

## PARTIE III

# Conversion et RTTI

Bruno Bachelet

Loïc Yon

# Implémentation d'une conversion (1/2)

- Possibilité 1: constructeur avec un seul argument

```
class Chaîne {  
    ...  
    Chaîne(const char * s);  
    ...  
};
```

- Fournit une conversion implicite `const char * → Chaîne`

```
void display(const Chaîne &);  
...  
display("aZeRtY"); ⇒ construction objet temporaire «Chaîne»
```

- Conversion implicite parfois non désirée

- `Vecteur::Vecteur(int n);` ⇒ conversion `int → Vecteur`

- Conversion explicite: mot-clé «`explicit`»

- `explicit Chaîne(const char * s);`

- `display("aZeRtY");` ⇒ erreur de compilation

- `display(Chaîne("aZeRtY"));` ⇒ obligation d'explicitement la conversion

# Implémentation d'une conversion (2/2)

- Possibilité 2: opérateur de conversion

```
class B {  
    ...  
    public: operator A() {  
        A a;  
        ... // Conversion de "this" dans "a"  
        return a;  
    }  
};
```

- Conversion implicite par défaut

- «`explicit`» possible depuis C++11

- Comment choisir entre les deux possibilités ?

- Constructeur: nécessite l'accès au code de la classe cible
  - Pas toujours possible de prévoir les conversions *a priori*
- Opérateur: nécessite l'accès au code de la classe source
  - N'est donc pas possible pour les types et les classes tierces

- Il existe plusieurs opérateurs de conversion
- *(type)*
  - ❑ Conversion de valeurs à partir d'opérateurs
- `static_cast`
  - ❑ Conversion de pointeurs/références avec vérification à la compilation
- `dynamic_cast`
  - ❑ Conversion de pointeurs/références avec vérification à l'exécution
- `const_cast`
  - ❑ Conversion portant uniquement sur l'aspect constant
- `reinterpret_cast`
  - ❑ Conversion de pointeurs sans vérification de type

## ■ Opérateur hérité du C

- ❑ Mais deux syntaxes possibles
- ❑ `c = (Chaine)s;`
- ❑ `c = Chaine(s);`

## ■ Conversion d'objets

- ❑ Effectuée à partir des opérateurs définis par le programmeur
- ❑ Aucun opérateur  $\Rightarrow$  conversion interdite

## ■ Conversion de types primitifs

- ❑ Opérateurs de conversion fournis par défaut

## ■ Conversion de pointeurs et références

- ❑ Toujours autorisée

## ■ Exemple

```
class Animal { ... virtual void manger(); ... };  
class Poisson : public Animal { ... void manger() override; ... };  
class Plante { ... };
```

```
Animal * girafe = new Animal();  
Animal * requin = new Poisson();  
Plante * sapin = new Plante();
```

```
Animal * animal; Poisson * poisson;
```

## ■ Conversions toujours autorisées

- ❑ `animal = (Animal *)sapin;` ⇒ (1) conversion fausse
- ❑ `poisson = (Poisson *)girafe;` ⇒ (2) conversion fausse
- ❑ `poisson = (Poisson *)requin;` ⇒ (3) conversion ok

## ■ Eviter l'utilisation de l'opérateur (*type*) pour des pointeurs/références

- ❑ Cas (1): détection possible à la compilation ⇒ opérateur «`static_cast`»
- ❑ Cas (2) & (3): détection à l'exécution ⇒ opérateur «`dynamic_cast`»

- Vérifie la conversion de pointeurs/références à la compilation
- Utilisé lors d'une conversion ascendante (*upcast*)
  - Conversion d'une classe fille vers une classe mère  
⇒ toujours possible
  - `animal = static_cast<Animal *>(poisson);`  
⇒ autorisée et valide
- Attention: conversion autorisée dès qu'il y a un lien d'héritage
  - `poisson = static_cast<Poisson *>(girafe);`  
⇒ autorisée mais incorrecte
  - Conversion descendante (mère → fille)  
⇒ vérification à l'exécution nécessaire  
⇒ utiliser l'opérateur «`dynamic_cast`»

- Conversion refusée s'il n'y a pas de lien d'héritage
  - ❑ `animal = static_cast<Animal *>(sapin);` ⇒ refusée
  - ❑ `int * pi = ...;`  
`float * pf = static_cast<float *>(pi);` ⇒ refusée
- Fonctionne de la même manière sur les références
- Conversion vers «`void *`» autorisée
  - ❑ `void * nawouak = static_cast<void *>(girafe);`
- Conversion depuis «`void *`» devrait être refusée
  - ❑ `animal = static_cast<Animal *>(nawouak);`
  - ❑ Peut être autorisée suivant le compilateur
  - ❑ Conseil: utiliser «`reinterpret_cast`» dans cette situation



# Opérateur *dynamic\_cast* (1/2)

---

- Vérifie la conversion de pointeurs/références à l'exécution
  - ❑ Même vérification que «*static\_cast*» effectuée à la compilation
  - ❑ Ne peut pas être employé pour convertir à partir de «*void \**»
- Utilisé lors d'une conversion descendante (*downcast*)
  - ❑ Conversion d'une classe mère vers une classe fille  
⇒ pas toujours possible
  - ❑ Une vérification à l'exécution est nécessaire
  - ❑ `poisson = dynamic_cast<Poisson *>(requin);` ⇒ autorisée
- A l'exécution, la conversion peut échouer
  - ❑ Conversion de pointeurs ⇒ pointeur nul retourné
  - ❑ Conversion de références ⇒ exception levée
  - ❑ `poisson = dynamic_cast<Poisson *>(girafe);` ⇒ refusée

- Conversion plus coûteuse que «*static\_cast*»  
⇒ à éviter donc quand «*static\_cast*» suffit

- Conversion par référence évite les recopies

```
Animal girafe;  
Poisson requin;
```

```
Animal & animal_1 = girafe;  
Animal & animal_2 = requin;
```

```
Poisson & poisson_1 = dynamic_cast<Poisson &>(animal_1);  
⇒ exception levée à l'exécution
```

```
Poisson & poisson_2 = dynamic_cast<Poisson &>(animal_2);  
⇒ conversion autorisée
```

- Permet de retirer l'aspect constant d'un objet
- N'a pas de signification sur une variable objet
  - ❑ `void f(const Chaine & c1) { ... Chaine c2 = c1; ... }`
  - ❑ La conversion ne pose aucun problème
  - ❑ Car une copie est effectuée et elle ne possède pas l'aspect constant
- Vraiment utile pour les références
  - ❑ `void f(const Chaine & c1)`  
    `{ ... Chaine & c2 = const_cast<Chaine &>(c1); ... }`
  - ❑ «`const_cast`» indispensable ici pour autoriser la conversion
- L'usage de cet opérateur est à éviter
  - ❑ Il permet de briser des règles fondamentales
  - ❑ Souvent, obligation d'utiliser «`const_cast`» ⇒ erreur de conception
    - Soit en imposant à tort la constance sur la variable
    - Soit en omettant des méthodes qui permettraient un accès constant
  - ❑ La solution «peut» être le modificateur «`mutable`»

- Conversion de pointeurs sans aucune vérification
  - Aucune instruction générée, simple changement de type du pointeur

- Exemple

```
struct ip_t { // Champs de bits
    unsigned int n1 : 8;
    unsigned int n2 : 8;
    unsigned int n3 : 8;
    unsigned int n4 : 8;
};

int main() {
    char * data = read_from_network();
    ip_t * ip = reinterpret_cast<ip_t *>(data);

    std::cout << ip->n1 << "." << ip->n2 << "."
               << ip->n3 << "." << ip->n4 << std::endl;
}
```

# Conversions: conclusion

	Chaine vers char *	Poisson * vers Animal *	Animal * vers Poisson *	Objet * vers void *	void * vers Objet *
(type)	<u>Oui</u>	Oui	Oui	Oui	Oui
static_cast	Oui	<u>Oui</u>	Oui	<u>Oui</u>	Ne devrait pas
dynamic_cast	Non applicable	Oui	<u>Oui</u> (après vérification)	Oui	Non applicable
reinterpret_cast	Non applicable	Oui	Oui	Oui	<u>Oui</u>

- *Run-Time Type Information*
- Très utile pour déterminer la classe réelle d'un objet à l'exécution
  - Celui-ci doit être pointé ou référencé
  - Pour que les liens d'héritage s'appliquent
- Même type de contrôle que «`dynamic_cast`»
- Mot-clé «`typeid`» retourne un objet de type «`type_info`»
  - `#include <typeinfo>`
- Exemple

```
Poisson p("Maurice",10,20,3);
Mammifere m("Rantanplan",5,9,17);
Animal * animal_1 = &p;
Animal * animal_2 = &m;

...

std::cout << typeid(*animal_1).name();
```

- L'objet «`type_info`» contient des informations sur le type
  - ❑ Nom du type: méthode «`name`»
  - ❑ Plus intéressant, opérateurs «`==`» et «`!=`»

- Permet de vérifier que deux objets sont du même type

```
if (typeid(*animal_1) == typeid(*animal_2))
    std::cout << "Ils sont de même type." << std::endl;
else
    std::cout << "Ils ne sont pas de même type." << std::endl;
```

- «`typeid`» peut s'appliquer sur un type

```
if (typeid(*animal_1) == typeid(Poisson))
    std::cout << "C'est un poisson.";
else std::cout << "Ce n'est pas un poisson.";
```

- Attention au piège: pensez à déréférencer les pointeurs

- ❑ Car pas de lien d'héritage entre les pointeurs
- ❑ Aucun lien entre «`Animal *`» et «`Poisson *`»

## ■ Exemple

```
Animal * ptr = new Poisson("Maurice",10,20,3);  
Animal & ref = *ptr;
```

## ■ Résultats de comparaisons de types

	<code>typeid(Animal)</code>	<code>typeid(Poisson)</code>	<code>typeid(Animal *)</code>	<code>typeid(Poisson *)</code>
<code>typeid(ptr)</code>	<code>!=</code>	<code>!=</code>	<code>==</code>	<code>!=</code>
<code>typeid(ref)</code>	<code>!=</code>	<code>==</code>	<code>!=</code>	<code>!=</code>
<code>typeid(*ptr)</code>	<code>!=</code>	<code>==</code>	<code>!=</code>	<code>!=</code>
<code>typeid(&amp;ref)</code>	<code>!=</code>	<code>!=</code>	<code>==</code>	<code>!=</code>