Полиморфизм
(продолжение)

Использование виртуального наследования для решения проблемы ромба

Так что же произойдет при создании экземпляра класса P l a t y p u s ? Сколько экземпляров класса A n i m a l получится в одном экземпляре класса P l a t y p u s ? Листинг 11.7 поможет ответить на этот вопрос.

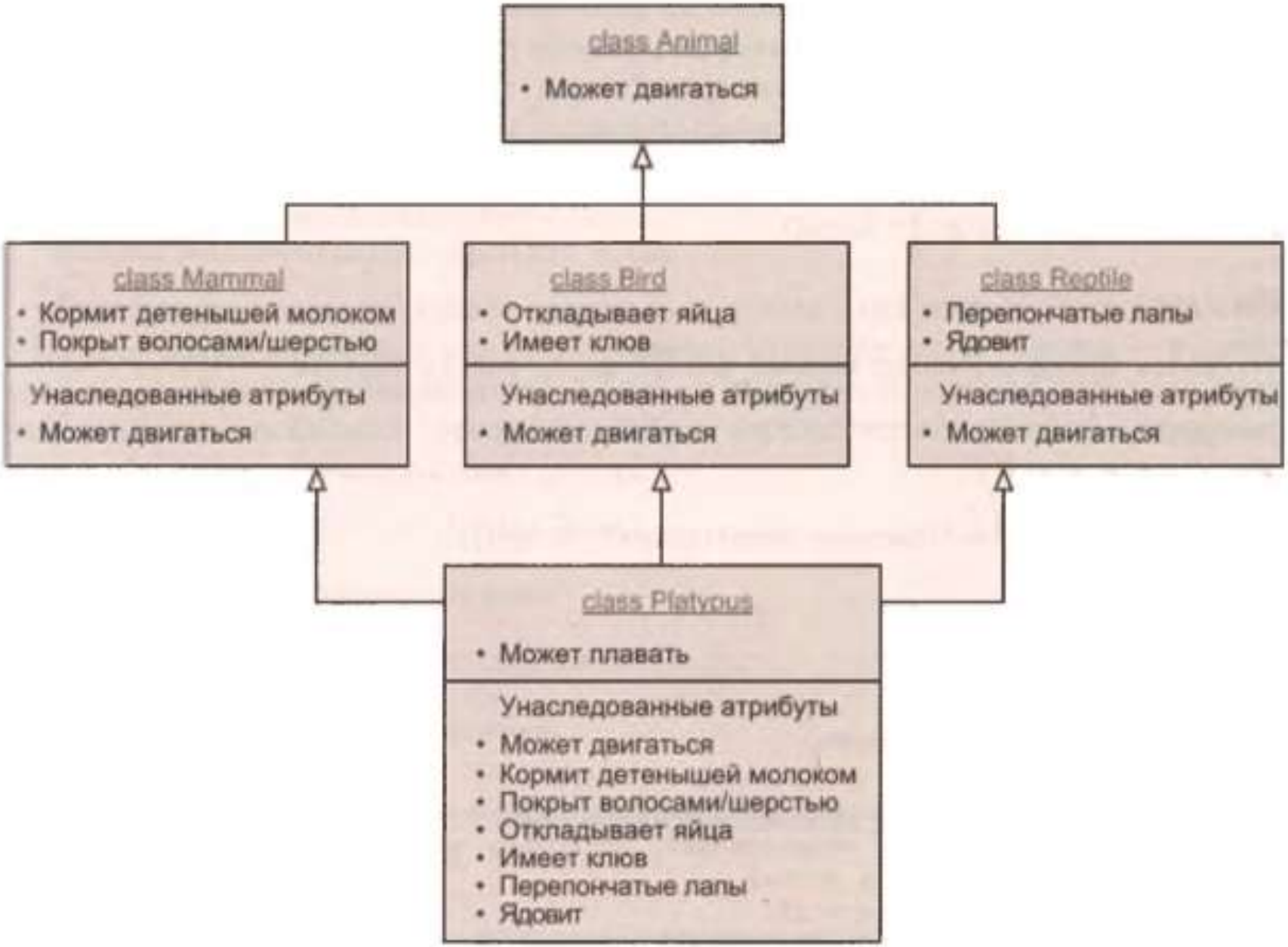

РИС. 11.2. Схема класса утконоса, демонстрирующего множественное наследование

Использование виртуального наследования для решения проблемы ромба

На занятии “Реализация наследования”, мы рассмотрели любопытный случай утконоса, который является млекопитающим, но частично и птицей, и рептилией. В этом случае класс утконоса P l a t y p u s должен происходить от классов Mammal, B i r d и R e p t i l e .

Однако каждый из них, в свою очередь, происходит от более обобщенного класса, A n i m a l (животное), как показано на рис. 11.2.

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Animal
 4: {
 5: public:
 6:     Animal()
 7:     {
 8:         cout << "Animal constructor" << endl;
 9:     }
10:
11:     // простая переменная
12:     int Age;
13: };
14:
15: class Mammal:public Animal
16: {
17: };
18:
19: class Bird:public Animal
20: {
21: };
22:
23: class Reptile:public Animal
24: {
25: };
26:
```

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

```
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29: public:
30:     Platypus()
31:     {
32:         cout << "Platypus constructor" << endl;
33:     }
34: };
35:
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // Снимите комментарий со следующей строки и получите отказ
41:     // компиляции. Age неоднозначен, поскольку есть три экземпляра
        // базового класса Animal
42:     // duckBilledP.Age = 25;
43:
44:     return 0;
45: }
```

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

Результат

```
Animal constructor
Animal constructor
Animal constructor
Platypus constructor
```

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

Анализ

Как демонстрирует вывод, благодаря множественному наследованию у всех трех базовых классов класса P l a t y p u s (происходящих, в свою очередь, от класса Animal) есть свой экземпляр класса Animal. Следовательно, для каждого экземпляра класса P l a t y p u s , как показано в строке 38, автоматически создаются три экземпляра класса Animal.

Но утконос — это одно животное, которое наследует определенные атрибуты классов Mammal, B i r d и R e p t i l e .

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

Анализ

Проблема с количеством экземпляров базового класса Animal не ограничивается только излишним использованием памяти. У класса Animal есть целочисленный член A n i m a l : : Age (который для демонстрации был оставлен открытым). При попытке получить доступ к переменной-члену A n i m a l: : Age через экземпляр класса P l a t y p u s , как показано в строке 42, вы получаете ошибку компиляции, потому что компилятор просто не знает, хотите ли вы установить значение переменной-члена Mammal: : A n i m a l: :Age,
или B i r d : : A n i m a l: : Age,
или R e p t i l e : : A n i m a l: :Age.

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

Анализ

При желании вы можете установить значения для всех трех:

```
duckBilledP.Mammal::Animal::Age = 25;
duckBilledP.Bird::Animal::Age = 25;
duckBilledP.Reptile::Animal::Age = 25;
```

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

Анализ

Безусловно, у одного утконоса должен быть только один возраст. Но все же класс P l a t y p u s должен происходить от классов Mammal, B i r d и R e p t i l e .

Решение — в *виртуальном наследовании* (virtual inheritance).

Если вы ожидаете, что производный класс будет использоваться как базовый, хорошей идеей будет определение его отношения к базовому с использованием ключевого слова v i r t u a l :

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

```
class Derived1: public virtual Base
{
    // ... переменные и функции
};
class Derived2: public virtual Base
{
    // ... переменные и функции
};
```

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса Animal в одном экземпляре класса Platypus

Улучшенный класс P l a t y p u s (фактически улучшенные классы Mammal, B i r d и R e p t i l e) приведен в листинге 11.8.

ЛИСТИНГ 11.8. Как ключевое слово **virtual** в иерархии наследования позволяет ограничить количество экземпляров базового класса **Animal** до одного

```
0: #include <iostream>
1: using namespace std;
2:
3: class Animal
4: {
5: public:
6:     Animal()
7:     {
8:         cout << "Animal constructor" << endl;
9:     }
10:
11:    // простая переменная
12:    int Age;
13: };
14:
```

ЛИСТИНГ 11.8. Как ключевое слово **virtual** в иерархии наследования позволяет ограничить количество экземпляров базового класса **Animal** до одного

```
15: class Mammal:public virtual Animal
16: {
17: };
18:
19: class Bird:public virtual Animal
20: {
21: };
22:
23: class Reptile:public virtual Animal
24: {
25: };
26:
```

ЛИСТИНГ 11.8. Как ключевое слово **virtual** в иерархии наследования позволяет ограничить количество экземпляров базового класса **Animal** до одного

```
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29: public:
30:     Platypus()
31:     {
32:         cout << "Platypus constructor" << endl;
33:     }
34: };
35:
```

ЛИСТИНГ 11.8. Как ключевое слово **virtual** в иерархии наследования позволяет ограничить количество экземпляров базового класса **Animal** до одного

```
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // нет ошибки компиляции, поскольку есть только один Animal::Age
41:     duckBilledP.Age = 25;
42:
43:     return 0;
44: }
```

ЛИСТИНГ 11.8. Как ключевое слово **virtual** в иерархии наследования позволяет ограничить количество экземпляров базового класса **Animal** до одного

Результат

```
Animal constructor
Platypus constructor
```

ЛИСТИНГ 11.8. Как ключевое слово **virtual** в иерархии наследования позволяет ограничить количество экземпляров базового класса **Animal** до одного

Анализ

Сравнив вывод с выводом листинга 11.7, можно сразу заметить, что количество экземпляров класса Animal уменьшилось до одного, что, наконец, отражает тот факт, чтосоздан был только один утконос. Все дело в ключевом слове v i r t u a l , использованном в отношениях между классами Mammal, B i r d и R e p t i l e , гарантирующем существование только одного экземпляра общего базового класса A n i m a l, если они будут объединены классом P l a t y p u s .

Это решает много проблем; одна из них — строка 41, которая теперь компилируется, как представлено в листинге 11.7.

Использование виртуального наследования для решения проблемы ромба

ПРИМЕЧАНИЕ

Проблема иерархии наследования, содержащей два или больше базовых класса, которые происходят от одного общего базового класса, приводит к необходимости разрешения неоднозначности при отсутствии виртуального наследования, называется *проблемой ромба* (diamond problem).

Это название возникло благодаря форме схемы классов, где прямоугольники классов и связи между ними создают ромбовидную фигуру.

Использование виртуального наследования для решения проблемы ромба

ПРИМЕЧАНИЕ
Ключевое слово **virtual** в языке C++ используется двух разных концепций:

1. Объявление функции *виртуальной* означает, что будет вызвана ее переопределенная версия, существующая в производном классе.

2. Отношения наследования, объявленные с использованием ключевого слова **virtual,** между классами **Derived1** и **Derived2,** происходящими от класса **Base,** означают, что экземпляр следующего класса, **Derived3,** происходящего от классов **Derived1** и **Derived2,** будет содержать только один экземпляр класса **Base.**

Виртуальные конструкторы копий невозможны

Технически в языке C++ невозможно получить виртуальные конструкторы копий. Но все же можно создать коллекцию (например, статический массив) типа Base* , каждый элемент которого является специализацией этого типа:

```
// Классы Tuna, Carp и Trout открыто происходят от базового класса Fish
Fish* pFishes[3];
Fishes[0] = new Tuna();
Fishes[1] = new Carp();
Fishes[2] = new Trout();
```

Виртуальные конструкторы копий невозможны

Виртуальные конструкторы копий невозможны, поскольку ключевое слово v i r t u a l в контексте методов базового класса, переопределяемых реализациями, доступным и в производном классе, свидетельствует о полиморфном поведении во время выполнения. Конструкторы, напротив, не полиморфны по своей природе, так как способны создавать экземпляр только фиксированного типа, а следовательно, язык C++ не позволяет использовать виртуальные конструкторы копий.

Виртуальные конструкторы копий невозможны

Определим собственную функцию клонирования:

```
class Fish
{
public:
    virtual Fish* Clone() const = 0; // чистая виртуальная функция
};

class Tuna:public Fish
{
// ... другие члены
public:
    Tuna * Clone() const // виртуальная функция клонирования
    {
        return new Tuna(*this); // вернуть новый объект класса Tuna,
                                // являющийся копией этого
    }
};
```

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

Таким образом, виртуальная функция C l o n e () моделирует виртуальный конструктор копий, который должен быть вызван явно, как представлено в листинге 11.9.

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:     virtual Fish* Clone() = 0;
7:     virtual void Swim() = 0;
8: };
9:
```

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

```
10: class Tuna: public Fish
11: {
12: public:
13:     Fish* Clone()
14:     {
15:         return new Tuna (*this);
16:     }
17:
18:     void Swim()
19:     {
20:         cout << "Tuna swims fast in the sea" << endl;
21:     }
22: };
23:
```

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

```
24: class Carp: public Fish
25: {
26:     Fish* Clone()
27:     {
28:         return new Carp(*this);
29:     }
30:     void Swim()
31:     {
32:         cout << "Carp swims slow in the lake" << endl;
33:     }
34: };
35:
```

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

```
int main()
{
    const int ARRAY_SIZE = 4;

    Fish* myFishes[ARRAY_SIZE] = {NULL};
    myFishes[0] = new Tuna();
    myFishes[1] = new Carp();
    myFishes[2] = new Tuna();
    myFishes[3] = new Carp();

    Fish* myNewFishes[ARRAY_SIZE];
    for (int Index = 0; Index < ARRAY_SIZE; ++Index)
        myNewFishes[Index] = myFishes[Index]->Clone();

```

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

```
50:     // вызов виртуального метода для проверки
51:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
52:         myNewFishes[Index]->Swim();
53:
54:     // очистка памяти
55:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
56:     {
57:         delete myFishes[Index];
58:         delete myNewFishes[Index];
59:     }
60:
61:     return 0;
62: }
```

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

Результат

```
Tuna swims fast in the sea
Carp swims slow in the lake
Tuna swims fast in the sea
Carp swims slow in the lake
```

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

Анализ

Строки 40-44 в функции m a i n () демонстрируют объявление статического массива указателей на базовый класс F i s h * и индивидуальное присвоение его элементам вновь созданных объектов класса Tuna, C a r p , Tuna и C a r p соответственно.

Обратите внимание на то, что этот массив m y F i s h e s способен хранить объекты, казалось бы, разных типов, которые связаны общим базовым классом F i s h .

Вы можете копировать в новый массив m y N e w F i s h e s типа F i s h * при помощи вызова в цикле виртуальной функции F i s h : : C l o n e ( ), как показано в строке 48.

ЛИСТИНГ 11.9. Классы **Tuna** и **Carp** с функцией **Clone** (), моделирующей виртуальный конструктор копий

Анализ

Обратите внимание, что массив очень мал: только четыре элемента. Он может быть много больше, хотя это и не будет иметь большого значения для логики копирования, а только потребует коррекции условия завершения цикла. Строка 52 фактически является проверкой, где вы вызываете виртуальную функцию F i s h : : Swim () для каждого хранимого в новом массиве элемента, чтобы проверить, скопировала ли функция C l o n e () объект класса Tuna как Tuna, а не только как F i s h .

Вывод демонстрирует, что все скопировано правильно.

Полиморфизм – итоги:

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Отмечайте виртуальными те функции базового класса, которые должны быть переопределены в производных классах	**Не забывайте** оснащать базовый класс виртуальным деструктором
Помните, что чистые виртуальные функции делают класс абстрактным, а сами эти функции должны быть реализованы в производном классе	**Не забывайте**, что компилятор не позволит создать экземпляр абстрактного класса
Учитывайте возможность использования виртуального наследования	**Не забывайте**, что виртуальное наследование гарантирует общий базовый класс от проблемы ромба и позволит создать только один его экземпляр
	Не путайте назначение ключевого слова `virtual` при использовании в создаваемой иерархии наследования с тем же словом в объявлении функций базового класса