Programmation avancée en C++
GIF-1003

Module 2 : Implantation de classe

T. Eude

UNIVERSITÉ LAVAL
Département d'informatique
et de génie logiciel

2. Implantation de classe

2.1 LE MODÈLE OBJET

Département
d'informatique -
Université Laval

UNIVERSITÉ LAVAL

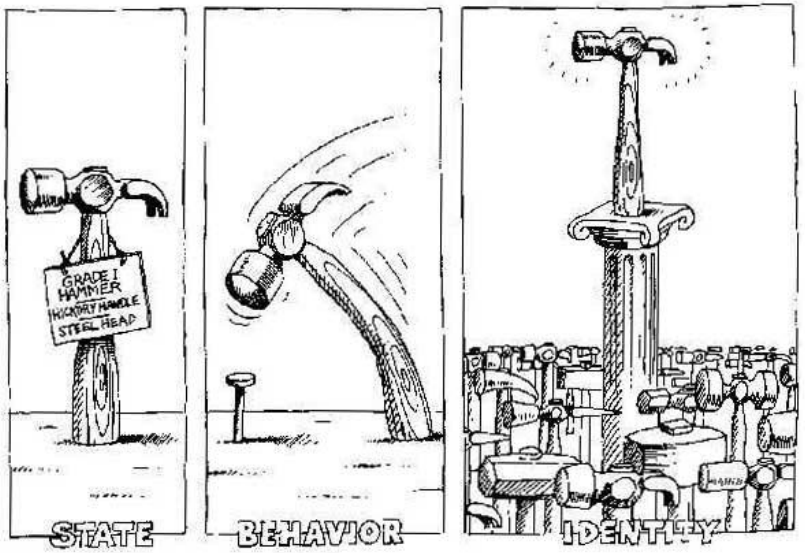
Les composantes de l'objet

État de l'objet :

- Valeurs de ses attributs ou de l'information qu'il contient.

Comportement de l'objet

- Ensemble des opérations que l'on peut effectuer sur celui-ci.

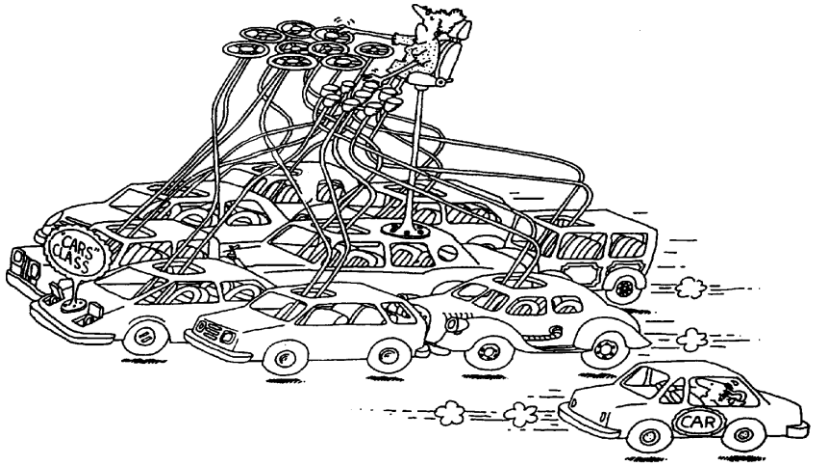


Identité de l'objet

- Deux objets peuvent être exactement dans le même état, mais sont différents de part leur identité unique

✓ex.: adresse en mémoire.

La classe ?



Objet = entité discrète qui
existe dans le temps
et dans l'espace.

Classe = ***abstraction*** qui n'existe que dans les programmes.

- ***doit*** représenter un ensemble d'objets qui partagent une ***structure commune*** et un ***comportement commun***.
- définit
  - les ***opérations*** permises sur les objets de la classe.
  - les ***états*** qu'ils peuvent prendre (transitions d'états).

instance, ***occurrence*** et ***objet*** sont synonymes

Interface et implémentation

L'***interface*** d'une classe

- définit ce qui est vu de l'***extérieur***.
  - ➢renforce la notion d'abstraction en ne montrant que ce qui est ***nécessaire***.
- contient principalement les déclarations de toutes les opérations applicables sur les instances de cette classe.

L'***implémentation*** d'une classe

est ce qui est caché à l'***intérieur***,

- principalement le code réalisant les opérations définies dans l'interface.

Interface d'une classe : exemple

```
class Pile
{
public:
          Pile  (int p_capacite);
          ~Pile ();
    int   pop   ();
    void  push  (int);
    int   top   () const;
    void  init  ();
private:
    ...
};
```

Synthèse

Objet

Classe

Interface

2.2 La classe

Département
d'informatique -
Université Laval

UNIVERSITÉ LAVAL

La classe et ses objets

La classe

- responsable d'assurer la validité de tous les objets créés à partir d'elle-même.

Un objet doit être ***valide*** de sa **création** jusqu'à sa **destruction**.

➢ ne pas permettre l'accès direct aux attributs!

➔ Encapsulation

La théorie du contrat pourrait gérer cette validité (nous y reviendrons).

Principe d'encapsulation

Problème :

Accès aux attributs **non contrôlée**.

- Possibilité de non-initialisation.
- Possibilité de mauvaise assignation.

Pas de garantie sur la **validité** des données.

Implémentation d'une classe : les étapes

Déclaration de la classe:

- attributs dans la section **privée**
- *méthodes* dans la section **publique**
  - ✓ Norme : Toujours spécifier la visibilité explicitement.

Implantation des méthodes

- après la déclaration de la classe
  - ➢ dans un fichier séparé (pour le C++).

Utiliser la classe

  - ➢ en créant des objets de ce type dans un programme.

Déclarer les attributs

État d'un objet : décrit par ses attributs:

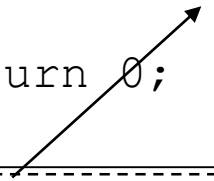
```
class Date
{
    // ...
private:
    int m_jour;
    int m_mois;
    int m_annee;
};
```

- ✓ Attributs déclarés **privés**
  - ➢ On ne peut y accéder directement.

Exemple : classe Date

```
class Date
{
private:
    int m_jour;
    int m_mois;
    int m_annee;
};
```

```
int main()
{
    Date uneDate;

    uneDate.m_jour = 30;
    uneDate.m_mois = 3;
    uneDate.m_annee = 1965;

    return 0;
}
```



Accès direct impossible !!!
Les attributs sont déclarés **privés.**

Déclarer puis implanter les méthodes

Rôle des **méthodes** de la classe :

La classe *Date* n'accepte pour valeur qu'un sous-ensemble des valeurs possibles.

- **contrôler** l'assignation des valeurs sur les champs.

- Exemple:

31/02/1965 n'est pas une date valide.

Objet courant

On peut référer à l'objet courant explicitement

- ✓en général ce n'est pas de mise.

En C++, **this** permet de référer au pointeur à l'**objet courant**.

```
void Date::initialise(int p_jour, int p_mois, int p_annee)
{
    this->m_jour = p_jour;
    this->m_mois = p_mois;
    this->m_annee = p_annee;
}
```

```
int main()
{
    Date uneDate;
    uneDate.initialise(30, 3, 1965);

    return 0;
}
```

Synthèse

Classe

- -
- 1
- 2

Initialisation

création d'un objet,

- ses attributs doivent être initialisés pour que celui-ci soit dans un état cohérent.
  - ➢ constructeur.

constructeur = méthode spéciale.

- a le même nom que la classe.
- est automatiquement appelé à la création d'un objet de la classe.
- méthode qui n'a pas de type de retour
- ne retourne pas de valeur.

Les constructeurs

Protection des attributs :

```
Date uneDate;
uneDate.m_jour = 31; uneDate.m_mois = 3; uneDate.m_annee = 1965;
```

Impossible !

- Par contre, on peut faire :

```
Date uneDate;
uneDate.initialise(31,3,1965);
```

  - Problème : pendant un certain temps l'objet Date n'est pas correctement initialisé.
  - Solution: utiliser la méthode spéciale qu'est le constructeur

Les constructeurs

```
class Date
{
public:
   Date(int p_jour, int p_mois, int p_annee);
   ...
};

Date::Date(int p_jour, int p_mois, int p_annee)
{
   m_jour = p_jour;
   m_mois = p_mois;
   m_annee = p_annee;
}
```

- Constructeur automatiquement appelé lorsqu'un objet Date est déclaré:
  `Date uneDate(1, 1, 1997);`
- ✓ Par contre, on ne peut plus écrire:
  `Date uneDate;`

Les constructeurs

Il est possible d'avoir plusieurs constructeurs dans une même classe.

Caractéristique du langage :

- Plusieurs méthodes du même nom
  = **surcharge de méthode**.

Lorsqu'une fonction surchargée est appelée, le compilateur sélectionne la bonne fonction selon les arguments: nombre, type, ordre..

Les constructeurs

On peut vouloir en même temps :

- un constructeur qui accepte la date complète
- un ***constructeur*** ***par défaut***
  - ➢constructeur auquel on ne fournit pas de donnée.

Exemple :

```
class Date
{
public:
    Date();
    Date(int p_jour, int p_mois, int p_annee);
    …
};
```

Les constructeurs

Mais aussi que :

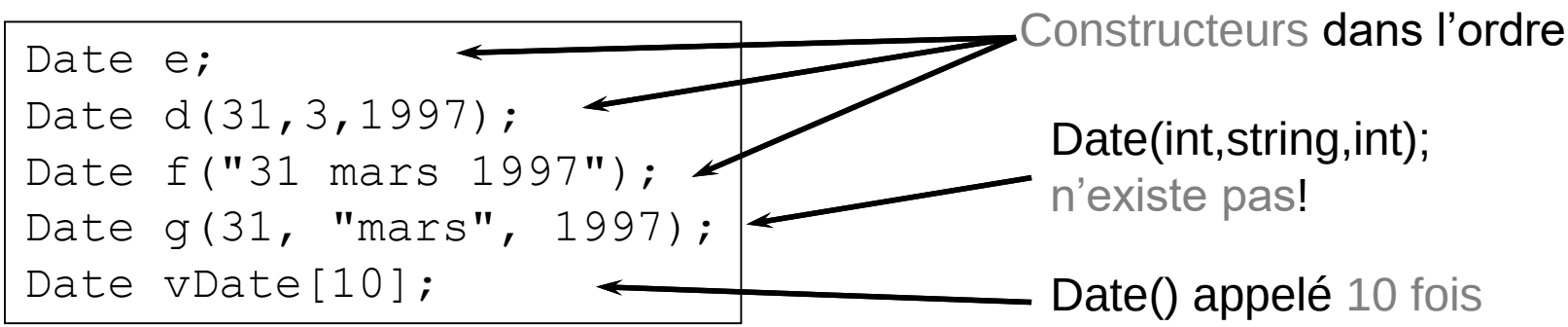
- Le constructeur par défaut de la classe Date puisse prendre la date courante du système en guise de date.
- Un constructeur permettant d'initialiser une date à partir d'une chaîne de caractères d'un format déterminé:

```
Date(const std::string& p_txtDate);
```

Les constructeurs : exemple

```
class Date
{
public:
    Date();
    Date(int p_jour, int p_mois, int p_annee);
    Date(const std::string& p_txtDate);
};
```

```
Date e;
Date d(31,3,1997);
Date f("31 mars 1997");
Date g(31, "mars", 1997);
Date vDate[10];
```



Construction des sous-objets

Utilisez une liste d'initialisation: construire tous les sous-objets avant le corps du constructeur parce que lorsqu'on y est rendu, ces objets sont déjà construit, i.e. on a déjà appelé leur constructeur par défaut.

```
Date::Date(int p_jour, int p_mois, int p_annee)
{
   // Les attributs ont déjà été initialisés par défaut.
   m_jour = p_jour;
   m_mois = p_mois;
   m_annee = p_annee;
}
```

Liste d'initialisation

```
Date::Date(int p_jour, int p_mois, int p_annee)
: m_jour(p_jour), m_mois(p_mois), m_annee(p_annee)
{
}
```

Construction des sous-objets

Exemple 2 :

```
Personne::Personne (const string& p_nom,
       const string& p_prenom, const string& p_adresse)
: m_nom(p_nom), m_prenom(p_prenom), m_adresse(p_adresse)
{
}
```

avec

```
string m_nom
string m_prenom
Adresse m_adresse
```

Construire un objet avec des valeurs par défaut pour ensuite les changer avec les valeurs désirées.

- Un appel de plus est fait inutilement.
- Dans de nombreux cas, erreur de compilation

Le destructeur

méthode particulière de la classe.

porte le même nom que la classe mais précédé par un tilde ~.

est exactement le complément du constructeur

- Le constructeur initialise les attributs et alloue des ressources s'il y a lieu,
- le destructeur libère les ressources si nécessaire.

Ne détruit pas l'objet:

- il fait le ménage à l'intérieur avant la destruction.

Le destructeur

Ne reçoit pas de paramètre et ne retourne pas de valeur.

Un seul par classe.

Facultatif:

- Le compilateur en génère un qui ne fait rien.

Est utile lorsque l'objet est responsable de ressources allouées dynamiquement.

- Ne pas implanter de destructeur s'il n'y a pas de désallocation de ressource.

Constructeur / Destructeur

```
class Date
{
public:
    Date(int p_jour, int p_mois, int p_annee);
    ~Date();
private:
    int m_jour;
    int m_mois;
    int m_annee;
};
Date::Date(int p_jour, int p_mois, int p_annee)
: m_jour(p_jour), m_mois(p_mois), m_annee(p_annee)
{
}
Date::~Date()
{
}
```

Initialisation seulement

Rien à faire? => ne pas implanter

```
int main()
{
    Date uneDate(1, 1, 1965);

    return 0;
}
```

Date déclarée
Constructeur appelé

Sortie du «scope»,
destructeur appelé

Synthèse

Constructeur :

Destructeur :

2. Concepts orientés objets

2.1 Le modèle objet

2.2 La classe

2.3 Méthodes par catégorie

Département d'informatique - Université Laval

UNIVERSITÉ LAVAL

Méthodes par catégorie

Accès

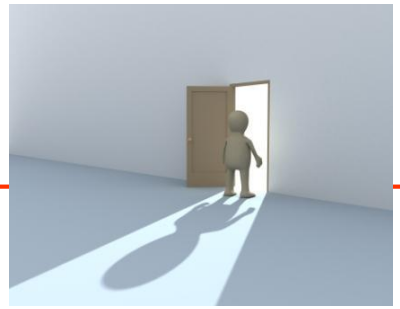
Assignation

Comparaison

Utilitaires

Statiques

• Méthode d'accès



```
class Date
{
public:
    ...
    int reqJour  () const;
    int reqMois  () const;
    int reqAnnee () const;
    ...
};
```

Accéder à l'information sans modifier ➔ méthodes constantes.

Méthodes constantes : protection des données

méthodes d'accès en lecture

vs

méthodes qui modifient l'état de l'objet.

- traitement spécial on doit les déclarer **constantes**.

- En C++, mot clé : const
- Indique l'intention de ne pas modifier l'objet dans cette méthode

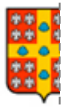
Méthodes constantes : exemple

```
class Date
{
public:
   ...
   int reqJour  () const;
   int reqMois  () const;
   int reqAnnee () const;
   ...
};
```

```
int Date::reqJour () const
{
   return m_jour;
}
```

Le mot-clé **const** se retrouve dans l'**interface** et dans l'**implantation**

Le compilateur fera la vérification que les attributs de l'objet ne seront pas modifiés dans cette méthode.



Méthodes constantes

Outil permettant d'améliorer la qualité du code de façon très importante

- il force la classification.

Il faut adhérer à cette façon de faire si on désire partager du code et utiliser des librairies qui ont adhéré à cette norme.

La librairie standard adhère à cette norme.

Méthodes constantes : exemple

```
class Date
{
public:
   void imprime();
private:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

```
void Date::imprime()
{
   cout << m_jour << ":" << m_mois << ":"
        << m_annee;
}
```

Ne compile pas!
le compilateur ne peut deviner que vous ne modifiez pas la date en l'imprimant...

```
class Message
{
public:
   void imprime() const;
private:
   string m_expediteur;
   Date   m_dateRecu;
};
```

```
void Message::imprime() const
{
   cout << m_expediteur << endl
        << m_dateRecu.imprime();
}
```

• Méthodes d'assignation

```
class Date
{
public:
   void asgJour  (int p_jour);
   void asgMois  (int p_mois);
   void asgAnnee (int p_annee);
private:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

```
void Date::asgJour(int p_jour)
{
   m_jour = p_jour;
}
void Date::asgMois(int p_mois)
{
   m_mois = p_mois;
}
void Date::asgAnnee(int p_annee)
{
   m_annee = p_annee;
}
```

💣 objet dans un état incohérent.

➢ `asgDate(int p_jour,int p_mois,int p_annee);` préférable!

→ contrôle des changements sur les attributs de la date sans jamais avoir d'état intermédiaire invalide.

Méthode d'assignation : exemple

```
class Date
{
public:
   void asgDate(int p_jour, int p_mois, int p_annee);
private:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

```
void Date::asgDate(int p_jour, int p_mois, int p_annee)
{
   m_jour = p_jour;
   m_mois = p_mois;
   m_annee = p_annee;
}
```

On pourra vérifier la validité du format des données

Méthode de comparaison

Implantation d'un nouveau type

- Nécessité de pouvoir comparer celui-ci par rapport à un objet du même type.

Exemple :

une Date comparée à une autre Date:

- Est-elle égale ?
- Est-elle inférieure ?

Ce type de méthode retourne un booléen.

Méthode de comparaison

```
class Date
{
public:
   bool estEgal(const Date& p_date) const;
private:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

```
bool Date::estEgal(const Date& p_date) const
{
   return m_jour == p_date.m_jour &&
          m_mois == p_date.m_mois &&
          m_annee == p_date.m_annee;
}
```

Évaluation de l'égalité sur la base de la valeur des attributs de l'**objet courant** par rapport à l'objet passé en paramètre.

```
int main()
{
   Date date1(4,9,2002);
   Date date2;
   if (date2.estEgal(date1))
   …
}
```

Synthèse

Méthodes utilitaires

Rappel: en général,

- toutes les méthodes publiques
- tous les attributs privés.

Méthodes utilitaires :

- à l'intérieur de la classe sans pour autant vouloir en faire un service publique.
- sont appelées par les autres méthodes de la classe.

Méthode utilitaire : exemple

```
class Date
{
public:
   ...
   std::string reqDateFormatee() const;
   ...
private:
   std::string reqNomMois() const;
};
```

Date unDate(2,5,1994);
cout << uneDate.reqDateFormatee() << endl;
Résultat : 2 mai 1994

```
std::string Date::reqDateFormatee () const
{
   ostringstream os;
   os << reqJour() << " "
      << reqNomMois() << " "
      << reqAnnee();
   return os.str();
}
```

Méthode utilitaire
utilisée à l'intérieur de la classe

```
std::string Date::reqNomMois () const
{
   static string NomMois[] = {"janvier", "février", "mars", "avril",
                              "mai", "juin", "juillet", "août",
                              "septembre", "octobre","novembre", "décembre"};
   return NomMois[reqMois()-1];
}
```

Méthodes statiques

Méthode de classe ne nécessitant pas la présence d'un objet pour le traitement

- méthode statique.

Comme une fonction mais associée à la classe.

Ne touche pas aux attributs de la classe, à moins que ceux-ci ne soient aussi statiques (attribut commun à tous les objets de la classe).

Méthodes statiques : exemple

```
--- INTERFACE ---
class Date
{
public:
          void asgDate       (int p_jour, int p_mois, int p_annee);
   static bool valideDate    (int p_jour, int p_mois, int p_annee);
   static bool estBissextile (int p_annee);
private:
   …
};
```

Pas de const pour une méthode statique

La méthode peut être déclarée Statique puisque aucun attribut non statique n'est utilisé

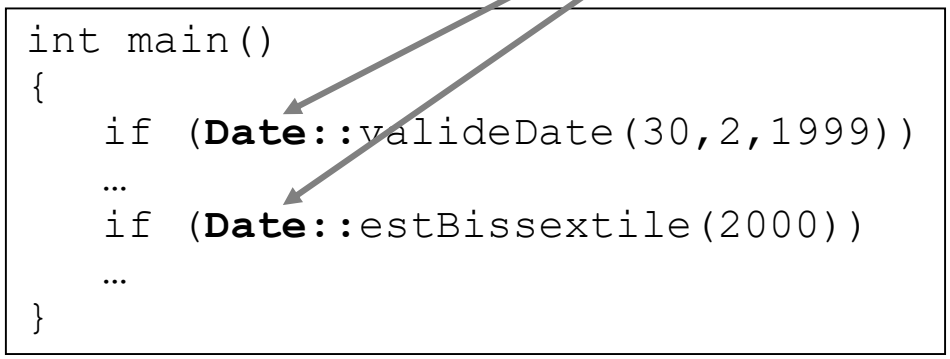
```
--- IMPLÉMENTATION ---
int Date::jourParMois[12] =
{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

bool Date::valideDate(int p_jour, int p_mois, int p_annee)
{
   return p_jour > 0 && p_mois > 0 && p_mois <= 12 &&
          ((p_mois == 2 && p_jour == 29 &&
           Date::estBissextile(p_annee))
           || (j < jourParMois[p_mois - 1]));
}
bool Date::estBissextile(int p_annee)
{
   return (((p_annee % 4 == 0) && (p_annee % 100 != 0)) ||
           ((p_annee % 4 == 0) && (p_annee % 100 == 0) &&
           (p_annee % 400 == 0)) );
}
```

Méthodes statiques

N'a pas besoin d'un objet du type de la classe pour être appelée.

Norme :

- ✓Toujours appeler les méthodes statiques avec le «scope» de la classe,



Méthode et attribut statiques : exemple

Etudiant.h

```
class Etudiant
{
public:
   Etudiant  () {m_nbInstances++;}
   ~Etudiant () {m_nbInstances--;}
   static int reqNbInstances () {return m_nbInstances;}
private:
   static int m_nbInstances;
};
```

Etudiant.cpp

```
int Etudiant::m_nbInstances=0;
```

Test.cpp

```
int main()
{
   {
      Etudiant et1, et2, et3;
      cout << "Résultat :" << endl;
      cout << Etudiant::reqNbInstances() << endl;
      Etudiant et4, et5;
      cout << Etudiant::reqNbInstances() << endl;
   }
   cout << Etudiant::reqNbInstances() << endl;
   return 0;
}
```

Attribut statique d'un objet

ne peut qu'utiliser des attributs statiques.

Attribut commun à tous les objets de la classe

Initialisation dans Etudiant.cpp

Résultat:
3
5
0

On sort du "scope": appel au destructeur

Synthèse

2. Concepts orientés objets
    2.1 Le modèle objet
    2.2 La classe
    2.3 Méthodes par catégorie

2.4 Implantation avancée

Département d'informatique - Université Laval

UNIVERSITÉ LAVAL

Implantation avancée

Problème de collision de noms

Surcharge de méthode

Le passage de paramètres

Le retour

Problème de collision de noms…

Qu'arrive-t-il si 2 fonctions globales différentes ont le même prototype ?

```
// MaLib1.h
int f();
```

```
// MaLib2.h
int f();
class A{};
```

```
#include "MaLib1.h"
#include "MaLib2.h"

int g(){
   return 5 + f();   //Quel f() ?
}
```

Solution : créer de namespaces

Namespace :

- regroupement logique de fonctions, de classes, etc.
- Possibilité de lui donner un nom.
  - NomDuNamespace::NomDeLEntite

Pas d'ambiguïté

exemple

```
// MaLib1.h
namespace util{
    int f();
}
```

```
// MaLib2.h
namespace tp1{
    int f();
    class A{};
}
```

```
#include "MaLib1.h"
#include "MaLib2.h"

int g(){
    tp1::A a;
    return 5 + util::f();    // Ah, ok
}
```

On se sert de l'opérateur d'appartenance ::

using namespace

namespace directement accessible :

- commande **using**

```
#include "MaLib1.h"
#include "MaLib2.h"

using namespace tp1;

int h(){
    A a;
    int x = f();
    return x + util::f();
}
```

Par le using, c'est tp1::f().

Bon usage du using

Inutile de mettre des using pour chacun des namespaces disponibles :

- l'ambiguïté **revient**.

Un using peut-être local :

```
// Quelque part
  {
    using namespace util;
    // util est accessible uniquement dans ce bloc
  }
```

Ne jamais utiliser de using dans un .h :

- tous les fichiers qui l'incluront seront *pollués*.

Namespace et librairie standard

librairie standard en C++ :

- dans le namespace **std :**
  - Fonctions I/O (console, fichiers, ...)
  - Classes de base (string, iterator, ...)
  - Conteneurs (vector, list, ...)
  - Algorithmes standardisés (tri, recherche, …)

souvent la commande

```
using namespace std;
```

dans les fichiers **.cpp** .

Exemple : classe Bidon

On peut rajouter quelque chose dans un namespace déjà défini ailleurs.

Bidon.h

```
#include "MaLib1.h"
#include "MaLib2.h"
#include <string>

namespace util
{
  class Bidon
  {
  public :
    std::string afficher()const;
  private:
    Tp1::A m_a;
    std::string m_tmp;
  };
}  // --- namespace util
```

Bidon.cpp

```
#include "Bidon.h"

namespace util
{

std::string Bidon::afficher()const
{
  string s;
  // traitement à faire
  return s;
}
}  // --- namespace util
```

Surcharge de méthode

Les langages orientés objet permettent une première forme de polymorphisme: la surcharge de méthode (ou de fonction).

Il est permis de créer plusieurs méthodes du même nom dans la classe pourvu qu'elles varient au niveau des paramètres.

Lorsqu'une méthode surchargée est appelée, le compilateur sélectionne la bonne selon les paramètres: nombre, type, ordre.

La surcharge doit être utilisée pour implanter des traitements similaires.

Surcharge de méthode (2)

Prenons un exemple d'une string:

```
class string
{
public:
    (…)
    int rechercher (char c) const;
    int rechercher (const char* sP) const;
    int rechercher (const string& s) const;
    (…)
};
```

Surcharge de méthode (3)

Attention: le compilateur utilise seulement la liste des paramètres pour distinguer les fonctions de même nom.

Le type de retour n'est pas pris en compte.

```
class Quelconque
{
public:
    (…)
    bool   methode1 (int p1, int p2);
    double methode1 (int p1, int p2);
    (…)
};
```

Pour le compilateur, ces deux méthodes sont identiques.

Paramètre par défaut (1)

Parfois, certaines méthodes ont besoins de plusieurs paramètres pour correctement faire le travail.

Toutefois pour les cas simples ou les plus fréquents, ces paramètres supplémentaires sont toujours les mêmes.

Il est possible d'utiliser la technique des paramètres par défaut.

Paramètre par défaut (2)

Soit une fonction qui permet d'imprimer un entier dans n'importe quelle base : 2, 10, 16…

La plupart du temps se sera en base 10.

```
void imprimer (int valeur, int base = 10);

int main()
{
    imprimer (31);
    imprimer (31, 10);
    imprimer (31, 16);
    imprimer (31, 2);
    return 0;
}
```

```
Résultat :
31 31 1F 11111
```

Paramètre par défaut (3)

Une autre implantation aurait pu utiliser la surcharge de fonction à la place :

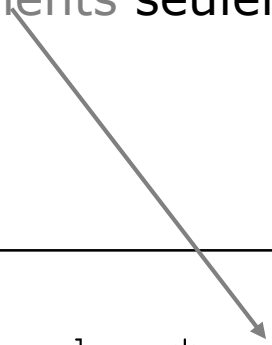
```
void imprimer (int valeur, int base);
void imprimer (int valeur)
{
    imprimer(valeur, 10);
}
int main()
{
    imprimer (31);
    imprimer (31, 10);
    imprimer (31, 16);
    imprimer (31, 2);
    return 0;
}
```

```
Résultat :
31 31 1F 11111
```

Paramètre par défaut (4)

Les paramètres par défaut sont vérifiés à la compilation et évalués à l'exécution.

Les paramètres par défaut doivent être fournis à partir des derniers arguments seulement.



Correct :

```
int foo(int a, int b=0, char* strP=0);
```

Incorrect :

```
int goo(int a=0, int b=0, char* strP);
int hoo(int a=0, int b, char* strP=0);
```

Surcharge de méthode et paramètre par défaut

l'utilisation de la surcharge de méthode peut entrer en conflit avec l'utilisation des paramètres par défaut…

```
class A
{
public:
    A();
    A(const A& obj);
    A(int v1=0, int v2=0);

private:
    int m_v1;
    int m_v2;
};
```

Que peut-on dire à propos de ces constructeurs ?

Il y aura conflit entre A() et A(int v1=0, int v2=0).

Surcharge des opérateurs

Définition:

Consiste à redéfinir des opérateurs usuels comme +, -, *, /, =, <, [], etc...
sur de nouveaux types : les classes.

Particulièrement intéressante avec
- les nombres complexes, les fractions, les vecteurs, les matrices, les chaînes de caractères...

Dans certaines situations, rend la lecture du code plus facile :

```
d = d1.plus(d2) --> d = d1 + d2
```

Listes des opérateurs permis

Liste complète des opérateurs qui peuvent être surchargés :

```
+    -    *    /    %    ^    &    |    ~    !

=    <    >    +=   -=   *=   /=   %=   ^=   &=

|=   <<   >>   <<=  >>=  ==   !=   <=   >=   &&

||   ++   --   ->   ->*  []   ()   ,
```

Ce qui n'est pas permis

On ne peut pas surcharger les opérateurs :

. .* :: ?:

On ne peut pas créer de nouveaux opérateurs :

|x| i.e. valeur absolue

y := x i.e. assignation à la Pascal

y = x**2 i.e. x à la puissance 2.

Définir un opérateur surchargé

En C++,

mot-clé operator suivi de l'opérateur.

Les règles de préséance et d'associativité sont les mêmes que les opérateurs usuels.

- respecter ces règles pour ne pas avoir des résultats allant contre l'intuition.

Il faut éviter les surprises et la confusion.

- Le langage ne renforce pas la signification d'un opérateur donné.

💣il est possible de définir l'opérateur + qui réalise une soustraction... À éviter!

Exemples

```
class Date
{
public:
   bool operator== (const Date& p_date) const;
   bool operator<  (const Date& p_date) const;
   friend std::ostream& operator<<(std::ostream& p_os,
                                   const Date& p_date);
private:
   long m_temps;
};
bool Date::operator==(const Date& p_date) const
{
   return m_temps == p_date.m_temps;
}
bool Date::operator<(const Date& p_date) const
{
   return m_temps < p_date.m_temps;
}
```

Exemples

```
std::ostream& operator<<(std::ostream& p_os, const Date& p_date)
{
   p_os << p_date.m_temps;
   return p_os;
}

// --- Exemple d'utilisation
int main()
{
   Date d1;
   Date d2(4,11,2001);
   bool bEgal = (d1 == d2);      //--- d1.operator==(d2)
   bool bInferieur = (d1 < d2); //--- d1.operator<(d2)
   cout << d1 << endl;           //--- operator<<(cout, d1)
   cout << d2 << endl;           //--- operator<<(cout, d2)

   return 0;
}
```

Opérateurs complémentaires

Le langage C++ ne rend pas équivalent les opérateurs `v += w` et `v = v + w`.

- Il faut définir les opérateurs `+=`, `=` et `+`.

`x++` pas nécessairement égal à `x = x + 1`

La règle :

- définir un ensemble complet d'opérateurs qui ont des interactions naturelles entre eux :
  `a = a + b --> a += b`.

Opérateur ++

```
Test de l'operateur de pre-increment :
  d4 est 18/03/1969
++d4 est 19/03/1969
  d4 est 19/03/1969
Test de l'operateur de post-increment :
  d4 est 19/03/1969
d4++ est 19/03/1969
  d4 est 20/03/1969
```

```
int main()
{
 Date d4(18, 3, 1969);

 cout << "Test de l'operateur de pre-increment :" << endl
      << "  d4 est" << d4 << endl;
 cout << "++d4 est" << ++d4 << endl;
 cout << "  d4 est" << d4 << endl;

 cout << "Test de l'operateur de post-increment :" << endl
 << "  d4 est" << d4 << endl;
 cout << "d4++ est" << d4++ << endl;
 cout << "  d4 est" << d4 << endl;

 return 0;
}
```

Opérateur ++

```
class Date
{
public:
Date();
…
Date& operator++();//pré-incrément
Date operator++(int p_increment);//post-incrément
…
```

Opérateur ++

```
//++d
Date& Date::operator ++()
{
 this->ajouteNbJour(1);
 return *this;
}

//d++
Date Date::operator ++(int p_increment)
{
 Date temp = *this;
 this->ajouteNbJour(1);
 return temp;
 //retour par valeur
}
```

retour par référence

sauvegarde de l'état courant de l'objet

retour de la copie de l'objet avant appel

Le passage de paramètre

Types de passage de paramètres en C++ ?

trois types :

- le passage par valeur,
- le passage par pointeur,
- le passage par référence (nouveauté par rapport au C).

Quand utiliser chacun de ces types?

- Le passage par valeur

Permet de passer des valeurs sans que la fonction appelée puisse modifier celles-ci.

- pas d'«effet de bord»
- Les modifications sur les données passées n'ont pas d'influence sur les données originales.

Coût de la copie lorsqu'on passe par valeur.

Le passage par valeur : Exemple

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

string MettreEnMajuscule (string p_texte)
{
   unsigned int dim = p_texte.size();
   for (int i=0; i<dim; i++)
   {
      p_texte[i] = (char)toupper(p_texte[i]);
   }
   return p_texte;
}

int main()
{
   string nom = "programmation";
   string NOM = MettreEnMajuscule(nom);

   cout << "Minuscule : " << nom << endl;
   cout << "Majuscule : " << NOM << endl;

   return 0;
}
```

Résultat:
Minuscule : programmation
Majuscule : PROGRAMMATION

Appel au constructeur copie

Le passage par pointeur

Avec le langage C,

- Pour éviter le coût de la copie
- on passait les paramètres par pointeur.

Approprié lorsque l'on veut que les données soient modifiées.

Mais lorsqu'on ne veut pas …?!

Le passage par pointeur : Exemple

Passage par pointeur :
Accès direct aux données originales

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

string MettreEnMajuscule (string* p_texteP)
{
   unsigned int dim = p_texteP->size();
   for (int i=0; i<dim; i++)
   {
      (*p_texteP)[i] = (char)toupper((*p_texteP)[i]);
   }
   return *p_texteP;
}

int main()
{
   string nom = "programmation";
   string NOM = MettreEnMajuscule(&nom);

   cout << "Minuscule : " << nom << endl;
   cout << "Majuscule : " << NOM << endl;

   return 0;
}
```

Résultat:
Minuscule : PROGRAMMATION
Majuscule : PROGRAMMATION

Le passage par référence

Le passage par référence, tout comme le passage par pointeur, donne directement accès aux données originales.

Pas de coût associé au passage par référence mais il y a risque de corruption des données par les fonctions appelées.

Le passage par référence : exemple

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

string MettreEnMajuscule (string& p_texte)
{
   unsigned int dim = p_texte.size();
   for (int i=0; i<dim; i++)
   {
      p_texte[i] = (char)toupper(p_texte[i]);
   }
   return p_texte;
}

int main()
{
   string nom = "programmation";
   string NOM = MettreEnMajuscule(nom);

   cout << "Minuscule : " << nom << endl;
   cout << "Majuscule : " << NOM << endl;

   return 0;
}
```

Passage par référence :
Accès direct aux données originales

Résultat:
Minuscule : PROGRAMMATION
Majuscule : PROGRAMMATION

Le passage par référence constante

Le langage C++ offre la possibilité de protéger les données originales en utilisant le passage par **référence** ET le mot **const**.

Il n'y a plus de coût associé à la copie des données et il n'y a plus de risque de modification des données originales.

Le passage par référence constante donne le même résultat que le passage par valeur, la copie en moins !

Le passage par référence constante : exemple

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

string MettreEnMajuscule (const string& p_texte)
{
   unsigned int dim = p_texte.size();
   string texte2 = p_texte; //ou string texte2(p_texte);
   for (int i=0; i<dim; i++)
   {
      texte2[i] = (char)toupper(p_texte[i]);
   }
   return texte2;
}

int main()
{
   string nom = "programmation";
   string NOM = MettreEnMajuscule(nom);

   cout << "Minuscule : " << nom << endl;
   cout << "Majuscule : " << NOM << endl;
   return 0;
}
```

Passage par référence constante : Accès direct aux données originales sans pouvoir les modifier.

Résultat:
Minuscule : programmation
Majuscule : PROGRAMMATION

Conclusion Passage de paramètres

Ne **jamais** utiliser le passage par **valeur** sauf pour les types de base du langage.

Toujours utiliser le passage par **référence constante** si on désire que les données originales **ne soient pas modifiées**.

Toujours utiliser le passage par **référence** si on désire que les données originales **soient modifiées**.

Le passage par **pointeur** pourra être utilisé dans quelques rares cas, notamment pour le polymorphisme. (Nous y reviendrons)

Retour par reference ou par valeur?

Par reference

```
Mot& reqAdresse() const;
```

- Risque de retour d'une variable locale
- Non respect de l'encapsulation

Par reference constante

```
const Mot& reqAdresse() const;
```

Par valeur

```
Mot reqAdresse() const;
```

- Pour retourner le contenu d'une variable locale
- Appelle le constructeur copie si c'est un objet (dépend du compilateur **RVO :** Return value optimization)
- Plus simple à utiliser

Synthèse : Implantation avancée

Problème de collision de noms

Surcharge de méthode

Le passage de paramètres

Le retour