Conversion d'objet

« upcasting / downcasting », conversion statique / dynamique, explicit, prévenir le « slicing »

Conversion classe dérivée → base

« upcasting »

Dans un diagramme d'héritage on dessine la classe de base en haut

- Un objet, pointeur/référence d'une classe dérivée peut être converti implicitement en un d'une classe de base

```
Clock c1(true);
TravelClock c2(true, "Paris", +6);
TravelClock c3(true, "Vancouver", -3);

vector<Clock> horloges1 = { c1, c2, c3 };

vector<Clock*> horloges2 = { &c1, &c2, &c3 };

// TravelClock* x = horloges2[1];
```

Conversion implicite des objets: « slicing »

Conversion implicite des pointeurs (ou référence) : objets intacts

NON Seulement vers une classe de base du type statique

Les « cast » selon le type statique

Moins il y en a, mieux c'est

```
char lettre = 'A';

// En C:
int asciiLettre1 = (int)lettre;

// En C++:
int asciiLettre2 = static_cast<int>(lettre);

// En C++11:
int asciiLettre3 = int{lettre};

char lettre2 = narrow_cast<char>(asciiLettre3);

char lettre3 = narrow<char>(asciiLettre3);
```

Le plus permissif (dangereux)

Empêche la conversion de pointeur en int, et vérifie l'héritage (moins dangereux)

Vérifie qu'il n'y a pas de perte possible de données pour les type de base (sécuritaire)
Mais « slicing » pour les objets

(GSL) Dit que la restriction est voulue

(GSL) Vérifie à l'exécution que la valeur n'est pas changée

Conversion classe base → dérivée « downcasting »

- Jamais implicite
  - Objet: peut définir un constructeur de conversion
  - Pointeur/référence: doit faire un « cast »

```
TravelClock* x = static_cast<TravelClock*>(horloges[1]);
```

Vérifie uniquement que le type statique a un lien d'héritage

```
Clock c1(true);
TravelClock* y = static_cast<TravelClock*>(&c1);
```

Undefined behavior d'utiliser cet objet

```
TravelClock* z = dynamic_cast<TravelClock*>(horloges[1]);
```

Vérifie que le type dynamique est bien du type demandé (ou en dérive). L'objet doit être polymorphe.

C'est quoi « objet polymorphe »?

« cast » selon le type dynamique: dynamic_cast

- Vérifie à l'exécution le type dynamique
  - Utilise le RTTI (run-time type information)
    - Permet **typeid** selon le type dynamique
    - Se trouve par la vtable
  - L'objet doit être polymorphe pour avoir une vtable
- Pour convertir un pointeur ou référence

```
Classe_Dérivée* ptr_dérivée =
        dynamic_cast<Classe_Dérivée*>(ptr_base);
```

Pointeur: **nullptr** si échoue

```
Classe_Dérivée& ref_dérivée =
        dynamic_cast<Classe_Dérivée&>(ref_base);
```

Référence: exception **bad_cast** si échoue

Exemple de dynamic_cast

Secretary et Manager « sont des » Employee;
Secretary a une méthode getResponsability, Manager une méthode getBureau.

```
int main() {
   vector<Employee*> v;
   /* ... */
   int nbSecretary = 0;
   for (auto&& e : v) {
      if (auto secretary = dynamic_cast<Secretary*>(e)) {
         ++nbSecretary;
         cout << secretary->getResponsability() << endl;
      }
      else if (auto manager = dynamic_cast<Manager*>(e))
         cout << manager->getBureau() << endl;
      /* ... */
   }
}
```

Auto pour ne pas écrire Secretary* deux fois (C++11)

Déclaration dans le « if » (C++17)

Changement dynamique de type; nul/faux si échoue

Appel de fonction de la classe dérivée

Limiter les « cast », que faire?

- « upcast »
  - Utiliser la conversion implicite

Fonctionne aussi pour unique_ptr et shared_ptr

- « downcast »
  - Préférer une méthode virtuelle si possible
    - et va avec la conception

Aussi plus efficace que dynamic_cast

Exemple précédent avec méthodes virtuelles isSecretary et print dans Employee:

```
int nbSecretary = 0;
for (auto&& e : v) {
   if (e->isSecretary())
      ++nbSecretary;
   e->print();
}
```

Fait du sens seulement si la classe de base peut savoir ce qu'est une secrétaire

Normal que les classes sachent comment s'afficher

Restriction de conversion: constructeur explicit

```
class A {
public:
      A(int x) { }
};
class B {
public:
      explicit B(int x) { }
};
void TesterA(A t) { cout << "A\n"; }
void TesterB(B t) { cout << "B\n"; }
int main() {
      A a = 2;              // Ok
      TesterA(2);           // Ok
      B b1 = 2;             // Non car explicit
      B b2(2);              // Ok
      TesterB(2);           // Non car explicit
      TesterB(B(2));        // Ok
}
```

Dit qu'il faut appeler explicitement ce constructeur

error C2664: 'void TesterB(B)' : impossible de convertir l'argument 1 de 'int' en 'B'
note: Le constructeur pour class 'B' est déclaré 'explicit'

Restriction de conversion: prévenir le « slicing »

Core Guidelines C.67

- Une classe polymorphe devrait empêcher la copie

```
Base(const Base&) = delete;
Base& operator= (const Base&) = delete;
```

Pas de constructeur de copie ni = de copie dans classe de base, se propage aux classes dérivées

Core Guidelines C.130

- Comment copier si nécessaire?
  - Méthode virtuelle « clone() » pour copier selon le type dynamique
  - Retourne un pointeur, soit
    - unique_ptr<Base>
    - owner<Dérivée*>

GSL

Toutes les définitions d'une méthode virtuelle doivent avoir les mêmes types, sauf le type de retour qui peut être « covariant », mais pas possible avec unique_ptr

Exemple clone sans constructeur de copie: base

```
class D {
public:
    D(int x) : x_(x) {}
    D(const D&) = delete;
    D& operator= (const D&) = delete;
    virtual ~D() = default;

    virtual unique_ptr<D> clone() const {
        return make_unique<D>(CopySlice, *this);
    }
protected:
    struct CopySlice_t { explicit CopySlice_t() = default; };
    static constexpr CopySlice_t CopySlice{};
public: // mais protégé:
    D(CopySlice_t, const D& d) : x_(d.x_) {}
private:
    int x_;
};
```

Constructeurs quelconques

Sans copie

Destructeur virtuel car polymorphe

Contre tag implicite {}

Copie dans un nouveau pointeur du bon type

Tag pour protéger la copie non voulue

« constructeur de copie » protégé par le tag;
doit être public pour que make_unique puisse l'appeler

Exemple clone sans constructeur de copie: dérivée

```
class E : public D {
public:
    E(int x, int y) : D(x), y_(y) {}

    unique_ptr<D> clone() const override {
        return make_unique<E>(CopySlice, *this);
    }

public: // mais protégé:
    E(CopySlice_t, const E& e) : D(CopySlice, e), y_(e.y_) {}

private:
    int y_;
};
```

Chaque classe redéfinit la méthode pour copier dans le bon type

Et définit son « constructeur de copie » protégé en utilisant celui de la classe de base pour copier cette sous-partie de l'objet