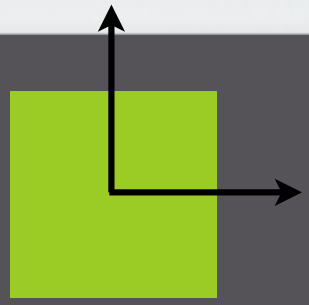
Polymorphisme et méthodes virtuelles

- 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**



Carré

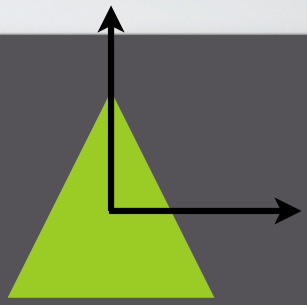
Interface

dessiner()
dessiner(couleur : entier)

Implémentation

x, y, L : réel

```
dessiner() {
    allerÀ(x-L/2, y-L/2)
    tracerVers(x+L/2, y-L/2)
    tracerVers(x+L/2, y+L/2)
    tracerVers(x-L/2, y+L/2)
    tracerVers(x-L/2, y-L/2)
}
```



Triangle (équilatéral)

Interface

dessiner()
dessiner(couleur : entier)

Implémentation

x, y, L : réel

```
dessiner() {
    allerÀ(x-L/2, y-Ltan(π/6)/2)
    tracerVers(x+L/2, y-Ltan(π/6)/2)
    tracerVers(x, y+L/2cos(π/6))
    tracerVers(x-L/2, y-Ltan(π/6)/2)
}
```

Héritage & polymorphisme > Liaison dynamique

- Une instance de classe dérivée est aussi une instance de la classe de base correspondante
  - on peut la manipuler comme telle
  - on peut affecter un objet dérivé à un objet de la classe de base
  - on peut appeler les méthodes de la classe de base
- Cas d'une méthode redéfinie d'un objet manipulé comme une instance de la classe de base
  - comportement par défaut : le code de la classe de base est appelé
  - méthodes virtuelles : le code redéfini (classe dérivée) est appelé

Polymorphisme > Définition

- Propriété qui permet à différentes classes de partager une interface commune
  - deux classes peuvent avoir une ou **plusieurs méthodes de même nom** (une classe est un “espace nommé”)
  - deux méthodes d’une même classe peuvent avoir le même nom et des paramètres différents
  - impossible en C : deux fonctions différentes = deux noms différents
- Simplification de l’interface
  - réutilisation de conventions (*ex* : multiplication par un vecteur/matrice)

en C, on devrait appeler une fonction adaptée pour chaque type d'objet passé en paramètre : **il faut donc un moyen de tester le type de l'objet**

Langage C

```
faireJouer (void *inst, int type) {
    switch (type) {
        case VIOLON:
            jouerViolon((Violon*)inst);
            break;
        case PIANO:
            jouerPiano((Piano*)inst);
            break;
        etc...
    }
}
...
faireJouer ((void*)monPiano, PIANO);
```

Langage C++

```
faireJouer (Instrument& inst) {
    inst.jouer();
}
...
faireJouer ( monViolon );
faireJouer ( monPiano );
```

code utilisé pour la méthode "faireJouer" :

- *celui de Instrument ?*
- *celui de l'objet passé en paramètre (Violon, Piano) ?*

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 double2string(_value);

    }

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

la méthode `eval()` est redéfinie pour la classe `Const`

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 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 _left->eval() + _right->eval(); }
};
```

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

```
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;

}
```

en sortie

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

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

Liaison statique

> Exemple

```
class Expression {
    string  _description;
public:

    double  eval() { return 0; }
};
```

0

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

```
class Binary {
protected:
    Expression  *_left, *_right;
public:

    // pas de méthode eval
};
```

```
class Plus {
public:

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

0 + 0

147

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

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**

Liaison dynamique

> Exemple

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; }
};
```

→ 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();
    }
};
```

→ 12.3 + 2.4

149

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

en sortie

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

Polymorphisme et méthodes virtuelles

> Destructeurs

- Lorsqu'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;
...
delete e;
```

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

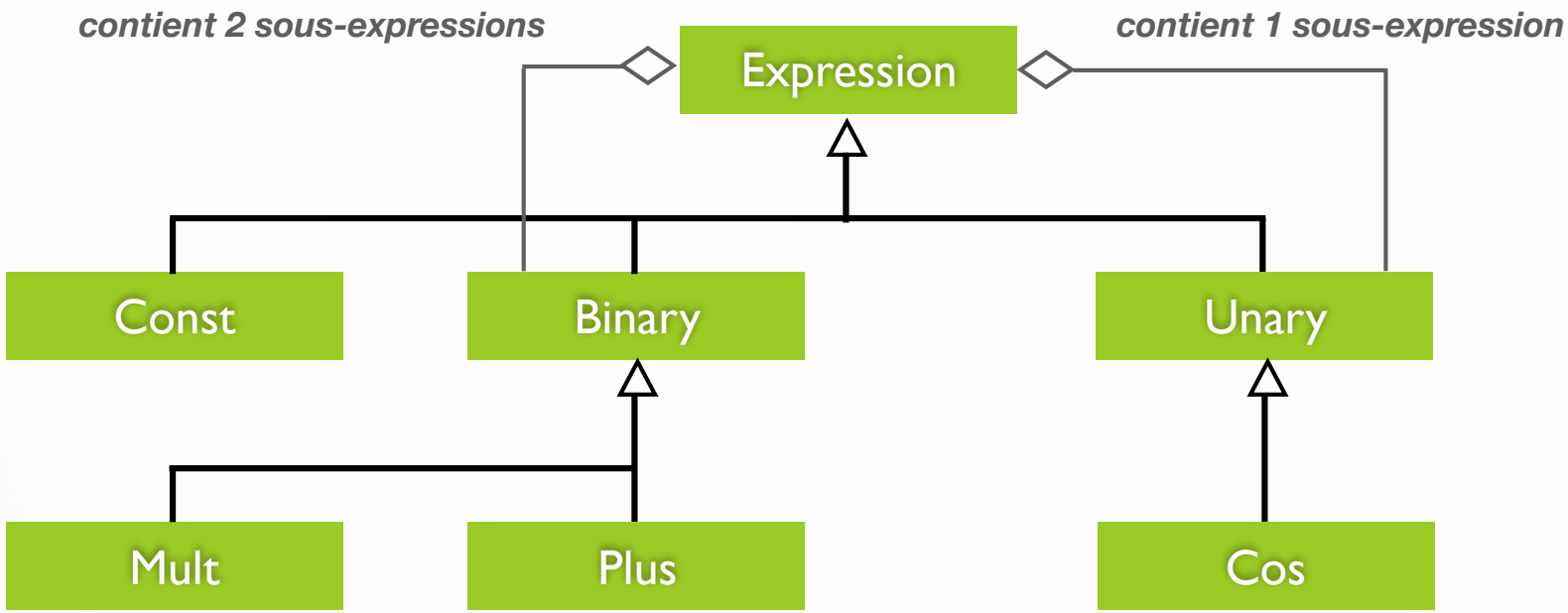
Classe abstraite
> Exemple (`Expression`)

- 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 : classe 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



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;
};
```

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

```
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; }
};
```

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

Synthèse

> Classes `Unary` et `Cos`

```
class Unary : public Expression {
protected:
    const Expression  *_op;  // une opérande (sous-expression)
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() + ")";
    }
};
```

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

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

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

la méthode `eval()` de la classe `Cos` calcule le cosinus du résultat de la méthode `eval()` appliquée à son opérande

Synthèse

> Classes `Binary`, `Plus` et `Mult`

```
class Binary : public Expression {
protected:
    const Expression  *_left, *_right;  // deux opérandes (sous-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(); }
};
```


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

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
```