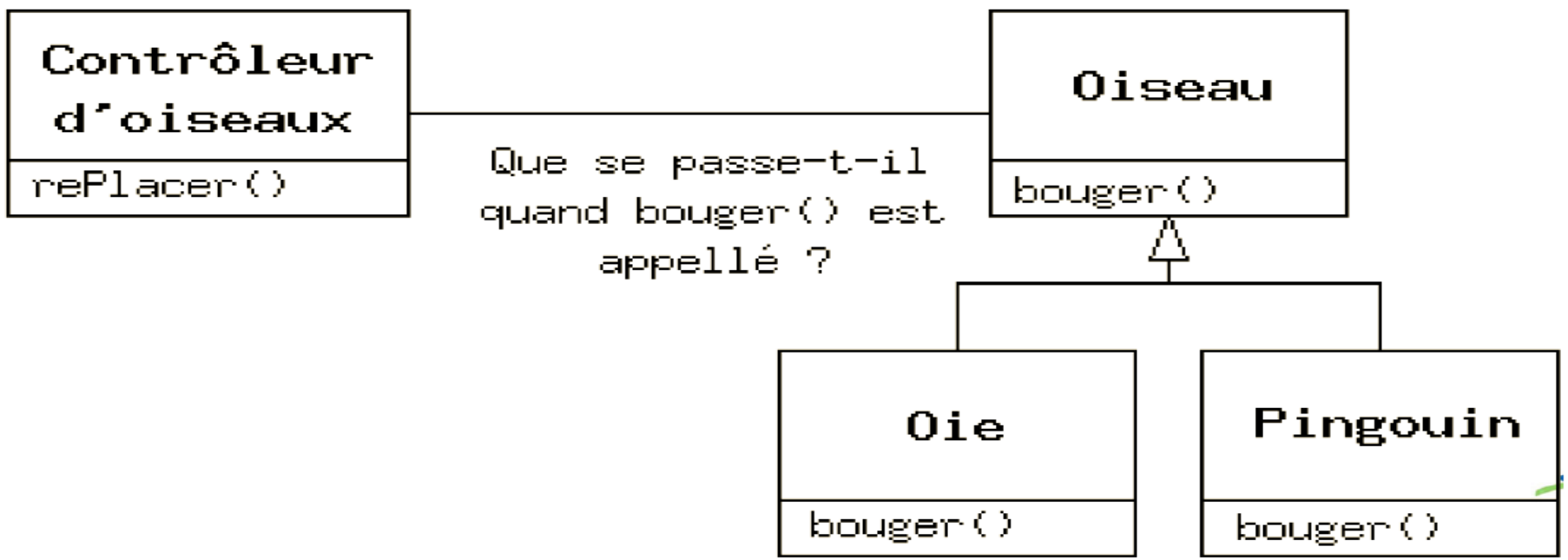
Comportement Polymorphe (1/4)

- Par exemple, dans le diagramme suivant, l'objet Contrôleur d'oiseaux travaille seulement avec des objets Oiseaux génériques, et ne sait pas de quel type ils sont. C'est pratique du point de vue de Contrôleur d'oiseaux, car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d'Oiseau avec lequel il travaille, ou le comportement de cet Oiseau.



Comportement Polymorphe (2/4)

- Comment se fait-il donc que, lorsque `bouger()` est appelé tout en ignorant le type spécifique de l'Oiseau, on obtienne le bon comportement (une Oie court, vole ou nage, et un Pingouin court ou nage) ?
- La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une association prédéfinie : le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter.
- En POO, le programme ne peut déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.

Comportement Polymorphe (3/4)

- Pour résoudre ce problème, les langages orientés objet utilisent le concept d'**association tardive**. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour, mais il ne sait pas exactement quel est le code à exécuter.
- Pour créer une association tardive, le compilateur C++ insère une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet. Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

Comportement Polymorphe (4/4)

- On déclare qu'on veut une fonction qui ait la flexibilité des propriétés de l'association tardive en utilisant le mot-clé **`virtual`**.
- On n'a pas besoin de comprendre les mécanismes de `virtual` pour l'utiliser, mais sans lui on ne peut pas faire de la programmation orientée objet en C++.
- En C++, on doit se souvenir d'ajouter le mot-clé `virtual` (devant une méthode) parce que, par défaut, les fonctions membres ne sont pas liées dynamiquement. Les fonctions virtuelles permettent d'exprimer des différences de comportement entre des classes de la même famille.
- Ces différences sont ce qui engendre un **comportement polymorphe**.

Exemple : comportement non polymorphe (1/2)

```
#include <iostream>
using namespace std;

class Forme {
   public:
      Forme() { cout << "constructeur Forme <|- "; }
      void dessiner() { cout << "je dessine ... une forme ?\n"; }
};

class Cercle : public Forme {
   public:
      Cercle() { cout << "Cercle\n"; }
      void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
   public:
      Triangle() { cout << "Triangle\n"; }
      void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

Exemple : comportement non polymorphe (2/2)

```
void faireQuelqueChose(Forme &f)
{
  f.dessiner(); // dessine une Forme
}

int main()
{
   Cercle c;
   Triangle t;

   faireQuelqueChose(c); // avec un cercle
   faireQuelqueChose(t); // avec un triangle

   return 0;
}
```

L'exécution du programme d'essai nous montre que nous n'obtenons pas un comportement polymorphe puisque c'est la méthode `dessiner()` de la classe `Forme` qui est appelée :

```
constructeur Forme <|- Cercle
constructeur Forme <|- Triangle
je dessine ... une forme ?
je dessine ... une forme ?
```

Exemple : comportement polymorphe (1/2)

```
#include <iostream>
using namespace std;

class Forme {
   public:
      Forme() { cout << "constructeur Forme <- "; }
      // la méthode dessiner sera virtuelle et fournira un comportement polymorphe
      virtual void dessiner() { cout << "je dessine ... une forme ?\n"; }
};

class Cercle : public Forme {
   public:
      Cercle() { cout << "Cercle\n"; }
      void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
   public:
      Triangle() { cout << "Triangle\n"; }
      void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

Exemple : comportement polymorphe (2/2)

```
void faireQuelqueChose(Forme &f)
{
  f.dessiner(); // dessine une Forme
}

int main()
{
   Cercle c;
   Triangle t;

   faireQuelqueChose(c); // avec un cercle
   faireQuelqueChose(t); // avec un triangle

   return 0;
}
```

L'exécution du programme d'essai nous montre maintenant que nous obtenons un comportement polymorphe puisque c'est la "bonne" méthode `dessiner()` qui est appelée :

```
constructeur Forme <|- Cercle
constructeur Forme <|- Triangle
je dessine un Cercle !
je dessine un Triangle !
```