НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Лекции по информатике и программированию

Лекция 7

Наследование и полиморфизм (часть 2)

Содержание

10. Виртуальные методы и динамический полиморфизм

10.1. Пример наследования методов

При наследовании объект-потомок реагирует на сообщения, определенные для его предка, точно так же, как на них реагировал бы объект-предок. Это происходит поскольку объект-потомок выполняет методы, унаследованные им от объекта-предка.

Но часто необходимо, чтобы объект-потомок реагировал на такие сообщения своим собственным способом.

Механизм ***виртуальных методов*** позволяет объектам-потомкам изменить поведение отдельных методов, унаследованных от объекта-предка.

Рассмотрим пример решения задачи описания графических элементов для создания сцены в компьютерной графике. Необходимо **описать графические объекты** различного типа (класса):

- *точки* (*пиксели*),
- *окружности*,
- *линии*,
- *ломаные*,
- *многоугольники* и др.

Основные действия с такими графическими объектами:

- показать объект на экране (`Show`),
- убрать объект с экрана (`Hide`),
- переместить объект на новое место (`Move`).

Сначала опишем класс отдельных *пикселей* (*pixel* – сокращение от «**pic**ture **el**ement»). У пикселя имеются две координаты (`x` и `y`) и цвет (`color`).

```
class Pixel {
    double x, y;  // координаты объекта
    int color;    // цвет объекта
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) { }  // конструктор
    void Show();  // показать объект
    void Hide();  // убрать объект
    void Move(double nx, double ny);  // переместить объект
};
```

Допустим, что методы `Show` и `Hide` уже описаны Используя их, опишем метод перемещения объекта `Move`:

```
void Pixel::Move(double nx, double ny) {
    Hide();  // убрать объект
    x = nx;  // изменить координату x объекта
    y = ny;  // изменить координату y объекта
    Show();  // снова показать объект
}
```

Теперь опишем класс для *окружности*, которая помимо двух координат центра и цвета имеет еще радиус (`radius`). Поэтому воспользуемся описанным классом `Pixel` в качестве класса-предка и **унаследуем** от него класс-потомок `Circle`.

(Замечание: такое наследование **нарушает принципы ООП**: окружность не является частным случаем точки (пикселя). Исправим это в дальнейшем, а пока для простоты рассматриваемого примера выполним наследование окружности от пикселя.)

Для такого наследования поля класса-предка `Pixel` необходимо сделать доступными классу-потомку `Circle`, например, с помощью режима `protected` (это простой, но не лучший способ):

```
class Pixel {
protected:
    double x, y;  // координаты объекта
    int color;    // цвет объекта
public:
    ...
};
```

В классе-потомке `Circle` для окружности придется реализовать свои методы (отличные от методов класса `Pixel`): конструктор (для инициализации дополнительного поля `radius`), методы `Show` и `Hide` для прорисовки и стирания окружности.

```
class Circle : public Pixel {
    double radius;  // радиус окружности
public:
    Circle(double x, double y, double rad, int color)
        : Pixel(x, y, color), radius(rad) { }  // конструктор
    void Show();
    void Hide();
};
```

А метод `Move` для окружности **ничем не отличается** от метода `Move` для пикселя, поэтому хотелось бы не описывать его заново, а воспользоваться унаследованным от класса-предка `Pixel`. Но здесь возникает **проблема**: вызов метода `Move` для объекта класса `Circle` возможен, но он не приведет к ожидаемым результатам.

Дело в том, что внутри метода `Move` происходит вызов методов `Hide` и `Show`, а при компиляции тела метода `Pixel::Move` компилятор заменит их на адреса методов

`Pixel::Hide` и `Pixel::Show` класса `Pixel`. Самому классу `Pixel` ничего не известно о существовании у него наследников, поэтому при вызове метода `Circle::Move` прорисовываться и стираться будет не окружность, а пиксель.

Описанную проблему невозможно решить методами статического полиморфизма, при котором подстановка нужных адресов происходит на этапе компиляции, до запуска программы.

10.2. Виртуальные методы

Решить возникшую проблему в языке Си++ позволяет механизм ***виртуальных методов***. Виртуальным (`virtual`) объявляется метод, для которого:

- **аналогичный метод** будет определен для объектов классов-потомков;
- для потомков **метод должен выполняться иначе**, чем для класса-предка.

В ООП ***виртуальным методом*** задается реакция объекта некоторого класса на определенный тип сообщений в случае, если:

- предполагается, что у данного класса будут **классы-потомки**;
- объекты классов-потомков будут **способны получать сообщения того же типа**;
- объекты некоторых или всех классов-потомков будут **реагировать на эти сообщения иначе,** чем это делает объект класса-предка.

В Си++ можно объявить виртуальным (`virtual`) любой метод, кроме конструкторов и статических (`static`) методов. При компиляции тип объекта, для которого вызывается виртуальный метод, может быть как классом-предком, так и классом-потомком.

В рассматриваемом примере:

```
class Pixel {
    ...
    virtual void Show();   // виртуальный метод
    virtual void Hide();   // виртуальный метод
    ...
};
```

Объявления методов `virtual` позволяют из унаследованного метода `Circle::Move` вызвать методы `Circle::Hide` и `Circle::Show` класса `Circle`.

В этой ситуации компилятор генерирует код сложнее, чем для обычного метода.

10.3. Таблица виртуальных методов

Если в классе описан хотя бы один виртуальный метод, то для класса создается (в единственном экземпляре) неизменяемая ***таблица виртуальных методов*** (*virtual method table,* VMT), содержащая указатели на каждый из описанных в классе виртуальных методов. Компилятор вставляет во все объекты этого класса **невидимое** поле – ***указатель на таблицу виртуальных методов*** (*VMT pointer,* `vmtp`).

Когда компилятор встречает вызов виртуального метода, он вставляет в объектный код программы последовательность инструкций:

- извлечь из объекта значение поля `vmtp`;
- обратиться по полученному адресу к таблице виртуальных методов VMT;
- взять из VMT адрес нужного метода;
- обратиться к методу, используя взятый адрес.

Для каждого класса-потомка создается своя собственная таблица виртуальных методов VMT (в одном экземпляре), содержащая адреса соответствующих версий виртуальных методов. Значение адреса этой таблицы заносится в невидимые поля `vmtp` каждого объекта класса-потомка. За инициализацию полей `vmtp` отвечает конструктор класса, в начало которого компилятор вставляет соответствующие команды.

В классе, в котором имеется хотя бы один виртуальный метод, рекомендуется **всегда явно описывать деструктор** и объявлять его **виртуальным**. Зачем это нужно, станет ясно позднее.

Полное описание классов для рассматриваемого примера:

```
class Pixel {
protected:
    double x, y;  // координаты объекта
    int color;    // цвет объекта
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) { }  // конструктор
    virtual ~Pixel() { }  // деструктор
    virtual void Show();  // показать пиксель
    virtual void Hide();  // убрать пиксель
    void Move(double nx, double ny);  // переместить объект
};

void Pixel::Move(double nx, double ny) {
    Hide();  // убрать объект
    x = nx;  // изменить координату x объекта
    y = ny;  // изменить координату y объекта
    Show();  // снова показать объект
}
```

```
class Circle : public Pixel {
    double radius;  // радиус окружности
public:
    Circle(double x, double y, double rad, int color)
        : Pixel(x, y, color), radius(rad) { }  // конструктор
    virtual ~Circle() { }  // деструктор
    virtual void Show();   // показать окружность
    virtual void Hide();   // убрать окружность
};
```

В описании методов класса `Circle` ключевое слово `virtual` можно не указывать. Методы, совпадающие по профилю с виртуальными методами класса-предка, объявляются виртуальными автоматически.

10.4. Вызов виртуальных методов

Если объект описан в виде обычной переменной типа класс, и обращение к его виртуальному методу происходит напрямую через имя этой переменной и точку:

```
Pixel obj;
...
obj.Show();  // Всегда вызывается Pixel::Show,
             // виртуальность не задействована.
```

то **механизм виртуальности не задействован**. Поскольку тип переменной известен во время компиляции и стать другим уже не может.

Но если **тип объекта**, для которого вызывают виртуальный метод, **может меняться** (хотя бы теоретически), то компилятор генерирует код для его вызова через таблицу виртуальных методов VMT **при любом** обращении к виртуальному методу.

Если обращение к виртуальному методу происходит через **адрес объекта** (*указатель* или *ссылку*), то виртуальный метод вызывается через VMT для этого объекта.

Например, для функции `func` с формальным параметром-указателем на `Pixel`:

```
void func(Pixel *ptr) {
    ...
    ptr->Show();   // Вызывается  Pixel::Show или Circle::Show,
                   // виртуальность задействована.
}
```

При вызове функции `func` для фактического параметра-указателя на `Circle`:

```
Circle *pobj;
...
func(pobj);  // В теле функции func вызывается метод Circle::Show.
```

Виртуальный метод вызывается через VMT в том числе при вызове метода из тел других методов, в которых объект идентифицируется указателем `this`.

Эти правила для *указателей* аналогично действуют и для *ссылок* `Pixel&` и `Circle&`.

10.5. Чисто виртуальные методы

Исправим упомянутое ранее нарушение принципов ООП, когда *окружность* была унаследована от *пикселя*, не являясь частным случаем пикселя.

Заметим, что *окружность* и *пиксель* – частные случаи **абстрактной** *геометрической фигуры*, у которой имеются две координаты **точки привязки** (`x` и `y`) и **цвет** (`color`). В качестве точки привязки можно взять координаты:

- всей фигуры (для *пикселя*),
- центра (для *окружности*),
- центра пересечения диагоналей (для *прямоугольника*),
- одной из вершин (для *треугольника* или *прямоугольника*) и т.д.

Для такой геометрической фигуры алгоритм перемещения по экрану останется таким же, как описанный ранее метод `Move`. Однако тела методов `Show` и `Hide` для геометрической фигуры описать **невозможно** (абстрактная фигура не имеет формы), но они должны быть обязательно описаны во всех классах-потомках.

Теперь с учетом этих соображений опишем класс геометрической фигуры

```
class GeomFigure {
protected:
    double x, y;  // координаты объекта
    int color;    // цвет объекта
public:
    GeomFigure(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) { }  // конструктор
    virtual ~GeomFigure() { } // виртуальный деструктор
    virtual void Show() = 0;  // показать объект, чисто виртуальный метод
    virtual void Hide() = 0;  // убрать объект, чисто виртуальный метод
    void Move(double nx, double ny);  // переместить объект
};
```

```
void GeomFigure::Move(double nx, double ny) {
    Hide();  // убрать объект
    x = nx;  // изменить координату x объекта
    y = ny;  // изменить координату y объекта
    Show();  // снова показать объект
}
```

Тело метода `Move` не изменяется по сравнению с описанным ранее.

А методы `Show` и `Hide` объявляются ***чисто виртуальными методами*** (*pure virtual methods*). Тела этих методов в классе `GeomFigure` не описываются. На место тел чисто виртуальных методов помещается специальная лексическая последовательность «`= 0;`». Для каждого чисто виртуального метода в таблице VMT, компилятор резервирует позицию, в которой значение адреса остается нулевым. При этом предполагается, что методы `Show` и `Hide` будут существовать у потомков класса `GeomFigure`.

10.6. Абстрактные классы

Класс, в котором имеется хотя бы один чисто виртуальный метод, называется ***абстрактным классом***. Компилятор не позволяет создавать объекты абстрактных классов. Единственное назначение абстрактных классов – служить классами-предками для порождения классов-потомков, в которых все чисто виртуальные методы будут конкретизированы.

Если в классе-потомке не описывается тело хотя бы одного метода, объявленного чисто виртуальным в классе-предке, то такой класс-потомок считается тоже абстрактным.

Теперь перепишем иерархию классов в соответствии с принципами ООП унаследовав *пиксель* и *окружность* от *геометрической фигуры*:

```
class Pixel : public GeomFigure {
public:
    Pixel(double x, double y, int color)
        : GeomFigure(x, y, color) { }  // конструктор
    virtual ~Pixel() { }  // виртуальный деструктор
    virtual void Show();  // показать пиксель, виртуальный метод
    virtual void Hide();  // убрать пиксель, виртуальный метод
};
class Circle : public GeomFigure {
    double radius;  // радиус окружности
public:
    Circle(double x, double y, double rad, int color)
        : GeomFigure(x, y, color), radius(rad) { }  // конструктор
    virtual ~Circle() { }  // виртуальный деструктор
    virtual void Show();   // показать окружность, виртуальный метод
    virtual void Hide();   // убрать окружность, виртуальный метод
};
void Pixel::Show() {
    ...  // показать пиксель на экране
}
void Pixel::Hide() {
    ...  // убрать пиксель с экрана
}
void Circle::Show() {
    ...  // показать окружность на экране
}
void Circle::Hide() {
    ...  // убрать окружность с экрана
}
```

10.7. Виртуальность в конструкторах и деструкторах

Заполнение полей при создании объекта выполняет конструктор. В том числе конструктор заполняет значением невидимое поле `vmtp`, содержащее указатель на таблицу виртуальных методов VMT.

При создании объекта-потомка, вначале создается объект-предок, как составная часть объекта-потомка. При таком создании объекта-предка срабатывает конструктор объекта-предка, который должен заполнить поле `vmtp` объекта-предка, но ничего не знает о потомках. Поэтому конструктор объекта-предка вносит в поле `vmtp` адрес таблицы VMT объекта-предка.

Если в этот момент своего выполнения конструктор объекта-предка вызовет виртуальный метод, то сработает метод, описанный в классе-предке, а не классе-потомке. Можно считать, что **во время выполнения конструктора механизм виртуальности не задействован**. Поэтому **из конструктора нельзя вызывать чисто виртуальные методы**.

Только после окончания создания объекта-предка конструктор объект-потомка, заполняя поля объекта-потомка, изменяет значение поля на адрес VMT объекта-потомка.

При выполнении деструктора происходит аналогичный процесс, но в обратном порядке. После завершения своего тела деструктор объекта-потомка заносит в поле `vmtp` адрес таблицы VMT объекта-предка. Поэтому сказанное для конструктора относится так же к деструктору.

Не следует обращаться к виртуальным методам из **конструкторов** и **деструкторов**.

10.8. Наследование ради конструктора

Чтобы задать объект типа *ломаная линия*, опишем класс `Polyline`, который хранит **список** координатных пар, задающих смещение каждой вершины ломаной относительно точки привязки. Для организации списка в приватной части класса опишем структуру, задающую элемент списка:

```
class Polyline : public GeomFigure {
    struct Vertex {     // описание типа элемента списка вершин
        double dx, dy;  // смещение вершины относительно точки привязки
        Vertex *next;   // указатель на следующую вершину списка
    };
    Vertex *first;  // указатель на начало списка вершин
public:
    Polyline(double x, double y, int color)
        : GeomFigure(x, y, color), first(0) { }  // конструктор
    virtual ~Polyline();  // виртуальный деструктор
    void AddVertex(double adx, double ady);  // добавить вершину
    virtual void Show();  // показать ломаную, виртуальный метод
    virtual void Hide();  // убрать ломаную, виртуальный метод
};
```

Поскольку количество вершин ломаной не известно заранее, то вначале (в конструкторе) будет создаваться ломаная, не имеющая ни одной вершины. А для добавления каждой новой вершины ломаной опишем метод `AddVertex`, выделяющий динамическую память:

```
void Polyline::AddVertex(double adx, double ady) {
    Vertex *temp = new Vertex;  // выделение памяти для новой вершины
    temp->dx = ax;
    temp->dy = ay;
    temp->next = first;
    first = temp;  // добавление новой вершины в начало списка
}
```

Тогда нужно прописать в деструкторе удаление списка и освобождение динамической памяти:

```
void Polyline::~Polyline() { // деструктор удаляет список вершин
    while (first) {  // цикл по всем вершинам списка
        Vertex *temp = first;  // текущая вершина берется из начала списка
        first = first->next;   // переход к следующей вершине
        delete temp;           // удаление текущей вершины
    }
}
```

Теперь для объектов типа *квадрат* опишем класс `Square`, унаследовав его от класса `Polyline`, поскольку квадрат можно представить как частный случай ломаной. Для такого квадрата ломаная содержит пять вершин: начинается в точке привязки, что соответствует смещению $(0, 0)$, проходит через точки $(a, 0)$, (a, a), $(0, a)$ и возвращается в точку привязки $(0, 0)$, замыкаясь на свое начало. Поэтому в классе-потомке `Square` достаточно описать только конструктор, который пять раз вызывает метод `AddVertex` для создания списка вершин:

```
class Square : public Polyline {
public:
    Square(double x, double y, int color, double a)
            : Polyline (x, y, color) {  // конструктор
        AddVertex(0, 0);
        AddVertex(0, a);
        AddVertex(a, a);
        AddVertex(a, 0);
        AddVertex(0, 0);
    }
};
```

Ничего более в классе `Square` описывать не требуется.

Наследованием ради конструктора называют упрощенный случай наследования, при котором класс-потомок не вводит новых полей и методов, а отличается от класса-предка только конструктором (или набором конструкторов). Такой класс-потомок создается, чтобы не повторять одни и те же действия при конструировании однотипных объектов.

10.9. Виртуальный деструктор

Ранее отмечалось, что если в классе имеется хотя бы один виртуальный метод, то рекомендуется **всегда явно описывать деструктор** и объявлять его **виртуальным**. Теперь объясним зачем это нужно.

При активном использовании полиморфизма часто приходится применять `delete` к указателю, имеющему тип «указатель на класс-предок», притом что указывать он может и на объект класса-потомка. В этой ситуации требуется вызвать деструктор, соответствующий типу уничтожаемого объекта, а не указателя на него.

Например, через указатель на `GeomFigure` создадим объект класса `Square` в динамической памяти, а затем уничтожим его:

```
GeomFigure *ptr;  // Указатель на объект класса GeomFigure.
ptr = new Square(3.4, 5.6, 0xff0000, 2.0);  // Выделение памяти для
...                                          // объекта класса Square.
delete ptr;  // Уничтожение объекта класса Square.
```

Поскольку деструктор класса `GeomFigure` является виртуальным, то его вызов произойдет через таблицу VMT уничтожаемого объекта. При этом вызывается деструктор класса `Square`, который вызовет деструктор класса `Polyline`, а тот в свою очередь вызывает деструктор класса `GeomFigure`. Правильный последовательный вызов деструкторов приведет к удалению из динамической памяти списка вершин ломаной, созданного при работе конструктора класса `Square`.

Если бы деструктор класса `GeomFigure` не был объявлен виртуальным, то вызов деструктора произошел бы по типу указателя `ptr`, то есть – только для класса `GeomFigure`. При этом динамическая память, выделенная для списка вершин ломаной, не освобождается, а доступ к ней теряется. Происходит «пожирание» динамической памяти.

Деструктор любого класса, имеющего хотя бы одну виртуальную функцию, следует **всегда** объявлять как **виртуальный** (не задумываясь, понадобиться это или нет).

10.10. Динамический полиморфизм

Пусть графическая сцена состоит из разных геометрических фигур, вводимых одновременно. При этом на момент написания программы неизвестно, сколько и каких объектов будет в сцене. Такое может быть, если описание сцены содержится во внешнем источнике (например, считывается из файла) или генерируется случайно.

В некоторый момент выполнения программы становится известно, сколько объектов содержит сцена. Создадим динамический массив указателей на объекты-потомки абстрактного класса `GeomFigure`:

```
int count;  // количество объектов в сцене
GeomFigure **scene;  // указатель на массив объектов
...
scene = new GeomFigure*[count];  // выделение памяти для массива указателей
...
```

Благодаря полиморфизму можно создавать различные объекты-потомки, помещая указатели на них в массив `scene`:

```
scene[i] = new Pixel(2.25, 10.75, 0xff0000);
...
scene[j] = new Circle(30.4, 25.7, 0x005500, 3.5);
...
scene[k] = new Square(22.3, 40.1, 0x005500, 10.0);
...
```

Независимо от типов объектов, можно помещать, удалять и перемещать объекты, используя виртуальные методы `Show`, `Hide` и метод `Move` соответственно.

С помощью виртуального деструктора можно уничтожить все объекты:

```
for (int i=0; i<count; ++i)  // цикл по всем объектам
    delete scene[i];  // уничтожение каждого объекта
delete [] scene;  // уничтожение сцены
```

Конкретные методы, которые нужно вызывать, становятся известны только во время выполнения программы, поэтому такой вид полиморфизма называется ***динамическим полиморфизмом***.

10.11. Деструкторы `private` и `protected`

Если описать деструктор в секции `private`, то уничтожение объекта описываемого класса становится возможным только в его методах (или в дружественных функциях). Описание деструктора в секции `protected`, добавляет возможность уничтожения объекта еще и в методах классов-потомков.

Объект класса, у которого деструктор является приватным, **нельзя** создать в виде **простой** (локальной или глобальной) **переменной**, объявляемой за пределами его методов и друзей. Ведь для такой переменной компилятор не сможет вставить в код вызов деструктора (для локальных переменных – при завершении функции, для глобальных – при завершении выполнения программы).

Такой прием используют, чтобы создать класс объекта, **всегда** размещаемого в **динамической памяти**. Обычно такие объекты при некоторых обстоятельствах удаляют себя сами, например, с помощью специального описанного метода:

```
class MyClass {
    ...
    ~MyClass() { }  // приватный деструктор
public:
    ...
    void Destruct() {  // метод уничтожения объекта
        delete this;   // освобождение динамической памяти
    }
};
```

Если в классе описан приватный (`private`) деструктор, то **наследование** от такого класса **невозможно**.

Обычно, если деструктор описан как защищенный (`protected`), то класс предназначен для создания наследников, а использование самого класса не предполагается. Если в классах-потомках деструкторы так же являются защищенными (`protected`), то объекты такой иерархии классов предназначены для существования **только в динамической памяти**.