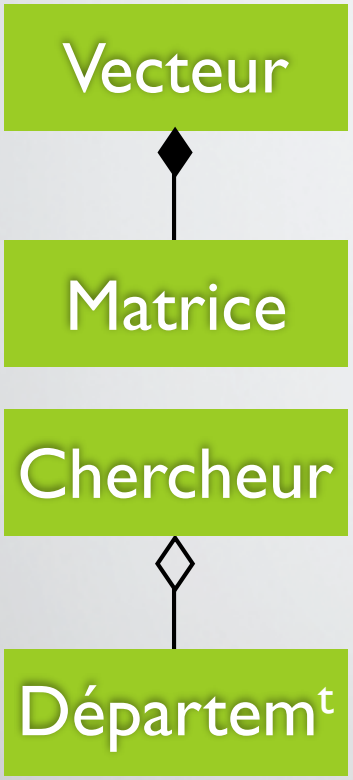
Héritage, héritage multiple

- L'héritage est un des fondements de la POO
  - permet de réutiliser, de spécialiser et d'étendre des classes
- On définit une nouvelle **classe dérivée** à partir d'une **classe existante de base**
  - la classe dérivée hérite des propriétés (attributs et méthodes) de la classe de base
- Le C++ autorise l'**héritage multiple** : une classe peut hériter de plusieurs classes de base

Modèle objet
> Composition & Agrégation

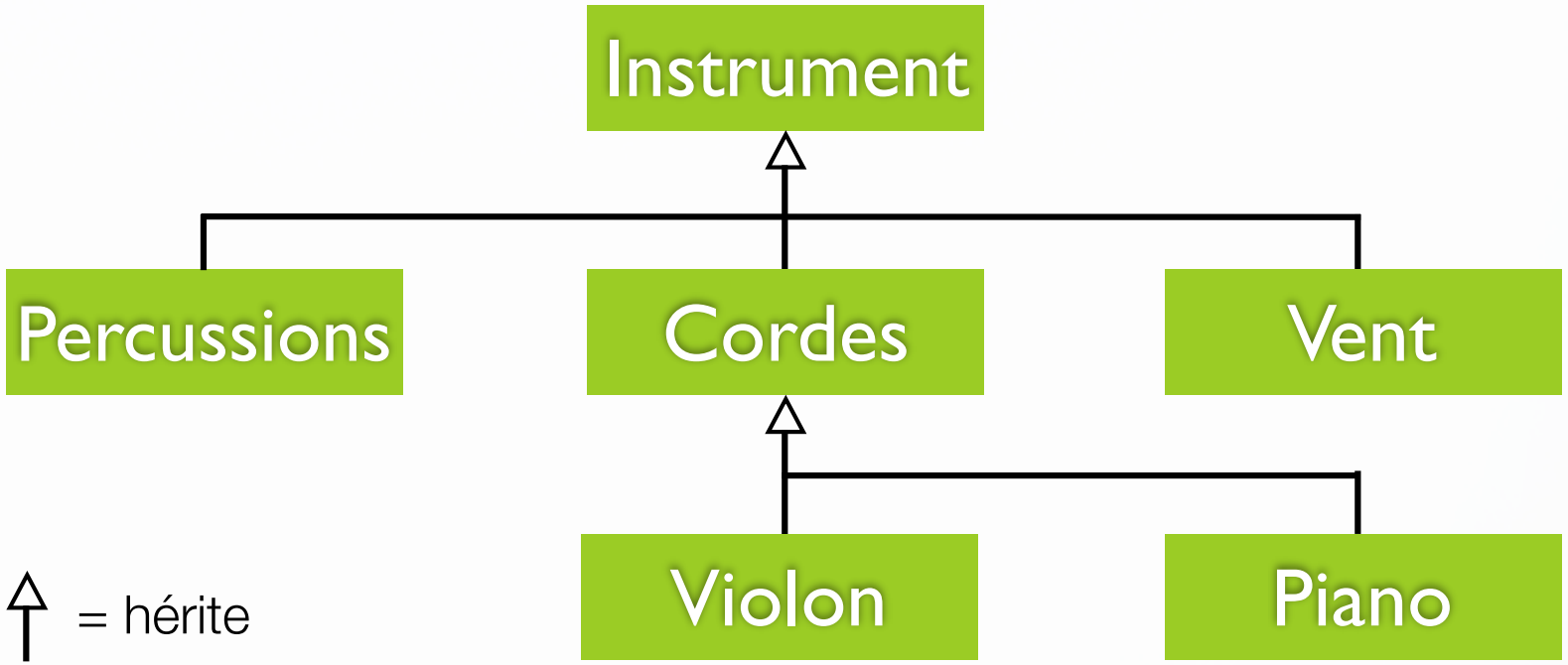


- **Composition : relation d'appartenance** entre deux classes
  - *ex* : une matrice 4x4 est composé de 4 vecteurs de dimension 4
  - la création (*resp*. destruction) d'une instance de la classe qui possède (matrice) entraîne la création (*resp*. destruction) de la seconde (vecteur)
- **Agrégation : relation sans appartenance** entre deux classes
  - *ex* : un département scientifique est composé (entre autres) de chercheurs
  - un chercheur peut aussi travailler dans un autre département
  - la disparition du département n'entraîne pas celle de ses chercheurs !!

Modèle objet
> Héritage (spécialisation)

- Propriété qui permet de construire de nouvelles classes d'objet à partir de classes existantes
  - la classe existante est appelée **classe de base**
  - la nouvelle classe est appelée **classe dérivée**, elle **hérite de la structure (attributs) et du comportement (méthodes)** de la classe de base
- But : enrichir la classe de base
  - en ajoutant de nouvelles méthodes ou de nouveaux attributs
  - en remplaçant des méthodes existantes
  - en étendant des méthodes existantes

Héritage
> Diagramme de classes



107

Instrument

Structure interne
string *nom, marque*;

Interface
init(string *nom*);
setMarque(string *m*);
jouerNote(entier *n*);

Instrument ⇧ Cordes

Structure interne
string *nom, marque*;
réel *cordes[]*;

Interface
init(string *nom*);
setMarque(string m);
jouerNote(entier *n*);
setCordes(réel r[]);

Cordes ⇧ Piano

Structure interne
string *nom, marque*;
réel *cordes[]*;
entier type;

Interface
init(string *nom*);
setMarque(string m);
jouerNote(entier *n*);
setCordes(réel r[]);
setType(entier *t*);

setMarque = hérité jouerNote = redéfini setCordes = nouveau ⇧ = hérite (*inherits*)

Héritage
> Utilisations

- Réutilisation de code en “factorisant” le code commun dans une classe de base
- Mettre en place un “protocole” (interface à laquelle doivent se conformer des classes dérivées)
- Créer des modules de fonctionnalités génériques (à spécialiser pour ses propres besoins)
- Apporter de légères modifications (*customisation* d’UI)
- Expérimenter (tester de nouvelles implémentations)

Templates (C++)

> Classes et méthodes génériques

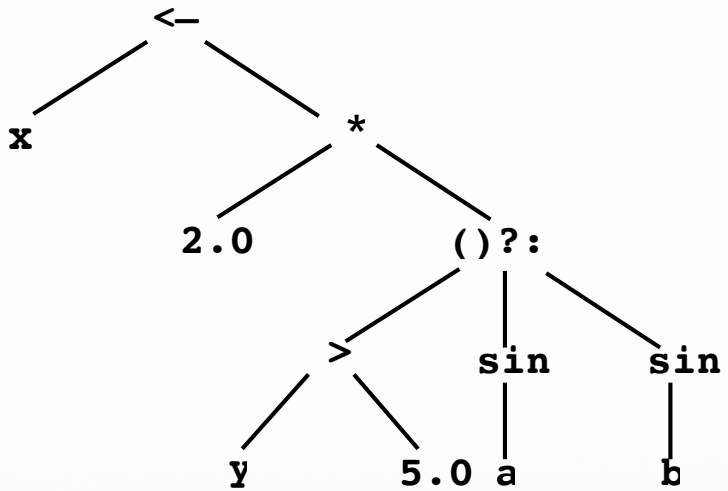
- Objectifs : concevoir des classes ou des fonctions génériques applicables à différents types de base
  - on écrit une seule fois le code pour un type générique T
  - le code est instancié par le compilateur pour tous les types particuliers souhaités (et compatibles)
- Très utile pour les structures de données classiques (*cf*. STL)
  - tableaux dynamiques, listes, tables de hachage, arbres *etc*...
  - applicable pour des types simples (int, float...) mais aussi des classes complexes

Modélisation objet
> Exemple

- Programmation d'un interpréteur d'expressions arithmétiques
  - objectif : créer un programme fonctionnel qui met en œuvre la (presque) totalité des concepts de POO !
  - mettre à disposition de l'utilisateur des outils de programmation simples pour la création, l'affichage et l'évaluation d'expressions évoluées
  - utilisation de **variables** : le seul type numérique utilisé est le réel (*double*)
  - expressions évoluées : les **affectations** dans des variables, les **expressions booléennes**, une **expression *if-then-else*** qui retourne le résultat de l'expression branchée, une **boucle *for*** !..

Expressions arithmétiques
> Représentation

- Une expression est représentée sous la forme d'un arbre dont chaque nœud est une sous-expression particulière
- Exemple : x <— 2.0 * ((y>5.0)? sin(a) : sin(b))

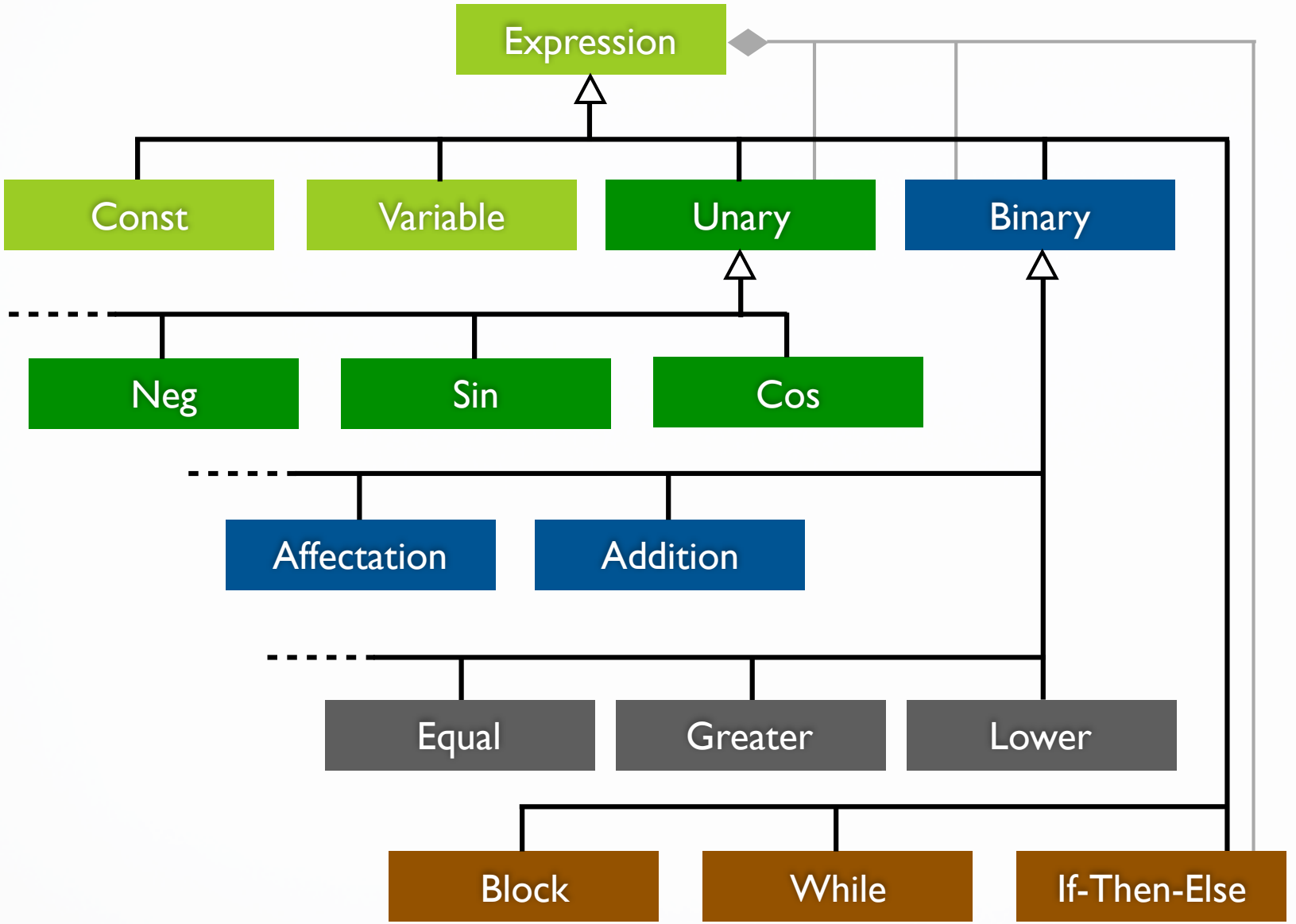


Expression	Const	Variable
- une représentation textuelle	- une représentation textuelle - une valeur réelle	- une représentation textuelle - une valeur
- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer (-> valeur de la var.) - affecter une valeur - obtenir représentation textuelle / afficher

Unary	Binary	Print
- une représentation textuelle - une sous-expression	- une représentation textuelle - deux sous-expressions	- une représentation textuelle - une sous-expression
- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer - obtenir représentation textuelle / afficher

Cos...	Addition...	Affectation
- une représentation textuelle - une sous-expression	- une représentation textuelle - une sous-expression	- une représentation textuelle - une variable - une sous-expression
- initialiser - évaluer (cosinus) - obtenir représentation textuelle / afficher	- initialiser - évaluer (somme) - obtenir représentation textuelle / afficher	- initialiser - évaluer (affecter la variable) - obtenir représentation textuelle / afficher

If-Then-Else	While	Block
- une représentation textuelle - une expression booléenne - deux sous-expressions	- une représentation textuelle - une sous-expression	- une représentation textuelle - une séquence de sous-expressions
- initialiser - évaluer (-> expr. branchée) - obtenir représentation textuelle / afficher	- initialiser - évaluer (-> dernière éval) - obtenir représentation textuelle / afficher	- initialiser - évaluer (-> dernière éval) - obtenir représentation textuelle / afficher



Modélisation objet de l'interpréteur

> Diagramme de classes

Expressions arithmétiques
> Classe mère `Expression`

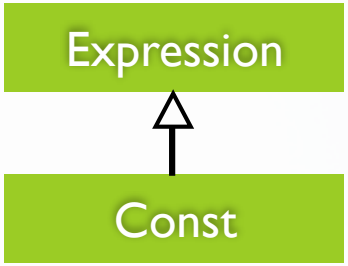
```
class Expression {
    string  _description;
public:
    Expression(const string& description="");

    string  getDescription() const;
    void    setDescription(const string& description);
    void    print() const { cout << getDescription(); }

protected:
    static unsigned long   _count;
public:
    static unsigned long getInstanceCount();
};
```

représentation textuelle d'une expression arithmétique

accesseur, mutateur et méthode pour l'affichage

attributs et méthodes de classe pour compter le nombre d'instances de la classe `Expression`

Héritage
> Héritage simple



- On souhaite créer une classe fille **`Const`**, qui nous permet de représenter un expression constante (valeur numérique)
  - une constante est une expression : on hérite donc de la classe de base `Expression`

le mot public signifie que **l'héritage est public** : l'accessibilité des membres de la classe de base est conservée à l'identique

```
// fichier const.h
#include "expression.h"

class Const : public Expression {
protected:
   double    _value;

public:
   Const(const double value=0.0);
   ~Const();

   double eval() const; // retourne sa valeur
};
```

```
class Expression {
    string  _description;
public:
    Expression(const string& description="");

    string  getDescription() const;
    void    setDescription(const string& description);
    void    print() const;

protected:
    static unsigned long   _count;
public:
    static unsigned long getInstanceCount();
};
```

La classe `Const` est une classe dérivée de `Expression`

```
class Const : public Expression {
protected:
   double   _value;

public:
   Const(const double value=0.0);
   ~Const() {}

   double eval() const;
};
```

les attributs et méthodes hérités n'apparaissent pas dans la définition de la classe fille mais elles **sont implicitement disponibles**

la méthode `eval()` est une nouvelle méthode

Héritage
> Constructeurs / destructeurs

- Si la classe `B` dérive de la classe `A`
  - il faut **d'abord initialiser `A`** (en utilisant le bon constructeur) **puis compléter l'initialisation des attributs spécifiques de `B`** : le constructeur de `A` (soit par défaut, soit spécifié) est appelé avant celui de `B`

```
B::B(int i, float f) : A(), _attr1(i), _attr2(f) {}
```

  - on peut choisir le contructeur de la classe de base `A` avant l'initialisation des attributs de `B`

```
B::B(int i, float f) : A(i), _attr2(f) {}
```

  - **le destructeur de `A` est automatiquement appelé après le destructeur de `B`** (inutile de le spécifier)

Définition de la classe `Const`

```
// fichier const.h

class Const : public Expression {
protected:
    double  _value;
public:
    Const(const double value=0.0);

    double  eval() const;
};


// fichier const.cc

Const::Const(const double value) : Expression("CSTE"), _value(value)
{
    // rien : inutile de faire value = _value
}

double Const::eval() const { return _value; }
```

on appelle un constructeur de la méthode de base `Expression` **pour initialiser les attributs hérités**

on initialise les nouveaux attributs de la classe Const

les méthodes héritées sont les méthodes publiques ou protégées de la classe `Expression`, donc déjà définies

Utilisation de la classe `Const`


```
// fichier main.cc

#include <iostream>
#include "const.h"

int main (int argc, char * const argv[])
{
    Const cste(12.3);

    cout << "description = " << cste.getDescription() << endl;

    cout << "value = " << cste.eval() << endl;

    cste.print();

    return 0;
}
```

description vient par héritage
de la classe `Expression`

en sortie

```
> description = CSTE
> value = 12.3
> CSTE
```

Héritage
> Redéfinition de méthode

- Exemple : modifier getDescription pour une constante

```
// fichier const.h

class Const : public Expression {
protected:
    double  _value;
public:
    Const(const double value=0.0);

    double  eval() const;
    string  getDescription() const {
       return  double2string(_value);
    }
};
```

on redéfinit la méthode `getDescription()` **de la classe** `Expression` **avec la même signature :** elle retourne alors la valeur constante sous forme de `string`

Utilisation de la classe `Const`


```
// fichier main.cc

#include <iostream>
#include "expression.h"

int main (int argc, char * const argv[])
{
    Const cste(12.3);

    cout << "description = " << cste.getDescription() << endl;

    cout << "value = " << cste.eval() << endl;

    cste.print();

    return 0;
}
```

on obtient bien maintenant
la valeur de la constante

en sortie

```
> description = 12.3
> value = 12.3
> CSTE
```

mais `print` affiche toujours CSTE
sauriez-vous dire pourquoi ?

Chemin d'appels pour `print()`

la méthode print appelée est celle de la classe **`Expression`** : elle appelle donc la méthode **`print()`** de la même classe, qui affiche **`CSTE`**

```
Const cste(12.3);
cste.print();
```

```
class Expression {
    string  _description;
public:
    Expression(const string&);

    string  getDescription() const;

    void    print() const;
};
```

→ **«CSTE»**

héritage

```
class Const {
    double  _value;
public:
    Const(const double);

    string  getDescription() const;

};
```

→ **`string(_value)`**

123

Héritage
> Constructeurs de copie

- Si la classe `B` dérive de la classe `A`, trois cas se présentent :
  - aucun constructeur de copie n'est défini dans `B` : les attributs spécifiques de `B` sont copiés membre à membre et la "partie" héritée de A est traitée comme un membre de type `A`. Le constructeur de copie de A (défini par l'utilisateur, sinon par défaut) sera appelé !
  - `B([const] B& x)` : il y a appel au constructeur par défaut de `A` défini par l'utilisateur (s'il n'existe pas il y a erreur de compilation). Il ne faut pas oublier d'affecter les attributs de la partie `A` avec ceux de l'objet copié.
  - `B([const] B& x) : A(...)` : il y a appel au constructeur de `A` spécifié.

Constructeurs de copie

> Exemple


```
// fichier const.h
class Const : public Expression {
protected:
   double  _value;
public:

   Const(const Const& c) : Expression(c), _value(c._value) {}

   string  getDescription() const;
   double  eval() const;
};
```

`c` est passé en paramètre du constructeur de copie `Expression` car c'est aussi une `Expression` !

on initialise `c2` comme une copie de `c1` : c'est le constructeur de copie qui est appelé

```
// fichier main.cc
int main (int argc, char * const argv[])
{
    Const c1(12.3);
    cout << "c1 = " << c1.eval() << endl;

    Const c2(c1);
    cout << "c2 = " << c2.eval() << endl;

    Expression exp(c2);
    cout << "exp = " << exp.getDescription() << endl;

    return 0;
}
```

c2 est une aussi expression : c'est le constructeur de copie de la classe `Expression` qui est appelé

en sortie

```
> c1 = 12.3
> c2 = 12.3
> exp = CSTE
```

Héritage
> Note à propos des `friends`

- **Lorsqu'une classe dérivée `B` (de `A`) déclare une fonction amie `f()`**, celle-ci n'a pas les autorisations d'accès qu'ont les fonctions membres de `B`. En particulier, **`f` n'a pas accès aux membres protégés de la classe de base `A`**
- **Une déclaration d'amitié ne s'hérite pas** : si `B` dérive de `A` et que la fonction `f()` est déclarée amie de `A`, `f` n'est pas amie de `B` !

```
class A {
   int x;
   friend void f(int t);
};

class B : public A {
   friend void g(int u);
   int y;
};
```

```
void f(int t) {
   B b;
   b.x = 1; //OK : x attribut de A
   b.y = t; //acces interdit
}
void g(int u) {
   B b;
   b.y = 2; //OK : y attribut de B
   b.x = 3; //acces interdit
}
```

Héritage
> Modes de dérivation

- Une dérivation peut être : **`public`**, **`protected`** ou **`private`**
  - ces modes changent les accès aux membres de la classe de base via la classe dérivée
  - par défaut (aucune mention du mode), C++ choisit le mode **`private`**
  - exemples :

```
class A { ... };

class B : A { ... }; <=> class B : private A { ... };

class C : protected A { ... };
```

Modes de dérivation

```
> class B : public A
```

Attributs de la classe de base	Accès dans une classe dérivée	Accès pour un client de la classe dérivée	Nouveau statut dans la classe dérivée
public	OUI	OUI	public
protected	OUI	NON	protected
private	NON	NON	private

Modes de dérivation

```
> class B : protected A
```

Attributs de la classe de base	Accès dans une classe dérivée	Accès pour un client de la classe dérivée	Nouveau statut dans la classe dérivée
public	OUI	NON	protected
protected	OUI	NON	protected
private	NON	NON	private

Modes de dérivation

```
> class B : private A
```

Attributs de la classe de base	Accès dans une classe dérivée	Accès pour un client de la classe dérivée	Nouveau statut dans la classe dérivée
public	OUI	NON	private
protected	OUI	NON	private
private	NON	NON	private

Modes de dérivation > Ajustements d'accès

- Lors d'un héritage protégé ou privé, **on peut préciser que certains membres de la classe de base conservent leur mode d'accès dans la classe dérivée**
  - attention, cela **ne permet pas d'augmenter ou de diminuer la visibilité** d'un membre de la classe de base, seulement de le conserver

```
class A {
public:
   void f1();
   void f2();
protected:
   void f3();
   void f4();
};
```

```
class B : private A {
public:
   A::f1; //f1() reste public dans B
   A::f3; //ERROR : un membre protégé ne peut devenir public
protected:
   A::f2; //ERROR : un membre public de peut devenir protégé
   A::f4; //f4() reste protégé dans B
};
```

on indique que la méthode `f4` de `A` conserve son accès protégé malgré l'héritage "privé" de `A` par `B`

Ajustements d'accès

> Exemple


```
// fichier const.h
class Const : public Expression {
protected:
    double  _value;
public:
    Const(const double value=0.0);
    Const(const Const& c);

    string  getDescription() const;
    double  eval() const;
protected:
    Expression::setDescription;
};
```

```
// fichier const.cc
int main (int argc, char * const argv[]) {
    Const c1(12.3);
    cout << "c1 = " << c1.eval() << endl;

    Const c2(c1);
    cout << "c2 = " << c2.eval() << endl;

    c2.setDescription("TOTO"); // ERROR : 'void Expression::setDescription(const
std::string&)' is inaccessible
    return 0;
}
```

on ne peut plus appeler la méthode `setDescription()` appliquée à un objet de classe `Const` car elle est devenue protégée !

Héritage
> Conversion de type

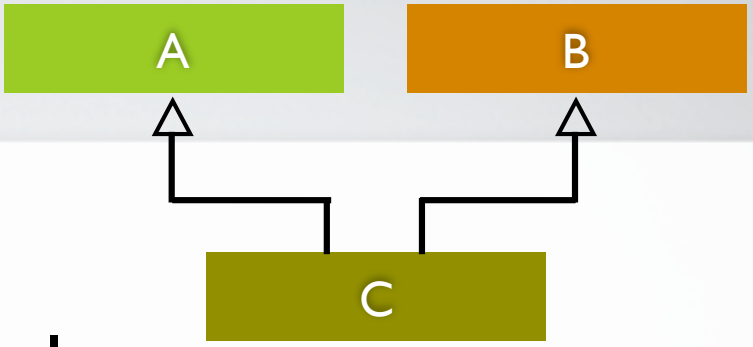
- On peut **convertir implicitement** une instance (via objet, référence ou pointeur) d'une classe dérivée en une instance de la classe de base **si l'héritage est public**
- **L'inverse est interdit** (comment initialiser les membres de la classe dérivée ?)

```
Expression   exp, *p_exp;
Const        cste, *p_cste;
...
exp = cste;       // OK !
p_exp = p_cste;   // OK !
cste = exp;       // ERROR !
p_cste = p_exp;   // ERROR !
p_cste = (Const*)p_exp;  // OK seulement si p_exp pointe vers Const !
```

Héritage
> Opérateur d'affectation

- Si la classe `B` dérive publiquement de la classe `A`
  - **si `B` ne redéfinit pas l'opérateur =**, **l'affectation** de deux objets de type `B` **se déroule membre à membre en considérant que la partie héritée de `A` est un membre**. Cette partie est alors traitée par l'opérateur d'affectation prévue dans `A` (redéfini ou par défaut)
  - **si `B` redéfinit l'opérateur =**, l'affectation de deux objets de `B` fait appel à cette fonction : **il faudra** donc **prendre en charge tout ce qui concerne l'affectation d'objet de type `A`**

Héritage
> Héritage multiple



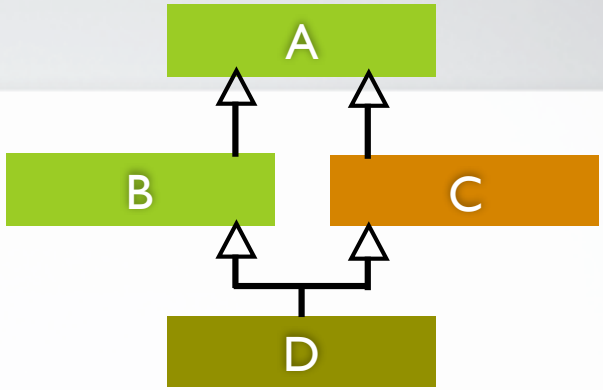
- Dériver une classe à partir de plusieurs classes de base
  - pour chaque classe de base on peut définir un mode d'héritage
  - exemple : `C` dérive de `A` et de `B`

la classe `C` dérive à la fois de `A` (protégée) et publiquement de `B`

```
class C : protected A, public B {
protected:
   int    *_attr_c;

public:
   C(bool b, int *p_i) : A(), B(b), _attr_c(p_i) {}
   ~C() {}

   ...
};
```

les constructeurs des membres dérivés sont appelés dans l'ordre d'héritage

Héritage
> Héritage virtuel



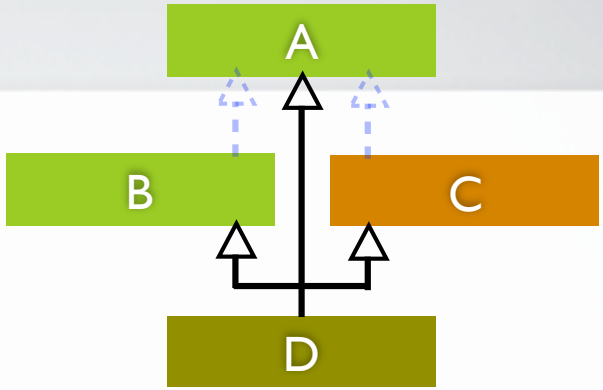
- Problème lorsqu'une classe hérite deux fois d'une autre

```
class A { int _x, _y; };
class B : public A { ... };
class C : public A { ... };

class D : public B, public C { ... };
```

- Les membres de la classe `A` sont dupliqués dans tous les objets de classe `D` (problème du losange)
  - on peut n'incorporer qu'une seule fois les membres de A dans D

```
class B : public virtual A { ... };
class C : private virtual A { ... };

class D : protected B, public C { ... };
```

Héritage
> Héritage virtuel



- Si A a été déclarée “virtuelle” dans B et C, un seul objet de A est construit : quels arguments transmettre au constructeur (de A) ?
  - dans ce cas, on peut exceptionnellement spécifier dans le constructeur de D des informations destinées à A

```
class A {
   int _x, _y;
};

class B : public virtual A { double _z; };
class C : public virtual A { };

class D : public B, public C {
public:
   D(int a, int b, double z) : B(a,b,z), A(a,b) {}
};
```