Тема 2.1 Виртуальные классы и функции

Виртуальные функции. Описание виртуальных функций.

Виртуальный метод (**виртуальная функция**) – в объектно-ориентированном программировании метод (функция) класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения. Таким образом, программисту необязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно лишь знать, что объект принадлежит классу или наследнику класса, в котором метод объявлен.

Виртуальные методы – один из важнейших приёмов реализации полиморфизма. Они позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника. При этом базовый класс определяет способ работы с объектами и любые его наследники могут предоставлять конкретную реализацию этого способа.

Одни языки программирования (например, C++, C#) требуют явно указывать, что данный метод является виртуальным. В других языках (например, Java, Python) все методы являются виртуальными по умолчанию (но только те методы, для которых это возможно; например в Java методы с доступом private не могут быть переопределены в связи с правилами видимости).

Базовый класс может и не предоставлять реализации виртуального метода, а только декларировать его существование. Такие методы без реализации называются «чистыми виртуальными» (перевод англ. pure virtual) или абстрактными. Класс, содержащий хотя бы один такой метод, тоже будет абстрактным. Объект такого класса создать нельзя (в некоторых языках допускается, но вызов абстрактного метода приведёт к ошибке). Наследники абстрактного класса должны предоставить реализацию для всех его абстрактных методов, иначе они, в свою очередь, будут абстрактными классами.

Для каждого класса, имеющего хотя бы один виртуальный метод, создаётся таблица виртуальных методов. Каждый объект хранит указатель на таблицу своего класса. Для вызова виртуального метода используется такой механизм: из объекта берётся указатель на соответствующую таблицу виртуальных методов, а из неё, по фиксированному смещению, – указатель на реализацию метода, используемого для данного класса. При использовании множественного наследования ситуация несколько усложняется за счёт того, что таблица виртуальных методов становится нелинейной.

Пример на C++, иллюстрирующий отличие виртуальных функций от невиртуальных:

Предположим, базовый класс Animal (животное) может иметь виртуальный метод eat (кушать). Подкласс (класс-потомок) Fish (рыба) переопределит метод eat() не так как его переопределит подкласс Wolf (волк), но можно вызвать eat() на любом экземпляре класса, унаследованного от класса Animal, и получить поведение eat(), соответствующее данному подклассу.

```
class Animal {
public:
 void /*невиртуальный*/ move(void) {
 std::cout << "This animal moves in some way" << std::endl;
 }
 virtual void eat(void) {}
};

// Класс "Animal" может обладать определением метода eat() при необходимости.
class Llama : public Animal {
public:
 // Невиртуальный метод move(), унаследован, но не переопределён
 void eat(void) {
 std::cout << "Llamas eat grass!" << std::endl;
 }
};
```

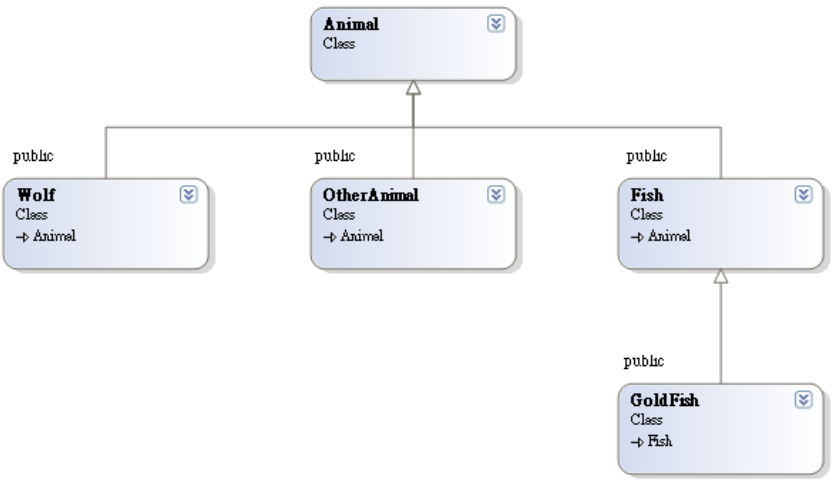

Рисунок 1 - Диаграмма класса Animal

Нестатические компонентные функции класса. Вызов виртуальных функций в конструкторе. Ограничения на использование виртуальных функций.

К механизму виртуальных функций обращаются в тех случаях, когда в базовый класс необходимо поместить функцию, которая должна по-разному выполняться в производных классах. Точнее, по-разному должна выполняться не единственная функция из базового класса, а в каждом производственном классе требуется свой вариант этой функции.

Виртуальными могут быть не любые функции, а только нестатические компонентные функции какого-либо класса. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор virtual может не использоваться.

В производном классе нельзя определять функцию с тем же именем и с тем же набором параметров, но с другим типом возвращаемого значения, чем у виртуальной функции базового класса. Это приводит к ошибке на этапе компиляции.

Если в производном классе ввести функцию с тем же именем и типом возвращаемого значения, что и виртуальная функция базового класса, но с другим набором параметров, то эта функция производного класса не будет виртуальной. В этом случае с помощью указателя на базовый класс при любом значении этого указателя выполняется обращение к функции базового класса (несмотря на спецификатор virtualи присутствие в производном классе похожей функции).

Простое наследование, множественное наследование.

Простое наследование

Класс, от которого произошло наследование, называется базовым или родительским. Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derived class).

В некоторых языках используются абстрактные классы. Абстрактный класс – это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

Множественное наследование

Множественное наследование – потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов. Практически всегда можно обойтись без использования данного механизма. Однако, если такая необходимость все-таки возникла, то, для разрешения конфликтов использования наследованных методов с одинаковыми именами, возможно, например, применить операцию расширения видимости – «::» – для вызова конкретного метода конкретного родителя.

Попытка решения проблемы наличия одинаковых имен методов в предках была предпринята в языке Eiffel, в котором при описании нового класса необходимо явно указывать импортируемые члены каждого из наследуемых классов и их именование в дочернем классе.

Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и др.) поддерживают возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним и тем же классом. Этот механизм позволяет во многом заменить множественное наследование – методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

Пример наследование в C++:

```
class A{ //базовый класс
};

class B : public A{ //public наследование
};

class C : protected A{ //protected наследование
};

class Z : private A{ //private наследование
};
```

Вызов полиморфных функций базового класса. Вызов полиморфных функций через базовые классы.

В ООП полиморфизм достигается не только описанным выше механизмом наследования и перегрузки методов родителя, но и *виртуализацией,* позволяющей родительским функциям обращаться к функциям потомков.

Полиморфизм реализуется через архитектуру класса, но полиморфными могут быть только функции-члены.

В C++ полиморфная функция привязывается к одной из возможных одноименных функций только в момент исполнения, когда ей передается конкретный объект класса. Другими словами, вызов функции в исходном тексте программы лишь обозначается, без точного указания на то, какая именно функция вызывается. Такой процесс известен как *позднее связывание.* Листинг 3.9 показывает, к чему может привести не полиморфное поведение обычных функций-членов.

I class Parent { **public:**

double Fl(double x) { return x*x; }

double F2(double x) { return Fl(x)/2; }

class Child **: public** Parent { **public:**

double Fl(double x) { return x*x*x; } };

void main() {

Child child;

cout “ child.F2(3) “ end1;

}