Полиморфизм

Полиморфизм

“Поли” в переводе с греческого языка означает *много*, а “морф” — *форма.*

Полиморфизм (polymorphism) — это средство объектно-ориентированных языков, позволяющее обрабатывать подобным образом объекты разных типов.

Полиморфизм

Это занятие посвящено полиморфному поведению , которое может быть реализовано на языке C++ через иерархию наследования, известную также как *полиморфизм подтипа* (subtype polymorphism).

Полиморфизм

На занятии “Реализация наследования”, вы видели, как классы Tuna и Carp наследовали открытый метод Swim () от класса Fish (см. листинг 10.1).

Классы Tuna и Carp способны предоставить собственные методы T u n a : : Swim () и C a r p : : Swim (), чтобы тунец и карп плавали по-разному. Но поскольку каждый из них является также рыбой, пользователь экземпляра класса Tuna вполне может использовать тип базового класса для вызова метода F i s h : : Swim (), который выполнит только общую часть F i s h : : Swim (), а не полную T u n a : : Swim (), даже при том, что этот экземпляр базового класса Fish является частью класса Tuna.

Эта проблема представлена в листинге 11.1.

Полиморфизм

ПРИМЕЧАНИЕ

Во всех примерах кода на этом занятии было удалено все, что не является абсолютно необходимым для объяснения рассматриваемой темы, а количество строк кода сведено к минимуму, чтобы улучшить удобочитаемость.

При реальном программировании классы необходимо создавать правильно, а также разрабатывать осмысленные иерархии наследования, учитывающие цели проекта и приложения в перспективе.

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса **Fish,** который принадлежит классу **Tuna**

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:     void Swim()
7:     {
8:         cout << "Fish swims! " << endl;
9:     }
10: };
11:
```

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса **Fish,** который принадлежит классу **Tuna**

```
12: class Tuna:public Fish
13: {
14: public:
15:     // override Fish::Swim
16:     void Swim()
17:     {
18:         cout << "Tuna swims!" << endl;
19:     }
20: };
21:
22: void MakeFishSwim(Fish& inputFish)
23: {
24:     // calling Fish::Swim
25:     inputFish.Swim();
26: }
27:
```

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса **Fish,** который принадлежит классу **Tuna**

```
28: int main()
29: {
30:     Tuna myDinner;
31:
32:     // calling Tuna::Swim
33:     myDinner.Swim();
34:
35:     // sending Tuna as Fish
36:     MakeFishSwim(myDinner);
37:
38:     return 0;
39: }
```

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса **Fish,** который принадлежит классу **Tuna**

Результат

```
Tuna swims!
Fish swims!
```

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса **Fish,** который принадлежит классу **Tuna**

Анализ

Класс Tuna специализирует класс Fish через открытое наследование, как показано в строке 12. Он также переопределяет метод F i s h : :S w i m ( ) . Функция main () напрямую вызывает метод T u n a : :S w i m () в строке 33 и передает объект m y D i n n e r (класса Tuna) как параметр для функции MakeFishSwim (), которая интерпретирует это как ссылку Fish& , как видно в ее объявлении (строка 22). Другими словами, вызов функции MakeFishSwim (Fish&) не заботит, что был передан объект класса Tuna, он обрабатывает его как объект класса Fish и вызывает метод F i s h : : Swim ().

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса **Fish,** который принадлежит классу **Tuna**

Анализ

Так, вторая строка вывода
означает, что тот же объект класса Tuna создал вывод, как у класса F i s h , без всякой специализации (с таким же успехом это мог быть класс Carp).
Однако пользователь, в идеале, ожидал бы, что объект класса Tuna поведет себя как тунец, даже если вызван метод F i s h : : Swim ().

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса **Fish,** который принадлежит классу **Tuna**

Анализ

Другими словами, когда метод

I n p u t F i s h .Swim () вызывается в строке 25, он ожидает, что будет выполнен метод T u n a : : Swim ().
Такое полиморфное поведение, когда объект известного класса типа F i s h может вести себя как объект фактического типа, а именно производный класс Tuna, может быть реализован, если сделать функцию

F i s h : : Swim () виртуальной.

Полиморфное поведение, реализованное при помощи виртуальных функций

Доступ к объекту класса F i s h возможен через указатель F i s h * или по ссылке Fish& .
Объект класса F i s h может быть создан индивидуально или как часть объекта класса Tuna или C a r p , производного от класса F i s h .
Неважно, как именно, но вы вызываете метод Swim (), используя этот указатель или ссылку:

pFish->Swim();
myFish.Swim();

Полиморфное поведение, реализованное при помощи виртуальных функций

Вы ожидаете, что объект класса F i s h будет плавать, как тунец, если это часть объекта класса Tuna, или как карп, если это часть объекта класса Carp, или как безымянная рыба, если объект класса F i s h был создан не как часть такого специализированного класса, как Tuna или Carp. Вы можете гарантировать это, объявив функцию Swim () в базовом классе F i s h как *виртуальную функцию* (virtual function):

Полиморфное поведение, реализованное при помощи виртуальных функций

```
class Base
{
   virtual ReturnType FunctionName (Parameter List);
};
class Derived
{
   ReturnType FunctionName (Parameter List);
};
```

Полиморфное поведение, реализованное при помощи виртуальных функций

Использование ключевого слова v i r t u a l означает, что компилятор гарантирует вызов любого переопределенного варианта затребованного метода базового класса. Таким образом, если метод Swim () объявлен как v i r t u a l , вызов m y F i s h . Swim () (m y F i s h имеет тип Fish&) приводит к вызову метода T u n a : : Swim (), как показано в листинге 11.2.

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:     virtual void Swim()
 7:     {
 8:         cout << "Fish swims!" << endl;
 9:     }
10: };
11:
```

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

```
12: class Tuna:public Fish
13: {
14: public:
15:     // переопределение Fish::Swim
16:     void Swim()
17:     {
18:         cout << "Tuna swims!" << endl;
19:     }
20: };
21:
```

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

```
22: class Carp:public Fish
23: {
24: public:
25:     // переопределение Fish::Swim
26:     void Swim()
27:     {
28:         cout << "Carp swims!" << endl;
29:     }
30: };
31:
```

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

```
32: void MakeFishSwim(Fish& InputFish)
33: {
34:     // вызов виртуального метода Swim()
35:     InputFish.Swim();
36: }
37:
```

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

```
38: int main()
39: {
40:     Tuna myDinner;
41:     Carp myLunch;
42:
43:     // передача Tuna как Fish
44:     MakeFishSwim(myDinner);
45:
46:     // передача Carp как Fish
47:     MakeFishSwim(myLunch);
48:
49:     return 0;
50: }
```

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

Результат

```
Tuna swims!
Carp swims!
```

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

Анализ

Реализация функции MakeFishSwim (Fish&) никак не изменилась с листинга 11.1, а вывод получился совсем иной. Метод F i s h : : Swim () не был вызван вообще из-за присутствия переопределенных версий T u n a : : Swim () и C a r p : : Swim (), которые получили преимущество над методом F i s h : : Swim (), поскольку последний был объявлен как виртуальная функция.

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

Анализ

Это очень важный момент. Он свидетельствует, что, даже не зная точный тип обрабатываемого объекта, класс которого происходит от класса

F i s h , реализация метода MakeFishSwim () способна привести к вызову разных реализаций метода Swim (), определенного в различных производных классах.

ЛИСТИНГ 11.2. Результат объявления метода **Fish: :Swim()** виртуальным

Анализ

Это полиморфизм: обработка различных рыб как общего типа F i s h , при гарантии выполнения правильной реализации метода Swim (), предоставляемого производными типами.

Потребность в виртуальных деструкторах

У средств, представленных в листинге 11.1, есть и оборотная сторона: неумышленный вызов функций базового класса из экземпляра производного, когда доступна специализация. Что будет, если функция применит оператор d e l e t e , используя указатель типа Base*, который фактически указывает на экземпляр производного класса?
Какой деструктор будет вызван? Рассмотрим листинг 11.3.

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

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

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

```
16: class Tuna:public Fish
17: {
18: public:
19:     Tuna()
20:     {
21:         cout << "Constructed Tuna" << endl;
22:     }
23:     ~Tuna()
24:     {
25:         cout << "Destroyed Tuna" << endl;
26:     }
27: };
28:
```

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

```
29: void DeleteFishMemory(Fish* pFish)
30: {
31:     delete pFish;
32: }
33:
```

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

```
34: int main()
35: {
36:     cout << "Allocating a Tuna on the free store:" << endl;
37:     Tuna* pTuna = new Tuna;
38:     cout << "Deleting the Tuna: " << endl;
39:     DeleteFishMemory(pTuna);
40:
41:     cout << "Instantiating a Tuna on the stack:" << endl;
42:     Tuna myDinner;
43:     cout << "Automatic destruction as it goes out of scope: " << endl;
44:
45:     return 0;
46: }
```

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

Результат

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Fish
Instantiating a Tuna on the stack:
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed Fish
```

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

Анализ

Функция main () создает экземпляр класса Tuna в динамической памяти, используя оператор new в строке 37, а затем сразу освобождает зарезервированную память, используя вспомогательную функцию DeleteFishMemory ()

в строке 39. Для сравнения: другой экземпляр класса Tuna создается в стеке как локальная переменная myDinner (строка 42) и выходит из области видимости по завершении функции main().

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

Анализ

Вывод создают операторы c o u t в конструкторах и деструкторах классов Fish и Tuna. Обратите внимание: несмотря на то, что обе части, Tuna и Fish , объекта были созданы в динамической памяти, поскольку использовался оператор new, при удалении был вызван только деструктор части Fish , а не класса Tuna. Это находится в абсолютном контрасте с созданием и удалением локального объекта myDinner, где вызываются все конструкторы и деструкторы.

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

Анализ

В листинге 10.7 был представлен правильный порядок создания и удаления классов в иерархии наследования, демонстрирующий, что должны быть вызваны все деструкторы, включая деструктор ~T u n a (). Здесь явно что-то неправильно.

ЛИСТИНГ 11.3. Функция, вызывающая оператор **delete** для типа **Base***

Анализ

Проблема в том, что код в деструкторе производного класса, объект которого был создан в динамической памяти при помощи оператора new, не будет вызван, если будет применен оператор d e l e t e для указателя типа B a se* .

В результате ресурсы не будут освобождены, произойдет утечка памяти и другие ненужные неприятности.
Чтобы избежать этой проблемы, используйте виртуальные деструкторы, как показано в листинге 11.4.

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа **Base***

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:     Fish()
7:     {
8:         cout << "Constructed Fish" << endl;
9:     }
10:    virtual ~Fish() // виртуальный деструктор!
11:    {
12:        cout << "Destroyed Fish" << endl;
13:    }
14: };
15:
```

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа **Base***

```
16: class Tuna:public Fish
17: {
18: public:
19:     Tuna()
20:     {
21:         cout << "Constructed Tuna" << endl;
22:     }
23:     ~Tuna()
24:     {
25:         cout << "Destroyed Tuna" << endl;
26:     }
27: };
28:
```

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа **Base***

```
29: void DeleteFishMemory(Fish* pFish)
30: {
31:     delete pFish;
32: }
33:
```

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа **Base***

```
34: int main()
35: {
36:     cout << "Allocating a Tuna on the free store:" << endl;
37:     Tuna* pTuna = new Tuna;
38:     cout << "Deleting the Tuna: " << endl;
39:     DeleteFishMemory(pTuna);
40:
41:     cout << "Instantiating a Tuna on the stack:" << endl;
42:     Tuna myDinner;
43:     cout << "Automatic destruction as it goes out of scope: "
            << endl;
44:
45:     return 0;
46: }
```

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа **Base***

Результат

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Tuna
Destroyed Fish
Instantiating a Tuna on the stack:
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed Fish
```

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа **Base***

Анализ

Единственное различие между листингами 11.4 и 11.3 — добавление ключевого слова v i r t u a l в строке 10, где был объявлен деструктор базового класса F i s h . Обратите внимание, что это изменение, по существу, заставило компилятор выполнить деструктор T u n a : : ~T u n a () в дополнение к деструктору F i s h : : ~F i s h (), когда был вызван оператор d e l e t e для указателя F i s h * , который фактически указывает на объект класса Tuna, как показано в строке 31.

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа **Base***

Анализ

Теперь вывод демонстрирует, что последовательность вызовов конструкторов и деструкторов одинакова, независимо от того, создан ли объект класса Tuna в динамической памяти с использованием оператора new, как показано в строке 37, или в стеке, как локальная переменная (строка 42).

Потребность в виртуальных деструкторах

ПРИМЕЧАНИЕ

Всегда объявляйте деструктор базового класса как **virtual:**

```
class Base
{
public:
        virtual  ~Base()  {}; // виртуальный деструктор
};
```

Это гарантирует, что никто с указателем **Base*** не сможет вызвать оператор **delete** способом, который не подразумевает вызова деструкторов производных классов.

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Функция M a k F i s h S w m (Fish&) в листинге 11.2 заканчивается вызовом метода Carp : : Swim () или T u n a : : Swim (), несмотря на то, что программист вызвал в ней метод F i s h : : S w i m ().

Безусловно, на момент компиляции компилятору ничего не известно о характере объектов, с которыми встретится такая функция, он не в состоянии гарантировать выполнение различных методов Swim () в различные моменты времени.

Очевидно, решение о том, какой метод Swim () должен быть вызван, принимается во время выполнения с использованием скрытой логики, реализую щ ей полиморфизм, предоставляемой компилятором во время компиляции.

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Рассмотрим класс Base, в котором объявлено *N* виртуальных функций:

```
class Base
{
public:
    virtual void Func1()
    {
        // реализация Func1
    }
    virtual void Func2()
    {
        // реализация Func2
    }
    // ... и так далее
    virtual void FuncN()
    {
        // реализация FuncN
    }
};
```

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Класс D e r i v e d , производный от класса Base, переопределяет метод B a s e : : Func2 (), предоставляя другие виртуальные функции непосредственно из класса Base:

```
class Derived: public Base
{
public:
    virtual void Func1()
    {
        // Func2 переопределяет Base::Func2
    }
    // нет реализации для Func2
    virtual void FuncN()
    {
        // реализация FuncN
    }
};
```

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Компилятор видит иерархию наследования и понимает, что класс Base определяет некоторые виртуальные функции, которые были переопределены в классе Derived .

Теперь компилятор должен составить таблицу, называемую *таблицей виртуальной функции* (Virtual Function Table — VFT), для каждого класса, который реализует виртуальную функцию, и производного класса, который переопределяет ее.

Другими словами, классы Base и Derived получают собственной экземпляр своей таблицы виртуальной функции.

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Таблицу виртуальной функции можно представить как статический массив, содержащий указатели на функцию, каждый из которых указывает на виртуальную функцию (или ее переопределенную версию), представляющую интерес (рис. 11.1).

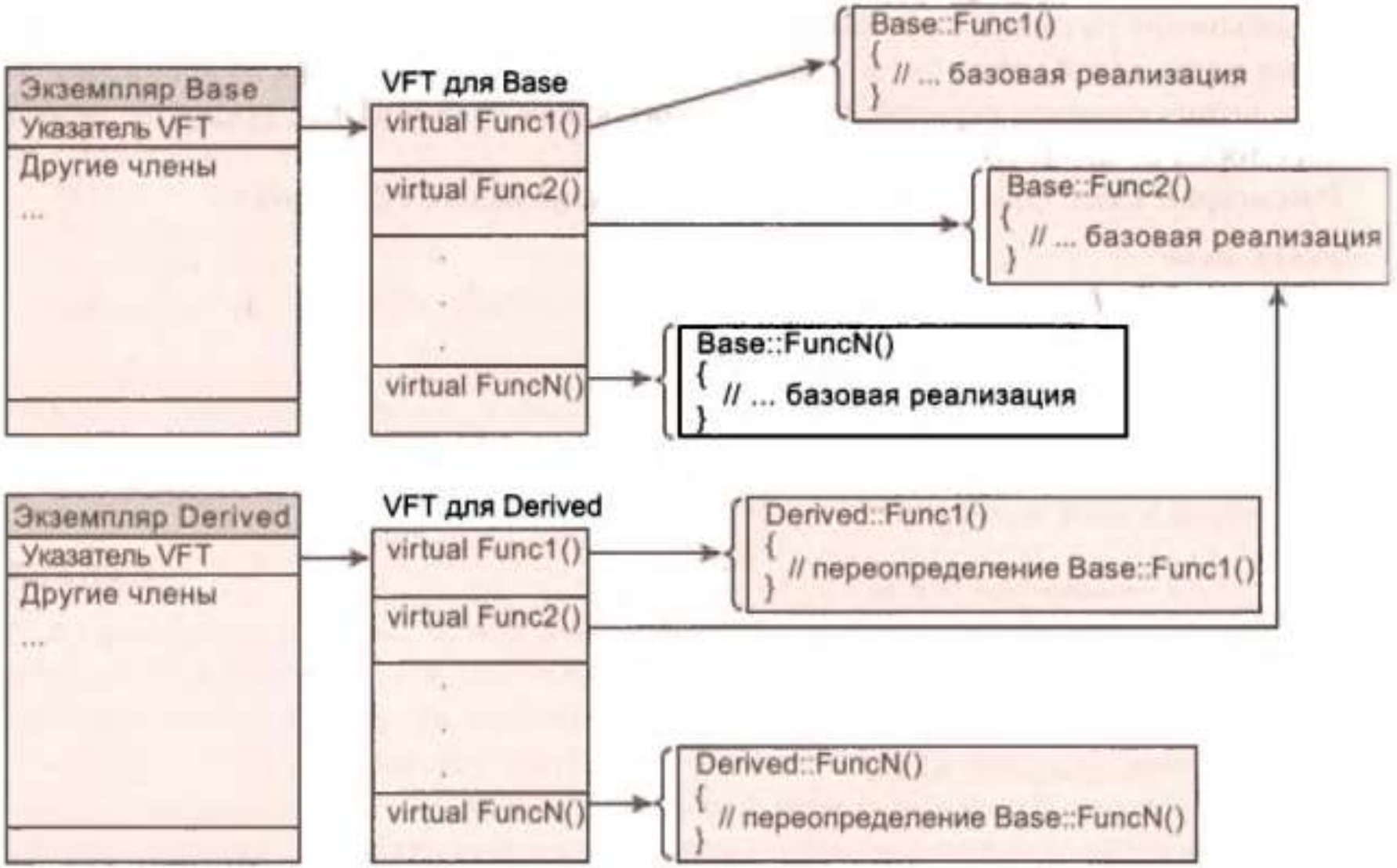

РИС. 11.1. Представление таблицы виртуальной функции для классов `Derived` и `Base`

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Таким образом, каждая таблица состоит из указателей на функции, каждый из которых указывает на доступную реализацию виртуальной функции.

В случае класса Derived все, кроме одного указателя на функцию в его таблице VFT, указывают на локальные реализации виртуального метода в классе Derived.

Класс Derived не переопределяет метод
Base : : Func2 (), а следовательно, указатель на функцию указывает на реализацию в классе Base.

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Это означает, что при следующем вызове пользователя класса Derived

CDerived objDerived;
objDerived.Func2();

компилятор осуществляет поиск в таблице VFT класса Derived и обеспечивает вызов реализации Base : : Func2 ().

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Это относится также к вызовам методов, которые были виртуально переопределены:

```
void DoSomething(Base& objBase)
{
    objBase.Func1(); // вызов Derived::Func1
}
int main()
{
    Derived objDerived;
    DoSomething(objDerived);
};
```

Как работают виртуальные функции. Понятие таблицы виртуальной функции

В данном случае, несмотря на то, что объект objDerived интерпретируется параметром objBase как экземпляр класса Base, указатель VFT в этом экземпляре все еще указывает на ту же таблицу, составленную для класса Derived.

Таким образом, функцией Func1(), выполняемой через этот указатель VFT, является, конечно, Derived : : Func1().

Вот как таблицы виртуальной функции помогают в реализации полиморфизма в C++.

Как работают виртуальные функции. Понятие таблицы виртуальной функции

Листинг 11.5 доказывает существование скрытого указателя VFT на примере сравнения размера двух идентичных классов, но у одного из них есть виртуальная функция, а у другого нет.

ЛИСТИНГ 11.5. Демонстрация наличия скрытого указателя VFT при сравнении двух одинаковых классов, функция одного из которых объявлена виртуальный

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class SimpleClass
 4: {
 5:     int a, b;
 6:
 7: public:
 8:     void DoSomething() {}
 9: };
10:
11: class Base
12: {
13:     int a, b;
14:
15: public:
16:     virtual void DoSomething() {}
17: };
18:
```

ЛИСТИНГ 11.5. Демонстрация наличия скрытого указателя VFT при сравнении двух одинаковых классов, функция одного из которых объявлена виртуальный

```
19: int main()
20: {
21:     cout << "sizeof(SimpleClass) = " << sizeof(SimpleClass) << endl;
22:     cout << "sizeof(Base) = " << sizeof(Base) << endl;
23:
24:     return 0;
25: }
```

ЛИСТИНГ 11.5. Демонстрация наличия скрытого указателя VFT при сравнении двух одинаковых классов, функция одного из которых объявлена виртуальный

Результат (Использование 64-Разрядного Компилятора)

sizeof(SimpleClass) = 8

sizeof(Base) = 16

ЛИСТИНГ 11.5. Демонстрация наличия скрытого указателя VFT при сравнении двух одинаковых классов, функция одного из которых объявлена виртуальный

- **Анализ**
  Этот пример ограничен до минимума. Вы видите два класса, SimpleClass и Base, которые идентичны по типам и количеству членов, но функция FuncDoSomething () в классе Base объявлена как виртуальная, а в классе SimpleClass как не виртуальная.
  Различие лишь в добавлении ключевого слова v i r t u a l , но компилятор создает таблицу виртуальной функции для класса Base и резервирует место для указателя на нее в том же классе Base , как его скрытый член. Этот указатель использует 4 дополнительных байта на 32-разрядной системе, что и является доказательством его существования.

Идентификация типа времени выполнения (Run Time Type Identification - RTTI)

- **ПРИМЕЧАНИЕ**

  Язык C++ позволяет запросить указатель Base*, если он имеет тип Derived*, при помощи оператора приведения типов dynamic_cast и последующего условного выполнения на основе результата запроса.

Идентификация типа времени выполнения (Run Time Type Identification - RTTI)

- **ПРИМЕЧАНИЕ**

  Это называется *идентификацией типа времени выполнения* (Run Time Type Identification - RTTI), и в идеале этого следует избегать, несмотря на поддержку большинством компиляторов C++.

  Дело в том, что необходимость узнать тип объекта производного класса по указателю базового класса обычно считается плохой практикой программирования.

Абстрактные классы и чистые виртуальные функции

Базовый класс, экземпляр которого не может быть создан, называется *абстрактным классом* (abstract base class).

Абстрактные классы и чистые виртуальные функции

Цель у такого базового класса только одна — от него получают производные классы.

Абстрактные классы и чистые виртуальные функции

Язык C++ позволяет создать абстрактный класс, используя чистые виртуальные функции.

Абстрактные классы и чистые виртуальные функции

Метод называют *чистым виртуальным* (pure virtual), когда его объявление выглядит так:

```
class АбстрактныйБазовый
{

    public:
    virtual void СделатьНечто() = 0; // чистый виртуальный метод
};
```

Абстрактные классы и чистые виртуальные функции

Это объявление, по существу, говорит компилятору о том, что метод *СделатьНечто* () должен быть реализован классом, который наследует класс *АбстрактныйБазовый:*

```
class Производный: public АбстрактныйБазовый
{
public:
    void СделатьНечто() // чистый виртуальный метод
    {
        cout << "Implemented virtual function" << endl;
    }
}
```

Абстрактные классы и чистые виртуальные функции

Таким образом, класс *АбстрактныйБазовый* выполнил свою задачу — заставил класс *Производный* предоставить реализацию для виртуального метода *СделатьНечто* ().

Такая возможность базового класса потребовать поддержки методов с определенным именем и сигнатурой в производных классах обеспечивает *интерфейс* (interface).

Абстрактные классы и чистые виртуальные функции

Вернемся к классу F i s h . Предположим, что тунец не может плавать быстро, поскольку класс Tuna не переопределил метод F i s h : :S w i m ( ) . Это ошибка реализации и большой недостаток.

Абстрактные классы и чистые виртуальные функции

Сделав класс F i s h абстрактным базовым классом с чистой виртуальной функцией Swim (), мы гарантируем, что класс Tuna, производный от класса F i s h , реализует метод T u n a : : Swim ( ) , т.е. тунец будет плавать как тунец, а не как любая рыба.

ЛИСТИНГ 11.6 . Класс Fish как абстрактный базовый класс для классов Tuna и Carp

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

ЛИСТИНГ 11.6 . Класс Fish как абстрактный базовый класс для классов Tuna и Carp

```
10: class Tuna:public Fish
11: {
12: public:
13:     void Swim()
14:     {
15:         cout << "Tuna swims fast in the sea!" << endl;
16:     }
17: };
18:
```

ЛИСТИНГ 11.6 . Класс Fish как абстрактный базовый класс для классов Tuna и Carp

```
19: class Carp:public Fish
20: {
21:     void Swim()
22:     {
23:         cout << "Carp swims slow in the lake!" << endl;
24:     }
25: };
26:
27: void MakeFishSwim(Fish& inputFish)
28: {
29:     inputFish.Swim();
30: }
31:
```

ЛИСТИНГ 11.6 . Класс Fish как абстрактный базовый класс для классов Tuna и Carp

```
32: int main()
33: {
34:     // Fish myFish; // Ошибка, нельзя создать экземпляр
                        // абстрактного класса
35:     Carp myLunch;
36:     Tuna myDinner;
37:
38:     MakeFishSwim(myLunch);
39:     MakeFishSwim(myDinner);
40:
41:     return 0;
42: }
```

ЛИСТИНГ 11.6 . Класс Fish как абстрактный базовый класс для классов Tuna и Carp

Результат

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

ЛИСТИНГ 11.6 . Класс Fish как абстрактный базовый класс для классов Tuna и Carp

Анализ

Существенна первая (закомментированная) строка функции main() (строка 34). Это демонстрирует, что компилятор не позволит создать экземпляр класса F i s h . Он ожидает чего-то более конкретного, такого, как специализация класса F i s h (класса Tuna, например), что имеет смысл и в реальности. Благодаря чистой виртуальной функции F i s h : : Swim (), объявленной в строке 7, оба класса, Tuna и Carp, вынуждены реализовать методы T u n a : : Swim () и C a r p : : Swim () соответственно.

ЛИСТИНГ 11.6 . Класс Fish как абстрактный базовый класс для классов Tuna и Carp

Анализ

Строки 27-30, где реализован метод MakeFishSwim (Fish&), демонстрируют, что, хотя экземпляр абстрактного класса и не может быть создан, ссылку или указатель на него вполне можно использовать.

Таким образом, абстрактные классы — это очень хороший способ потребовать от всех производных классов реализации определенных функций. Если в классе T r o u t (форель), производном от класса F i s h , забыть реализовать метод T r o u t : : Swim (), компиляция потерпит неудачу.

Абстрактные классы и чистые виртуальные функции

ПРИМЕЧАНИЕ

Абстрактные базовые классы (Abstract Base Class) зачастую называют просто ABC.

Абстрактные классы и чистые виртуальные функции

Классы ABC накладывают на ваш проект или программу определенные ограничения.