



## *COURS POO en C++*

Prof. Nacer-Eddine Zergainoh

[Nacer-Eddine.Zergainoh@univ-grenoble-alpes.fr](mailto:Nacer-Eddine.Zergainoh@univ-grenoble-alpes.fr)



1

### *Plan du cours*

- Concepts de la POO (2h)
- Du C au langage C++ (1h)
- Fonctions (1h)
- Classes et Objets (3h)
- Surcharge d'opérateurs (1h)
- Héritage et polymorphisme (3h)
- Flux d'entrées-Sorties (1h)
- Développement de systèmes orientés objets (?h)

2

## Concepts de la POO

- Objectifs
- Evolution du C vers le C++
- Exemple du code source C++
- Concept Objet
- Héritage
- Abstraction de données
- Méthodes de conception orientée objet
- Environnement de développement

3

## *Objectifs*

- ☞ Assimiler les principes de la programmation OO.
- ☞ Maîtriser la POO en C++ , La syntaxe C++.
- ☞ Spécifier, concevoir et implanter en C++ des grandes applications

4

## Evolution du C vers le C++

### - Développement du langage C :

- Le langage C a été développé au début des années 70 par **Dennis RITCHIE** pour permettre la portabilité du système d'exploitation **UNIX** (premier portage d'UNIX en 1975).

#### – Ouvrage de référence :



The C Programming Language  
Brian W. KERNIGHAN & Dennis M. RITCHIE  
Prentice Hall, 1978  
  
(traduit en français chez Masson)

- Il a été normalisé en 1989 sous la référence : *American Standard X3.J159-1989*

#### – Deuxième ouvrage de référence :



- l'utilisation du langage C faisant référence à la syntaxe de l'ouvrage de 1978 est notée **K&R** et celle respectant la norme de 1989 sous forme **ANSI-C**.

The C Programming Language  
(2<sup>nd</sup> edition, ANSI-C)  
Brian W. KERNIGHAN & Dennis M. RITCHIE  
Prentice Hall, 1988  
  
(traduit en français chez Masson)

☞ Dès le début 1980, des ingénieurs ont cherché à faire évoluer le C à la programmation de type "objet"

5

### - Vers le langage C++ :

- Ces travaux sur l'évolution du langage C vers l'objet, lancés au sein des Bell Laboratories (AT&T) début 1980, aboutirent à une première réalisation en 1983 qui prit alors le nom de C++. Le principal artisan de ce travail est Bjarne STROUSTRUP qui publie un premier ouvrage de référence en 1986 :

#### – Ouvrage de référence :



The C++ Programming Language  
Bjarne STROUSTRUP  
Addison Wesley, 1986  
  
(traduit en français chez Addison Wesley)

- Le C++ a subi plusieurs évolutions et un second ouvrage de référence fut publié en 1991 qui constitue aujourd'hui le manuel de base de C++:

#### – Deuxième ouvrage de référence :



The C++ Programming Language  
(2<sup>nd</sup> edition)  
Bjarne STROUSTRUP  
Addison Wesley, 1991  
  
(traduit en français chez Addison Wesley)

- Un autre ouvrage complémentaire indispensable a été publié une première fois en 1990, puis réédité avec des modifications en 1994 pour constituer la base des propositions de normalisation du langage :



The Annotated C++ Reference Manual  
(ANSI Base Document)  
Margaret A. ELLIS, Bjarne STROUSTRUP  
Addison Wesley, 1994

6

– Les liaisons entre le C et le C++ :

- ◆ Le C++ est un sur-ensemble du C
- ◆ Intégration du C dans le C++ (préservation de la syntaxe)
- ◆ Evolution de la syntaxe vers une plus grande rigueur
- ◆ Simplicité et une logique dans la programmation
- ◆ Normalisation ...

7

### Exemple de Code source C++

– Le C++ contient le langage C :

- A priori, presque tout programme C écrit en respectant les normes ANSI-C peut être compilé en C++. Le langage de programmation C++ hérite du support pour la progr. procédurale de C: fonctions, arithmétiques, instructions de sélection et constructions itératives. Comme en C, tout program. C++ contient une fonction `main`. Le fichier `iostream.h` contient les E/S standard.

– Extensions C++

- Importance de la notion de type
- Extensions pour la syntaxe des fonctions
- Extensions pour gérer des "classes" et des "objets"
- De nouvelles librairies : uniquement des librairies de classes
- Gestion orientée "objet" des anomalies

```
/* Commentaire normal du C-ANSI...*/
// commentaire du C++

#include <iostream.h>
int MyFunction(int ,int)
{
    int main(void) ←
    {
        int n,m,z;

        m = 5, n = 4, z = 8;
        int s = m + n + z; //déclaration
                            //d'une variable

        cout << "Somme : " << s << '\n';

        c = MyFunction(s, n);
        cout << "Result :" << c << endl;

        return 0;
    }

    int MyFunction(intx,int y)
    {
        int z;
        z=2*x+3*y;
        return z;
    }
}
```

points communs avec le C :

- présence de la fonction `main`
- les types du C sont disponibles
- toutes les fonctions du C sont accessibles etc...

Somme :17  
MyFunction C: 46

8

## Approche "objet"

- La programmation objets introduit plusieurs notions :

- ◆ Encapsulation
- ◆ Classe
- ◆ Instance
- ◆ Dérivation et héritage
- ◆ Modèle

- Un concept qui se rapproche de la **programmation par objets** est celui de "**boîte noire**" en **électronique** : en essaie de **masquer** le plus possible le contenu et le **fonctionnement interne** et l'on dispose simplement de **fonctions d'entrée et de sortie**.



- ◆ d'augmenter la quantité de code qu'est capable de maîtriser un développeur,
- ◆ de faciliter la mise au point des applications,
- ◆ de renforcer la réutilisabilité de portions importantes de code ;

- ➔ Le principe de base est celui de **classes** qui offrent la possibilité **d'encapsuler** à la fois des **données** et du **code** et à partir desquelles on peut créer des **instances** ou **objets**.

9

### – La programmation « classique »:

- ➔ offre quelques possibilités **d'Encapsulation** comme par exemple en **C** les **variables locales** dans un sous-programme ou les **variables statiques** dans une unité de compilation.

### – Dans la programmation « objet »:

- ➔ Le code **encapsulé** dans une **classe** est réparti en différents traitements ou **méthodes**.
- ➔ Chaque **méthode** correspond normalement à une fonction bien précise et bien délimitée. Ces traitements sont répartis en **méthodes internes** et **méthodes externes**.
- ➔ Les **méthodes internes** ne sont utilisables qu'à partir d'autres méthodes de la classe,
- ➔ tandis que les **méthodes externes** peuvent être sollicitées depuis l'extérieur de la classe.

10

- Pour faire "réagir" un objet ou communiquer avec lui, on lui envoie un stimulus sous forme d'un **message**, éventuellement accompagné de paramètres. A la réception d'un message donné, l'objet déclenche une méthode "externe" appropriée.
- L'ensemble des méthodes externes d'une classe constitue l'**interface de la classe**; leur fonction est de traiter les messages susceptibles d'être reçus par les objets issus de cette classe.

#### Principes de base :

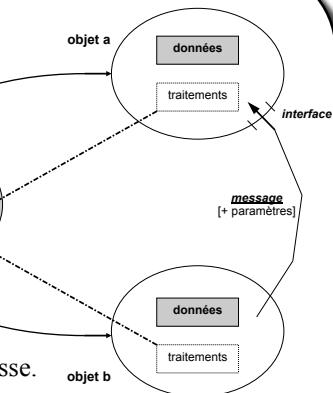
**Classe** = données + méthodes

**Objet** = instance d'une classe

**Message** = stimulus adressé à un objet (provoque le déclenchement d'une méthode)

**Interface** = ensemble des méthodes "externes"

**Encapsulation** = données + méthodes "internes"



11

## Héritage

**Principe de la dérivation** : L'un des aspects fondamentaux de la Programmation Orientée Objet (POO) est la notion de **dérivation de classes**. Partant d'une classe ou de plusieurs classes générales, le principe est de construire de nouvelles classes, comprenant toutes les propriétés de ces classes générales (ou classes de bases) afin de les enrichir de nouvelles propriétés.

→ On définit tout d'abord une classe qui possède les propriétés générales de toutes les formes:

```
Class forme
{
    point centre;
    ...
public:
    point position() { return center; }
    void deplacer(point vers) {center = vers; draw(); }
    virtual void dessiner();
    virtual void tourner(int);
    ...
};
```

→ Les fonctions dont l'implantation est spécifique pour chaque forme sont marquées par le mot clé virtual.

**Plus on dérive**

→ { plus on enrichit }  
→ { plus on spécialise }

→ Pour définir les fonctions virtuelles, on commencera par définir une classe dérivée de la classe forme:

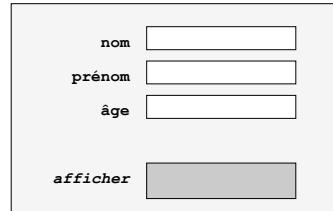
```
Class cercle : public forme
{
    int rayon;
    public :
        void dessiner() { ... }
        void tourner(int) {};
};
```

12

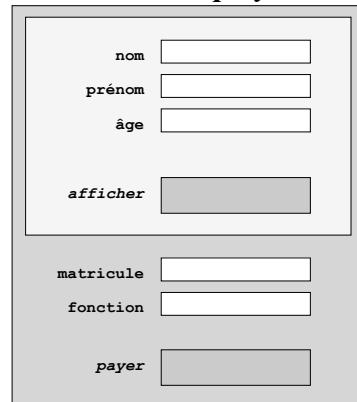
## Exemple Héritage

→ Lors d'une dérivation, la classe dérivée hérite des propriétés de la classe de base.

**classe ident**



**classe employe**



classe de base

**classe ident**

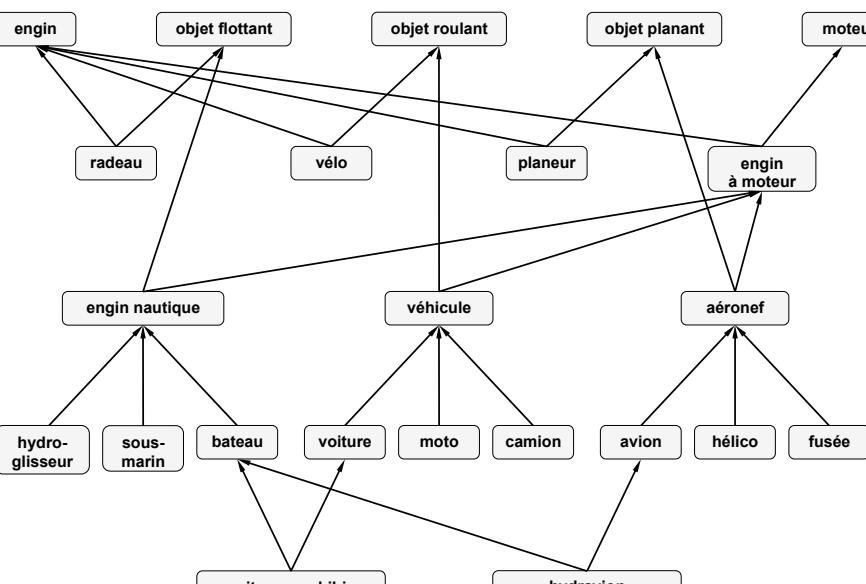
classe dérivée

**classe employe**

la classe **employe**  
dérive  
de la classe **ident**

13

## Héritage Multiple



14

## Polymorphisme

- ♦ **Polymorphisme de méthodes:** Faculté d'une méthode de s'adapter à des objets issus de classes différentes (mais en dérivation).

Exemple: La méthode "démarrer un moteur" de la classe "**moteur**" peut s'adapter aux objets des classes dérivées "**bateau**", "**voiture**" et "**avion**"...

- ♦ **Polymorphisme de classes:** Faculté d'une classe de présenter différentes facettes lors de dérivation

➔ Le polymorphisme de classes repose sur le polymorphisme des traitements.

15

## Abstraction de données

La programmation par abstraction de données consiste à fournir des facilités pour définir des opérations pour un type défini par l'utilisateur.

- ♦ **Initialisation et destruction:**

Contrairement aux types prédéfinis, aucun système ne peut définir, par défaut, les fonctions **d'initialisation** et de **destruction** des types définies par l'utilisateur. Il lui appartient donc de fournir ces fonctions.

► Considérons un type utilisateur (que l'on appellera tableau) qui est constitué de sa taille et d'un pointeur vers les éléments du tableau. La création d'un objet de type tableau se fait en définissant une variable de la classe tableau.

➔ Que doit-on créer à l'initialisation?

➔ Faut-il allouer la place nécessaire pour coder le tableau? etc ...

❖ Les réponses ne peuvent être fournies que par l'utilisateur ayant défini le type tableau.

➔ Une première solution consiste à se **définir une fonction «init»** que l'utilisateur se forcera à appeler avant toute utilisation d'un objet d'un type utilisateur.

16

```

Class tableau
{
    int taille;
    int* t;
public :
    void init(int taille);
    ...
};

void une_fonction()
{
    tableau t;
    ...
    t.init(0); // N'utiliser t
    // Qu'après son
    // Initialisation
}

```

Cette solution est peu agréable.

♦ C++ fournit un mécanisme plus astucieux pour faire l'initialisation:

➔ l'appel de la fonction d'initialisation se fait automatiquement à la définition de la variable. La fonction d'initialisation se nomme constructeur.

♦ De même, la destruction d'un objet d'un type défini par l'utilisateur se fera automatiquement:

➔ A condition que l'utilisateur définissent une fonction destruction adaptée au type qu'il a défini. La fonction de destruction se nomme destructeur.

```

Class tableau
{
    int taille;
    int* t;
public:
    tableau(int); //Constructeur
    ~tableau(); //Destructeur
};

tableau::tableau(int n) //constructeur
{
    if (s<=0) erreur();
    taille = n;
    t = New int[n];
}
tableau::~tableau() //Destructeur
{
    delete[] t;
}

```

17

## Méthodes de conception orientée objets

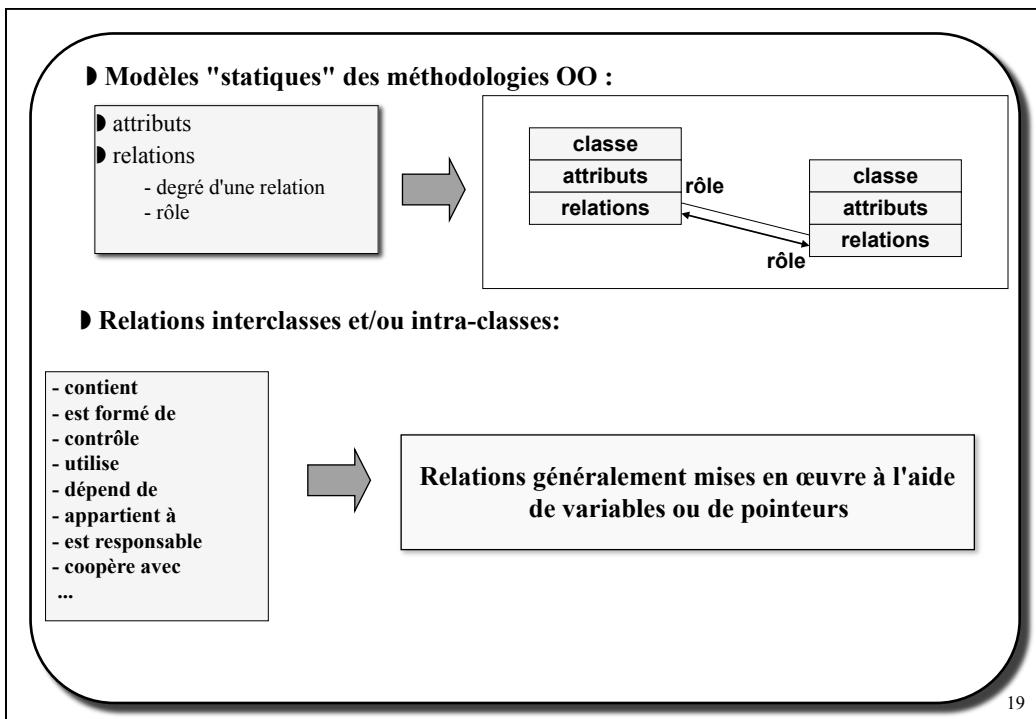
❖ Méthodes de conception :

- Ensemble de modèles plus ou moins formels, de notations et d'outils :
- ◆ **OMT (Object Modeling Technique) :**
  - Modèle statique (spécification des hiérarchies de classes)
  - Modèle dynamique (évolutions des objets)
  - Modèle fonctionnel (interactions entre les objets)

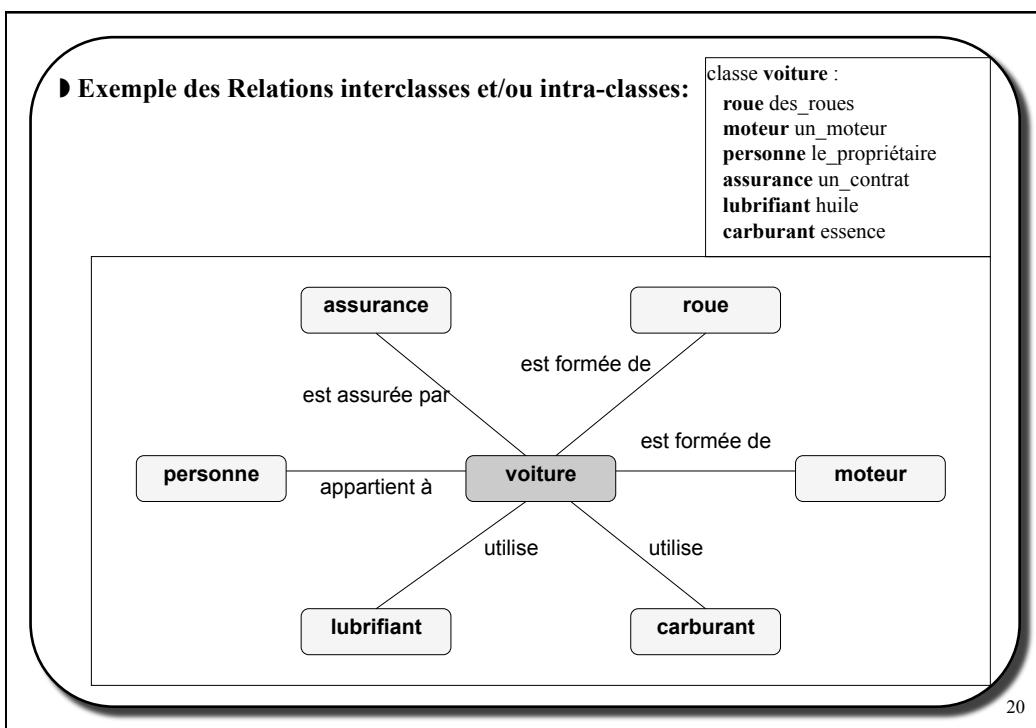
---

<p>► <b>OOM (Object Oriented Modeling):</b></p> <ul style="list-style-type: none"> <li>► modèle conceptuel</li> <li>► modèle logique</li> <li>► modèle physique</li> </ul>	<p>► <b>UML (Unified object Modeling Language) :</b></p> <ul style="list-style-type: none"> <li>► modèle conceptuel</li> <li>► modèle logique</li> <li>► modèle physique</li> </ul>
--	---

18



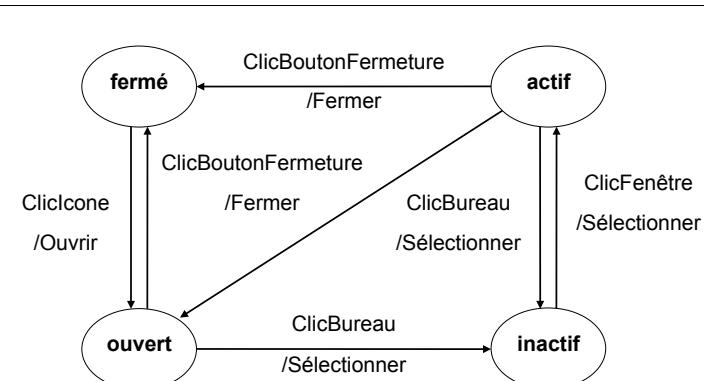
19



20

► Modèles "dynamiques" des méthodologies OO :

- ◆ évolution de l'état des objets (automates, réseau de Pétri...)
- ◆ états, transitions, événements, ...



21

► C++ et les concepts objets:

► Encapsulation:

- Classes
- Attributs de contrôle des accès
- Relation d'amitié

► Type de données :

- Classes
- Conversion entre classes (implicites ou explicites)

► Héritage :

- Héritage simple
- Héritage multiple

► Polymorphisme et généricité :

- Surcharge des méthodes, des fonctions et des opérateurs
- Fonctions virtuelles, classes abstraites
- Modèle de classes et modèles de fonctions

22

## Environnement de développement

- **Composants de base :** ➔ Trois composants de base :

- **Compilateur du langage**

- **Éditeur de projets**

- Notion de projet
    - Notion d'environnement "cible »

- **Outil de mise au point**

- Visualisation des classes à l'aide d'un “ **browser** ” de classes,
    - Possibilité d'examiner dynamiquement le fonctionnement,
    - Suivi de l'exécution.

- **Librairies complémentaires:**

- **Librairies d'interface utilisateur**

- exemple : *Microsoft Foundation Class (MFC)*

- **Librairies complémentaires**

- exemple : *Standard Template Library (STL)*

- **Outils de construction d'interface utilisateur:**

- Outils de construction visuelle

- Nécessité d'un environnement fenêtré

23

## Du C au C++

- **Les commentaires**
- **Entrées/sorties**
- **Les manipulateurs**
- **Les conversions explicites**
- **Les variables**
- **Les constantes**
- **Les types composés**
- **Allocation mémoire**

24

## □ Les commentaires

- Le langage C++ offre un nouvelle façon d'ajouter des **commentaires**.
- En plus des symboles « /\* et \*/ » utilisés en C, le langage C++ offre les symboles « // » qui permettent d'ignorer tout jusqu'à la fin de la ligne.

**Exemple :**   /\* commentaire traditionnel sur plusieurs lignes valide en C et C++ \*/

```
void main() { // commentaire de fin de ligne valide en C++
#ifndef 0
    // une partie d'un programme en C ou C++ peut toujours
    // être ignorée par les directives au préprocesseur
    // #if .... #endif
#endif
}
```

- Il est préférable d'utiliser les symboles // pour la plupart des commentaires.
- N'utiliser les commentaires C ( /\* \*/ ) que pour isoler des blocs importants d'instructions.

25

## □ Les entrées/sorties

- Les entrées/sorties en langage C s'effectue généralement par les fonctions **scanf** et **printf** de la librairie standard du langage C « **stdio.h** ». Il est possible d'utiliser ces fonctions pour effectuer les entrées/sorties de vos programmes.
- Cependant les programmeurs C++ préfèrent les entrées/sorties par flux (ou **flot** ou **stream** ).
- Quatre flots sont prédéfinis lorsque vous avez inclus le fichier d'en-tête « **iostream.h** » :
  - **cout** qui correspond à la sortie standard
  - **cin** qui correspond à l'entrée standard
  - **cerr** qui correspond à la sortie standard d'erreur non tamponné
  - **clog** qui correspond à la sortie standard d'erreur tamponné.
- L'opérateur (<<) permet d'envoyer des valeurs dans un flot de sortie,
- L'opérateur (>>) permet d'extraire des valeurs d'un flot d'entrée.
- Tout comme pour la fonction scanf, les espaces sont considérés comme des séparateurs entre les données par le flux **cin**.
- Notez l'absence de l'opérateur & dans la syntaxe du **cin**. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

26

## ► Exemple des entrées/sorties

```
#include <iostream.h>
void main() {
    int i=123;
    float f=1234.567;
    char ch[80]={"Bonjour\n", rep;
    cout << "i=" << i << "  f=" << f << "  ch=" << ch;
    cout << "i = ? ";
    cin >> i;           // lecture d'un entier
    cout << "f = ? ";
    cin >> f;           // lecture d'un réel
    cout << "rep = ? ";
    cin >> rep;          // lecture d'un caractère
    cout << "ch = ? ";
    cin >> ch;           // lecture du premier mot d'une chaîne
    cout << "ch= " << ch; // c'est bien le premier mot ...
}
```

/\*-- résultat de l'exécution--  
i=123 f=1234.57 ch=Bonjour  
i = ? 12  
f = ? 34.5  
rep = ? y  
ch = ? c++ is easy  
ch= c++ --\*/

27

## ♦ Pourquoi CIN, COUT, CERR CLOG ?

### ► Vitesse d'exécution plus rapide :

- ▶ La fonction `printf` doit analyser à l'exécution la chaîne de formatage,
- ▶ Tandis qu'avec les flots, la traduction est faite à la compilation.

### ► Vérification de type : pas d'affichage erroné:

```
#include <stdio.h>
#include <iostream.h>
void main() {
    int i=1234;
    double d=567.89;
    printf("i= %d  d= %lf !!!!!\n", i, d); //^erreur: %lf normalement
    cout << "i= " << i << "  d= " << d << "\n";// OK,
}
```

/\* Résultat de l'exécution \*\*

i= 1234 d= -5243 !!!!!!  
i= 1234 d= 567.89

### ► Taille mémoire réduite :

- ▶ Seul le code nécessaire est mis par le compilateur,
- ▶ Alors que pour «`printf`» tout le code correspondant à toutes les possibilités d'affichage est mis.
- ▶ On peut utiliser les flux avec les types utilisateurs (surcharge possible des opérateurs `>>` et `<<`).

28

## □ LES MANIPULATEURS

- Sont des éléments qui modifient la façon dont les éléments sont lus ou écrits dans le flot.
- Les principaux manipulateurs sont :

<b>dec</b>	lecture/écriture d'un entier en décimal
<b>oct</b>	lecture/écriture d'un entier en octal
<b>hex</b>	lecture/écriture d'un entier en hexadécimal
<b>endl</b>	insère un saut de ligne et vide les tampons
<b>setw(int n)</b>	affichage de n caractères
<b>setprecision(int n)</b>	affichage de la valeur avec n chiffres (avec arrondis)
<b>setfill(char)</b>	définit le caractère de remplissage
<b>flush</b>	vide les tampons après écriture

```
#include <iostream.h>
#include <iomanip.h>
void main() {
    int i = 1234;
    float p = 12.3456;
    cout << "|\n" << setw(8) << setfill('*')
        << hex << i << "|\\n" << "|"
        << setw(6) << setprecision(4)
        << p << "|\n" << endl;
}
```

/\*-- résultat de l'exécution --\*/
|\*\*\*\*4d2|
|\*12.35|

29

## □ Les CONVERSIONS EXPLICITES

- En C++, comme en langage C, il est possible de faire des conversions explicites de type, bien que le langage soit plus fortement typé :

```
double d;
int i;
i = (int) d;
```

- Le C++ offre aussi une **notation fonctionnelle** pour faire une **conversion explicite de type** :

```
double d;
int i;
i = int(d);
```

- Cette façon de faire ne marche que pour les types simples et les types utilisateurs.

- Pour les **types pointeurs ou tableaux** le problème peut être résolu en définissant un nouveau type :

```
double d;
int *i;
typedef int *ptr_int;
i = ptr_int(&d);
```

- ⇒ La conversion explicite de type est utile lorsqu'on travaille avec des pointeurs du type void \*

30

## □ DEFINITION DE VARIABLES

- En C++ vous pouvez déclarer les variables ou fonctions n'importe où dans le code.
- La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.
- Ceci permet :
  - De définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité. C'est particulièrement utile pour des grosses fonctions ayant beaucoup de variables locales.
  - D'initialiser un objet avec une valeur obtenue par calcul ou saisie.
- Exemple :

```
#include <stdio.h>

void main() {
    int i=0;      // définition d'une variable
    i++;         // instruction
    int j=i;      // définition d'une autre variable
    j++;         // instruction
    int somme(int n1, int n2); // déclaration d'une fonction
    printf("%d+%d=%d\n", i, j, somme(i, j)); // instruction

    cin >> i;
    const int k = i; // définition d'une constante initialisée
                      // avec la valeur saisie
}
```

31

## □ VARIABLE DE BOUCLE

- On peut déclarer une variable de boucle directement dans l'instruction for.
- Ceci permet de n'utiliser cette variable que dans le bloc de la boucle.
- Exemple :

```
#include <iostream.h>

void main() {
    for(int i=0; i<10; i++)
        cout << i << ' ';
    // i n'est pas utilisable à l'extérieur du bloc for
}

/*-- résultat de l'exécution -----
0 1 2 3 4 5 6 7 8 9
-----*/
```

32

## □ VISIBILITE DES VARIABLES

- L'opérateur de résolution de portée :: permet d'accéder aux variables globales plutôt qu'aux variables locales.

```
#include <iostream.h>

int i = 11;

void main() {
    int i = 34;
    { int i = 23;

        ::i = ::i + 1;
        cout << ::i << " " << i << endl;
    }
    cout << ::i << " " << i << endl;
}
```

/\*-- résultat de l'exécution--\*/  
12 23  
12 34

- L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (lisibilité). Il est préférable de donner des noms différents plutôt que de réutiliser les mêmes noms.
- En fait, on utilise beaucoup cet opérateur pour définir hors d'une classe les fonctions membres ou pour accéder à un identificateur dans un espace de noms.

33

## □ VARIABLES REFERENCES

- En plus des variables normales et des pointeurs, le C++ offre les variables références.
- Une variable référence permet de créer une variable qui est un « synonyme » d'une autre.
- Dès lors, une modification de l'une affectera le contenu de l'autre.
- Exemple:

```
int i;
int & ir = i;           // ir est une référence à i
int *ptr;

i=1;
cout << "i= " << i << " ir= " << ir << endl; // affichage de : i= 1 ir= 1
ir=2;
cout << "i= " << i << " ir= " << ir << endl; //affichage de : i= 2 ir= 2
ptr = &ir;
*ptr = 3;
cout << "i= " << i << " ir= " << ir << endl; // affichage de : i= 3 ir= 3
```

- Une variable référence doit obligatoirement être initialisée et le type de l'objet initial doit être le même que l'objet référence.
- Intérêt : passage des paramètres par référence et utilisation d'une fonction en lvalue (voir cours suivant)

34

## ■ LES CONSTANTES

- Les habitués du C ont l'habitude d'utiliser la directive du préprocesseur #define pour définir des constantes. Il est reconnu que l'utilisation du préprocesseur est une source d'erreurs difficiles à détecter.
- En C++, l'utilisation du préprocesseur se limite aux cas les plus sûrs :
  - Inclusion de fichiers
  - Compilation conditionnelle.
- Le mot réservé const permet de définir une constante.
  - L'objet ainsi spécifié ne pourra pas être modifié durant toute sa durée de vie.
  - Il est indispensable d'initialiser la constante au moment de sa définition.

► Exemple :

```
const int N = 10;           // N est un entier constant.
const int MOIS=12, AN=1995; // 2 constantes entières
int tab[2 * N];           // autorisé en C++ (interdit en C)
```

35

## ■ LES CONSTANTES et POINTEURS

- Il faut distinguer ce qui est pointé du pointeur lui-même.

► La donnée pointée est constante :

```
const char *ptr1 = "QWERTY";
ptr1++;
// autorisé
*ptr1 = 'A'; // ERROR: assignment to const type
```

► Le pointeur est constant :

```
char * const ptr2 = "QWERTY";
ptr2++;
// ERROR: increment of const type
*ptr2 = 'A'; // autorisé
```

► Le pointeur et la donnée sont constants :

```
const char * const ptr3 = "QWERTY";
ptr3++;
// ERROR: increment of const type
*ptr3 = 'A'; // ERROR: assignment to const type
```

36

## ■ LES TYPES COMPOSÉS

- En C++, comme en langage C, le programmeur peut définir des nouveaux types en définissant des **struct**, **enum** ou **union**.
- Mais contrairement au langage C, l'utilisation de **typedef** n'est plus obligatoire pour renommer un type.
- Exemple:

```
struct FICHE { // définition du type FICHE
    char *nom, *prenom;
    int age;
};

FICHE adherent, *liste;
```

// en C, il faut ajouter la ligne :  
// typedef struct FICHE FICHE;

```
enum BOOLEEN { FAUX, VRAI }; // définition du type BOOLEEN
BOOLEEN trouve;
trouve = FAUX;
trouve = 0; // ERREUR en C++ :
            // vérification stricte des types
trouve = (BOOLEEN) 0; // OK
```

// en C, il faut ajouter la ligne  
// typedef enum BOOLEEN BOOLEEN;

37

## ■ ALLOCATION MÉMOIRE

- Le C++ met à la disposition du programmeur deux opérateurs **new** et **delete** pour remplacer respectivement les fonctions **malloc** et **free** (bien qu'il soit toujours possible de les utiliser).

### ► L'opérateur new

- L'opérateur **new** réserve et initialise l'espace mémoire qu'on lui demande. Il retourne l'adresse de début de la zone mémoire allouée.

```
int *ptr1, *ptr2, *ptr3;
ptr1 = new int;           // allocation dynamique d'un entier
ptr2 = new int [10];      // allocation d'un tableau de 10 entiers
ptr3 = new int(10);       // allocation d'un entier avec Initialisation

struct date {int jour, mois, an; };
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};
ptr4 = new date;          // allocation dynamique d'une structure
ptr5 = new date[10];      // allocation dynamique d'un tableau de structure
ptr6 = new date(d); // allocation dynamique d'une structure avec initialisation
```

- En cas d'erreur d'allocation par **new**, une exception **bad\_alloc** est lancée s'il n'y a pas de fonction d'interception définie par l'utilisateur.

38

- L'allocation des tableaux à plusieurs dimensions est possible :
 

```
typedef char TAB[80]; // TAB est un synonyme de : tableau de 80 caractères
TAB *ecran;
ecran = new TAB[25]; // ecran est un tableau de 25 fois 80 caractères
ecran[24][79] = '$';
ou
char (*ecran)[80] = new char[25][80];
ecran[24][79] = '$';
```
- Attention à ne pas confondre :
 

```
int *ptr = new int[10]; // création d'un tableau de 10 entiers
avec
int *ptr = new int(10); // création d'un entier initialisé à 10
```

#### ► L'opérateur delete

- L'opérateur **delete** libère l'espace mémoire alloué par **new** à un seul objet.
- tandis que l'opérateur **delete[]** libère l'espace mémoire alloué à un tableau d'objets.
 

```
delete ptr1; // libération d'un entier
delete[] ptr2; // libération d'un tableau d'entier
```

- L'application de l'opérateur **delete** à un **pointeur nul** est légale et n'entraîne aucune conséquence fâcheuse (l'opération est tout simplement ignorée).

39

#### ► La fonction d'interception « **set\_new\_handler** »

- Si une allocation mémoire par **new échoue**, une fonction d'erreur utilisateur peut être appelée.
- La fonction **set\_new\_handler**, déclarée dans **new.h**, permet d'installer votre propre fonction d'erreur.

```
#include <iostream.h>
#include <stdlib.h> // exit()
#include <new.h> // set_new_handler()
//fonction d'erreur d'allocation mémoire dynamique
void erreur_mémoire(void) {
    cerr << "\n La mémoire disponible est insuffisante !!!" << endl;
    exit(1);
}
void main() {
    set_new_handler( erreur_mémoire );
    double *tab = new double [1000000000];
}
```

- Si la fonction d'interception **erreur\_mémoire** ne comporte pas un **exit**, une nouvelle demande d'allocation mémoire est faite. Et cela jusqu'à ce que l'allocation réussisse.

```
void erreur_mémoire(void) {
    static int n = 0;
    cerr << ++n << " fois, mémoire insuffisante" << endl;
    if ( n==10 ) exit(1);
}
```

40

## LES FONCTIONS

- Déclaration des fonctions
- Passage par référence
- Valeur par défaut des paramètres
- Fonction inline
- Surcharge de fonctions
- Retour d'une référence
- Utilisation des fonctions écrites en C
- Fichier d'en-têtes pour C et C++

41

### □ DECLARATION DES FONCTIONS

- Le langage C++ impose au programmeur de déclarer le nombre et le type des arguments de la fonction.
  - Ces déclarations sont identiques aux prototypes de fonctions de la norme C-ANSI.
  - Cette déclaration est par ailleurs obligatoire avant utilisation (contrairement à la norme C-ANSI).
- La déclaration suivante `int f1();` où `f1` est déclarée avec une liste d'arguments vide est interprétée en C++ comme la déclaration `int f1(void);`
  - La norme C-ANSI considère que `f1` est une fonction qui peut recevoir un nombre quelconque d'arguments, eux mêmes de type quelconques, comme si elle était déclarée `int f1( ... );`.
- Une fonction dont le type de la valeur de retour n'est pas void, doit obligatoirement retourner une valeur.

42

## □ PASSAGE PAR REFERENCE

- En plus du **passage par valeur**, le C++ définit le **passage par référence**.
- Lorsque l'on **passe** à une **fonction** un **paramètre par référence**, cette **fonction** reçoit un "**synonyme**" du **paramètre réel**.
- Toute **modification** du **paramètre référence** est **répercutee** sur le **paramètre réel**.
- Exemple :

```
void echange(int &n1, int &n2) { // n1 est un alias du paramètre réel i
    // n2 est un alias du paramètre réel j
    int temp = n1;
    n1 = n2; // toute modification de n1 est répercutee sur i
    n2 = temp; // toute modification de n2 est répercutee sur j
}

void main() {
    int i=2, j=3;
    echange(i, j);
    cout << "i= " << i << " j= " << j << endl; // affichage de:i=3 j=2
}
```

- Comme vous le remarquez, l'appel se fait de manière très simple.
  - Utilisez les **références** quand **vous pouvez**,
  - utiliser les **pointeurs** quand **vous devez**.
- Cette facilité **augmente la puissance** du langage mais doit être utilisée avec **précaution**, car elle **ne protège** plus la **valeur du paramètre réel** transmis **par référence**.

43

## □ VALEUR PAR DEFAUT DES PARAMETRES

- Certains **arguments d'une fonction** peuvent prendre **souvent la même valeur**.
- Le C++ permet de **déclarer** des **valeurs par défaut** dans le **prototype de la fonction**, pour ne pas avoir à spécifier ces valeurs à chaque appel.

- Exemple :

```
void print(long valeur, int base = 10);

void main() {
    print(16); // affiche 16 (16 en base 10)
    print(16, 2); // affiche 10000 (16 en base 2)
}

void print(long valeur, int base){
    cout << ltostr(valeur, base) << endl;
}
```

- Les **paramètres par défaut** sont **obligatoirement** les **derniers de la liste**.
- Ils **ne sont déclarés** que dans le **prototype de la fonction** et pas dans sa définition.

44

## □ FONCTION inline

- Le mot clé **inline** remplace avantageusement l'utilisation de **#define** du préprocesseur pour définir des pseudo-fonctions.
- Afin de rendre l'exécution plus rapide d'une fonction et à condition que celle ci soit de courte taille, on peut définir une fonction avec le mot réservé **inline**.
- Le compilateur générera, à chaque appel de la fonction, le code de celle ci.
- Les fonctions inline se comportent comme des fonctions normales et donc, présentent l'avantage de vérifier les types de leurs arguments, ce que ne fait pas la directive **#define**.
- Exemple :

```
#include <iostream.h>
inline int carre(int n);           // déclaration
void main() {
    cout << carre(10) << endl;
}

inline int carre(int n) {           // inline facultatif à la définition, mais préférable
    return n * n;
}
```

- Contrairement à une fonction normale, la portée d'une fonction **inline** est réduite au module dans lequel elle est déclarée.

45

## □ Surcharge de fonctions

- Une fonction se définit par :
  - son nom,
  - sa liste typée de paramètres formels,
  - le type de la valeur qu'elle retourne.
- Mais seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la signature de la fonction.
- On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents :
- Exemple:

```
void main() {
    cout << "1 + 2 = " << somme(1, 2) << endl;
    cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
    cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;
}

int somme( int n1, int n2){
    return n1 + n2;
}

int somme( int n1, int n2, int n3){
    return n1 + n2 + n3;
}

double somme( double n1, double n2) {
    return n1 + n2;
}
```

➔ Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction.

46

### Exemples:

```
enum Jour {DIMANCHE, LUNDI, MARDI, MERCRIDI, JEUDI, VENDREDI, SAMEDI};  
enum Couleur {NOIR, BLEU, VERT, CYAN, ROUGE, MAGENTA, BRUN, GRIS};  
  
void f1(Jour j);  
void f1(Couleur c);  
// OK : les types énumérations sont tous différents  
  
void f2(char *str) { /* ... */ }  
void f2(char ligne[80]) { /* ... */ }  
// Erreur: redéfinition de la fonction f  
// char * et char [80] sont considérés de même type  
  
int sommel(int n1, int n2) {return n1 + n2;}  
int sommel(const int n1, const int n2) {return n1 + n2;}  
// Erreur: la liste de paramètres dans les déclarations  
// des deux fonctions n'est pas assez divergente  
// pour les différencier.  
  
int somme2(int n1, int n2) {return n1 + n2;}  
int somme2(int & n1, int & n2) {return n1 + n2;}  
// Erreur: la liste de paramètres dans les déclarations  
// des deux fonctions n'est pas assez divergente  
// pour les différencier.
```

47

## ■ RETOUR D'UNE REFERENCE

### ► Fonction retournant une référence

- Une fonction peut retourner une valeur par référence et on peut donc agir, à l'extérieur de cette fonction, sur cette valeur de retour.
- La syntaxe qui en découle est plutôt inhabituelle et déroutante au début.

```
#include <iostream.h>
```

```
int t[20]; // variable globale -> beurk !  
int & nIeme(int i) {  
    return t[i];  
}  
void main() {  
    nIeme(0) = 123;  
    nIeme(1) = 456;  
    cout << t[0] << " " << ++nIeme(1);  
}
```

// -- résultat de l'exécution -----  
// 123 457

- Tout se passe comme si **nIeme(0)** était remplacé par **t[0]**.
- Une fonction retournant une référence (non constante) peut être une lvalue.
- La variable dont la référence est retournée, doit avoir une durée de vie permanente.

48

## ■ Utilisation d'une fonction écrite en C

- Le compilateur C++ génère pour chaque fonction un nom dont l'éditeur de liens aura besoin. Le nom généré en C++ se fait à partir de la signature de la fonction.
- Ainsi, à une fonction surchargée 2 fois, correspondra à 2 fonctions de noms différents dans le module objet.
- En C, la signature de la fonction ne comporte que le nom de la fonction.
- Pour pouvoir utiliser dans un programme C++ des fonctions compilées par un compilateur C, il faut déclarer ces fonctions de la façon suivante :

```
extern "C" int f1(int i, char c);  
extern "C" void f2(char *str);
```

ou

```
extern "C" {  
    int f1(int i, char c);  
    void f2(char *str);  
}
```

- Par contre, les fonctions f1 et f2 ne pourront pas être surchargees

49

## ■ Fichier d'en-têtes pour C et C++

- Le symbole \_\_cplusplus est défini pour les compilateurs C++ uniquement.
- Il facilite l'écriture de code commun aux langages C et C++, et plus particulièrement pour les fichiers d'en-têtes :

```
#ifdef __cplusplus  
    extern "C" {  
#endif  
#include <stdio.h>  
void fail(char *msg);  
#ifdef __cplusplus  
    }  
#endif
```

- Cet exemple peut être compilé aussi bien en C qu'en C++.

50

## Classes et Objets

### ■ Notion de classe et d'instance

- Droits d'accès aux membres
- Trois types de classes
- Recommandation

### ■ Membres d'une classe

- Définition des fonctions membres
- Instanciation d'une classe & utilisation des objets
- Fonction membres constance et surcharge de méthodes par ces fonctions

### ■ Constructeurs et destructeur

- Constructeur par défaut et avec arguments & destructeurs
- Constructeur de copie
- Liste d'initialisation d'un constructeur

### ■ Pointeurs et auto-référence

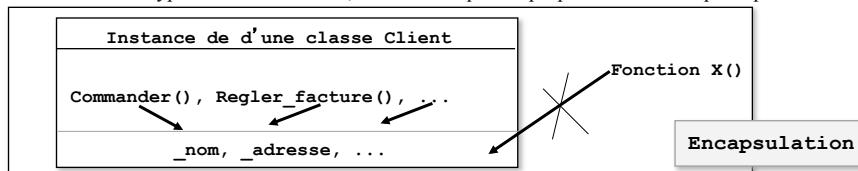
### ■ Complément sur les classes

- Membres statiques
- Fonctions et classes amies
- Fonctions et classes génériques
- Exemple complet

51

### Notion de classe et d'instance

- Un **modèle orienté objet** est constitué de **plusieurs objets** qui correspondent à des **entités de vie réelle**.
- Chaque **objet** maintient des **informations qui lui sont propres** (le client «Dupont» à un nom, une adresse, ...etc.) et possède un **comportement particulier** (le client «Martin» passe la commande N°130, règle la facture N°13302 ...).
- «Dupont» et «Martin» ont tous les deux des **attributs** et un **comportement commun**. Ils sont issus d'une même **classe** qui décrit leur **structure interne**.
- **Définition**
- Une **classe** est une **modélisation abstraite** d'une **collection d'objets du monde réel** ou d'un **problème type à résoudre**.
- Il **donc nécessaire** de **séparer** les **détails** liés à **l'implantation de la classe (section private)**, de ses **propriétés générales** qui doivent être **visibles de l'extérieur (interface de la classe -section public)**.
- représentation d'un type abstrait de donnée, c'est à dire qu'elle propose une section publique



52

## □ Notion de classe et d'instance

- Une classe décrit le modèle structurel d'un **objet** :
  - ▷ Un ensemble des **attributs** (ou champs ou données membres) décrivant sa structure
  - ▷ Un 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
```

53

## □ Droits d'accès aux membres

- 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 accessibles 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.

54

## □ Trois 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 {  
    void set_date(int, int, int); // méthodes publiques (par défaut)  
    void next_date();  
    // autres méthodes ....  
    private :  
        // données privées  
        int _jour, _mois, _an;  
};
```

► **union Classe2 { /\* ... \*/ };**

- Tous les membres sont par défaut d'accès public
- 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

55

## □ Recommandation

► 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**

► **Exemple d' une Classe Avion :**

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

majuscule

publics en premier

Méthodes en minuscules

caractère « \_ »

56

## □ DEFINITION DES FONCTIONS MEMBRES

- En général, la **déclaration d'une classe** contient les prototypes des **fonctions membres** de la classe.
- Les **fonctions membres** sont définies dans un module séparé ou plus loin dans le code source.
- Syntaxe de la définition hors de la classe d'une méthode :

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

- Dans la **fonction membre** on a un accès direct à tous les membres de la classe.
- La définition de la méthode peut aussi 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***.
  - **Rappel:** la visibilité d'une fonction ***inline*** est restreinte au module seul dans laquelle elle est définie.

57

### → Exemple de définition de méthode de la classe Avion :

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

void Avion::init(char m[], char *t, float p) {
    if ( strlen(m) != 5 ) {
        erreur("Immatriculation invalide");
        strcpy(_immatriculation, "?????");
    }
    else strcpy(_immatriculation, m);
    _type = new char [strlen(t)+1];
    strcpy(_type, t);
    _poids = p;
}

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

Déclaration d ' une classe

Définition des méthodes en externes

► Définition des méthodes « *inline* » à l' intérieur ou à l' extérieur de la déclaration de la classe

```
class Nombre {
public :
    void setnbre(int n) { _nbre = n; }
    int getnbre() { return _nbre; }
    void affiche();
private :
    int _nbre;
};

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

58

## □ 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
Avion *av3= new Avion; // un pointeur initialisé
Avion *av = new Avion [10] // création (dynamique) d'un tableau d'instances
```

## □ UTILISATION DES OBJETS

- Après avoir créé une **instance** (de façon **statique** ou **dynamique**) on peut accéder aux **attributs** et **méthodes** de la **classe**.
- L'accès se fait comme 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();
```

59

## □ 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**.
  - Il 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; }
    int getnbre() const { return nbre; }
    void affiche() const;
private :
    int _nbre;
};
inline void Nombre::affiche() const {
    cout << "Nombre = " << _nbre << endl;
}
```

Méthodes constante

seule les **fonctions const** peuvent être appelées pour un **objet constant**

Exemple  
const Nombre n1; // une constante  
n1.affiche(); // OK  
n1.setnbre(15); // ERREUR:  
Nombre n2;  
n2.affiche(); // OK  
n2.setnbre(15); // OK

- 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.

60

## □ 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 :  
    char & nieme(int n); // (1)  
    char nieme(int n) const; // (2)  
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;  
    // initialisation de ch1  
    const String ch2;  
    // initialisation de ch2  
    cout << ch1.nieme(1);  
    cout << ch2.nieme(1);  
    ch2.nieme(3) = 'u'; //ERREUR  
}
```

appel de la méthode (1)

appel de la méthode (2)

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;`

61

## □ CONSTRUCTEURS ET DESTRUCTEURS

- Les **données membres** d'une classe ne peuvent pas être initialisées:
  - Il faut donc prévoir une méthode d'initialisation.
  - 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 fâcheuses 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 le même **nom de la classe** et qui ne retourne rien (pas même un void).
- Exemple :

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

► Lors de la création de chaque instance de la classe **Nombre**, le **constructeur Nombre::Nombre()** sera automatiquement appelé

62

## □ CONSTRUCTEURS ET DESTRUCTEURS

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

- Exemple:

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

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

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

Nombre n1; //Appel du constructeur sans argument Nombre::Nombre();
Nombre n2(13); //Appel du constructeur avec argument Nombre::Nombre(int);
```

► Si le **concepteur** de la classe ne spécifie pas de constructeur, le compilateur générera un constructeur par défaut.

► le fait de préciser =0 indique une valeur par défaut (0), qui permet d'appeler le constructeur sans lui passer de paramètre.

► comme les autres fonctions, les constructeurs peuvent être surchargés.

► Il est alors possible d'utiliser 'l'un ou l'autre constructeur lors de la création, d'un objet.

63

## □ CONSTRUCTEURS ET DESTRUCTEURS

### ► Constructeur et tableaux de d'objets

- Si une classe possède un constructeur par défaut, alors chaque élément d'un tableau est initialisé automatiquement.
- Exemple: **Nombre n[10];** //Appel du constructeur **Nombre::Nombre();** pour les objets **n[0] n[1] ... n[9].**

### ► Constructeur et objets dynamiques

- Lorsque un seul objet est créé à la fois, n'importe quel constructeur est utilisable:
- Exemple: **Nombre \*pn1=new Nombre;** //Appel du constructeur **Nombre::Nombre();**
- Exemple: **Nombre \*pn1=new Nombre(3);** //Appel du constructeur **Nombre::Nombre(int)**
- Exemple: **Nombre \*pn=new Nombre[10];** //Appel **Nombre::Nombre()** pour les 10 objets **n[0] n[1] ... n[9].**

### ► Exemple:

```
Nombre n1;           // correct, appel du constructeur par défaut
Nombre n2(10);       // correct, appel du constructeur à 1 paramètre
Nombre n3 = Nombre(10); // idem que n2
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 défaut
Nombre tab2[3] = { Nombre(10), Nombre(20), Nombre(30) };
// initialisation des 3 objets du tableau par les nombres 10, 20 et 30
```

64

## □ CONSTRUCTEURS ET DESTRUCTEURS

### ► Destructeur

- Le **destructeur** permet de libérer l'espace réservé lors de l'instantiation d'un objet (allocation dynamique dans le constructeur).
- 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 le même **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 () {  
    // ...  
}
```

Déclaration du **Destructeur**.

Définition du **Destructeur**.

- Si des constructeurs sont définis avec des paramètres, le compilateur ne générera pas le constructeur par défaut.
- Les **constructeurs et destructeurs** sont les seules méthodes non constantes qui peuvent être appelées pour des objets constants.

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

65

## □ CONSTRUCTEURS COPIE

- Le **constructeur de copie** est utilisé dans plusieurs circonstances :
  - **Initialisation d'un objet** à l'aide d'une autre objet de la même classe
  - **passage d'un objet comme argument** vers un objet comme **paramètre**, lors d'un appel de fonction
  - **retour d'un objet** d'une classe comme valeur renournée par une fonction.
- Il ne peut y avoir qu'un **seul constructeur de copie** par classe.
- Il s'agit obligatoirement d'une fonction membre qui peut être d'accès public, protégé ou privé.
- Le **constructeur de copie** est fonction membre dont le **nom** est obligatoirement celui **de la classe**:
  - Pas de valeur renournée et pas d'indication de type rentré
  - Argument unique, obligatoirement **sous forme d'une référence** de la classe (constante ou non).

### ► Exemple:

```
class chaine {  
    public:  
        chaine(const char * chainconst)  
        chaine(int longueur);  
        chaine(const chaine &ch);  
        ~chaine();  
        //...  
    Private:  
        int longueur;  
        char *_val;  
};  
chaine::chaine(const chaine &ch) {  
    _longueur=ch._longueur;  
    _val=new char[_longueur+1];  
    strcpy(_val, ch._val); }
```

Définition de deux constructeurs avec argument

Déclaration du **constructeur de copie**

Définition du **constructeur de copie**

```
chaine ch1 (" bonjour");  
chaine ch2(3);  
chaine ch3(ch1);  
chaine ch4 =ch1;
```

Appel du constructeur 1

Appel du constructeur 2

Appel du constructeur de copie

66

## □ CONSTRUCTEURS COPIE

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

```
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**
- Exemple:

```
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
}
```

67

## □ 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

```
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:
    _z = b; // OK : affectation
}
```

l'affectation à une constante est interdite

Et comment initialiser l'objet membre \_y ???

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

Réponse: la liste d'initialisation.

68

## □ Liste d'initialisation d'un constructeur

- 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 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**.

69

## □ LE POINTEUR : Auto-référence d'un objet : *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, permettant ainsi à la méthode d'accéder aux membres de l'objet.
  - 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 :

```
class 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 :

```
class X {
public:
    X &f1() { cout << "X "; return *this; }
private:
};
```

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

```
void main() {
    X x;
    x.f1().f1().f1();
```

// affiche : X X X

► La valeur *this* ne peut pas être changée.  
► *this* ne peut pas être explicitement déclaré

70

## □ Quand utiliser le pointeur *this*?

Dans certains contextes, il est nécessaire d'utiliser le **pointeur *this***:

- Pour vérifier que l'**objet** qui appelle la méthode existe bien, et éviter une faute de protection générale du système si l'**objet n'a pas été alloué**:
  - `if (this == void) return;`

- Pour **retourner un pointeur** sur l'**objet** qui a appelé la méthode:
  - `return this;`

- Pour **retourner une référence** sur l'**objet** qui a appelé la méthode:
  - `return *this;`

► Cela permet par exemple d'enchaîner des opérations sur même objet (exemple précédent)

► Cela permet par exemple d'enchaîner des assignations (dans le cas de surcharge d'opérateur)

- Tout **objet créé** comporte toujours une **donnée d'accès privé** nommée ***this***. Il s'agit d'un **pointeur** que l'on **ne peut pas modifier** et qui contient **automatiquement l'adresse de cet objet**.

71

## □ 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
```

72

## □ LES MEMBRES STATIQUES

► 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() { cout << nb << endl; }
    private:
        static int nb;
};

Ex1::nb = 0;

void main() {
    Ex1.affiche();
    Ex1 a, b, c;
    Ex1.affiche();
    a.affiche();
    EX1::affiche();
}
```

73

## □ Fonctions et classes 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.
- Une fonction amie (friend) est une fonction indépendante :
  - Qui n'est pas une méthode de la classe;
  - Et qui n'est pas appliquée à l'instance courante de la classe .
  - Elle agit sur des objets de la classe et peut manipuler des données membres: !!!!
  - C'est une infraction aux règles d'encapsulation pour des raisons d'efficacité.
- Une classe amie est une classe dont toutes les fonctions peuvent être amies d'une autre classe.
- Exemple de fonction amie :

```
class Nombre {
    friend int manipule_nbre ();
    public :
        int getnbre();
        // ...
    private :
        int _nbre;
    };
    int manipule_nbre(Nombre n) {return n._nbre + 1;}
```

74

► Exemple de classe amie :

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

class Screen {
    friend class Window;
    public:
        //...
    private :
        //...
};
```

Déclaration d'une classe amie

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

75

## □ Fonctions et classes génériques

- L'un des objectifs actuel des développeurs est de concevoir des «bibliothèques de composants génériques» utilisées dans des contexte très variés, et qui doivent donc être paramétrables.
- Le C++ grâce au mécanisme de «template» permet de décrire des fonctions et des classes génériques qui peuvent s'appliquer à des objets de classes différentes.
- L'idée est donc d'utiliser les mêmes lignes de code source quel que soit le type des paramètres passés aux fonctions.
- Lorsque l'algorithme est le même pour plusieurs types de données, il est possible de créer un modèle de fonction.
- C'est un modèle à partir duquel le compilateur générera les fonctions qui lui seront nécessaires.

► Exemple 1 :

```
template <class T>
void affiche(T *tab, unsigned int nbre) {
    for(int i = 0; i < nbre; i++) cout << tab[i] << " ";
    cout << endl;
}
void main() {
    int tabINT[7] = {5, 4, 2, 8, 16, 5, 19};
    affiche(tabINT, 7);
    double tabDOUBLE[2] = {11.1, 3.4};
    affiche(tabDOUBLE, 2);
    char *tabCHAR[] = {"OK", "cool", "Merci"};
    affiche(tabCHAR, 3);
}
```

76

## □ Fonctions et classes génériques

### ► Exemple 2 :

```
template <class T>      // déclaration du modèle
T min(T, T);
// ... la suite de votre programme
template <class T>      // définition du modèle
T min(T t1, T t2) {
    return t1 < t2 ? t1 : t2;
}
```

On peut aussi utiliser  
deux modèles différents

```
template <class X, class Y>
int valeur(X x, Y y) { // ... }
```

### ► Exemple 3 :

► Les fonctions génériques peuvent, tout comme les fonctions normales, être surchargées.

► Il est donc possible de spécialiser une fonction à une situation particulière, comme dans ce qui suit:

```
template <class T>
T min(T t1, T t2) {
    return t1 < t2 ? t1 : t2;
}
char *min(char *s1, char *s2) {
    return strcmp(s1, s2)<0 ? s1 : s2;
}
void main() {
    cout<<min(10, 20)<<" "<< min("Ingénieur", "Maitrise")<<endl;
}
```

Surchage de fonction générique

77

## □ Exemple de Pile d'entier

### ► Ce fichier pile.h contient les déclarations de la classe et les fonctions **inline**

```
class Pile
{
public:
    Pile(int max): _max(max){ _nb_elem=0; _etat=vide; _table=new int[max] }
    ~Pile(){ delete [] _table; }
    void empiler(int);
    int depiler();
    void afficher();
    int nb_elements();
private:
    const int _max;
    int * _table;
    int _nb_elem;
    enum pile_etat {ok, plein, vide} ;
    pile_etat _etat;
};
```

Constructeur de la classe pile

Liste d'initialisation

Destructeur

Déclaration d'un nouveau type

78

## □ Exemple de Pile d' entier

- Ce fichier **pile.cpp** contient l' implantation des méthodes publiques de la classe (il n'y a pas de méthode privées dans cet exemple).

```
void pile::empiler(int element) {
    if(_etat != plein) table[_nb_elem++] = element;
    else cout << "pile pleine" << endl;
    if(nb_elem >= _max) _etat = plein;
    else _etat = ok;
}
int pile::depiler()
{
    int element = 0 ;
    if(_etat != vide) element = _table[--_nb_elem];
    else cout << "pile vide" << endl ;
    if(_nb_elem <= 0) _etat = vide ;
    else _etat = ok ;
    return element ;
}
void pile:: afficher() {
    for(int i = _nb_elem-1 ; i>=0 ; i--) cout<<_table[i]<<endl
}
inline int pile::nb_elements() {return _nb_elem ; }
```

79

# SURCHARGE D'OPERATEURS

- Introduction à la surcharge d'opérateurs
- Surcharge par une fonction membre
- Surcharge par une fonction globale
- Opérateur d'affectation
- Surcharge de ++
- Opérateurs de conversion

80

## ■ Introduction à la surcharge d'opérateurs

- Le concepteur d'une classe doit fournir à l'utilisateur de celle ci toute une série d'opérateurs agissant sur les objets de la classe. Ceci permet une syntaxe intuitive de la classe.
- Par exemple, il est plus intuitif et plus clair d'additionner deux matrices en **surchargeant l'opérateur d'addition** et en écrivant :

```
result = m0 + m1; que d'écrire matrice_add(result, m0, m1);
```

### ► Règles d'utilisation :

- Il faut veiller à respecter l'esprit de l'opérateur.
- Il faut faire avec les types utilisateurs des opérations identiques à celles que font les opérateurs avec les types prédéfinis.
- La plupart des opérateurs sont surchargeables.
- Les opérateurs suivants ne sont pas surchargeables : **:: . .\* ?: sizeof**
- Il n'est pas possible de :
  - changer sa priorité
  - changer son associativité
  - changer sa pluralité (unaire, binaire, ternaire)
  - créer de nouveaux opérateurs

81

- ➔ Quand l'opérateur + (plus par exemple) est appelé, le compilateur génère un appel à la fonction operator + (plus).

- ➔ Ainsi, l'instruction **a = b + c;** est équivalente aux instructions :
  - **a = operator+(b, c); // fonction globale**
  - **a = b.operator+(c); // fonction membre**

### ➔ Les opérateurs

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

**➔ Ne peuvent être surchargés que comme des fonctions membres.**

82

## □ SURCHARGE PAR UNE FONCTION MEMBRE

- Par exemple, la surcharge des opérateurs + et = par des fonctions membres de la classe Matrice s'écrit :

```

const int dim1= 2, dim2 = 3; ← dimensions de la Matrice

class Matrice {← Matrice
    public:
        // ...
        Matrice operator=(const Matrice &n2);
        Matrice operator+(const Matrice &n2);
        // ...
    private:
        int _matrice[dim1][dim2];
        // ...
};

// ...
void main() {
    Matrice a, b, c;
    b + c ;← appel à : b.operator+( c );
    a = b + c;← appel à :
    a.operator=( b.operator+( c ) );
}

```

- Une fonction membre (non statique) peut toujours utiliser le pointeur caché this.
- Ci-dessus, this fait référence à l'objet b pour l'opérateur + et à l'objet a pour l'opérateur =.

83

### ► Définition de la fonction membre operator+ :

```

Matrice Matrice::operator+(const Matrice &c) {
    Matrice m;
    for(int i = 0; i < dim1; i++)
        for(int j = 0; j < dim2; j++)
            m._matrice[i][j] = this->_matrice[i][j] + c._matrice[i][j];
    return m;
}

```

- Quand on a le choix, l'utilisation d'une fonction membre pour surcharger un opérateur est préférable.
- Une fonction membre renforce l'encapsulation.
- Les opérateurs surchargés par des fonctions membres se transmettent aussi par héritage (sauf l'affectation).
- La fonction membre operator + peut elle même être surchargée, pour dans l'exemple qui suit, additionner à une matrice un vecteur :

```

class Matrice { //matrice dim1 x dim2
public: // ...
    Matrice operator=(const Matrice &n2);
    Matrice operator+(const Matrice &n2);
    Matrice operator+(const Vecteur &n2);
};

Matrice b, c;
Vecteur v;

b + c;      // appel à b.operator+(c);
b + v;      // appel à b.operator+(v); addition
            // entre une matrice et un vecteur
v + b;      // appel à v.operator+(b);
            // --> ERREUR si la classe Vecteur n'a
            // pas défini l'addition entre un vecteur
            // et une matrice

```

84

## □ SURCHARGE PAR UNE FONCTION GLOBALE

- Cette façon de procéder est plus adaptée à la surcharge des opérateurs binaires.
- Elle permet d'appliquer des conversions implicites au premier membre de l'expression.
- Exemple :

```
Class Nombre{
    friend Nombre operator+(const Nombre &, const Nombre &);
public:
    Nombre(int n = 0) { _nbre = n; }
    //...
private:
    int _nbre;
};

Nombre operator+(const Nombre &nbr1, const Nombre &nbr2) {
    Nombre n;
    n._nbre = nbr1._nbre + nbr2._nbre;
    return n;
}

void main() {
    Nombre n1(10);
    n1 + 20;
    30 + n1;
}
```

85

## □ OPERATEUR D'AFFECTATION

- C'est le même problème que pour le **constructeur de copie**.
- Le compilateur C++ construit par défaut un opérateur d'affectation "bête".
- L'opérateur **d'affectation** est obligatoirement une fonction membre et il doit fonctionner correctement dans les deux cas suivants :

```
x x1, x2, x3;
1) x1 = x1;
2) x1 = x2 = x3;
```

3 instances de la classe X

**Exemple : Opérateur d'affectation de la classe Matrice :**

```
const int dim1= 2, dim2 = 3;

class Matrice {
    public: const Matrice &operator=(const Matrice &m);
    private: int _matrice[dim1][dim2];
};

const Matrice &Matrice::operator=(const Matrice &m) {
    if ( &m != this ) {
        for(int i = 0; i < dim1; i++)
            for(int j = 0; j < dim2; j++)
                this->_matrice[i][j] = m._matrice[i][j];
    }
    return *this;
}
```

86

## □ SURCHARGE DE ++ et --

- Notation préfixée :

- fonction membre: `X operator++();`
  - fonction globale: `X operator++(X &);`

- Notation postfixée :

- fonction membre: `X operator++(int);`
  - fonction globale: `X operator++(X &, int);`

- Exemple:

```
class BigNombre {
    public:
        // ...
        BigNombre operator++();
        BigNombre operator++(int);
        // ...
};

void main() {
    BigNombre n1;
    n1++;
    ++n1;
}
```

préfixée

postfixée

notation postfixée

notation préfixée

87

## □ OPERATEURS DE CONVERSION

- Dans la définition complète d'une classe, il ne faut pas oublier de définir des opérateurs de conversions de types.

- Il existe deux types de conversions de types :

- La **conversion de type prédéfini** (ou défini préalablement) vers le type classe en question.
    - Ceci sera effectué grâce au **constructeur de conversion**.
  - la conversion du type classe vers un type prédéfini (ou défini préalablement).
    - Cette conversion sera effectuée par des fonctions de conversion.

- **Constructeur de conversion** :

- Un **constructeur avec un seul argument** de type T permet de réaliser une conversion d'une variable de type T vers un objet de la classe du constructeur.

```
class Nombre {
    public:
        Nombre(int n) { _nbre = n; }
    private:
        int _nbre;
};
void f1(Nombre n) { /* ... */ }
void main() { Nombre n = 2; f1(3); }
```

idem que : Nombre n(2);

Appel du constructeur Nombre(int) pour réaliser la conversion de l'entier 3 en un Nombre. Pas d'appel du constructeur de copie

Dans cet exemple, le **constructeur avec un argument** permet donc d'effectuer des **conversions d'entier en Nombre**.

88

► Fonction de conversion:

- Une **fonction de conversion** est une **méthode** qui effectue une conversion vers un **type T**.
- Elle est nommée operator T().
- Cette méthode n'a pas de type de retour (comme un **constructeur**), mais doit cependant bien retourner une valeur du type T.

```
class Article {  
    public :  
        Article(double prix=0.0):_prix(prix) {}  
        operator double() const { return _prix; }  
    private :  
        double _prix;  
};  
  
void main() {  
    double total;  
    Article biere(17.50);  
    total = 7 * biere;  
}
```

utilisation implicite de  
la conversion  
Article -> double

89

## Héritage et polymorphisme

- Qu'est ce que l'héritage ?
  - Exemple d'un mode de dérivation « public »
  - Comportement d'une classe dérivée
- Hiérarchie de classes
  - Principe
  - Conversion de type
- Héritage multiple
  - Principe
  - Problème liés à l'héritage multiple
- Le polymorphisme
  - Problème à résoudre : liaison statique
  - Solution : liaison dynamique
  - Classes abstraites
- Quelques précisions sur l'héritage
  - Modes d'accès au membres d'une classe
  - Autres modes de dérivation

90

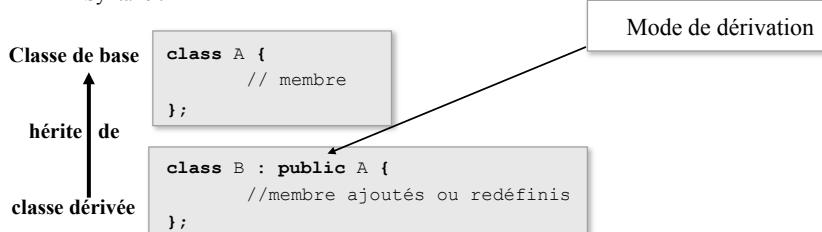
## □ Qu'est ce que l'héritage ?

- ➔ Permet de créer une nouvelle classe à partir d'une classe existante appelée **classe de base**.
- ➔ La **classe dérivée** hérite automatiquement des caractéristiques de la **classe de base**
  - ◆ **Attributs**
  - ◆ **Méthodes** (sauf le constructeur).
- ➔ La **classe dérivée** peut définir des attribut et des méthodes supplémentaires.
- ➔ L'héritage permet :
  - ◆ De réutiliser le code de la classe de base,
  - ◆ D'adapter la classe dérivée au cas particulier.
- ➔ Il existe trois modes de dérivation applicables dans des contextes différents:
  - ◆ **public**
  - ◆ **private**
  - ◆ **protected**.

91

## □ Qu'est ce que l'héritage ?

- ➔ L'héritage, également appelé **dérivation**, permet de créer une nouvelle classe à partir d'une classe déjà existante, la **classe de base** (ou super classe).
  - ◆ "Il est plus facile de modifier que de réinventer »
- ➔ La nouvelle classe (ou classe dérivée ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la **classe de base** et ainsi réutiliser le code déjà écrit pour la **classe de base**.
- ➔ On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.
- ➔ Syntaxe :



- La classe **B** hérite de façon publique de la classe A.
- Tous les membres publics ou protégés de la classe A font partie de l'interface de la classe B.

92

## ■ Exemple du mode de dérivation public :

→ **Exemple d' une mini-application:** Gestion des salariés d ' une entreprise

♦ **Enoncé:** Chaque salarié a un nom et un numéro de bureau auquel est associé un numéro de téléphone. L'entreprise emploie 3 catégories différentes de salariés, pour lesquels les modalités de calcul du salaire sont différentes:

- ◆ Les **employés**: leur salaire est égal au produit de leur taux par le nombre d'heures qu'ils effectuent dans le mois.
- ◆ Les **commerciaux**: leur salaire est calculé selon le même principe que celui des employés avec en plus un pourcentage sur leur chiffre d'affaire du mois
- ◆ Les **directeurs**: ils touchent un salaire fixe auquel s'ajoute une prime définie en méthode du nombre de salariés qu'ils encadrent.

→ Le C++, en **POO**, propose une solution élégante et efficace pour exprimer cette application:

- ◆ Tous les salariés possèdent des caractéristiques communes :
  - ◆ Un **nom**
  - ◆ Un **bureau**
- ◆ Il est donc possible de définir une **classe salarié** qui possède les attributs et les opérations communs à tous les salariés de l'entreprise.

93

```
class salaries
{
public:
    salaries(char* nom int bureau = 0);
    ~salaries() {delete[] nom; }
    void afficher();
    int set_bureau(int bureau) {_bureau = bureau;}
    char* get_telephone();
private:
    char* _nom;
    int _bureau;
};

salaries::salaries (char* nom int bureau) {
    _nom = new char[strlen(nom)+1];
    strcpy(_nom, nom);
    set_bureau(bureau);
}

void salaries:: afficher(){
    cout<<"nom:"<<nom <<"bureau:"<<bureau
    <<"tel:"<<get_telephone()<<endl;
}

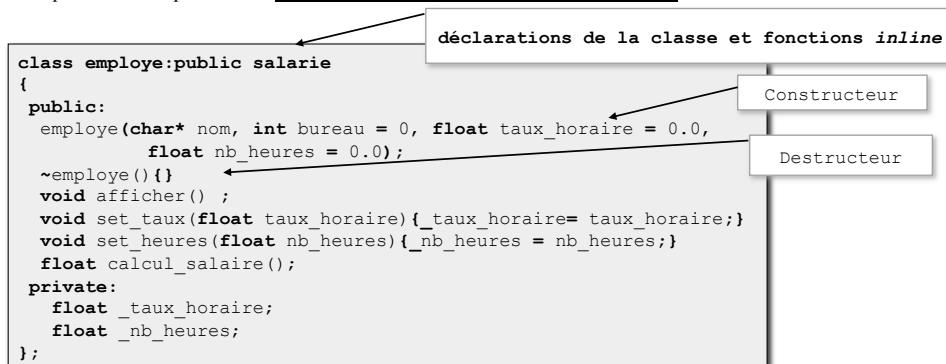
char* salaries:: get_telephone(){
    static char* telephone[]={"001", "002", "003", "004", "005", "006"};
    int nb_tels = sizeof(telephone)/sizeof(*telephone); //      6
    if(_bureau > 0 && _bureau <= nb_tels) return telephone[_bureau-1];
    else return "000";
}
```

The diagram illustrates the structure of the `salaries` class. It shows the class definition with its public and private sections. The `public` section contains the constructor (`salaries(char* nom int bureau = 0)`), destructor (`~salaries()`), and methods `afficher()` and `set_bureau(int bureau)`. The `private` section contains the private attributes `_nom` and `_bureau`. The diagram also highlights the `Constructeur` (constructor), `Destructeur` (destructor), and the `implantation des méthodes de la classe salaries` (implementation of the salary class methods). Arrows point from the class definition to these specific parts of the code.

94

## ■ Comportement d' une classe dérivée:

➔ Il est possible de définir une classe « **employe** » qui hérite des caractéristiques de la classe « **salarie** » et qui définit uniquement les attributs et méthodes spécifiques aux employés.



➔ Le **constructeur** d' une classe de base doit être appelé pour chaque instance d' une classe dérivée, dans la définition même de son constructeur, afin d' initialiser les attributs héritées par la classe dérivée.

95

### Implantation des méthodes de la classe employe

#### Implantation du constructeur de la classe employe

Appel du constructeur de la classe salarie

#### Surcharge de la méthode afficher

appel de la méthode afficher de la classe salarie

```

employe::employe(char* nom int bureau,
                 float taux_horaire, float nb_heures)
    :salarie(nom, bureau)
{
    set_taux(taux_horaire);
    set_heures(nb_heures);
}

void employe::afficher()
{
    salarie::afficher();
    cout<< "taux_h:<< _taux_horaire
          <<"heures work:<< _nb_heures << endl;
}

float employe :: calcul_salaire()
{
    return _taux_horaire* _nb_heures;
}

```

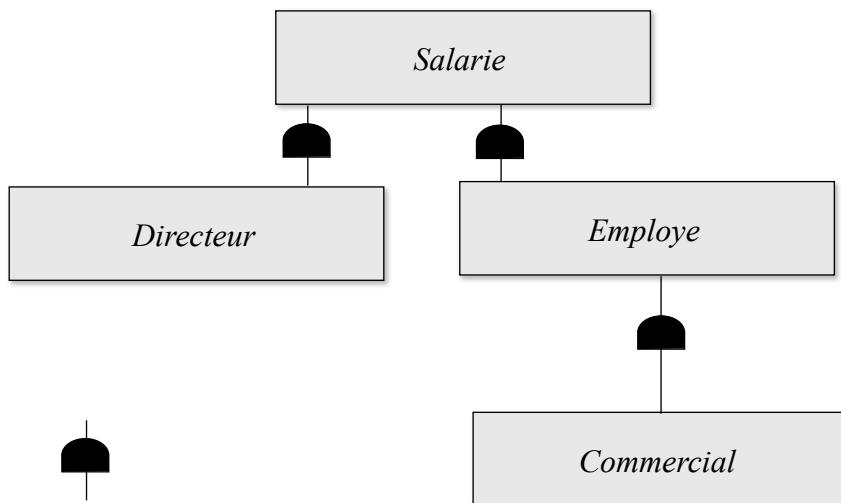
96

## □ Hiérarchie de classes

- Lorsque plusieurs classes sont dérivées à partir d'une même classe de base, cela définit une hiérarchie de classes.
- Si on envisage la classe commercial: un commercial est un employé auquel sont ajoutés deux éléments spécifiques:
  - le chiffre d' affaire
  - le pourcentage
- La classe employe est donc une classe de base pour la classe commercial.
- La classe salarie est une classe de base indirecte commercial.
- En revanche si on envisage une classe directeur:
  - C'est une classe dérivée de classe salarie , mais pas de la classe employe.

97

## Hiérarchie de l' application



*Signifie hérite de*

98

déclarations de la classe et fonctions *inline*

```

class commercial:public employe
{
public:
    commercial(char* nom int bureau = 0, float taux_horaire = 0.0,
               float nb_heures = 0.0, float pourcentage=0.0,
               float chiffre_affaire);
    ~commercial(){}
    void afficher();
    void set_pourcentage(float pourcentage){_pourcentage = pourcentage;}
    void set_chiffre_affaire(float chiffre_affaire){
        _chiffre_affaire = chiffre_affaire;
    }
    float calcul_salaire();
private:
    float _pourcentage;
    float _chiffre_affaire;
};

```

99

implantation des méthodes de la classe commercial

```

commercial::commercial(char* nom int bureau, float taux_horaire,
                      float nb_heures, float pourcentage, float chiffre_affaire)
:employeur(nom, bureau, taux_horaire, nb_heures)
{
    set_pourcentage(pourcentage);
    set_chiffre_affaire(chiffre_affaire);
}
void commercial::afficher()
{
    employer::afficher();
    cout<<" pourcentage des ventes"<< _pourcentage
         <<" chiffre affaire :"<< _chiffre_affaire <<endl;
}
float commercial :: calcul_salaire()
{
    return employer::calcul_salaire() +(_pourcentage* _chiffre_affaire);
}

```

100

```

class directeur:public salarie
{
public:
    directeur(char* nom, int bureau = 0, float fixe = 0.0,
              float prime = 0.0, int nb_employe=0);
    ~ directeur(){}
    void afficher() ;
    void set_fixe(float fixe){_fixe = fixe;}
    void set_prime(float prime){_prime = prime;}
    void set_nb_employe(int nb_employe){_nb_employe = nb_employe;}
    float calcul_salaire();
private:
    float _fixe ;
    float _prime;
    int _nb_employe;
};

```

déclarations de la classe et fonctions *inline*

Constructeur

Destructeur

101

```

directeur::directeur(char* nom, int bureau, float fixe,
                     float prime, int nb_employes)
    :salarie(nom, bureau)
{
    set_fixe(fixe);
    set_prime(prime);
    set_nb_employe(nb_employe);
}
void directeur::afficher()
{
    salarie::afficher();
    cout<<" salaire fixe:"<< _fixe <<"la prime"<< _prime <<
         <<"nombre de salarié:"<< _nb_employe << endl;
}
float directeur :: calcul_salaire()
{
    return _fixe + _prime;
}

```

Implantation des méthodes de la classe directeur

102

## MODE DE DERIVATION

- Lors de la définition de la classe dérivée il est possible de spécifier le **mode de dérivation** par l'emploi d'un des mots-clé suivants :
  - **public**
  - **protected**
  - **private**
- Ce **mode de dérivation** détermine quels membres de la classe de base sont accessibles dans la classe dérivée.
- Au cas où aucun **mode de dérivation** n'est spécifié, le compilateur C++ prend par défaut le mot-clé:
  - **private** pour une classe
  - **public** pour une structure (attributs).
- Les membres privés de la classe de base ne sont jamais accessibles par les membres des classes dérivées.

103

## MODE DE DERIVATION

### ► Héritage public :

- Il donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée.
- C'est la forme la plus courante d'héritage, car il permet de modéliser les relations « Y est une sorte de X » ou « Y est une spécialisation de la classe de base X ».

### ► Exemple :

```
class Vehicule {  
    public:  
        void pub1();  
    protected:  
        void prot1();  
    private:  
        void priv1();  
};  
  
class Voiture : public Vehicule {  
    public:  
        int pub2() {  
            pub1();      // OK  
            prot1();    // OK  
            priv1();    // ERREUR  
        }  
};
```

```
Voiture safrane;  
safrane.pub1(); // OK  
safrane.pub2(); // OK
```

104

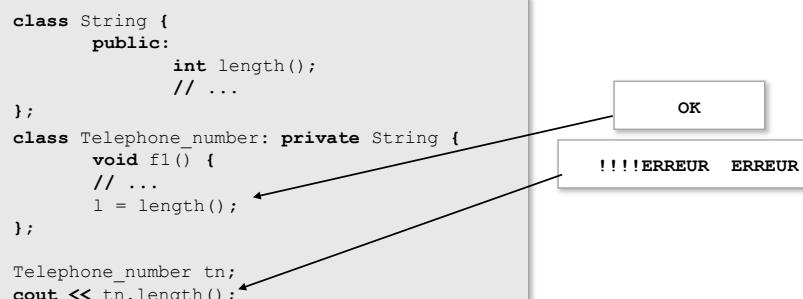
## MODE DE DERIVATION

### ► Héritage privé :

- Il donne aux membres publics et protégés de la classe de base le statut de membres privés dans la classe dérivée.
- Il permet de modéliser les relations «Y est composé de un ou plusieurs X».
  - Plutôt que d'hériter de façon privée de la classe de base X, on peut faire de la classe de base une donnée membre (composition).

#### ► Exemple :

```
class String {  
    public:  
        int length();  
        // ...  
};  
class Telephone_number: private String {  
    void f1() {  
        // ...  
        l = length();  
    };  
  
    Telephone_number tn;  
    cout << tn.length();
```



OK

!!!!ERREUR ERREUR

105

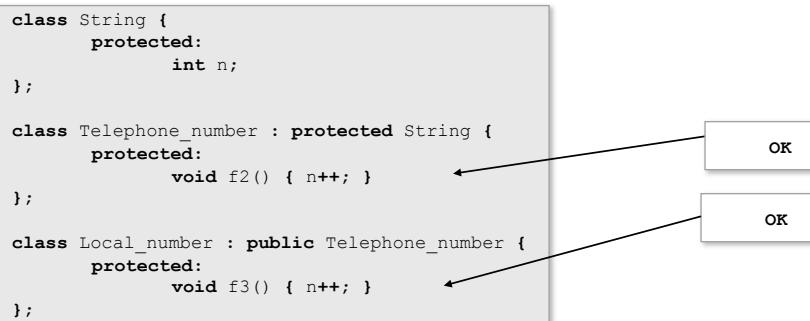
## MODE DE DERIVATION

### ♦ Héritage protégé :

- Il donne aux membres publics et protégés de la classe de base le statut de membres protégés dans la classe dérivée.
- L'héritage fait partie de l'interface mais n'est pas accessible aux utilisateurs.

#### ► Exemple :

```
class String {  
    protected:  
        int n;  
};  
  
class Telephone_number : protected String {  
    protected:  
        void f2() { n++; }  
};  
  
class Local_number : public Telephone_number {  
    protected:  
        void f3() { n++; }  
};
```



OK

OK

106

## Tableau résumé de l'accès aux membres

		Statut dans la classe de base	Statut dans la classe dérivée
mode de dérivation	public	public	public
		protected	protected
		private	inaccessible
	protected	public	protected
		protected	protected
		private	inaccessible
	private	public	private
		protected	private
		private	inaccessible

107

### REDEFINITION DE METHODES DANS LA CLASSE DERIVEE

- On peut redéfinir une **fonction** dans une **classe dérivée** si on lui donne le **même nom** que dans la **classe de base**.
- Il sera possible de les différencier avec l'**opérateur :: de résolution de portée**.

Exemple :

```
class X {
    public:
        void f1();
        void f2();
    protected:
        int XXX;
};
```

```
class Y : public X {
    public:
        void f2();
        void f3();
};
```

```
void Y::f3() {
    X::f2();
    X::XXX = 12;
    f1();
    f2();
}
```

The diagram illustrates the resolution of member functions in class Y:

- An arrow points from the call to `X::f2()` to a callout box labeled "appel de f2 de la classe Y".
- An arrow points from the assignment `X::XXX = 12;` to a callout box labeled "accès au membre XXX de la classe X".
- An arrow points from the call to `f1()` to a callout box labeled "appel de f1 de la classe X".
- An arrow points from the final call to `f2()` to a callout box labeled "f2 de la classe X".

108

## AJUSTEMENT D'ACCES

- Lors d'un héritage protégé ou privé, nous pouvons spécifier que certains membres de la classe ancêtre conservent leur mode d'accès dans la classe dérivée.

► Ce mécanisme, appelé **déclaration d'accès**, ne permet en aucun cas d'augmenter ou de diminuer la visibilité d'un membre de la classe de base.

- Exemple :

```
class X {
    public:
        void f1();
        void f2();
    protected:
        void f3();
        void f4();
};

class Y : private X {
    public:
        X::f1;
        X::f3; // ERREUR;
    protected:
        X::f4;
        X::f2; // ERREUR;
};
```

f1() reste public dans Y  
un membre protégé ne peut pas devenir public  
f4() reste protégé dans Y  
un membre public ne peut pas devenir protégé

109

## HERITAGE DES CONSTRUCTEURS/DESTRUCTEURS

- Les **constructeurs**, **destructeurs** et **opérateurs d'affectation** ne sont jamais hérités.
- Les **constructeurs** par défaut des classes de bases sont automatiquement appellés avant le constructeur de la classe dérivée.
- Pour ne pas appeler les **constructeurs** par défaut, mais des **constructeurs** avec des paramètres, vous devez employer une liste d'initialisation.
- L'appel des **destructeurs** se fera dans l'ordre inverse des constructeurs.

- Exemple 1 :

```
class Vehicule {
    public:
        Vehicule() { cout << "Vehicule" << endl; }
        ~Vehicule() { cout << "~Vehicule" << endl; }
};
class Voiture : public Vehicule {
    public:
        Voiture() { cout << "Voiture" << endl; }
        ~Voiture() { cout << "~Voiture" << endl; }
};
void main() {
    Voiture *R21 = new Voiture;
    // ...
    delete R21;
}
```

/\*se programme affiche :\*/  
Vehicule  
Voiture  
~Voiture  
~Vehicule

110

► Exemple 2: appel des constructeurs avec paramètres :

```

class Vehicule {
    public:
        Vehicule(char *nom, int places);
        //...
    };

    class Voiture : public Vehicule {
        private:
            int _cv; // puissance fiscale
        public:
            Voiture(char *n, int p, int cv);
            // ...
    };

    Voiture::Voiture(char *n, int p, int cv): Vehicule(n, p)
    {
        /*... */
    }
}

```

111

## HERITAGE ET AMITIE

- L'amitié pour une classe s'hérite, mais uniquement sur les membres de la classe hérités.
- Elle ne se propage pas aux nouveaux membres de la classe dérivée et ne s'étend pas aux générations suivantes.
- Exemple :

```

class A {
    friend class test1;
    public:
        A( int n= 0): _a(n) {}
    private:
        int _a;
};

class test1 {
    public:
        test1(int n= 0): _a0(n) {}
        void affiche1() { cout << _a0._a << // OK: }
    private:
        A _a0;
};

class test2: public test1 {
    public:
        test2(int z0= 0, int z1= 0): test1(z0), _a1(z1) {}
        void Ecrit() { cout << _a1._a; // ERREUR: }
    private:
        A _a1;
};

```

test1 est amie de A

test2 n'est pas amie de A

- L'amitié pour une fonction ne s'hérite pas. A chaque dérivation, vous devez redéfinir les relations d'amitié avec les fonctions.

112

## CONVERSION DE TYPE DANS UNE HIERARCHIE DE CLASSES

- Il est possible de convertir implicitement une instance d'une classe dérivée en une instance de la classe de base si l'héritage est public.
- L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres de la classe dérivée.

► Exemple :

```
class Vehicule {  
    public:  
        void f1();  
        // ...  
};  
  
class Voiture : public Vehicule {  
    public:  
        int f1();  
        // ...  
};  
  
void traitement1(Vehicule v) {  
    // ...  
    v.f1(); // OK  
    // ...  
}
```



```
void main()  
{  
    Voiture R25;  
    traitement1( R25 );  
}
```

113

- De la même façon on peut utiliser des pointeurs ou des références :

- Un pointeur sur un objet d'une classe dérivée peut être implicitement converti en un pointeur sur un objet de la classe de base.
- Une référence sur un objet d'une classe dérivée peut être implicitement converti en référence sur un objet de la classe de base.
- Cette conversion n'est possible que si l'héritage est public, car la classe de base doit posséder des membres public accessibles
- Ce n'est pas le cas d'un héritage protected ou private.
- C'est le type du pointeur qui détermine laquelle des méthodes f1() est appelée:

```
void traitement1(Vehicule *v) {  
    // ...  
    v->f1(); // OK  
    // ...  
}  
  
void main()  
{  
    Voiture R25;  
    traitement1( &R25 );  
}
```

114

## HERITAGE MULTIPLE

- En langage C++, il est possible d'utiliser l'**héritage multiple**.
- Il permet de créer des classes dérivées à partir de plusieurs classes de base.
- Pour chaque classe de base, on peut définir le **mode d'héritage**.

```

class A {
public:
    void fa() { /* ... */ }
protected:
    int _x;
};

class B {
public:
    void fb() { /* ... */ }
protected:
    int _x;
};

class C: public B, public A {
public:
    void fc();
};

```

résolution de portée  
pour lever l'ambiguité

```

void C::fc() {
    int i;
    fa();
    i = A::_x + B::_x;
}

```

115

## HERITAGE MULTIPLE

### ► Ordre d'appel des constructeurs

- Dans l'**héritage multiple**, les **constructeurs** sont appelés dans l'ordre de déclaration de l'héritage.
- Dans l'exemple suivant, le constructeur par défaut de la classe C appelle:
  - le constructeur par défaut de la classe B,
  - puis celui de la classe A
  - et en dernier lieu le constructeur de la classe dérivée.
  - même si une liste d'initialisation existe.

```

class A {
public:
    A(int n=0) { /* ... */ }
    // ...
};

class B {
public:
    B(int n=0) { /* ... */ }
    // ...
};

class C: public B, public A {
public:
    C(int i, int j) : A(i), B(j) { /* ... */ }
};

void main() {C objet_c; }

```

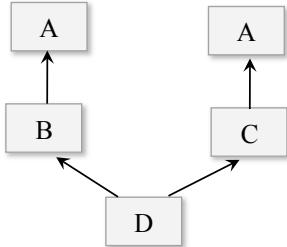
ordre d'appel des constructeurs  
des classes de base et  
pourtant la liste d'initialisation  
est différente!!!!

liste d'initialisation

appel des constructeurs B(), A() et C()

116

## HERITAGE VIRTUEL



```

Class A { int _basse; };

Class B: public A {/*...*/}

Class C: public A {/*...*/}

Class D: public B, public C {/*...*/}
  
```

► Position du problème:

- Un objet de la classe **D** contiendra **deux fois** les données héritées de la classe de base A:
  - **Une fois** par héritage de la classe **B**
  - **Et une autre fois** par **C**.
- Il y a donc **deux fois** le membre \_base dans la classe **D**
- L'accès au **membre \_base** de la **classe A** se fait en **levant l'ambiguïté**.

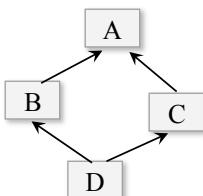
```

void main() {
    D od;
    od._base = 0;      // ERREUR, ambiguïté
    od.B::_base = 1;  // OK
    od.C::_base = 2;  // OK
}
  
```

117

## HERITAGE VIRTUEL

- Il est possible de n'avoir qu'une **occurrence** des membres de la classe de base, en utilisant **l'héritage virtuel**.
- Pour que la classe D n'hérite qu'une seule fois de la classe A, il faut que les classes B et C héritent virtuellement de A.



```

Class A { int _basse; };

Class B: virtual public A {/*...*/}

Class C: virtual public A {/*...*/}

Class D: public B, public C {/*...*/}
  
```

Permet de n'avoir dans la classe D qu'une seule occurrence des données héritées de la classe de base A.

```

void main() {
    D od;
    od._base = 0; // OK, pas d'ambiguïté
}
  
```

- Il ne faut pas confondre ce statut « **virtual** » de déclaration d'héritage des classes avec celui des **membres virtuels** que nous allons étudier.
- Ici, Le mot-clé « **virtual** » précise au compilateur les classes à ne pas dupliquer.

118

## POLYMORPHISME

- L'héritage nous permet de réutiliser le code écrit pour la classe de base dans les autres classes de la hiérarchie des classes de l'application.
- Le polymorphisme rendra possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérarchie des classes.
- En C++, le polymorphisme est mis en œuvre par l'utilisation des fonctions virtuelles.

### ► Fonctions virtuelles

```
class ObjGraph {  
public:  
    void print() const { cout << "ObjGraph::print()"; }  
};  
class Bouton: public ObjGraph {  
public:    void print() const { cout << "Bouton::print()"; }  
};  
class Fenetre: public ObjGraph {  
public:  
    void print() const { cout << "Fenetre::print()"; }  
};  
void traitement(const ObjGraph &og) {  
    // ...  
    og.print();  
}
```

Qu'affiche ce programme ???

```
void main() {  
    Bouton OK;  
    Fenetre windows97;  
    traitement(OK);  
    traitement(windows97);  
}
```

119

## POLYMORPHISME

- Comme nous l'avons déjà vu, l'instruction «`og.print()` de `traitement()`» appellera la méthode `print()` de la classe `ObjGraph`.

### ► La réponse est donc :

```
void main() {  
    Bouton OK;  
    Fenetre windows97;  
    traitement(OK);  
    traitement(windows97);  
}
```

affichage de `ObjGraph::print()`

affichage de `ObjGraph::print()`

- Si dans la fonction `traitement()` nous voulons appeler la méthode « `print()` » selon la classe à laquelle appartient l'instance, nous devons définir, dans la classe de base, la méthode « `print()` » comme étant virtuelle :

```
class ObjGraph {  
public:  
    // ...  
    virtual void print() const {  
        cout << "ObjetGraph::print()" << endl;  
    }  
};
```

120

- Pour plus de clarté, le mot-clé « **virtual** » peut être répété devant les méthodes `print()` des classes `Bouton` et `Fenetre`:

```
class Bouton: public ObjGraph {
public:
    virtual void print() const {
        cout << "Bouton::print()";
    }
};

class Fenetre: public ObjGraph {
public:
    virtual void print() const {
        cout << "Fenetre::print()";
    }
};
```

- On appelle ce comportement, le **polymorphisme**.

- Lorsque le compilateur rencontre une méthode virtuelle, il sait qu'il faut attendre l'exécution pour déterminer la bonne méthode à appeler.

121

## ► Destructeur virtuel

- Il ne faut pas oublier de définir le **destructeur** comme « **virtual** » lorsque l'on utilise une méthode virtuelle:

```
class ObjGraph {
public:
    //...
    virtual ~ObjGraph() {
        cout << "fin de ObjGraph\n";
    }
};

class Fenetre : public ObjGraph {
public:
    // ...
    virtual ~Fenetre() {
        cout << "fin de Fenêtre ";
    }
};

void main() {
    Fenetre *Windows97 = new Fenetre;
    ObjGraph *og = Windows97;
    // ...
    delete og;
}
```

affichage de:  
**fin de Fenêtre fin de ObjGraph**

Si le **destructeur** n'avait pas été **virtuel**, l'affichage aurait été:  
**fin de ObjGraph**

122

- Un **constructeur**, par contre, ne peut pas être déclaré comme **virtuel**.
  - Une **méthode statique** ne peut, pas, être déclaré comme **virtuelle**.
  - Lors de l'**héritage**, le statut de l'accessibilité de la **méthode virtuelle** (*public*, *protégé* ou *privé*) est conservé dans toutes les classes dérivées, même si elle est redéfinie avec un statut différent.
- Le statut de la classe de base prime.

123

## □ CLASSES ABSTRAITES

- Il arrive souvent que la méthode virtuelle définie dans la classe de base serve de cadre générique pour les méthodes virtuelles des classes dérivées.
  - Ceci permet de garantir une bonne homogénéité de votre architecture de classes.
- Une **classe** est dite **abstraite** si elle contient au moins une méthode virtuelle pure.
- On ne peut pas créer d'instance d'une **classe abstraite**
- Une **classe abstraite** ne peut pas être utilisée comme argument ou type de retour d'une fonction.
- Par contre, les pointeurs et les références sur une **classe abstraite** sont parfaitement légitimes et justifiés.
- **Méthode virtuelle pure:** une telle méthode se déclare en ajoutant un = **0** à la fin de sa déclaration.

```
class ObjGraph {
public:
    virtual void print() const = 0;
};

void main() {
    ObjGraph og; // ERREUR
    // ...
}
```

- On ne peut utiliser une **classe abstraite** qu'à partir d'un pointeur ou d'une référence.
- Contrairement à une méthode virtuelle "normale", une **méthode virtuelle pure** n'est pas obligé de fournir une définition pour ObjGraph::print().
- Une **classe dérivée** qui ne redéfinit pas une **méthode virtuelle pure** est elle aussi abstraite

124

## FLUX

- Généralités sur les flux
- Flots et classes
- Le flot de sortie *ostream* et ses méthodes
- Le flot d'entrée *istream* et ses méthodes
- Contrôle de l'état d'un flux
- Associer un flot d'E/S à un fichier
- Formatage de l'information
- Les manipulateurs

125

### □ Généralités sur les flux

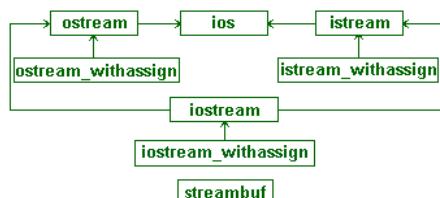
- Un **flux** (ou stream) est une abstraction logicielle représentant un flot de données entre :
  - Une source produisant de l'information
  - Une cible consommant cette information.
- Il peut être représenté comme un buffer et des mécanismes associés à celui-ci.
- Quand le **flux** est créé, il prend en charge l'acheminement de ces données.
- Comme pour le langage C, les instructions entrées/sorties ne font pas partie des instructions du langage.
- Elles sont dans une librairie standardisée qui implémente les flots à partir de classes.
- Par défaut, chaque programme C++ peut utiliser **3 flots** :
  - **cout** qui correspond à la sortie standard
  - **cin** qui correspond à l'entrée standard
  - **cerr** qui correspond à la sortie standard d'erreur.
- Pour utiliser d'autres flots, vous devez donc créer et attacher ces flots à des fichiers (fichiers normaux ou fichiers spéciaux) ou à des tableaux de caractères.

126

## □ Flots et classes : manipulation des périphériques standards

► Les classes déclarées dans « **iostream.h** » permettent la manipulation des périphériques standards :

- ▶ **ios** : classe de base des entrées/sorties par flot.
  - ▶ Elle contient un objet de la classe **streambuf** pour la gestion des tampons d'entrées/sorties.
- ▶ **istream**: classe dérivée de **ios** pour les flots en entrée.
- ▶ **ostream**: classe dérivée de **ios** pour les flots en sortie.
- ▶ **iostream**: classe dérivée de **istream** et de **ostream** pour les flots bidirectionnels.
- ▶ **istream\_withassign**, **ostream\_withassign** et **iostream\_withassign** :
  - ▶ classes dérivées respectivement de **istream**, **ostream** et **iostream** et qui ajoute l'opérateur d'affectation.
  - ▶ Les flots standards **cin**, **cout** et **cerr** sont des instances de ces classes.

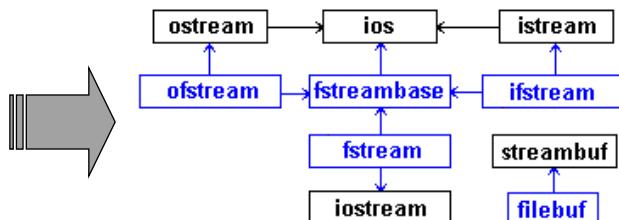


127

## □ Flots et classes : manipulation des fichiers disques

► Les classes déclarées dans « **fstream.h** » permettent la manipulation des fichiers disques :

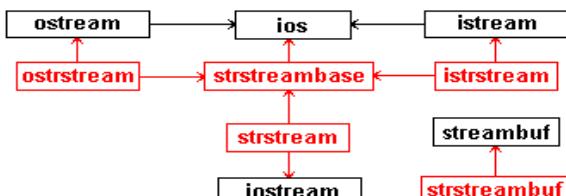
- ▶ **fstreambase** : classe de base pour les classes dérivées : **ifstream**, **ofstream** et **fstream**.
  - ▶ Elle même est dérivée de **ios**;
  - ▶ Elle contient un objet de la classe **filebuf** (dérivée de **streambuf**).
- ▶ **ifstream** : classe permettant d'effectuer des entrées à partir des fichiers.
- ▶ **ofstream** : classe permettant d'effectuer des sorties sur des fichiers.
- ▶ **fstream** : classe permettant d'effectuer des entrées/sorties à partir des fichiers.



128

## ❑ Flots et classes : simulation des opérations d' E/S

- Les classes déclarées dans **strstream.h** permettent de simuler des opérations d'entrées/sorties avec des tampons en mémoire centrale.
- Elles opèrent de la même façon que les fonctions du langage C **sprintf()** et **sscanf()** :
  - ▷ **strstreambase** : classe de base pour les classes suivantes.
    - ▷ Elle contient un objet de la classe strstreambuf (dérivée de streambuf).
  - ▷ **istrstream** : classe dérivée de **strstreambase** et de **istream** permettant la lecture dans un tampon mémoire (à la manière de la fonction **sscanf**).
  - ▷ **oststream** : classe dérivée de **strstreambase** et de **ostream** permettant l'écriture dans un tampon mémoire (à la manière de la fonction **sprintf**).
  - ▷ **strstream** : classe dérivée de **istrstream** et de **iostream** permettant la lecture et l'écriture dans un tampon mémoire.



129

## ❑ Flot de sortie: **ostream**

- ▷ Il fournit des sorties formatées et non formatées (dans un streambuf)
- ▷ Surdéfinition de l'opérateur <<
- ▷ Il gère les types prédéfinis du langage C++
- ▷ On peut (doit) le surdéfinir pour ses propres types

### ► Surdéfinition de l'opérateur d'injection dans un flot :

- ▷ Exemple :

```
class Exemple {  
    friend ostream &operator<<(ostream &os, const Exemple &ex);  
public : // ...  
private :  
    char _nom[20];  
    int _valeur;  
};  
ostream &operator<<(ostream &os, const Exemple &ex) {  
    return os << ex._nom << " " << ex._valeur;  
}  
void main() {  
    Exemple e1;  
    cout << "Exemple = " << e1 << endl;  
}
```

La fonction **operator<<** doit retourner une référence sur l'ostream pour lequel il a été appelé afin de pouvoir le cascader dans une même instruction, comme dans la fonction **main()**.

130

## □ Flot de sortie : méthodes de *ostream*

- En plus de l'opérateur d'injection (`<<`), la classe ostream contient les principales méthodes suivantes:

- ▷ **ostream & put(char c);** insèrent un caractère dans le flot
    - ▷ Exemple : `cout.put('\n');`
  - ▷ **ostream & write(const char \*, int n);** insèrent n caractères dans le flot
    - ▷ Exemple : `cout.write("Bonjour", 7);`
  - ▷ **streampos tellp();** retourne la position courante dans le flot
  - ▷ **ostream & seekp(streampos n);** se positionne à n octet(s) par rapport au début du flux.
    - ▷ Les positions dans un flot commencent à 0,
    - ▷ Le type streampos correspond à la position d'un caractère dans le fichier.
  - ▷ **ostream & seekp(streamoff dep, seek\_dir dir);** se positionne à dep octet(s) par rapport :
    - ▷ Au début du flot : `dir = beg`
    - ▷ A la position courante : `dir = cur`
    - ▷ A la fin du flot : `dir = end` (et `dep` est négatif)
  - ▷ **ostream & flush();** vide les tampons du flux
- Exemple :
- ```
streampos old_pos = fout.tellp();
fout.seekp(0, end);
cout << "Taille du fichier : " << fout.tellp() << " octet(s)\n";
fout.seekp(old_pos, beg);
```
- 
- mémorise la position  
se positionne à la fin du flux  
se repositionne comme au départ

131

## □ Flot d'entrée : *istream*

- Il fournit des entrées formatées et non formatées (dans un streambuf)

- ▷ Surdéfinition de l'opérateur >>
- ▷ Il gère les types prédéfinis du langage C++
- ▷ On peut (doit) le surdéfinir pour ses propres types
- ▷ Par défaut >> ignore tous les espaces (voir `ios::skipws`)

- Surdéfinition de l'opérateur d'extraction >> :

- ▷ Exemple :

```
class Exemple {
    friend istream &operator>>(istream &is, Exemple &ex);
public :
    // ...
private :
    char _nom[20];
    int _valeur;
};
istream &operator>>(istream &is, Exemple &ex) {
    return is >> ex._nom >> ex._valeur;
}
void main() {
    Exemple e1, e2;
    cout << "Entrez un nom et une valeur : ";
    cin >> e1 >> e2;
}
```

► La fonction `operator>>` doit retourner une référence sur l'istream pour lequel il a été appelé, afin de pouvoir le caser dans une même instruction

132

## □ Flot d'entrée : méthodes de *istream*

► En plus de l'opérateur d'extraction (`>>`), la classe *istream* contient les principales **méthodes** suivantes :

### ► lecture d'un caractère :

- `int get();` retourne la valeur du caractère lu (ou *EOF* si la fin du fichier est atteinte)
- `istream & get(char &c);` extrait le premier caractère du flux (même si un espace) et le place dans *c*.
- `int peek();` lecture non destructrice du caractère suivant dans le flux.
- Retourne le code du caractère ou *EOF* si la fin du fichier est atteinte.

### ► lecture d'une chaîne de caractères :

- `istream & get(char *ch, int n, char delim='\\n');` extrait *n-1* caractères du flux et les place à l'adresse *ch*.
  - La lecture s'arrête au délimiteur qui est par défaut le '`\n`' ou la fin de fichier.
  - Le délimiteur ('`\n`' par défaut) n'est pas extrait du flux.
- `istream & getline(char *ch, int n, char delim='\\n');` comme la méthode précédente sauf que le délimiteur est extrait du flux mais n'est pas recopié dans le tampon.
- `istream & read(char *ch, int n);` extrait un bloc d'au plus *n* octets du flux et les place à l'adresse *ch*. Le nombre d'octets effectivement lus peut être obtenu par la méthode `gcount()`.

133

## □ Flot d'entrée : méthodes de *istream*

- `int gcount();` retourne le nombre de caractères non formatés extraits lors de la dernière lecture.
- `streampos tellg();` retourne la position courante dans le flot.
- `istream & seekg(streampos n);` se positionne à *n* octet(s) par rapport au début du flux.
  - Les positions dans un flot commencent à *0*
  - Le type `streampos` correspond à la position d'un caractère dans le fichier.
- `istream & seekg(streamoff dep, seek_dir dir);` se positionne à *dep* octet(s) par rapport:
  - au début du flot : `dir = beg`
  - à la position courante : `dir = cur`
  - à la fin du flot : `dir = end` (et *dep* est négatif!)
- `istream & flush();` vide les tampons du flux

134

## □ Contrôle de l'état d'un flux

- La classe ***ios*** décrit les aspects communs des flots **d'entrée/sortie**.
  - C'est une classe de base **virtuelle** pour tous les objets flots.
  - Vous n'aurez jamais à créer des objets de la classe « ***ios*** ».
  - Vous utiliserez ses méthodes pour tester l'état d'un flot ou pour contrôler le formatage des informations.
- **Méthodes de la classe de base « *ios* » :**
  - ***int good();*** retourne une valeur différente de zéro si la dernière opération *d'entrée/sortie* s'est effectuée avec succès et une valeur nulle en cas d'échec.
  - ***int fail();*** fait l'inverse de la méthode précédente.
  - ***int eof();*** retourne une valeur différente de zéro si la fin de fichier est atteinte et une valeur **nulle** dans le cas contraire.
  - ***int bad();*** retourne une valeur différente de zéro si vous avez tenté une opération interdite et une valeur nulle dans le cas contraire.
  - ***int rdstate();*** retourne la valeur de la variable d'état du flux. Retourne une valeur nulle si tout va bien.
  - ***void clear();*** remet à zéro l'indicateur d'erreur du flux. C'est une opération **obligatoirement** à faire après qu'une erreur se soit produite sur un flux.

135

## □ Associer un flot d'E/S à un fichier

- Il est possible de créer un objet flot associé à un fichier autre que les fichiers d'entrées/sorties standards.
- 3 classes permettant de manipuler des fichiers sur disque sont définies dans ***fstream.h*** :
  - ***ifstream*** : (dérivée de ***istream*** et de ***fstreambase***) permet l'accès du flux en lecture seulement
  - ***ofstream*** : (dérivée de ***ostream*** et de ***fstreambase***) permet l'accès du flux en écriture seulement
  - ***fstream*** : (dérivée de ***iostream*** et de ***fstreambase***) permet l'accès du flux en lecture/écriture.
- Chacune de ces classes utilise un buffer de la classe ***filebuf*** pour synchroniser les opérations entre le buffer et le flot.
- La classe ***fstreambase*** offre un lot de méthodes communes à ces classes (***open, close, attach ...***).

136

## □ Associer un flot d'E/S à un fichier

### ► Ouverture du fichier et association avec le flux :

► C'est la méthode `open()` qui permet d'ouvrir le fichier et d'associer un flux avec ce dernier.

► `void open(const char *name, int mode, int prot=filebuf::openprot);`

► `name` : nom du fichier à ouvrir

► `prot` : il définit les droits d'accès au fichier (par défaut les permissions de lecture/écriture sont positionnés) dans le cas d'une ouverture avec création (sous UNIX).

► `mode` : mode d'ouverture du fichier (`enum open_mode` de la classe `ios`) :

```
enum open_mode {
    app,           // ajout des données en fin de fichier.
    ate,           // positionnement à la fin du fichier.
    in,            // ouverture en lecture (par défaut pour ifstream).
    out,           // ouverture en écriture (par défaut pour ofstream).
    binary,        // ouverture en mode binaire (par défaut en mode texte).
    trunc,         // détruit le fichier s'il existe et le recrée (par défaut
                   // si out est spécifié sans que ate ou app ne soit activé).
    noreplace,     // si le fichier existe, l'ouverture échoue,
                   // sauf si ate ou app sont activés.
};
```

► Pour les classes «`ifstream, ofstream`» le mode par défaut est respectivement `ios::in` et `ios::out`.

137

## □ Associer un flot d'E/S à un fichier

### ► Exemple

```
#include <fstream.h>
ifstream f1;
f1.open("essai1.tmp");
ofstream f2;
f2.open("essai2.tmp");
fstream f3;
f3.open("essai3.tmp", ios::in | ios::out);
```

ouverture en lecture du fichier  
ouverture en écriture du fichier  
ouverture en lecture/écriture

► On peut aussi appeler les constructeurs des différentes classes et combiner les 2 opérations de définition du flot et d'ouverture.

```
#include <fstream.h>

ifstream f1("essai1.tmp"); // ouverture en lecture du fichier
ofstream f2("essai2.tmp"); // ouverture en écriture du fichier
fstream f3("essai3.tmp", ios::in | ios::out); // ouverture en lecture/écriture
```

138

## □ Formatage de l'information

- Rappelons qu'avec les fonctions *printf()* et *scanf()* du langage on doit fournir pour chaque opération d'entrée/sortie l'indicateur de format.
  - *%d, %f, %s, ...*
- En POO C++ chaque flot conserve en permanence un ensemble d'indicateurs spécifiant l'état de formatage.
  - Ceci permet de donner un comportement par défaut au flot.
- L'indicateur de format du flot, est un entier long défini (en *protected*) dans la classe ios, dont les classes stream héritent.

139

## □ Formatage de l'information

- Extrait de *ios* :

```
class ios {
public :
    // ...
    enum {
        skipws,      // ignore les espaces en entrée
        left,        // justifie les sorties à gauche
        right,       // justifie les sorties à droite
        internal,    // remplissage après le signe ou la base
        dec,         // conversion en décimal
        oct,         // conversion en octal
        hex,         // conversion en hexadécimal
        showbase,    // affiche l'indicateur de la base
        showpoint,   // affiche le point décimal avec les réels
        uppercase,   // affiche en majuscules les nombres hexadécimaux
        showpos,     // affiche le signe + devant les nombres positifs
        scientific,  // notation 1.234000E2 avec les réels
        fixed,       // notation 123.4 avec les réels
        unitbuf,    // vide les flots après une insertion
        stdio,       // permet d'utiliser stdout et cout
    };
protected :
    long x_flags; // indicateur de format
    // ...
};
```

140

## Formatage de l'information

- Les **valeurs** précédentes peuvent se combiner avec l'opérateur « | » comme dans l'exemple :
  - `ios::showbase | ios::showpoint | ios::showpos`
- Des **constantes** sont aussi définies dans la classe `ios` pour accéder à un groupe d'indicateurs :
  - `static const long basefield;` permet de choisir la base (`dec`, `oct` ou `hex`)
  - `static const long adjustfield;` permet de choisir son alignement (`left`, `right` ou `internal`)
  - `static const long floatfield;` permet de choisir sa notation pour les réels (scientific ou fixed)
- Les **méthodes** suivantes (définies dans la classe `ios`) permettent de lire ou de modifier la valeur des indicateurs de format :
  - `long flags();` Retourne la valeur de l'indicateur de format
  - `long flags(long f);`
    - Modifie l'ensemble des indicateurs en concordance avec la valeur de `f`.
    - Retourne l'ancienne valeur de l'indicateur.
  - `long setf(long setbits, long field);`
    - Remet à zéro les bits correspondants à `field` (`basefield`, `adjustfield` ou `floatfield`)
    - Positionne ceux désignés par `setbits`. Elle retourne l'ancienne valeur de l'indicateur de format.
  - `long setf(long f);`
    - Positionne l'indicateur de format.
    - Elle retourne l'ancienne valeur de l'indicateur de format.
  - `long unsetf(long);` Efface les indicateurs précisés.

141

## Formatage de l'information

### ► Exemples :

```
cout << setf(ios::dec, ios::basefield) << i;
cout << setf(ios::left, ios::adjustfield) << hex << 0xFF;
cout << setf(ios::internal, ios::adjustfield) << hex << 0xFF;
cout << setf(ios::scientific, ios::floatfield) << f;
long old = cout.setf(ios::left, ios::adjustfield);
cout << data;
cout.setf(old, ios::adjustfield);
cout.setf(0L, ios::basefield);
```

affiche : 000xFF

affiche : 0x00FF

état par défaut de basefield

positionne l'indicateur de format

### ► Exemples :

```
cout.setf( ios::skipws );
cout.setf( ios::dec | ios::right );
```

142

## □ Formatage de l'information

- Les méthodes de la classe `ios` manipulant le format des informations :
  - `int width(int)` ; Positionne la largeur du champ de sortie.
  - `int width()` ; Retourne la largeur du champ de sortie.
  - `char fill(char)` ; Positionne le caractère de remplissage
  - `char fill()` ; Retourne la valeur du caractère de remplissage
  - `int precision(int)` ; Positionne le nombre de caractères qu'occupe un réel.
  - `int precision()` ; Retourne la précision (voir ci-dessus).

- Exemples :

```
cout << "Largeur par défaut du champ de sortie : ";
cout << cout.width() << endl ← affiche: 0
cout.width(10);
cout.fill('*');
cout << '|' << 1234 << "|\\n"; ← affiche: |*****1234|
cout.setprecision(6);
cout << 12.345678 << endl; ← affiche: 12.3457
(noter l'arrondi à l'affichage)
```

143

## □ Les manipulateurs

- L'usage des méthodes de formatage de l'information rend les instructions lourdes à écrire et un peu trop longues.
- Les manipulateurs permettent d'écrire un code plus compact et plus lisible.
- Ainsi, au lieu d'écrire :

```
cout.width(10);
cout.fill('*');
cout.setf(ios::hex, ios::basefield);
cout << 123;
cout.flush();
```

- On préférera écrire l'instruction équivalente :

```
cout << setw(10) << setfill('*') << hex << 123 << flush;
```

- Dans laquelle, `setw` et `setfill` sont des manipulateurs avec un argument alors que `hex` et `flush` sont des manipulateurs sans argument.
- De plus nous pourrons écrire nos propres manipulateurs.

144

## □ Les manipulateurs

- ▶ Comment cela marche-t-il ?

▶ Les **manipulateurs** les plus célèbres sont ***endl*** et ***flush***. Vous les connaissez certainement. Voici comment ils sont définis :

```
ostream &flush(ostream &os) { return os.flush(); }
ostream &endl(ostream &os) { return os << '\n' << flush; }
```

- ▶ Pour pouvoir les utiliser dans une instruction comme :

```
cout << endl;
```

▶ la fonction **operator<<** est surchargeée dans la classe ostream comme :

```
ostream &ostream::operator<<(ostream& (* f)(ostream &)) {
    (*f) (*this);
    return *this;
}
```

- ▶ Il est donc très simple de définir son propre **manipulateur** (sans paramètre) :

▶ Exemple : création du **manipulateur tab** :

```
ostream &tab(ostream &os){ return os << '\t'; }
cout << 12 << tab << 34;
```

145

## □ Les manipulateurs

- ▶ Manipulateurs prédefinis :

▶ Les classes ios, istream et ostream implémentent les **manipulateurs prédefinis**.

▶ Le fichier d'entête iomanip.h :

- ▶ Défini un certain nombre de ces **manipulateurs**.
- ▶ Offre la possibilité de créer ses propres **manipulateurs**.
  - ▶ **dec** : la prochaine E/S utilise la base décimale
  - ▶ **hex** : la prochaine E/S utilise la base hexadécimale
  - ▶ **oct** : la prochaine E/S utilise la base octale
- ▶ **endl** : écrit un '\n' puis vide le flot
- ▶ **ends** : écrit un '\0' puis vide le flot
- ▶ **flush** : vide le flot
- ▶ **ws** : saute les espaces (sur un flot en entrée uniquement)
- ▶ **setbase(int b)** : Positionne la base **b** pour la prochaine sortie.
  - ▶ **n** vaut **0** pour le décimal, **8** pour loctal et **16** pour lhexadécimal.
- ▶ **setfill(int c)** : positionne le caractère de remplissage **c** pour la prochaine E/S
- ▶ **setprecision(int p)** : positionne la précision à **p** chiffres pour la prochaine E/S
- ▶ **setw(int l)** : positionne la largeur à **n** caractères.
- ▶ **setiosflags(long n)** : active les bits de l'indicateur de format spécifiés par l'entier **n**.
  - ▶ On l'utilise comme la méthode **flags()**.
- ▶ **resetiosflags(long b)** : désactive les bits de l'indicateur de format.

146

## □ Les manipulateurs

- Création d'un nouveau manipulateur (avec un paramètre) :
  - L'implémentation d'un **nouveau manipulateur** est implanté en **2 parties** :
    - le **manipulateur**: sa forme générale est (pour un *ostream*) :

```
ostream &nom_du_manipulateur(ostream &, type );
```

► *type* est le type du paramètre du manipulateur.
    - Cette fonction ne peut pas être appelée directement par une instruction d'entrée/sortie.
    - Elle sera appelée seulement par l'applicateur.
  - l'**applicateur**: il appelle le manipulateur.
    - C'est une fonction globale. Sa forme générale est :

```
xxxMANIP( type ) nom_du_manipulateur(type arg) {  
    return xxxMANIP(type) (nom_du_manipulateur, arg);  
}
```
  - Avec *xxx* valant :
    - *O*, pour les flocs manipulant un *ostream* (ou ses dérivées),
    - *I*, pour les flocs manipulant un *istream*
    - *S*, pour les flocs manipulant un *ios*
    - *IO* pour les flocs manipulant un *iostream*.

147

## □ Les manipulateurs

- Exemple d'un manipulateur pour afficher un entier en binaire :

```
#include <iomanip.h>
#include <limits.h> // ULONG_MAX

ostream &bin(ostream &os, long val) {
    unsigned long masque = ~(ULONG_MAX >> 1);
    while ( masque ) {
        os << ((val & masque) ? '1' : '0');
        masque >>= 1;
    }
    return os;
}

OMANIP(long) bin(long val) {
    return OMANIP(long) (bin, val);
}

void main() {
    cout << "1997 en binaire = " << bin(1997) << endl;
}
```

148

## Introduction à la programmation MFC

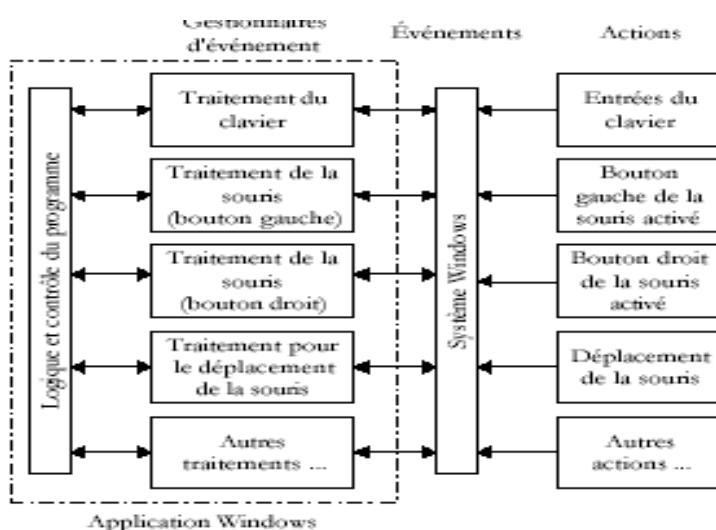
Le cadre de travail MFC (Microsoft Foundation Classes) sert à faciliter le développement des applications Windows

L' étude de MFC:

- Bien connaître l' apport de ce cadre de travail;
- Faciliter la construction des interfaces graphiques;
- Saisir la puissance de l' approche orientée objet;
- Exemples concrets de réalisation orientée objet.

149

## Philosophie de base



150

## Expédition et routage des messages (1)

- L' utilisateur du programme déclenche des événements par des actions:
  - L' utilisateur clique sur un bouton.
  - L' utilisateur modifie la taille d' une fenêtre.
  - Etc.
- Le programme lui-même peut également déclencher des événements:
  - L' activation d' une alarme Windows.
  - Événements envoyés par la logique du programme.
  - Etc.

151

## Expédition et routage des messages (2)

- Les événements Windows sont traduits en messages par le MFC.
- Noter bien, dans la littérature MFC, les événements reliés aux éléments de l' interface graphique sont appelés des **notifications**.
- Une grande partie du travail d' un programme Windows consiste à traiter les messages et les notifications.
- On appelle aussi le modèle de Windows la programmation événementielle.

152

## Expédition et routage des messages (3)

- La syntaxe de ces macros:

DECLARE\_MESSAGE\_MAP

Placée dans la déclaration d'une classe dérivée de MFC dans le fichier .h

```
BEGIN_MESSAGE_MAP(CXtractCommGUIDlg, CDialog)
//{{AFX_MSG_MAP(CXtractCommGUIDlg)
ON_WM_PAINT()
:
:
ON_BN_CLICKED(ID_ACTION, OnAction)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Gestionnaires des événements

Placée dans la définition d'une classe dérivée de MFC dans le fichier .cpp

153

## Expédition et routage des messages (4)

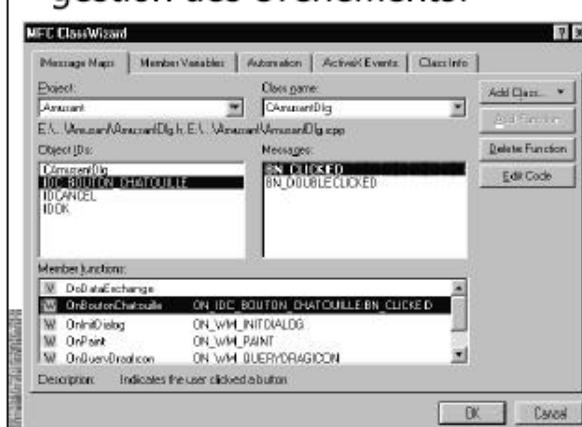
- Il existe un nombre élevé de macros pour la gestion des événements!

C'est pour cette raison que l'on utilise le ClassWizard.

Le ClassWizard présente les événements possibles des éléments graphiques d'un projet de programmation.

Ainsi, on n'a pas à se souvenir le nom de ces macros.

On active le ClassWizard par etat 8



154

## **Quelques classes de MFC (1)**

- Le MFC comprend:
  - Classes représentant les éléments de l'interface graphique.
    - Fenêtres, boutons, listes déroulantes, combo box, etc.
  - Classes utilitaires facilitant la programmation orientée objet.
    - Point (coordonnées de la souris, dimension des fenêtres, etc.)
    - Collections (semblables à STL mais moins *riche*)
    - Date, fichiers, etc.
    - Enveloppes (*wrappers*) des fonctions systèmes Win32.

155

## **Quelques classes de MFC (2)**

- Classe **cobject**:
  - Classe de base abstraite utilisée pour la dérivation des autres classes de MFC.
  - Elle est l'ancêtre de la plupart des classes de MFC.
  - Elle réalise le concept de sérialisation (persistance des objets).
  - Elle entrepose les informations nécessaires pour la détermination des classes (*runtime classe information*).
  - Elle permet le diagnostic des objets (*object dumps*).

10

156

## **Quelques classes de MFC (3)**

- Classe **CWinApp**:
  - Elle représente une application Windows.
  - Le AppWizard génère une classe dérivée de **CWinApp** pour chaque programme.
  - Puisqu'elle représente une application Windows, elle dispose aussi d'une queue de messages.
  - Puisqu'elle dispose d'une queue de messages, le AppWizard générera également les macros  
**DECLARE\_MESSAGE\_MAP**  
et  
**BEGIN\_MESSAGE\_MAP(, )**

11

157

## **Quelques classes de MFC (4)**

```
class CAmusantApp : public CWinApp
{
public:
    CAmusantApp();
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CAmusantApp)
public:
    virtual BOOL InitInstance();
//}}AFX_VIRTUAL

// Implementation
//{{AFX_MSG(CAmusantApp)
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

12

158

## **Quelques classes de MFC (5)**

- Il existe une fonction membre très importante dans la classe `CWinApp`:
- Il s'agit de `InitInstance()`.
- C'est dans cette fonction membre que l'on règle les paramètres de l'application:
  - Indiquer le type de bibliothèque MFC à utiliser.
  - Enregistrer/créer les clés appropriées dans le registre de Windows.
  - Enregistrer le patron document/vue.
  - Charger l'icône de l'application, etc.

13

159

## **Quelques classes de MFC (6)**

- Lorsqu'il s'agit d'une application basée sur un panneau de dialogue:
  - Créer le panneau de dialogue.
  - Donnant ainsi l'aspect visuel du programme.
- Donc, `CWinApp` n'a pas de représentation graphique !
- Il s'agit plutôt d'une classe de gestion (*house keeping*) nécessaire à tous les programmes MFC.

160

## Quelques classes de MFC (7)

```
BOOL CAmusantApp::InitInstance()
{
#ifdef _AFXDLL
    Enable3dControls();           // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();     // Call this when linking to MFC statically
#endif

    CAmusantDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is dismissed with Cancel
    }
    // Since the dialog has been closed, return FALSE so that we exit the
    // application, rather than start the application's message pump.
    return FALSE;
}
```

15

161

## Quelques classes de MFC (8)

- Classe **CWnd**:
  - Classe abstraite représentant une fenêtre.
  - En effet, dans le MFC, tous les contrôles (éléments d'interface graphique) sont des fenêtres !
  - Donc, **CWnd** est la classe de base de tous les contrôles.
  - Cette classe dispose d'un ensemble de fonctions membres pour réaliser la gestion et la manipulation d'une fenêtre.
  - Évidemment, **CWnd** est dérivée de **CObject**.

16

162

## **Quelques classes de MFC (9)**

- Classe **CDialog**:
  - Elle est dérivée de **CWnd**.
  - Classe représentant un panneau de dialogue.
  - Elle possède donc une représentation graphique.
  - De plus, elle possède la capacité de gérer le traçage des contrôles déposés.
  - C'est pour cette raison les applications basées sur un panneau de dialogue sont les plus simples à réaliser.

163

## **Application MDI (1)**

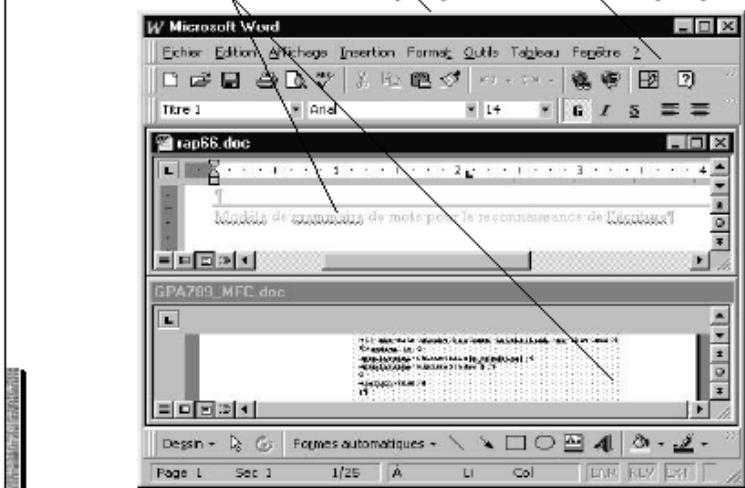
- MDI ? Multiple Document Interface
- Ce genre d'application utilise le concept de DOC/VIEW.
- DOC ? Entrepôt de données.
- VIEW ? Méthode de présentation des données.
- Donc, dans une application MDI, on peut avoir:
  - Plusieurs entrepôts de données
    - Type de données différentes (son, image, texte, etc.)
  - Plusieurs façons de présenter les données
    - Visionneur, graphisme, éditeur, etc.

18

164

## Application MDI (2)

Fenêtres enfants   Cadre principal   Menu dans le cadre principal



165

## Application MDI (3)

Le cadre principal est créé par CMainFrame

Le menu et les barres d'outils sont créés par CMainFrame

Les fenêtres enfants sont créées par CMDIChildWnd

Les classes principales d'une application MDI:

CMainFrame  
CMDIChildWnd  
CMenu  
CToolBar

Normalement, on crée nos propres classes MDI par dérivation.

20

166

## **Application MDI (4)**

- Dans une application MDI de MFC, la gestion des fenêtres est réalisée automatiquement.

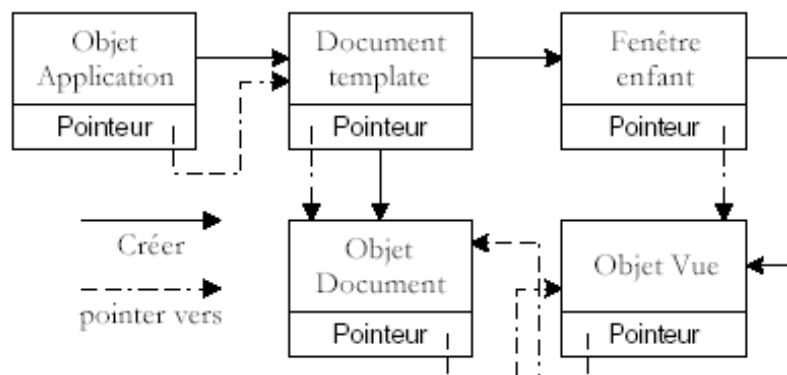


Autrement dit, les options de ce menu sont normalement réalisées par le code généré par le AppWizard.

167

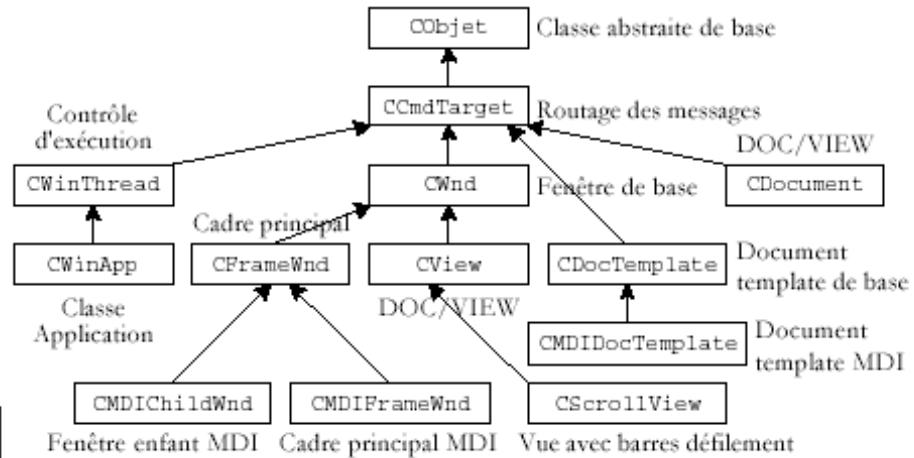
## Application MDI (5)

- Les relations qui existent dans le DOC/VIEW:



168

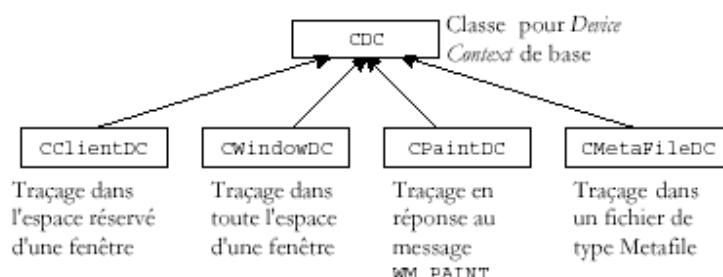
## Hiérarchie partielle des classes MDI



169

## Affichage graphique (1)

- Tous les affichages sont réalisés de manière graphique.
- On ne peut dessiner directement à l'écran.
- L'affichage est réalisé par le biais d'un objet de classe CDC.



24

170

## Affichage graphique (2)

- L'événement correspondant à la mise jour des éléments graphiques d'une application:
  - WM\_PAINT
- La macro normalement associée à cet événement est:
  - ON\_WM\_PAINT()
- La fonction membre représentée par cette macro est:
  - OnPaint()

171

## Affichage graphique (3)

- Voici à quoi ressemble l'affichage graphique dans une application MFC:

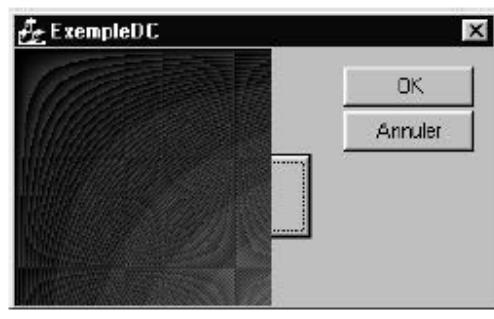
```
// 1) construire un client DC du panneau  
//     de dialogue (CDialog est dérivé de CWnd)  
CCClientDC DlgDC(this);  
  
// 2) créer un dessin sur le bureau !  
for (int x=0; x<150; x++)  
    for (int y=0; y<150; y++)  
        // Mettre des pixels en couleur  
        DlgDC.SetPixel(x, y, x * y);  
  
// 3) libérer le DC  
// Pas besoin pour les classes dérivées  
// de CDC !
```

26

172

## Affichage graphique (4)

- Affichage dans l'espace réservé d'une fenêtre (dans ce cas un panneau de dialogue):

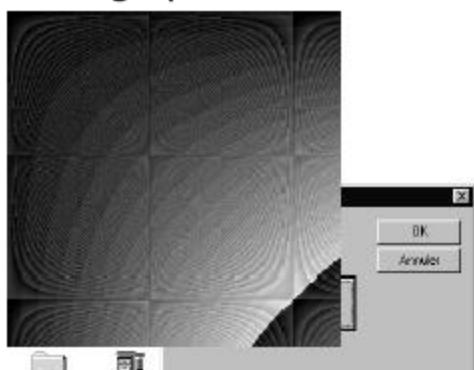


Le DC (Device Contexte) à utiliser est CClientDC.

173

## Affichage graphique (5)

- Affichage partout sur le bureau:



Le DC (Device Contexte) à utiliser est CDC.

28

174