Langage C++ et programmation orientée objet

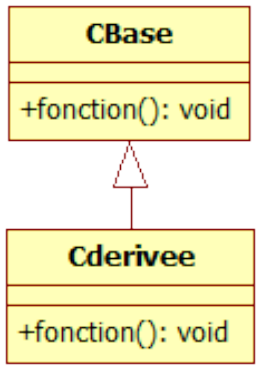
Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://claroline connect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

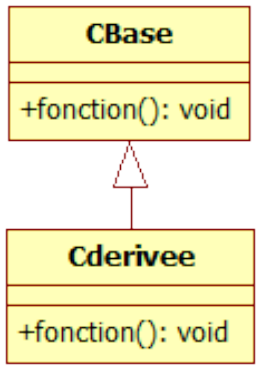
Introduction



Que se passe-t-il si on définit une méthode avec la même signature à la fois dans la classe de base et dans la classe dérivée (avec une accessibilité publique) ?

Cela compile ! **Il y a 2 méthodes de même nom associées à un objet de type Cderivee !**

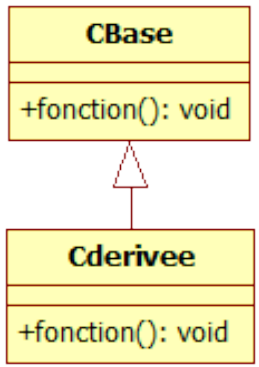
Introduction



Que se passe-t-il si on définit une méthode avec la même signature à la fois dans la classe de base et dans la classe dérivée (avec une accessibilité publique) ?

Cela compile ! Il y a 2 méthodes de même nom associées à un objet de type Cderivee !

Introduction



Que se passe-t-il si on définit une méthode avec la même signature à la fois dans la classe de base et dans la classe dérivée (avec une accessibilité publique) ?

Cela compile ! **Il y a 2 méthodes de même nom associées à un objet de type Cderivee** !

Phénomène de masquage

```
int main()
{
    Cderivee d ;
    d.Cderivee::fonction() ; // Appel forcé à la méthode de la classe Cderivee
    d.CBase::fonction() ;    // Appel forcé à la méthode de la classe CBase
    d.fonction() ;           // Quelle est la méthode appelée? CBase::fonction() ou
                             // Cderivee::fonction() ?
    return EXIT_SUCCESS ;
}
```

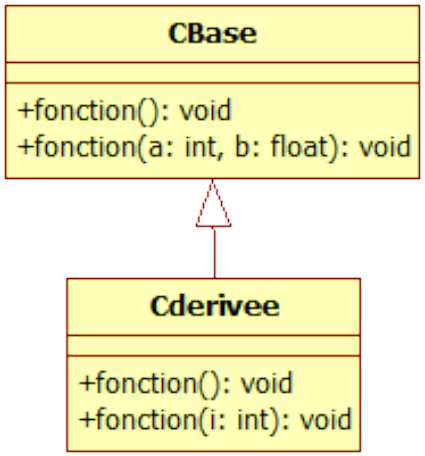
Règle *:Si une classe de base et une classe dérivée ont une méthode de même nom (et pas forcément de même signature), celle de la classe dérivée* ***masque*** *l'autre.*

Phénomène de masquage

```
int main()
{
    Cderivee d ;
    d.Cderivee::fonction() ; // Appel forcé à la méthode de la classe Cderivee
    d.CBase::fonction() ;    // Appel forcé à la méthode de la classe CBase
    d.fonction() ;           // Quelle est la méthode appelée? CBase::fonction() ou
                             // Cderivee::fonction() ?
    return EXIT_SUCCESS ;
}
```

Règle :*Si une classe de base et une classe dérivée ont une méthode de même nom (et pas forcément de même signature), celle de la classe dérivée **masque** l'autre.*

Phénomène de masquage




```
int main()
{
    Cderivee d ;
    d.fonction() ;        // la méthode appelée est Cderivee::fonction()
    d.fonction(4) ;       // la méthode appelée est Cderivee::fonction(int)
    d.fonction(2, 5.2f) ; // ERREUR de compilation!

    return EXIT_SUCCESS ;
}
```

Phénomène de masquage

Remarque : le phénomène de masquage s'applique aussi pour des attributs de même identifiant...

Fonctionnement automatique

```
class CBase
{   ...
  public :
    CBase(...) ; // un constructeur != du constructeur de recopie
    ~CBase() ;   // le destructeur
    ...
} ;
class CDerivee : public CBase
{   ...
  public :
    CDerivee(...) ; // un constructeur != du constructeur de recopie
    ~CDerivee() ;   // le destructeur
    ...
} ;
```

Si le constructeur de CBase **n'a pas d'argument**, alors le compilateur C++ va gérer automatiquement l'appel au constructeur de CBase lorsqu'on construit un objet de type CDerivee : *l'appel au constructeur de* CBase *précèdera celui de* CDerivee.

Fonctionnement automatique

```
class CBase
{   ...
  public :
    CBase(...) ; // un constructeur != du constructeur de recopie
    ~CBase() ;   // le destructeur
    ...
} ;
class CDerivee : public CBase
{   ...
  public :
    CDerivee(...) ; // un constructeur != du constructeur de recopie
    ~CDerivee() ;   // le destructeur
    ...
} ;
```

Si le constructeur de CBase **n'a pas d'argument**, alors le compilateur C++ va gérer automatiquement l'appel au constructeur de CBase lorsqu'on construit un objet de type CDerivee : *l'appel au constructeur de* CBase *précèdera celui de* CDerivee.

Fonctionnement automatique

```
class CBase
{   ...
  public :
    CBase(...) ; // un constructeur != du constructeur de recopie
    ~CBase() ;   // le destructeur
    ...
} ;
class CDerivee : public CBase
{   ...
  public :
    CDerivee(...) ; // un constructeur != du constructeur de recopie
    ~CDerivee() ;   // le destructeur
    ...
} ;
```

Même principe pour le destructeur : le compilateur C++ va gérer automatiquement l'appel au destructeur de CBase lorsqu'on détruit un objet de CDerivee : *l'appel au destructeur de* CBase *succèdera à celui de* CDerivee.

Cas d'un constructeur de CBase avec des arguments ?

```
class CPoint
{   float m_x, m_y ;
  public :
    CPoint(float, float) ; // un constructeur != du constructeur de recopie
    ~CPoint() ;            // le destructeur
    ...
} ;

class CPoint2DCoul : public CPoint
{   int m_couleur ;
  public :
    // un constructeur != du constructeur de recopie
    // ICI je suis obligé de faire un appel explicite à CPoint
    CPoint2DCoul(float x, float y, int col) : CPoint(x,y) { m_couleur = col ; }
    ~CPoint2DCoul() ;   // le destructeur
    ...
} ;
```

Remarque : Si je déclare CDerivee d(...) ; , je suis obligé d'utiliser un constructeur de la classe CDerivee.

Cas d'un constructeur de CBase avec des arguments ?

```
class CPoint
{   float m_x, m_y ;
  public :
    CPoint(float, float) ; // un constructeur != du constructeur de recopie
    ~CPoint() ;            // le destructeur
    ...
} ;

class CPoint2DCoul : public CPoint
{   int m_couleur ;
  public :
    // un constructeur != du constructeur de recopie
    // ICI je suis obligé de faire un appel explicite à CPoint
    CPoint2DCoul(float x, float y, int col) : CPoint(x,y) { m_couleur = col ; }
    ~CPoint2DCoul() ;   // le destructeur
    ...
} ;
```

Remarque : Si je déclare CDerivee d(...); , je suis obligé d'utiliser un constructeur de la classe CDerivee.

L'accessibilité protégée

Jusqu'à présent nous avons vu l'héritage public.

L'**héritage public conserve les règles d'accessibilité aux membres de la classe de base** : si un attribut/une méthode est public (resp. *private* ou **protected**) dans la classe de base, alors :

- il sera accessible (resp. *non-accessible* ou **non-accessible**) **à un utilisateur extérieur** de la classe dérivée ;
- il sera accessible (resp. *non-accessible* ou **accessible**) **à l'intérieur de la classe dérivée**.

Règle : *Un attribut/une méthode **protégé** est non-accessible depuis l'extérieur, mais il reste accessible dans une classe dérivée.*

L'accessibilité protégée

Jusqu'à présent nous avons vu l'héritage public.

L'**héritage public conserve les règles d'accessibilité aux membres de la classe de base** : si un attribut/une méthode est public (resp. *private* ou **protected**) dans la classe de base, alors :

- il sera accessible (resp. *non-accessible* ou **non-accessible**) **à un utilisateur extérieur** de la classe dérivée ;
- il sera accessible (resp. *non-accessible* ou **accessible**) **à l'intérieur de la classe dérivée**.

Règle : *Un attribut/une méthode **protégé** est non-accessible depuis l'extérieur, mais il reste accessible dans une classe dérivée.*

L'accessibilité protégée

Jusqu'à présent nous avons vu l'héritage public.

L'**héritage public conserve les règles d'accessibilité aux membres de la classe de base** : si un attribut/une méthode est public (resp. *private* ou **protected**) dans la classe de base, alors :

- il sera accessible (resp. *non-accessible* ou **non-accessible**) **à un utilisateur extérieur** de la classe dérivée ;
- il sera accessible (resp. *non-accessible* ou **accessible**) **à l'intérieur de la classe dérivée.**

Règle : *Un attribut/une méthode **protégé** est non-accessible depuis l'extérieur, mais il reste accessible dans une classe dérivée.*

L'accessibilité protégée

Jusqu'à présent nous avons vu l'héritage public.

L'**héritage public conserve les règles d'accessibilité aux membres de la classe de base** : si un attribut/une méthode est public (resp. *private* ou **protected**) dans la classe de base, alors :

- il sera accessible (resp. *non-accessible* ou **non-accessible**) **à un utilisateur extérieur** de la classe dérivée ;
- il sera accessible (resp. *non-accessible* ou **accessible**) **à l'intérieur de la classe dérivée**.

Règle : *Un attribut/une méthode **protégé** est non-accessible depuis l'extérieur, mais il reste accessible dans une classe dérivée.*

L'accessibilité protégée

Jusqu'à présent nous avons vu l'héritage public.

L'**héritage public conserve les règles d'accessibilité aux membres de la classe de base** : si un attribut/une méthode est public (resp. *private* ou **protected**) dans la classe de base, alors :

- il sera accessible (resp. *non-accessible* ou **non-accessible**) **à un utilisateur extérieur** de la classe dérivée ;
- il sera accessible (resp. *non-accessible* ou **accessible**) **à l'intérieur de la classe dérivée**.

Règle : *Un attribut/une méthode **protégé** est non-accessible depuis l'extérieur, mais il reste accessible dans une classe dérivée*.

Exemple

```
class CRadin
{  float m_argent ; // Attribut privé
  public :
    CRadin(...) ;   // un constructeur != du constructeur de recopie
    ~CRadin() ;     // le destructeur
    ...
} ;
class CGenereuxAvecFamille
{ protected :
    float m_argent ;            // Attribut protégé
  public :
    CGenereuxAvecFamille(...) ; // un constructeur != du constructeur de recopie
    ~CGenereuxAvecFamille() ;   // le destructeur
    ...
} ;
class CEnfant : public CRadin // l'héritié d'un radin ne peut pas accéder à son argent
{   ...
  public :
    // un constructeur != du constructeur de recopie
    CEnfant(...) : CRadin(...) { ... }
    ~CEnfant() ;   // le destructeur
    ...
} ;
```

Exemple

```
class CRadin
{  float m_argent ; // Attribut privé
  public :
    CRadin(...) ;   // un constructeur != du constructeur de recopie
    ~CRadin() ;     // le destructeur
    ...
} ;
class CGenereuxAvecFamille
{ protected :
    float m_argent ;            // Attribut protégé
  public :
    CGenereuxAvecFamille(...) ; // un constructeur != du constructeur de recopie
    ~CGenereuxAvecFamille() ;   // le destructeur
    ...
} ;
class CEnfant : public CGenereuxAvecFamille // l'héritié d'un parent généreux peut faire
{   ...                                     // ce qu'il veut avec son argent
  public :
    // un constructeur != du constructeur de recopie
    CEnfant(...) : CGenereuxAvecFamille(...) { ... }
    ~CEnfant() ;   // le destructeur
    void depenser(float montant) { m_argent -= montant ; }
    ...
} ;
```

Héritage et membres protégés de la classe de base

```
class CPoint2D
{
  protected :
    float m_x, m_y; // 2 membres protégés
  public :
    void CPoint2D(float x, float y): m_x(x), m_y(y) {}
    ...
} ;

class CPoint2DCoul : public CPoint2D // héritage public (conserve les règles d'accessibilité)
{ // membre privé de CPoint2DCoul
    int m_couleur ; // code représentant une couleur
  public :
    CPoint2DCoul(float x, float y, int c): m_x(x), m_y(y), m_couleur(c) {} // **FAUX** :
                                                    // attribut(valeur) n'est autorisé
                                                    // que si attribut appartient
                                                    // à la classe courante
    ...
} ;
```

Héritage et membres protégés de la classe de base

```
class CPoint2D
{
  protected :
    float m_x, m_y; // 2 membres protégés
  public :
    void CPoint2D(float x, float y): m_x(x), m_y(y) {}
    ...
} ;

class CPoint2DCoul : public CPoint2D // héritage public (conserve les règles d'accessibilité)
{ // membre privé de CPoint2DCoul
    int m_couleur ; // code représentant une couleur
  public :
    CPoint2DCoul(float x, float y, int c): CPoint2D(x,y), m_couleur(c) {} // **CORRECT**
    ...
} ;
```

Héritage et membres protégés de la classe de base

```
class CPoint2D
{
  protected :
    float m_x, m_y; // 2 membres protégés
  public :
    void CPoint2D(float, float) ;
    ...
} ;

class CPoint2DCoul : public CPoint2D // héritage public (conserve les règles d'accessibilité)
{ // membre privé de CPoint2DCoul
    int m_couleur ; // code représentant une couleur
  public :
    CPoint2DCoul(float, float, int) ;
    void affiche () const ;
    ...
} ;

void CPoint2DCoul::affiche() const
{   // A l'intérieur de la classe CPoint2DCoul, on peut accéder aux membres protégés de la
    // classe CPoint2D
    cout << "(" << m_x << ", " << m_y << ") ; code couleur = " << m_couleur << endl ;
}
```

Les déclarations d'amitié ne s'héritent pas !

```
class CPoint2D {
    friend class CBidon ; // Les méthodes de CBidon n'ont pas accès au champ m_couleur de
                          // CPoint2DCoul
  protected :
    float m_x, m_y; // 2 membres protégés
  public :
    void CPoint2D(float, float) ;
    ...
} ;
class CPoint2DCoul : public CPoint2D // héritage public (conserve les règles d'accessibilité)
{   friend class CBidon2 ; // Les méthodes de CBidon2 ont accès à tous les attributs
                           // (privés/protégés/publics) de CPoint2DCoul et aux
                           // attributs protégés/publics de CPoint2D
    int m_couleur ; // code représentant une couleur
  public :
    CPoint2DCoul(float, float, int) ;
    // L'opérateur << a bien accès aux attributs protégés/publics de CPoint2D
    friend std::ostream& operator <<(std::ostream&, const CPoint2DCoul&) ;
    ...
} ;
```

Rappel : une amie a les mêmes privilèges qu'une fonction membre

Les déclarations d'amitié ne s'héritent pas !

```
class CPoint2D {
    friend class CBidon ; // Les méthodes de CBidon n'ont pas accès au champ m_couleur de
                          // CPoint2DCoul
  protected :
    float m_x, m_y; // 2 membres protégés
  public :
    void CPoint2D(float, float) ;
    ...
} ;
class CPoint2DCoul : public CPoint2D // héritage public (conserve les règles d'accessibilité)
{   friend class CBidon2 ; // Les méthodes de CBidon2 ont accès à tous les attributs
                           // (privés/protégés/publics) de CPoint2DCoul et aux
                           // attributs protégés/publics de CPoint2D
    int m_couleur ; // code représentant une couleur
  public :
    CPoint2DCoul(float, float, int) ;
    // L'opérateur << a bien accès aux attributs protégés/publics de CPoint2D
    friend std::ostream& operator <<(std::ostream&, const CPoint2DCoul&) ;
    ...
} ;
```

Rappel : une amie a les mêmes privilèges qu'une fonction membre

3 mots-clés : public, protected et private

Du plus permissif au moins permissif :

- **public** : accessible à tous : utilisateurs extérieurs, classes dérivées et intérieur de la classe ;
- **protected** : accessible à l'intérieur de la classe et aux classes dérivées ;
- **private** : accessible seulement au sein de la classe.

Une classe Dérivée a accès aux membres publics et protégés de sa classe de Base.
Mais quelle est la règle d'accessibilité de ces membres hérités au sein de la classe dérivée ?

3 mots-clés : public, protected et private

Du plus permissif au moins permissif :

- **public** : accessible à tous : utilisateurs extérieurs, classes dérivées et intérieur de la classe ;
- **protected** : accessible à l'intérieur de la classe et aux classes dérivées ;
- **private** : accessible seulement au sein de la classe.

Une classe Dérivée a accès aux membres publics et protégés de sa classe de Base.
Mais quelle est la règle d'accessibilité de ces membres hérités au sein de la classe dérivée ?

3 mots-clés : public, protected et private

Du plus permissif au moins permissif :

- **public** : accessible à tous : utilisateurs extérieurs, classes dérivées et intérieur de la classe ;
- **protected** : accessible à l'intérieur de la classe et aux classes dérivées ;
- **private** : accessible seulement au sein de la classe.

Une classe Dérivée a accès aux membres publics et protégés de sa classe de Base.
Mais quelle est la règle d'accessibilité de ces membres hérités au sein de la classe dérivée ?

3 mots-clés : public, protected et private

Du plus permissif au moins permissif :

- **public** : accessible à tous : utilisateurs extérieurs, classes dérivées et intérieur de la classe ;
- **protected** : accessible à l'intérieur de la classe et aux classes dérivées ;
- **private** : accessible seulement au sein de la classe.

Une classe Dérivée a accès aux membres publics et protégés de sa classe de Base.

Mais quelle est la règle d'accessibilité de ces membres hérités au sein de la classe dérivée ?

3 mots-clés : public, protected et private

Du plus permissif au moins permissif :

- **public** : accessible à tous : utilisateurs extérieurs, classes dérivées et intérieur de la classe ;
- **protected** : accessible à l'intérieur de la classe et aux classes dérivées ;
- **private** : accessible seulement au sein de la classe.

Une classe Dérivée a accès aux membres publics et protégés de sa classe de Base.
Mais quelle est la règle d'accessibilité de ces membres hérités au sein de la classe dérivée ?

Accessibilité des membres hérités

- **Héritage public** : tous les membres hérités au sein de la classe Dérivée ont la même accessibilité que dans la classe de Base.
- **Héritage protégé** : un membre public dans la classe de Base devient un membre protégé dans la classe Dérivée : un utilisateur peut accéder directement au membre seulement s'il utilise la classe de Base ;
- **Héritage privé** : tous les membres hérités au sein de la classe Dérivée sont inaccessibles depuis l'extérieur et aux classes filles.

Accessibilité des membres hérités

- **Héritage public** : tous les membres hérités au sein de la classe Dérivée ont la même accessibilité que dans la classe de Base.
- **Héritage protégé** : un membre public dans la classe de Base devient un membre protégé dans la classe Dérivée : un utilisateur peut accéder directement au membre seulement s'il utilise la classe de Base ;
- **Héritage privé** : tous les membres hérités au sein de la classe Dérivée sont inaccessibles depuis l'extérieur et aux classes filles.

Accessibilité des membres hérités

- **Héritage public** : tous les membres hérités au sein de la classe Dérivée ont la même accessibilité que dans la classe de Base.
- **Héritage protégé** : un membre public dans la classe de Base devient un membre protégé dans la classe Dérivée : un utilisateur peut accéder directement au membre seulement s'il utilise la classe de Base ;
- **Héritage privé** : tous les membres hérités au sein de la classe Dérivée sont inaccessibles depuis l'extérieur et aux classes filles.

Accessibilités des membres hérités de la classe de Base au sein de la classe Dérivée

Statut initial	Héritage public	Héritage protected	Héritage private	Accès dans Dérivée
public	public	protected	private	Oui
protected	protected	protected	private	Oui
private	*private*	*private*	*private*	*Non*

Seule accessibilité depuis l'extérieur de la classe Dérivée.

Règle : *L'accessibilité d'un membre hérité de la classe de Base au sein de la classe dérivée est l'accessibilité la plus contrainte entre l'accessibilité initiale et le mode de dérivation.*

Accessibilités des membres hérités de la classe de Base au sein de la classe Dérivée

Statut initial	**Héritage public**	**Héritage protected**	**Héritage private**	Accès dans Dérivée
public	public	protected	private	Oui
protected	protected	protected	private	Oui
private	*private*	*private*	*private*	*Non*

Seule accessibilité depuis l'extérieur de la classe Dérivée.

Règle : *L'accessibilité d'un membre hérité de la classe de Base au sein de la classe dérivée est l'accessibilité la plus contrainte entre l'accessibilité initiale et le mode de dérivation.*

Accessibilités des membres hérités de la classe de Base au sein de la classe Dérivée

Statut initial	**Héritage public**	**Héritage protected**	**Héritage private**	Accès dans Dérivée
public	public	protected	private	Oui
protected	protected	protected	private	Oui
private	*private*	*private*	*private*	*Non*

Seule accessibilité depuis l'extérieur de la classe Dérivée.

Règle : *L'accessibilité d'un membre hérité de la classe de Base au sein de la classe dérivée est l'accessibilité la plus contrainte entre l'accessibilité initiale et le mode de dérivation*.

Intérêts des 3 types d'héritage ?

- **Héritage public** : conserver pour un utilisateur extérieur les opérations/fonctionnalités publiques de la classe mère : *spécialisation d'une classe mère* ;
- **Héritage privé** : encapsulation d'un objet du type de la classe de Base ; *restreindre les accès à l'interface et aux attributs de la classe de Base* ;
- **Héritage protégé** : encapsulation d'un objet du type de la classe de Base, qui reste accessible aux classes dérivées ; *restreindre les accès à l'interface et aux attributs de Base*.

Remarque : conceptuellement, il semble mieux d'utiliser la composition ou l'agrégation que l'héritage privé ou protégé.

Intérêts des 3 types d'héritage ?

- **Héritage public** : conserver pour un utilisateur extérieur les opérations/fonctionnalités publiques de la classe mère : *spécialisation d'une classe mère* ;
- **Héritage privé** : encapsulation d'un objet du type de la classe de Base ; *restreindre les accès à l'interface et aux attributs de la classe de Base* ;
- **Héritage protégé** : encapsulation d'un objet du type de la classe de Base, qui reste accessible aux classes dérivées ; *restreindre les accès à l'interface et aux attributs de Base*.

Remarque : conceptuellement, il semble mieux d'utiliser la composition ou l'agrégation que l'héritage privé ou protégé.

Intérêts des 3 types d'héritage ?

- **Héritage public** : conserver pour un utilisateur extérieur les opérations/fonctionnalités publiques de la classe mère : *spécialisation d'une classe mère* ;
- **Héritage privé** : encapsulation d'un objet du type de la classe de Base ; *restreindre les accès à l'interface et aux attributs de la classe de Base* ;
- **Héritage protégé** : encapsulation d'un objet du type de la classe de Base, qui reste accessible aux classes dérivées ; *restreindre les accès à l'interface et aux attributs de Base*.

Remarque : conceptuellement, il semble mieux d'utiliser la composition ou l'agrégation que l'héritage privé ou protégé.

Intérêts des 3 types d'héritage ?

- **Héritage public** : conserver pour un utilisateur extérieur les opérations/fonctionnalités publiques de la classe mère : *spécialisation d'une classe mère* ;
- **Héritage privé** : encapsulation d'un objet du type de la classe de Base ; *restreindre les accès à l'interface et aux attributs de la classe de Base* ;
- **Héritage protégé** : encapsulation d'un objet du type de la classe de Base, qui reste accessible aux classes dérivées ; *restreindre les accès à l'interface et aux attributs de Base*.

Remarque : conceptuellement, il semble mieux d'utiliser la composition ou l'agrégation que l'héritage privé ou protégé.

Plan

Introduction

Tout objet du type CDerivee **est un objet de type** CBase**, mais un objet de type** CBase **n'est pas un objet de type** CDerivee.

Par exemple, CPoint2DCoul est bien un CPoint2D, mais pas l'inverse.

Introduction

Tout objet du type CDerivee **est un objet de type** CBase**, mais un objet de type** CBase **n'est pas un objet de type** CDerivee. Par exemple, CPoint2DCoul est bien un CPoint2D, mais pas l'inverse.

Introduction

En C++, cette compatibilité se manifeste concrètement par l'existance de *conversions implicites* :

- d'un objet de type CDerivee en un objet de type CBase ;
- d'un pointeur (ou d'une référence) sur un type CDerivee en un pointeur (ou une référence) sur un type CBase.

Introduction

En C++, cette compatibilité se manifeste concrètement par l'existance de *conversions implicites* :

- d'un objet de type CDerivee en un objet de type CBase ;
- d'un pointeur (ou d'une référence) sur un type CDerivee en un pointeur (ou une référence) sur un type CBase.

Introduction

```
class CPoint2D
{
    ...
} ;
class CPoint2DCoul : public CPoint2D
{
    ...
} ;
int main()
{
    CPoint2D p_base ;
    CPoint2DCoul p_derivee ;
    p_base = p_derivee ;          // affectation légale (conversion implicite), qui utilise
                                  // l'opérateur = de la classe CPoint2D
    p_derivee = p_base ;          // ERREUR, affectation illégale!

    CPoint2D* ad_p_base = &p_base ;
    CPoint2DCoul* ad_p_derivee = &p_derivee ;
    ad_p_base = ad_p_derivee ;              // affectation légale (conversion implicite)
    ad_p_derivee = ad_p_base ;              // ERREUR, affectation illégale!

    return EXIT_SUCCESS ;
}
```

Conversion standard VS conversion par pointeur (ou référence)

La **conversion standard** consiste à **convertir un objet** de type CDerivee en un objet de type CBase. *Elle entraîne la perte des données attributs propres à la classe* CDerivee.

La **conversion par pointeur** (ou référence) consiste à **convertir une adresse** (ou une référence) d'un objet de type CDerivee en une adresse (ou une référence) d'un objet de type CBase. *Elle n'entraîne pas de perte des données attributs propres à la classe* CDerivee *(valeur adresse inchangée).*

Conversion standard VS conversion par pointeur (ou référence)

La **conversion standard** consiste à **convertir un objet** de type CDerivee en un objet de type CBase. *Elle entraîne la perte des données attributs propres à la classe* CDerivee.

La **conversion par pointeur** (ou référence) consiste à **convertir une adresse** (ou une référence) d'un objet de type CDerivee en une adresse (ou une référence) d'un objet de type CBase. *Elle n'entraîne pas de perte des données attributs propres à la classe* CDerivee *(valeur adresse inchangée)*.

Exemple

```
class CBase
{
    ... // définit son propre constructeur de recopie
} ;

class CDerivee : public CBase
{
    ...
  public :
    // Si le constructeur par recopie de CDerivee est défini (et public), aucun appel automatique
    // au constructeur par recopie de CBase ne sera mis en place (choix du C++)
    CDerivee(const CDerivee& d) : CBase(d) // OK grâce à la conversion implicite standard
    {
        ... // partie spécifique à CDerivee
    }
} ;
```

Exemple

```
class CBase
{
    ... // définit son propre Opérateur d'affectation
} ;

class CDerivee : public CBase
{
    int n ;
    ...
  public :
    CDerivee& operator = (const CDerivee& d)
    {
        if(this == &d) return *this ;
        CBase* mon_ad = this ; // OK grâce à la conversion implicite par pointeur
        CBase* autre_ad = &d ; // OK grâce à la conversion implicite par pointeur

        *mon_ad = *autre_ad ;  // Appel à l'opérateur d'affectation de la classe CBase

        // partie spécifique à CDerivee
        n = d.n ;
        ...
        return *this ;
    }
} ;
```

Plan

Cadre du polymorphisme en C++

Le **polymorphisme**, c'est au moins 1 méthode virtuelle dans la classe de base qui est redéfinie dans une classe dérivée, puis cette méthode est appelée depuis un type statique référence ou pointeur sur le type de base : le choix de la bonne méthode à appeler se fait alors en fonction du type effectif stocké dans la référence ou le pointeur sur le type de base.

Le polymorphisme n'est pas :

- la surcharge de fonctions membres dans une seule classe ou à la fois dans la classe de base et une classe dérivée.
- la seule redéfinition de fonctions membres (classe de base + classe dérivée), car il faut aussi manipuler les objets via des références ou des pointeurs stockés dans une référence ou un pointeur sur le type de base.

Cadre du polymorphisme en C++

Le **polymorphisme**, c'est au moins 1 méthode virtuelle dans la classe de base qui est redéfinie dans une classe dérivée, puis cette méthode est appelée depuis un type statique référence ou pointeur sur le type de base : le choix de la bonne méthode à appeler se fait alors en fonction du type effectif stocké dans la référence ou le pointeur sur le type de base.

Le polymorphisme n'est pas :

- la surcharge de fonctions membres dans une seule classe ou à la fois dans la classe de base et une classe dérivée.
- la seule redéfinition de fonctions membres (classe de base + classe dérivée), car il faut aussi manipuler les objets via des références ou des pointeurs stockés dans une référence ou un pointeur sur le type de base.

Cadre du polymorphisme en C++

Le **polymorphisme**, c'est au moins 1 méthode virtuelle dans la classe de base qui est redéfinie dans une classe dérivée, puis cette méthode est appelée depuis un type statique référence ou pointeur sur le type de base : le choix de la bonne méthode à appeler se fait alors en fonction du type effectif stocké dans la référence ou le pointeur sur le type de base.

Le polymorphisme n'est pas :

- la surcharge de fonctions membres dans une seule classe ou à la fois dans la classe de base et une classe dérivée.
- la seule redéfinition de fonctions membres (classe de base + classe dérivée), car il faut aussi manipuler les objets via des références ou des pointeurs stockés dans une référence ou un pointeur sur le type de base.

Introduction : typage "statique" et ses limites

```
class CAnimal {
    ...
  public :
    void avancer() ;
} ;

class CLievre : public CAnimal {
    ...
  public :
    void avancer() ; // Redefinition de avancer qui masque celle de CAnimal
} ;

class CTortue : public CAnimal {
    ...
  public :
    void avancer() ; // Redefinition de avancer qui masque celle de CAnimal
} ;
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle méthode avancer sera appelée?
   ...
}
```

Introduction : typage "statique" et ses limites

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle méthode avancer sera appelée?
   ...
}
```

Typage "statique" : *la méthode du type déclaré est celle appelée !* Ici on a un tableau de CAnimal*, ça sera donc la méthode avancer de CAnimal qui sera appelée, même si une adresse est de type "dynamique" (effectif) CLievre*.

Serait-il possible de forcer le compilateur à choisir la méthode avancer en fonction du type "dynamique" ?
Oui, avec le mécanisme des **fonctions virtuelles**.

Introduction : typage "statique" et ses limites

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle méthode avancer sera appelée?
   ...
}
```

Typage "statique" : *la méthode du type déclaré est celle appelée !* Ici on a un tableau de CAnimal*, ça sera donc la méthode avancer de CAnimal qui sera appelée, même si une adresse est de type "dynamique" (effectif) CLievre*.

Serait-il possible de forcer le compilateur à choisir la méthode avancer en fonction du type "dynamique" ?

Oui, avec le mécanisme des **fonctions virtuelles**.

Introduction : typage "statique" et ses limites

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle méthode avancer sera appelée?
   ...
}
```

Typage "statique" : *la méthode du type déclaré est celle appelée !* Ici on a un tableau de CAnimal*, ça sera donc la méthode avancer de CAnimal qui sera appelée, même si une adresse est de type "dynamique" (effectif) CLievre*.

Serait-il possible de forcer le compilateur à choisir la méthode avancer en fonction du type "dynamique" ?

Oui, avec le mécanisme des **fonctions virtuelles**

Introduction : typage "statique" et ses limites

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle méthode avancer sera appelée?
   ...
}
```

Typage "statique" : *la méthode du type déclaré est celle appelée !* Ici on a un tableau de CAnimal*, ça sera donc la méthode avancer de CAnimal qui sera appelée, même si une adresse est de type "dynamique" (effectif) CLievre*.

Serait-il possible de forcer le compilateur à choisir la méthode avancer en fonction du type "dynamique" ?
Oui, avec le mécanisme des **fonctions virtuelles**.

Fonction virtuelle et mot-clé virtual

```
class CAnimal {
    ...
  public :
    virtual void avancer() { ... } // Méthode déclarée virtuelle au sein de la classe de Base.
                                   // Le choix de la méthode avancer n'est plus réalisé
                                   // à la compilation, mais dynamiquement en fonction
                                   // du type exact de l'objet appelant (si appel via pointeur).
} ;
class CLievre : public CAnimal {
    ...
  public : // Inutile de respécifier le mot-clé virtual dans les classes dérivées!
    void avancer() { ... }

} ;
class CTortue : public CAnimal {
    ...
  public : // Inutile de respécifier le mot-clé virtual dans les classes dérivées!
    void avancer() { ... }
} ;
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0]=new CAnimal(...); ANIMAUX[1]=new CLievre(...); ANIMAUX[2]=new CTortue(...); ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // La bonne méthode avancer sera appelée!
   ...
```

Quand faut-il utiliser une fonction virtuelle ?

Une fonction f doit être définie en tant que fonction virtuelle si :

- f **est redéfinie dans les classes dérivées** avec exactement la même signature (même identifiant et mêmes paramètres formels) ;
- le type de retour de f dans les classes dérivées est soit le même, soit un pointeur ou une référence sur un type Dérivée du type utilisé dans la classe de Base (*valeurs de retour covariantes*) ;
- et **si f est appelée via des pointeurs ou références sur la classe de Base**.

Une classe qui définit au moins une fonction virtuelle est dite ***polymorphe*** (plusieurs comportements possibles). CAnimal est un type polymorphe.

Quand faut-il utiliser une fonction virtuelle ?

Une fonction f doit être définie en tant que fonction virtuelle si :

- f **est redéfinie dans les classes dérivées** avec exactement la même signature (même identifiant et mêmes paramètres formels) ;
- le type de retour de f dans les classes dérivées est soit le même, soit un pointeur ou une référence sur un type Dérivée du type utilisé dans la classe de Base (*valeurs de retour covariantes*) ;
- et **si f est appelée via des pointeurs ou références sur la classe de Base**.

Une classe qui définit au moins une fonction virtuelle est dite ***polymorphe*** (plusieurs comportements possibles). CAnimal est un type polymorphe.

Quand faut-il utiliser une fonction virtuelle ?

Une fonction f doit être définie en tant que fonction virtuelle si :

- f **est redéfinie dans les classes dérivées** avec exactement la même signature (même identifiant et mêmes paramètres formels) ;
- le type de retour de f dans les classes dérivées est soit le même, soit un pointeur ou une référence sur un type Dérivée du type utilisé dans la classe de Base (*valeurs de retour covariantes*) ;
- et **si f est appelée via des pointeurs ou références sur la classe de Base**.

Une classe qui définit au moins une fonction virtuelle est dite ***polymorphe*** (plusieurs comportements possibles). CAnimal est un type polymorphe.

Quand faut-il utiliser une fonction virtuelle ?

Une fonction f doit être définie en tant que fonction virtuelle si :

- f **est redéfinie dans les classes dérivées** avec exactement la même signature (même identifiant et mêmes paramètres formels) ;
- le type de retour de f dans les classes dérivées est soit le même, soit un pointeur ou une référence sur un type Dérivée du type utilisé dans la classe de Base (*valeurs de retour covariantes*) ;
- et **si f est appelée via des pointeurs ou références sur la classe de Base**.

Une classe qui définit au moins une fonction virtuelle est dite ***polymorphe*** (plusieurs comportements possibles). CAnimal est un type polymorphe.

Quand faut-il utiliser une fonction virtuelle ?

Une fonction f doit être définie en tant que fonction virtuelle si :

- f **est redéfinie dans les classes dérivées** avec exactement la même signature (même identifiant et mêmes paramètres formels) ;
- le type de retour de f dans les classes dérivées est soit le même, soit un pointeur ou une référence sur un type Dérivée du type utilisé dans la classe de Base (*valeurs de retour covariantes*) ;
- et **si f est appelée via des pointeurs ou références sur la classe de Base**.

Une classe qui définit au moins une fonction virtuelle est dite ***polymorphe*** (plusieurs comportements possibles). `CAnimal` est un type polymorphe.

Incovénients et restrictions

Le **temps d'exécution** est un peu plus long que pour une méthode non-virtuelle.

Seule une fonction membre non-statique peut être virtuelle.
Un constructeur ne peut pas être virtuel et ne doit pas appeler de méthode virtuelle.
Un destructeur *défini* doit être virtuel si sa classe possède une méthode virtuelle.

La **redéfinition d'une fonction virtuelle dans une classe dérivée n'est pas obligatoire.**

Incovénients et restrictions

Le **temps d'exécution** est un peu plus long que pour une méthode non-virtuelle.

Seule une fonction membre non-statique peut être virtuelle.
Un constructeur ne peut pas être virtuel et ne doit pas appeler de méthode virtuelle.
Un destructeur *défini* doit être virtuel si sa classe possède une méthode virtuelle.

La **redéfinition d'une fonction virtuelle dans une classe dérivée n'est pas obligatoire.**

Incovénients et restrictions

Le **temps d'exécution** est un peu plus long que pour une méthode non-virtuelle.

Seule une fonction membre non-statique peut être virtuelle. Un constructeur ne peut pas être virtuel et ne doit pas appeler de méthode virtuelle.

Un destructeur *défini* doit être virtuel si sa classe possède une méthode virtuelle.

La **redéfinition d'une fonction virtuelle dans une classe dérivée n'est pas obligatoire.**

Incovénients et restrictions

Le **temps d'exécution** est un peu plus long que pour une méthode non-virtuelle.

Seule une fonction membre non-statique peut être virtuelle.
Un constructeur ne peut pas être virtuel et ne doit pas appeler de méthode virtuelle.
Un destructeur *défini* doit être virtuel si sa classe possède une méthode virtuelle.

La redéfinition d'une fonction virtuelle dans une classe dérivée n'est pas obligatoire.

Incovénients et restrictions

Le **temps d'exécution** est un peu plus long que pour une méthode non-virtuelle.

Seule une fonction membre non-statique peut être virtuelle.
Un constructeur ne peut pas être virtuel et ne doit pas appeler de méthode virtuelle.
Un destructeur *défini* doit être virtuel si sa classe possède une méthode virtuelle.

La **redéfinition d'une fonction virtuelle dans une classe dérivée n'est pas obligatoire**.

Introduction

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle fonction avancer sera appelée?
   ...
}
```

On souhaite interdire l'instanciation d'animaux du type CAnimal. En effet, le type CAnimal est plus destiné à être un *modèle générique d'animal*, qu'à être un animal instanciable.
Dans ce cadre, on souhaite seulement déclarer la méthode avancer **dans la classe** CAnimal **et obliger la définition d'une méthode de même signature dans toute classe dérivée de** CAnimal.
Comment procéder ?

Introduction

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle fonction avancer sera appelée?
   ...
}
```

On souhaite interdire l'instanciation d'animaux du type CAnimal. En effet, le type CAnimal est plus destiné à être un *modèle générique d'animal*, qu'à être un animal instanciable.

Dans ce cadre, on souhaite seulement déclarer la méthode avancer dans la classe CAnimal et obliger la définition d'une méthode de même signature dans toute classe dérivée de CAnimal.

Comment procéder ?

Introduction

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle fonction avancer sera appelée?
   ...
}
```

On souhaite interdire l'instanciation d'animaux du type CAnimal. En effet, le type CAnimal est plus destiné à être un *modèle générique d'animal*, qu'à être un animal instanciable.
Dans ce cadre, on souhaite seulement déclarer la méthode avancer **dans la classe** CAnimal **et obliger la définition d'une méthode de même signature dans toute classe dérivée de** CAnimal.

Comment procéder ?

Introduction

```
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...);
      ...
   for(int i=0; i<20; i++) ANIMAUX[i]->avancer(); // Quelle fonction avancer sera appelée?
   ...
}
```

On souhaite interdire l'instanciation d'animaux du type CAnimal. En effet, le type CAnimal est plus destiné à être un *modèle générique d'animal*, qu'à être un animal instanciable.
Dans ce cadre, on souhaite seulement déclarer la méthode avancer **dans la classe** CAnimal **et obliger la définition d'une méthode de même signature dans toute classe dérivée de** CAnimal**.**
Comment procéder ?

Fonction virtuelle pure

```
class CAnimal {
    ...
  public :
    virtual void avancer() = 0 ; // méthode virtuelle pure (pas de définition, seulement
                                 // cette déclaration)
} ;

class CLievre : public CAnimal {
    ...
  public :
    void avancer() { ... }
} ;

class CTortue : public CAnimal {
    ...
  public :
    void avancer() { ... }
} ;
int main() {
   CAnimal* ANIMAUX[20] ; // Tableaux d'adresses de CAnimal
   ANIMAUX[0] = new CAnimal(...); // ERREUR
   ANIMAUX[1] = new CLievre(...); ANIMAUX[2] = new CTortue(...); ...
   ...
}
```

Classes abstraites

Rappel : Une classe abstraite est une classe qui ne peut pas être instanciée.

En C++, une classe est abstraite si elle contient au moins une méthode virtuelle pure, ou si tous ses contructeurs sont inaccessibles à l'extérieur.

Dans le slide précédent, CAnimal est une classe abstraite.

En C++, une classe dérivée (avec un constructeur public) qui hérite d'une classe abstraite est abstraite (*non-instanciable*) :

- si elle ne définit pas toutes les fonctions virtuelles pures de ses classes mères ;
- ou si une fonction vituelle pure y est déclarée.

Classes abstraites

Rappel : Une classe abstraite est une classe qui ne peut pas être instanciée.

En C++, une classe est abstraite si elle contient au moins une méthode virtuelle pure, ou si tous ses contructeurs sont inaccessibles à l'extérieur.

Dans le slide précédent, CAnimal est une classe abstraite.

En C++, une classe dérivée (avec un constructeur public) qui hérite d'une classe abstraite est abstraite (*non-instanciable*) :

- si elle ne définit pas toutes les fonctions virtuelles pures de ses classes mères ;
- ou si une fonction vituelle pure y est déclarée.

Classes abstraites

Rappel : Une classe abstraite est une classe qui ne peut pas être instanciée.

En C++, une classe est abstraite si elle contient au moins une méthode virtuelle pure, ou si tous ses contructeurs sont inaccessibles à l'extérieur.

Dans le slide précédent, CAnimal est une classe abstraite.

En C++, une classe dérivée (avec un constructeur public) qui hérite d'une classe abstraite est abstraite (*non-instanciable*) :

- si elle ne définit pas toutes les fonctions virtuelles pures de ses classes mères ;
- ou si une fonction vituelle pure y est déclarée.

Classes abstraites

Rappel : Une classe abstraite est une classe qui ne peut pas être instanciée.

En C++, une classe est abstraite si elle contient au moins une méthode virtuelle pure, ou si tous ses contructeurs sont inaccessibles à l'extérieur.

Dans le slide précédent, CAnimal est une classe abstraite.

En C++, une classe dérivée (avec un constructeur public) qui hérite d'une classe abstraite est abstraite (*non-instanciable*) :

- si elle ne définit pas toutes les fonctions virtuelles pures de ses classes mères ;
- ou si une fonction vituelle pure y est déclarée.

Classes abstraites

Rappel : Une classe abstraite est une classe qui ne peut pas être instanciée.

En C++, une classe est abstraite si elle contient au moins une méthode virtuelle pure, ou si tous ses contructeurs sont inaccessibles à l'extérieur.

Dans le slide précédent, CAnimal est une classe abstraite.

En C++, une classe dérivée (avec un constructeur public) qui hérite d'une classe abstraite est abstraite (*non-instanciable*) :

- si elle ne définit pas toutes les fonctions virtuelles pures de ses classes mères ;
- ou si une fonction vituelle pure y est déclarée.

dynamic_cast< adresse ou référence type dérivé> (adresse ou référence type Base)

```
class CAnimal {
    ...
  public :
    virtual void avancer() { ... }
} ;
class CLievre : public CAnimal {
    ...
  public :
    void avancer() { ... }
} ;
class CTortue : public CAnimal {
    ...
  public :
    void avancer() { ... }
} ;
int main() {
   CLievre l ;
   CAnimal* ad_an = &l;
   CLievre* ad_l = dynamic_cast<CLievre*>( ad_an ) ; // OK car type dynamique de ad_an = CLievre*
   CTortue* ad_t = dynamic_cast<CTortue*>( ad_an ) ; // ERREUR dynamique !
   ...
}
```

Quand et comment le cast dynamique réussit ou échoue ?

Le cast dynamique aboutit seulement si le type dynamique, au moment de l'exécution du cast, est soit le type demandé, soit un type descendant du type demandé dans un contexte de polymorphisme.

Lorsque le cast dynamique n'aboutit pas :

- il fournit le pointeur NULL s'il s'agit d'une conversion explicite de pointeur ;
- il déclenche une exception bad_cast s'il s'agit d'une conversion explicite de référence.

Quand et comment le cast dynamique réussit ou échoue ?

Le cast dynamique aboutit seulement si le type dynamique, au moment de l'exécution du cast, est soit le type demandé, soit un type descendant du type demandé dans un contexte de polymorphisme.

Lorsque le cast dynamique n'aboutit pas :

- il fournit le pointeur NULL s'il s'agit d'une conversion explicite de pointeur ;
- il déclenche une exception bad_cast s'il s'agit d'une conversion explicite de référence.

Quand et comment le cast dynamique réussit ou échoue ?

Le cast dynamique aboutit seulement si le type dynamique, au moment de l'exécution du cast, est soit le type demandé, soit un type descendant du type demandé dans un contexte de polymorphisme.

Lorsque le cast dynamique n'aboutit pas :

- il fournit le pointeur NULL s'il s'agit d'une conversion explicite de pointeur ;
- il déclenche une exception `bad_cast` s'il s'agit d'une conversion explicite de référence.