# Лабораторная работа №5.
# Виртуальные функции

**Цель работы**: Изучение технологии создания виртуальных функций.

**Содержание**



---

## Общие сведения

---

Виртуальный означает видимый, но не существующий в реальности. Когда используются виртуальные функции, программа, которая, казалось бы, вызывает функцию одного класса, может в этот момент вызывать функцию совсем другого класса.

А для чего вообще нужны виртуальные функции? Представьте, что имеется набор объектов разных классов, но вам хочется, чтобы они все были в одном массиве и вызывались с помощью одного и того же выражения. Например, в графической программе есть разные геометрические фигуры: треугольник, шар, квадрат и т. д. В каждом из этих классов есть функция draw(), которая прорисовывает на экране фигуры.

Теперь, допустим, вам необходимо создать картинку, сгруппировав некоторые из этих элементов.

Как бы сделать это без лишних сложностей? Подход к решению этой задачи таков: создайте массив указателей на все неповторяющиеся элементы картинки:

```
shape* ptarr[100J; //массив из 100 указателей на фигуры
```

Если в этом массиве содержатся указатели на осе необходимые геометрические фигуры, то вся картинка может быть нарисована в обычном цикле:

```
for (int j=0; j<N; j++)
        ptarr[j]->draw();
```

Это потрясающая возможность! Абсолютно разные функции выполняются с помощью одного и того же вызова! Если указатель в массиве **ptarr** указывает на окружность, вызовется функция, рисующая окружность, если он указывает на треугольник, то рисуется треугольник.

Это называетя **полиморфизм** - функции выглядят одинаково, но реально вызываются разные функции, в зависимости от значения **ptarr[j]**. Полиморфизм — одна из ключевых особенностей объектно-ориентированного программирования (ООП) наряду с классами и наследованием.

Чтобы использовать полиморфизм, необходимо выполнять некоторые условия.

Во-первых, все классы (все эти треугольнички, окружности и т. д.) должны являться наследниками одного и того же базового класса.

Во-вторых, функция draw() должна быть объявлена виртуальной (virtual) в базовом классе.

Рассмотрим несколько небольших программ, которые в дальнейшем соберем в единый проект.

[ В начало ]

---

## Доступ к обычным методам через указатели

---

Пример 1. Рассмотрим, что будет, когда базовый и производные классы содержат функции с одним и тем же именем, и к ним обращаются с помощью указателей, но без использования виртуальных функций.

Листинг программыа **NOTVIRT1**

```
// notvirt.cpp
// Доступ к обычный функциям через указатели
#include <iostream>
        using namespace std;
        ////////////////////////////////////////////////////////
        class Base{                     //Базовый класс
          public: void show() {         //Обычная функция
                cout << "Base\n";
          }
        }
        ////////////////////////////////////////////////////////
        class Dervl  : public Base{     //Производный класс 1
          public: void snow(){
                cout << "Dervl\n";
          }
        };
        ////////////////////////////////////////////////////////
        class Derv2 : public Base{      //Производный класс 2
          public: void show(){
                cout << "Derv2\n";
          }
        );

////////////////////////////////////////////////////////
int main(){
        Dervl dvl;                      //Объект производного класса 1
        Derv2 dv2;                      //Объект производного класса 2
        Base* ptr;                      //Указатель на базовый класс
        ptr = &dvl;                     //Адрес dvl занести в указатель
        ptr->show();                  //Выполнить show()
        ptr = &dv2;                     //Адрес dv2 занести в указатель
        ptr->show();                  //Выполнить show()
        return 0;
}
```

Итак, классы Dervl и Derv2 являются наследниками класса Base. В каждом из этих трех классов имеется метод **show()**. В **main()** мы создаем объекты классов **Dervl** и **Derv2**, а также указатель на класс **Base**. Затем адрес объекта порожденного класса мы заносим в указатель базового класса:

```
ptr - &dvl:     //Адрес объекта порожденного класса заносим в //указатель базового
```

Теперь хорошо бы понять, какая же, собственно, функция **Base::show()** или **Dervl::show()** выполняется в этой строчке:

```
ptr->show();
```

Опять же, в последних двух строчках программы **NOTVIRT** мы присвоили указателю адрес объекта, принадлежащего классу **Derv2**, и снова выполнили:

```
ptr->show();
```

Так какая же из функций **show()** реально вызывается?

Результат выполнения программы дает простой ответ:

```
Base
Base
```

Таким образом, всегда выполняется метод базового класса. Компилятор не смотрит на содержимое указателя ptr, а выбирает тот метод, который удовлетворяет типу указателя.

Да, иногда именно это нам и нужно, но таким образом не решить поставленную в начале этой темы проблему доступа к объектам разных классов с помощью одного выражения.

[ В начало ]

## Доступ к виртуальным методам через указатели

Внесем в листинг программы изменение: поставим ключевое слово **virtual** перед объявлением функции **show()** в базовом классе.

Листинг программы **VIRT**

```
// virt.cpp
// Доступ к виртуальным функциям через указатели
#include <iostream.h>
  using namespace std:
  ////////////////////////////////////////////////////////////
  class Base{                          //Базовый класс
    public: virtual void show() {    //Виртуальная функция
          cout <<"Base\n"ж
        }
  };
  ////////////////////////////////////////////////////////////

  class Dervl: public Base {         //Производный класс 1
    public: void show(){
          cout << "Derv1\n";
    }
  }

  ////////////////////////////////////////////////////////////
  class Derv2 : public Base {       //Производный класс 2
    public: void show(){
          cout << "Derv2\n";
        }
  };

  ////////////////////////////////////////////////////////////
  int main(){
    Dervl dvl:                     //Объект производного класса 1
    Derv2 dv2:                     //Объект производного класса 2
    Base* ptr:                     //Указатель на базовый класс
    ptr = &dvl:                    //Адрес dvl занести в указатель
    ptr->show();                 //Выполнить show()
    ptr - &dv2:                    //Адрес dv2 занести в указатель
    ptr->show();                 //Выполнить show()
    return 0;
)
```

На выходе имеем:

```
Dervl
Derv2
```

Теперь выполняются методы производных классов, а не базового. Для этого мы изменили содержимое **ptr** с адреса из класса **Dervl** на адрес из класса **Derv2**, и изменилось реальное выполнение **show()**. Значит, один и тот же вызов

```
  ptr->show();
```

ставит на выполнение разные функции в зависимости от содержимого **ptr**. Компилятор выбирает функцию, удовлетворяющую тому, что занесено в указатель, а не типу указателя, как было в программе **N0TVIRT**.

[ В начало ]

## Позднее связывание

Возникает вопрос, как компилятор узнает, какую именно функцию ему компилировать? В программе **N0TVIRT** у компилятора нет проблем с выражением:

```
ptr->show();
```

- он всегда компилирует вызов функции **show()** из базового класса.

Однако в программе **VIRT** компилятор не знает, к какому классу относится содержимое **ptr**. Ведь это может быть адрес объекта как класса **Dervl**, так и класса **Derv2**. Какую именно версию **draw()** вызывает компилятор — тоже загадка. На самом деле компилятор не очень понимает, что ему делать, поэтому откладывает принятие решения до фактического запуска программы. А когда программа уже поставлена на выполнение, когда известно, на что указывает **ptr**, тогда будет запущена соответствующая версия **draw()**.

Такой подход называется поздним или динамическим связыванием. (Выбор функций в обычном порядке, во время компиляции, называется ранним связыванием или статическим связыванием). Позднее связывание требует больше ресурсов, но дает выигрыш в возможностях и гибкости.

[ В начало ]

## Абстрактные классы и чистые виртуальные функции

Рассмотрим класс **shape** из программы **MULTSHAP**.

Мы никогда не станем создавать объект из класса shape, разве что начертим несколько геометрических фигур — кругов, треугольников и т. п.

Базовый класс, объекты которого никогда не будут реализованы, называется абстрактным классом. Такой класс может существовать с единственной целью — быть родительским по отношению к производным классам, объекты которых будут реализованы. Еще он может служить звеном для создания иерархической структуры классов.

Обратим внимание на то, что объекты родительского класса не предназначены для реализации. Можно, конечно, заявить об этом в документации, но это никак не защитит базовый класс от использования не но назначению. Для того, чтобы защитить программу от такой ситуации достаточно ввести в класс хотя бы одну ***чистую виртуальную функцию***. ***Чистая виртуальная функция*** — это функция, после объявления которой добавлено выражение **=0**. Продемонстрируем это в примере VIRTPURE:

Листинг программы VIRTPURE

```
virtpure.cpp // Чистая виртуальная функция
include <iostream>
using namespace std:
  class Base{                          //базовый класс
    public: virtual void show() = 0;  //чистая виртуальная функция
  };
////////////////////////////////////////////////
  class Dervl: public Base{            //порожденный класс 1
    public: void show() {
          cout << "Dervl\n";
        }
  };

////////////////////////////////////////////////
  class Derv2: public Base{            //порожденный класс 2
    public: void show() {
      cout << "Derv2\n";
        };
  }

////////////////////////////////////////////////
int main(){
  // Base bad:          //невозможно создать объект
                        //из абстрактного класса
  Base* arr[2];         //массив указателей на базовый класс
  Dervl dvl;            //Объект производного класса 1
  Derv2 dv2;            //Объект производного класса 2
  arr[0] = &dvl;        //Занести адрес dvl в массив
  arr[l] = &dv2;        //Занести адрес dv2 в массив
  arr[0]->show();       //Выполнить функцию show()
  arr[l]->show();       //над обоими объектами
  return 0;
}
```

Здесь виртуальная функция **show()** объявляется так:

```
virtual void show()=0;  //чистая виртуальная функция
```

Знак равенства не имеет ничего общего с операцией присваивания. Нулевое значение ничему не присваивается. Конструкция =0 — это просто способ сообщить компилятору, что функция будет чистой виртуальной. Если теперь в **main()** попытаться создать объект класса **Base**, то компилятор выдаст ошибку, если объект абстрактного класса попытаются реализовать. Он выдаст даже имя чистой виртуальной функции, которая делает класс абстрактным. Помните, что хотя это только объявление, но определение функции **show()** базового класса не является обязательным.

Как только в базовом классе окажется чистая виртуальная функция, необходимо будет позаботиться о том, чтобы избежать ее употребления во всех производных классах, которые вы собираетесь реализовать. Если класс использует чистую виртуальную функцию, он сам становится абстрактным, никакие объекты из него реализовать не удастся (производные от него классы, впрочем, уже не имеют этого ограничения). Более из эстетических соображений, нежели из каких-либо иных, можно все виртуальные функции базового класса сделать чистыми.

Между прочим, мы внесли еще одно, не связанное с предыдущими, изменение в **VIRTPURE:** теперь адреса методов хранятся в массиве указателей и доступны как элементы этого массива. Обработка этого, впрочем, ничем не отличается от использования единственного указателя. **VIRTPURE** выдает результат, не отличающийся от **VIRT:**

```
Derv1
Derv2
```

[ В начало ]

---

## Виртуальные функции и класс *person*

---

Теперь, уже зная, что такое виртуальные функции, рассмотрим области их применения. Новая программа использует класс **person**, на основе которого созданы два новых класса: **student** и **professor**. Каждый из этих производных классов содержит метод **isOutstanding()**. С помощью этой функции администрация школы может создать список выдающихся педагогов и учащихся, которых следует наградить за их успехи.

Листинг программы **VIRTPERS**

```
// vitrpers.cpp
// виртуальные функции и класс person
#include <iostream>
using namespace std;
////////////////////////////////////////////
  class person{                       //класс person
    protected:
          char name[40];
    public:
          void getName(){
      cout << "Введите имя:";
          cin >> name;
    }
    void putName(){
          cout << "Имя:" <<name << endl;
        }
    virtual void getData{} * 0;       //чистые
    virtual bool isOutstandingO =0:   //виртуальные
                                      //функции
  }
////////////////////////////////////////////

  class student : public person {  //класс student
    private:
          float gpa;                  //средний балл
    public:
          void getData(){             //запросить данные об ученике у
                                      //пользователя
        person::getName();
            cout << "Средний балл ученика:";
            cin >> gpa;
```

```
        }
     bool isOutstanding() {
          return (gpa > 3.5) ? true : false:
        }
  }

////////////////////////////////////////////
  class professor : public person{ //класс professor
    private:
      int numPubs:                 //число публикаций
    public:
        void getData(){               //запросить данные о педагоге у
                                     //пользователя
        person::getName(){
          cout <<"Число публикаций:";
          cin >> gpa;
        }
        bool isOutstanding() {
          return (gpa > 100) ? true : false;
        }
  };
////////////////////////////////////////////
int main(){
  person* persPTR[100];                          //массив указателей на person
  int n = 0;                                     //число людей, вьесенных в список char choice:
  do{
    cout <<"Учащийся (s) или педагог (p): ";
        cin >> choice;
        if(choice=='s')                              //Занести нового ученика
      persPtr[n] = new  student;                 // в массив
        else                                         //Занести нового
      persPtr[n] = new professor;                //педагога в массив
          persPtr[n++]->getData():                   //Запрос данных о персоне
     cout <<"Ввести еще персону (y/n)? "; //создать еще
                                                 //персону
          cin >> choice;
  } while( choice—'y' );                         //цикл, пока ответ 'y'
  for(int j=0: j<n: j++){
    persPlr[j]->putName();                       //Вывести все имена.
        if(persPtr[j]->isOutstanding())              //сообщать о
        cout <<" Это выдающийся человек!\n";   //выдающихся
  }
  return 0;
}                                                //Конец main()
```

Классы

Класс person — абстрактный, так как содержит чистые виртуальные функции **getData()**u **isOutstanding()**. Никакие объекты класса **person** не могут быть созданы. Он существует только в качестве базового класса для **student** и **professor**. Эти порожденные классы добавляют новые экземпляры данных для базового класса. **Student** содержит переменную дра типа **float**, представляющую собой средний балл учащегося. В классе **Professor** мы создали переменную **numPubs** типа **int**, которая представляет собой число публикаций педагога. Учащиеся со средним баллом **свыше 3,5** и педагоги, опубликовавшие **более 100 статей**, считаются выдающимися. (Мы, пожалуй, воздержимся от комментариев по поводу критериев оценки. Будем считать, что они выбраны условно.)

Функция **isOutstanding()**

**IsOutstanding()** объявлена чистой виртуальной функцией в классе **person**. В классе **student** эта функция возвращает **true**, если балл больше **3,5**, в противном случае — **false**. В классе **professor** она возвращает **true**, если **numPubs** больше **100**.

Функция **GetData()** запрашивает у пользователя **GPA**, если она запускается для обслуживания класса **student**, или число публикаций для **professor**.

Программа main()

В функции **main()** мы вначале даем пользователю возможность ввести несколько имен учащихся и педагогов. К тому же программа спрашивает средний балл учащихся и число публикаций педагогов. После того, как пользователь закончит ввод данных, программа выводит на экран имена всех учащихся и педагогов, помечая выдающихся. Приведем пример работы программы:

```
Учащийся (s) или педагог Ср):  s
Введите имя: Сергеев Михаил
Средний балл ученика: 1.2

Ввести еще персону (у/п)? у
Учащийся (s) или педагог (р):  s
Введите имя: Пупкин Василий
Средний балл ученика: 3.9

Ввести еще персону (у/п)? у
Учащийся (s) или педагог (р): s
Введите имя: Борисов Владимир
Средний балл ученика: 4.9

Ввести еще персону (у/п)? у
Учащийся (s) или педагог  (р); р
Введите имя: Михайлов Сергей
Число публикаций: 714

Ввести еще персону (у/п)? у
Учащийся (s) или педагог (р): р
Введите имя:  Владимиров Борис
Число публикаций:  13

Ввести еще персону (у/п)? n

Имя: Сергеев Михаил
Имя-. Пупкин Василий Это выдающийся человек!
Имя; Борисов Владимир Это выдающийся человек!
Имя: Михайлов Сергей Это выдающийся человек!
Имя: Владимиров Борис
```

[ В начало ]

## Виртуальные функции в графическом примере

Рассмотрим еще один пример использования виртуальных функций.

Необходимо нарисовать некоторое число фигур с помощью одного и того же выражения. Это с успехом осуществляет программа **VIRTSHAP**.

Листинг программы **VIRTSHAP**

```
// vlrtshap.cpp
// Виртуальные функции и геометрические фигуры
#include <iostream>
using namespace std:
#include "msoftcon.h"      //для графических функций
////////////////////////////////////////////////////////
  class shape {                                    //базовый класс
    protected:
      int xCo, yCo:                                //координаты центра
      color fillcolor:                             //цвет
      fstyle fill style:                           //заполнение
    public:                                        //конструктор без аргументов
      shape():xCo(0),yCo(0),fillcolor(cWHITE),fillstyle(SOLID_FILL)
                {  }                                        //конструктор с четырьмя аргументами
    shape(int x, int y, color fc, fstyle fs): xCo(x),yCo(y),fillcolor(fc),fillstyle(fs)
                {  }
        virtual void draw() = 0{                       //чистая виртуальная функция
      set_color(fillcolor):
      set_fill_style(fillstyle);
    }
  }
////////////////////////////////////////////////////////
  class ball : public shape {
    private:
      int radius:                                  //центр с координатани(xCo, yCo)
    public:
```

```
         ball() : shape()                                  //конструктор без аргументов
       {    )
//конструктор с пятью аргументами
     ball(int x, int y,int r,color fc, fstyle fs): shape(x, y, fc, fs), radius(r)
                {  }
     void draw()  {                                      //нарисовать шарик
       shape::draw():
       draw circle(xCo, yCo.  radius):
     }
  }: ////////////////////////////////////////////////////
  class rect ; public shape {
    private:
      int width, height:                                 //(xCo. yCo) - верхний левый угол public:
      rect() : shape0, height(0), width(0)               //конструктор //без аргументов
            { }                                             //конструктор с шестью аргументами
      rect(int x, int y, int h, int w, color fc, fstyle fs) :
    shape(x, y, fc, fs), height(h), width(w)
          { }
        void draw() {                                       //нарисовать прямоугольник
      shape::draw();
      draw_rectangle(xCo, yCo, xCo+width, yCo+height);
      set_color(cWHITE);                                 //нарисовать диагональ
      draw line(xCo, yCo, xCo+width, yCo+height);
    }
  };

////////////////////////////////////////////////////////
  class tria : public shape {
     private:
       int height:                                       //(xCo. yCo) - вершина пирамиды
     public:
       tria() : shape(), height(0)                       //конструктор без аргументов
         { }                                             //конструктор с пятью аргументами
       tria(int x, int y, int h, color fc, fstyle fs) : shape(x, y, fc, fs). height(h)
         { }
           void draw() {                                     //нарисовать треугольник
         shape::draw();
         draw_pyramid(xCo. yCo. height):
       }
  };
////////////////////////////////////////////////////////
  int main(){
   int j;
   init_graphics();                                      //инициализация графики
   shape* pShapes[3];                                    //массив указателей на фигуры
   pShapes[0] = new ball(40,12,5,cBLUE,X_FILL);          //определить три фигуры
   pShapes[l] = new rect(12,7,10,15,cRED,SOLIDFILL):
   p$hapes[2] = new tria(60,7,11,cGREEN,MEDIUMFILL);
   for(j=0: j<3; j++) pShapes[j]->draw();                //нарисовать все фигуры
   for(j=0; j<3; j++) delete pShapes[j];                 //удалить все фигуры
   set_cursor_pos(l. 25);
   return 0;
}
```

В **main()** мы задаем массив **ptarr** указателей на фигуры. Затем создаем три объекта, по одному из каждого класса, и помещаем их адреса в массив. Теперь с легкостью можно нарисовать все три фигуры:

```
ptarr[j]->draw():
```

Переменная **j** при этом меняется в цикле.

Как видите, это довольно мощный инструмент для соединения большого числа различных графических элементов в одно целое.

[ В начало ]

## Виртуальные деструкторы

Деструкторы базового класса обязательно должны быть виртуальными!

Допустим, чтобы удалить объект порожденного класса, вы выполнили **delete** над указателем базового класса, указывающим на порожденный класс. Если деструктор базового класса не является виртуальным, тогда **delete**, будучи обычным методом, вызовет деструктор для базового класса вместо того, чтобы запустить деструктор для порожденного класса. Это приведет к тому, что будет удалена только та часть объекта, которая относится к базовому классу. Программа **VIRTDEST** демонстрирует это.

Листинг программы **NoVIRTDEST**

```
//novirtdest.cpp
//Тест невиртуальных и виртуальных деструкторов
#include <iostream>
using namespace std:
////////////////////////////////////////////////////////
  class Base {
    public:
      ~Base() {                         //невиртуальный деструктор
        cout << "Base удален\n";
      }
   };
////////////////////////////////////////////////////////
  class Derv : public Base {
    public: ~Derv(){
      cout << "Derv удален\n";
    }
  }:
////////////////////////////////////////////////////////
int main(){
  Base* pBase = new Derv;
  delete pBase;
  return 0;
)
```

Программа выдаст такой результат:

```
Base удален
```

Это говорит о том, что деструктор для **Derv** не вызывается вообще! К такому результату привело то, что деструктор базового класса в приведенном листинге невиртуальный.

Листинг программы **VIRTDEST**

```
//virtdest.cpp
//Тест невиртуальных и виртуальных деструкторов
#include <iostream>
using namespace std:
////////////////////////////////////////////////////////
  class Base {
    public:
      virtual ~Base(){                          //виртуальный деструктор
        cout << "Base удален\n";
      }
   };
////////////////////////////////////////////////////////
  class Derv : public Base {
    public: ~Derv(){
      cout << "Derv удален\n";
    }
  }:
////////////////////////////////////////////////////////
int main(){
  Base* pBase = new Derv;
  delete pBase;
  return 0;
)
```

Теперь результатом работы программы является:

```
Derv удален
Base удален
```

Только теперь обе части объекта порожденного класса удалены корректно. Конечно, если ни один из деструкторов ничего особенно важного не делает (например, просто освобождает память, занятую с помощью new), тогда их виртуальность перестает быть такой уж необходимой. Но в общем случае, чтобы быть уверенным в том, что объекты порожденных классов удаляются так, как нужно, следует всегда делать деструкторы в базовых классах виртуальными. Большинство библиотек классов имеют базовый класс, в котором есть виртуальный деструктор, что гарантирует нам наличие виртуальных деструкторов в порожденных классах.

[ В начало ]

---

### Индивидуальное задание

Пересмотреть проект, разработанный при выполнении лабораторной работы № 1 ( № 8). Разработать и программно реализовать иерархию классов с использованием виртуальных функций.

[ В начало ]