reproduction interdite sans l'autorisation de l'auteur

94



## Définition des Paramètres Formels

### Syntaxe

⇒ Liste de paramètres :  
séparateur , (virgule)

◆ const

- Argument correspondant à ce paramètre non modifié par la fonction
- En cas d'absence, refus des appels avec un argument constant
- const nécessaire uniquement avec des arguments pointeur ou référence puisque dans les autres cas, aucun moyen de modifier la variable (passage par valeur)

◆ type : primitif, dérivé ( &, \*, [ ] ) ou défini par l'utilisateur

- & : référence à l'argument communiquée à la fonction
- Sinon, passage par valeur (expression possible dans ce cas)

◆ nom : définition classique



© MM - Reproduction interdite sans l'autorisation de l'auteur

95



## Définition d'une Fonction

### Cas particulier : la fonction `main`

Rappel : Syntaxe d'une définition de fonction

[ `inline` ] type nom ( [ ParamètresFormels ] ) bloc

```
// Cas particulier de main : syntaxe complexe
int main (int nbArg, char *listArg [])
// Mode d'emploi :
// ...
{
```

⇒ Entier ( $\geq 1$ ) donnant le nombre de chaînes de caractères transmises au moment de l'appel de la fonction `main`, c'est-à-dire du lancement du programme

⇒ Tableau de pointeurs sur les différentes chaînes de caractères  
⇒ type dérivé : tableau de pointeurs



Rappel : Syntaxe d'une définition de paramètre formel

[ `const` ] type nom



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Définition d'une Fonction

### Cas particulier : appel de la fonction `main`

```
// Cas particulier de main : syntaxe plus complexe
int main (int nbArg, char *listArg [])
// Mode d'emploi :
{
```

...> concat nomFic1 5 < fic2 ..
Nom de l'exécutable : concat
Argument effectif n°1 : nomFic1
Argument effectif n°2 : 5

Valeurs des paramètres dans `main`

|                            |                        |
|----------------------------|------------------------|
| <code>nbArg</code>         | = 3                    |
| <code>listArg [ 0 ]</code> | = "concat"             |
| <code>listArg [ 1 ]</code> | = "nomFic1"            |
| <code>listArg [ 2 ]</code> | = "5"                  |
| <code>listArg [ 3 ]</code> | = <code>nullptr</code> |

⇒ 2 arguments effectifs ( `nomFic1` et 5 )
+ nom de l'exécutable (concat)
⇒ `nbArg`  $\geq 1$  pour tous les appels
⇒ Remarque : < `fic2` traité par l'interpréteur de commandes

⇒ Nom de l'exécutable : toujours en 0

⇒ Arguments formels : toujours sous forme de chaînes de caractères

⇒ Marque de fin de la liste des arguments
⇒ par définition, `listArg[nbArg]=0`



## Déclaration d'une Fonction

- Utilisé par le compilateur...
  - ◆ Pour vérifier le type de la valeur renvoyée par l'exécution de la fonction
  - ◆ Pour valider le type des arguments attendus par la fonction à chaque appel ⇒ mise en place des éventuelles conversions avant l'appel
- Une déclaration
  - ◆ Indispensable avant tout appel d'une fonction
  - ◆ Très souvent, positionnée dans une interface
- Pour une méthode, la déclaration fait partie de l'interface de la classe (Cf. La Classe)
- **Rappel** : une définition a valeur de déclaration



## Déclaration d'une Fonction

- Syntaxe
 

↳ **Attention** : ne pas oublier le point-virgule pour les déclarations de fonction (pas de bloc)

`[ extern ] type nom ( [ ParamètresFormels ] ) ;`

  - ◆ **extern** : facultatif (et généralement omis)
  - ◆ **type** est le type de la valeur renvoyée par l'exécution de la fonction (cf. Définition d'une fonction)
  - ◆ **nom**
    - Identificateur au sens du C++
  - ◆ **ParamètresFormels** : syntaxe voisine de celle décrite dans la définition d'une fonction
  - ◆ Point-virgule '**;**'
  - Le point-virgule, à la place du bloc (`{ . . . }`), indique qu'il s'agit d'une déclaration et non d'une définition de fonction



## Déclaration des Paramètres Formels

### Syntaxe

⇒ Liste de paramètres :  
 séparateur , (virgule)

```
[const] type [nom] [= valeur]
```

- ◆ **const** (cf. Définition d'une Fonction C++)
- ◆ **type** (cf. Définition d'une Fonction C++)
- ◆ **nom**
  - Définition classique mais peut être omis
  - Pour des raisons évidentes d'auto-documentation, il est recommandé de conserver les noms des paramètres formels
- ◆ **valeur** : valeur par défaut pour le paramètre formel
  - Possible pour les paramètres formels se trouvant **en fin de liste**
  - Paramètres effectifs associés peuvent être omis lors de l'appel
  - Allègement de l'appel



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Déclaration des Paramètres Formels

### Exemple

⇒ **Attention** : erreur de syntaxe  
puisque \*= est un opérateur  
⇒ espace obligatoire

⇒ Valeur par défaut pour le dernier paramètre  
⇒ appel `f ( 10, &x );` est licite  
⇒ appel `f ( 10, &x, false );` est aussi licite

⇒ L'absence des noms des paramètres formels  
nuit à la lisibilité du code

```
// Exemple de déclarations de fonctions
int f (int, float *, bool = true);
int f (int taille, float *moyenne, bool affichage = true);

int f (int, float *= nullptr, bool = true);

int f (int = -3, float *, bool = true);
```

⇒ **Attention** : erreur de syntaxe puisque le  
paramètre initialisé n'est pas en fin de liste



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Appel d'une Fonction

### Syntaxe

⇒ Liste de paramètres :  
 séparateur , (virgule)

**nom ( [ ParamètresEffectifs ] )**

- ◆ Apparaît obligatoirement dans une expression
- ◆ **nom**
  - Identificateur de la fonction (Cf. Définition d'une Fonction C++)
- ◆ **ParamètresEffectifs**
  - Liste de paramètres effectifs (expressions) séparés par une virgule
  - Liste vide pour une fonction sans paramètre formel

```
// Exemple de déclaration de fonction
int Convertir (int valeur, int baseD = 16, int baseS = 10);

// Différents appels possibles
x = Convertir (256); // x vaut 0x100
y = Convertir (x, 8, 16); // y vaut 0400

// Autre syntaxe
y = Convertir (x = Convertir (256), 8, 16);
```

© IMI - Reproduction interdite sans l'autorisation de l'auteur



## Appel d'une Fonction

### Principe

- ◆ Vérification de la conformité des types des arguments avec la déclaration de la fonction
- ◆ Selon la déclaration, communication à la fonction...
  - De la valeur calculée, et éventuellement convertie, de chacun des arguments
  - De la référence de chacun des arguments
- ◆ Exécution du bloc correspondant au corps de la fonction
- ◆ Au retour, selon la déclaration, utilisation...
  - De la valeur de l'**expression** de retour, dans l'expression où apparaissait l'appel de la fonction, avec une **double conversion possible**
    - **expression** doit être du type de retour défini pour la fonction  
⇒ conversion éventuelle
    - Appel de la fonction toujours dans une expression au niveau de l'appelant  
⇒ règle de conversion dans les expressions appliquées
  - De la référence, dans l'expression où apparaissait l'appel



© IMI - Reproduction interdite sans l'autorisation de l'auteur



## Programmation Orientée Objet – C++

---

### PLAN du COURS



**Naufrage**

- ...
- Fonctions
- Instructions
  - ◆ Synthèse
  - ◆ Instructions de Sélection
  - ◆ Instructions Itératives
  - ◆ Instructions de Saut
- Classe
- Héritage



**Divin ?**



**Bricolage**

© MM - Reproduction interdite sans l'autorisation de l'auteur

104



INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON



## Synthèse

---

- Définition / déclaration  
`type déclarateur [initialiseur] ;`
- Instruction simple  
`expression ;`
- Bloc  
`{ suite-instructions }`
- Instruction de sélection  
`if`  
`switch`
- Instruction itérative  
`while`  
`do ... while`  
`for`

- Instruction de saut  
`return`  
`break`  
`continue`  
`goto`
- Lancement d'une exception  
`throw`

**Remarques**

- ⇒ Une déclaration est une instruction ⇒ peut se placer n'importe où dans le code source
- ⇒ Une affectation est une expression et non une instruction ⇒ peut devenir une instruction simple
- ⇒ Une `suite-instructions` peut être vide ou composée d'instruction(s)
- ⇒ Il n'existe pas d'instruction d'appel de procédure (`call`)

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

## Instruction if



■ Syntaxe

```
if (condition)
 instruction
[else
 instruction]
```

◆ **condition**

- Expression booléenne ou convertible en booléen  
⇒ `if ( x )` est équivalent à `if ( x != 0 )` ⇒ Rappel : toute valeur non nulle est considérée comme `true`
- Lisibilité des conditions
 

```
if (p.Sommet () != 0) // pour tester entiers et pointeurs
if (p.Vide ()) // pour tester les booléens
```
- Cas particulier : `condition` peut se réduire à une définition de variable initialisée, dont la portée est limitée à l'instruction `if`
  - Ne pas oublier les parenthèses autour de la `condition`

◆ La clause `else` est facultative

◆ **Conseil** : toujours mettre un **bloc** pour `instruction`, même si cela n'est pas nécessaire

© AMI - Reproduction interdite sans l'autorisation de l'auteur


107

## Instruction if



■ Exemple

⇒ Ne pas oublier les parenthèses

⇒ expression booléenne ou  
**Type variable = expression-entière**

// Exemple d'instruction conditionnelle if

```
if (a < b)
 max = b;
else
 max = b;
```

⇒ instruction = instruction simple  
⇒ expression ;

if ( a < b )
{
 // bloc inutile mais conseillé
 max = b;
}
else
{
 // bloc inutile mais conseillé
 max = a;
}

⇒ instruction = bloc  
⇒ { suite-instructions }

// Beaucoup plus simple... avec l'expression conditionnelle
// et une instruction simple
max = ( a < b ) ? b : a;

© AMI - Reproduction interdite sans l'autorisation de l'auteur


108

**INSA**

## Instruction switch



■ Syntaxe classique

```
switch (condition)
{
 case expression-constante1:
 instruction
 [break;]
 case expression-constante2:
 instruction
 [break;]
 ...
 [default :
 instruction
 [break;]]
}
```

⇒ Ordre indifférent pour les différents cas (même `default`), mais valeurs obligatoirement différentes

⇒ La condition est évaluée puis comparée aux différentes expressions constantes de la liste

⇒ Si la valeur de l'expression est trouvée dans la liste, l'exécution commence à partir de la liste d'instructions correspondante

⇒ Elle continue jusqu'à la rencontre d'un ordre de sortie explicite `break`  
⇒ pas de fin automatique à l'apparition de la prochaine clause `case` !

⇒ En cas d'absence de `break`, on enchaîne sur les cas suivants, jusqu'à la fin du `switch`

⇒ Si la valeur de l'expression n'est pas trouvée dans la liste et si l'option `default` existe, l'exécution commence à partir de la liste d'instructions correspondante

⇒ Si la valeur de l'expression n'est pas trouvée dans la liste et si l'option `default` est absente, l'instruction `switch` n'a pas d'action

## Instruction switch



■ Exemple

```
switch (opérande)
{
 case MULTIPLICATION : x *= y; break;
 case DIVISION : x /= y; break;
 case ADDITION : x += y; break;
 case SOUSTRACTION : x -= y; break;
 case INCREMENT2 : x++; ← Poursuite de l'exécution
 case INCREMENT1 : x++; break; ← 2 cas identiques
 case PISSANCE : cout << "Pas fait !" << endl;
 case MODULO : cout << "Bug !" << endl;
 default : exit (1);
}
```

⇒ `break` inutile (même sans l'`exit`)  
⇒ dernier cas de la sélection

© MM - Reproduction interdite sans l'autorisation de l'auteur

<div style="position: absolute; bottom: 0px; left: 15px; width: 100px; height: 100px; background-color: darkdarkdarkdarkdarkdarkdarkdarkdarkdarkdarkdarkdarkdarkdarkred; border-radius



## *Instruction while*

**while ( condition ) ;**

**instruction**

**Attention :** pas de point-virgule ici !  
 sinon, **while** sans corps de boucle

- Syntaxe

- ◆ **condition**
  - Expression booléenne ou convertible en booléen
  - Ne pas oublier les parenthèses autour de la **condition**
- ◆ Effectue le corps de la boucle (**instruction**) tant que la **condition** est **true**
- ◆ Arrêt prématuré de l'itération par l'instruction **break** (mais aussi **return**, **goto** et **throw**) ⇒ évite le positionnement d'indicateurs forçant la sortie d'une boucle
- ◆ De manière similaire, abandon de l'itération en cours pour passer directement à la suivante avec l'instruction **continue**
- ◆ **Conseil** : toujours mettre un **bloc** pour **instruction**, même si cela n'est pas nécessaire

**INSA** | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## *Instruction do . . . while*

**do**

**instruction**

**while ( condition );**

**Attention :** point-virgule obligatoire !

- Syntaxe

**⇒ instruction exécutée au moins une fois, puisque la condition est à la fin**

- ◆ **condition**
  - Expression booléenne ou convertible en booléen
  - Ne pas oublier les parenthèses autour de la **condition**
- ◆ Effectue le corps de la boucle (**instruction**) tant que la **condition** est **true**, c'est-à-dire jusqu'à ce que la **condition** soit **false** !
- ◆ Arrêt prématuré de la boucle possible comme pour **while**
- ◆ Abandon possible de l'itération en cours comme pour **while**
- ◆ Instruction peu utilisée, souvent source d'erreurs
- ◆ **Conseil** : toujours mettre un **bloc** pour **instruction**, même si cela n'est pas nécessaire

**INSA** | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Instruction for

### ■ Syntaxe

```
for ([inst_init] ; [expression2] ; [expression3])
 instruction
```

- ◆ **inst\_init**: peut être une définition de variable ou une expression d'initialisation
- ◆ **expression2** : condition de continuation de la boucle
  - Si **expression2** est absente, elle est considérée vraie (boucle infinie)
- ◆ **expression3** : expression de fin de boucle
- ◆ Chacune des expressions peut être omise mais les ";" doivent être conservés ⇒ **for** ( ; ; ) boucle infinie du C++
- ◆ **Remarque** : aucune restriction n'est faite sur les expressions, ce qui permet d'adopter tout type de progression non arithmétique, et aussi de modifier en cours d'itération indice, progression et borne



## Instruction for

### ■ Équivalence **for** ⇔ **while**

|                                                                     |                                                                                                                                                                            |
|---------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre><b>for</b> ( inst_init ; expr2 ; expr3 )     instruction</pre> | <pre>inst_init; // initialisation boucle <b>while</b> ( expr2 ) // continuation {     instruction; // corps boucle     expr3; // passage à l'itération } // suivante</pre> |
|---------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### ■ Exemple

⇒ Remarque : utilisation possible de l'opérateur , dans les différentes expressions

⇒ Définition d'une variable dans l'**inst\_init**  
⇒ attention à la portée de la variable

⇒ Expression de fin de boucle complexe

```
const int TAILLE = 10;
int vecteur [TAILLE];
for (int i = 0 ; i < TAILLE ; vecteur [i++] = 0)
{ // bloc inutile
}
```

⇒ Bloc inutile mais exigé par le guide de style



## Instructions de Saut

### ■ **return**

- ◆ Fin d'exécution d'une fonction (cf. Appel d'une fonction)
  - **return;** : si la fonction est une procédure (type **void**)
  - **return expression;** : renvoi de la valeur de l'expression si type  $\neq$  **void**

### ■ **break**

- ◆ 2 significations différentes...
  - Dans une instruction de sélection **switch** : sortir de la liste des branches de la sélection multiple (cf. Instruction **switch**)
   
⇒ *sauter à la fin du switch*
  - Dans une instruction itérative **for**, **while** et **do ... while** : arrêter prématurément l'instruction sans qu'il soit nécessaire de tester l'expression de contrôle de la boucle (**condition**)
   
⇒ *sauter à la fin de l'instruction itérative*



## Instructions de Saut

### ■ **continue**

- ◆ Permet de passer directement à l'itération suivante dans une instruction itérative
  - Dans les instructions **while** ou **do ... while**, évaluation immédiate de l'expression de contrôle
  - Dans l'instruction **for**, évaluation de l'expression de fin de boucle en premier, suivie de la condition de continuation

### ■ **goto**

- ◆ En principe, inutile et surtout facile à éviter
- ◆ Syntaxe

```
goto étiquette;
...
étiquette : instruction
```

⇒ **étiquette** est un identificateur C++  
qui n'a pas besoin d'être déclaré/défini



## Guide de Style INSA-IF – Instruction

### ◆ Présentation

- **T-7** Lorsqu'une instruction comme `if` ou `for` ne porte que sur une seule instruction simple et non sur un bloc, placer des accolades même si ce n'est pas nécessaire.
- **T-8** N'écrire qu'une seule instruction par ligne. Ne pas utiliser l'opérateur ',', pour exécuter plusieurs instructions.
- **T-9** Mettre les imbrications des instructions en valeur par indentation et aligner les ouvertures et fermetures de blocs.
- **T-10** Signaler les instructions `for` qui ne comportent pas de bloc.

### ◆ Formes algorithmiques

- **T-15** Les formes algorithmiques sont suffisamment riches : exclure toute utilisation du branchement inconditionnel `goto`.
- **T-22** Pour une boucle infinie, ne pas écrire `while ( true )` mais `for ( ; ; )`.



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Guide de Style INSA-IF – Instruction

### ◆ Formes algorithmiques (suite)

- **T-23** Placer une instruction `break` à la fin de chaque `case`, surtout du dernier, en cas d'ajout ultérieur.  
Il est également judicieux de commenter l'absence de `break` pour distinguer l'omission involontaire de l'absence effectivement voulue.
- **T-24** Dans le cas d'une sélection portant sur une variable de type énuméré, traiter explicitement tous les cas.  
Ne pas placer de cas `default` pour pouvoir contrôler que toutes les valeurs possibles sont traitées correctement, en particulier lors de l'ajout d'une nouvelle valeur dans le type.



© INSA - Reproduction interdite sans l'autorisation de l'auteur

**Programmation Orientée Objet – C++**

## PLAN du COURS



**Naufrage**

- ...
- Instructions
- Classe
  - ◆ Concept de Classe
  - ◆ Visibilité des Attributs / Méthodes
  - ◆ Interface et Réalisation d'une Classe
  - ◆ Déclaration et Définition de Méthodes
  - ◆ Constructeurs / Destructeur
  - ◆ Appel de Méthode
  - ◆ Variable et Méthode de Classe
- Héritage



**Divin ?**



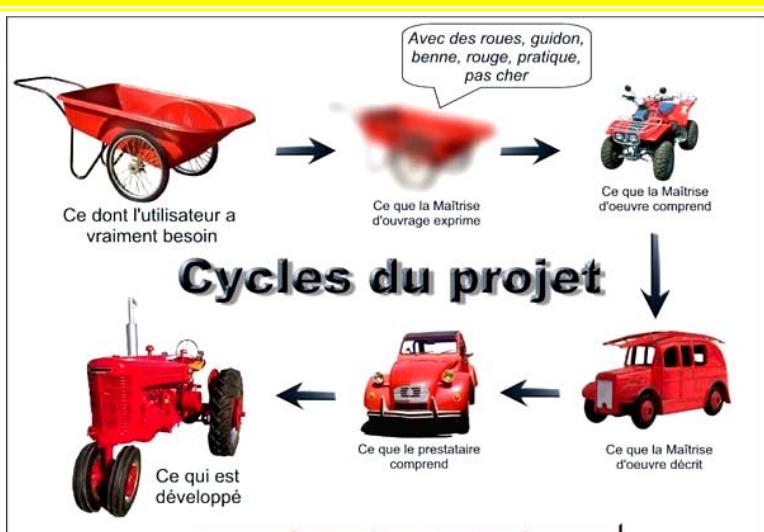
**Bricolage**

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur

119

**Classe**



Ce dont l'utilisateur a vraiment besoin

Avec des roues, guidon, benne, rouge, pratique, pas cher

Ce que la Maîtrise d'œuvre exprime

Ce que la Maîtrise d'œuvre comprend

**Cycles du projet**



Ce qui est développé

Ce que le prestataire comprend

Ce que la Maîtrise d'œuvre décrit

Ce que chacun comprend...

http://www.projetsinformatiques.com

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Concept de Classe

- Classe : définition d'un nouveau type
  - ◆ Associe un identificateur, le nom de la classe, à...
    - Une collection de méthodes (ou fonctions membres) : opérations applicables aux objets, instances de cette classe
    - Une collection d'attributs (variables d'instance ou données membres) de type quelconque
    - Des niveaux de visibilité : accessibilité depuis le reste du programme pour chacun des membres
      - Publique pour les méthodes
      - Privée pour les attributs
    - ◆ Définition d'un certain degré d'abstraction (*information hiding*)
    - Évolutivité, maintenabilité
- Structure
  - ◆ **struct**, type défini par l'utilisateur, est un cas particulier de classe  
⇒ tout est public



© MM - Reproduction interdite sans l'autorisation de l'auteur



## Visibilité des Attributs et Méthodes

- Contrôlée par 3 paragraphes de l'interface...
  - ◆ **public** :
  - Les éléments (méthodes et attributs) déclarés dans un tel paragraphe sont accessibles par tout programme utilisateur de la classe
  - À éviter pour les attributs ⇒ plus d'encapsulation des données !
  - ◆ **protected** :
  - Très utile dans un contexte d'héritage
  - Seules les méthodes de la classe et celles de tous ses descendants (héritage) peuvent accéder à ces éléments
  - Accès interdit pour tous les autres
  - ◆ **private** :
  - On ne peut accéder à ces éléments que depuis une des méthodes de la classe ⇒ encapsulation parfaite



© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple d'Interface d'une Classe

```
// Interface définissant une classe Point (fichier Point.h) qui concrétise
// l'abstraction "point" dans un espace 2D d'entiers
class Point
{
 public : // Déclaration des membres (méthodes/attributs) publics
 void Deplacer (const Point & delta);
 // Déplace le point du vecteur delta

 void Afficher (const char *message = "") const;
 // Affiche les coordonnées du point précédées par un
 // message optionnel

 Point (int abs = 0, int ord = 0);
 // Construit un point à partir de 2 entiers

 protected : // Déclaration des membres (méthodes/attributs) protégés
 int x; // Les attributs x et y du point sont protégés
 int y;
};

 ⇒ Visibilité des méthodes et des attributs
 ⇒ Attention : En cas d'absence de paragraphes de visibilité,
 par défaut, c'est le paragraphe private : qui est retenu
 ⇒ Attention : ne pas oublier le ; qui marque
 la fin de la définition de la classe

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple d'Interface d'une Classe

```
----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined (POINT_H)
#define POINT_H

----- Interfaces utilisées
----- Constantes
----- Types

----- Rôle de la classe <Point>
----- Description du rôle de la classe : ce que fait la classe

class Point
{
 public :
 //----- PUBLIC
 //----- Méthodes publiques
 //----- Surcharge d'opérateurs
 //----- Constructeurs - destructeur

 //----- PRIVE
 protected :
 //----- Méthodes protégées
 //----- Attributs protégés
};

----- Autres définitions dépendantes de <Point>

#endif // ! defined (POINT_H)

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple d'Interface d'une Classe

```

//----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined (POINT_H)
#define POINT_H

//----- Interfaces utilisées ----- Constantes -----
//----- Rôle de la classe <Point>
// Description du rôle de la classe : ce que fait la classe
//----- Description du rôle de la classe : ce que fait la classe
//----- Types -----
class Point
{
 //----- PUBLIC -----
 public :
 //----- Méthodes publiques -----
 //----- Surcharge d'opérateurs -----
 void Deplacer (const Point & delta);
 // Mode d'emploi :
 // Déplace le point du vecteur delta
 //----- Contrat :
 // Aucun

 protected :
 //----- Constructeurs - destructeur -----
 void Afficher (const char *message = "") const;
 // Mode d'emploi :
 // Affiche les coordonnées du point précédées d'un
 // message optionnel
 //----- Contrat :
 // Aucun
};

//----- Autres définitions dépendantes de <Point>

#endif // ! defined (POINT_H)

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple d'Interface d'une Classe

```

//----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined (POINT_H)
#define POINT_H

//----- Interfaces utilisées ----- Constantes -----
//----- Rôle de la classe <Point>
// Description du rôle de la classe : ce que fait la classe
//----- Description du rôle de la classe : ce que fait la classe
//----- Types -----
class Point
{
 //----- PUBLIC -----
 public :
 //----- Constructeurs - destructeur -----
 Point (int abs = 0, int ord = 0);
 // Mode d'emploi :
 // Construit un point à partir de 2 entiers
 //----- Contrat :
 // Aucun

 protected :
 //----- Attributs protégés -----
 int x; // Les attributs x et y du point sont
 int y; // protégés
};

//----- Autres définitions dépendantes de <Point>

#endif // ! defined (POINT_H)

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple d'Interface d'une Classe

```

//----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined (POINT_H)
#define POINT_H
class Point
{
public :
 //----- Méthodes publiques
 void Deplacer (const Point & delta);
 // Mode d'emploi :
 // Déplace le point du vecteur delta
 // Contrat : aucun

 void Afficher (const char *message = "") const;
 // Mode d'emploi :
 // Affiche les coordonnées du point précédées d'un message optionnel
 // Contrat : aucun

 //----- Constructeurs - destructeur
 Point (int abs = 0, int ord = 0);
 // Mode d'emploi :
 // Construit un point à partir de 2 entiers
 // Contrat : aucun

protected :
 //----- Attributs protégés
 int x; // Les attributs x et y du point sont protégés
 int y;
};

#endif // ! defined (POINT_H)

```

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Déclaration d'une Classe

### ■ Syntaxe

```
class NomDeClasse;
```

- ◆ Déclaration d'une classe de nom **NomDeClasse**
- ◆ Pas de description de la classe...
  - Impossibilité de définir des objets, instance de la classe, à partir de cette déclaration
  - Possibilité de définir et/ou déclarer des pointeurs et/ou des références sur cette classe
- ◆ Résout le problème des références avant ⇒ utilisation d'une classe non encore définie



## Déclaration d'une Méthode

### ■ Syntaxe

```
type nomMéthode ([ParamètresFormels]) [const];
```

- ◆ **type** est le type de la valeur renvoyée par la méthode...
  - Type primitif (**bool, char, int, float...**)
  - Type dérivé pointeur (**Type \***) ou référence (**Type &**)
  - Type défini par l'utilisateur (classe, énumération, structure, union)
  - **void** (procédure  $\Rightarrow$  pas de valeur renvoyée par la méthode)
- ◆ **nomMéthode** : identificateur au sens du C++
- ◆ **ParamètresFormels**
- ◆ **const** : ne modifie pas les attributs de l'objet auquel on applique la méthode
- ◆ Point-virgule '**;**'
- Indique qu'il s'agit d'une déclaration et non d'une définition de méthode (**inline**)



© INSA - Reproduction interdite sans l'autorisation de l'auteur

129



## Déclaration des Paramètres Formels

### ■ Syntaxe

```
[const] type [nom] [= valeur]
```

- ◆ **const**
  - Argument correspondant non modifié par la méthode
  - **const** nécessaire uniquement avec des arguments pointeur ou référence puisque dans les autres cas, aucun moyen de modifier l'argument (passage par valeur  $\Rightarrow$  copie de la valeur)
- ◆ **type**
  - Type primitif (**bool, char, int, float...**)
  - Type dérivé tableau ([**1**]), pointeur (**Type \***) ou référence (**Type &**)
  - Type défini par l'utilisateur (classe, énumération, structure, union)
- ◆ Passage par valeur (expression possible), sauf s'il s'agit d'une référence



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

130



## Déclaration des Paramètres Formels

- Syntaxe (suite)

```
[const] type [nom] [= valeur]
```

- ◆ **nom**
  - Identificateur classique C++ mais peut être omis
  - Pour des raisons évidentes d'auto-documentation, il est recommandé de conserver les noms des paramètres formels
- ◆ **valeur** : valeur par défaut pour le paramètre formel
  - Possible pour les paramètres formels se trouvant en fin de liste
  - Paramètres effectifs associés peuvent être omis lors de l'appel
  - Allègement de l'invocation de la méthode
- ◆ **Attention** : toute méthode a un argument implicite (paramètre formel implicite) nommé **this**
  - N'apparaît jamais dans la déclaration de la méthode



© MM - Reproduction interdite sans l'autorisation de l'auteur

131



## Pointeur this

- Équivalent de...
  - ◆ THIS de Simula
  - ◆ self de Smalltalk
- **this**
  - ◆ Dans une méthode non **static**, pointeur sur l'objet pour lequel la méthode a été appelée
- Type de **this**
  - ◆ Dans une méthode non **const** d'une classe **C**  
⇒ **C \* const** (pointeur constant sur objet non constant)
  - ◆ Dans une méthode **const** d'une classe **C**  
⇒ **const C \* const** (pointeur constant sur objet constant)



© MM - Reproduction interdite sans l'autorisation de l'auteur

132



## Exemple de Réalisation d'une Classe

```

// Réalisation de la classe Point (fichier Point.cpp)
using namespace std;
#include <iostream> // Inclusion utile à la définition de la méthode Afficher
#include "Point.h" // La réalisation inclut toujours l'interface de sa classe

void Point::Deplacer (const Point & delta)
// Déplace le point du vecteur delta (commentaire inutile)
{
 x += delta.x;
 y += delta.y;
} // Plus de valeurs par défaut à la définition des méthodes :
 // Paramètre message de la méthode Afficher
 // Paramètres abs et ord du constructeur d'objets

void Point::Afficher (const char *message) const
// Affiche les coordonnées du point précédées par un message optionnel (inutile)
{
 cout << message << x << "," << y << endl;
}

Point::Point (int abs, int ord)
// Construit un point à partir de 2 entiers (commentaire inutile)
{
 x = abs;
 y = ord;
}

```

⇒ Définition des méthodes : 2 modifications par rapport à la définition d'une fonction  
 [ inline ] type nomClasse::nom ( [ ParamètresFormels ] ) [ const ] bloc  
 • Apparition de la désignation de sa portée : le nom de la classe (Point)  
 • Apparition du mot-clé const (optionnel)

⇒ Définition des paramètres formels : syntaxe classique et connue  
 [ const ] type nom

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple de Réalisation d'une Classe

```

----- Réalisation de la classe <Point> (fichier Point.cpp) -----

----- INCLUDE -----
----- Include système -----
using namespace std;
#include <iostream>

----- Include personnel -----
#include "Point.h"

----- Constantes -----
----- PUBLIC -----
----- Méthodes publiques -----
----- Surcharge d'opérateurs -----
----- Constructeurs - destructeur -----
----- PRIVE -----
----- Méthodes protégées -----

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple de Réalisation d'une Classe

```

//----- Réalisation de la classe <Point> (fichier Point.cpp) -----
//----- INCLUDE ----- Include système
//-----
using namespace std;
#include <iostream>
#include "Point.h"
void Point::Afficher (const char *message) const
// Algorithme :
// Aucun, la méthode est trop simple !
{
 cout << message << x << "," << y << endl;
} //--- Fin de Afficher
//----- Méthodes publiques
//-----
//⇒ Définition des méthodes en s'appuyant sur le squelette fourni
//----- Surcharge d'opérateurs
//----- Constructeurs - destructeur
//----- PRIVE
//-----
//----- Méthodes protégées

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Exemple de Réalisation d'une Classe

```

//----- Réalisation de la classe <Point> (fichier Point.cpp) -----
//----- INCLUDE ----- Include système
//-----
using namespace std;
#include <iostream>
Point::Point (int abs, int ord)
// Algorithme :
// Aucun, le constructeur est trop simple !
{
#if defined (MAP) // MAP : Mise Au Point
 cout << "Appel au constructeur de <Point>" << endl;
#endif
 x = abs;
 y = ord;
} //--- Fin de Point (Constructeur)
//----- Constructeurs - destructeur
//-----
//----- PRIVE
//-----
//----- Méthodes protégées

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



## *Exemple de Réalisation d'une Classe*

```

----- Réalisation de la classe <Point> (fichier Point.cpp) -----
----- Include système

using namespace std;
#include <iostream>
//-
#include "Point.h"
//-
// Algorithme : aucun
void Point::Deplacer (const Point & delta)
{
 x += delta.x;
 y += delta.y;
} //---- Fin de Deplacer

//-
// Algorithme : aucun
void Point::Afficher (const char *message) const
{
 cout << message << x << "," << y << endl;
} //---- Fin de Afficher

//-
// Algorithme : aucun
Point::Point (int abs, int ord)
{
 x = abs;←
 y = ord;←
} //---- Fin de Point (Constructeur)

```

⇒ Attention : Des lignes de commentaire ont été supprimées dans le squelette pour que cela tienne sur le transparent. Il ne faut pas modifier le squelette

Méthodes publiques

Constructeurs - destructeur

⇒ Notation équivalente mais très peu utilisée :  
`this->x = abs; ou (*this).x = abs`  
`this->y = ord; ou (*this).y = ord`

⇒ Permet de lever des ambiguïtés dans certains cas d'homonymie (paramètre ⇔ attribut)

JOURNAL OF MANAGEMENT EDUCATION



## *Définition d'une Méthode*

## ■ Syntax

```
[inline] type classe::nom ([ParamètresFormels]) [const] bloc
```

- ◆ **type** est le type de la valeur renvoyée par la méthode...
    - Type primitif (**bool**, **char**, **int**, **float**...)
    - Type dérivé pointeur (**Type \***) ou référence (**Type &**)
    - Type défini par l'utilisateur (classe, énumération, structure, union)
    - **void** (procédure  $\Rightarrow$  pas de valeur renvoyée par la méthode)
  - ◆ **classe** : indication de portée de la méthode
  - ◆ **nom** : nom de la méthode  $\Rightarrow$  identificateur au sens du C++
  - ◆ **ParamètresFormels**
  - ◆ **const** : ne modifie pas les attributs de l'objet auquel on applique la méthode
  - ◆ Bloc : définition classique
    - Indique qu'il s'agit d'une définition de méthode

B MN - Recognition sans l'autorisation d'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



## *Définition des Paramètres Formels*



↳ Liste de paramètres :  
 séparateur , (virgule)

- Syntaxe
 

[ const ] type nom

  - ◆ **const** (optionnel)
    - Argument correspondant non modifié par la méthode
    - **const** nécessaire uniquement avec des arguments pointeur ou référence puisque dans les autres cas, aucun moyen de modifier l'argument (passage par valeur ⇒ copie de la valeur)
  - ◆ **type**
    - Type primitif (**bool**, **char**, **int**, **float**...)
    - Type dérivé tableau ([ ]) , pointeur (**Type \***) ou référence (**Type &**)
    - Type défini par l'utilisateur (classe, énumération, structure, union)
  - ◆ nom : identificateur classique C++, obligatoire
  - ◆ **Rappel** : toute méthode a un argument implicite nommé **this**
    - N'apparaît jamais dans la définition de la méthode


INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur

## *Constructeur d'Objet*



- Méthode particulière d'une classe
  - ◆ Même nom que la classe
  - ◆ Méthode sans résultat (même pas **void**) qui peut avoir des paramètres formels, avec des valeurs par défaut
  - ◆ Une classe peut avoir plusieurs constructeurs ⇒ plusieurs façons d'initialiser les objets de la classe (surcharge)
  - ◆ Si aucun constructeur n'est défini, le compilateur engendre un constructeur par défaut (sans argument) ⇒ **à éviter !**
- Rôle
  - ◆ Allocation mémoire pour les attributs dynamique de l'objet
  - ◆ Initialisation de leurs valeurs (arguments du constructeur)  
⇒ état initial stable pour l'objet
  - ◆ Automatiquement invoqué à chaque création d'un objet


INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Constructeur d'Objet

### ■ Remarques

- ◆ Constructeur par défaut : pas de paramètres
  - Syntaxe à l'instanciation :
 

```
Classe c; et non pas Classe c ();
```
- ◆ Une classe a toujours au moins un constructeur, mais pas forcément un constructeur par défaut
- ◆ En effet, dès qu'un constructeur d'objets existe dans une classe, le compilateur n'engendre plus automatiquement un constructeur par défaut, même s'il est nécessaire
- ◆ En cas de constructeurs multiples (surcharge), le choix du bon constructeur s'effectue en fonction des arguments
- ◆ Pour créer un tableau d'instances d'une classe C, sans initialisation, un constructeur par défaut doit exister



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Constructeur d'Objet par Défaut

```
class Exemple
{
 public :
 Exemple () ; // ❶ Constructeur par défaut
 Exemple (int unX, int unY) ; // ❷ Autre constructeur
 protected :
 int x;
 int y;
 };
 //----- Réalisation de la classe
Exemple::Exemple () // ❶ Constructeur par défaut
{
 x = y = 0;
} //---- Fin de Exemple (constructeur par défaut)
Exemple::Exemple (int unX, int unY) // ❷ Autre constructeur
{
 x = unX;
 y = unY;
} //---- Fin de
//----- Fin de
int main ()
{
 Exemple e1 () ; // Erreur : aucun constructeur n'est appelé !
 Exemple e2; // Constructeur ❶ appelé
 Exemple e3 (1, 2); // Constructeur ❷ appelé
 Exemple e4 (2); // Erreur de compilation : pas de constructeur adéquat
 return 0;
} //---- Fin du main
```

⇒ Multiplication des constructeurs peu judicieuse  
⇒ Utilisation de valeurs par défaut pour les paramètres

⇒ Attention : ce n'est pas l'instanciation d'un objet e1 de la classe Exemple, mais la déclaration d'une fonction e1, sans paramètre et qui rend une valeur de type Exemple



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Constructeur d'Objet par Défaut

```
// Pour la classe Exemple du transparent "Constructeur d'Objet par Défaut"
//
const int MAX = 2;

int main ()
{
 Exemple t1[MAX];
 // Appel du constructeur par défaut Exemple (), pour chaque élément du tableau t1
 // => Constructeur par défaut obligatoire pour une telle instanciation

 Exemple t2[MAX] = { Exemple (1, 1) };
 // Appel du constructeur par défaut Exemple (), pour le 2ème élément du tableau t2
 // => Constructeur par défaut obligatoire pour une telle instanciation

 Exemple t3[MAX] = { Exemple (1, 1), Exemple (2 , 2) };
 // Appel du constructeur paramétré Exemple (int, int), pour tous les éléments
 // du tableau t3 => Constructeur par défaut non obligatoire pour une telle instanciation

 return 0;
} //---- Fin du main
```



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Initialisation des Attributs d'un Objet

### ■ Syntaxe

```
Classe :: Classe (ParamètresFormels)
 : attribut1 (expr1), attribut2 (expr2), ...
{ // Corps du constructeur
}
```

- ◆ Uniquement à la définition d'un constructeur ( { ... } )  
⇒ jamais dans la déclaration
- ◆ Initialisation des attributs avant de développer le corps
- ◆ Préférable à l'emploi d'affectation dans le corps
- ◆ Principe
  - si attribut<sub>i</sub> est de type primitif, expr<sub>i</sub> désigne la valeur initiale de cet attribut
  - si attribut<sub>i</sub> est un objet de classe C, expr<sub>i</sub> désigne la liste des arguments à communiquer au constructeur adéquat de la classe C (cascade d'appels possible selon la nature de l'objet)



© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Initialisation des Attributs d'un Objet

```

//----- Interface de la classe <Pixel> (fichier Pixel.h) -----
#ifndef ! defined (PIXEL_H)
#define PIXEL_H
#include "Couleur.h" // Interface de la classe Couleur
#include "Point.h" // Interface de la classe Point
class Pixel
{
public :
 //----- Constructeurs - destructeur
 Pixel (Couleur coul = Couleur :: BLANC, int abs = 0, int ord = 0);
 // Mode d'emploi :
 // Construit un pixel à partir de 2 entiers (un point) et d'une couleur
protected :
 //----- Attributs protégés
 Point p; // Point identifiant le pixel
 Couleur c; // Couleur du point
};
#endif // ! defined (PIXEL_H) // Inclusion à faire : interface autosuffisante
//----- Réalisation de la classe <Pixel> (fichier Pixel.cpp) -----
//----- Include personnel
#include "Pixel.h"
//----- Constructeurs - destructeur
Pixel::Pixel (Couleur coul, int abs, int ord)
 : p (abs, ord), c (coul)
// Algorithme : aucun
{ // Le corps du constructeur est vide !
} //----- Fin de Pixel (constructeur) // La réalisation inclut toujours l'interface de sa classe

```

© MM - Reproduction interdite sans l'autorisation de l'auteur

⇒ Liste de paramètres ⇒ constructeurs adéquats pour les 2 attributs



## Constructeur de Copie

- Méthode particulière d'une classe
  - ◆ Création d'un objet et initialisation de cet objet par la valeur d'un objet existant ⇒ Même nom que la classe
  - ◆ Méthode sans résultat (même pas **void**) qui a un seul paramètre formel ⇒ pour une classe C, il a toujours la forme :
 
$$C ( \ const\ C\ &\ unObjet );$$
  - ◆ Constructeur de copie par défaut
    - Si aucun constructeur de copie n'est défini, le compilateur engendre un constructeur de copie par défaut ⇒ **initialisation par copie simple**, attribut par attribut
    - **Attention** : ce fonctionnement ne convient plus dès que l'un des attributs est un pointeur (copie en profondeur exigée) ⇒ **à éviter !**
    - Fonctionnement récursif si un attribut est de type classe...
      - Appel du constructeur de copie de la classe, s'il existe
      - Appel du constructeur de copie par défaut



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON





## Constructeur de Copie

- Appel au constructeur de copie
  - ◆ Lors du passage **par valeur** d'un objet à une fonction / méthode
  - ◆ Lors du retour **par valeur** d'un objet par une fonction / méthode
  - ◆ Lors de la création d'un objet avec un autre objet comme valeur initiale
- Remarque
  - ◆ Il est important de bien faire la différence entre initialisation et affectation

⇒ Syntaxe à privilégier

```
Classe o2 = o1; // Équivalent à classe o2 (o1);
// Construction de l'objet o2 et initialisation de cet objet à partir des valeurs de l'objet o1
// ⇒ Appel du constructeur de copie défini par l'utilisateur ou fourni par le compilateur C++
Classe o2; // Construction de l'objet o2 et initialisation de cet objet à partir du
// constructeur par défaut défini par l'utilisateur ou fourni par le compilateur C++
o2 = o1; // Affectation à l'objet o2 de la valeur de l'objet o1
// Appel de l'opérateur d'affectation défini par l'utilisateur ou fourni par le compilateur C++
```

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Destructeur d'Objet

- Méthode particulière d'une classe
  - ◆ Même nom que la classe, précédé par le caractère '**~**'
  - ◆ Méthode sans résultat et sans paramètre formel
  - ◆ ⇒ Un seul destructeur par classe
  - ◆ Indispensable si l'objet contient des attributs de type pointeur (création dynamique lors de la construction de l'objet)
  - ◆ Si aucun destructeur n'est défini, le compilateur engendre un destructeur par défaut ⇒ **à éviter !**
  - ◆ Automatiquement invoqué pour détruire un objet à la fin de sa vie...
    - Pour un objet local à un bloc, à la fermeture du bloc dans lequel il a été défini (implicitement)
    - Pour un objet implanté statiquement, à la fin de l'exécution du programme (implicitement)
    - Pour un objet alloué dynamiquement (opérateur **new**), lors de l'exécution de l'opérateur **delete** (explicitement)

**~NomClasse ( );**

© INSA - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON





## Durée de Vie

- La durée de vie d'un objet dépend de sa nature...
  - ◆ Local à un bloc
    - L'objet est créé à sa définition et détruit à la sortie du bloc
  - ◆ Statique
    - A un bloc : l'objet est créé à sa définition et détruit à la fin du programme
    - Au programme : l'objet est créé au début de l'exécution du programme et détruit à la fin du programme
  - ◆ Dynamique
    - L'objet est créé et détruit sous le contrôle du programme  
⇒ Utilisation des opérateurs **new** et **delete**
  - ◆ Attribut d'un autre objet
    - L'objet est créé lors de la construction de l'objet contenant et détruit lors de la destruction du contenant



## Destructeur d'Objet

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

class Vecteur
{
public :
 Vecteur (int taille = 10); // Constructeur de la classe
 ~Vecteur (); // Destructeur de la classe

protected :
 int tailleMax;
 int *elements; // Attribut de type pointeur sur int
};

//-----
Vecteur::Vecteur (int taille) : tailleMax (taille)
{
 cout << "Appel du constructeur avec tailleMax = " << tailleMax << endl;
 elements = new int [tailleMax];
} //---- Fin de Vecteur (constructeur)

Vecteur::~Vecteur ()
{
 cout << "Appel du destructeur pour tailleMax = " << tailleMax << endl;
 delete [] elements;
} //---- Fin de ~Vecteur (destructeur)
```

⇒ **Attention** : à compléter pour une gestion totale du vecteur  
⇒ attribut manquant : **nbElements** pour gérer, à tout instant, le nombre d'éléments effectivement présents dans le vecteur (objet non stable après la construction...)

⇒ Destructeur obligatoire pour récupérer la place mémoire des éléments



## Constructeur / Destructeur d'Objet

Résultat de l'exécution du main ?

```
Appel du constructeur avec tailleMax = 1
Appel du constructeur avec tailleMax = 2
Appel du constructeur avec tailleMax = 10
Appel du constructeur avec tailleMax = 10
Appel du constructeur avec tailleMax = 10
Appel du constructeur avec tailleMax = 4
Appel du constructeur avec tailleMax = 10
Appel du destructeur pour tailleMax = 10
```

```
static Vecteur v1 (1);

int main ()
{
 Vecteur v2 (2);
 Vecteur *v3 = new Vecteur [3];
 Vecteur *v4 = new Vecteur (4);
 {
 Vecteur v5;
 delete v3;
 static Vecteur v6 (6);
 }

 return 0;
} //--- Fin du main
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur


## Constructeur / Destructeur d'Objet

⇒ Déficit d'appels de la méthode destructeur :  
8 appels au constructeur contre 6 appels  
au destructeur...  
Pourquoi ?

⇒ Il manque l'instruction simple **delete v4**;  
⇒ L'appel au destructeur pour l'objet v1 n'est  
plus tracé ⇒ la destruction intervient après  
la fin du programme

```
Appel du constructeur avec tailleMax = 1
Appel du constructeur avec tailleMax = 2
Appel du constructeur avec tailleMax = 10
Appel du constructeur avec tailleMax = 10
Appel du constructeur avec tailleMax = 10
Appel du constructeur avec tailleMax = 4
Appel du constructeur avec tailleMax = 10
Appel du constructeur pour tailleMax = 10
Appel du destructeur pour tailleMax = 10
Appel du destructeur pour tailleMax = 10
Appel du constructeur avec tailleMax = 6
Appel du constructeur avec tailleMax = 6
Appel du destructeur pour tailleMax = 10
Appel du destructeur pour tailleMax = 6
Appel du destructeur pour tailleMax = 6
Press any key to continue
```

Résultat de l'exécution du main ?

```
static Vecteur v1 (1);

int main ()
{
 Vecteur v2 (2);
 Vecteur *v3 = new Vecteur [3];
 Vecteur *v4 = new Vecteur (4);
 {
 Vecteur v5;
 delete [] v3;
 static Vecteur v6 (6);
 }

 return 0;
} //--- Fin du main
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur


## *Appel d'une Méthode*



■ Syntaxe

```
objet.NomMéthode ([ParamètresEffectifs])
```

```
objet->NomMéthode ([ParamètresEffectifs])
```

- ◆ Apparaît obligatoirement dans une expression
- ◆ **objet** ⇒ Opération sur un objet pointé
- Identificateur d'un objet, instance d'une classe C
- ◆ **NomMéthode**
  - Méthode de la classe C dont l'objet **objet** est une instance
- ◆ **ParamètresEffectifs**
  - Liste des paramètres effectifs, séparés par une virgule
  - Liste vide pour une méthode sans paramètre formel
  - **Attention** : parenthèses obligatoires, même s'il n'y a pas de paramètres effectifs

⇒ Liste de paramètres :  
séparateur , (virgule)

**INSA** INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur

153



## *Utilisation d'Objets d'une Classe*



```
----- Réalisation du module <TPoint> (fichier TPoint.cpp) -----
----- INCLUDE ----- Include système
----- Include personnel ----- Include personnel
----- PRIVE ----- PRIVE
----- Constantes ----- Constantes
----- Types ----- Types
----- Variables statiques ----- Variables statiques
----- Fonctions privées ----- Fonctions privées
----- PUBLIC ----- PUBLIC
----- Fonctions publiques ----- Fonctions publiques

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

int main ()
{
 Point origine; // Création des objets origine et p, instances de la classe Point par
 Point p (5, 7); // appels de la méthode Point (constructeur) qui initialise les attributs
 p.Afficher ();
 p.Deplacer (Point (-1, 4));
 p.Afficher ("Position = ");
 p.x = 4;
 return 0;
} //---- Fin du main
```

----- Réalisation du module <TPoint> (fichier TPoint.cpp) -----  
----- INCLUDE ----- Include système  
----- Include personnel ----- Include personnel  
----- PRIVE ----- PRIVE  
----- Constantes ----- Constantes  
----- Types ----- Types  
----- Variables statiques ----- Variables statiques  
----- Fonctions privées ----- Fonctions privées  
----- PUBLIC ----- PUBLIC  
----- Fonctions publiques ----- Fonctions publiques

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Passage / Retour par Référence &

- Alternative au passage par valeur pour les arguments et / ou le retour d'une fonction / méthode
- Avantages
  - ◆ Efficacité : pas de copie au moment du passage de paramètres
  - ◆ Possibilité de modification des paramètres, sauf si **const** est utilisé
  - ◆ Permet d'utiliser le retour comme une variable et pas seulement comme une valeur ⇒ très utile lors de la surcharge des opérateurs

⇒ Passage de paramètre par référence

Exemple de déclaration

TypeRetour & NomDeMéthode ( Type & nom, ... );

⇒ Retour par référence



© INSA - Reproduction interdite sans l'autorisation de l'auteur



155



## Passage par Référence &

- Exemple avec une fonction ordinaire **Permuter**

```
// En C classique
void Permuter (int *x, int *y)
{
 int tampon;
 tampon = *x;
 *x = *y;
 *y = tampon;
}

int main ()
{
 int i = 0;
 int j = 5;
 Permuter (&i, &j);
 return 0;
}
```

⇒ Ne pas confondre :
 

- & adresse de ...
- & référence à ...

```
// En C++
void Permuter (int &x, int &y)
{
 int tampon;
 tampon = x;
 x = y;
 y = tampon;
}

int main ()
{
 int i = 0;
 int j = 5;
 Permuter (i, j);
 return 0;
}
```

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Retour par Référence &

```
#if ! defined (POINT_H)
#define POINT_H

class Point
{
public :
 void Afficher (const char *message = "") const;
 // Affiche les coordonnées du point précédées d'un message optionnel

 Point & Modifier (const Point p);
 // Modifie les coordonnées du point avec p et renvoie le nouveau point

 Point (int abs = 0, int ord = 0);
 // Construit un point à partir de 2 entiers

 Point (const Point & unPoint);
 // Construit un point à partir d'un point existant (unPoint)

 ~Point ();
 // Détruit un point

protected :
 int x; // Les attributs x et y du point sont protégés
 int y;
};

#endif // ! defined (POINT_H)
```

Point.h

⇒ Retour par référence d'un objet de type Point

© MII - Reproduction interdite sans l'autorisation de l'auteur



## Retour par Référence &

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

Point & Point::Modifier (const Point p)
{ x += p.x; y += p.y;
 return *this; }
```

⇒ Le type de this est Point \* const  
compte tenu de la déclaration de la méthode  
(absence de const) ⇒ return \*this;

```
} //---- Fin de Modifier
```

```
void Point::Afficher (const char *message) const
```

```
{ cout << message << x << "," << y << endl;
} //---- Fin de Afficher
```

```
Point::Point (int abs, int ord) : x (abs), y (ord)
{ cout << "Appel au constructeur avec (" << x << "," << y << ")" << endl;
} //---- Fin de Point (Constructeur)
```

⇒ Mise en place du traçage des  
constructeurs et du destructeur  
⇒ En compilation conditionnelle  
dans les squelettes

```
Point::Point (const Point & unPoint)
{ cout << "Appel au constructeur de copie" << endl;
 x = unPoint.x; y = unPoint.y;
} //---- Fin de Point (Constructeur de copie)
```

```
Point::~Point ()
{ cout << "Appel au destructeur pour (" << x << "," << y << ")" << endl;
} //---- Fin de ~Point (Destructeur)
```

© MII - Reproduction interdite sans l'autorisation de l'auteur



## Retour par Référence &

```

//----- Include système ----- TPoint.cpp
using namespace std;
#include <iostream>
//----- Include personnel -----
#include "Point.h"
//----- PRIVE Constantes -----
//----- Types -----
//----- Variables statiques -----
//----- Fonctions privées -----
//----- PUBLIC -----
//----- Fonctions publiques -----
int main ()
{
 Point p1 (1, 2);
 p1.Afficher ();
 Point p2 = p1;
 p2.Afficher ();
 p2.Modifier (p1);
 p2.Afficher ();
 p2.Modifier (p2.Modifier (p1)).Afficher ();
 return 0;
} //----- Fin du main

```

**Résultat de l'exécution du main ?**

⇒ Passage par valeur d'un objet de type **Point**  
   ⇒ Appel du constructeur de copie de la classe **Point**

⇒ Enchaînement classique d'appels de méthodes

⇒ Retour de **Modifier** : une référence sur un objet de type **Point**  
   ⇒ Utilisable dans tous les contextes où un type **Point** est attendu

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Variable de Classe

- Caractéristiques
  - ◆ Variable de classe ou donnée membre statique
  - ◆ Variable partagée par tous les objets, instances d'une classe  
⇒ Un seul exemplaire
  - ◆ Souvent utilisée pour compter les instances d'une classe
  - ◆ Définition possible de constantes de classe (**const**)
- Manipulation      **static type nomVariable; // déclaration**
  - ◆ Variable      **type NomDeClasse::nomVariable = valeur; // définition**
    - Déclaration avec le préfixe **static** dans la définition de la classe  
⇒ Interface de la classe (fichier .h)
    - Définition et initialisation dans la réalisation de la classe (fichier .cpp)
  - ◆ Constante
    - Définition et initialisation avec le préfixe **static const** dans la définition de la classe ⇒ Interface de la classe (fichier .h)

**static const type nomConstante = valeur;**

© INSA - Reproduction interdite sans l'autorisation de l'auteur

**INSA** INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

160



## Variable de Classe

- Exemple

```
#if ! defined (VECTEUR_H)
#define VECTEUR_H

class Vecteur
{
public :
 Vecteur (int taille = TAILLE_DEFAUT); // Constructeur de la classe
 ~Vecteur (); // Destructeur de la classe
 // ...
protected :
 int tailleMax;
 int *elements; // Attribut de type pointeur sur int
 static int nombreDeVecteur; // Variable de classe
 static const int TAILLE_DEFAUT = 10; // Constante de classe
};

#endif
```

Vecteur.h

© INSA - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



161



## Variable de Classe

- Exemple (suite)

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

#include "Vecteur.h"

int Vecteur::nombreDeVecteur = 0; // Définition et initialisation de la variable de classe

Vecteur::Vecteur (int taille) : tailleMax (taille)
{
 cout << "Appel du constructeur avec tailleMax = " << tailleMax << endl;
 elements = new int [tailleMax];
 cout << "Nombre d'instances de vecteur = " << ++nombreDeVecteur << endl;
} //---- Fin de Vecteur (constructeur)

Vecteur::~Vecteur ()
{
 cout << "Appel du destructeur pour tailleMax = " << tailleMax << endl;
 delete [] elements;
 cout << "Nombre d'instances de vecteur = " << --nombreDeVecteur << endl;
} //---- Fin de ~Vecteur (destructeur)
```

Vecteur.cpp

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Méthode de Classe

### ■ Caractéristiques

- ◆ Méthode de classe ou fonction membre statique
- ◆ Méthode partagée par tous les objets, instances d'une classe
  - Un seul exemplaire pour tous les objets ⇒ pas de pointeur `this`
  - Une fonction ordinaire dont la portée est réduite à la classe
  - Un accès contrôlé par `public`, `protected` et `private`

### ■ Manipulation

- ◆ Déclaration avec le préfixe `static` dans la définition de la classe  
⇒ Interface de la classe (fichier .h)

```
static type NomMéthode ([ParamètresFormels]);
```

- ◆ Définition dans la réalisation de la classe (fichier .cpp)

```
type NomDeClasse::NomMéthode ([ParamètresFormels])
{
 // ...
}
```

⇒ Attention : plus de mot-clé `static`

- ◆ Appel

```
NomDeClasse::NomMéthode ([ParamètresEffectifs]);
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Méthode de Classe

### ■ Exemple

```
#if ! defined (VECTEUR_H)
#define VECTEUR_H

class Vecteur
{
public :
 Vecteur (int taille = TAILLE_DEFAUT); // Constructeur de la classe
 ~Vecteur (); // Destructeur de la classe
 static int NombreTotal (); // Méthode de classe
 // ...
protected :
 int tailleMax;
 int *elements; // Attribut de type pointeur sur int
 static int nombreDeVecteur; // Variable de classe
 static const int TAILLE_DEFAUT = 10; // Constante de classe
};

#endif
```

Vecteur.h

© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Méthode de Classe

### ■ Exemple (suite)

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

int Vecteur::nombreDeVecteur = 0; // Définition et initialisation de la variable de classe

int Vecteur::NombreTotal () // Définition de la méthode de classe
{
 return nombreDeVecteur;
} //----- Fin de NombreTotal

Vecteur::Vecteur (int taille) : tailleMax (taille)
{
 cout << "Appel du constructeur avec tailleMax = " << tailleMax << endl;
 elements = new int [tailleMax];
 cout << "Nombre d'instances de vecteur = " << ++nombreDeVecteur << endl;
} //----- Fin de Vecteur (constructeur)

Vecteur::~Vecteur ()
{
 cout << "Appel du destructeur pour tailleMax = " << tailleMax << endl;
 delete [] elements;
 cout << "Nombre d'instances de vecteur = " << --nombreDeVecteur << endl;
} //----- Fin de ~Vecteur (destructeur)
```

Vecteur.cpp

© INSA - Reproduction interdite sans l'autorisation de l'auteur



## Guide de Style INSA-IF – Classe

### ◆ Typographie

- **T-6** Convention d'écriture : cela permet de connaître la nature ou le statut d'un élément simplement par l'examen de son nom.

| Statut ⇒<br>↓ Objet                      | Public                 | Protégé – Privé        |
|------------------------------------------|------------------------|------------------------|
| Constante                                | <b>NB_MAX</b>          | <b>NB_MAX</b>          |
| Type ( <b>typedef</b> , <b>enum</b> ...) | <b>CollectionTriee</b> | <b>CollectionTriee</b> |
| Classe                                   | <b>Noeud</b>           | <b>Noeud</b>           |
| Variable / Attribut                      | <b>NombreElements</b>  | <b>nombreElements</b>  |
| Fonction / Méthode                       | <b>IsolerMot</b>       | <b>isolerMot</b>       |

© INSA - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON





## Guide de Style INSA-IF – Classe

- ◆ Organisation
  - **O-1** Distinguer dans chaque classe la partie *interface* (**MaClasse.h**) qui est exportée de la partie *réalisation* (**MaClasse.cpp**) qui est privée.
  - **C-1** Construire de petites classes bien ciblées et à la sémantique cohérente plutôt qu'une grande classe qui fait tout.
  - **O-18** Avant de commencer l'écriture d'une classe à caractère général, examiner la liste des classes de votre bibliothèque : par bonheur, elle existe peut-être déjà...
- ◆ Constructeur
  - **O-6** Chaque constructeur doit initialiser **tous** les attributs de l'objet  
⇒ état stable et connu pour un objet.
  - **C-3** Le constructeur doit allouer **lui-même** toutes les zones dynamiques de l'objet.
  - **O-11** Une classe dont un attribut est alloué dynamiquement dans le constructeur doit avoir un constructeur de copie et doit définir l'opérateur d'affectation (Cf. Forme canonique).



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

167



## Guide de Style INSA-IF – Classe

- ◆ Destructeur
  - **C-4** Le destructeur doit libérer toutes les zones allouées dans le constructeur.
  - **C-6** La destruction de tous les objets créés est impérative pour que la place mémoire soit récupérée. Pour s'en assurer, on pourra compter les créations et les destructions de toutes les instances de la classe et de ses descendantes.
- ◆ Attribut
  - **O-4** Ne pas déclarer d'attribut public.
  - **E-14** Ne pas initialiser un attribut par l'intermédiaire d'un autre attribut.
  - **C-12** Ne pas rendre systématiquement les attributs accessibles publiquement et surtout ne pas fournir systématiquement de méthodes pour les modifier (**Get** et **Set**). Cette pratique viole de fait l'encapsulation et diminue les capacités d'extension.



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

168



## Guide de Style INSA-IF – Classe

- ◆ Forme canonique
  - **O-11** Définir au minimum les méthodes de la forme canonique afin de pouvoir manipuler un objet d'une classe de façon similaire à un objet d'un type de base.
  - **O-12** Toujours définir un constructeur de copie et un opérateur d'affectation.
  - **O-13** Définir un constructeur de copie erroné lorsque son utilisation est illicite (déclaration sans définition). La même chose peut être réalisée pour l'opérateur d'affectation.
  - **O-14** Pour l'opérateur d'affectation, attention aux écritures  
 $x = x;$
- ◆ Déclaration des méthodes
  - **O-8** L'en-tête de chaque méthode publique contient une description de la fonction réalisée citant tous les paramètres de l'en-tête et précisant les valeurs de retour. **Il faut dire ce que fait la méthode et non pas comment elle le fait.**



## Guide de Style INSA-IF – Classe

- ◆ Déclaration des méthodes (suite)
  - **C-8** Déclarer les méthodes avec le statut **const** si elles ne modifient pas les attributs de l'objet afin de permettre leur utilisation sur des objets **const**.
  - **C-10** Ne pas utiliser le passage par valeur pour les classes sans besoin justifié (problème de coût).
- ◆ Pointeur
  - **T-26** Dans une méthode ne pas écrire **this->attribut** ou **this->méthode** puisque tout accès aux attributs et méthodes est sous-entendu "de la classe".
  - **C-5** Une méthode ne doit pas détruire un objet passé en paramètre sinon l'appelant se retrouve avec un objet inutilisable.
- ◆ Erreur courante
  - **E-6** Dans une méthode, on ne doit pas renvoyer l'adresse d'une variable locale. Celle-ci étant gérée dynamiquement, à la sortie de la méthode la place occupée par cette variable est considérée comme libre.



## ***Programmation Orientée Objet – C++***

**PLAN du COURS**



**Naufrage**

- ...
- Classe



**Divin ?**

- Héritage
  - ◆ Héritage en C++ et Visibilité
  - ◆ Constructeurs / Destructeurs et Héritage
  - ◆ Affection et Héritage
  - ◆ Spécialisation par Redéfinition
  - ◆ Liaison Statique et Dynamique
  - ◆ Polymorphisme
  - ◆ Classe Abstraite



**Frisson**



**Bricolage**

**INSA** INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
© MM - Reproduction interdite sans l'autorisation de l'auteur

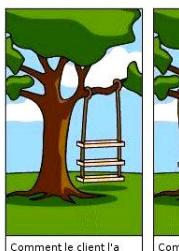
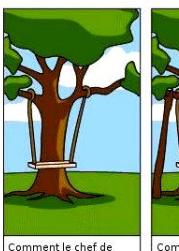
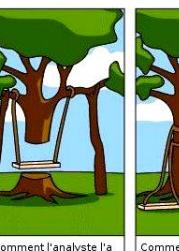
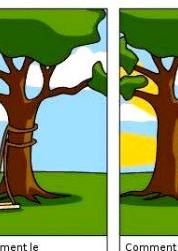
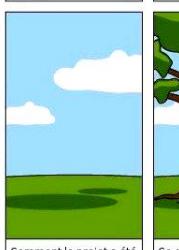
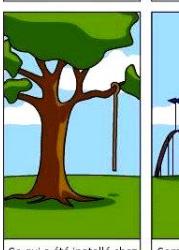
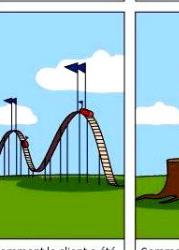
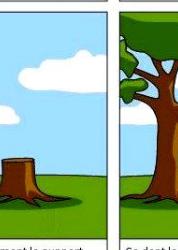


171



## ***Héritage***

**Réalisation d'un projet informatique en SSII**

|                                                                                                                              |                                                                                                                                  |                                                                                                                              |                                                                                                                                      |                                                                                                                                       |
|------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
|  <p>Comment le client l'a décrit</p>      |  <p>Comment le chef de projet l'a compris</p> |  <p>Comment l'analyste l'a schématisé</p> |  <p>Comment le programmeur l'a écrit</p>          |  <p>Comment le business consultant l'a décrit</p> |
|  <p>Comment le projet a été documenté</p> |  <p>Ce qui a été installé chez le client</p>  |  <p>Comment le client a été facturé</p>   |  <p>Comment le support technique est effectué</p> |  <p>Ce dont le client avait réellement besoin</p> |

© MM - Reproduction interdite sans l'autorisation de l'auteur

## Généralités sur l'Héritage

- Conception par spécialisation
  - ◆ Classes conçues par spécialisation de classes plus générales
  - ◆ Relation **EstUn** ou **SorteDe** (héritage **public**)
- Intérêt de l'héritage
  - ◆ Évite la redéfinition dans une classe dérivée du comportement des classes ancêtres
  - ◆ ⇒ Permet de se préoccuper du **comportement particulier** à la classe
- Organisation des classes selon une hiérarchie
  - ◆ Arbre d'héritage, en cas d'héritage simple
  - ◆ Graphe d'héritage, en cas d'héritage multiple
- Notation graphique

**INSA** INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

↳ On dit que :  
 B hérite ou dérive de A  
**B EstUn A ou que B est une SorteDe A**  
 B est une classe descendante ou dérivée de A  
 A est une classe ancêtre ou de base de B

173

© MM - Reproduction interdite sans l'autorisation de l'auteur

## Généralités sur l'Héritage

- Sémantique de l'héritage
 

Principe de **substitution**  
 Si on a : **A  $\Leftarrow \dots \Leftarrow B$**  (B hérite de A) alors  
**B** doit pouvoir s'employer partout où **A** est employé  
**Attention** : l'inverse n'est pas vrai
- Sous-type : relation de préordre sur les types
  - ◆ Les descendants d'une classe **A** en sont des sous-types
  - ◆ Les opérations définies dans la classe **A** sont applicables

Principe du **sous-typage**  
 Si on a : **A  $\Leftarrow B \Leftarrow C \Leftarrow \dots$**  alors  
 tout objet instance d'une classe héritant de **A** (donc  
 de classe **A, B, C, ...**) est de type **A**

174

© MM - Reproduction interdite sans l'autorisation de l'auteur

**INSA** INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON



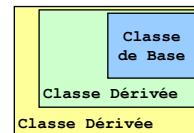
## Héritage en C++

### ■ Syntaxe

```
class ClasseDérivée : [mode] ClasseDeBase
{ // Compléments d'interface pour la classe dérivée
};
```

- ◆ **ClasseDérivée** : nom de la classe dérivée ⇒ identificateur au sens du C++
- ◆ **ClasseDeBase** : nom de la classe de base (ou ancêtre) ⇒ identificateur au sens du C++
- ◆ **mode** : conditionne la visibilité des éléments de la classe de base depuis la classe dérivée
  - **public** (cas le plus fréquent), **protected** ou **private**
  - Valeur par défaut : **private**

La classe de base implémente en principe des caractéristiques générales, qui sont aussi des caractéristiques des classes dérivées, alors que les classes dérivées se contentent de définir des spécialisations de la classe de base



© INSA - Reproduction interdite sans l'autorisation de l'auteur

175



## Héritage Public

### ■ Syntaxe

```
class B : public A
{ // Compléments d'interface pour la classe dérivée
};
```

- ◆ L'héritage dit public est la forme la plus importante d'héritage en C++
- ◆ Pour l'utilisateur de la classe **B**, les membres de la classe **A** conservent leur visibilité :
  - Les membres publics de **A** sont également des membres publics de **B**
  - Les membres protégés de **A** sont également des membres protégés de **B**
  - Les membres privés de **A** ne sont pas accessibles dans **B**

L'héritage **public** correspond à une relation de type "**est un**". Dire qu'une classe **B** dérive publiquement d'une classe **A** équivaut à dire que **B** est un **A**.



© INSA - Reproduction interdite sans l'autorisation de l'auteur

176

## Héritage Public



- Exemple

```
class Oiseau
{
 // Description d'un oiseau
};

class Aigle : public Oiseau
{
 // Description spécifique d'un aigle
};
```

⇒ Un **Aigle est un Oiseau**, donc l'héritage **public** est de mise

⇒ Tout ce qui s'applique à un **Oiseau** est également applicable à un **Aigle**, **mais pas le contraire !**

⇒ L'**Aigle** définit des **spécificités** qui ne sont applicables qu'à lui-même :

- Ainsi, l'**Aigle** est un **rapace**
- Ce qui ne veut pas dire que tous les **Oiseaux** sont des rapaces !

**INSA** | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
177
© MM - Reproduction interdite sans l'autorisation de l'auteur

## Héritage Privé



- Syntaxe

```
class B : private A
{
 // Compléments d'interface pour la classe dérivée
};
```

- ◆ Pour l'utilisateur de la classe **B**, tous les membres de la classe **A** deviennent **private**
  - Cependant, il est possible d'indiquer explicitement, par des déclarations d'accès, les membres **public** de **A** que l'on souhaite laisser **public**, donc accessibles aux clients de **B**
- ◆ L'héritage **private** correspond à une relation du type "**est implémenté sous forme de**" ⇒ utilisation de l'héritage pour une réutilisation de code et non de comportement
- ◆ L'héritage **private** peut souvent être remplacé par une relation de contenance (attribut privé de l'objet)

178
© MM - Reproduction interdite sans l'autorisation de l'auteur



## Héritage Privé

- Exemple

⇒ La dérivation n'implique rien de précis pour l'utilisateur de la classe **CarnetAdresse** ⇒ L'héritage **private** n'a d'importance que pour le concepteur de la classe dérivée

```
class Liste
{
 // Description de la classe de base
};

class CarnetAdresse : private Liste
{
 // Description de la classe dérivée
};
```

⇒ **Attention** : Un client de la classe **CarnetAdresse** n'aura pas la possibilité d'accéder à la classe de base  
 ⇒ Tout est privé dans la classe dérivée à cause de l'héritage !



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

179



## Visibilité des Éléments Hérités

- Résumé

| VISIBILITÉ<br>des données                                           | Mot-clé utilisé pour l'héritage |           |           |           |
|---------------------------------------------------------------------|---------------------------------|-----------|-----------|-----------|
|                                                                     | public                          | protected | private   |           |
| Mot-clé<br>utilisé pour<br>les attributs<br>et pour les<br>méthodes | public                          | publique  | protégée  | privée    |
|                                                                     | protected                       | protégée  | protégée  | privée    |
|                                                                     | private                         | interdite | interdite | interdite |

Les données publiques d'une classe mère deviennent soit publiques, soit protégées, soit privées selon que la classe fille hérite en public, protégé ou en privé. Les données privées de la classe mère sont toujours inaccessibles, et les données protégées deviennent soit protégées, soit privées.

- ◆ Exemple de chaînes d'héritage



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

180



## Visibilité des Éléments Hérités

- Exemple : chaîne d'héritage **public**

```
class A
{
 private :
 int donneePrivee;

 protected :
 int donneeProtegee;

 public :
 int DonneePublique;
};
```

```
class B1 : public A
{
 void M1 ()
 {
 DonneePublique = 1; // OK : membre public
 donneeProtegee = 2; // OK : membre protégé
 donneePrivee = 3; // Erreur : membre privé
 }
};

class B2 : public B1
{
 void M2 ()
 {
 DonneePublique = 4; // OK : membre public
 donneeProtegee = 5; // OK : membre protégé
 donneePrivee = 6; // Erreur : membre privé
 }
};
```

© INI - Reproduction interdite sans l'autorisation de l'auteur

⇒ Conservation des visibilités des membres à travers l'héritage


INI INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
181



## Visibilité des Éléments Hérités

- Exemple : chaîne d'héritage **private**

```
class A
{
 private :
 int donneePrivee;

 protected :
 int donneeProtegee;

 public :
 int DonneePublique;
};
```

```
class B1 : private A
{
 void M1 ()
 {
 DonneePublique = 1; // OK : membre public
 donneeProtegee = 2; // OK : membre protégé
 donneePrivee = 3; // Erreur : membre privé
 }
};

class B2 : private B1
{
 void M2 ()
 {
 DonneePublique = 4; // Erreur : membre privé
 donneeProtegee = 5; // Erreur : membre privé
 donneePrivee = 6; // Erreur : membre privé
 }
};
```

© INI - Reproduction interdite sans l'autorisation de l'auteur

⇒ L'utilisation d'un héritage **public** ici ne change rien aux erreurs

⇒ Membre **private** à cause du premier héritage :

```
class B1 : private A
```

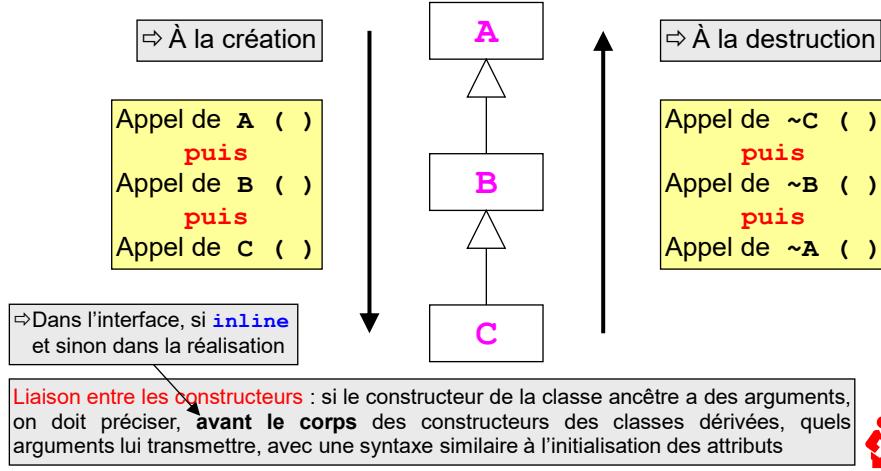

INI INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
182



## Constructeurs / Destructeurs et Héritage

### ■ Enchaînement des constructeurs / destructeurs

⇒ Pour un objet de la classe C



© MM - Reproduction interdite sans l'autorisation de l'auteur



183



M



M



## Exemple d'Héritage

```

#ifndef _PIXEL_H_
#define _PIXEL_H_
#include "Point.h"
class Pixel : public Point // Un Pixel est un Point (ou une sorte de)
{
public :
 enum Couleur { ROUGE, VERT, BLEU };

 Pixel (int abs, int ord, Couleur laCouleur = ROUGE)
 : Point (abs, ord), c (laCouleur)
 { // La construction d'un Pixel s'appuie sur la construction d'un Point
 // Initialisation de l'attribut c de Pixel
 }

 void Colorier (Couleur laCouleur = BLEU)
 { // Méthode spécifique à un Pixel
 c = laCouleur;
 // Les méthodes d'un Point, comme par exemple
 // Déplacer, Modifier et même Afficher
 // (affichage partiel), s'appliquent à un Pixel
 }

protected :
 Couleur c; // Attribut spécifique à un Pixel
};

#endif // _PIXEL_H_

```

Annotations:

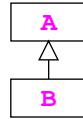
- ⇒ Héritage `public`: relation de type "est un"
- ⇒ Utilisation du constructeur de `Point`
- ⇒ Initialisation de l'attribut `c` de `Pixel`
- ⇒ Les méthodes d'un `Point`, comme par exemple `Déplacer`, `Modifier` et même `Afficher` (affichage partiel), s'appliquent à un `Pixel`
- ⇒ Les attributs `x` et `y` d'un `Point` caractérisent aussi un `Pixel`: un `Pixel` est donc un `Point` auquel on rajoute une couleur

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Constructeur de Copie et Héritage

- Enchaînement des constructeurs de copie
  - ◆ Fonction de l'**existence ou non** d'un constructeur de copie dans la classe ancêtre **A** et dans la classe dérivée **B**
- Règles
  - ◆ La classe dérivée **B** n'a pas de constructeur de copie...
    - Appel au constructeur de copie par défaut de la classe **B**,
    - Qui appelle par défaut le constructeur de copie de l'ancêtre **A**
    - Si la classe ancêtre **A** n'a pas de constructeur de copie...
      - ⇒ Utilisation du constructeur de copie par défaut de l'ancêtre **A**
  - ◆ La classe dérivée **B** a un constructeur de copie...
    - Si ce constructeur de copie appelle le constructeur de copie de l'ancêtre **A** dans son en-tête
 
$$\text{B} \left( \text{const } \text{B} \& \text{unB} \right) : \text{A} \left( \dots \right) \{ \dots \}$$
    - Alors le constructeur de copie de l'ancêtre **A** est appelé
    - Sinon, le **constructeur sans argument** de l'ancêtre **A** est appelé



© MM - Reproduction interdite sans l'autorisation de l'auteur

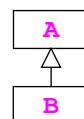


INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



## Affectation et Héritage

- Enchaînement des opérateurs d'affectation
  - ◆ Supposons que **B** hérite de **A** et considérons l'affectation entre objets, instances de **B**
- Règles
  - ◆ Si la classe **B** surcharge l'opérateur d'affectation (**operator** `=`), on ne fera appel qu'à ce dernier
  - ◆ Si la classe **B** ne surcharge pas l'opérateur d'affectation,
    - On fera appel à l'opérateur d'affectation de la classe ancêtre **A**, surchargé ou par défaut, pour les attributs hérités de **A**
    - On fera appel à l'affectation par défaut pour les attributs spécifiques de la classe dérivée **B**



© MM - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON





## Affectation d'Objets

- Soit la hiérarchie simple...

```
class C : public B { ... };
class B : public A { ... };
class A { ... };
```

- ◆ Alors tout objet, instance de **A**, **B** ou **C** est de type **A** (ou de type compatible avec **A**) ⇒ Principe du sous-typage
- ◆ **Conséquence** : on peut affecter un objet instance de **B** ou de **C** à un objet instance de **A**, **mais pas l'inverse !**

- Exemple

⇒ Le type **Couleur** est défini dans l'espace de noms de **Pixel** ⇒ il faut préciser cette appartenance dans l'écriture des valeurs de type **Couleur**

```
// Soit les classes Point et Pixel définies précédemment
Point p (1, 2);
Pixel pi (3, 4, Pixel::BLEU);
p = pi; // Du descendant vers l'ancêtre ⇒ affectation autorisée
// p reste un Point et p.x vaut 3 et p.y vaut 4 (affectation partielle)
pi = p; // De l'ancêtre vers le descendant ⇒ affectation interdite
```



© INSA - Reproduction interdite sans l'autorisation de l'auteur

187



## Affectation de Pointeurs d'Objets

- Exemple

```
// Soit les classes Point et Pixel définies précédemment
Point p (1, 2);
Point *pp;
Pixel *ppi;
PP = new Pixel (8, 9, Pixel::VERT);
// Affectation autorisée, car on peut toujours affecter à un pointeur de Point,
// l'adresse d'un objet d'une classe descendante (donc de Pixel)
pp->Colorier (Pixel::ROUGE);
// Appel interdit, pp reste un pointeur sur un objet de la classe Point
// ⇒ la méthode Colorier n'est donc pas définie
ppi = &p;
// Affectation interdite, l'attribut spécifique à Pixel (c) n'est pas affecté
```

⇒ Que donnerait l'invocation de la méthode  
Colorier à partir du pointeur ppi ?  
ppi->Colorier ( Pixel::Bleu );



© INSA - Reproduction interdite sans l'autorisation de l'auteur

188





## Spécialisation par Redéfinition

```
#if ! defined (POINT_H)
#define POINT_H

class Point
{
public :
 virtual void Afficher (const char *message = "Point : ") const
 // Affiche les caractéristiques du Point précédées d'un message optionnel
 {
 cout << message << x << "," << y << endl;
 } //---- Fin de Afficher

 Point (int abs = 0, int ord = 0);
 // Construit un Point à partir de 2 entiers

 Point (const Point & unPoint);
 // Construit un Point à partir d'un Point existant (unPoint)

 virtual ~Point ();
 // Détruit un Point

protected :
 int x; // Les attributs x et y du point sont protégés
 int y;
};

#endif // ! defined (POINT_H)
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Spécialisation par Redéfinition

```
#if ! defined (PIXEL_H)
#define PIXEL_H
#include "Point.h"

class Pixel : public Point
{
public :
 enum Couleur { ROUGE, VERT, BLEU };

 void Afficher (const char *message = "Pixel : ") const
 // Affiche les caractéristiques d'un Pixel précédées d'un message optionnel
 {
 cout << message << x << "," << y << endl;
 cout << "couleur : " << c << endl;
 } //---- Fin de Afficher
```

⇒ Redéfinition de la méthode  
Afficher dans la classe Pixel  
⇒ Spécialisation de l'affichage

⇒ x et y sont visibles dans le descendant  
⇒ Attributs protected et héritage public

```
Pixel (int abs, int ord, Couleur laCouleur = ROUGE)
 : Point (abs, ord), c (laCouleur) {}
```

// La construction d'un Pixel s'appuie sur la construction d'un Point

```
virtual ~Pixel ();
// Détruit un Pixel
```

```
protected :
 Couleur c; // Attribut spécifique
};
```

⇒ On aurait pu réutiliser la méthode Afficher de l'ancêtre  
avec le corps suivant :

```
{
 Point::Afficher (message);
 cout << "couleur : " << c << endl;
} //---- Fin de Afficher
```

© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Liaison Statique / Dynamique

### ■ Exemple : utilisation du mot-clé **virtual**

```
// Soit les classes Point et Pixel définies précédemment
Point *pp = new Point (7, 13);
pp->Afficher ();
// Appel de la méthode Point::Afficher (), ce qui est tout à fait normal
// puisque pp est un pointeur sur un objet de type Point
...
pp = new Pixel (5, 17, Pixel::VERT);
pp->Afficher (); // Appel de la méthode Pixel::Afficher (), bien que pp soit un pointeur
// sur un objet de type Point
// => Liaison dynamique (late binding) à cause de la déclaration virtual de
// Afficher dans la classe de base
```

Si la méthode **Afficher** de **Point** n'était pas déclarée / définie **virtual**, on aurait eu une liaison statique (*early binding*) et donc un appel à **Point::Afficher ()** au lieu de **Pixel::Afficher ()**

### ■ Syntaxe corrigée : déclaration d'une méthode

|                                                                               |                                                                 |                                                                                                                                                   |
|-------------------------------------------------------------------------------|-----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>[ virtual ] type NomMéthode ( [ ParamètresFormels ] ) [ const ];</code> | ↳ <b>virtual</b> obligatoire pour obtenir une liaison dynamique | ↳ <b>NomMéthode</b> ne peut être redéfinie dans les descendants qu'avec la même liste de paramètres et le même type de retour ⇒ la même signature |
|-------------------------------------------------------------------------------|-----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|



© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Liaison Statique / Dynamique

### ■ Danger d'une liaison statique

```
class Point
{
public :
 ...
 void Afficher (const char *message = "Point : ");
 void Deplacer (int dx, int dy)
 {
 x += dx;
 y += dy;
 Afficher ("Point Deplace : ");
 } //--- Fin de Deplacer
 ...
};

class Pixel : public Point
{
public :
 ...
 void Afficher (const char *message = "Pixel");
 void Colorier (...);
 Pixel (...);
 ...
};
```



```
Point p (3, 6);
Pixel pi (5, 10, Pixel::VERT);

p.Afficher ();
pi.Afficher ();

p.Deplacer (1, 1);
pi.Deplacer (1, 1);
```

⇒ Attention : c'est la méthode **Point::Afficher ()** qui a été appellée ⇒ liaison statique à cause de l'absence du mot-clé **virtual** (déclaration de **Afficher** dans **Point**)



© AMI - Reproduction interdite sans l'autorisation de l'auteur



## Polymorphisme

### ■ Présentation

- ◆ Concept issu de la théorie des types
- ◆ Notion très importante en programmation orientée objets
- ◆ Possible en C++ grâce à l'héritage **public** et à la liaison dynamique
- ◆ Grâce au polymorphisme, des éléments d'un type général identique peuvent se comporter différemment en fonction de leur implémentation
- ◆ Autrement dit, une même opération peut se comporter différemment pour différentes classes issues d'une même arborescence d'héritage



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON



© INSA - Reproduction interdite sans l'autorisation de l'auteur

193



## Polymorphisme

### ■ Exemple

```
class Oiseau
{
 public :
 ...
 virtual void Vole ();
 ...
};

class Colombe : public Oiseau { ... };
class Perroquet : public Oiseau { ... };
class Aigle : public Oiseau { ... };
class Poulet : public Oiseau { ... };
class Autruche : public Oiseau
{
 ...
 void Vole ()
 {
 cerr << "Les autruches ne volent pas !" << endl;
 }
 ...
};
```

⇒ Colombe, Aigle... héritent publiquement de Oiseau et sont à ce titre de type Oiseau  
 ⇒ Rappel : Le contraire n'est pas vrai !  
 ⇒ un Oiseau n'est pas une Colombe !

⇒ Conséquence : Quiconque a besoin d'un objet de type Oiseau peut également utiliser un objet de type Autruche..., mais si un objet de type Poulet est requis, seul un objet de type Poulet pourra faire l'affaire

⇒ Corollaire : Une fonction prenant comme argument un objet de type Oiseau peut également utiliser un objet de type Autruche OU Poulet... mais pas l'inverse !

⇒ Polymorphisme : il exprime que des éléments (Colombe, Aigle, Autruche...) d'un type général identique (Oiseau) peuvent se comporter différemment selon la manière dont ils ont été générés  
 ⇒ Colombe et Aigle sont des oiseaux qui volent alors que l'Autruche est un oiseau qui ne vole pas (redéfinition de Vole) !



© INSA - Reproduction interdite sans l'autorisation de l'auteur

194



## Implémentation et Polymorphisme

### ■ Les problèmes...

- ◆ Une classe de base n'a pas forcément la même taille que sa classe dérivée, de taille supérieure ou à la rigueur égale
- ◆ Une fonction acceptant une instance de la classe de base doit également pouvoir accepter n'importe quelle instance d'une classe dérivée publiquement ⇒ Il faudrait être en mesure, lors de l'exécution, de calculer la taille de la pile en fonction de l'instance effectivement passée
- ◆ Lors du passage d'un paramètre par valeur, le compilateur effectue une copie de l'instance dans la pile, en utilisant le constructeur de copie ⇒ Comme l'instance effectivement transmise n'est connue qu'au moment de l'exécution, il faudrait que le compilateur génère du code capable d'effectuer ces diverses opérations lors de l'exécution

#### Conséquence

Le polymorphisme n'est pas utilisable avec le passage de paramètres par valeur à cause du surcroît de traitement, lors de l'exécution, qu'il engendrerait pour résoudre ces problèmes



## Implémentation et Polymorphisme

### ■ Exemple

⇒ Résultat de l'exécution  
B::Affiche () appelé  
⇒ Liaison dynamique

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

class A
{
public :
 virtual void Affiche () const
 { cout << "A::Affiche () appelé" << endl; }
};

class B : public A
{
public :
 void Affiche () const // Redéfinition de Affiche
 { cout << "B::Affiche () appelé" << endl; }
};
```

⇒ Le compilateur a dû générer une copie de l'instance unB passée comme paramètre dans la pile ⇒ pour des raisons d'optimisation, il s'est servi de la partie A de l'objet unB pour générer une copie de type A (ce n'est pas le cas pour le passage par référence ou pointeur)

```
void f (const A & unA)
{
 unA.Affiche ();
}

int main ()
{
 B unB;
 f (unB);
 return 0;
}
```

```
void f (const A *unA)
{
 unA->Affiche ();
}

int main ()
{
 A *unB = new B;
 // instancié comme un B !
 f (unB);
 return 0;
}
```

```
void f (A unA)
{
 unA.Affiche ();
}

int main ()
{
 B unB;
 f (unB);
 return 0;
}
```

A  
B

Copie dans la pile  
instance unB

⇒ Résultat de l'exécution  
A::Affiche () appelé  
⇒ C'est une copie de A qui est générée dans la pile (passage par valeur d'unB)



## Modélisation et Polymorphisme

### ■ Faiblesse du modèle **Oiseau**

- ◆ Le modèle **Oiseau** proclame que...
  - Les oiseaux savent voler (méthode **Vole** de la classe **Oiseau**)
  - Les autruches sont des oiseaux (héritage **public** de **Oiseau**)
  - En conséquence, les autruches savent voler !
  - Mais, le fait d'essayer de voler pour une autruche est **une erreur** !
    - ⇒ Redéfinition de la méthode **Vole** avec un message d'erreur
- ◆ En fait, le modèle défini est insuffisant pour les oiseaux...
  - **Dire que les oiseaux savent voler est faux !**
  - ⇒ Il faut élaborer un modèle plus sophistiqué



## Modélisation et Polymorphisme

### ■ Modèle sophistiqué

```
class Oiseau
{
 // Pas de méthode Vole définie
};

class OiseauVolant : public Oiseau
{
 public :
 virtual void Vole () ;
};

class OiseauNonVolant : public Oiseau
{
 // Pas de méthode Vole définie
};

class Autruche : public OiseauNonVolant
{
 // Pas de méthode Vole définie
};

class Aigle : public OiseauVolant
{
 // Méthode Vole définie par OiseauVolant, éventuellement redéfinie par Aigle
};

... autruche.Vole () ; // Erreur détectée par le compilateur
```

⇒ Dans ce modèle, il est parfaitement clair qu'il existe des oiseaux sachant voler et des oiseaux ne sachant pas voler

⇒ Tous deux appartiennent à la catégorie plus générale des oiseaux

⇒ Mais l'**Autruche** n'appartient qu'à la catégorie des oiseaux ne sachant pas voler

⇒ L'**Aigle** appartient, quant à lui, à la catégorie des oiseaux sachant voler

⇒ Toute tentative de faire voler une autruche générera une erreur, non plus à l'exécution, mais déjà à la compilation !



## Modélisation et Polymorphisme

### Rappel : intérêt de `virtual`

```
class Oiseau
{
 // Pas de méthode Vole définie
};

class OiseauVolant : public Oiseau
{
 public :
 virtual void Vole () ;
};

class Aigle : public OiseauVolant
{
 void Vole () ; // Comportement adapté
};

class Colibri : public OiseauVolant
{
 void Vole () ; // Comportement adapté
};
...
```

⇒ Redéfinition de `Vole` par les classes dérivées pour coller au mieux au comportement effectif des objets (spécialisation)

⇒ L'héritage `public` correspond à une relation de type "est un"  
 ⇒ Toutes les méthodes applicables à la classe de base le sont aussi à la classe dérivée  
 ⇒ Réutilisation complète de l'interface de la classe de base, **mais pas forcément de l'implémentation**... à cause de `virtual`

⇒ La classe `Aigle` et la classe `Colibri` héritent de l'interface de la classe `OiseauVolant` ⇒ il existe une méthode `Vole ()` qui leur permet de voler !  
 ⇒ Mais là s'arrête la similitude puisque la façon de voler de chacun de ces oiseaux est très différente ⇒ on dira que leur implémentation de la méthode `Vole ()` diffère

⇒ Mise à disposition d'une implémentation par défaut pour la méthode `Vole`  
 ⇒ Redéfinition (obligatoire) de `Vole` et possible grâce à l'utilisation de mot-clé `virtual`

© MM - Reproduction interdite sans l'autorisation de l'auteur



## Méthode Virtuelle Pure

### Le contexte pour l'application `Oiseau...`

- ◆ Déclaration d'une méthode `virtual Vole` pour la classe `OiseauVolant`, avec sa définition (réalisation)
- ◆ En cas d'oubli de redéfinition dans un descendant :
  - Le comportement par défaut est utilisé (celui de l'ancêtre)
  - Mais, ce comportement par défaut n'est pas forcément adéquat pour le descendant

`virtual void Vole () = 0;`

### Le remède : la méthode virtuelle pure

- ◆ Obliger l'utilisateur à toujours redéfinir la méthode `Vole` dans les descendants en modifiant sa déclaration dans l'ancêtre (`=0`)
- Suppression possible mais non obligatoire de la définition dans l'ancêtre ⇒ Invocation explicite avec l'opérateur de portée `::`
- ◆ La vérification de la redéfinition de la méthode `Vole` dans chaque descendant est alors à la charge du compilateur

© MM - Reproduction interdite sans l'autorisation de l'auteur



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
LYON





## Classe Abstraite

- Le contexte pour l'application Oiseau...
  - ◆ La méthode virtuelle pure **Volé** n'a pas de réalisation associée dans la classe **OiseauVolant**
  - ◆ L'invocation de cette méthode n'est donc plus possible
  - ◆ Logiquement, la définition d'une instance de **OiseauVolant** ne doit plus être possible puisque son comportement sera forcément incomplet
  - ◆ ⇒ La classe **OiseauVolant** est devenue une **classe abstraite**
    - Définition d'une interface à l'usage des descendants
    - Non instanciable ⇒ Vérification faite par le compilateur

⇒ En C++, une classe est considérée comme abstraite, si sa définition contient au moins une méthode virtuelle pure

```
virtual type NomMéthode ([paramètres]) [const] = 0;
```

© MM - Reproduction interdite sans l'autorisation de l'auteur


## Guide de Style INSA-IF – Héritage

- ◆ Destructeur
  - **O-7** Pour autoriser les descendants à exécuter leur propre destructeur, il faut toujours déclarer un destructeur et de plus il faut le déclarer **virtual**.
- ◆ Classe abstraite
  - **O-24** Une classe qui ne comporte pas de méthode virtuelle pure est instanciable. Si c'est en réalité une classe abstraite, au sens où il ne faut pas l'instancier, détecter la création non autorisée d'une instance en déclarant le constructeur **protected** : il ne sera accessible que par les descendants qui peuvent donc encore l'appeler dans leur propre constructeur.

© MM - Reproduction interdite sans l'autorisation de l'auteur
