






Раздел «Язык Си» . OOP-Virt :

- [Виртуальные функции. Полиморфизм. Абстрактные классы.](#)
 - [Видео](#)
 - [Виртуальные функции.](#)
 - [Абстрактные классы](#)
 -  [Задачи](#)
 -  [Задача 1](#)
 - [Передача указателя на нестатические методы как параметра.](#)
 -  [Задача 2](#)
 -  [Задача 3](#)
 -  [Задача 4](#)

Виртуальные функции. Полиморфизм. Абстрактные классы.

Видео

- [Полиморфизм](#)
- [Виртуальные функции и рисование графики](#)

Виртуальные функции.

Объекты классов-наследников могут быть преобразованы к типу родительского класса. Также указатели на объекты классов-наследников могут быть преобразованы к указателям типа родительского класса.

При этом для "обычных" методов (функций) выполняется следующее правило: вызывается метод именно того класса, к типу которого произошло преобразование.

Например.

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Person{
public:
    void print();
};

class Worker:public Person{
public:
    void print();
};

void Person::print(){
    cout<<" Person!!!"<<endl;
};

void Worker::print(){
    cout<<" Worker!!!"<<endl;
};

int main(){
    Person a;// персона
    Person *p;// укзатель на объект класса Person
    Worker b;// рабочий
    // печать от персоны
    // вызов функции print() класса Person
    a.print();
    // печать от рабочего
    // вызов функции print() класса Worker
```

```
>./myprog
Person!!!
Worker!!!
Person!!!
Person!!!
```

Поиск

Поиск

Раздел «Язык Си»

[Главная](#)
[Зачем учить C?](#)
[Определения](#)

Инструменты:

[Поиск](#)
[Изменения](#)
[Index](#)
[Статистика](#)

Разделы

[Информация](#)
[Алгоритмы](#)
[Язык Си](#)
[Язык Ruby](#)
[Язык](#)
[Ассемблера](#)
[El Judge](#)
[Парадигмы](#)
[Образование](#)
[Сети](#)
[Objective C](#)

[Logon>>](#)

```

    b.print();
    // преобразование b к классу Person
    // и печать. Работает функция
    // класса Person
    ((Person)b).print();

    // передача адреса b указателю на
    // объект класса Person
    // и печать. Работает функция
    // класса Person
    p = &b;
    p->print();
}

```

Однако часто возникает необходимость как-то определить тип исходного класса объекта и вызвать метод именно этого класса независимо от типа указателя на объект.

Для этого используются **виртуальные функции**.

Виртуальные функции объявляются виртуальными в классе-родителе. Информация про них запоминается в **таблицу виртуальных функций**.

Это позволяет C++ при вызове через указатель использовать функцию именно исходного класса независимо от того к какому типу указателя (в рамках наследования) произошло преобразования.

Например.

```

#include <iostream>
#include <cstdlib>
#include <fstream>

using namespace std;

// Родительский класс для
// классов Worker и Crow
class Anybody{
    int apple;
protected:
    static int basket;
public:
    Anybody();
    Anybody(int);
    int getApple();

    // функция печати объявлена виртуальной
    // она становится виртуальной для всех
    // классов-наследников
    virtual void put();
};

class Worker:public Anybody{
public:
    // у класса Worker не будет
    // конструктора по-умолчанию
    Worker(int);

    // переопределение виртуальной функции
    void put();
};

class Crow:public Anybody{
public:
    // у класса Crow не будет
    // конструктора с параметрами
    Crow();

    // переопределение виртуальной функции
    void put();
};

// Реализация функций родительского класса
Anybody::Anybody(){

```

```
    apple = rand() %50;
};
Anybody::Anybody(int a){
    apple = abs(a) % 50;
};

int Anybody::getApple(){
    return apple;
};

void Anybody::put(){
    cout<<"Anybody - apple: "<<apple<<endl;
};

// реализация функций класса Worker

// в конструкторе с параметром можно сразу
// указать значение параметра "по-умолчанию"
// тогда, если писать параметр при создании
// объекта, параметр будет равен этому
// значению "по-умолчанию"
Worker::Worker(int a = 49 ):Anybody(a){};

// переопределение виртуальной функции для класса Worker
void Worker::put(){
    cout<<"Рабочий: я собираю по "<<getApple()<<" яблок"<<endl;
};

// реализация метода класса Crow
Crow::Crow(){};

// переопределение виртуальной функции для класса Crow
void Crow::put(){
    cout<<"Ворона: я ворую по "<<getApple()<<" яблок"<<endl;
};

// Как работают виртуальные функции
int main(){
    Anybody a;
    // указатель на родительский класс
    Anybody *p;
    // конструктора "по-умолчанию" нет
    // но мы определили значение обязательного
    // параметра "по-умолчанию"
    // теперь можно его не указывать
    Worker w,
    // а здесь указали параметр
    Worker w1(22);
    Crow cr;
    // указателю на Anybody передали адрес a (Anybody)
    p = &a;
    // печать (вызов put() класса Anybody
    p->put();

    // указателю на Anybody передали адрес w1 (Worker)
    // тип указателя Anybody
    p = &w1;
    // вызов виртуальной функции "по-указателю"
    // вызывается функция класса Worker
    p->put();
    // указателю на Anybody передали адрес w1 (Crow)
    // тип указателя Anybody
    p = &cr;
    // вызов виртуальной функции "по-указателю"
    // вызывается функция класса Crow
    p->put();

    // Преобразование объекта w к классу Anybody
    // и вызов печати
    // вызовется функция класса Anybody
    ((Anybody)w).put();
}
```

```
}

```

```
>./myprog
Anybody - apple: 33
Рабочий: я собираю по 22 яблок
Ворона: я ворую по 36 яблок
Anybody - apple: 49

```

Заметим, что для всех функций печати, у нас одинаковый вызов `p->put()`. Однако все функции работают по-разному. Это явление называется **полиморфизм**

Абстрактные классы

Обычно при наследовании предполагается что у классов будет значительная общая часть. Но часто нужно чтобы совпадал лишь интерфейс классов. А сами реализованные функции имеют никак с друг другом алгоритмически не связаны.

Для объявления такого интерфейса и предоставления возможности обращаться к виртуальным функциям совершенно непохожих объектов существуют **Абстрактные классы**

Классы, в которых есть хотя бы одна нереализованная функция – **абстрактные классы** .

В **абстрактных классах** объявляются имена и интерфейс функций, но их реализация не предполагается. Все функции должны быть реализованы в классах – наследниках. Из-за того что, в абстрактных классах существует хотя бы одна нереализованная функция, эти классы не могут порождать объекты. Но можно использовать указатели на эти классы.

```
#include <iostream>
#include <cstdlib>
#include <fstream>

using namespace std;

// Абстрактный класс
class Anybody{
    int apple;
protected:
    static int basket;
public:
    Anybody();
    Anybody(int);
    int getApple();

    // не виртуальная функция
    void put();
    // виртуальная функция, которая не реализуется в этом классе
    virtual void act() = 0;
};

class Worker:public Anybody{
public:
    Worker(int);
    // будет работать как обычная функция
    void put();
    // обязательно нужно реализовать
    // будет работать как виртуальная
    void act();
};

class Crow:public Anybody{
public:
    Crow();
    // будет работать как обычная функция
    void put();
    // обязательно нужно реализовать
    // будет работать как виртуальная
    void act();
};

int Anybody::basket = 0;

```

```

Anybody::Anybody(){
    apple = rand() %50;
};
Anybody::Anybody(int a){
    apple = abs(a) % 50;
};

int Anybody::getApple(){
    return apple;
};

// просто печатает что в корзине
void Anybody::put(){
    cout<<"basket: "<<basket<<endl;
};

Worker::Worker(int a = 49 ):Anybody(a){};

// функция в наследнике просто так же называется как и у
// родителя.
void Worker::put(){
    cout<<"Рабочий: я собираю по "<<getApple()<<" яблок"<<endl;
};

// реализация виртуальной функции
void Worker::act(){
    basket += getApple();
};

Crow::Crow(){};

// функция в наследнике просто так же называется как и у
// родителя.
void Crow::put(){
    cout<<"Ворона: я ворую по "<<getApple()<<" яблок"<<endl;
};

// реализация виртуальной функции
void Crow::act(){
    basket -= getApple();
    if(basket < 0) basket = 0;
};

int main(){
    // указатель на Anybody
    // объектов быть не может.
    Anybody *p;
    // два рабочих
    Worker w,w1(22);
    // ворона
    Crow cr;
    // передача адреса указателю на w1
    // преобразование к типу Anybody
    p = &w;
    // вызов виртуальной функции
    // работает функция Worker
    p->act();
    // печать. работает функция Worker
    w.put();
    // put() - не виртуальная.
    // работает функция Anybody
    p->put();
    p = &w1;

    p->act();
    w1.put();
    p->put();

    p = &cr;
    p->act();

```

```

    cr.put();
    p->put();
}

```

```

>./myprog
Рабочий: я собираю по 49 яблок
basket: 49
Рабочий: я собираю по 22 яблок
basket: 71
Ворона: я ворую по 33 яблок
basket: 38

```

Можно использовать массив указателей на класс **Anybody**. Тогда **main()** будет выглядеть совсем просто

```

int main(){
// Массив указателей на 3 Anybody
Anybody *p[3];
Worker w,w1(22);
Crow cr;
// первый работает рабочий
p[0] = &w;
// второй прилетает ворона
p[1] = &cr;
// третий работает рабочий
p[2] = &w1 ;

// все в саду 4 часа
for(int h = 0; h<4;h++){
    cout<<endl<<h<<" час:"<<endl;

// обращаемся к каждому Anybody из массива
// по-очереди
    for(int i = 0; i< 3; i++){
// что-то делает
        p[i]->act();
// печать
        p[i]->put();
    }
}
}

```

Задачи

Задача 1

К предыдущему примеру задаче добавить класс **Customer** – наследник от **Worker** . Переопределить для него виртуальную функцию **act()** – забирает яблоки из корзины и кладет сколько-нибудь денег в общий с **Worker** кошелек.

В саду **N** персон. Из файла вводятся числа Первое число **N** – количество персон. Затем вводятся **N** чисел (положительные, отрицательные и ноль) – количество яблок. Если число 0 – это ворона, если число ≥ 0 – это количество яблок, которое собирает рабочий, если < 0 – это количество яблок, которое покупает покупатель.

Написать программу-модель, которая показывает сколько яблок в корзине каждый час.

Все, кто в саду, должны создаваться оператором **new** и их указатели должны запоминаться в массив.

Передача указателя на нестатические методы как параметра.

Слегка усложненная задача Усложним задачу про рабочих, ворон и покупателей. Пусть каждый из них выполняет свои действия в зависимости от времени дня. В своей модели мы будем придерживаться следующего масштаба: 1 секунда – один час.

Рабочие начинают работу в саду в 8.00 и уходят в 17.00, вороны прилетают в 6.00 и улетают в 18.00, покупатели приходят за яблоками с 10.00 и до 17.00. Причем, каждый персонаж выполняет свои действия с собственной периодичностью.

Таким образом, для каждого мы должны иметь по три **будильника**, и будильник должен каким-то образом знать в какой момент какие методы вызывать для объекта.

Рассмотрим пример реализации передачи параметров на метод класса

```
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

class Z;
class A1;
// Объявим Fun как указатель на функцию класса Z возвращающую void и
// не имеющую параметров
// оператор ::* объявляет указатель на функцию класса
typedef void (Z::*Fun)();

// абстрактный класс Z
class Z{
public:
    virtual void f1()=0;
    virtual void f2()=0;
};

// наследник Z
class Z1:public Z{
public:
    void f1();
    void f2();
};

// наследник Z
class Z2:public Z{
public:
    void f1();
    void f2();
};

// Будущий будильник
class A1{
// указатель на объект,
// чьи функции будем вызывать
    Z* pz;
public:
// конструктор
    A1(Z*);
// функция класса A1 с параметром - указателем на функцию класса Z
    void go(Fun);
};

// реализация абстрактных функций
void Z1::f1(){
    cout<<"11AAAAA\n";
};

void Z1::f2(){
    cout<<"12RRRzzz\n";
};

void Z2::f1(){
    cout<<"21AAAAA\n";
};

void Z2::f2(){
    cout<<"22RRR\n";
};

// конструктор будущего будильника
A1::A1(Z* a){
    pz = a;
};
```

```
// реализация функции, вызывающей функцию
// класса Z по указателю на нее
void A1::go(Fun fptr){
// функции мы определили виртуальными,
// значит вызовутся функции соответствующих классов
(pz->*fptr)();
};

int main(){

    Z1 az;
    Z2 ax;
    A1 d1(&az);
    A1 d2(&ax);
    d1.go(&Z::f2);
    d1.go(&Z::f2);
    d2.go(&Z::f1);
    d2.go(&Z::f2);

}
```

Задача 2

Написать интерфейс всех классов для решения "слегка усложненной задачи".

Задача 3

Реализовать все классы для "слегка усложненной задачи" и промоделировать работу сада за 3 дня. Данные на всех получать из файла. Продумать формат файла.

Задача 4

Нужно работать с картинками.

Необходимо иметь класс, который:

1. рисует прямоугольник
2. рисует треугольник
3. рисует окружность
4. рисует линию
5. рисует точку
6. сохраняет картинку в графический файл

У нас есть реализованный раньше класс **Image**

```
#include <wx/wx.h>
#include <wx/image.h>
#include <fstream>
#include <iostream>
#include <string.h>
using namespace std;

// Класс Image - наследник "системного" класса wxImage
// wxImage "умеет" работать с картинками
class Image:public wxImage{
    wxColor pen; // цвет линий
    wxColor fill; // цвет красить
    int w,h; // размер картинки
public:
    // Создать картину из файла (имя - в строке wxString)
    Image(string);
    // создать пустую картину, с фоном back
    Image(int w, int h, wxColor back);
    // деструктор
    ~Image();
    // рисование линии по двум точкам
    void DrawLine(wxPoint one, wxPoint sec);
    // установить цвет линий
    void setPen(wxColor a);
```



```

// установить цвет заливки
void setFill(wxColor a);
// закрасить точку p цветом a
void ColorPoint(wxPoint p, wxColor a);
// закрасить прямоугольник цветом a, внутри которого точка p
void FillRec(wxPoint p, wxColor a);
// сохранить картинку в файл. Имя файла в строке
void saveToFile(string);
};

// Реализация

Image::Image( string file):wxImage(wxString(file.c_str()), wxConvUTF8),wxBITMAP_TYPE_PNG){

    w = GetWidth();
    h = GetHeight();

};

Image::Image(int w, int h, wxColor back):wxImage(w,h){
    wxInitAllImageHandlers();
    wxPoint one(1,1),sec(w-1,h-1);
    wxRect rec(one,sec);
    this->SetRGB(rec, back.Red(),back.Green(),back.Blue());
};

void Image::setPen(wxColor a){
    pen = a;
};
void Image::setFill(wxColor a){
    fill = a;
};
void Image::ColorPoint(wxPoint p, wxColor c){
    wxRect rc(p,p);
    this->SetRGB(rc,c.Red(),c.Green(),c.Blue());
};
void Image::FillRec(wxPoint p, wxColor c){
    wxColor place(GetRed(p.x,p.y),GetGreen(p.x,p.y),GetBlue(p.x,p.y));
    wxColor check(GetRed(p.x,p.y - 1),GetGreen(p.x,p.y - 1),GetBlue(p.x,p.y - 1));
    int yh,yl,xl,xp;
    int x,y;
    // cout << (place != c)<<endl;
    for(y = p.y-1 ; place == check; y--){
        check.Set(GetRed(p.x,y), GetGreen(p.x,y), GetBlue(p.x,y));
    };
    yh = y + 1;
    check.Set(GetRed(p.x,p.y + 1), GetGreen(p.x,p.y + 1), GetBlue(p.x,p.y + 1));
    for(y = p.y + 1 ; place == check; y++){
        check.Set(GetRed(p.x,y), GetGreen(p.x,y), GetBlue(p.x,y));
    };
    yl = y - 1;
    check.Set(GetRed(p.x - 1,p.y ), GetGreen(p.x - 1,p.y), GetBlue(p.x - 1,p.y));
    for(x = p.x - 1 ; place == check; x--){
        check.Set(GetRed(x,p.y), GetGreen(x,p.y), GetBlue(x,p.y));
    };
    xl = x + 1;
    check.Set(GetRed(p.x + 1,p.y), GetGreen(p.x + 1,p.y), GetBlue(p.x + 1,p.y));
    for( x = p.x + 1 ; place == check; x++){
        check.Set( GetRed(x,p.y), GetGreen(x,p.y), GetBlue(x,p.y));
    };
    xp = x - 1;

    wxRect rec(wxPoint(xl,yh), wxPoint(xp,yl));
    this->SetRGB(rec, c.Red(), c.Green(), c.Blue());
};

void Image::DrawLine(wxPoint one, wxPoint sec){

```

```

    int b,f;
    if (one.x < sec.x){
        b = one.x;
        f = sec.x;
    }else{
        f = one.x;
        b = sec.x;
    }
    for(int x = b; x < f ; x++){
        int y=((sec.y - one.y ) * x + (sec.x * one.y - one.x * sec.y )) / (sec.x - one.x );
        // cout<<"x="<<x<<" y="<<y<<endl;
        wxPoint fr(x,y);
        wxRect rec(fr,fr);
        this->SetRGB(rec, pen.Red(), pen.Green(), pen.Blue());
    }

    if (one.y < sec.y){
        b = one.y;
        f = sec.y;
    }else{
        f = one.y;
        b = sec.y;
    }
    for(int y = b; y < f ; y++){
        int x = ((sec.x - one.x ) * y + (sec.y * one.x - one.y * sec.x )) / (sec.y - one.y);
        // cout<<"x="<<x<<" y="<<y<<endl;
        wxPoint fr(x,y);
        wxRect rec(fr,fr);
        this->SetRGB(rec, pen.Red(), pen.Green(), pen.Blue());
    }
};

void Image::saveToFile(string file){
    wxString stl(file.c_str(), wxConvUTF8);
    this->SaveFile(stl,wxBITMAP_TYPE_PNG);
};

Image::~Image(){
    wxImage::Destroy();
};

// Пример использования

int main(){
    // Создать "пустую" желтую картинку размером 200x200
    Image im(200,200,wxColor(255,255,0));
    // имя файла картинки
    string s = "f1.png";
    // установить цвет линий - белый
    im.setPen(wxColor(255,255,255));
    // нарисовать линию от точки (20,20) до точки (60,20)
    im.DrawLine(wxPoint(20,20),wxPoint(60,20));
    im.DrawLine(wxPoint(20,20),wxPoint(20,60));
    im.DrawLine(wxPoint(20,60),wxPoint(60,60));
    im.DrawLine(wxPoint(60,20),wxPoint(60,60));
    // Закрасить прямоугольник, в котором точка (30,30)
    im.FillRect(wxPoint(30,30), wxColor(255,0,0));
    // Закрасить одну точку внутри прямоугольника
    im.ColorPoint(wxPoint(40,40),wxColor(0,255,0));
    // нарисовать "косую" линию
    im.DrawLine(wxPoint(20,20),wxPoint(30,60));
    // сохранить картинку в файл
    im.saveToFile(s);
}

```

Для указаний цветов здесь используются объекты класса **wxColor**. Для указания точек на картинке используются объекты класса **wxPoint**

Написать класс – наследник **Image**. Нужно добавить функции:

1. рисования прямоугольника
2. рисования треугольника
3. рисования наклонного креста

4. рисования окружности
5. ** если на картинке только прямоугольники, подсчитывать сколько прямоугольников

Для компиляции такой программы нужно использовать файл ***comp***

Первая команда (ОДИН РАЗ!!)

```
chmod u+x comp
```

Теперь каждый раз, когда нужно компилировать. **сpp** в названии файла с текстом программы не указывать!!

```
./comp muprog
```

Задача 3

Для класса **Image** создать абстрактный класс **Fig**. Написать и запрограммировать классы-наследники:

1. **Rec** - прямоугольник
2. **Line** - линия
3. **Tri** - треугольник
4. **Circle** - круг

Создать несколько объектов, поместить указатели на них (**Fig**) в массив и нарисовать их все на черной картинке. Пример для класса **Line**

```
//.....

// Абстрактный класс
class Fig{
    wxPoint center;
protected:
    // объекту нужен указатель на картинку (на чем будет рисовать)
    Image *im;
public:
    Fig(Image*);
    // Виртуальные функции
    virtual void Draw() = 0;
    virtual void moveTo(wxPoint)=0;
    virtual void Fill(wxColor)=0
};

class Line:public Fig{
    wxPoint one,sec; // концы отрезка
public:
    // Конструктор с указателем на картинку
    Line(Image*);
    // Установка концов отрезка
    void setPoints(wxPoint a, wxPoint b);
    // Цвет линии
    void setColor(wxColor cl);
    // Переопределение виртуальной функции Draw()
    void Draw();
    //переопределить самостоятельно
    void moveTo(wxPoint);
    void Fill(wxColor);
};

// Получаем указатель на картинку
Fig::Fig(Image* m){
    im = m;
};

// Конструктор. Передача адреса картинки
Line::Line(Image* a):Fig(a){};

// Установка точек
void Line::setPoints(wxPoint a, wxPoint b){
    one = a;
    sec = b;
};
```

```
// Установка цвета
void Line::setColor(wxColor cl){
    im->setPen(cl);
};

// переопределение виртуальной функции
// рисование
void Line::Draw(){
    im->DrawLine(one,sec);
};

// Пример использования
int main(){
    // Создать "пустую" желтую картинку размером 200x200
    Image im(200,200,wxColor(255,255,0));
    // Создаем Line. передаем адрес картинки
    Line f1(&im);
    // установка точек на картинке
    f1.setPoints(wxPoint(10,100),wxPoint(190,100));
    // установка цвета
    f1.setColor(wxColor(255,0,255));
    // рисование
    f1.Draw();
    string s="vpic.png";
    im.saveToFile(s);
}
```

-- [TatyanaOvsyannikova2011](#) - 08 Apr 2016

Attachment	Action	Size	Date	Who	Comment
 comp	manage	0.1 K	31 Mar 2021 - 10:10	TatyanaOvsyannikova2011	

(с) Материалы раздела "Язык Си" публикуются под лицензией [GNU Free Documentation License](#).