
Langage de Programmation Orientée Objet : C++

Dr Khalid GABER

Passage du C au C++

Commentaire et Type construit

- ◆ Commentaires :

// ceci est un commentaire

- ◆ type énuméré, n-uplet :

enum *Couleur* {rouge,vert,orange};

struct *Compte* { int code, float solde};

...

Couleur feux ; // var. de type énuméré

Compte cpte; // var. de type n-uplet

Prototype de fonctions

- ◆ le type **void**

- ☞ indique qu'une fonction ne renvoie pas de valeur.

- ```
void f(int c) { };
```

- ◆ prototype de fonctions :

- ☞ signature de la fonction

- ☞ permet un contrôle

- ```
void echanger(int, int);
```

Surcharge (surdéfinition) des fonctions

- ◆ C++ permet de distinguer deux fonctions de mêmes noms sur le type de leurs arguments d'appels :

- ☞ `void init(float r) { ... };`

- ☞ `void init(char *chaine) { ... };`

- ◆ paramètre par défaut :

- ☞ `void test(int i, float x = 6.23) {... };`

- ☞ `test(5);`

- ☞ `test(9, 80.4);`

Passage de Paramètres par Copie

- ◆ Ce mode de passage consiste à ne déposer que l'adresse de l'argument dans la pile d'exécution au moment de l'appel.

☞ nécessite l'utilisation de pointeurs pour modifier un argument.

// version adresses explicites :

```
void echange( int *x, int *y)  
{ int z = *x;  
  *x = *y; *y = z; };
```

// utilisation :

```
d=4; e=6;  
echange( &d, &e);  
cout << d << " " << e << endl;
```

Passage de Paramètres par référence

- ♦ Manipulation implicite des adresses en rendant le mode de passage transparent au client.

// version adresses implicites :

```
void echange(int &x, int &y)
```

```
{ int z = x;
```

```
  x = y;  y = z;
```

```
}
```

```
d=4; e=6;
```

```
echange(d,e);
```

```
cout << d << " " << e << endl;
```



magique

Gestion de la mémoire dynamique

♦ Opérateurs **new** et **delete**

☞ new : allocation de mémoire sur le tas

float* r = new float; // var. élémentaire

float* r = new (float); // variante

float* t = new float[20]; // var. tableau

☞ delete : restitution de la mémoire

delete r; // var. élémentaire

delete [] t; // var. tableau

Sécurité des fonctions

♦ « **const** » permet de :

☞ protéger le résultat d'une fonction :

```
const int lire_code(Employe e);
```

☞ spécifier qu'un argument ne peut être que LU :

```
void maj_nom(Employe &e, const char *n);
```

Entrée- Sortie

- ◆ Flots (stream)

- ☞ entrée : cin (stdin en C)

- ☞ sortie : cout (stdout en C)

- ☞ erreur : cerr (stderr en C)

- ◆ Opérateur d'écriture <<

- cout << "Nom : " << emp.nom << endl;

- ◆ Opérateur de lecture >>

- cin >> emp.code;

- ◆ Les entrées/sorties sont formatées par défaut.

E/S : gestion des délimiteurs

- ♦ En C++ les séparateurs :

- ☞ ne sont pas traités en tant que caractère par l'opérateur ">>"

- ☞ mais servent à séparer 2 valeurs consécutives dans le fichier d'entrée standard

- ♦ `get(char &)` traite les séparateurs comme des caractères normaux :

```
#include <iostream>
```

```
void main()
```

```
{ char c;
```

```
    while (cin.get(c)) cout << c; }
```

E/S : fichier (1)

- ♦ librairie : `fstream.h`
- ♦ 3 types prédéfinies de fichier
 - ☞ `ifstream` lecture seule
 - ☞ `ofstream` écriture seule
 - ☞ `fstream` lecture/écriture
- ♦ toutes les primitives de manipulations de flot peuvent être appliqués à des flots auxquels on a rattaché un fichier.

E/S : fichier

◆ Association d'un fichier et d'un flot

☞ `open(char *nomDuFichier)`

- ◆ ouvre un fichier en l'associant à un flot déclaré précédemment

☞ `close()`

- ◆ ferme le fichier associé au flot
- ◆ coupe la liaison flot/fichier

E/S : fichier (Exemple)

```
#include <fstream.h>
#include <stdlib.h>

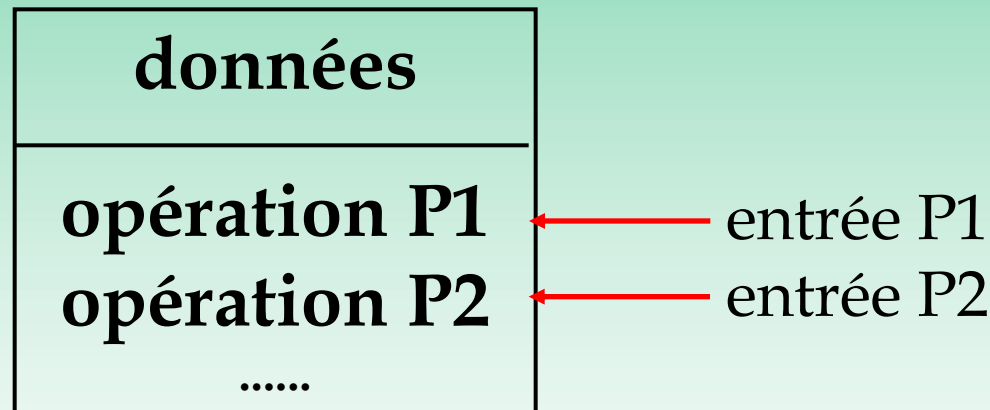
void main()
{ ifstream monFichier;

    monFichier.open("test.txt");
    while (monFichier >> car) nbre++;
    monFichier.close();

    ....
}
```

Encapsulation ?

- ◆ regroupement sous un même NOM :
 - ☞ des données
 - ☞ des opérations manipulant les données



- ◆ **une opération appartenant à une encapsulation ne peut manipuler que les données de cette encapsulation.**

Une encapsulation est

◆ Vue de l'extérieur

- ☞ une région mémoire
- ☞ une entité unique et indépendante
- ☞ une liste de points d'entrée

◆ Vue de l'intérieur

- ☞ des données structurées locales
- ☞ des opérations locales correspondant chacune à un point d'entrée et qui manipulent ces données

Prototypes et exemplaires

- ◆ un Prototype représente les objets des propriétés communes :
 - ☞ structurelle : attributs,
 - ☞ comportementales : méthodes.

Vocabulaire en C++

- ♦ **Classe** : prototype
- ♦ **Instances** : exemplaires créés à partir du prototype

Classe et Instance en C++

- ◆ classe :

```
class CCercle {  
    CPoint centre;           // attributs  
    CPositif rayon ;  
  
    void init ();           // méthodes  
    void dessinesToi();  
};
```

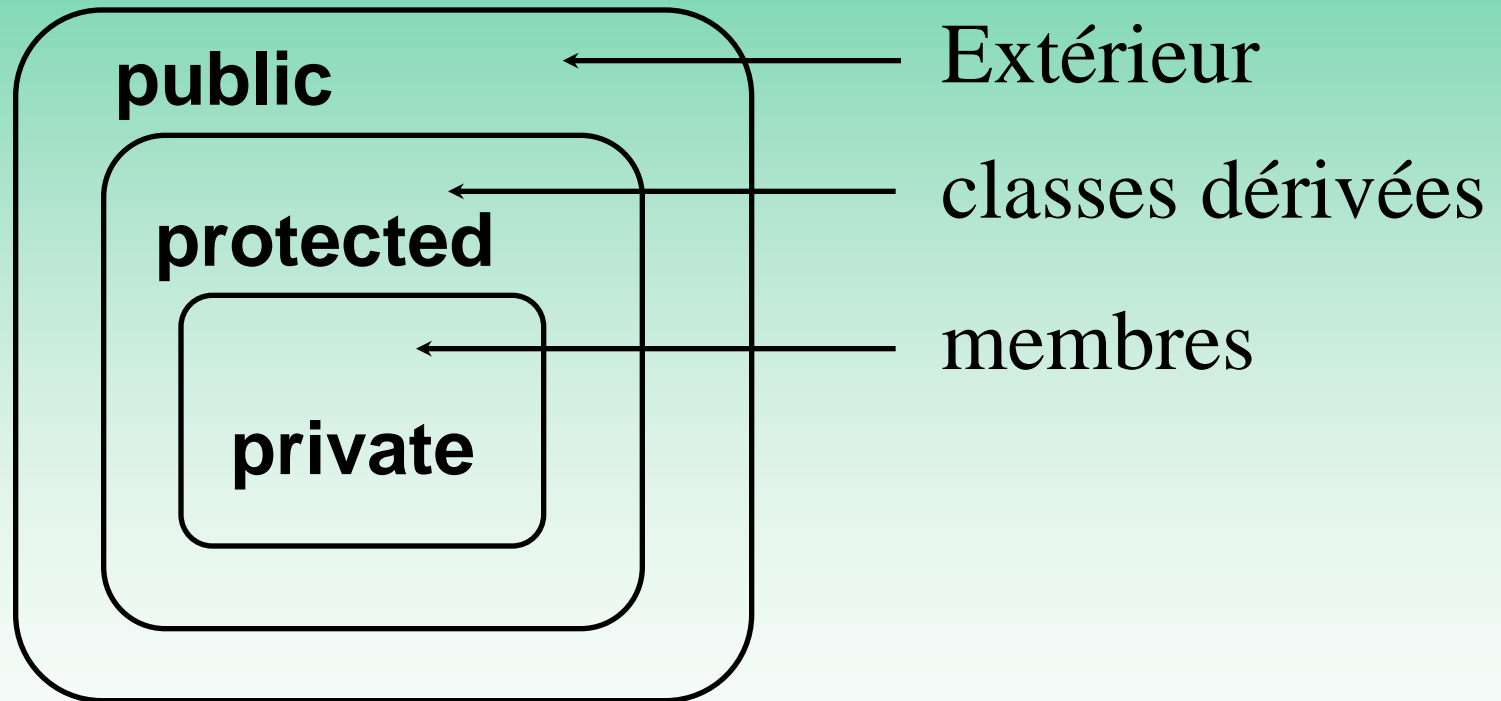
- ◆ instance :

```
CCercle C;
```

- ◆ initialisation :

```
C.init();
```

Structure d'une Classe



Déclaration d'une Classe

- ◆ fournir un nouveau type d'Objets.
- ◆ déclarer les **MEMBRES** de la classe :
 - ☞ **Attributs** : données-membres;
 - ☞ **Méthodes** : fonctions-membres.
- ◆ déclarer des fonctions "**amies**" :
 - ☞ n'appartenant pas à la classe;
 - ☞ se comportant comme des « méthodes ».

Méthodes

- ◆ lors de la déclaration de la classe, les méthodes seront :

- ☞ déclarée :

- signature : type et arguments.

- ☞ déclarée ET définie (**inline**) :

- description du corps de la fonction.

Méthodes

- ◆ Une Fonction membre peut :
 - ☞ recevoir tous les types d'arguments
 - ☞ retourner :
 - ◆ des expressions de tous types,
 - ◆ des pointeurs sur des objets de types divers,
 - ◆ des objets de type référence sur des types divers.

Méthodes inline

```
class Employe
{
    public:
        int getMatricule() { return matricule; } // fonction inline
        void setMatricule(int);                // descriptions séparées
        void voir();
    private:
        int matricule;
};
```


Définir une méthode

```
nom_classe :: nom_méthode(arguments) { ... }
```

```
void Employe::voir()  
{ cout << "matricule : " << matricule << endl; }
```

```
void Employe::setMatricule(int m)  
{ (*this).matricule = m; }
```

Message

◆ Définition d'une transmission de message :

☞ Receveur,

☞ Sélecteur de méthodes,

☞ Arguments.



```
instance_recept . sélecteur_de_méthode( args );
```

receveur

opérateur d'accès

Accès à un membre

♦ Opérateurs d'accès à membre (attributs ou méthodes) :

 . pour les instances de la classe

 -> pour les pointeurs sur des instances.

```
Employe martin;
```

```
Employe *pDupond;
```

```
martin.getMatricule();    // instance
```

```
pDupond->voir();        // pointeur sur une instance
```

Constructeur

- ◆ permet l'**initialisation** automatique d'une **instance de classe**, lors de sa déclaration.
- ◆ un **constructeur** est une fonction membre :
 - ☞ qui ne renvoi pas de valeur,
 - ☞ qui porte le nom de la classe à laquelle elle appartient.
- ◆ un constructeur peut être **surchargé** (comme toute fonction).

Constructeur (exemple)

Allocation dynamique d'un certain nombre de caractères au moment de la définition d'un objet de type Chaîne :

```
class Chaîne {  
    char* ch;          // champ privé  
public :  
    Chaîne();           // constructeur par défaut  
    Chaîne(int);        // constructeur Spécialisé  
    void saisie(char *);  
    void affiche() { cout << ch << endl; }  
    ~Chaîne();          // destructeur  
};  
  
Chaîne::Chaîne()  
{ ch = new char[100];  
  cout << « constructeur par défaut\n »; }
```

Constructeur (exemple)

```
Chaine::Chaine(int taille)
{ ch = new char[taille];
  cout <<« constructeur Spesialise\n »;
}

void Chaine::saisie(char *s)
{ strcpy(ch,s);
}

Chaine::~~Chaine()
{ delete [] ch;
  cout << « Destructeur ..... \n »;
}
```

Constructeur - suite

le constructeur est surchargé deux fois:

- ♦ une fois pour allouer par défaut 100 octets,
- ♦ une seconde fois pour allouer un nombre d'octets donnés

Constructeur (exemple)

```
main ()  
{  
    Chaine une;           // constr. par défaut  
    Chaine deux(20);      // constr. spécialisé  
    ...  
}
```


Constructeur de Copie

- ♦ **initialiser** une instance avec une autre instance de la même Classe.
- ♦ **indispensable** pour le passage de paramètre par Copie.

Chaine $s1 = s2$;

- ♦ copie standard bit à bit, de $s2$ dans $s1$
- ♦ si l'un des champs est dynamique, on risque de détruire 2 fois ce champ

Constructeur de Copie

```
class Chaine {  
private:  
    char *ch;    // champ privé  
public:  
    Chaine();    // constructeur par défaut  
    Chaine(int);    // constructeur spécialisé  
    Chaine( const Chaine &);    // constructeur de copie  
};  
Chaine::Chaine(const Chaine& e) {  
    this->ch = new char[strlen(e.ch) + 1];    // nouvel objet dynamique  
    strcpy(this->ch,e.ch);    // recopie de la valeur de e  
    Cout << « constructeur de copie .....\\n»  
}
```

Surcharges des Opérateurs et Fonctions Amies

Opérateurs surchargés

C++ permet de redéfinir les opérateurs :

➡ affectation

➡ arithmétiques

➡ comparaisons

➡ accès aux composantes d'un tableau

➡ etc

Opérateurs surchargés

```
class Pourcent {  
    int valeur;          // champ privé  
public:  
    Pourcent() { valeur = 0; }  
    Pourcent(int a) { valeur = a; if (valeur>100||valeur<0) valeur  
        =0; }  
    void affiche() { cout << a.valeur; }  
    // surcharge des opérateurs arithmétiques  
    Pourcent opertor+(int);          // opérateur BINAIRE  
    Pourcent opertor+(Pourcent);  
    void operator+=(int);  
    void operator++();                // opérateur UNAIRE  
};
```

Opérateurs surchargés

```
void Pourcent::operator++()  
{  
    valeur ++;  
    if (valeur > 100 || valeur<0) valeur=0;  
}  
  
void Pourcent::operator+=( int x )  
{  
    valeur= valeur + x;  
    if (valeur > 100 || valeur<0) valeur = 0;  
}
```

Opérateurs surchargés

```
Pourcent Pourcent::operator+( int x )
{ Pourcent p;
  p.valeur = this -> valeur +x
  if (p.valeur > 100 || p.valeur<0)
    p.valeur = 0;
  return p;
}
```

Opérateur d'affectation

- ◆ affecter un objet : lui attribuer quelque chose après sa définition
- ◆ deux possibilités :
 - ☞ affectation d'une partie de l'objet à l'aide d'une fonction membre ou d'un ami
 - ☞ affectation de tout l'objet par un autre objet

Déclaration de l'opérateur d'affectation

◆ **Classe & operator=(const classe&)**

```
class Pourcent {
```

```
...
```

```
    Pourcent & operator= (const Pourcent & );
```

```
...
```

```
};
```

Définition de l'opérateur d'affectation

```
Pourcent & Pourcent::operator=(const Pourcent
    & p)
{
    if (this == &a) return *this;
    valeur = p.valeur;
    return *this;
}
```

Opérateur []

```
class Chaîne {  
    char * ch;        // champ privé  
    public :  
        .....  
    char operator[](int);  
        .....  
};
```

Fonctions Amies

- ◆ fonction amie d'une classe est autorisée à accéder aux membres privés de cette classe

- ◆ déclaration d'une fonction amie

```
class A{  
    int x;  
    public:  
        A() { x = 0; };  
        friend void f(A);
```

Méthodes Amies

- ♦ une méthode d'une classe A doit pouvoir accéder aux parties privées d'une autre classe B
- ♦

```
class A { ... void methode_amie( B ); ... };
```
- ♦

```
class B{ ... friend void A::methode_amie(B); };
```

Classe Amie

- ◆ Toutes les méthodes de la classe B peuvent accéder aux parties privées de la classe A

```
class A {  
    private:  
    ...  
    public:  
        friend class B;  
    ...  
};
```

Surcharge d'opérateurs

```
class Pourcent {  
    int valeur;    // champ privé  
public:  
    .....  
    friend Pourcent operator+( int , const Pourcent &);  
};  
Pourcent operator+( int a, const Pourcent &p)  
{ Pourcent c;  
    c.valeur = p.valeur + a;  
    return c;}
```

Surcharge des opérateurs d'E/S

- ♦ Toutes les procédures d'entrées-sortie sont implémentées comme des opérations sur des fichiers
- ♦ C++ utilise deux classes prédéfinies : `istream` et `ostream`
- ♦ C++ permet de surcharger les deux opérateurs `<<` et `>>`

Surcharge de l'opérateur <<

- ♦ impossible de surcharger les opérateurs << et >> comme membre de la classe Pourcent, car ils sont membres respect., des classes prédéfinies ostream et istream
- ♦ il faut utiliser des fonctions amies :

```
class Pourcent{
```

```
.....
```

```
friend ostream& operator<<(ostream&, const Pourcent  
    &);
```

```
friend istream& operator>>(istream&, Pourcent &); }
```

Surcharge de l'opérateur <<

```
ostream& operator<< (ostream& flot, const  
    Pourcent & p)  
{  
    return flot << p.valeur << endl;  
}
```

♦ utilisation :

```
Pourcent toto;
```

```
cout << toto;
```

Surcharge de l'opérateur >>

```
istream& operator>> (istream& flot, Pourcent & p)
{
    flot >> p.valeur;
    return flot;
}
```

◆ utilisation :

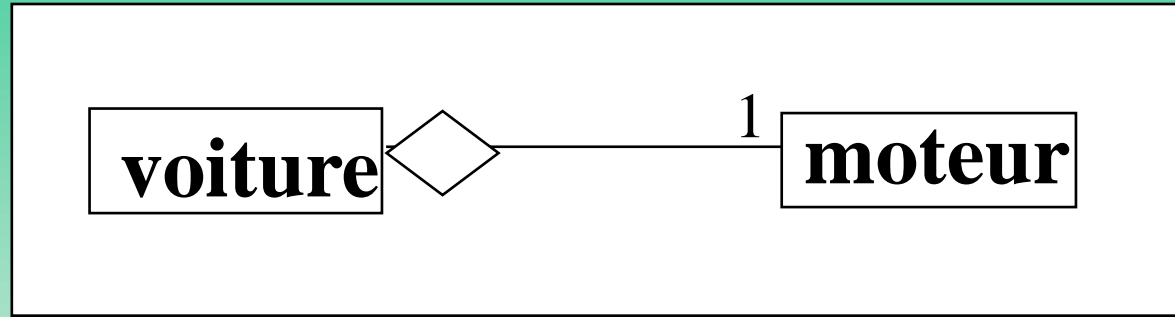
Pourcent toto;

cin >>toto;

Spécification par Composition

- ♦ La Composition est le processus qui consiste à créer des types nouveaux en combinant les anciens.
- ♦ La Composition est une relation A-UN
- ♦ la Spécialisation est une relation EST-UN
- ♦ La Composition crée des objets nouveaux
- ♦ la Spécialisation spécialise des objets existants.

Objet Composé



- ♦ la classe VOITURE possède une instance (champ) de la classe MOTEUR :

```
class Voiture
```

```
{ Moteur    monMoteur;
```

```
..... };
```

Constructeur d'un Objet Composé

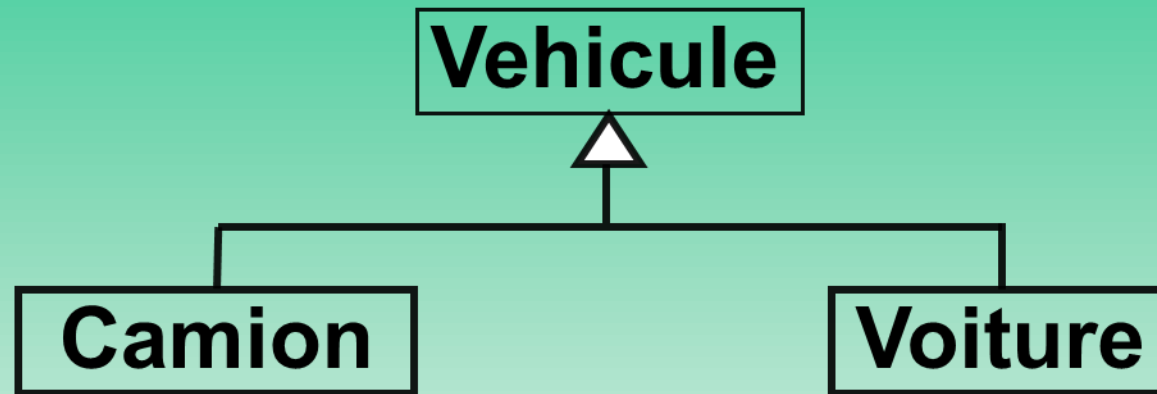
- ◆ le constructeur de la classe VOITURE doit transmettre convenablement les arguments au constructeur de la classe MOTEUR
- ◆ Au moment de la définition d'une instance de VOITURE, il y a dans l'ordre :
 - ☞ appel du constructeur de la classe MOTEUR,
 - ☞ appel du constructeur de la classe VOITURE

Constructeur d'un Objet Composé

```
Compose :: Compose (arg1, arg2, arg3)
         : champClasse1(arg1), champClasse2(arg)
         { ... utilisation de l' arg3 ...};
```

```
Voiture :: Voiture(int puissMoteur, int immat)
         : monMoteur(puissMoteur)
         {
           numero = immat; }
```

Déclaration d'une classe Dérivée



```
class Camion : public Vehicule { ... };
class Voiture : public Vehicule { ... };
```

↑
Classe

DERIVEE

↑
Classe
de BASE

Champs d'une classe Dérivée

- ◆ Une Classe DERIVEE contient sa classe de BASE
- ◆ Tous les champs de la classe de BASE deviennent des champs de la classe DERIVEE :
 - ☞ les champs publics et protégés peuvent être manipulés par la classe DERIVEE
 - ☞ les champs privés ne peuvent être accédés directement.

Méthodes de la classe Dérivée

- ◆ la classe Dérivée hérite des méthodes protégés et publiques de la classe de Base
- ◆ la classe Dérivée peut surcharger une méthode de la classe de Base
 - ☞ pour la redéfinir,
 - ☞ pour la masquer

Méthodes de la classe Dérivée

```
class Vehicule
{ int numero;
  float prix;
  char marque[20];
public:
  Vehicule();
  Vehicule(int,float,char*)
  void affiche();
  void setPrix(float);
};
```

```
class Voiture:public Vehicule
{ int type;
  public:
    Voiture();
    Voiture(int, float, char*, int)
    void setType(int);
    void affiche();
};
```

une instance de Voiture a accès à
Vehicule::affiche()
Vehicule::setPrix(5000)

Méthodes de la classe Dérivée

- ◆ l'opérateur :: permet l'accès aux méthodes des ancêtres :

☞ **ancêtre::sélecteur_de_méthode**

```
void Voiture::affiche()  
{ Vehicule::affiche(); // utilise la méthode de la  
                        //classe de Base  
  cout << " type : " << type << "\n"; // compléments de  
                                         //la méthode  
};
```

Constructeurs et Destructeurs

- ◆ lorsque la classe de BASE possède un *constructeur par défaut* celui-ci est appelé automatiquement avant l'appel du constructeur de la classe DERIVEE, pour initialiser les champs de la classe de BASE.

Voiture::Voiture()

{ // appel implicite de Vehicule::Vehicule()

type = 4;

};

Constructeurs et Destructeurs

- ◆ Un constructeur de la classe DERIVEE peut faire un appel explicite à un constructeur de la classe de BASE afin d'initialiser les champs hérités.

```
Voiture:: Voiture(int n, float p, char* m , int t)  
    : Vehicule(n,p,m) // appel explicite  
{  
    type = t;  
};
```

Constructeurs et Destructeurs

- ♦ **Les Destructeurs sont appelés dans l'ordre inverse**
 - ☞ **de DERIVEES vers la BASE.**
- ♦ **Les Constructeurs sont appelés dans l'ordre descendant des CLASSES**
 - ☞ **de BASE vers DERIVEES.**

POLYMORPHISME ET CLASSE ABSTRAITE

Compatibilité des affectations

- ♦ Il est possible d'affecter une instance d'une classe DERIVEE à une instance de la classe de BASE :

Compte c;

CoompteEpargne ce;

c = ce ; // correcte

- ♦ seuls les champs de la classe COMPTE sont recopiés dans c.

Compatibilité par héritage

- ◆ Une variable d'une classe Dérivée peut être utilisée partout où cela est possible pour la classe de BASE
- ◆ Le contraire n'est pas vrai : la classe de Base ne possède pas tous les champs de la classe Dérivée
 - ☞ `ce = c; // erreur`

Compatibilité par héritage

- ◆ les pointeurs d'une classe de Base sont compatibles avec ceux des classes Dérivées

```
Compte *pc;
```

```
Codevi *pce;
```

```
pc = pce; // correct
```

- ◆ l'affectation d'une instance de la classe Dérivée vers la classe de Base fait perdre de l'information (les champs spécifiques à la classe Dérivée sont perdues)
- ◆ l'affectation avec les pointeurs ne fait rien perdre

Généricité par Héritage
