Лекция 16

Виртуальные функции и абстрактные классы

16.1. Виртуальные функции

Рассмотрим случай, когда в базовом и производном классах объявлены одинаковые функции (одинаковое имя, одинаковое количество и типы аргументов).

Программа 16.1. Невиртуальные функции

```
// Файл NoVirtF.cpp

#include <iostream>
using namespace std;

struct base{                                // Базовый класс base
    void fun(int i)                         // Функция базового класса
    { cout << "base::i = " << i << endl; }
};

struct derive : public base{                // Производный класс derive
    void fun(int i)                         // Функция производного класса
    { cout << "derive::i = " << i << endl; }
};

void main()
{
    base B, *pb = &B;               // Объект base и указатель на base
    derive D, *pd = &D;             // Объект derive и указатель на derive
    base* pbd = &D;                 // Указатель на базовый класс base
                                    // инициализируется адресом объекта
                                    // производного класса derive
    pb->fun(1);  // pb  - указатель на базов. класс,  вывод: base::i   = 1
    pd->fun(5);  // pb  - указатель на произв. класс, вывод: derive::i = 5
    pbd->fun(4); // pdb - указатель на базовый класс, вывод: base::i   = 4;
}
```

Программа выводит:

```
base::i = 1
```

```
derive::i = 5
base::i = 4
```

Указателю на *базовый* класс можно присвоить адрес объекта *производного* класса, как это делано в инструкции:

```
base* pbd = &D;
```

Это допустимо, так как в производном классе есть все члены, которые есть и в базовом. Обратное невозможно, т.е. *указателю на производный класс* нельзя присвоить *адрес объекта базового класса*, т.к. в базовом классе, вообще говоря, нет всех компонент, которые есть в производном. Например:

```
base B;          // Объект базового класса
derive *pd;      // Указатель на производный класс
pd = &B;         // Ошибка! Нельзя указателю на производный класс
                 // присвоить адрес объекта базового класса
```

Так как pbd есть указатель на *базовый* класс, при обращении

```
pbd -> fun(4);
```

вызывается функция *базового* класса base::fun(), а не функция производного класса derive::fun(), несмотря на то, что указатель pbd имеет значение адреса объекта производного класса.

Выбор нужной функции производится при компиляции программы в соответствии с *типом* указателя, а не с его *значением*, которое может быть разным в разные моменты работы программы. Такой режим называется *ранним* или *статическим* связыванием.

Присваивание указателю на базовый класс адреса объекта производного класса может быть вызвано естественным желанием вызвать функцию *производного* класса, но это желание не будет выполнено для обычных функций. Реализовать это желание можно с помощью *виртуальных* функций, для которых реализуется *позднее* или *динамическое* связывание, производимое в ходе работы программы.

Программа 16.2. Виртуальные функции

```
// Файл VirtualFunc.cpp

#include <iostream>
using namespace std;

struct base{                          // Базовый класс

    virtual void vfun(int i)          // Виртуальная функция базового класса
    { cout << "base::i = " << i  << endl; }
};
```

```
struct derive1 : public base{        // Производный класс derive1

    virtual void vfun(int i)         // Переопределение виртуальной функции
    { cout << "derive1::i = " << i << endl; }
};

struct derive2 : public base{        // Еще один производный класс derive2

    virtual  void vfun(int i)        // Переопределение виртуальной Функции
    { cout << "derive2::i = " << i << endl; }
};

void main()
{
    base B, *pb = &B;              // Объект base и указатель на base
    derive1 D1, *pd1 = &D1;        // Объект derive1 и указатель на derive1
    derive2 D2, *pd2 = &D2;        // Объект derive2 и указатель на derive2
    pb ->vfun(13);        // Выводит       base::i = 13
    pd1->vfun(2);         // Выводит       derive1::i = 2
    pd2->vfun(3);         // Выводит       derive2::i = 3
    pb = &D1;             // Указателю на base даем значение адреса derive1
    pb->vfun(13);         // Выводит       derive1::i = 13
    pb = &D2;             // Указателю на base даем значение адреса derive2
    pb->vfun(13);         // Выводит       derive2::i = 13
}
```

Программа выводит:

```
base::i = 13
derive1::i = 2
derive2::i = 3
derive1::i = 13
derive2::i = 13
```

В программе трижды повторяется одна и та же инструкция:

```
pb ->vfun(13);
```

При каждом ее выполнении получается разный результат, так как указатель pb принимает в ходе работы программы три разных значения.

Интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от *значения* указателя. В табл.16.1 указаны функции, вызываемые в зависимости от значения указателя.

Таблица 16.1. Вызовы виртуальной функции

Значение указателя pb	Функция, вызываемая инструкцией pb->vfun()
&B	base::vfun()
&D1	derive1::vfun()
&D2	derive2::vfun()

Так как при компиляции программы невозможно предвидеть, какое значение примет указатель на базовый класс во время работы

программы, вопрос о выборе *виртуальной* функции решается на этапе *выполнения* программы и определяется *значением* указателя.

При вызове через указатель *невиртуальной* функции принимается во внимание *только тип*, а не *значение* указателя.

Виртуальными могут быть только функции-члены класса.

Если функция объявлена виртуальной в базовом классе, она будет виртуальной во всех производных класса даже без использования в них ключевого слова `virtual`. Однако повторное использование ключевого слова `virtual` делает программу более понятной.

16.2. Абстрактные классы

Абстрактным называется класс, в котором есть хотя бы одна *чистая виртуальная* функция. Чистой виртуальной функцией называется функция-член, которая не имеет тела:

```
virtual <ТИП><ИМЯ_ФУНКЦ>(<СПИСОК_ФОРМАЛЬН_ПАРАМ>) = 0;
```

Конструкция `= 0` называется *чистый спецификатор*. Он заменяет тело функции.

Например, следующая функция является чистой виртуальной:

```
virtual void fpure(void) = 0;
```

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Она служит основой для подменяющих ее функций в производных классах.

Абстрактный класс может быть использован только как базовый для производных классов. Нельзя создавать объекты абстрактных классов.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. Эти общие понятия, обычно, невозможно использовать непосредственно. Например, можно говорить об общем понятии «фигура» на плоскости. Это понятие включает свойства, присущие любой конкретной фигуре: треугольнику, прямоугольнику, окружности. Для любой фигуры можно указать ее местоположение с помощью координат некоторой точки. Но изобразить можно всегда только конкретную фигуру, а не фигуру вообще.

Программа 16.3. Абстрактный класс плоских фигур

В программе создан абстрактный класс `Figure` фигур на плоскости, который включает координаты некоторой центральной точки `x`, `y` и метод `Move()`, которые имеются у любой плоской фигуры. Метод `Show()`

для изображения фигуры объявлен как чисто виртуальный, из-за чего класс `Figure` становится абстрактным. На базе абстрактного класса `Figure` определены конкретные классы эллипсов и прямоугольников. Для рисования фигур на экране используется графическая система OpenGL.

Объявления классов помещены в файл `Figures.h`.

```
// Файл Figures.h

#ifndef FIGUREH
#define FIGUREH

#include <glut.h>
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
using namespace std;

class Figure{                          // Класс фигур на плоскости
protected:
    float x, y;                        // Координаты центральной точки фигуры
public:
    Figure(float xx, float yy) : x(xx), y(yy)         // Конструктор
    { }

    virtual void Show() = 0;           // Чистая виртуальная функция
                                       // рисования фигуры

    void Move(float dx, float dy)      // Переместить фигуру на dx и dy
    {
        x+=dx; y+= dy;                 // Изменяем координаты фигуры
        Show();                        // Показываем фигуру на новом месте
    }
};

class MyEllipse : public Figure{       // Класс эллипсов
    float a, b;                        // Полуоси эллипса
public:

    MyEllipse(float xx, float yy, float aa, float bb) :   // Конструктор
        Figure(xx, yy),            // Вызов конструктора базового класса
        a(aa), b(bb)               // Вызов конструкторов членов класса
    { }                            // Тело конструктора класса MyEllipse

    void Show();                   // Рисование эллипса
};

class MyRect : public Figure{      // Класс прямоугольников со сторонами,
                                   // параллельными осям координат
    float a, b;                    // Стороны прямоугольника
public:

    MyRect(float xx, float yx, float aa, float bb) :      // Конструктор
        Figure(xx, yx),            // Вызов конструктора базового класса
        a(aa), b(bb)               // Вызов конструкторов членов класса
```

```
    {}                            // Пустое тело конструктора класса MyRect

    void Show();                  // Рисование прямоугольника
};

#endif
```

Функция `Show()` сделана чисто виртуальной потому, что изобразить абстрактную фигуру или «фигуру вообще», невозможно, так как можно нарисовать только конкретные фигуры: прямоугольники, окружности и т.п. Эта функция используется в функции `Move()`, реализующей логику движения фигуры: изменить координаты и нарисовать фигуру в новом положении. В конкретных классах, производных от класса `Figure`, метод `Show()` следует переопределять, метод же `Move()` можно использовать из базового класса, так как логика движения любой фигуры одинакова.

Приведем реализацию классов прямоугольников `MyRect` и эллипсов `MyEllipse`. Заметим, что использование для названий классов более естественных имен (`Rectangle`, `Ellipse`) может привести к конфликту с такими именами, используемыми в библиотеках.

```
// Файл Figures.CPP

#include "Figures.h"

void MyRect::Show()                     // Рисование прямоугольника
{
    glClear(GL_COLOR_BUFFER_BIT);       // Очистка буфера
    glColor3f(1.0f, 0.0f, 0.0f);        // Установка текущего цвета (красный)

    glBegin(GL_LINE_LOOP);              // Начало группы вершин

        glVertex2f(x - a / 2, y - b / 2);     // Левый нижний угол
        glVertex2f(x - a / 2, y + b / 2);     // Левый верхний угол
        glVertex2f(x + a / 2, y + b / 2);     // Правый верхний
        glVertex2f(x + a / 2, y - b / 2);     // Правый нижний

    glEnd();                            // Конец группы вершин

    glFlush();                // Функция glFlush() требует начать рисование
}

void MyEllipse::Show()                  // Рисование эллипса
{
    const int NV = 100;                 // Число вершин для рисования эллипса
    glClear(GL_COLOR_BUFFER_BIT);       // Очистка буфера
    glColor3f(0.0f, 0.0f, 1.0f);        // Установка текущего цвета (синий)
    double df = 2.0 * M_PI / NV;        // Шаг по углу

    glBegin(GL_LINE_LOOP);              // Начало группы вершин

    for(int i = 0; i < NV; ++i){
        double f = df * i;                               // Текущий угол
        glVertex2f(x + a * cos(f), y + b * sin(f));    // Текущая вершина
    }
```

```
    glEnd();                            // Конец группы вершин

    glFlush();                  // Функция glFlush() требует начать рисование
}
```

При рисовании прямоугольника задаются его 4 вершины так, как это делалось при рисовании многоугольников в программе 11.11.

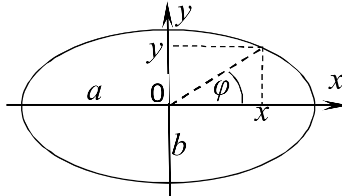

Рис. 16.1. Геометрия эллипса

При рисовании эллипса задаются `NV = 100` вершин, координаты которых вычисляются по формулам (рис.16.1):

$$x = a\cos\varphi,\ \ y = \ b\sin\varphi,$$

где a, b – полуоси эллипса.

Тестирующая программа сделана по образцу программы 11.11.

```
// Файл TestFigures.cpp

#include "Figures.h"

// Глобальные переменные
Figure* pp;                             // Указатель на текущую фигуру
Figure* ptr_rect;                       // Указатель на прямоугольник
Figure* ptr_ell;                        // Указатель на эллипс

void display()
{
    pp->Show();
}

const float STEP = 0.05f;               // Величина смещение

const unsigned char  ESC = 27;          // Код клавиши Esc

// OrdinaryKeys: вызывается при нажатии обычных клавиш
void OrdinaryKeys(unsigned char key, int x, int y)
{
    switch(key){
        case ESC:
            exit(0);                    // Завершение программы
        break;
        case '1':
            pp = ptr_rect;              // Выбираем прямоугольник
```

```
            pp ->Show();             // и рисуем
        break;
        case '2':
            pp = ptr_ell;            // Выбираем эллипс
            pp ->Show();             // и рмсуем
        break;
    }
}

// SpecialKeys: вызывается при нажатии функциональных и
// управляющих клавиш

void SpecialKeys(int key, int x, int y)
{
    switch(key){
        case GLUT_KEY_LEFT :         // Нажата стрелка влево
            pp->Move(-STEP, 0);      // Сдвиг фигуры влево
        break;
        case GLUT_KEY_RIGHT :        // Нажата стрелка вправо
            pp->Move(STEP, 0);       // Сдвиг фигуры вправо
        break;
        case GLUT_KEY_UP :           // Нажата стрелка вверх
            pp->Move(0, STEP);       // Сдвиг фигуры вверх
        break;
        case GLUT_KEY_DOWN :         // Нажата стрелка вниз
            pp->Move(0, -STEP);      // Сдвиг фигуры вниз
        break;
    }
}

int main(int argc, char* argv[])
{
    setlocale(LC_ALL, "Russian");
    MyEllipse E(0.0, 0.0, 0.8, 0.5); // Создаем эллипс
    MyRect R(0.0, 0.0, 1.0, 0.5);    // Создаем прямоугольник
    pp = ptr_rect = &R;              // Вначале работаем с прямоугольником
    ptr_ell = &E;
    cout << "Нажмите:\n"
        << "1 -   для работы с прямоугольником\n"
        << "2 -   для работы с эллипсом\n"
        << "Esc - для выхода\n";
    glutInit(&argc, argv);           // Инициализация системы glut

    glutCreateWindow("Прямоугольник и эллипс");// Создание окна с заголовком

    glutKeyboardFunc(&OrdinaryKeys); // Регистрация функции, обрабатывающей
                                     // нажатие обычных клавиш
    glutSpecialFunc(&SpecialKeys);   // Регистрация функции, обрабатывающей
                                     // нажатие специальных клавиш
    glutDisplayFunc(&display);       // Регистрация функции перерисовки окна
    glClearColor(1.0,1.0,1.0,1.0);   // Установка белог цвета фона
    glutMainLoop();                  // Цикл ожидания и обработки событий
    return 0;
}
```

Функции display(), OrdinaryKeys() и SpecialKeys() работают с указателем на абстрактный класс pp, которому в ходе работы программы присваивается значение либо указателя на прямоугольник ptr_rect, либо указателя на эллипс ptr_ell. Если будут еще добавлены классы плоских фигур, производные от абстрактного класса Figure, указанные функции будут работать и с ними без всяких изменений. В этом заключается сила наследования.

В графическом окне при нажатии цифры 1 появляется прямоугольник, а при нажатии цифры 2 – эллипс (рис.16.2). Фигуры можно перемещать, нажимая клавиши со стрелками. Программа завершается при нажатии Esc.

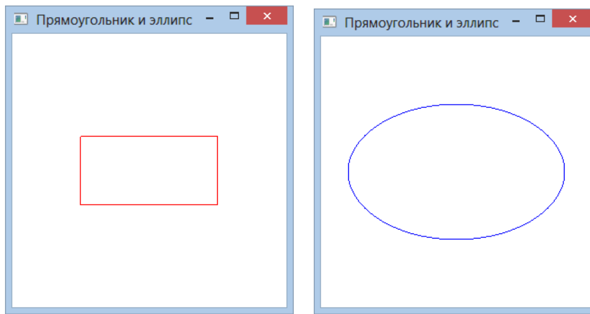

Рис. 16.2. Прямоугольник и эллипс

Вызов виртуальных функций

Функции класса можно вызывать для объекта класса (используя оператор "точка" (.)), через указатели на объекты класса (§16.1) (используя оператор "стрелка" (->)) и из других функций этого класса. Программа 16.3 иллюстрирует последний способ. В базовом классе Figure есть одна чистая виртуальная функция Show(), наличие которой позволяет написать в этом классе обычную функцию Move().

Производные классы MyEllipse и MyRect имеют собственную функцию Show(). Для объектов производных классов E и R вызывается функция базового класса Move(), которая в момент написания и компиляции класса Figure ничего «не знала» о будущих классах-потомках. Тем не менее, функция Move() вызывает при своей работе нужную функцию Show(), ориентируясь на *тип* объекта, для которого вызывается. Это реализуется благодаря тому, что функция Show() – виртуальная, и выбор нужной виртуальной функции решается на этапе выполнения программы, когда становится известен тип объекта, для которого вызывается функция.

16.3. Совместимость типов

При наследовании *производный* класс совместим с *базовым*, т.е. производные классы можно использовать вместо базовых. Это распространяется на экземпляры классов, указатели на объекты, формальные и фактические параметры. Производный класс включает в себя все члены-данные (поля) и все функции-члены (методы) базового класса и может быть что-то ещё, поэтому гарантируется, что при присваивании объекту базового класса объекта производного класса все поля будут заполнены. Обратное, т.е. присваивание объекту-потомку значения объекта-предка может оставить некоторые поля незаполненными, поэтому запрещено. Например, в программе 15.5 есть базовый класс

```
class Date{…};
```
и производный класс

```
class Pers: Date {…};
```

Рассмотрим фрагмент программы, где используются эти классы.

```
Date dt;                 // Дата, создаваемая конструктором по умолчанию
Pers VIL("Ленин В.И.", Date(22, 4, 1870)); // Переменная класса Pers
dt = VIL;                   // Допустимо, в dt скопируется дата из VIL
Date bds(21, 12, 1879);                    // Переменная класса Date
Pers SIV("Сталин И.В.", Date(22, 4, 1870));// Переменная класса Pers
SIV = bds;                                 // Недопустимо
```

Есть ограничения для абстрактных классов. Абстрактные классы нельзя использовать для задания типа параметра функции или типа значения, возвращаемого из функции, например,

```
void f(Figure ff);   // Запрещено, т.к. ff имеет тип абстрактного класса
Figure g();          // Нельзя возвращать значение абстрактного класса
```

Формальным параметром функции может быть *указатель абстрактного класса*. Тогда в качестве фактического аргумента в функцию передается указатель на объект производного класса. Например,

```
void fgh(Figure *pf)              // Функция с аргументом – указателем
{...}                             // на абстрактный базовый класс

MyEllipse elps(50, 50, 20, 10, GREEN);        // Объекты конкретных
MyRect rct(40, 30, 20, 10, RED);              // классов

fgh(&elps);                 // Вызов функции с аргументами, указывающими

fgh(&rct);                  // на объекты разных производных типов
```