Héritage (suite)

Gestion des méthodes

Soient une classe de base `A` et une classe `B` qui dérive de `A` :

```
#include <iostream>
class A {
public:
  void f() {std::cerr<<"A::f"<<std::endl;}
  void g() {std::cerr<<"A::g"<<std::endl;}
};
class B : public A {
public:
  void f() {std::cerr<<"B::f"<<std::endl;}
  void h() {std::cerr<<"B::h"<<std::endl;}
};
```

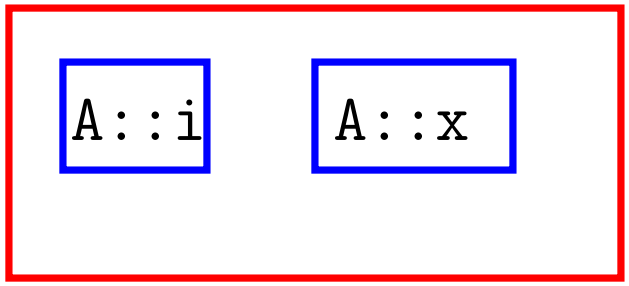
```
#include "x.h"
int main()
{
   A a;
   B b;

   a.f(); // appel A::f
   a.g(); // appel A::g

   b.f(); // appel B::f
   b.g(); // appel A::g
   b.h(); // appel B::h
   return 0;
}
```

Gestion des attributs

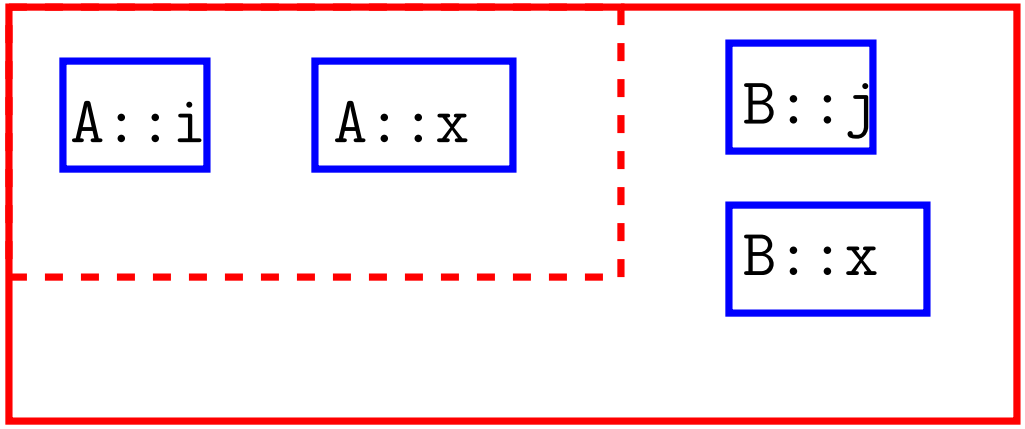
Soient une classe de base `A` et une classe `B` qui dérive de `A` :

```
#include <iostream>
class A {
 public:
   int i;
   int x;
};
class B : public A {
 public:
   int j;
   double x;
};
```

Un objet `a` de type `A` est organisé en mémoire comme suit :



Un objet `b` de type `B` est organisé en mémoire comme suit :



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

int main()
{
   A a;
   B b;
   std::cerr<<"a:"<< sizeof(a)<<std::endl;
   std::cerr<<"b:"<< sizeof(b)<<std::endl;

   a = b; // correct
   b = a; // non correct
   return 0;
}
```

`a = b` *est correct* parce qu'il y a assez d'information dans `b` pour initialiser `a`.

`b = a` *n'est pas correct* parce qu'il n'y a pas assez d'information dans `a` pour initialiser `b`.

```
#include "x.h"

void F(A aa) { }
void G(B & bb) { }

int main()
{
  A a;
  B b;
  F(a); // correct
  F(b); // correct
  G(a); // non correct
  G(b); // correct
}
```

`F(a)` et `G(b)` *sont corrects* parce la variable transmise en entrée a exactement le même type que dans la déclaration des fonctions.

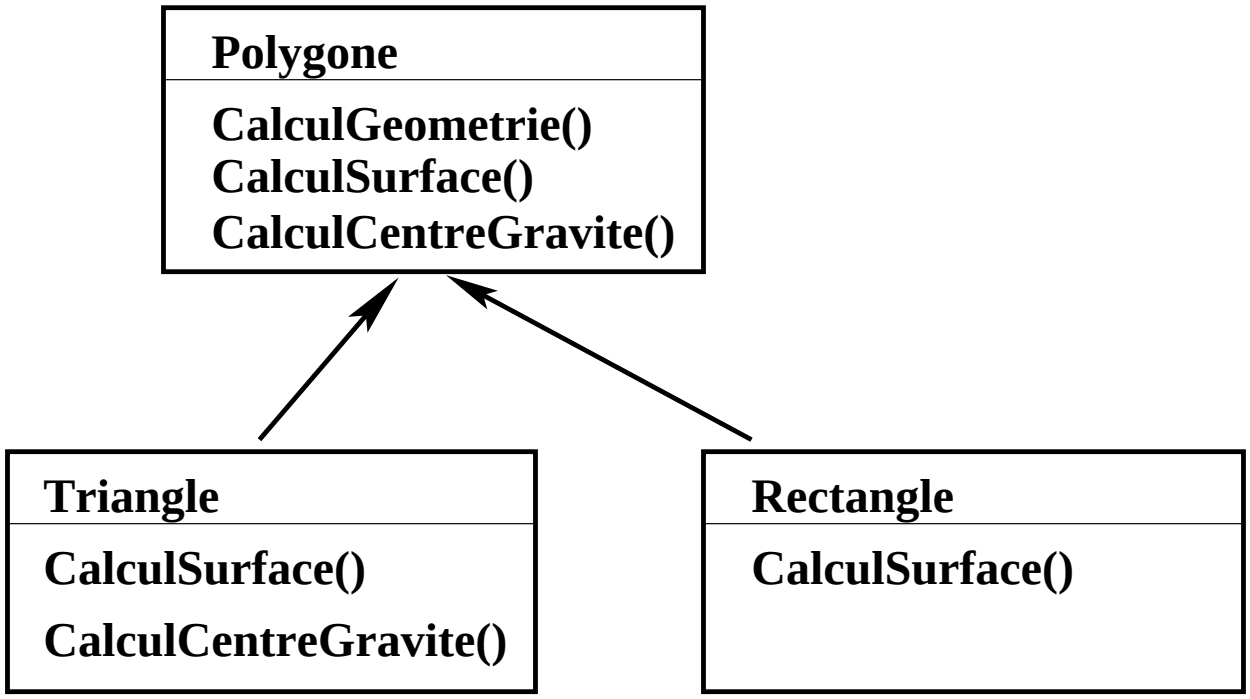
`F(b)` *est correct* parce qu'il y a assez d'information dans `b` pour exécuter `F`.

`G(a)` *n'est pas correct* parce qu'il n'y pas a assez d'information dans `a` pour exécuter `G`.

Polymorphisme

Exemple

On considère l'héritage suivant :



avec

```
void Polygone::CalculGeometrie() {
  CalculSurface();
  CalculCentreGravite();
};
```

avec

```
void Polygone::CalculGeometrie() {
  CalculSurface();
  CalculCentreGravite();
};

Rectangle R; R.CalculGeometrie();
```

avec

```
void Polygone::CalculGeometrie() {
  CalculSurface();
  CalculCentreGravite();
};

Rectangle R; R.CalculGeometrie();
```

`R` appelle `CalculGeometrie()` qui utilisera

`Polygone::CalculSurface()` et

`Polygone::CalculCentreGravite()`

même si les méthodes

`Triangle::CalculSurface()`

`Triangle::CalculCentreGravite()`

existent.

Solution proposée par C++ :

Fonctions virtuelles

Méthode d'une classe dont la déclaration dans l'interface de la classe est précédée par : virtual.

Dans une hiérarchie de classes, le choix de la méthode sera celle de la classe "réelle" d'un objet, ou de son plus proche parent en remontant l'arborescence

$\Rightarrow$ dans l'interface des différentes classes :

```
virtual void CalculSurface();
virtual void CalculCentreGravite();
```

Fonctions virtuelles : autre exemple

Soit l'équation :

$$u(t_{n+1}) = u(t_n) - (t_{n+1} - t_n) \times \sum_{i=1}^{n} O_i(u(t_n))$$

(qui s'obtient en discrétisant les dérivations dans une équation aux dérivées partielles, par exemple $\frac{\partial u}{\partial t} + (u.\nabla)u - \Delta u = 0$, avec $n = 2$ et $O_1(u) = (u.\nabla)u$, $O_2(u) = -\Delta u$)

On suppose disposer d'une classe `Champ` pour représenter u, d'une classe `Operateur` (opérateur abstrait) et d'un ensemble de classes dérivées de `Operateur` :

`LaplacienNegatif`, `Convection`, `Gradient` ...

$\Rightarrow$ on pourra simuler l'équation par un code du type

```
Champ v, dv;
Operateur **O;
O = new (Operateur*)[n];
...
// Partie spécifique à l'équation
O[0] = new Convection;
O[1] = new LaplacienNegatif;
...
// Partie générale, valable pour toutes
// équations vues à la page précédente
dv = 0;
for (i=0; i<n; i++) dv += O[i](v);
dv *= dt;
v -= dv;
```