





Раздел «Язык Си» . OOP-Her_begin :

- [Наследование](#)
 - [Видео](#)
 -  [Задача 1](#)
 -  [Задача 2](#)
 - [Область protected](#)
 -  [Задача 3](#)
 - [Вызов функций для работы с закрытой областью класса-родителя](#)
 -  [Задача 4](#)

Наследование

Видео

- [Видео о классах и классовых отношениях \(коротко\).](#)

Задача про работников и ворон

В саду работают работники и собирают яблоки. Каждый свое количество яблок в час. Кроме того эти яблоки воруют вороны. Тоже каждая может украсть свое количество в час (разом).

Нужно написать классы **Worker** и **Crow** для моделирования количества яблок в корзине с учетом похищения.

Пример интерфейсов классов:

```
// Ворона
class Crow{
    int apple;// украденные яблоки за 1 час
    int *basket;// указатель на корзину с яблоками
public:
    // конструктор. Вычисляется случайное количество яблок
    Crow();
    // Деструктор. Указатель на корзину должен стать 0
    ~Crow();
    // получит указатель на корзину
    void getBasket(int *pbasket);
    // возвращает количество украденных яблок
    int steal();
};
//Все функции класса Crow нужно написать и отладить.

// Работник
class Worker{
    int apple; // количество яблок, которые он собирает в час
    int *basket;// указатель на корзину с яблоками
public:
    // Конструктор по умолчанию. Мы не можем указывать
    // сколько яблок он соберет. Будет случайно
    Worker();

    // Деструктор. Указатель на корзину должен стать 0
    ~Worker();
    // возвращает количество яблок, которые собрал работник
    int job();
    // получит указатель на корзину
    void getBasket(int *pbasket);
};
```

Из этих примеров видно, что функциональность обоих классов сильно совпадает. К тому же есть корзина, указатель на которую нужно передавать как параметр. А это не очень удобно.

Можно определить некий класс **Person**, у которого есть одинаковый для всех атрибут **apple** и статический объект – **basket**, а также общие для работника и ворон функции.

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

Поиск

Раздел «Язык Си»

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

Инструменты:

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

Разделы

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

[Logon>>](#)

```

using namespace std;
// Класс, в котором есть общие для работников и ворон
// атрибуты и функции

class Person{
    int apple; // яблоки "знают" и работники и вороны
public:
    Person(); // конструктор. Количество яблок - случайно
    Person(int app); // инициализирующий конструктор.
    int getApple(); // возвращает количество яблок
    void print(); // печать

    // корзина - статический объект
    // в области public, значит ее могут использовать все
    static int basket;
};
// Сначала в корзине 0
int Person::basket=0;

// Реализация

Person::Person(){
    apple=rand() % 100;
    cout<<"Появился кто-то. Хочет "<<apple<<" яблок в час"<<endl;
};

Person::Person( int app){
    // Проверка. Сокращенная запись
    // Если app < 0, то яблок - 0, если >=0, то не больше 100
    apple=(app < 0)? 0 : app % 100;
    cout<<"Появился кто-то. Хочет "<<apple<<" яблок в час"<<endl;
};
// получить яблоки
int Person::getApple(){
    return apple;
};
// печать
void Person::print(){
    cout<<apple<<" яблок в час"<<endl;
    // basket - статическая переменная - общая для всех
    // сразу печататем сколько в ней яблок
    cout<<" в корзине: "<<basket<<endl;
};

```

Теперь можно написать новый класс **Worker** – наследник класса **Person**. У нового класса **Worker** уже сразу будут функции: **print()**, **getApple()** и конструкторы от класса **Person**.

Объекты класса наследника уже "умеют" то, что предоставил им класс-родитель.

```

// Объявляем класс Worker как наследник
class Worker:public Person{
public:
    // Новый конструктор по-умолчанию
    Worker();
    // Новый инициализирующий кнструктор
    Worker(int app);
    // новая функция, ее не было
    void act();
};

// Реализация

Worker::Worker(){
    // в новом конструкторе можно добавить действий
    cout<<"Я работник, я собираю ";
    // этот print() уже есть в классе - родителе Person
    // его можно просто вызвать.
    print() ;
};

// Новый инициализирующий конструктор
// Когда создается объект класса-наследника, сначала всегда создается
// объект класса - родителя.
// apple - в закрытой области класса-родителя, значит
// не доступен Worker/

```

```
// Чтобы передать параметры объекту класса Person,
// нужно ЯВНО вызвать инициализирующий конструктор класса Person

Worker::Worker(int app):Person(app){
    cout<<"Я работник, мне сказали собирать ";
    print() ;
};
// basket - в открытой области. Можно спокойно им пользоваться
void Worker::act(){
    basket += getApple(); // кладет яблоки в корзину.
};

// Пример использования Worker
int main(){
    Person p1, p2(220), p3(-3);
    Worker a1, a2(30);
    a1.act();// работает
    a2.act();
    a2.act();// работает
    a1.print();

    // этот print() будет вызван для объекта класса Person,
    // а не для Worker
    // МОЖНО ПРЕОБРАЗОВЫВАТЬ ОБЪЕКТ к КЛАССУ-РОДИТЕЛЮ
    ((Person)a2).print();

    return 0;
}
```

Вороны тоже пользуются корзиной и "знают" про яблоки. Можно написать класс **Crow** как наследник класса **Person**.

```
class Crow:public Person{
// про яблоки все уже написано в классе Person
public:
    Crow(); // конструктор
    Crow(int app); // инициализирующий конструктор
    void steal(); // вороны воруют яблоки из корзины
};
```



🔧 Задача 1

Реализовать и проверить класс **Crow**

🔧 Задача 2

Работники и вороны в саду 8 часов. Написать программу-модель, чтобы посмотреть сколько яблок в корзине каждый час. Использовать классы-наследники **Worker** и **Crow**

Область **protected**

Яблоки, которые собрали работники покупают покупатели. НИКТО кроме работников, ворон и покупателей не должен иметь доступ к корзине.

Для того, чтобы к атрибуту имели доступ только функции классов-наследников есть область **protected**.

"Спрячем" **basket** в область **protected**

```
class Person{
    int apple;
public:
```

```

    Person();
    Person(int app);
    int getApple();
    void print();
protected:
// теперь изменять basket могут только
// объекты классов-наследников
    static int basket;
}

class Worker:public Person{
public:
    Worker();
    Worker(int app);
    void act();
protected:
// Для денег за яблоки заведем кошелек
    static int cache;
};

int Worker::cache = 0;

// класс Покупатель. Он берет яблоки и оставляет в деньги в cache
// имеет доступ к cache
class Customer:public Worker{
// про яблоки уже все известно
    int money;
public:
    Customer();// конструктор только по-умолчанию, деньги случайно, но <= 20
// покупатель может и собирать яблоки - функция act() у него тоже есть

// берет яблоки и оставляет деньги в cache
    void buy();

    void print();
};

// класс Crow имеет доступ к basket, но не имеет к cache
class Crow:public Person{
public:
    Crow();
    Crow(int app);
    void act();
};

```

Задача 3

Реализовать функции класса **Customer**. Если яблок в корзине нет, покупатель яблоки не берет.

Вызов функций для работы с закрытой областью класса-родителя

```

#include <iostream>
#include <cstdlib>

using namespace std;
//Родительский класс
class A{
    int x;
public:
    A(int);
    A operator+(A);
// вывод в поток: на консоль в файл и т.д.
    ostream& put(ostream&);
};

// Класс-наследник
class B:public A{
    int y; // дополнительный атрибут
public:
// В конструкторе 2 параметра: для x и y
    B(int, int);
// При сложении складываются атрибуты x и y
    B operator+(B);
};

```

```

// вывод в поток
ostream& put(ostream&);
};

A::A(int a){
    x = a;
    // сообщит о работе конструктора
    cout<<"A!!\n";
};

A A::operator+(A a){
    A tmp(0);
    tmp.x = x + a.x;
    return tmp;
};

// Поток называется s. Для него определен оператор вывода (<<)
ostream& A::put(ostream& s){
    s<<"x="<<x<<" ";
};

// Свободная функция - оператор вывода
// имеет два параметра: поток и объект
ostream& operator<<(ostream& s, A a){
    // вызов функции вывода объекта A
    return a.put(s);
};

// Инициализирующий конструктор B.
// Явный вызов инициализирующего конструктора A
// для параметра x
B::B(int a, int b):A(a){
    y = b;
    cout<<"B!!\n";
};

// Сложение
B B::operator+(B b){
    B tmp(0,0); // временный объект
    // Указатели на объекты класса A
    // для вызова операций класса A
    // и заполнения области, соответствующей классу A
    // в объекте класса B
    A *ptmp,*pb;
    // адреса объектов класса B переданы
    // указателям класса A
    ptmp = &tmp;
    pb = &b;
    // Разыменовывание *ptmp - объект класса A
    // *pb - объект класса A
    // *((A*)this) - преобразование текущего объекта к классу A
    // Итого, работает + для класса A
    (*ptmp) = (*pb) + *((A*)this);
    tmp.y = y + b.y;
    return tmp;
};

// Вывод в поток
ostream& B::put(ostream& s){
    // преобразуем текущий объект к классу A
    A t = *((A*)this);
    // работает переопределенный для класса A оператор вывода
    s<<t;
    s<<"y="<<y<<endl;
    return s;
};

// переопределение оператора вывода для класса B
// свободная функция
ostream& operator<<(ostream& s, B b){
    return b.put(s);
};

int main(){
    A a(4),b(5),res(0);
    cout<<a;
    cout<<b;
    res = a + b;
    cout<<res;
}

```

```
B ab(2,5), bb(7,8), resb(0,0);  
cout<<ab;  
resb = ab + bb;  
cout<<resb;  
}
```

Задача 4


Для класса **image** написать класс-наследник, в котором реализованы дополнительные функции:

1. рисование прямоугольника
2. рисование треугольника
3. закраска прямоугольника
4. закраска треугольника
5. рисование многоугольника
6. рисование круга <\o1>

Конструктор класса-наследника может выглядеть так:

```
MyImage::MyImage(string file):Image(wxString(file.c_str(), wxConvUTF8), wxBITMAP_TYPE_PNG){};
```

-- [TatyanaOvsyannikova2011](#) - 23 Mar 2016

Attachment 	Action	Size	Date	Who	Comment
 nasl1.png	manage	37.1 K	23 Mar 2016 - 14:52	TatyanaOvsyannikova2011	

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