

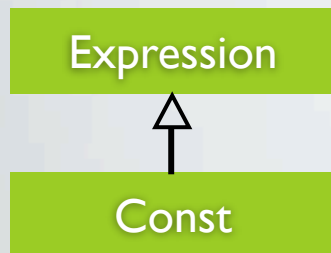
Héritage, héritage multiple

- 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

Héritage

> Héritage simple

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



```
// 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
};
```

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

```
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();
};
```

■ Héritage,
■ héritage multiple
■ > Exemple

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

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

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

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

    double eval() const;
};
```

la méthode **eval()** est
une nouvelle méthode

129

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 constructeur 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;  
};
```

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

// 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 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 "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;
```

```
}
```

en sortie

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

description vient par héritage
de la classe Expression

■ Héritage,
■ héritage multiple
■ > Exemple

132

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;
}
```

en sortie

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

mais print affiche toujours CSTE
sauriez-vous dire pourquoi ?

■ Héritage,
■ héritage multiple
■ > Exemple

on obtient bien maintenant
la valeur de la constante

134

Chemin d'appel pour print

■ Héritage,
■ héritage multiple
■ > Exemple

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

```
class Expression {  
    string _description;  
public:  
    Expression(const string&);  
  
    string getDescription() const;  
    void print() const;  
};
```

«CSTE»

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

string(_value)

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

135

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.

```
// 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 !

```
// 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 = " << e.getDescription() << endl;
```

```
    return 0;
}
```

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

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
```

137

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

Dans 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`

Dans 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`

Dans 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 ne 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::str
    return 0;
}
```

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

Héritage,
héritage multiple

144

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 !
```

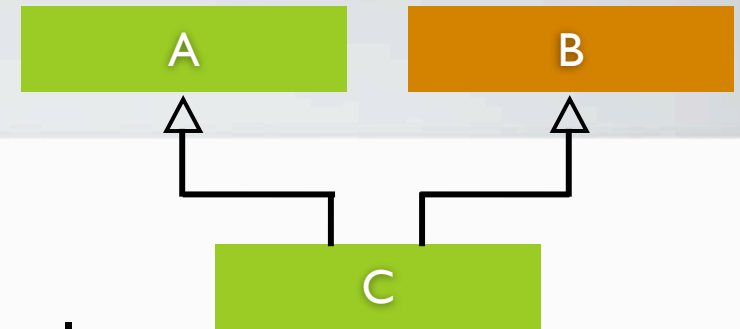
Héritage

> 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 (redéfinie 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

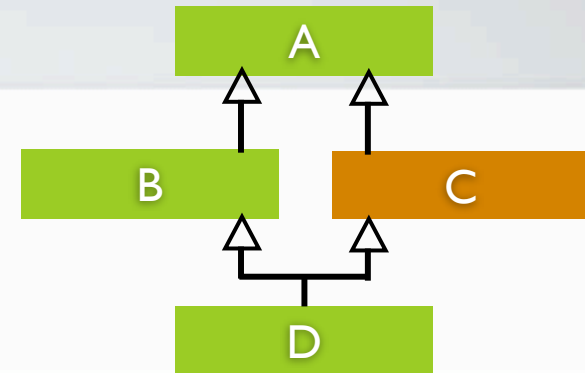
```
class C : protected A, public B {  
protected:  
    int *_attr_c;  
  
public:  
    C(bool b, int *p_i) : A(), B(b), _attr_c(p_i) {}  
    ~C() {}  
  
    ...  
};
```

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

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 { ... };
```

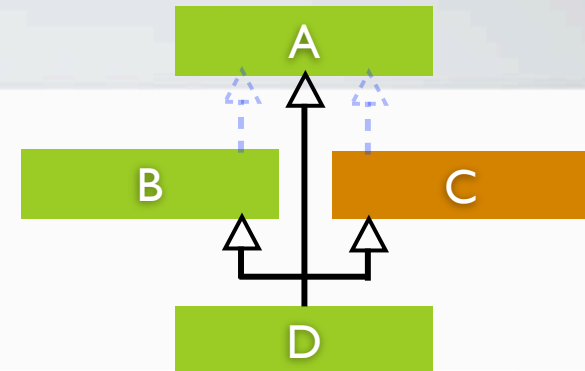
- Les membres de la classe **A** sont dupliqués dans tous les objets de classe **D** (problème du losange)

- on peut n'incorporer qu'une seule fois les membres de **A** dans **D**

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

Polymorphisme et méthodes virtuelles

- L'**héritage** permet de réutiliser le code écrit pour la classe de base dans les classes dérivées
- Le **polymorphisme** permet de choisir dynamiquement (à l'exécution) une méthode en fonction de la classe effective de l'objet sur lequel elle s'applique
 - **le choix de la méthode s'appelle liaison dynamique**
- En C++, un pointeur peut recevoir l'adresse de n'importe quel objet de classe descendante mais la méthode choisie est systématiquement celle du type pointé
 - **par défaut, C++ réalise une liaison statique, déterminée au moment de la compilation**

Liaison statique

> Exemple (1)

- Ajout de la méthode eval à la classe Expression

```
class Expression {  
    // attributs (privés par défaut)  
    string _description;  
public:  
    double eval() { return 0.0; }  
    string getDescription() const;  
    void setDescription(const string& description);  
    void print() const;  
};
```

on suppose qu'une expression
est évaluée à 0.0 par défaut

Classe Const : *rappel*

```
class Const : public Expression {  
  
protected:  
    double _value;  
  
public:  
    Const(const double value) : Expression("CSTE"), _value(value) {}  
    Const(const Const& c) : Expression("CSTE"), _value(c._value) {}  
    string getDescription() const {  
        return string_from_double(_value);  
    }  
    double eval() const { return _value; }  
};
```

On crée la classe Binary (opérateur binaire générique) qui hérite de la classe Expression (car c'est une expression !)

```
class Binary : public Expression {  
  
protected:  
    Expression *_left, *_right; // deux sous-expressions  
  
public:  
    Binary(const Expression *l,  
           const Expression *r,  
           const string& desc="OPBIN") : Expression(desc), _left(l), _right(r) { }  
  
    string getDescription() const {  
        // retourne la concaténation «( _left OPBIN _right )»  
        return "(" + _left->getDescription() + " "  
            + Expression::getDescription() + " "  
            + _right->getDescription() + ")";  
    }  
};
```

on redéfinit la méthode
getDescription pour qu'elle
retourne «(opleft OPBIN opright)»
par concaténation des chaînes de
caractères

on redéfinit la méthode
getDescription pour qu'elle
retourne «(opleft OPBIN opright)»
par concaténation des chaînes de
caractères

153

On crée la classe Plus (addition) qui hérite de la classe Binary (car c'est un opérateur binaire !)

```
class Plus : public Binary {  
public:  
    Plus(const Expression *l,  
         const Expression *r) : Binary(l, r, "+") { }  
  
    double eval() const { return l->eval() + r->eval(); }  
};
```

```
int main () {  
  
    Const c1(12.3), c2(2.4);  
  
    Plus add(&c1, &c2);  
  
    cout << "add desc = " << add.getDescription() << endl;  
    cout << "add = " << add.eval() << endl;  
  
    return 0;  
}
```

évaluer une addition consiste simplement à sommer l'évaluation de ses deux sous-expressions gauche et droite

en sortie

```
> add desc = ( CSTE + CSTE )  
> add = 0
```



154

Pourquoi la somme est-elle égale à 0 et non à 14.7 ???

`_left` et `_right` sont considérés
comme des instances de la classe
`Expression` et non de la classe
`Const`

```
class Expression {  
    string _description;  
public:  
    double eval() { return 0; }  
};
```

```
class Binary {  
protected:  
    Expression *_left, *_right;  
public:  
    // pas de méthode eval  
};
```

```
class Plus {  
public:  
    double eval() {  
        return _left->eval()+_right->eval();  
    }  
};
```

```
Const c1(12.3), c2(3.4);  
Plus add(&c1, &c2);  
add.eval();
```

0

0 + 0

155

Polymorphisme et méthodes virtuelles

> Liaison dynamique

- En C++, on peut demander une liaison dynamique en définissant des méthodes virtuelles

- dans notre cas, il suffit de déclarer la méthode `eval()` virtuelle :

```
class Expression {  
    virtual double eval() const{  
        return 0.0;  
    }  
};
```

- **ce mot clé précise au compilateur qu'il ne doit pas déterminer la méthode à appeler au moment de la compilation**
- **il mettra un dispositif en place pour déterminer au moment de l'exécution quelle méthode `eval()` appeler, en fonction du type exact de l'objet**

à l'exécution, la machine va chercher la méthode `eval()` de la classe réelle de `_left` et `_right`, c-à-d. `Const`, car `eval()` est virtuelle

```
class Expression {  
    string _description;  
public:  
    virtual double eval() { return 0; }  
};
```

```
class Binary {  
protected:  
    Expression *_left, *_right;  
public:  
    // pas de méthode eval  
};
```

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

```
class Plus {  
public:  
    double eval() {  
        return _left->eval()+_right->eval();  
    }  
};
```

```
Const c1(12.3), c2(2.4);  
Plus add(&c1, &c2);  
add.eval();
```

0

12.3 + 2.4
157

en sortie

```
> add desc = ( CSTE + CSTE )  
> add = 14.7
```



Polymorphisme et méthodes virtuelles

> Destructeurs

- Lorsque'on définit une classe destinée à être dérivée, **il faut obligatoirement déclarer son destructeur virtuel**
- cela permet d'assurer une liaison dynamique au niveau du destructeur
- sans cela, la libération mémoire serait incomplète

```
Plus *p = new Plus(&exp1,&exp2);
```

```
p->print();
```

```
...
```

```
Expression *e = p; //<=> e = m;
```

```
...
```

```
delete p;
```

si le destructeur de `Expression` n'est pas déclaré virtuel, celui de `Plus` ne sera pas appelé à cette instruction !

```
class Expression {  
    virtual ~Expression();  
};
```

Polymorphisme et méthodes virtuelles

> Méthodes virtuelles pures

- Dans notre système, une `Expression` générique n'a pas de raison d'avoir une valeur !
- la méthode `eval()` ne devrait rien retourner...
- Une solution consiste à déclarer la méthode `eval()` comme méthode virtuelle pure (**sans implémentation**)

```
// fichier expression.h
```

```
class Expression {  
    ...  
public:  
    virtual double eval() const = 0;  
};
```

en ajoutant `'= 0'` à la fin d'une déclaration de méthode, **on indique qu'elle est virtuelle pure**

Polymorphisme et méthodes virtuelles

> Classe abstraite

- Une classe qui comporte au moins une méthode virtuelle pure est dite classe abstraite
 - on ne peut pas instancier une classe abstraite
 - mais on peut utiliser des pointeurs ou des références sur une classe abstraite
- Une méthode virtuelle pure (dans une classe de base abstraite) doit obligatoirement être redéfinie dans une **classe dérivée**. Sans cela, la classe dérivée restera également abstraite

Classe abstraite

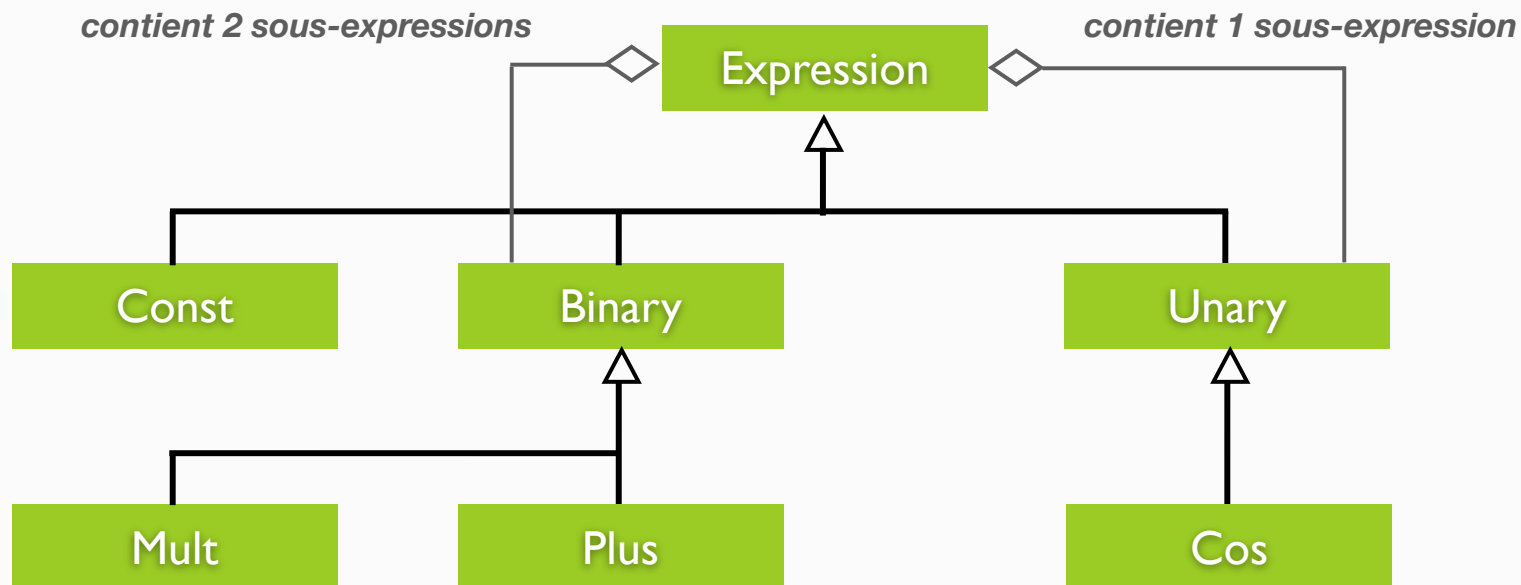
> Exemple (Expression)

- La classe Expression est abstraite

```
int main (int argc, char * const argv[]) {  
  
    Expression e("Titi");           // IMPOSSIBLE (classe abstraite)  
  
    e.setDescription("Toto");  
    e.print();  
  
    Expression *pe;                  // OK : c'est un pointeur (pas une instance)  
    pe = new Expression("Chapeau pointu"); // NON (classe abstraite)  
  
    Const c1(0.5), c2(23.1); // OK : classes dérivée non abstraites  
  
    pe = new Plus(&c1,&c2); // OK : pe est un pointeur de classe mère  
  
    cout << pe->eval() << endl; // OK : quelle méthode eval() appelée ???  
}
```

Synthèse

> Diagramme de classe de l'interpréteur



**Modélisation
objet de
l'interpréteur
> v1.0**

162

Synthèse

> Classes Expression et Const

```
class Expression {
    string _description;

public:
    Expression(const string& desc="EXP") : _description(desc) {}
    Expression(const Expression& exp) : _description(exp._description) {}
    virtual ~Expression() {}

    virtual string getDescription() const { return _description; }
    virtual void print() const { cout << this->getDescription() << endl; }

    virtual double eval() const = 0;
};
```

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

public:
    Const(const double value=0.0) : Expression("CSTE"), _value(value) {}
    Const(const Const& c) : Expression(c), _value(c._value) {}

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

■ Polymorphisme
■ et méthodes
■ virtuelles

Expression est une **classe abstraite** : elle contient une méthode virtuelle pure (sans implémentation)

la classe **Const** n'est pas abstraite : elle implémente la méthode virtuelle pure **eval()** de sa classe mère

163

Synthèse

> Classes Unary et Cos

```
class Unary : public Expression {
protected:
    const Expression *_op; // one operand
public:
    Unary(const Expression *op,
          const string& desc = "UN") : Expression(desc), _op(op) {}
    virtual ~Unary() { delete _op; }

    string getDescription() const {
        return Expression::getDescription() + "("
            + _op->getDescription() + ")";
    }
};
```

```
class Cos : public Unary {
public:
    Cos(const Expression *op) : Unary(op, "cos") {}

    double eval() const { return cos(_op->eval()); }
};
```

■ Polymorphisme
■ et méthodes
■ virtuelles

164

Synthèse

> Classes Binary, Plus et Mult

```
class Binary : public Expression {
protected:
    const Expression *_left, *_right; // two sub-expressions
public:
    Binary(const Expression *l, const Expression *r,
           const string& desc = "BIN") : Expression(desc), _left(l), _right(r) {}
    virtual ~Binary() { delete _left; delete _right; }

    string getDescription() const {
        return "(" + _left->getDescription() + " "
            + Expression::getDescription() + " "
            + _right->getDescription() + ")";
    }
};

class Mult : public Binary {
public:
    Mult(const Expression *l, const Expression *r) : Binary(l, r, "*") {}

    double eval() const { return _left->eval() * _right->eval(); }
};

class Plus : public Binary {
public:
    Plus(const Expression *l, const Expression *r) : Binary(l, r, "+") {}

    double eval() const { return _left->eval() + _right->eval(); }
};
```

■ Polymorphisme
■ et méthodes
■ virtuelles

la classe **Binary** est aussi abstraite car elle n'implémente pas la méthode **eval()** !

Plus et **Mult** redéfinissent la méthode **eval()** : elles combinent les résultats de l'évaluation de leurs sous-expression

165

Synthèse

> Fonction principale

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

int main (int argc, char * const argv[]) {
    Plus exp(new Const(5.0),
             new Mult(new Const(2.0), new Cos(new Const(M_PI))));

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

    cout << "exp.eval() = " << exp.eval() << endl;

    return 0;
}
```

en sortie

```
> exp = (5 + (2 * cos(3.14159)))
> exp.eval() = 3
```

Polymorphisme
et méthodes
virtuelles

166