Programmation orientée objet

Conversion d'objet, explicit, mutable

Plan

- Conversion classe dérivée en classe de base (`upcasting`)
- Conversion de la classe de base en classe dérivée (`downcasting`)
- Conversion lors de la compilation
- Conversion lors de l'exécution
- explicit
- mutable

Classes Immeuble et Habitation

```
class Immeuble
{
public:
     void afficher();
};



class Habitation :  public Immeuble
{
  public:
     void afficher();

};
```

Conversion d'objets (dérivée => base)

- Un objet de la classe dérivée peut-être converti implicitement en un objet de la classe de base.

```
void tester(Immeuble unImmeuble)
{
   unImmeuble.afficher();
}


   Habitation monHabitation;
   tester(monHabitation);
```

Conversion d'objets (dérivée => base)

```
Immeuble   monImmeuble;   // classe de base
Habitation monHabitation; // classe dérivée
monImmeuble = monHabitation;

// monHabitation est convertit en objet Immeuble.
```

Conversion d'objets (dérivée => base)

- Un pointeur (ou une référence) sur un objet de la classe dérivée peut-être converti implicitement en un pointeur de la classe de base. (upcasting)

```
void tester( Immeuble * ptr)
{
   ptr->afficher();
}


   Habitation * ptrHabitation = new (MonHabitation);
   tester (PtrHabitation);
```

Conversion d'objets Polymorphisme

```
Immeuble* ptrImmeuble;
ptrImmeuble = new Habitation();
```

- `ptrImmeuble` pointe vers un espace mémoire de type Habitation.
- Mais à partir du pointeur `ptrImmeuble`, on ne peut accèder qu'à l'interface public de la classe Immeuble.
- Est-ce que l'on peut accéder à l'interface de la classe Habitation à partir du pointeur `ptrImmeuble` ?
- Solution: Fonction virtuelle (chapitre Polymorphisme)

Conversion d'objet (base => dérivée)

- Une conversion de la classe de base vers la classe dérivée n'est pas autorisée.
- Il faut créer un constructeur de la classe de base vers la classe dérivée.

```
void tester(Habitation uneHabitation)
{
   uneHabitation.Afficher();
}

   Immeuble monImmeuble;
   tester (monImmeuble);

// Il faut créer le constructeur Habitation(Immeuble)
```

Conversion de type Compilation ou Exécution

- À la compilation
  - static_cast<>
  - reinterpret_cast<>
- À l'exécution
  - dynamic_cast<>

Opérateur static_cast

- Changement de type à la compilation.
- Conversion standard (int en double, char en void *…..)
- Les anciens opérateurs de transtypage de C sont désuets.
- static_cast est un mot réservé.

Exemple 1 : static_cast

```
char lettre = 'A';

// En C:
unsigned int asciiLettre = (unsigned int) lettre;

// En C++:
unsigned int asciiLettre =
                    static_cast <unsigned int> (lettre);
```

Exemple 2 : static_cast

```
class Immeuble
{ /* … */ };
class Industrie : public Immeuble
{ /* … */ };

   // downcasting
   Immeuble * PtrImmeuble = new Industrie;
   Industrie * PtrIndustrie = static_cast<Industrie*> (PtrImmeuble);

   PtrIndustrie->Afficher(); // fonction afficher d'Industrie
```

Opérateur reinterpret_cast

- Permet de convertir n'importe quel type de pointeur en entier et vice-versa.

- Exemple :

```
Route * ptrRoute = new Route;
Immeuble * ptrUnImmeuble =
            reinterpret_cast <Immeuble*> (ptrRoute);
ptrUnImmeuble->afficher();
```

Run Time Type Information

- Permettant la liaison dynamique.
- Mécanisme particulier pour obtenir l'identification de type d'un objet ou d'une expression seulement à l'exécution.
- Polymorphisme.
- Opérateur: `typeid()` Pour connaître le type de l'objet lorsque l'on a un pointeur ou une référence.

Opérateur dynamic_cast

- Pour convertir un pointeur (référence) de la classe de base en pointeur (référence) de la classe dérivée (downcasting).
- Édition des liens dynamiques.

```
Classe_Dérivée* =
   dynamic_cast<Classe_Dérivée*> (Classe_Base*); // ptr

Classe_Dérivée& =
   dynamic_cast<Classe_Dérivée&> (Classe_Base&); // ref
```

Exemple dynamic_cast

```
vector<SourceEnergie*> tableau;

tableau.push_back(new CentraleHydro(
   "Beaulac", 200, 1200, Point3D(2,2), 31, "BaieJames") );

tableau.push_back(new CentraleNucleaire(
   "Burgey",  200, 1200, Point3D(2,2), 31, 0) );
```

Exemple dynamic_cast (suite)

```
CentraleNucleaire* ptrCentraleNucleaire;
CentraleHydro* ptrCentraleHydro;

for(unsigned int i = 0; i < tableau.size(); i++)
{
      if( typeid(*tableau[i]) == typeid(CentraleNucleaire) )
      {
               ptrCentraleNucleaire = dynamic_cast<CentraleNucleaire*>(tableau[i]);
               cout << "Nucleaire  Reacteurs : "
               << ptrCentraleNucleaire->GetNbReacteurs() << endl;
      }
      else if ( typeid(*Tableau[i]) == typeid(CentraleHydro) )
      {
               ptrCentraleHydro = dynamic_cast <CentraleHydro*> (tableau[i]);
               cout << "Hydro  Turbines : "
               << ptrCentraleHydro->GetNbTurbines() << endl;
      }
      else
               cout << " erreur " << endl;
}
```

Rappel

- Implicitement
  - Classe de Base = classe Dérivée
  - Pointeur classe de base = pointeur classe dérivée (upcasting)
- Par Constructeur
  - Classe dérivée = classe Base
- static_cast ou dynamic_cast
  - Pointeur classe dérivée = pointeur classe de base (downcasting)

Opérateur const_cast

- Permet de modifier ***const*** d'une expression.

```
Immeuble::Immeuble (const Immeuble& im)
{
    . . .
    const_cast <Immeuble&> (im).Superficie++; // opération valide

    // OU
    Immeuble& immC = const_cast <Immeuble&> (im);
    immC.Superficie++;
    . . .
}
```

Ajouts à la norme ANSI : Constructeur explicit

- Pour empêcher qu'un constructeur serve à des conversions de type implicitement, on le précède du mot clé explicit.

Constructeur explicit

```
class Toto {
   int attribut_;
   public:
        explicit Toto ( int Un=0)
                {attribut_ = Un;}
};
void Tester( Toto t) {
   cout << "Tester" << endl;
}
int main() {
   int UneValeur = 6;
   Tester(UneValeur);
   return 0;
}
```

```
error C2664: 'Tester' : cannot convert parameter 1 from 'int' to 'class Toto'
load resolution was ambiguous
```

Attributs membres mutables

- const_cast permet d'annuler la constance d'un objet.

Autre alternative

- Un attribut mutable peut être modifié dans une fonction membre constante.
- const_cast il faut le spécifier dans l'implémentation de la fonction membre.
- mutable est spécifié dans la définition de la classe.

Exemple mutable

```
class Toto
{
   mutable int attribut_;
   public:
        explicit Toto ( int Un=0){
               attribut_ = Un;
        }
        int getAttribut() const {
               return ++attribut_; // opération valide
        }
};
```