LES CLASSES

- **Définition d’une classe**
- **Droits d’accès**
- **Types de classes**
- **Recommandations de style**
- **Définition des fonctions membres**
- **Instanciation d’une classe**
- **Utilisation des objets**
- **Fonctions membres constantes**
- **Exemple complet : pile d’entiers (1)**
- **Constructeurs et destructeurs**
- **Exemple : pile d’entiers (2)**
- **Constructeur copie**
- **Classes imbriquées**
- **Affectation et initialisation**
- **Liste d’initialisation d’un constructeur**
- **Le pointeur *this***
- **Les membres statiques**
- **Classes et fonctions amies**

DEFINITION D’UNE CLASSE

Rappel :

la classe décrit le modéle structurel d'un objet :

- ensemble des attributs (ou champs ou données membres) décrivant sa structure
- ensemble des opérations (ou méthodes ou fonctions membres) qui lui sont applicables.

Une classe en C++ est une structure qui contient :

- des fonctions membres
- des données membres

Les mots réservés *public* et *private* délimitent les sections visibles par l'application.

Exemple :

```
class Avion {
  public :   // fonctions membres publiques
    void init(char [], char *, float);
    void affiche();

  private :  // membres privées
    char  immatriculation[6], *type; // données membres privées
    float poids;
    void  erreur(char *message);     // fonction membre privée
};  // n'oubliez pas ce ; après l'accolade
```

DROITS D'ACCES

L'**encapsulation** consiste à masquer l'accès à certains attributs et méthodes d'une classe.

Elle est réalisée à l'aide des mots clés :

- ***private*** : les membres privés ne sont accessibles que par les fonctions membres de la classe. La partie privée est aussi appelée **réalisation**.

- ***protected*** : les membres protégés sont comme les membres privés. Mais ils sont aussi acessibles par les fonctions membres des classes dérivées (voir l'héritage).

- ***public*** : les membres publics sont accessibles par tous. La partie publique est appelée **interface**.

Les mots réservés *private* , *protected* et *public* peuvent figurer plusieurs fois dans la déclaration de la classe.

Le droit d'accès ne change pas tant qu'un nouveau droit n'est pas spécifié.

TYPES DE CLASSES

- **struct Classe1 { /* ... */ };**
    - tous les membres sont par défaut d'accès public
    - le contrôle d'accès est modifiable
    - Cette structure est conservée pour pouvoir compiler des programmes écrits en C

  Exemple :

```
struct Date {
                        // méthodes publiques (par défaut)
    void set_date(int, int, int);
    void next_date();
    // autres méthodes ....

  private :             // données privées
    int _jour, _mois, _an;
};
```

- **union Classe2 { /* ... */ };**
    - tous les membres sont d'accès public par défaut
    - le contrôle d'accès n'est pas modifiable

- **class Classe3 { /* ... */ };**
    - tous les membres sont d'accès private (par défaut)
    - le contrôle d'accès est modifiable

  C'est cette dernière forme qui est utilisée en programmation objet C++ pour définir des classes.

RECOMMANDATIONS DE STYLE

Mettre :

- la première lettre du nom de la classe en majuscule
- la liste des membres publics en premier
- les noms des méthodes en minuscules
- le caractère _ comme premier caractère du nom d'une donnée membre

---

DEFINITION DES FONCTIONS MEMBRES

- En général, la déclaration d'une classe contient simplement les prototypes des fonctions membres de la classe.

```
class Avion {
  public :
    void init(char [], char *, float);
    void affiche();
  private :
    char  _immatriculation[6], *_type;
    float _poids;
    void  _erreur(char *message);
};
```

- Les fonctions membres sont définies dans un module séparé ou plus loin dans le code source.

  Syntaxe :

```
type_valeur_retournée Classe::nom_fonction( paramètres_formels )
{
   // corps de la fonction
}
```

  Exemple de définition de méthode de la classe Avion :

```
void Avion::init(char m[], char *t, float p) {
  strcpy(_immatriculation, m);
  _type = new char [strlen(t)+1];
  strcpy(_type, t);
  _poids = p;
}

void Avion::affiche() {
  cout << _immatriculation << "  " << _type;
  cout << "  " << _poids << endl;
}
```

- La définition de la méthode peut avoir lieu à l'intérieur de la déclaration de la classe.

Dans ce cas, ces fonctions sont automatiquement traitées par le compilateur comme des fonctions *inline*.

Une fonction membre définie à l'extérieur de la classe peut être aussi qualifiée explicitement de fonction *inline*.

Exemple :

```
class Nombre {
  public :
    void setnbre(int n) { nbre = n; } // fonction inline
    int  getnbre() { return nbre; }   // fonction inline
    void affiche();
  private :
    int nbre;
};

inline void Nombre::affiche() {   // fonction inline
  cout << "Nombre = " << nbre << endl;
}
```

Rappel :

la visibilité d'une fonction *inline* est restreinte au module seul dans laquelle elle est définie.

INSTANCIATION D'UNE CLASSE

De façon similaire à une *struct* ou à une *union*, le nom de la classe représente un nouveau type de donnée.

On peut donc définir des variables de ce nouveau type; on dit alors que vous créez des **objets** ou des **instances** de cette classe.

Exemple :

```
Avion av1;    // une instance simple (statique)
Avion *av2;   // un pointeur (non initialisé)
Avion compagnie[10];  // un tableau d'instances

av2 = new Avion; // création (dynamique) d'une instance
```

UTILISATION DES OBJETS

Après avoir créé une instance (de façon statique ou dynamique) on peut accéder aux attributs et méthodes de la classe.

Cet accès se fait comme pour les structures à l'aide de l'opérateur **.** (point) ou −> (tiret supérieur).

Exemple :

```
av1.init("FGBCD", "TB20", 1.47);
av2->init("FGDEF", "ATR 42", 80.0);
compagnie[0].init("FEFGH","A320", 150.0);

av1.affiche();
av2->affiche();
compagnie[0].affiche();

av1.poids = 0; // erreur, poids est un membre privé
```

FONCTIONS MEMBRES CONSTANTES

Certaines méthodes d'une classe ne doivent (ou ne peuvent) pas modifier les valeurs des données membres de la classe, ni retourner une référence non constante ou un pointeur non constant d'une donnée membre :

on dit que ce sont des fonctions membres constantes.

Ce type de déclaration renforce les contrôles effectués par le compilateur et permet donc une programmation plus sûre sans coût d'exécution. Il est donc **très souhaitable** d'en déclarer aussi souvent que possible dans les classes.

Exemple :

```
class Nombre {
  public :
    void setnbre(int n) { nbre = n; }

    // méthodes constantes
    int  getnbre() const { return nbre; }
    void affiche() const;

  private :
    int nbre;
};
```

```
inline void Nombre::affiche() const {
  cout << "Nombre = " << nbre << endl;
}
```

Une fonction membre *const* peut être appelée sur des objets constants ou pas, alors qu'une fonction membre non constante ne peut être appelée que sur des objets non constants.

Exemple :

```
const Nombre n1;  // une constante

n1.affiche();    // OK
n1.setnbre(15);  // ERREUR: seule les fonctions const peuvent
                 // être appelées pour un objet constant

Nombre n2;
n2.affiche();    // OK
n1.setnbre(15);  // OK
```

Surcharge d'une méthode par une méthode constante :

Une méthode déclarée comme constante permet de surcharger une méthode non constante avec le même nombre de paramètres du même type.

Exemple :

```
class String {
  public :
    void init(char *ch = "");
    // etc ...
    char & nieme(int n);        // (1)
    char   nieme(int n) const;  // (2)

    // etc...
  private :
    char *_str;
};
```

S'il n'y a pas l'attribut *const* dans la deuxième méthode, le compilateur génère une erreur ("String::nieme() cannot be redeclared in class").

Cette façon de faire permet d'appliquer la deuxième méthode *nieme()* à des objets constants et la première méthode *nieme()* à des objets variables :

```
void main() {
  String ch1;
  ch1.init("toto");

  const String ch2;
  ch2.init("coco");
```

```
  cout << ch1.nieme(1);  // appel de la méthode (1)
  cout << ch2.nieme(1);  // appel de la méthode (2)
}
```

L'écriture d'une instruction comme :

ch2.nieme(3) = 'u';

ne sera pas compilable, car la méthode (2) ne retourne pas une référence.

Si pour des raisons d'efficacité, la méthode (2) doit retourner une référence , on retournera alors une référence constante et l'on écrira donc :

```
    const char & nieme(int n) const;   // (2 bis)
```

Et que se passe−t−il si j'écris :

```
    char & nieme(int n) const;   // (2 bis)
    //on retourne une référence non constante
```

au lieu de :

```
    const char & nieme(int n) const;   // (2 bis)
          //on retourne une référence constante
```

le programme suivant se compile parfaitement et donne un résultat surprenant :

```
void main() {
  const String ch2;
  ch2.init("coco");

  ch2.nieme(3) = 'u';

  ch2.affiche(); // affiche "cocu" ! vous n'avez vraiment pas de chance...
}
```

La question maintenant est :

Comment une méthode constante a−t−elle pu modifier la valeur d'un objet constant ?

Quand on déclare une méthode constante, le compilateur vérifie que la méthode ne modifie pas une donnée membre. Et c'est le cas de la méthode *nieme()*. Pour générer une erreur il aurait fallu qu'elle change la valeur de *_str*, chose qu'elle ne fait pas. Elle ne peut changer la valeur que d'un caractère pointé par *_str*.

EXEMPLE COMPLET : PILE D'ENTIERS (1)

```
// IntStack.C : pile d'entiers ------------------------------

#include <iostream.h>
#include <assert.h>
#include <stdlib.h>  // rand()

class IntStack {
  public:
    void init(int taille = 10); // création d'une pile

    void push(int n);   // empile un entier au sommet de la pile
    int pop();          // retourne l'entier au sommet de la pile
    int vide() const;   // vrai, si la pile est vide
    int pleine() const; // vrai, si la pile est pleine
    int getsize() const { return _taille; }
  private:
    int _taille;  // taille de la pile
    int _sommet;  // position de l'entier à empiler
    int *_addr;   // adresse de la pile
};

void IntStack::init(int taille ) {
  _addr = new int [ _taille = taille ];
  assert( _addr != 0 );
  _sommet = 0;
}

void IntStack::push(int n) {
  if ( ! pleine() )
    _addr[ _sommet++ ] = n;
}

int IntStack::pop() {
  return  ( ! vide() ) ? _addr[ --_sommet ] : 0;
}

int IntStack::vide() const {
  return ( _sommet == 0 );
}

int IntStack::pleine() const {
  return ( _sommet == _taille );
}

void main() {
  IntStack pile1;

  pile1.init(15);            // pile de 15 entiers

  while ( ! pile1.pleine() ) // remplissage de la pile
    pile1.push( rand() % 100 );
  while ( ! pile1.vide() ) // Affichage de la pile
    cout << pile1.pop() << " ";
  cout << endl;
}
```

CONSTRUCTEURS ET DESTRUCTEURS

- Les données membres d'une classe ne peuvent pas être initialisées; il faut donc prévoir une méthode d'initialisation de celles−ci (voir la méthode *init* de l'exemple pécédent).

- Si l'on oublie d'appeler cette fonction d'initialisation, le reste n'a plus de sens et il se produira très certainement des surprises facheuses dans la suite de l'exécution.

- De même, après avoir fini d'utiliser l'objet, il est bon de prévoir une méthode permettant de détruire l'objet (libération de la mémoire dynamique ...).

Le **constructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à l'instanciation de l'objet, assurant ainsi une initialisation correcte de l'objet.

Ce constructeur est une fonction qui porte comme nom, le nom de la classe et qui ne retourne pas de valeur (pas même un *void*).

Exemple :

```
class Nombre {
  public :
    Nombre(); // constructeur par défaut
    // ...
  private :
    int _i;
};

Nombre::Nombre() {
  _i = 0;
}
```

On appelle **constructeur par défaut** un constructeur n'ayant pas de paramètre ou ayant des valeurs par défaut pour tous les paramètres.

```
class Nombre {
  public :
    Nombre(int i=0); // constructeur par défaut
    // ...
  private :
    int _i;
};


Nombre::Nombre(int i) {
  _i = i;
```

```
}
```

- si le concepteur de la classe ne spécifie pas de constructeur, le compilateur génèrera un constructeur par défaut.
- comme les autres fonctions, les constructeurs peuvent être surchargés.

```
class Nombre {
  public :
    Nombre();      // constructeur par défaut
    Nombre(int i); // constructeur à 1 paramètre
  private :
    int _i;
};
```

Le constructeur est appelé à l'instanciation de l'objet. Il n'est donc pas appelé quand on définit un pointeur sur un objet ...

Exemple :

```
Nombre n1;     // correct, appel du constructeur par défaut
Nombre n2(10); // correct, appel du constructeur à 1 paramètre
Nombre *ptr1, *ptr2;  // correct, pas d'appel aux constructeurs

ptr1 = new Nombre;  // appel au constructeur par défaut
ptr2 = new Nombre(12); // appel du constructeur à 1 paramètre

Nombre tab1[10]; // chaque objet du tableau est initialisé
                 // par un appel au constructeur par défaut

Nombre tab2[3] = { Nombre(10), Nombre(20), Nombre(30) };
                 // initialisation des 3 objets du tableau
                 // par les nombres 10 20 et 30
```

De la même façon que pour les constructeurs, le **destructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet.

Ce destructeur est une fonction :

- qui porte comme nom, le nom de la classe précédé du caractère (tilda)
- qui ne retourne pas de valeur (pas même un void)
- qui n'accepte aucun paramètre (le destructeur ne peut donc pas être surchargé)

Exemple :

```
class Exemple {
  public :
    // ...
    ~Exemple();
  private :
    // ...
```

```
};

Exemple::~Exemple() {
  // ...
}
```

Comme pour le constructeur, le compilateur génèrera un destructeur par défaut si le concepteur de la classe n'en spécifie pas un.

EXEMPLE : PILE D'ENTIERS AVEC CONSTRUCTEURS ET DESTRUCTEUR

```
// IntStack.C : pile d'entiers ------------------------------

#include <iostream.h>
#include <assert.h>
#include <stdlib.h>  // rand()

class IntStack {
  public:
    IntStack(int taille = 10);     // constructeur par défaut
    ~IntStack() { delete[] _addr; } // destructeur

    void push(int n);   // empile un entier au sommet de la pile
    int pop();          // retourne l'entier au sommet de la pile
    int vide() const;   // vrai, si la pile est vide
    int pleine() const; // vrai, si la pile est pleine
    int getsize() const { return _taille; }
  private:
    int _sommet;
    int _taille;
    int *_addr;  // adresse de la pile
};

IntStack::IntStack(int taille) {
  _addr = new int [ _taille = taille ];
  assert( _addr != 0 );
  _sommet = 0;
}

// ...
// le reste des méthodes est sans changement
//

void main() {
  IntStack pile1(15);        // pile de 15 entiers

  while ( ! pile1.pleine() ) // remplissage de la pile
    pile1.push( rand() % 100 );

  while ( ! pile1.vide() ) // Affichage de la pile
    cout << pile1.pop() << "   ";
  cout << endl;
```

```
}
```

CONSTRUCTEUR COPIE

Présentation du problème :

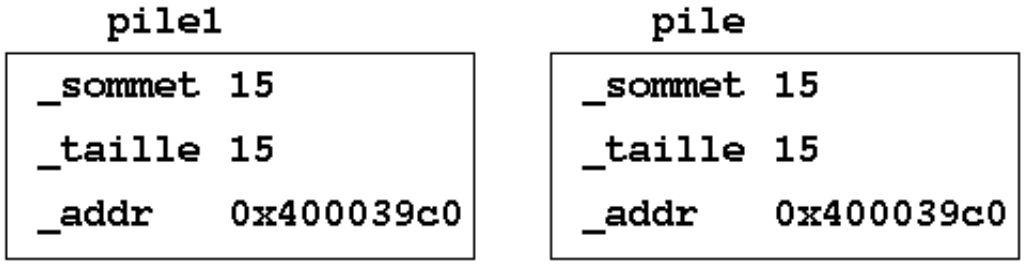
Reprenons la classe *IntStack* avec un constructeur et un destructeur et écrivons une fonction (*AfficheSommet*) qui affiche la valeur de l'entier au sommet de la pile qui est passée en paramètre.

Exemple d'appel de cette fonction :

```
void main() {
  IntStack pile1(15);  // création d'une pile de 15 entiers

  // ...

  AfficheSommet( pile1 );

  // ...
}

void AfficheSommet( IntStack pile ) {
  cout << "Sommet de la pile : " << pile.pop() << endl;
}
```

Ici, la pile est passée par valeur, donc il y a création de l'objet temporaire nommé *pile* créé en copiant les valeurs du paramètre réel *pile1*.

L'affichage de la valeur au sommet de la pile marche bien, mais la fin de cette fonction fait appel au destructeur de l'objet local *pile* qui libère la mèmoire allouée par l'objet *pile1* parce que les données membres de l'objet *pile* contiennent les mêmes valeurs que celles de l'objet *pile1*.



Pour éviter cela :

- il faut définir la fonction *AfficheSommet* comme :

```
void AfficheSommet( IntStack & pile ) {
  cout << "Sommet de la pile : " << pile.pop() << endl;
}
```

  Mais une opération comme *pile.pop()* dépile un entier de la pile *pile1* !!!

- il faut faire une copie intelligente : création du **constructeur de copie**.

- Le constructeur de copie est invoqué à la construction d'un objet à partir d'un objet existant de la même classe.

```
Nombre n1(10); // appel du constructeur à 1 paramètre
Nombre n2(n1); // appel du constructeur de copie
Nombre n3=n1;  // appel du constructeur de copie
```

- Le constructeur de copie est appelé aussi pour le passage d'arguments par valeur et le retour de valeur

```
Nombre traitement(Nombre n) {
  static Nombre nbre;

  // ...

  return nbre;      // appel du constructeur de copie
}

void main() {
  Nombre n1, n2;

  n2 = traitement( n1 ); // appel du constructeur de copie
}
```

- Le compilateur C++ génère par défaut un constructeur de copie "bête"

Constructeur de copie de la classe *IntStack* :

```
class IntStack {
  public:
    IntStack(int taille = 10);       // constructeur par défaut
    IntStack(const IntStack & s);    // constructeur de copie
    ~IntStack() { delete[] _addr; }  // destructeur
    // ...

  private:
    int _taille;  // taille de la pile
    int _sommet;  // position de l'entier à empiler
    int *_addr;   // adresse de la pile
};

// ...
```

```
IntStack::IntStack(const IntStack & s) {  // constructeur de copie
  _addr = new int [ _taille = s._taille ];
  _sommet = s._sommet;
  for (int i=0; i< _sommet; i++)  // recopie des éléments
    _addr[i] = s._addr[i];
}
```

CLASSES IMBRIQUEES

Il est possible de créer une classe par une relation d'appartenance : relation **a un** ou **est composée de**.

Exemple : une voiture a un moteur, a des roues ...

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

class Roue  { /* ... */ };

class Voiture {
  public:
    // ....
  private:
    Moteur _moteur;
    Roue   _roue[4];
    // ....
};
```

AFFECTATION ET INITIALISATION

Le langage C++ fait la différence entre l'**initialisation** et l' **affectation**.

- l'affectation consiste à modifier la valeur d'une variable (et peut avoir lieu plusieurs fois)

- l'initialisation est une opération qui n'a lieu qu'une fois immédiatement après que l'espace mémoire de la variable ait été alloué. Cette opération consiste à donner une valeur initiale à l'objet ainsi créé.

LISTE D'INITIALISATION D'UN CONSTRUCTEUR

Soit la classe :

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

class X {
  public:
    X(int a, int b, Y y);
    ~X();
    // ....
  private:
    const int _x;
    Y _y;
    int _z;
};

X::X(int a, int b, Y y) {
  _x = a; // ERREUR: l'affectation à une constante est interdite
  _z = b; // OK : affectation

  // et comment initialiser l'objet membre _y ???
}
```

Comment initialiser la donnée membre constante _x et appeler le constructeur de la classe Y ?

Réponse : la liste d'initialisation.

La phase d'initialisation de l'objet utilise une liste d'initialisation qui est spécifiée dans la définition du constructeur.

Syntaxe :

```
nom_classe::nom_constructeur( args ... ) : liste_d_initialisation
{
  // corps du constructeur
}
```

Exemple :

```
X::X(int a, int b, Y y) : _x( a ) , _y( y ) , _z( b ) {
  // rien d'autre à faire
}
```

- L'expression *_x(a)* indique au compilateur d'initialiser la donnée membre *_x* avec la valeur du paramètre *a*.

- L'expression *_y(y)* indique au compilateur d'initialiser la donnée membre *_y* par

un appel au constructeur (avec l'argument *y*) de la classe *Y*.

LE POINTEUR *this*

Toute méthode d'une classe X a un paramètre caché : le pointeur *this*.

Celui contient l'adresse de l'objet qui l'a appelé.

Il est implicitement déclaré comme (pour une variable) :

```
X * const this;
```

et comme (pour un objet constant) :

```
const X * const this;
```

et initialisé avec l'adresse de l'objet sur lequel la méthode est appelée.

Il peut être explicitement utilisé :

```
classe X {
  public:
    int f1() { return this->i; }
    // idem que :  int f1() { return i; }

  private:
    int i;
    // ...
};
```

Une fonction membre qui retourne le pointeur *this* peut être chaînée, étant donné que les opérateurs de sélection de membres . (point) et −> (tiret supérieur) sont associatifs de gauche à droite :

exemple :

```
classe X {
  public:
    X f1() { cout << "X "; return *this; }
    // ...
  private:
    // ...
};

void main() {
```

```
  X x;

  x.f1().f1().f1(); // affiche : X X X
}
```

Note :

La fonction membre *f1* a tout intérêt de retourner une référence :

```
 X &f1() { cout << "X "; return *this; }
```

LES MEMBRES STATIQUES

Ces membres sont utiles lorsque l'on a besoin de gérer des données communes aux instances d'une même classe.

- **Données membres statiques :**

  Si l'on déclare une donnée membre comme *static*, elle aura la même valeur pour toutes les instances de cette classe.

  ```
  class Ex1 {
    public:
      Ex1() { nb++; /* ... */ }
      ~Ex1() { nb--; /* ... */ }
    private:
      static int nb;  // initialisation impossible ici
  };
  ```

  L'initialisation de cette donnée membre statique se fera en dehors de la classe et en global par une déclaration :

  ```
  int Ex1::nb = 0;  // initialisation du membre static
  ```

- **Fonctions membres statiques :**

  De même que les données membres statiques, il existe des fonctions membres statiques.
  - ne peuvent accéder qu'aux membres statiques,
  - ne peuvent pas être surchargés,
  - existent même s'il n'y a pas d'instance de la classe.

  ```
  class Ex1 {
  ```

```
  public:
    Ex1() { nb++; /* ... */ }
    ~Ex1() { nb--; /* ... */ }
    static void affiche() {  // fonction membre statique
      cout << nb << endl;
    }
  private:
    static int nb;
};

int Ex1::nb = 0; // initialisation du membre static (en global)

void main() {
  Ex1.affiche();   // affiche 0 (pas d'instance de Ex1)

  Ex1 a, b, c;
  Ex1.affiche();   // affiche 3
  a.affiche();     // affiche 3
}
```

CLASSES ET FONCTIONS AMIES

"Un ami est quelqu'un qui peut toucher vos parties privées."

Dans la définition d'une classe il est possible de désigner des fonctions (ou des classes) à qui on laisse un libre accès à ses membres privés ou protégés.

C'est une infraction aux règles d'encapsulation pour des raisons d'efficacité.

- **Exemple de fonction amie :**

```
class Nombre {

  // ici je désigne les fonctions qui pourront accéder
  // aux membres privés de ma classe
  friend int manipule_nbre(); // fonction amie

  public :
    int getnbre();
    // ...
  private :
    int _nbre;
};


int  manipule_nbre(Nombre n) {
  return n._nbre + 1;  // je peux le faire en tout légalité
                       // parce que je suis une fonction amie
                       // de la classe Nombre.
}
```

- **Exemple de classe amie :**

```
class Window;  // déclaration de la classe Window

class Screen {
  friend class Window;

  public:
    //...
  private :
    //...
};
```

Les fonctions membres de la classe *Window* peuvent accéder aux membres non−publics de la classe *Screen*.
