Programmation Orientée Objet en **C++** (

Emanuel Aldea <emanuel.aldea@u-psud.fr>
http://hebergement.u-psud.fr/emi

M1-E3A

Troisième chapitre

L’héritage

L'héritage

Déclaration :

- La déclaration d'une classe peut faire intervenir une clause d'héritage d'une ou plusieurs autres classes :

```
class Enfant : public Parent1 , protected Parent2
{
  char c;
  ...
};
```

L'héritage

Déclaration :

- La déclaration d'une classe peut faire intervenir une clause d'héritage d'une ou plusieurs autres classes :

```
class Enfant : public Parent1 , protected Parent2
{
  char c;
  ...
};
```

- La classe enfant hérite tous les membres des classes parents. La syntaxe d'accès de l'intérieur ou de l'extérieur reste la même

L'héritage

Déclaration :

- La déclaration d'une classe peut faire intervenir une clause d'héritage d'une ou plusieurs autres classes :

```
class Enfant : public Parent1 , protected Parent2
{
  char c;
  ...
};
```

- La classe enfant hérite tous les membres des classes parents. La syntaxe d'accès de l'intérieur ou de l'extérieur reste la même
- Il est interdit d'hériter de soi-même

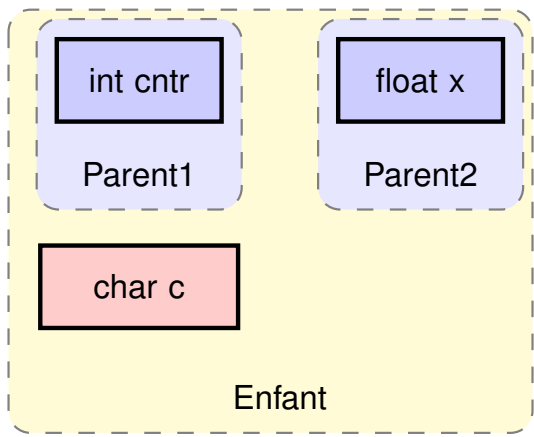
L'héritage

Déclaration :

- La déclaration d'une classe peut faire intervenir une clause d'héritage d'une ou plusieurs autres classes :

```
class Enfant : public Parent1 , protected Parent2
{
  char c;
  ...
};
```

- La classe enfant hérite tous les membres des classes parents. La syntaxe d'accès de l'intérieur ou de l'extérieur reste la même
- Il est interdit d'hériter de soi-même



L'intérêt de l'héritage

Objectifs et idée générale

- **étendre** la fonctionnalité des classes déjà existantes : une instance de la classe `Enfant` est au même temps un objet de type `Parent1` et un objet de type `Parent2`, en ayant en plus d'autres membres et méthodes

L'intérêt de l'héritage

Objectifs et idée générale

- **étendre** la fonctionnalité des classes déjà existantes : une instance de la classe `Enfant` est au même temps un objet de type `Parent1` et un objet de type `Parent2`, en ayant en plus d'autres membres et méthodes
- **réutilisation du code** : la classe dérivée `Enfant` hérite tous les membres et méthodes des classes de base

L'intérêt de l'héritage

Objectifs et idée générale

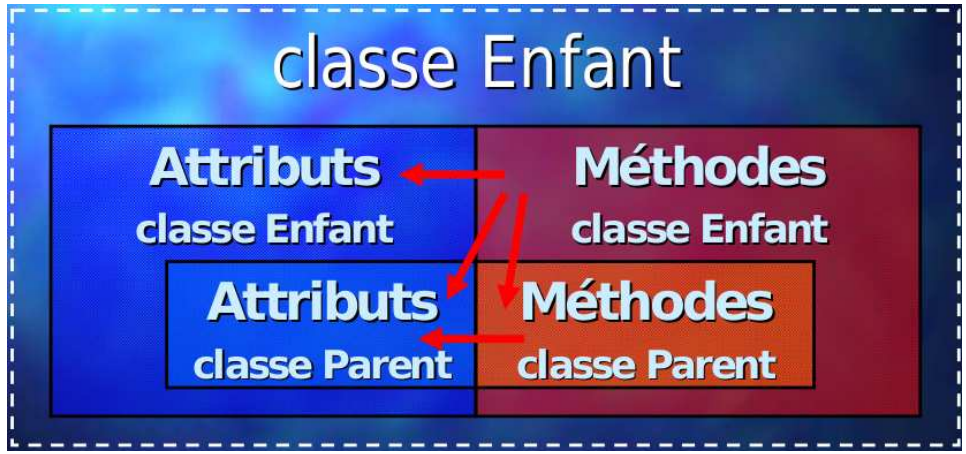
- **étendre** la fonctionnalité des classes déjà existantes : une instance de la classe `Enfant` est au même temps un objet de type `Parent1` et un objet de type `Parent2`, en ayant en plus d'autres membres et méthodes
- **réutilisation du code** : la classe dérivée `Enfant` hérite tous les membres et méthodes des classes de base
- **réutilisation du code** : l'héritage encourage le programmeur à s'appuyer sur du code déjà testé et de très bonne qualité, et empêche la multiplication inutile de code dans un projet

L'intérêt de l'héritage

Objectifs et idée générale

- **étendre** la fonctionnalité des classes déjà existantes : une instance de la classe `Enfant` est au même temps un objet de type `Parent1` et un objet de type `Parent2`, en ayant en plus d'autres membres et méthodes
- **réutilisation du code** : la classe dérivée `Enfant` hérite tous les membres et méthodes des classes de base
- **réutilisation du code** : l'héritage encourage le programmeur à s'appuyer sur du code déjà testé et de très bonne qualité, et empêche la multiplication inutile de code dans un projet
- un pointeur vers une instance `Enfant` peut être converti dans un pointeur vers une instance `Parent`, et utilisé dans cette forme. L'opération inverse n'est valide que si l'objet pointé est réellement de type `Enfant` à l'origine.

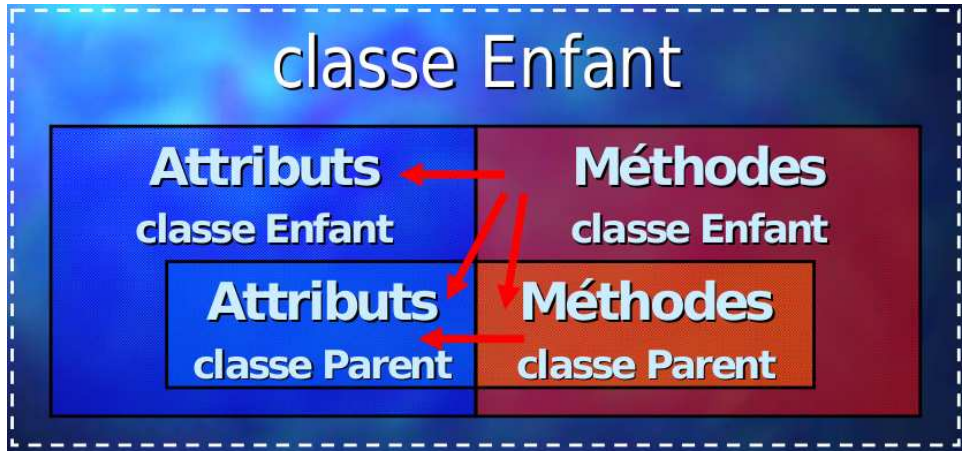
Héritage monoparental



Caractéristiques

- le cas illustratif de base (et le plus fréquent)

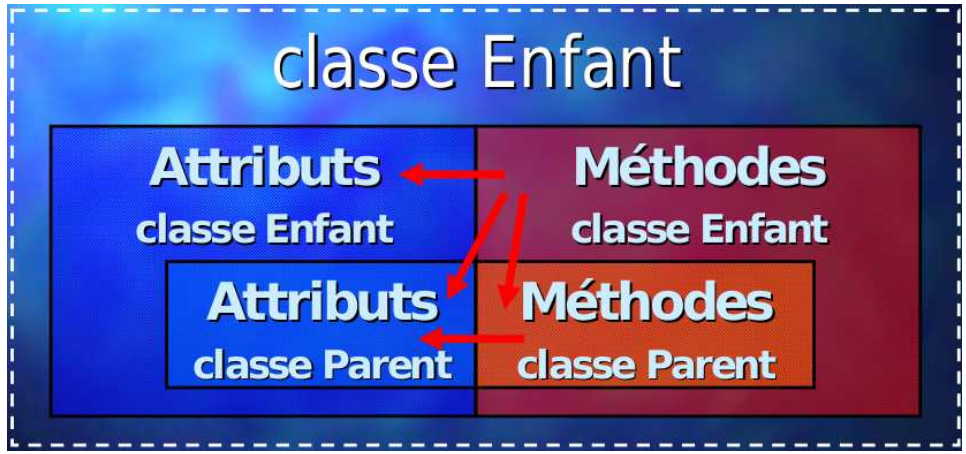
Héritage monoparental



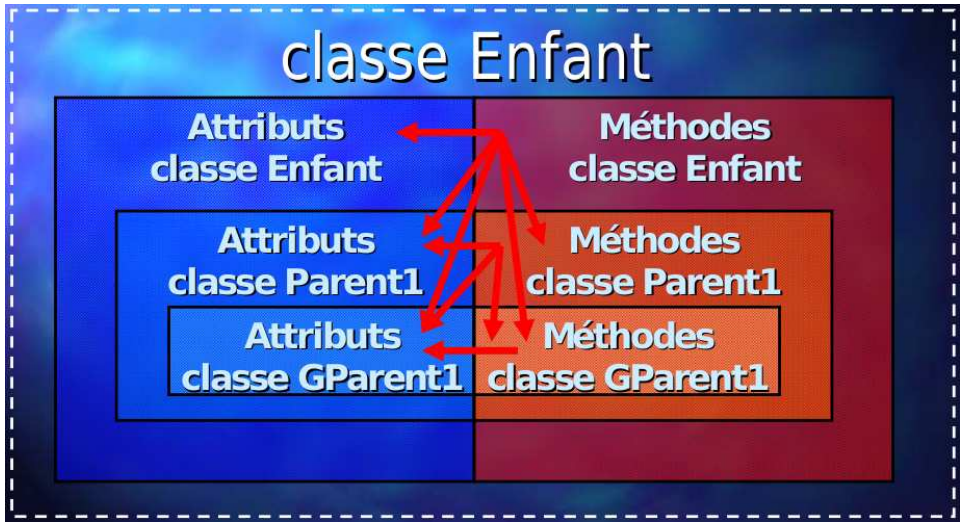
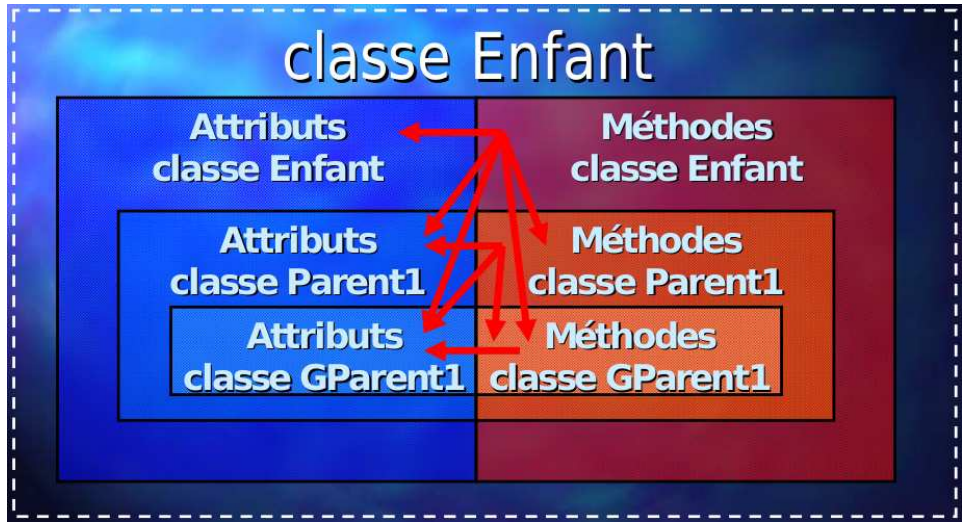
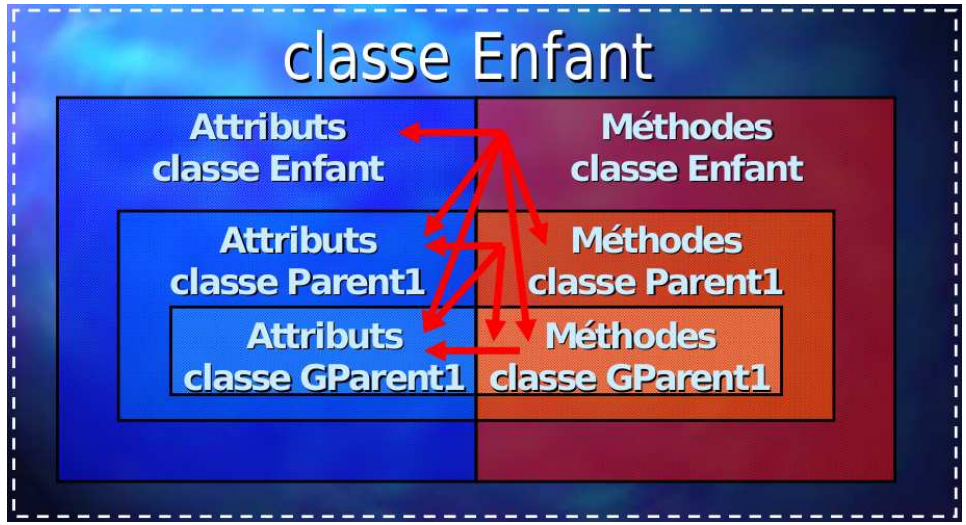
Caractéristiques

- le cas illustratif de base (et le plus fréquent)
- **intérêt** : les méthodes de la classe dérivée `Enfant` peuvent exploiter les attributs et méthodes de la classe de base `Parent`

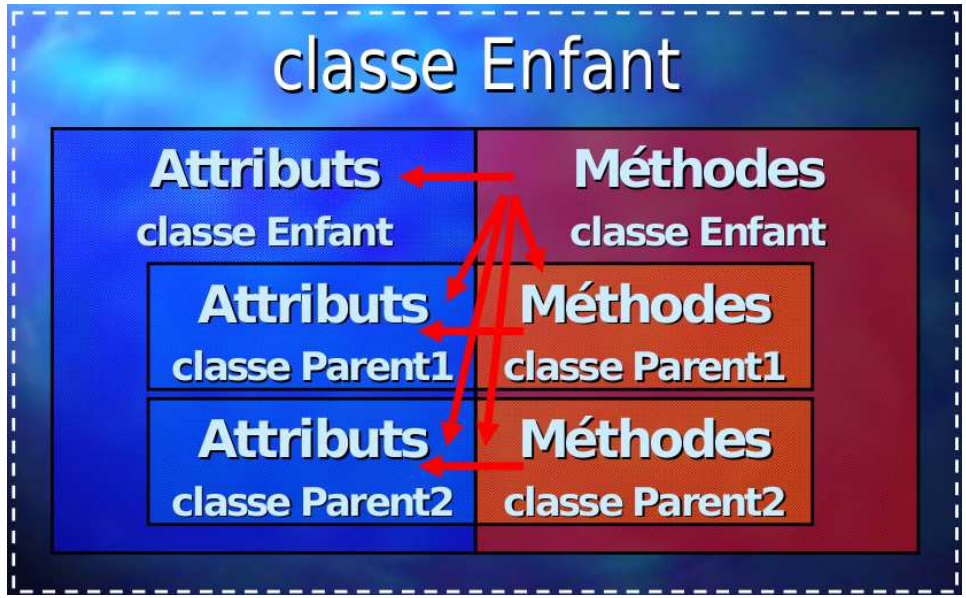
Héritage monoparental



Caractéristiques

- le cas illustratif de base (et le plus fréquent)
- **intérêt** : les méthodes de la classe dérivée `Enfant` peuvent exploiter les attributs et méthodes de la classe de base `Parent`
- ```
  class Enfant : public Parent
  {
    ...
  };
  ```
  ```

Héritage multiple



Caractéristiques

- un cas fréquent également : illustration des hiérarchies des classes

Héritage multiple



Caractéristiques

- un cas fréquent également : illustration des hiérarchies des classes
- **intérêt** : les classes dérivées sont des instances spécialisées des classes de base, et la spécialisation peut se faire en cascade

Héritage multiple



Caractéristiques

- un cas fréquent également : illustration des hiérarchies des classes
- **intérêt** : les classes dérivées sont des instances spécialisées des classes de base, et la spécialisation peut se faire en cascade
- ```
  class Parent1 : public GParent1
  {
    ...
  };
  class Enfant : public Parent1
  {
    ...
  };
  ```
  ```

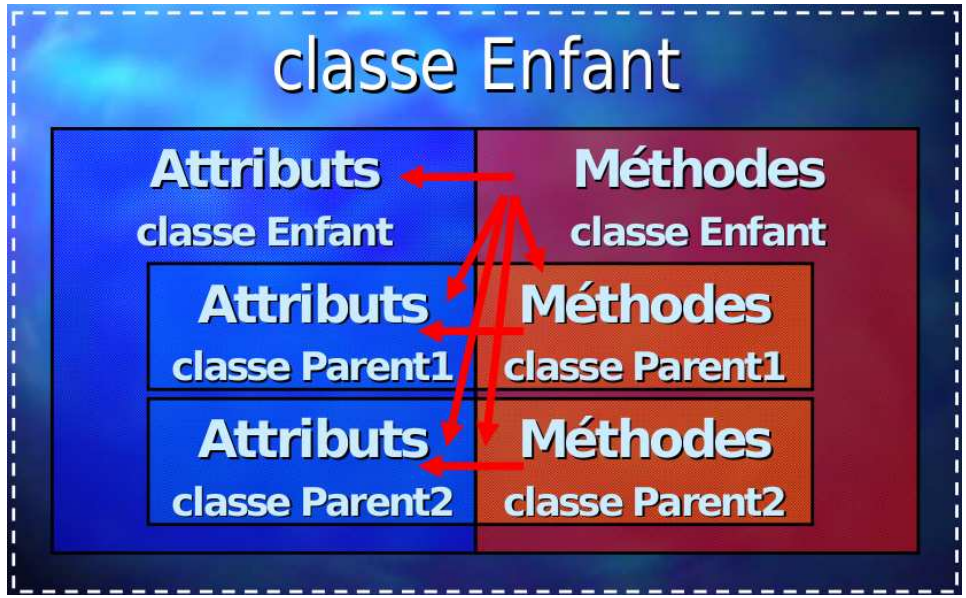
Héritage multi-parental



Cas 1 : parents indépendants

- un cas moins fréquent : illustration des hiérarchies des classes

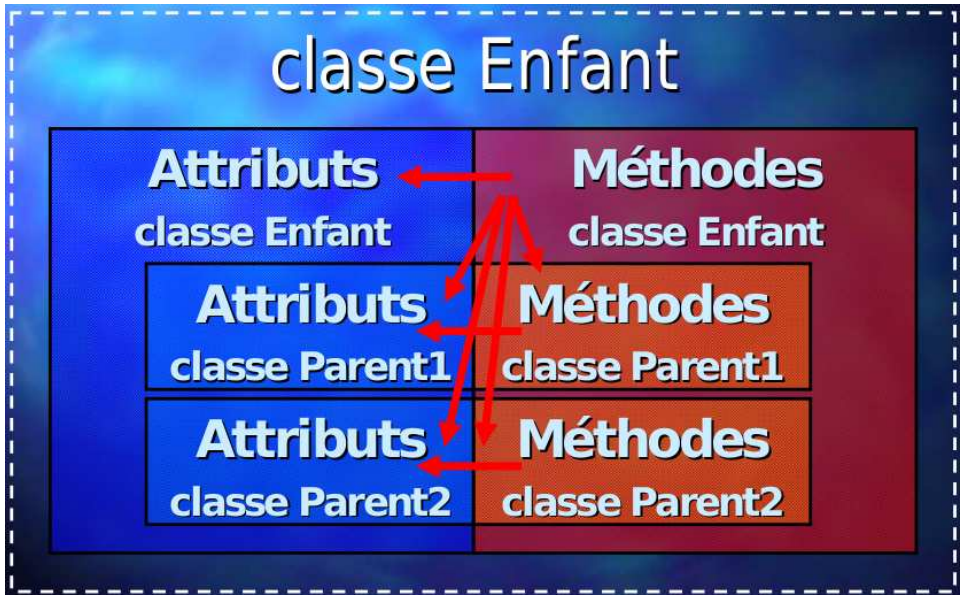
Héritage multi-parental



Cas 1 : parents indépendants

- un cas moins fréquent : illustration des hiérarchies des classes
- **intérêt** : certaines classes héritent des fonctionnalités nécessaires à partir de deux classes de base indépendantes

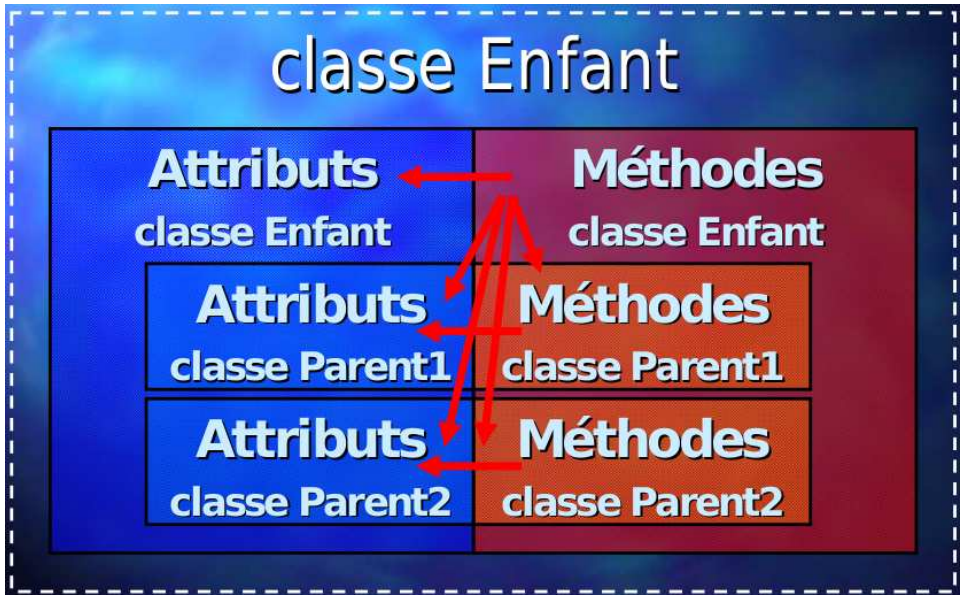
Héritage multi-parental



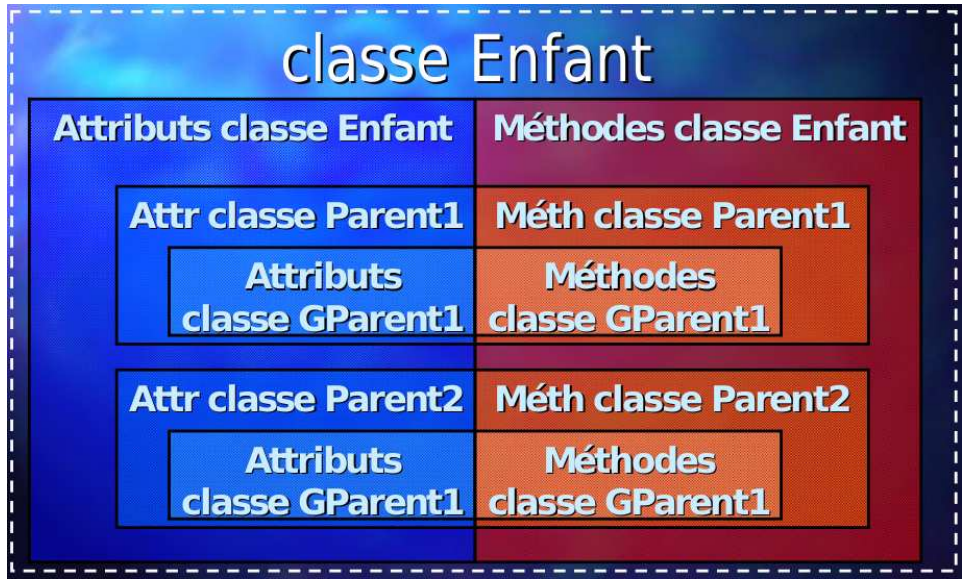
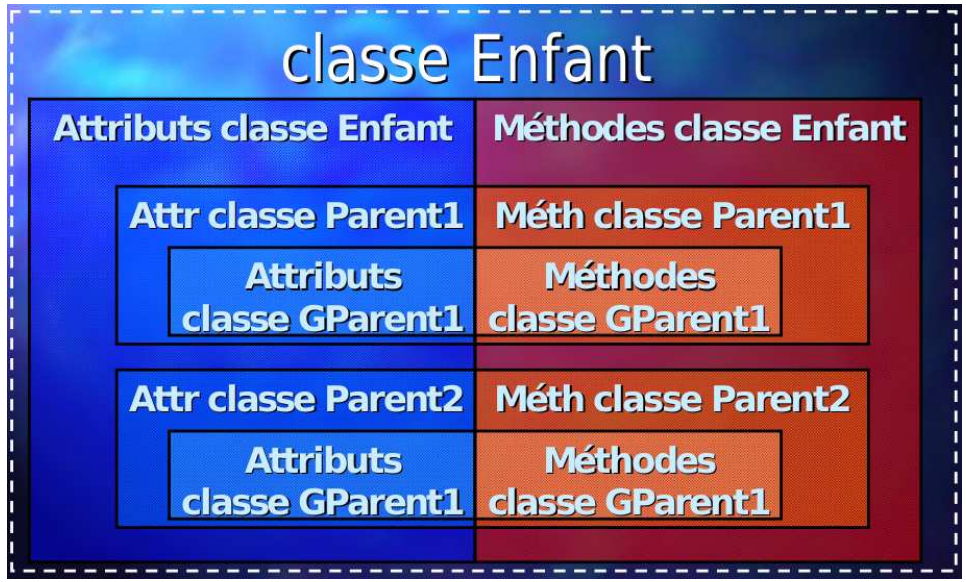
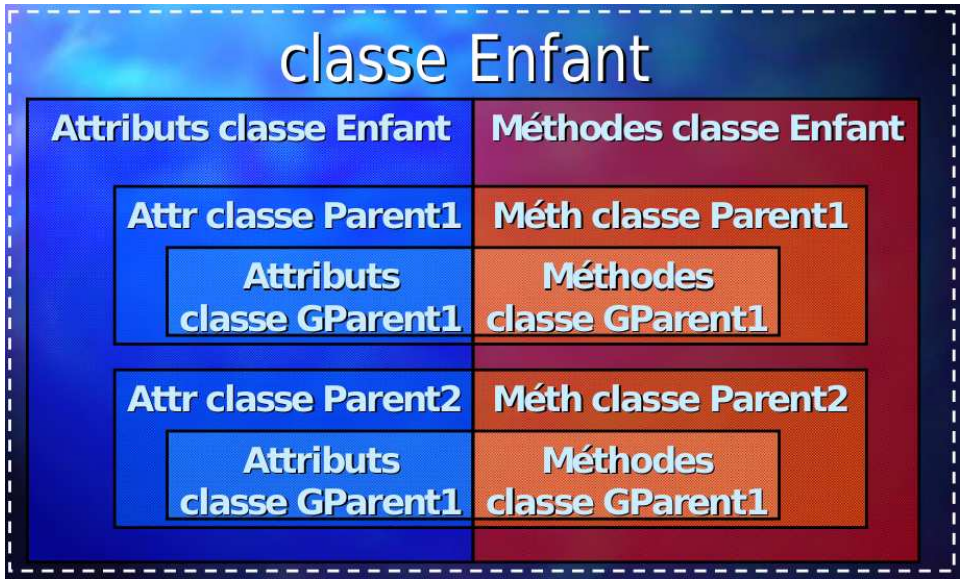
Cas 1 : parents indépendants

- un cas moins fréquent : illustration des hiérarchies des classes
- **intérêt** : certaines classes héritent des fonctionnalités nécessaires à partir de deux classes de base indépendantes
- **à éviter si possible** : il y a des pratiques recommandées qui font éviter dans la plupart des cas ce type de héritage

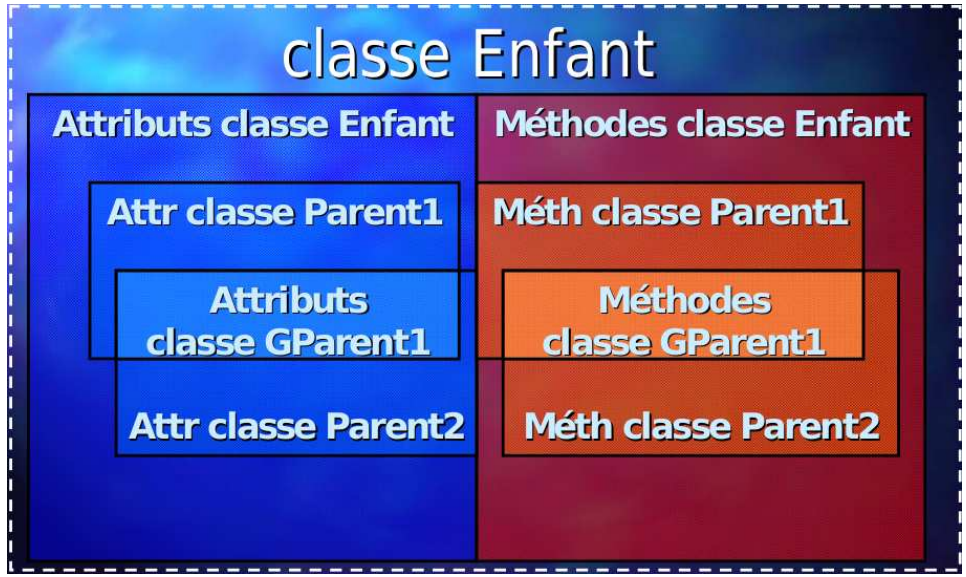
Héritage multi-parental



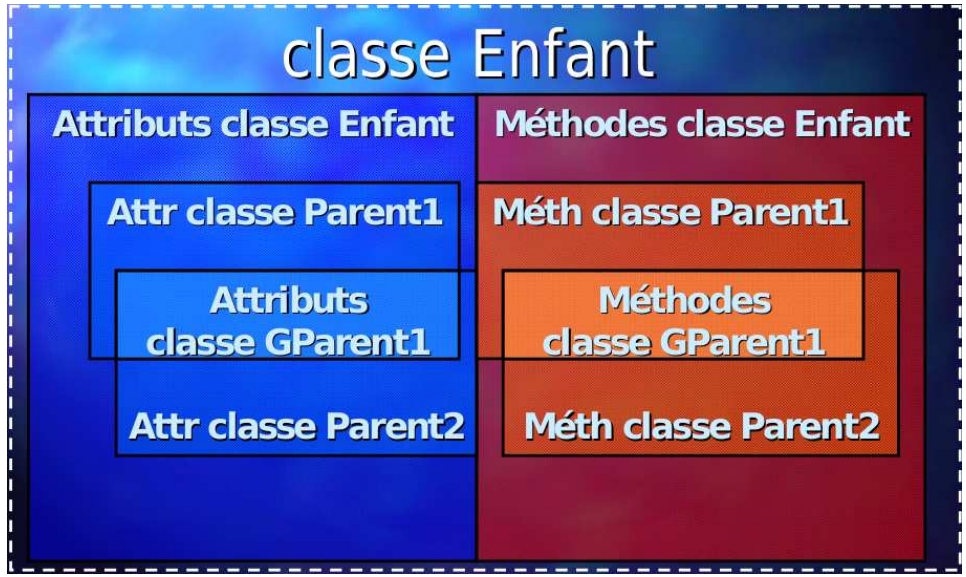
Cas 1 : parents indépendants

- un cas moins fréquent : illustration des hiérarchies des classes
- **intérêt** : certaines classes héritent des fonctionnalités nécessaires à partir de deux classes de base indépendantes
- **à éviter si possible** : il y a des pratiques recommandées qui font éviter dans la plupart des cas ce type de héritage
- ```
  class Enfant : public Parent1, public Parent2
  {
    ...
  };
  ```
  ```

Héritage multi-parental



Cas 2 : parents ayant une classe de base commune

- **à éviter absolument** : des ambiguïtés graves car la classe de base commune aux parents est dupliquée à l'intérieur de l'objet `Enfant`

Héritage multi-parental



Cas 2 : parents ayant une classe de base commune

- **à éviter absolument** : des ambiguïtés graves car la classe de base commune aux parents est dupliquée à l'intérieur de l'objet `Enfant`
- situation connue sous le nom du **diamant de la mort**

Héritage multi-parental



Cas 2 : parents ayant une classe de base commune

- **à éviter absolument** : des ambiguïtés graves car la classe de base commune aux parents est dupliquée à l'intérieur de l'objet `Enfant`
- situation connue sous le nom du **diamant de la mort**
- ```
  class Parent1 : public GParent
  {  ... };
  class Parent2 : public GParent
  {  ... };
  class Enfant : public Parent1, public Parent2
  {  ... };
  ```
  ```

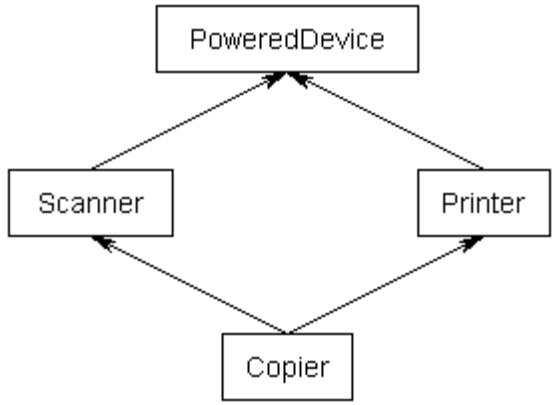
Héritage multi-parental



Première solution pour éviter le diamant de la mort

- déclarer le héritage du `GParent` comme **virtual**
- ```
  class Parent1 :  virtual public GParent
  {  ... };
  class Parent2 :  virtual public GParent
  {  ... };
  class Enfant : public Parent1, public Parent2
  {  ... };
  ```
  ```

Héritage multi-parental



Première solution pour éviter le diamant de la mort

- déclarer le héritage du `GParent` comme **virtual**
- ```
  class Parent1 :  virtual public GParent
  {  ... };
  class Parent2 :  virtual public GParent
  {  ... };
  class Enfant : public Parent1, public Parent2
  {  ... };
  ```
- on appelle automatiquement le constructeur sans paramètres du `GParent` (s'il n'existe pas : erreur !)
  ```

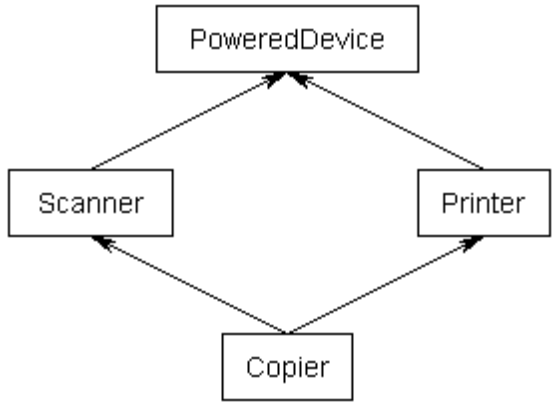
Héritage multi-parental



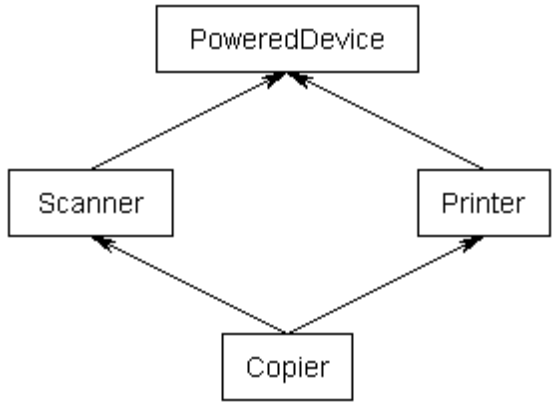
Deuxième solution pour éviter le diamant de la mort

- à la place de dire que le copieur est un scanner + printer

Héritage multi-parental



Deuxième solution pour éviter le diamant de la mort

- à la place de dire que le copieur est un scanner + printer
- ```
  class ComplexObj
  {
     Parent1 scanner;
     Parent2 printer;
  };
  ```
  ```

Héritage multi-parental



Deuxième solution pour éviter le diamant de la mort

- à la place de dire que le copieur est un scanner + printer
- ```
  class ComplexObj
  {
     Parent1 scanner;
     Parent2 printer;
  };
  ```
- **composition** : designer une classe complexe conteneur
  ```

Héritage - vie des objets

Les constructeurs / destructeurs

- Syntaxe d'appel des constructeurs des classes hérités : la même que pour les attributs

Héritage - vie des objets

Les constructeurs / destructeurs

- Syntaxe d'appel des constructeurs des classes hérités : la même que pour les attributs
- La classe `Enfant` peut appeler explicitement (sans être obligée) tout constructeur non - privé des classes parents (un seul par classe héritée)

Héritage - vie des objets

Les constructeurs / destructeurs

- Syntaxe d'appel des constructeurs des classes hérités : la même que pour les attributs
- La classe `Enfant` peut appeler explicitement (sans être obligée) tout constructeur non - privé des classes parents (un seul par classe héritée)
- L'appel des constructeurs des grand-parents : interdit ou dangereux (en fonction de compilateur)

Héritage - vie des objets

Les constructeurs / destructeurs

- Syntaxe d'appel des constructeurs des classes hérités : la même que pour les attributs
- La classe `Enfant` peut appeler explicitement (sans être obligée) tout constructeur non - privé des classes parents (un seul par classe héritée)
- L'appel des constructeurs des grand-parents : interdit ou dangereux (en fonction de compilateur)
- L'ordre d'appel effectif = l'ordre de déclaration

Héritage - vie des objets

Les constructeurs / destructeurs

- Syntaxe d'appel des constructeurs des classes hérités : la même que pour les attributs
- La classe `Enfant` peut appeler explicitement (sans être obligée) tout constructeur non - privé des classes parents (un seul par classe héritée)
- L'appel des constructeurs des grand-parents : interdit ou dangereux (en fonction de compilateur)
- L'ordre d'appel effectif = l'ordre de déclaration
- Destruction : le compilateur appelle, après le corps du destructeur, tous les destructeurs des classes héritées

Héritage - visibilité des membres

Cas d'usage en héritage multiple

- Par défaut tous les attributs et les méthodes héritées sont visibles à la classe `Enfant` (en respectant les droits d'accès). Mais on peut avoir des situations ambiguës.

Héritage - visibilité des membres

Cas d'usage en héritage multiple

- Par défaut tous les attributs et les méthodes héritées sont visibles à la classe `Enfant` (en respectant les droits d'accès). Mais on peut avoir des situations ambiguës.
- Visibilité cachée par un membre de l'enfant

Héritage - visibilité des membres

Cas d'usage en héritage multiple

- Par défaut tous les attributs et les méthodes héritées sont visibles à la classe `Enfant` (en respectant les droits d'accès). Mais on peut avoir des situations ambiguës.
- Visibilité cachée par un membre de l'enfant
- Ambiguïté entre deux membres portant le même nom mais se trouvant dans deux sous-objets différents

Héritage - visibilité des membres

Cas d'usage en héritage multiple

- Par défaut tous les attributs et les méthodes héritées sont visibles à la classe `Enfant` (en respectant les droits d'accès). Mais on peut avoir des situations ambiguës.
- Visibilité cachée par un membre de l'enfant
- Ambiguïté entre deux membres portant le même nom mais se trouvant dans deux sous-objets différents
- **Solution** : opérateur de résolution de portée `::` vers l'espace du sous-objet

Héritage - accès

- L'accès aux membres hérités est soumis aux droits d'accès résultant de l'héritage

Héritage - accès

- L'accès aux membres hérités est soumis aux droits d'accès résultant de l'héritage
- Sans le préciser, par défaut, l'héritage est de type **private**

Héritage - accès

- L'accès aux membres hérités est soumis aux droits d'accès résultant de l'héritage
- Sans le préciser, par défaut, l'héritage est de type **private**
- L'héritage ne fait que renforcer les contraintes d'accès (voir le tableau des droits)

Héritage - accès

- L'accès aux membres hérités est soumis aux droits d'accès résultant de l'héritage
- Sans le préciser, par défaut, l'héritage est de type **private**
- L'héritage ne fait que renforcer les contraintes d'accès (voir le tableau des droits)
- **Exception** : les opérateurs membres sont toujours hérités en mode public (il est impossible de restreindre leur accès)

Héritage - accès

- L'accès aux membres hérités est soumis aux droits d'accès résultant de l'héritage
- Sans le préciser, par défaut, l'héritage est de type **private**
- L'héritage ne fait que renforcer les contraintes d'accès (voir le tableau des droits)
- **Exception** : les opérateurs membres sont toujours hérités en mode public (il est impossible de restreindre leur accès)

Accès initial	H. public	H. protected	H. private
public	public	protected	private
protected	protected	protected	private
private	pas d'accès	pas d'accès	pas d'accès

Surcharge et polymorphisme

- En C++ le **polymorphisme** est la capacité à donner le même nom à des fonctions / méthodes / opérateurs qui ont un comportement diffèrent en fonction des paramètres, des objets et du contexte dans lequel on les emploie.

Surcharge et polymorphisme

- En C++ le **polymorphisme** est la capacité à donner le même nom à des fonctions / méthodes / opérateurs qui ont un comportement diffèrent en fonction des paramètres, des objets et du contexte dans lequel on les emploie.
- Exemples en C “classique" : les opérateurs $+ \quad - \quad / \quad * \quad = \quad < \quad >$

Surcharge et polymorphisme

- En C++ le **polymorphisme** est la capacité à donner le même nom à des fonctions / méthodes / opérateurs qui ont un comportement diffèrent en fonction des paramètres, des objets et du contexte dans lequel on les emploie.
- Exemples en C “classique" : les opérateurs $+ \quad - \quad / \quad * \quad = \quad < \quad >$
- Comportement différent réel / entier / signé etc.

Types de polymorphisme

- Polymorphisme **statique de surcharge**

Types de polymorphisme

- Polymorphisme **statique de surcharge**
  - on peut utiliser un objet enfant à la place du parent parce qu'il le contient

Types de polymorphisme

- Polymorphisme **statique de surcharge**
  - on peut utiliser un objet enfant à la place du parent parce qu'il le contient
  - deux versions : **statique** et **dynamique**

Types de polymorphisme

- Polymorphisme **statique de surcharge**
  - on peut utiliser un objet enfant à la place du parent parce qu'il le contient
  - deux versions : **statique** et **dynamique**
- Polymorphisme **statique** : comportement connu et déterminé à la compilation

Types de polymorphisme

- Polymorphisme **statique de surcharge**
  - on peut utiliser un objet enfant à la place du parent parce qu'il le contient
  - deux versions : **statique** et **dynamique**
- Polymorphisme **statique** : comportement connu et déterminé à la compilation
- Polymorphisme **dynamique** : comportement inconnu à la compilation et déterminé à l'exécution, par l'objet

Polymorphisme d'héritage statique

- Une classe enfant peut surcharger (redéfinir) le corps d'une méthode du parent à la condition de respecter le même prototype (même signature)

Polymorphisme d'héritage statique

- Une classe enfant peut surcharger (redéfinir) le corps d'une méthode du parent à la condition de respecter le même prototype (même signature)
- Les nouvelles méthodes de l'enfant appelleront par défaut la méthode surchargée alors que les méthodes héritées appelleront la méthode initiale du parent

Polymorphisme d'héritage statique

```
class Parent
{ // ...
const char* GetName () {return "Parent";}
void AffName () { printf ("%s \ n", GetName ());}
};
class Enfant : public Parent
{ // ...
const char* GetName () {return "Enfant";}
void Aff2Name() { printf ("%s \ n", GetName ());}
void Aff3Name() { printf ("%s \ n",Parent:: GetName ());}
void Aff4Name() { printf ("%s \ n", AffName ());}
};

void main()
{ Enfant enf1, *penf2=new Enfant;
Parent par1;
printf ("%s \ n",enf1. GetName ()); //"Enfant"
printf ("%s \ n",penf2 - > GetName ()); //"Enfant"
printf ("%s \ n",(Parent*)penf2 - > GetName ()); //"Parent"
printf ("%s \ n",par1. GetName ()); //"Parent"
printf ("%s \ n",((Parent*)&enf1) - > GetName ());//"Parent"
enf1. AffName (); //"Parent"
penf2 - >Aff2Name(); //"Enfant"
enf1.Aff3Name(); //"Parent"
((Parent)enf1). AffName (); //"Parent"
enf1.Aff4Name(); //"Parent"
};
```

Polymorphisme d'héritage dynamique

- Le PHD n'est actif que pour les appels directs , sans opérateur de résolution de portée

Polymorphisme d'héritage dynamique

- Le PHD n'est actif que pour les appels directs , sans opérateur de résolution de portée
- En utilisant le qualificatif **virtuel** dans la déclaration de prototype d'une méthode du parent, la surcharge devient effective **aussi** pour les méthodes du parent héritées par l'enfant

Polymorphisme d'héritage dynamique

- Le PHD n'est actif que pour les appels directs , sans opérateur de résolution de portée
- En utilisant le qualificatif **virtuel** dans la déclaration de prototype d'une méthode du parent, la surcharge devient effective **aussi** pour les méthodes du parent héritées par l'enfant
- La redéfinition remplace la méthode des parents

Polymorphisme d'héritage dynamique

- Le PHD n'est actif que pour les appels directs , sans opérateur de résolution de portée
- En utilisant le qualificatif **virtuel** dans la déclaration de prototype d'une méthode du parent, la surcharge devient effective **aussi** pour les méthodes du parent héritées par l'enfant
- La redéfinition remplace la méthode des parents
- Le caractère virtuel d'une méthode est défini dans la classe parent et il ne change plus à travers les héritages

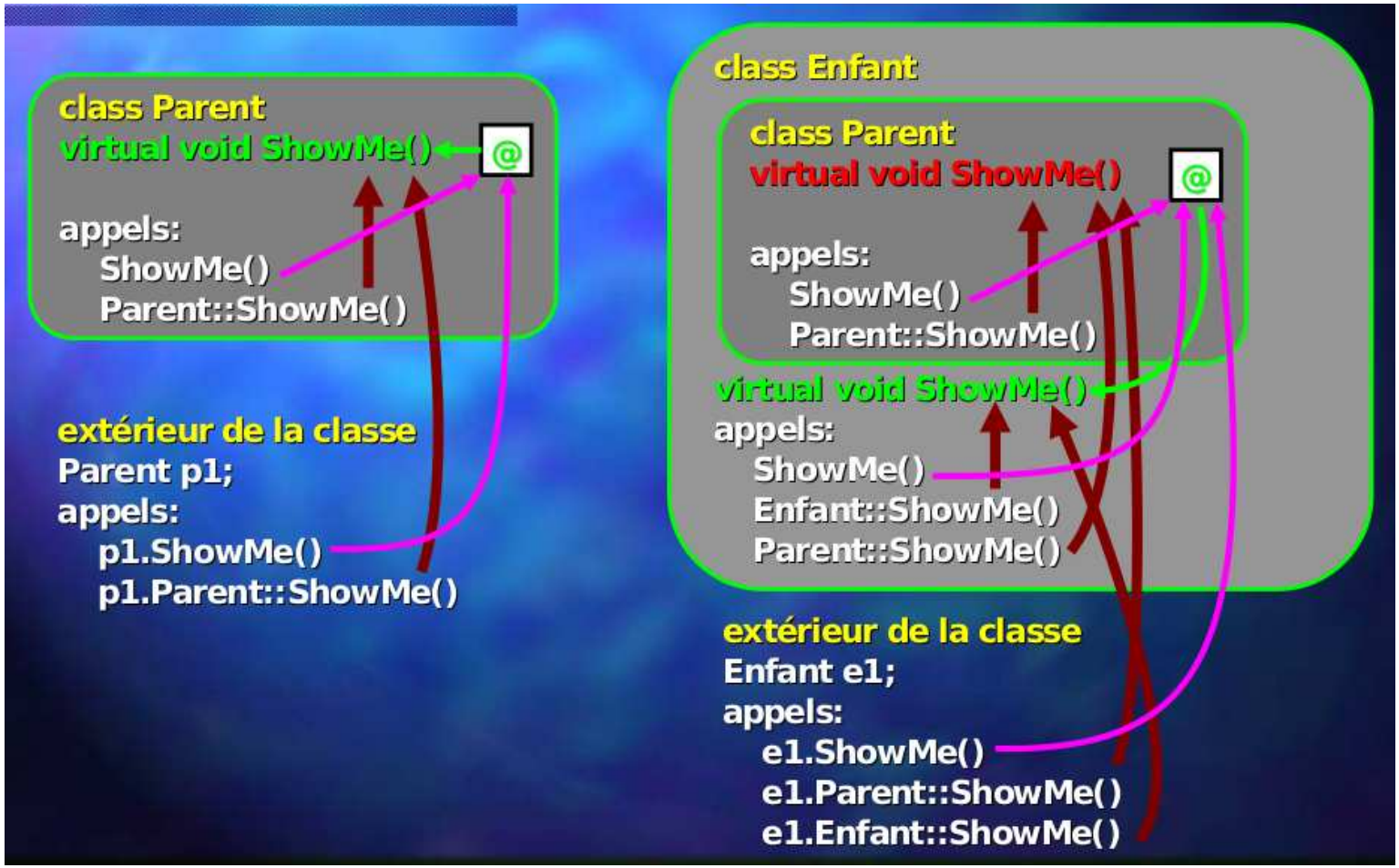
Polymorphisme d'héritage dynamique

- La version initiale de la méthode virtuelle surchargée reste visible aux parents et aux enfants seulement par l'intermédiaire de l'opérateur de portée `Parent:: Methode ()`

Polymorphisme d'héritage dynamique

- La version initiale de la méthode virtuelle surchargée reste visible aux parents et aux enfants seulement par l'intermédiaire de l'opérateur de portée `Parent:: Methode ()`
- L'adresse d'une méthode virtuelle est un attribut statique caché (tables virtuelles) de la classe, tout appel direct passe par cette adresse

Polymorphisme d'héritage dynamique



Polymorphisme d'héritage dynamique

```
class Parent
{ // ...
virtual const char* GetName () {return "Parent";}
void AffName () { printf ("%s \ n", GetName ());}
};
class Enfant : public Parent
{ // ...
const char* GetName () {return "Enfant";}
void Aff2Name() { printf ("%s \ n", GetName ());}
void Aff3Name() { printf ("%s \ n",Parent:: GetName ());}
void Aff4Name() { printf ("%s \ n", AffName ());}
};

void main()
{ Enfant enf1, *penf2=new Enfant;
Parent par1;
printf ("%s \ n",enf1. GetName ()); //"Enfant"
printf ("%s \ n",penf2 - > GetName ()); //"Enfant"
printf ("%s \ n",(Parent*)penf2 - > GetName ()); //"Enfant"
printf ("%s \ n",par1. GetName ()); //"Parent"
printf ("%s \ n",((Parent*)&enf1) - > GetName ());//"Enfant"
enf1. AffName (); //"Enfant"
penf2 - >Aff2Name(); //"Enfant"
enf1.Aff3Name(); //"Parent"
((Parent)enf1). AffName (); //"Parent"
enf1.Aff4Name(); //"Enfant"
};
```

Polymorphisme d'héritage dynamique

- La classe parent peut ne pas définir la méthode virtuelle : alors elle devient une méthode virtuelle pure

```
class Parent { // ...
virtual const char* GetName () =0 ;
};
```

Polymorphisme d'héritage dynamique

- La classe parent peut ne pas définir la méthode virtuelle : alors elle devient une méthode virtuelle pure

```
class Parent { // ...
virtual const char* GetName () =0 ;
};
```

- Concrètement, l'adresse de la méthode (dans la table virtuelle) est initialisée à zéro

Polymorphisme d'héritage dynamique

- La classe parent peut ne pas définir la méthode virtuelle : alors elle devient une méthode virtuelle pure

```
class Parent { // ...
virtual const char* GetName () =0 ;
};
```

- Concrètement, l'adresse de la méthode (dans la table virtuelle) est initialisée à zéro
- Les héritiers ont l'obligation de définir cette méthode, en respectant le prototype

Polymorphisme d'héritage dynamique

- La classe parent peut ne pas définir la méthode virtuelle : alors elle devient une méthode virtuelle pure

```
class Parent { // ...
virtual const char* GetName () =0 ;
};
```

- Concrètement, l'adresse de la méthode (dans la table virtuelle) est initialisée à zéro
- Les héritiers ont l'obligation de définir cette méthode, en respectant le prototype
- **Base abstraite** : une classe qui a au moins une méthode virtuelle pure ; elle ne peut pas avoir d'instances

Polymorphisme d'héritage dynamique

- La classe parent peut ne pas définir la méthode virtuelle : alors elle devient une méthode virtuelle pure

```
class Parent { // ...
virtual const char* GetName () =0 ;
};
```

- Concrètement, l'adresse de la méthode (dans la table virtuelle) est initialisée à zéro
- Les héritiers ont l'obligation de définir cette méthode, en respectant le prototype
- **Base abstraite** : une classe qui a au moins une méthode virtuelle pure ; elle ne peut pas avoir d'instances
- Une base abstraite ne sert que pour être héritée : c'est le point de départ d'une famille de classes, par exemple