Реализация наследования

Реализация наследования

Объектно-ориентированное программирование основано на четырех важных аспектах:

- *инкапсуляция* (encapsulation),
- *абстракция (*abstraction),
- *наследование* (inheritance)
- *полиморфизм* (polymorphism)

Реализация наследования

- *Наследование* — это мощнейший способ многократного использования атрибутов и краеугольный камень полиморфизма.

Реализация наследования

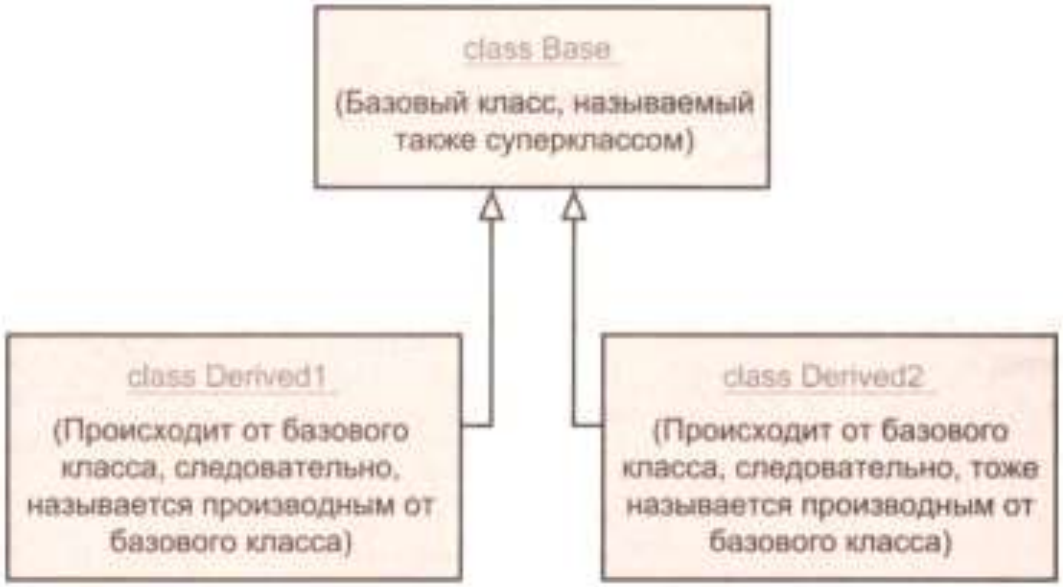

РИС. 10.1. Наследование классов

Наследование и происхождение

На рис. 10.1 приведена схема отношений между *базовым классом* (base class) и происходящими от него *производными классами* (derived class).

Наследование и происхождение

Примечание

Отношения между производным и базовым классами применимы только к *открытому наследованию* (public inheritance).

Это занятие начинается с рассмотрения открытого наследования, чтобы объяснить саму концепцию наследования на примере его наиболее распространенной формы, прежде чем переходить к закрытому и защищенному наследованию.

Наследование и происхождение

Чтобы проще объяснить эту концепцию, рассмотрим базовый класс B i r d (Птица). От класса B i r d происходят классы Crow (Ворона), P a r r o t (Попугай) и Kiwi (Киви). Класс B i r d определяет большинство основных атрибутов птицы, таких как наличие крыльев, откладывание яиц, способность лететь (у большинства). Производные классы, такие как Brow, P a r r o t и Kiwi, унаследовали бы эти атрибуты и скорректировали бы их (например, класс Kiwi не имел бы реализации метода F l y () (летать)). Еще несколько примеров наследования приведено в табл. 10.1.

Наследование и происхождение

ТАБЛИЦА 10.1. Примеры открытого наследования из повседневной жизни

Базовый класс	Примеры производных классов
`Fish` (Рыба)	`Goldfish` (Золотая рыбка), `Carp` (Карп), `Tuna` (Тунец) (Тунец *есть* рыба)
`Mammal` (Млекопитающее)	`Human` (Человек), `Elephant` (Слон), `Lion` (Лев), `Platypus` (Утконос) (Утконос *есть* млекопитающее)
`Bird` (Птица)	`Crow` (Ворона), `Parrot` (Попугай), `Ostrich` (Страус), `Kiwi` (Киви), `Platypus` (Утконос) (Утконос *есть* также и птица!)
`Shape` (Форма)	`Circle` (Круг), `Polygon` (Многоугольник) (Круг *есть* форма)
`Polygon` (Многоугольник)	`Triangle` (Треугольник), `Octagon` (Восьмиугольник) (Восьмиугольник *есть* многоугольник, который *есть* форма)

Наследование и происхождение

Эта таблица демонстрирует то, что если надеть объектно-ориентированные очки, то примеры наследования можно увидеть повсюду вокруг.

F i s h — это базовый класс для класса Tuna, поскольку Тунец, как и Карп, является рыбой и имеет все присущие рыбе характеристики, такие как хладнокровие. Однако Тунец отличается от Карпа внешним видом, скоростью плавания и тем фактом, что это морская рыба. Таким образом, классы Tuna и C a r p наследую т общие характеристики от общего базового класса F i s h , но все же специализируют атрибуты своего базового класса, чтобы отличатся друг от друга (рис. 10.2).

Наследование и происхождение

РИС. 10.2. Иерархические отношения между классами Tuna, Carp и Fish

Наследование и происхождение

Утконос может плавать, но все же это млекопитающее животное, поскольку кормит детенышей молоком, птица (и похож на птицу), поскольку кладет яйца, и рептилия, поскольку ядовит. Таким образом, класс P l a t y p u s можно представить наследником двух базовых классов, класса Mammal и класса Bird , чтобы наследовать возможности млекопитающих и птиц. Это называется *множественным наследованием* (multiple inheritance) и обсуждается далее на этом занятии.

Синтаксис наследования C++

Как унаследовать класс Carp от класса Fish и вообще унаследовать класс *Производный* от класса *Базовый*?

В языке C++ для этого используется следующий синтаксис:

Синтаксис наследования C++

```
// объявление базового класса
class Базовый
{
    // ... члены базового класса
};
// объявление производного класса
class Производный: МодификаторДоступа Базовый
{
    // ... члены производного класса
};
```

Синтаксис наследования C++

МодификаторДоступа может быть любой, чаще всего используется модификатор p u b l i c , для отношений “производный класс *есть* базовый класс” (is-a), или модификаторы p r i v a t e и p r o t e c t e d для отношений “производный класс *содержит* базовый класс” (has-a).

Синтаксис наследования C++

Иерархическое представление наследования классом C a r p класса F i s h было бы таким:

```
class Fish
{
    // ... члены класса Fish
};

class Carp:public Fish
{
    // ... члены класса Carp
};
```

Синтаксис наследования C++

Иерархическое представление наследования классом C a r p класса F i s h было бы таким:

```
class Fish
{
    // ... члены класса Fish
};

class Carp:public Fish
{
    // ... члены класса Carp
};
```

Синтаксис наследования C++

Замечание о терминологии

Читая о наследовании, вы встретите такие термины, как *наследуется от* (inherits from) и происходит от (derives from). Они имеют одинаковый смысл.

Точно так же *базовый класс* (base class) иногда называют суперклассом (super class).

Класс, происходящий от базового, называемый *производным классом* (derived class), может упоминаться как *подкласс* (subclass).

Синтаксис наследования C++

Пригодные для компиляции версии классов Carp и Tuna, производных от класса Fish , представлены в листинге 10.1.

Листинг 10.1. Пример иерархии наследования

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:     bool isFreshWaterFish;
 7:
 8:     void Swim()
 9:     {
10:         if (isFreshWaterFish)
11:             cout << "Swims in lake" << endl;
12:         else
13:             cout << "Swims in sea" << endl;
14:     }
15: };
16:
```

Листинг 10.1. Пример иерархии наследования

```
17: class Tuna: public Fish
18: {
19: public:
20:     Tuna()
21:     {
22:         isFreshWaterFish = false;
23:     }
24: };
25:
26: class Carp: public Fish
27: {
28: public:
29:     Carp()
30:     {
31:         isFreshWaterFish = true;
32:     }
33: };
34:
```

Листинг 10.1. Пример иерархии наследования

```
35: int main()
36: {
37:     Carp myLunch;

38:     Tuna myDinner;
39:
40:     cout << "About my food:" << endl;
41:
42:     cout << "Lunch: ";
43:     myLunch.Swim();
44:
45:     cout << "Dinner: ";
46:     myDinner.Swim();
47:
48:     return 0;
49: }
```

Листинг 10.1. Пример иерархии наследования

Результат

```
About my food:
Lunch: Swims in lake
Dinner: Swims in sea
```

Листинг 10.1. Пример иерархии наследования

Результат

```
About my food:
Lunch: Swims in lake
Dinner: Swims in sea
```

Листинг 10.1. Пример иерархии наследования

Анализ

Обратите внимание на строки 37 и 38 в функции main (), где создаются объекты myLunch и myDinner классов Carp и Tuna соответственно. В строках 43 и 46 я прошу свой завтрак и обед поплавать, вызвав их метод Swim (), который они должны поддерживать.

Теперь, посмотрим на определение класса Tuna в строках 17-24 и класса Carp в строках 26-33. Как можно заметить, эти классы весьма компактны, и ни один из них, кажется, не определяет метод, Swim (), который мы сумели успешно вызывать в функции main ().

Листинг 10.1. Пример иерархии наследования

Анализ

Очевидно, метод Swim () исходит от класса Fish , определенного в строках 3-15, и унаследованный ими. Поскольку класс Fish объявляет метод Swim () открытым, происходящие от него классы Tuna и Carp наследуют его (в ходе открытого наследования, осуществляемого в строках 17 и 26) и автоматически предоставляют.

Обратите внимание, как конструкторы классов Carp и Tuna инициализирует флаг базового класса FreshWaterFish, который играет роль при решении, что отображает метод Fish : : Swim ()

Модификатор доступа protected

В листинге 10.1 у класса Fish есть открытый атрибут FreshWaterFish , значение которого устанавливается производными классами Tuna и Carp , чтобы настроить (или *специализировать* (specialize)) поведение рыбы и адаптировать ее к морской и пресной воде.

Однако в коде листинга 10.1 обнаружился серьезный недостаток: если вы захотите, то даже в функции main () сможете вмешаться в значение этого флага, который был отмечен как p u b l i c , а следовательно, открыт для манипулирования извне класса Fish при помощи, например, следующего кода:

Модификатор доступа protected

```
myDinner.FreshWaterFish = true; // сделать тунца пресноводной рыбой!
```

Такого, очевидно, следует избегать. Необходимо средство, позволяющее определенным атрибутам в базовом классе быть доступными только для производного класса, но не для внешнего мира. Это означает, что логический флаг FreshWaterFish в классе Fish должен быть доступен для классов Tuna и Carp, которые происходят от него, но не для функции main (), где создаются экземпляры класса Tuna или Carp . Вот где пригодится ключевое слово p r o t e c t e d .

Модификатор доступа protected

ПРИМЕЧАНИЕ

Ключевые слова **protected** (защищенный), а также **public** (открытый) и |**private** (закрытый) являются модификаторами доступа.

Когда вы объявляете атрибут как **protected,** вы фактически делаете его доступным для производных классов и друзей, одновременно делая его недоступным для всех остальных, включая функцию **main ().**

Модификатор доступа protected

Если необходимо, чтобы определенный атрибут в базовом классе был доступен для его производных классов, следует использовать модификатор доступа p r o t e c t e d , как показано в листинге 10.2.

ЛИСТИНГ 10.2. Улучшенный класс Fish, использующий ключевое слово protected для предоставления его переменных-членов только производным классам

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

class Fish
{
protected:
    bool FreshWaterFish; // доступно только производным классам

public:
    void Swim()
    {
        if (FreshWaterFish)
            cout << "Swims in lake" << endl;
        else
            cout << "Swims in sea" << endl;
    }
};
```

ЛИСТИНГ 10.2. Улучшенный класс Fish, использующий ключевое слово protected для предоставления его переменных-членов только производным классам

```
class Tuna: public Fish
{
public:
    Tuna()
    {
        FreshWaterFish = false; // установка значения защищенного
                                // члена базового класса
    }
};

class Carp: public Fish
{
public:
    Carp()
    {
        FreshWaterFish = false;
    }
};
```

ЛИСТИНГ 10.2. Улучшенный класс Fish, использующий ключевое слово protected для предоставления его переменных-членов только производным классам

```
int main()
{
    Carp myLunch;
    Tuna myDinner;

    cout << "Getting my food to swim" << endl;

    cout << "Lunch: ";
    myLunch.Swim();

    cout << "Dinner: ";
    myDinner.Swim();

    // Снимите комментарий со строки ниже, чтобы убедиться в
    // недоступности защищенных членов извне иерархии класса
    // myLunch.FreshWaterFish = false;

    return 0;
}
```

ЛИСТИНГ 10.2. Улучшенный класс Fish, использующий ключевое слово protected для предоставления его переменных-членов только производным классам

Результат

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

ЛИСТИНГ 10.2. Улучшенный класс Fish, использующий ключевое слово protected для предоставления его переменных-членов только производным классам

- **Анализ**
  Несмотря на совпадение вывода листингов 10.1 и 10.2, здесь в класс Fish , определенный в строках 3 -16, внесены фундаментальные изменения.

  Первое и самое очевидное изменение — логическая переменная-член Fish : : FreshWaterFish стала защищенной, а следовательно, недоступной из функции main (), как свидетельствует строка 51 (снимите комментарий, чтобы увидеть ошибку компиляции).

  Тем не менее этот параметр с модификатором доступа p r o t e c t e d доступен из производных классов Tuna и Carp, как показано в строках 23 и 32 соответственно.

  Фактически эта небольшая программа демонстрирует использование ключевого слова p r o t e c t e d для обеспечения защиты атрибута базового класса, который должен быть унаследован, от обращения извне иерархии класса.

ЛИСТИНГ 10.2. Улучшенный класс Fish, использующий ключевое слово protected для предоставления его переменных-членов только производным классам

- **Анализ**
  Это очень важный аспект объектно-ориентированного программирования — комбинация абстракции данных и наследования для обеспечения безопасного наследования производными классами атрибутов базового класса, в которые не может вмешаться никто извне этой иерархической системы.

Инициализация базового класса — передача параметров для базового класса

- Что, если базовый класс содержит перегруженный конструктор, которому во время создания экземпляра требуется передать аргументы? Как будет инициализирован такой базовый класс при создании экземпляра производного класса? Фокус в использовании списков инициализации и вызове соответствующего конструктора базового класса через конструктор производного класса, как демонстрирует следующий код:

Инициализация базового класса — передача параметров для базового класса

```
class Base
{
public:
    Base(int SomeNumber) // перегруженный конструктор
    {
        // Сделать нечто с SomeNumber
    }
};
Class Derived: public Base
{
public:
    Derived(): Base(25) // создать экземпляр класса Base с аргументом 25
    {
        // код конструктора производного класса
    }
};
```

Инициализация базового класса — передача параметров для базового класса

- Этот механизм может весьма пригодиться в классе Fish при предоставлении логического входного параметра для его конструктора, инициализирующего переменную-член F i s h : : F r e s h W a t e r F i s h .

  Так, базовый класс F i s h может гарантировать, что каждый производный класс вынужден будет указать, является ли рыба пресноводной или морской, как представлено в листинге 10.3.

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

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

class Fish
{
protected:
    bool FreshWaterFish; // доступно только производным классам

public:
    // конструктор класса Fish
    Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}

    void Swim()
    {
        if (FreshWaterFish)
            cout << "Swims in lake" << endl;
        else
            cout << "Swims in sea" << endl;
    }
};
```

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

```
class Tuna: public Fish
{
public:
    Tuna(): Fish(false) {}
};

class Carp: public Fish
{
public:
    Carp(): Fish(true) {}
};
```

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

```
int main()
{
    Carp myLunch;
    Tuna myDinner;

    cout << "Getting my food to swim" << endl;

    cout << "Lunch: ";
    myLunch.Swim();

    cout << "Dinner: ";
    myDinner.Swim();

    // Снимите комментарий со строки 48, чтобы убедиться в
    // недоступности защищенных членов извне иерархии класса
    // myLunch.FreshWaterFish = false;

    return 0;
}
```

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

Результат

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

- **Анализ**

  Теперь у класса Fish есть конструктор, который получает заданный по умолчанию параметр, инициализирующий переменную F i s h : : F r e s h W a t e r F i s h .

  Таким образом, единственная возможность создать объект класса Fish — это предоставить параметр, который инициализирует защищенный член.

  Так, класс Fish гарантирует, что защищенный член класса не будет содержать случайного значения, если пользователь производного класса забудет его установить.

  Теперь производные классы Tuna и Carp вынуждены определить конструктор, создающий экземпляр базового класса Fish с правильным параметром (t r u e или f a l s e , указывающим, пресноводная ли это рыба), как показано в строках 24 и 30 соответственно.

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

- **ПРИМЕЧАНИЕ**

  Как можно заметить в листинге 10.3, производный класс никогда не обращался непосредственно к логической переменной-члену **Fish::FreshWaterFish,** несмотря на то, что она является защищенной, поскольку ее значение было установлено конструктором класса **Fish.**

  Чтобы гарантировать максимальную защиту, если производные классы не нуждаются в доступе к атрибуту базового класса, отметьте его как **private.**

Производный класс, переопределяющий методы базового класса

- Если производный класс реализует те же функции с теми же возвращаемыми значениями и сигнатурами, что и базовый класс, от которого он происходит, то он фактически переопределяет этот метод базового класса, как показано в следующем коде:

Производный класс, переопределяющий методы базового класса

```
class Base
{
public:
    void DoSomething()
    {
        // код реализации... Делает нечто
    }
};

class Derived:public Base
{
public:
    void DoSomething()
    {
        // код реализации... Делает нечто другое
    }
};
```

Производный класс, переопределяющий методы базового класса

Таким образом, если бы метод

DoSomething () должен быть вызван с использованием экземпляра

класса Derived, то это не задействовало бы функциональные возможности в классе Base.

Производный класс, переопределяющий методы базового класса

Если классы Tuna и Carp должны реализовать собственный метод Swim (), который существует также и в базовом классе как Fish : : Swim (), то его вызов в методе main () так, как показано в следующем отрывке листинга 10.3,

```
36:     Tuna myDinner;
// ... другие строки
44:     myDinner.Swim();
```

Производный класс, переопределяющий методы базового класса

привел бы к выполнению локальной реализации метода Tuna : : Swim ( ), которая, по существу, переопределяет метод Fish : : Swim () базового класса.

Это демонстрирует листинг 10.4.

ЛИСТИНГ 10.4. Производные классы **Tuna** и **Carp**, переопределяющие метод **Swim()** базового класса **Fish**

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: private:
 6:    bool isFreshWaterFish;
 7:
 8: public:
 9:    // Fish constructor
10:    Fish(bool isFreshWater) : isFreshWaterFish(isFreshWater){}
11:
12:    void Swim()
13:    {
14:       if (isFreshWaterFish)
15:          cout << "Swims in lake" << endl;
16:       else
17:          cout << "Swims in sea" << endl;
18:    }
19: };
20:
```

ЛИСТИНГ 10.4. Производные классы **Tuna** и **Carp**, переопределяющие метод **Swim()** базового класса **Fish**

```
21: class Tuna: public Fish
22: {
23: public:
24:     Tuna(): Fish(false) {}
25:
26:     void Swim()
27:     {
28:         cout << "Tuna swims real fast" << endl;
29:     }
30: };
31:
```

ЛИСТИНГ 10.4. Производные классы **Tuna** и **Carp**, переопределяющие метод **Swim()** базового класса **Fish**

```
32: class Carp: public Fish
33: {
34: public:
35:     Carp(): Fish(true) {}
36:
37:     void Swim()
38:     {
39:         cout << "Carp swims real slow" << endl;
40:     }
41: };
42:
```

ЛИСТИНГ 10.4. Производные классы **Tuna** и **Carp**, переопределяющие метод **Swim()** базового класса **Fish**

```
43: int main()
44: {
45:     Carp myLunch;
46:     Tuna myDinner;
47:
48:     cout << "About my food" << endl;
49:
50:     cout << "Lunch: ";
51:     myLunch.Swim();
52:
53:     cout << "Dinner: ";
54:     myDinner.Swim();
55:
56:     return 0;
57: }
```

ЛИСТИНГ 10.4. Производные классы **Tuna** и **Carp**, переопределяющие метод **Swim()** базового класса **Fish**

Результат

```
About my food
Lunch: Carp swims real slow
Dinner: Tuna swims real fast
```

Вызов методов базового класса в производном классе

Обычно метод Fish : : Swim () содержал бы обобщенную реализацию плавания, применимого ко всем рыбам, включая тунцов и карпов.

Если специализированные реализации методов Tuna : : Swim () и Carp : : Swim () должны использовать обобщенную реализацию метода базового класса Fish : : Swim (), используйте оператор области видимости (: :) , как показано в следующем коде:

ЛИСТИНГ 10.5. Использование оператора области видимости (::) для вызова методов базового класса из методов производных классов и функции **main ()**

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

class Fish
{
private:
    bool FreshWaterFish;

public:
    // конструктор класса Fish
    Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}

    void Swim()
    {
        if (FreshWaterFish)
            cout << "Swims in lake" << endl;
        else
            cout << "Swims in sea" << endl;
    }
};
```

ЛИСТИНГ 10.5. Использование оператора области видимости (::) для вызова методов базового класса из методов производных классов и функции **main ()**

```
11: class Tuna: public Fish
12: {
13: public:
14:     Tuna(): Fish(false) {}
15:

26:     void Swim()
27:     {
28:         cout << "Tuna swims real fast" << endl;
29:     }
30: };
31:
32: class Carp: public Fish
33: {
34: public:
35:     Carp(): Fish(true) {}
36:
37:     void Swim()
38:     {
39:         cout << "Carp swims real slow" << endl;
40:         Fish::Swim();
41:     }
42: };
43:
```

ЛИСТИНГ 10.5. Использование оператора области видимости (::) для вызова методов базового класса из методов производных классов и функции **main ()**

```
44: int main()
45: {
46:     Carp myLunch;
47:     Tuna myDinner;
48:
49:     cout << "Getting my food to swim" << endl;
50:
51:     cout << "Lunch: ";
52:     myLunch.Swim();
53:
54:     cout << "Dinner: ";
55:     myDinner.Fish::Swim();
56:
57:     return 0;
58: }
```

ЛИСТИНГ 10.5. Использование оператора области видимости (::) для вызова методов базового класса из методов производных классов и функции **main ()**

Результат

```
Getting my food to swim
Lunch: Carp swims real slow
Swims in lake
Dinner: Swims in sea
```

ЛИСТИНГ 10.5. Использование оператора области видимости (::) для вызова методов базового класса из методов производных классов и функции **main ()**

- **Анализ**
  Метод Carp : : Swim () в строках 37-41 демонстрирует вызов функции Fish : : Swim () базового класса с использованием оператора области видимости (: :).

  Строка 55, с другой стороны, демонстрирует возможность использования оператора области видимости (: :) для вызова метода базового класса Fish : : Swim () из функции main () с использованием объекта производного класса, в данном случае Tuna.

Производный класс, скрывающий методы базового класса

- Переопределение может принять критическую форму, и тогда метод Tuna : : Swim () потенциально способен скрыть все доступные перегруженные версии функции Fish : : Swim (), даже приведя к неудаче компиляции, когда перегружаются используемые версии (поэтому они и называется *скрытыми* (hidden)), как показано в листинге 10.6.

ЛИСТИНГ 10.6. Сокрытие методом **Tuna: :Swim()** перегруженного метода **Tuna: :Swim(bool)**

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

class Fish
{
public:
    void Swim()
    {
        cout << "Fish swims... !" << endl;
    }

    void Swim(bool FreshWaterFish)
    {
        if (FreshWaterFish)
            cout << "Swims in lake" << endl;
        else
            cout << "Swims in sea" << endl;
    }
};
```

ЛИСТИНГ 10.6. Сокрытие методом **Tuna: :Swim()** перегруженного метода **Tuna: :Swim(bool)**

```
class Tuna: public Fish
{
public:
    void Swim()
    {
        cout << "Tuna swims real fast" << endl;
    }
};

int main()
{
    Tuna myDinner;

    cout << "Getting my food to swim" << endl;

    // myDinner.Swim(false); // отказ компиляции: Fish::Swim(bool)
                             // скрыт методом Tuna::Swim()
    myDinner.Swim();

    return 0;
}
```

ЛИСТИНГ 10.6. Сокрытие методом **Tuna: :Swim()** перегруженного метода **Tuna: :Swim(bool)**

Результат

```
Getting my food to swim
Tuna swims real fast
```

ЛИСТИНГ 10.6. Сокрытие методом **Tuna: :Swim()** перегруженного метода **Tuna: :Swim(bool)**

- **Анализ**
  Эта версия класса F i s h немного отличается от тех, которые вы видели до сих пор.
  Кроме минимизации версией, для объяснения текущей проблемы данная версия класса F i s h содержит два перегруженных метода Swim (): один не получает никаких параметров (строки 6-9), а другой получает параметр типа b o o l (строки 11-17).

  Поскольку класс Tuna наследуется от класса F i s h открыто (строка 20), не будет ошибкой ожидать, что обе версии метода

  F i s h : : Swim () будут доступны через экземпляр класса Tuna. Однако в результате того факта, что класс Tuna реализует собственную версию метода T u n a : : Swim () (строки 23-26), функция F i s h : : Swim (b o o l) скрывается от компилятора. Если снять комментарий со строки 35, произойдет отказ компиляции.

ЛИСТИНГ 10.6. Сокрытие методом **Tuna: :Swim()** перегруженного метода **Tuna: :Swim(bool)**

Так, если необходимо вызвать функцию Fish : : Swim (b o o l) через экземпляр класса Tuna, возможны следующие решения.

- Решение 1. Используйте оператор области видимости в функции main ():

myDinner.Fish::Swim();

ЛИСТИНГ 10.6. Сокрытие методом **Tuna: :Swim()** перегруженного метода **Tuna: :Swim(bool)**

- Решение 2. Используйте в классе Tuna ключевое слово using , чтобы показать скрытые методы Swim () в классе Fish :

```
class Tuna: public Fish
{
public:
    using Fish::Swim; // показать скрытые методы Swim()
                      // в базовом классе Fish

    void Swim()
    {
        cout << «Tuna swims real fast» << endl;
    }
};
```

ЛИСТИНГ 10.6. Сокрытие методом **Tuna: :Swim()** перегруженного метода **Tuna: :Swim(bool)**

- Решение 3. Переопределите все перегруженные варианты метода Swim () в классе Tuna (если хотите, вызовите метод Fish : : Swim (. . .) через

Tuna : : Fish (. . .)) :

```
class Tuna: public Fish
{
public:
    void Swim(bool FreshWaterFish)
    {
        Fish::Swim(FreshWaterFish);
    }

    void Swim()
    {
        cout << «Tuna swims real fast» << endl;
    }
};
```

Порядок создания

- При создании объекта класса Tuna, производного от класса Fish, конструктор класса Tuna будет вызван до или после конструктора класса Fish?

  Кроме того, каков порядок издания таких атрибутов класса, как

  Fish : : FreshWaterFish, при создании экземпляра объектов в иерархии класса?

Порядок создания

- Дело в том, что объекты базового класса создаются перед объектами производного.

  Таким образом, часть Fish объекта класса Tuna создается сначала, чтобы его члены, в частности открытые и защищенные, были готовы для использования, когда будет создаваться часть Tuna.

  В ходе создания экземпляра класса Fish и Tuna такие атрибуты, как Fish : : FreshWaterFish, создаются до вызова конструктора Fish : : Fish (), гарантируя существование атрибутов на момент работы конструктора с ними.

  То же самое относится к конструктору Tuna : : Tuna ().

Порядок удаления

- Когда экземпляр класса Tuna выходит из области видимости, последовательность удаления противоположна последовательности создания.

  В листинге 10.7 приведен простой пример, демонстрирующий последовательность создания и удаления.

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class FishDummyMember
 4: {
 5: public:
 6:     FishDummyMember()
 7:     {
 8:         cout << "FishDummyMember constructor" << endl;
 9:     }
10:
11:     ~FishDummyMember()
12:     {
13:         cout << "FishDummyMember destructor" << endl;
14:     }
15: };
16:
```

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

```
17: class Fish
18: {
19: protected:
20:     FishDummyMember dummy;
21:
22: public:
23:     // Fish constructor
24:     Fish()
25:     {
26:         cout << "Fish constructor" << endl;
27:     }
28:
29:     ~Fish()
30:     {
31:         cout << "Fish destructor" << endl;
32:     }
33: };
34:
```

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

```
35: class TunaDummyMember
36: {
37: public:
38:     TunaDummyMember()
39:     {
40:         cout << "TunaDummyMember constructor" << endl;
41:     }
42:
43:     ~TunaDummyMember()
44:     {
45:         cout << "TunaDummyMember destructor" << endl;
46:     }
47: };
48:
49:
```

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

```
50: class Tuna: public Fish
51: {
52: private:
53:     TunaDummyMember dummy;
54:
55: public:
56:     Tuna()
57:     {
58:         cout << "Tuna constructor" << endl;
59:     }
60:     ~Tuna()
61:     {
62:         cout << "Tuna destructor" << endl;
63:     }
64:
65: };
66:
67: int main()
```

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

Результат

```
FishDummyMember constructor
Fish constructor
TunaDummyMember constructor
Tuna constructor
Tuna destructor
TunaDummyMember destructor
Fish destructor
FishDummyMember destructor
```

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

Анализ

Функция main ( ), представленная в строках 67-70, поразительно мала для объема создаваемого ею вывода. Создания экземпляра класса Tuna достаточно для этих строк вывода, поскольку операторы cout вставлены в конструкторы и деструкторы всех задействованных объектов.

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

Анализ

Для демонстрации создания и удаления переменных определены два вымышленных класса, FishDummyMember и TunaDummyMember, с операторами cout в конструкторах и деструкторах. Классы Fish и Tuna содержат члены для каждого из этих вымышленных классов (строки 20 и 53).

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

Анализ

Вывод указывает, что создание объекта класса Tuna фактически начинается сверху иерархии. Так, часть базового класса Fish в составе класса Tuna создается первой, при этом такие его члены, как F i s h : : dummy, создаются сначала.
Далее следует конструктор класса Fish , который естественно выполняется после создания таких атрибутов, как dummy.

ЛИСТИНГ 10.7. Порядок создания и удаления базового класса, производного класса и его членов

Анализ

После создания экземпляра базового класса создание экземпляра Tuna продолжается созданием экземпляра T u n a : : dummy и Завершается выполнением кода конструктора T u n a : : T u n a ().
Вывод демонстрирует, что последовательность удаления прямо противоположна.