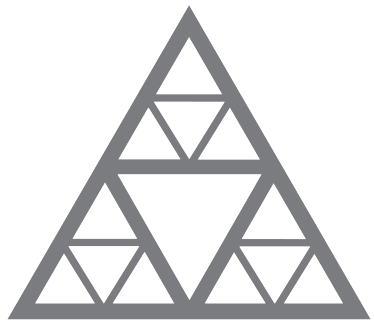
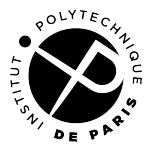
PRALG: Héritage, polymorphisme

Pascal Monasse
pascal.monasse@enpc.fr



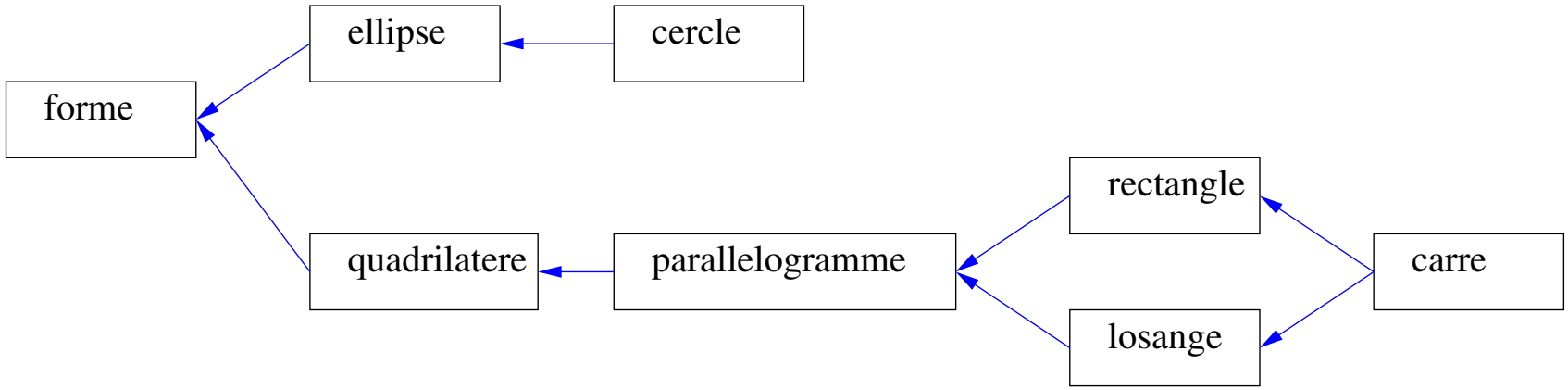
ÉCOLE NATIONALE DES
PONTS
ET CHAUSSÉES



IP PARIS

Héritage

Idée : étendre ou spécialiser une classe existante en utilisant ses données (champs) et son comportement (méthodes).




```
class Forme {...};

class Ellipse: public Forme {
  FloatPoint2 centre, extremite; //demi grand axe
  float axe; //longueur du petit axe
public:
  void transforme(float dx, float dy, float angle);
};

class Cercle: public Ellipse {...};
```

Héritage

```
class Quadrilatere: public Forme {
  FloatPoint2 sommets[4];
public:
  void transforme(float dx, float dy, float angle);
};

class Parallelogramme: public Quadrilatere {...};

class Rectangle: public Parallelogramme {...};

class Losange: public Parallelogramme {...};

class Carre: public Rectangle, public Losange {...};
```

- Classe `Carre` : héritage *multiple*, OK mais peut se révéler délicat, à éviter en général.

Héritage

```
Rectangle R;
R.tranforme(5.1f, -1.2f, float(M_PI/3));
```

- C'est `Quadrilatere::transforme` qui est appelé.
- Ce serait `Parallelogramme::transforme` s'il était défini.
- `Rectangle` a aussi le champ `FloatPoint2 sommets[4]`.
- Mais ceci ne fonctionne pas :

```
class Rectangle: public Parallelogramme {
public:
  FloatPoint2 sommet(int i) const {
    assert(0<=i && i<4);
    return sommets[i]; // Non, champ private
  }
};
```

Héritage

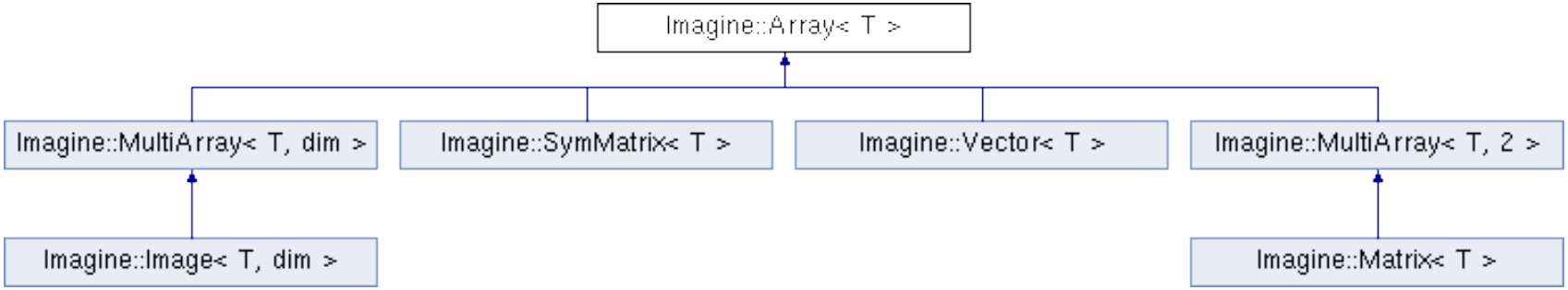
Une classe n'a pas forcément accès a tous ses champs !
Solution :

```
class Quadrilatere: public Forme {
protected:
  FloatPoint2 sommets[4];
public:
  void transforme(float dx, float dy, float angle);
};
```

- protected se comporte comme public dans une classe qui hérite, et comme private pour l'extérieur.

```
class Parallelogramme: public Quadrilatere {
public:
  FloatPoint2 centre() const {
    FloatPoint2 somme = sommets[0]; //OK
    for(int i=1; i<4; i++)
      somme += sommets[i];
    return somme/4;
  }
};
Quadrilatere Q;
Q.sommets[0].x() = 0; //Non, Q::sommets non public
Parallelogramme P;
P.sommets[0].x() = 0; //Non, P::sommets non public
```

Exemple de diagramme d'héritage



- `MultiArray` hérite de `Array` (linéarisation du tableau multi-dimensionnel). Exemple :

```
MultiArray<double,2> MA(3,3);
MA(1,0) = M_PI; // operator() de MultiArray
cout << MA[1]; // operator[] provient de Array
```

- `SymMatrix` n'hérite pas de `Matrix` :-(. Explication : `SymMatrix` ne stocke que la partie triangulaire supérieure.
- Conséquence : on doit définir Matrix*SymMatrix et SymMatrix*Matrix même si l'implémentation est identique à Matrix*Matrix.

Polymorphisme

- Pointeurs et références sur une classe fille le sont aussi sur une classe mère.

```
void f(Parallelogramme& P);
Parallelogramme P;
f(P); //Appel standard
Rectangle R;
f(R); //Utilise le polymorphisme
Forme* p = &P; //OK
f(*p); //Non
f(*(Parallelogramme*)p); //OK, mais a eviter
Parallelogramme* q =
   dynamic_cast<Parallelogramme*>(p);
if(q)
   f(*q); //Mieux
```

- L'opérateur dynamic_cast permet de retrouver le *vrai* type de l'objet pointé.

Méthode virtuelle

```
class Quadrilatere: public Forme {
  ...
public:
  virtual void transforme(float dx, float dy, float ang)
};
class Parallelogramme: public Quadrilatere {
public:
  void transforme(float dx, float dy, float ang) {
    Quadrilatere::transforme(dx, dy, ang);
    cout << "Parallelogramme!" << endl;
  }
};
void pivot(Quadrilatere& Q) {Q.transforme(0,0,M_PI/2);}
Quadrilatere Q;
pivot(Q); //Appelle Quadrilatere::transforme
Parallelogramme P;
pivot(P); //Appelle Parallelogramme::transforme
Quadrilatere* p=&P;
pivot(*p); //Appelle Parallelogramme::transforme
```

Méthode virtuelle

Le polymorphisme fonctionne aussi dans la classe de base :

```
class Quadrilatere: public Forme {
  virtual void transforme(float dx, float dy, float ang)
    ...
    cout << type() << endl;
  }
  virtual std::string type() { return "Quadrilatere"; }
};
class Parallelogramme: public Quadrilatere {
  std::string type() { return "Parallelogramme"; }
};
Quadrilatere Q;
pivot(Q); //Affiche "Quadrilatere"
Parallelogramme P;
pivot(P); //Affiche "Parallelogramme"
```

Limites

Attention :

- Le polymorphisme ne s'applique pas au type lui-même

```
void g(Quadrilatere Q) { pivot(Q); }
Parallelogramme P;
g(P); //Affiche Quadrilatere
```

Explication : g appelle le constructeur par copie
`Quadrilatere::Quadrilatere(const Quadrilatere& obj)`
avec `obj`$=P$.

- Le polymorphisme est inopérant pour la résolution d'appel des fonctions surchargées :

```
void f(const Quadrilatere& Q); //(1)
void f(const Rectangle& R);    //(2)
Quadrilatere Q;
f(Q); //Appelle (1)
Rectangle R;
f(R); //Appelle (2)
Quadrilatere* p = &R;
f(*p); //Appelle (1) et non (2)
```

Destructeur

Lors d'un delete, le destructeur est appelé. Pour qu'il s'applique au vrai type, le destructeur des classes de base doit être virtuel :

```
class Forme {
public:
  virtual ~Forme() { cout << "~Forme" << endl; }
};
class Ellipse: public Forme {
public:
  virtual ~Ellipse();
};
Ellipse::~Ellipse() { cout << "~Ellipse" << endl; }
Forme* F = new Ellipse;
delete F; //Affiche "~Ellipse ~Forme"
```

C'est important :

```
class A {
public:
  virtual ~A() {} //Ne pas omettre virtual !
};
class B: public A {
  int* tab;
public:
  B(int n) { tab = new int[n]; }
  ~B() { delete [] tab; }
};
A* b = new B(1000000);
delete b; //OK, pas de fuite memoire car appelle ~B avant ~A.
```

- Ne jamais hériter d'une classe sans destructeur virtuel (comme `std::vector`).

Constructeurs

Les constructeurs ne sont pas hérités, mais pour créer un objet de classe fille il faut créer d'abord l'objet de classe mère :

```
class Quadrilatere: public Forme {
public:
  Quadrilatere() {}
  Quadrilatere(const FloatPoint2 pts[4]);
};
Quadrilatere::Quadrilatere(const FloatPoint2 pts[4]): Forme() {...}
class Rectangle: public Quadrilatere {
public:
  Rectangle(const FloatPoint2 pts[4]): Quadrilatere(pts) {...}
};
FloatPoint2 pts[4] = {...};
Rectangle R(pts);
Rectangle S; //Non, n'existe pas
```

Omettre l'indication du constructeur de la classe mère à appeler signifie appeler le constructeur sans argument (il faut donc qu'il existe).

Méthode virtuelle pure

Désigne une fonctionnalité abstraite d'une famille, et la classe est alors abstraite (pas d'objet de ce type).

```
class Forme {
public:
  virtual ~Forme() {}
  virtual void transforme(float, float, float)=0; // Virtuelle pure
};
class Ellipse: public Forme {
public:
  ~Ellipse() {}
  void transforme(float dx, float dy, float angle) {}
};
class Quadrilatere: public Forme {
public:
  ~Quadrilatere() {}
  void transforme(float dx, float dy, float angle) {}
};
void f(Forme& F) { F.transforme(...); }
Ellipse E;
f(E); // Ellipse::transforme
Quadrilatere Q;
f(Q); // Quadrilatere::transforme
Forme F; //Non, Forme est une classe abstraite
```

“Constructeur virtuel”

On peut créer un objet du même type qu’un objet donné :

```
class Forme {
public:
  virtual ~Forme() {}
  virtual Forme* clone() const=0;
};
class Ellipse: public Forme {
public:
  Forme* clone() const { return new Ellipse(*this); }
};
class Quadrilatere: public Forme {
public:
  Forme* clone() const { return new Quadrilatere(*this); }
};
Forme* clone(Forme& F) { return F.clone(); }
Ellipse E;
Forme* Ebis = clone(E); //*Ebis est en fait une Ellipse
Quadrilatere Q;
Forme* Qbis = clone(Q); //*Qbis est en fait un Quadrilatere
```

- Notez que typiquement les méthodes `clone` appellent le constructeur par copie.

Limite : tableaux homogènes

On ne peut pas mélanger des ellipses et rectangles dans un même tableau, mais on peut y mettre des Forme*.

```
Forme F[2]; //Non, Forme est une classe abstraite
std::vector<Forme> V; //Non plus
Forme* F[2]; //OK
std::vector<Forme*> V;
F[0] = new Ellipse;
F[1] = new Rectangle;
V.push_back(new Ellipse);
V.push_back(new Rectangle);
delete F[0];
delete F[1];
for(std::vector<Forme*>::iterator it=V.begin();
    it!=V.end(); ++it)
  delete *it;
```

Limite : Pas de propagation aux paramètres template

```
void f(std::vector<Quadrilatere>& V);
std::vector<Quadrilatere> Q;
f(Q); //OK
std::vector<Rectangle> R;
f(R); //Ne compile pas
```

Limite : diverses

- L'appel d'une méthode virtuelle dans le constructeur n'appelle pas la méthode du vrai type.
  Explication : l'objet en construction n'est pas encore complet.
  Lors de la construction d'une `Ellipse`, on construit d'abord une `Forme`, mais dans le constructeur de `Forme` ce n'est pas encore une `Ellipse`.
- L'appel à une méthode virtuelle a un coût, car la méthode à appeler n'est pas connue lors de la compilation mais à l'exécution (dynamique), ce qui empêche des optimisations.
  $\rightarrow$ ne pas mettre systématiquement les méthodes virtuelles.

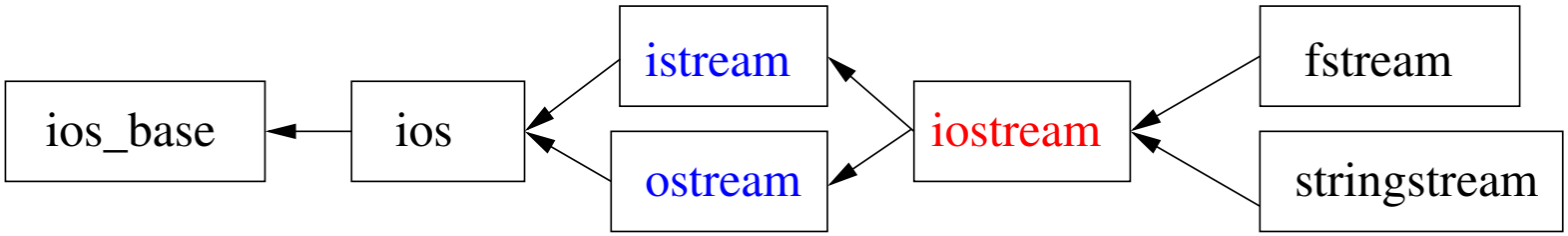
Comment ça marche

Dès qu'un type contient une méthode virtuelle, on ajoute un pointeur sur une "virtual table" (ici B:: __vtbl) qui est un tableau de pointeurs sur fonction.

```
class A {
    int a;
    bool b;
public:
    ~A() {}
};
class B {
    int a;
    bool b;
public:
    virtual ~B() {}
};
cout << sizeof(A) << '-' << sizeof(B) << endl; // 8 16
```

Exemples

- Entrées/sorties de la bibliothèque standard



- Bibliothèque QtWidgets de Qt