Centrale-Lyon - Programmation Orientée Objet - Cours Info 2A (3/4)

Programmation Orientée Objet en C++ (2ème partie)

- Réutilisation de classes par héritage
- Modes d'héritage et constructeurs
- Fonctions et classes amies
- Opérateurs
- Entrées / sorties en C++

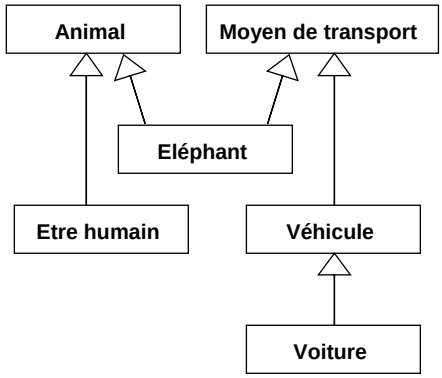
Version 2.2 – 8/11/2005

Réutilisation de classe par dérivation et héritage

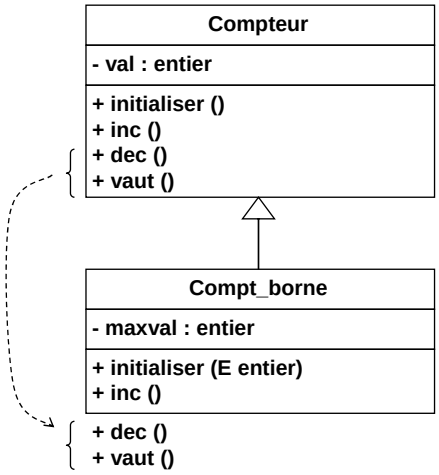
- Présentation
  - si on doit réaliser une classe voisine d'une classe existante, il est inutile de redéfinir complètement une nouvelle classe ;
    on crée une classe dérivée (ou héritière) de la première en indiquant seulement les différences par rapport à la classe de base
  - la classe dérivée hérite des membres de la classe de base ;
    un objet instance de la classe dérivée a accès à ses membres propres ainsi qu'à ceux de la classe de base (sauf limitations dues aux zones privées !)
  - la définition d'une classe dérivée d'une classe A représente la relation "est_un" (ou "sorte_de") entre B et A
  - C++ permet de définir une classe dérivée de plusieurs classes (héritage multiple)
  - l'utilisation répétée de l'héritage de classe permet de constituer une hiérarchie de classes représentant une relation de spécialisation entre ces classes

Exemples (Cf cours d'introduction [n°1/4])

- Hiérarchie de classes



- Compteur et Compt_borne



Cours Info 2A (3/4) C++ (2ème partie) 3

Modes d'héritage (1/2)

- Rappel sur les zones d'une classe :
  - un élément de la zone ***private*** ou ***protected*** est :
    - ➔ inacessible par une instance
    - ➔ accessible par une fonction membre (ou méthode)
  - un élément de la zone ***public*** est :
    - ➔ accessible par une instance
    - ➔ accessible par une fonction membre (ou méthode)

```
class Truc
 {
 private:
  int x;
 protected:
  int y;
 public:
  int z;
  void f();
  ...
 };

int main (void)
 {
 Truc u;
 u.x = 35;   // Erreur
 u.y = 45;   // Erreur
 u.z = 55;   // OK
 }
```

Cours Info 2A (3/4) C++ (2ème partie) 4

Modes d'héritage (2/2)

- Les constructeurs, le destructeur et l'opérateur = de la classe de base ne sont pas hérités par la classe dérivée
- 3 modes d'héritage sont possibles :
  - ***private*** (mode par défaut, si rien n'est spécifié !)
  - ***protected***
  - ***public*** (le plus courant)
- Accessibilité des attributs et méthodes
  - Héritage *public* :
    - → Zone ***public*** de la classe de base reste ***public*** dans la classe dérivée
    - → Zone ***protected*** reste ***protected*** dans la classe dérivée
    - → Zone **private** devient inacessible dans la classe dérivée
  - Héritage protected ou private:
    - → Très peu utilisé
    - → Voir dans le polycopié (§ 4.2.5)

Cours Info 2A (3/4) C++ (2ème partie) 5

Modifications par rapport à la classe de base

- Rappel: une classe dérivée peut introduire 3 types de différences par rapport à une classe de base :
  - ajout d'attributs
  - ajout de fonctions membres
  - modifications de fonctions membres (surchage)

```
class Base
{
   int j;
public:
   void g();
   void h();
};
```

```
class Derivee : public Base
{
   int j;
public:
   void f();
   void g();
}

int main (void)
{
   Base a;
   Derivee b;
   a.g();   // de la classe Base
   b.g();   // de la classe Derivee
   b.h();   // de la classe Base
}
```

Cours Info 2A (3/4) C++ (2ème partie) 6

Constructeurs, destructeur et héritage (1/2)

- Constructeurs :
  - Le constructeur de la classe dérivée appelle le constructeur de la classe de base avant d'effectuer son propre traitement (appel du constructeur sans argument qui doit donc exister !)
  - Sinon, on peut appeler explicitement le constructeur voulu de la classe de base :
    - ➔ après l'en-tête du constructeur de la classe dérivée
    - ➔ introduit par ':'
    - ➔ identifié par le nom de la classe (≠ de la composition !)

```
class Base
{
   int i,j;
public:
   Base (int x, int y) { i=x; j=y; }
   ...
};

class Derivee : public Base
{
   float z;
public:
   Derivee (int x, int y, float zz) :
                        Base (x,y)  { z=zz; }
   ...
}
```

Cours Info 2A (3/4) C++ (2ème partie) 7

Constructeurs, destructeur et héritage (2/2)

- Remarques :
  - Dans le cas d'héritage multiple, on peut appeler tous les constructeurs des classes de base en les séparant par des virgules
  - Dans le cas d'une hiérarchie de classes on peut avoir une série d'appels en cascade de constructeurs
  - Un constructeur peut être complexe ; il peut :
    - ➔ d'une part appeler les constructeurs des classes de base dont il dérive,
    - ➔ d'autre part appeler les constructeurs de membres instances d'autre classes (composition d'objets)
- Destructeur :
  - le destructeur de la classe de base est appelé automatiquement par le destructeur de la classe dérivée

Cours Info 2A (3/4) C++ (2ème partie) 8

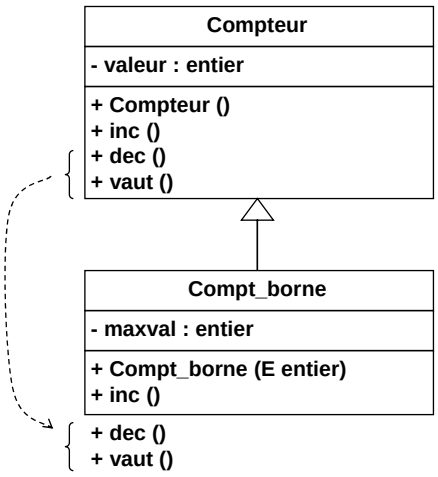
Accès aux méthodes de la classe de base

- Appel des méthodes de la classe de base
  - une fonction membre de la classe dérivée peut appeler une fonction d'une classe ancêtre (pourvue qu'elle ne soit pas *private,* ou que les modes d'héritages utilisés l'ont rendue inaccessible)
  - en cas d'ambiguïté entre noms de fonctions identiques situées à différents niveaux de la hiérarchie de classes, on utilise le nom complet (avec l'opérateur d'accès)
- En pratique:
  - si on appelle une méthode d'une classe, on regarde si elle est définie dans la classe et, si c'est le cas, on l'appelle;
  - sinon, on regarde dans la classe mère et ainsi de suite dans la hiérarchie pour trouver où elle est définie
  - si on ne la trouve nulle part, alors il y a erreur de compilation

Cours Info 2A (3/4) C++ (2ème partie) 9

Exemple : Compteur et Comp_borne - V 2 (1/2)

- Schéma UML :




```
class Compteur
{
private:
   int valeur;
public:
   Compteur ();
   void inc ();
   void dec ();
   int vaut ();
};

Compteur::Compteur ()
{   valeur = 0;  }

void Compteur::inc ()
{   ++valeur;  }

void Compteur::dec ()
{   if (valeur >0)  --valeur;  }

int Compteur::vaut ()
{   return valeur;  }
```

Cours Info 2A (3/4) C++ (2ème partie) 10

Exemple : Compteur et Comp_borne - V 2 (2/2)

- Remarques :
  - Appel explicite au constructeur parent (inutile dans ce cas puisque c'est le constructeur par défaut)
  - Pour accéder à la valeur du compteur, *Compt_borne* doit appeler la méthode vaut() car l'attribut *valeur* est inaccessible. (On aurait aussi pu utiliser une zone *protected* dans la définition de *Compteur)*

```
class Compt_borne : public Compteur
{
private:
   int maxval;
public:
   Compt_borne (int);
   void inc ();
};

Compt_borne::Compt_borne (int max) :
                          Compteur ()
{
   maxval=max;
}

void Compt_borne::inc ()
{
   if (vaut ()<maxval) Compteur::inc ();
}
```

Cours Info 2A (3/4) C++ (2ème partie) 11

Fonctions amies

- Fonction non membre d'une classe mais ayant accès à sa partie privée
  - ***f*** accède à la partie privée ***x*** de la variable ***t*** instance de la classe Truc
- A voir :
  - Polycopié § 3.7

```
class Truc
{
   int x;
public:
  Truc (int);
   void afficher () const;       // fonction membre
   friend int f (const Truc &);// fonction amie
};

void Truc::afficher () const
{   cout << x;  }

int f (const Truc & t)   // définition de la fonction
{   return t.x;  }          // externe f

int main (void)
{
   int y;
   Truc a(5);
   a.afficher ();       // appel fonction membre
   y = f(a);            // appel fonction externe
}
```

Cours Info 2A (3/4) C++ (2ème partie) 12

Classes et méthodes amies

- Classe amie :
  - les méthodes de la classe Truc ont accès à la partie privée de la classe Machin

```
class Machin
{
private:
   ...
public:
   ...
   friend Truc;        // la classe Truc est amie
   ...                 // de la classe Machin
};
```

- Méthode amie (ou fonction membre amie) :
  - la fonction f de la classe Truc a accès à la partie privée de la classe Machin

```
class Machin
{
private:
   ...
public:
   ...
   friend Truc::f (int);
   ...
};
```

Cours Info 2A (3/4) C++ (2ème partie) 13

Opérateurs (1/3)

- Notion d'opérateur et de surcharge

```
int a, b=4, c=2;
double ad, bd=4.3, cd=2.5;
a = b + c ;
ad = bd + cd ;
```

  - '+' correspond ici à 2 traitements différents selon qu'il est utilisé pour une addition d'entiers ou pour une addition de réels
  - '+' est surchargé
- Surcharge d'opérateurs en C++
  - on peut surcharger un opérateur portant sur des types de base pour qu'il s'applique sur des types créés (objets)
  - on doit spécifier les traitements à effectuer sur les opérandes

Cours Info 2A (3/4) C++ (2ème partie) 14

Opérateurs (2/3)

- Exemple de surcharge d'opérateur
  - création d'une classe pour représenter les nombres complexes
  - surcharge des opérateurs arithmétiques et de = pour traiter des instances de complexes

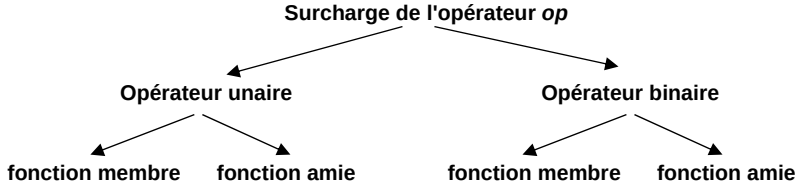
```
Complexe a, b(4.3, 3.4), c(2.5, 5.2) ;
a = b + c ;
```

- Règles pour la surcharge
  - surcharge des opérateurs déjà existants en C++
  - pas de changement de "gabarit" d'un opérateur :
    - → même arité,
    - → même priorité,
    - → même associativité
  - surcharge d'opérateurs uniquement pour des opérandes instances de classes

Cours Info 2A (3/4) C++ (2ème partie) 15

Opérateurs (3/3)

- Mise en œuvre de la surcharge d'un opérateur
  - définition d'une fonction ayant le nom "operator*op*" où *op* est l'opérateur à surcharger
    - → exemple : operator+ ()
  - la fonction "operator*op*" peut être soit :
    - → une méthode de la classe
    - → une fonction amie de la classe
- Statut d'une fonction de surcharge d'un opérateur



Cours Info 2A (3/4) C++ (2ème partie) 16

Opérateurs unaires (1/3)

- Opérateur unaire avec fonction membre
  - la fonction membre n'a pas d'arguments car l'opérande est implicitement l'instance de la classe concernée
  - **this** est l'attribut caché qui pointe sur l'instance en cours de traitement
  - ***this** est donc l'objet renvoyé
  - Note: ici c'est l'opérateur préfixé qui est surchargé :
    - ➔ ++c

```
class Complexe
{
   double reel, imaginaire ;
public:
   Complexe operator++ ();
  …
};

Complexe Complexe::operator++ ()
{
   reel++ ;
   imaginaire++ ;
   return *this ;
}
```

Cours Info 2A (3/4) C++ (2ème partie) 17

Opérateurs unaires (2/3)

- Opérateur unaire avec fonction amie
  - la fonction amie a un argument de type référence sur la classe qui correspond à l'opérande
  - Note: ici c'est l'opérateur préfixé qui est surchargé :
    - ➔ ++c

```
class Complexe
{
   double reel, imaginaire ;
public:
   friend Complexe operator++ (Complexe &);
   …
};

Complexe operator++ (Complexe & comp)
{
   comp.reel++ ;
   comp.imaginaire++ ;
   return comp ;
}
```

Cours Info 2A (3/4) C++ (2ème partie) 18

Opérateurs unaires (3/3)

- Opérateur unaire postfixé
  - utilisation d'un argument supplémentaire à la définition
  - cet argument n'apparaît pas dans l'utilisation de l'opérateur
  - Remarque : on retourne la valeur avant incrémentation !

```
class Complexe
{
   double reel, imaginaire ;
public:
   Complexe operator++ (int);
   …
};

Complexe Complexe::operator++ (int x)
{
   Complexe c = *this;
   reel++ ;
   imaginaire++ ;
   return c ;
}
```


Opérateurs binaires (1/2)

- Opérateur binaire avec une fonction membre
  - le premier opérande est implicite et il est du type de la classe concernée
  - le deuxième opérande est l'argument de la fonction membre

```
class Complexe
{
   double reel, imaginaire ;
public:
   Complexe operator+= (const Complexe &);
   …
};

Complexe Complexe::operator+= (
                        const Complexe & comp)
{
   reel += comp.reel ;
   imaginaire += comp.imaginaire ;
   return *this ;
}
```

Opérateurs binaires (2/2)

- Opérateur binaire avec une fonction amie
  - la fonction amie a 2 argument correspondant aux 2 opérandes

```
class Complexe
{
   double reel, imaginaire ;
public:
   friend Complexe operator+= (
            Complexe &, const Complexe &);
   …
};

Complexe operator+= (
            Complexe & a, const Complexe & b)
{
   a.reel += b.reel ;
   a.imaginaire += b.imaginaire ;
   return a ;
}
```


Choix fonction membre / fonction amie (1/2)

- Opérateur unaire :
  - on peut utiliser indifféremment une fonction membre ou amie
- Opérateur binaire :
  - la réalisation avec une fonction membre n'est possible que si le premier opérande est du type de la classe
  - pour une surcharge d'opérateur binaire avec le premier opérande de type quelconque et le deuxième opérande du type de la classe on doit utiliser une fonction amie de la classe
  - Exemple: réel + complexe

Choix fonction membre / fonction amie (2/2)

- Exemple : surcharge de l'opérateur + pour les complexes

```
class Complexe
{
   double reel, imaginaire ;
public:
   friend Complexe operator+ (double, const Complexe &);
   …
};
Complexe operator+ (double d, const Complexe & c)
{
   Complexe r ;
   r.reel = d + c.reel ;
   r.imaginaire = c.imaginaire ;
   return r ;
}
```

```
int main (void)
{
   Complexe a,b ;
   double d ;
   ...
   b = d + a ;
   ...
}
```

Cours Info 2A (3/4) C++ (2ème partie) 23

Surcharge de l'opérateur =

- Problème :
  - dans l'expression ***a=b***, l'opérateur = doit libérer les zones de données liées à ***a*** avant de lui affecter ***b***
  - l'opérateur = correspond à l'appel du destructeur suivi de l'appel du constructeur par copie
- Remarques :
  - l'opérateur = n'est pas utilisé dans le cas d'une définition de variable
  - ceci est nécessaire pour écrire des cascades d'affectation (a = b = c)

```
class Date
{
   int jour, an;
   char * mois;
public:
   Date (int, char *, int);
   ~Date ();
   Date & operator= (const Date &);
};
Date & Date::operator= (const Date & d)
{
   if (this != &d)      // test cas a=a
   {
      delete [] mois;
      mois = new char [strlen(d.mois)+1] ;
      strcpy (mois, d.mois);
      jour = d.jour ;
      an = d.an ;
   }
   return *this ;
}
```

Cours Info 2A (3/4) C++ (2ème partie) 24

Surcharge de l'opérateur []

- But :
  - permet d'avoir une écriture plus simple pour un accès aux composants d'une structure de données
- NB:
  - même si les données sont dans la zone publique, on a une écriture lourde si on ne surcharge pas []
    - ➔ x = v.pt[3]

```
class Vecteur_int
{
   int * pt;
public:
   Vecteur_int (int taille)
            { pt = new int[taille]; }
   int elem (int ind)
            { return pt[ind]; }
   int operator[] (int ind)
            { return pt[ind]; }
   ...
};

int main (void)
{
   Vecteur_int v(20);
   int x;
   …
   x = v.elem(3);
   x = v[3];
}
```

Cours Info 2A (3/4) C++ (2ème partie) 25

Entrées - sorties (1/6)

- Ecriture de variables de types de base
  - classe ***ostream*** (flot de sortie)
  - opérateur **<<** surchargé pour tous les types de base
- Opérateur binaire <<
  - opérande gauche : flot de sortie (instance de la classe ostream)
  - opérande droit : variable ou constante à sortir
    - ➔ flot << variable
  - << est associatif à gauche et renvoie le flot, d'où écriture en cascade possible :
    - ➔ flot << variable1 << variable2 << variable3

```
class ostream
{
   ...
public:
   ostream & operator<< (char *);
   ostream & operator<< (int);
   ostream & operator<< (long);
   ostream & operator<< (double);
   ostream & put (char);
   ...
};
```

Cours Info 2A (3/4) C++ (2ème partie) 26

Entrées - sorties (2/6)

- Instance prédéfinie de ostream : **cout**
  - **cout** est un objet prédéfini, instance de la classe ostream
  - cout désigne le flot de sortie standard : l'écran
  - l'expression
    cout<<variable
    a pour effet de sortir la variable sur l'écran
- Autres flots :
  - **cerr** et **clog** sont aussi 2 flots prédéfinis et dirigés par défaut sur l'écran

```
#include <iostream>
using namespace std;
...
int main (void)
{
   int x = 4;
   double y = 8.32;
   ...
   cout << "x = " << x << " y = " << y;
}

/*
  la sortie sur l'écran de ce programme
  est :
  x = 4 y = 8.32
*/
```

Cours Info 2A (3/4) C++ (2ème partie) 27

Entrées - sorties (3/6)

- Lecture de variables de types de base
  - classe ***istream*** (flot d'entrée)
  - opérateur **>>** surchargé pour tous les types de base

```
class istream
{
   ...
public:
   istream & operator>> (char *);
   istream & operator>> (int &);
   istream & operator>> (long &);
   istream & operator>> (double &);
   istream & get (char &);
   ...
};
```

- Opérateur binaire >>
  - opérande gauche : flot d'entrée (instance de la classe istream)
  - opérande droit : variable à lire
    ➔ flot >> variable
  - >> est associatif à gauche et renvoie le flot, d'où écriture en cascade possible :
    ➔ flot >> variable1 >> variable2 >> variable3

Cours Info 2A (3/4) C++ (2ème partie) 28

Entrées - sorties (4/6)

- Instance prédéfinie de istream : **cin**
  - **cin** est un objet prédéfini, instance de la classe istream
  - cin désigne le flot d'entrée standard : le clavier
  - l'expression
    cin>>variable
    a pour effet de lire la valeur tapée au clavier dans la variable

```
#include <iostream>
using namespace std;
...
int main (void)
{
   int x;
   double y;
   ...
   cout << "Entrer la valeur de x :";
   cin >> x;
   cout << "Entrer la valeur de y :";
   cin >> y;
   ...
}
```

- NB :
  - istream, ostream, cin, cout, cerr, clog sont déclarés dans ***iostream*** qu'il faut inclure dans les programmes
  - Il faut aussi ajouter ***using namespace std;***

Cours Info 2A (3/4) C++ (2ème partie) 29

Entrées - sorties (5/6)

- Ecriture de variables de type défini par l'utilisateur :
  - utilisation de cout, de la classe ostream et de << pour écrire des instances de classes définies par l'utilisateur
  - la classe ostream est prédéfinie et fermée : impossible de surcharger << dans ostream pour prendre en compte les nouveaux types
  - surcharge de << à l'aide d'une fonction amie de la classe concernée

```
#include <iostream>
using namespace std;
class Complexe
{
   double re, im ;
public:
   Complexe ();
   Complexe (double, double);
   friend ostream & operator<< (
            ostream &, const Complexe &);
   …
};
ostream & operator<< (
            ostream & f, const Complexe & c)
{
   return f << "(" << c.re << "," << c.im << ")" ;
}
```

Cours Info 2A (3/4) C++ (2ème partie) 30

Entrées - sorties (6/6)

- Exemple d'utilisation :
  - Après surcharge de l'opérateur << il est possible d'écrire toute sortie de la classe Complexe

```
…
int main (void)
{
   ofstream fic ("essai.txt");
   Complexe c1(0.5,1.5);

   cout << c1;
   clog << c1;
   fic << c1;   // écriture sur fichier disque
}

// affichage écran:
// (0.5,1.5)(0.5,1.5)
```

Cours Info 2A (3/4) C++ (2ème partie) 31