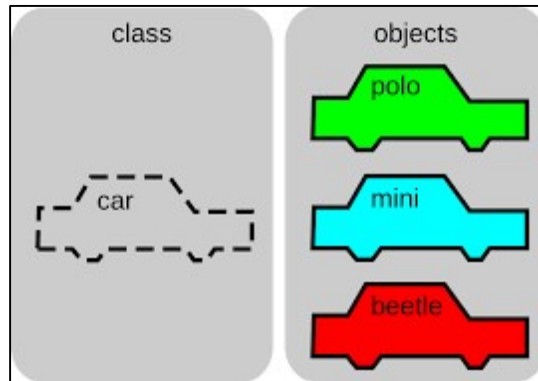


# ЛАБОРАТОРНАЯ РАБОТА №1 ПО ПРЕДМЕТУ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

## ТЕМА: КЛАССЫ И ОБЪЕКТЫ В ЯЗЫКЕ C++

**ЦЕЛЬ РАБОТЫ:** изучить структуру класса, атрибуты доступа к компонентам класса; рассмотреть принцип работы конструкторов (с параметрами, без параметров, с параметрами по умолчанию, конструктора копирования), деструкторов при работе с объектом, статические и константные данные и методы класса.



### ИСПОЛЬЗОВАНИЕ СТРУКТУР И КЛАССОВ

Объектно-ориентированное программирование является стандартом в технологии современного программирования и в подавляющее большинство новейших средств разработки заложен именно **объектный подход**. Язык C++ обладает всеми необходимыми возможностями современного объектно-ориентированного языка программирования.

Основная идея ввода в использование языка C++ классов заключается в том, чтобы предоставить программисту средства для создания новых собственных типов данных, которые могут использоваться так же, как и встроенные типы.

**Класс – это абстракция, это тип, определяемый пользователем.** Объявление класса начинается с ключевого слова *class*. Объявление класса синтаксически подобно объявлению структуры. Например, можно создать структуру с именем *Point*, которая содержит координаты *x* и *y*, описывающие точки на экране дисплея:

```
struct Point{  
    int x,y;  
};
```

Пусть необходимы две функции, которые позволяют задать точку (функция *setPixel()*) и получить координаты точки (функция *getPixel()*):

```
void setPixel(int, int);
void getPixel(int*, int*);
```

В этом примере данные и функции, работающие с этими данными, отделены друг от друга. Связь между ними можно установить, если создать структуру в следующем виде:

```
struct Point{
    int x,y;
    void setPixel(int, int);
    void getPixel(int*, int*);
};
```

Данные  $x$  и  $y$ , объявленные в приведенной структуре, называются компонентами-данными, а функции – компонентами-функциями или методами структуры. Экземпляр структуры объявляется следующим образом:

```
имя_структуры имя_экземпляра;
```

Теперь мы можем обратиться к данным экземпляра структуры и вызвать методы. Для этой цели можно использовать операции доступа точка «.» и стрелка «->».

```
//Пример №1. Пример использования данных-компонентов и
компонентов-функций
#include<iostream>
#include <iomanip>
using namespace std;
struct Point { // объявление структуры
    int x, y;
    void setPixel(int, int); //функция для установления значения
переменных x и y
    void getPixel(int*, int*); //функция для получения значения
x и y
};
int main() {
    setlocale(LC_ALL, "Russian");
    int x, y;
    Point point, * pointer;
    pointer = &point;
    point.setPixel(50, 100);
    pointer->getPixel(&x, &y);
    cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

```

void Point::setPixel(int a, int b) {
    x = a;
    y = b;
}
void Point::getPixel(int* a, int* b) {
    *a = x;
    *b = y;
}

```

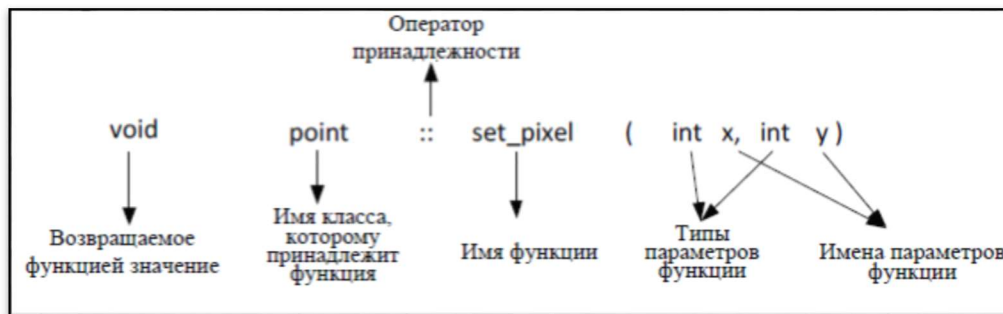
Результаты работы программы:

```
x = 50, y = 100
```

Поскольку различные структуры могут иметь функции с одинаковыми именами, то при описании функции необходимо указывать, для какой структуры она описывается:

```
void Point::setPixel(int a, int b){ тело функции }
```

Синтаксис описания функции, принадлежащей структуре, имеет следующий вид:



Оператор принадлежности «::» иначе называется оператором разрешения области видимости (оператором разрешения контекста). **Это оператор самого высокого приоритета.**

В языке C++ структура, классы и структуры можно назвать близкими родственниками. **Основное отличие между C++-структурой и C++-классом состоит в том, что по умолчанию элементы класса являются закрытыми, а элементы структуры — открытыми.** В остальном же структуры и классы имеют одинаковое назначение. И классы, и структуры могут иметь сочетание открытых и закрытых элементов, могут использовать наследование и могут иметь функции-элементы.

C++-программисты тип *class* используют главным образом для определения формы объекта, который содержит методы-элементы и данные, а тип *struct* — для создания объектов, которые содержат только элементы данных. Иногда для описания структуры, которая не содержит

методы-элементы, используется аббревиатура *POD* (*Plain Old Data*, простые старые данные).

Для создания класса необходимо указать вместо ключевого слова *struct* ключевое слово *class*, например:

```
class Point{
    int x,y;
    void setPixel(int, int);
    void getPixel(int *, int *);
};
```

## ОБЪЕДИНЕНИЯ

**Объединение (*union*)** – тип данных, переменная которого может хранить (в разное время) объекты различного типа и размера. В результате появляется возможность работы в одной и той же области памяти с данными различного вида. Для описания объединения используется ключевое слово *union*, а соответствующий синтаксис аналогичен структурам и классам. Пусть задано объединение:

```
class MouseEvent {};\n\nclass KeyboardEvent {};\n\nunion Event{\n    MouseEvent mouse;\n    KeyboardEvent keyboard;\n    int eventCode;\n};
```

Размер переменной объединения *Event* будет равен размеру самого большого из трех приведенных типов. В один и тот же момент времени переменная объединения *Event* может иметь значение только одной из переменных *mouse*, *keyboard* или *eventCode*.

В языке C++ класс, определяемый посредством ключевых слов *struct*, *class*, *union*, включает в себя методы и данные, создавая новый тип объектов. Компоненты класса имеют ограничения на доступ. Эти ограничения определяются ключевыми словами *private*, *protected*, *public*. **Для ключевого слова *class* по умолчанию все компоненты будут *private*.** Это означает, что содержимое класса будет недоступно для использования вне компонентов класса. Ограничения доступа для некоторого компонента класса можно изменить, записав перед ним модификатор доступа – ключевое слово *public*, *private*, *protected*. Таким образом, упрощенную форму описания класса можно записать в виде:

```
class имя_класса{\n    данные и методы с атрибутом private (по умолчанию)
```

```
protected:
    данные и методы с атрибутом protected
public:
    данные и методы с атрибутом public
} объекты этого класса через запятую;
```

**Обычно данные класса имеют атрибут доступа *private* или *protected*, а методы – *public*. Значения атрибутов доступа:**

– ***private*** – элемент класса с атрибутом *private* может использоваться только методами собственного класса и функциями-«друзьями» этого же класса; по умолчанию все элементы класса, объявленного с ключевым словом *class*, имеют атрибут доступа *private*;

– ***protected*** – тот же доступ, что и *private*, но дополнительно элемент класса может использоваться методами и функциями-«друзьями» производного класса, для которого данный класс является базовым;

– ***public*** – элемент класса может использоваться любой функцией, т. е. защита на доступ к элементу снимается.

**Элементы структуры (*struct*) и объединения (*union*) по умолчанию имеют доступ *public*. Для ключевого слова *struct* и *union* атрибут доступа можно явно переопределить на *private* или *protected*. Но типы *struct* и *union* не являются частью парадигмы ООП.**

Класс или структура может содержать любое количество секций с необходимыми атрибутами. Секция начинается с атрибута доступа и двоеточия после него. Секция заканчивается в конце описания структуры (класса) или началом другой секции.

```
//Пример №2. Пример использования атрибутов доступа к элементам
//класса
#include <iostream>
#include <iomanip>
#include <windows.h>
using namespace std;
class String {
    char str[25]; //поле с атрибутом доступа private
public:
    void setString(const char*); //функция инициализации строки
    void displayString(); //функция отображения строки str на
    char* returnString(); //функция возврата строки
};
void String::setString(const char* s) {
    strcpy_s(str, s); //копирование s в str
}
void String::displayString() { cout << str << endl; }
char* String::returnString() { return str; }
```

```
int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    String string;//объявление объекта типа String
    string.setString("Город Минск");
    string.displayString();
    cout << string.returnString() << endl;
    return 0;
}
```

Результаты работы программы:

```
Город Минск
Город Минск
```

## КОНСТРУКТОР

Использование функций для установки начальных значений полей объекта часто приводит к ошибкам. В связи с этим введена специальная функция, позволяющая инициализировать объект в процессе его декларирования (определения). Эта функция называется **конструктором**. Конструктор является элементом класса и имеет то же имя, что и класс.

```
class String{
    char str[25];
public:
    String(char *s){ //конструктор
        strcpy(str,s);
    }
};
```

**Конструктор может иметь и не иметь аргументы, но он никогда не возвращает значение (даже типа *void*).** Класс может иметь несколько конструкторов, что позволяет использовать несколько различных способов инициализации объектов. Конструктор является функцией, а значит, он может быть перегружен. Чтобы перегрузить конструктор класса, достаточно объявить его во всех нужных форматах и определить действие, связанное с каждой формой конструктора.

Конструктор вызывается при создании объекта. Это означает, что он вызывается при выполнении инструкции объявления объекта. Конструкторы глобальных объектов вызываются в самом начале выполнения программы, еще до обращения к функции *main()*. Что касается локальных объектов, то их конструкторы вызываются каждый раз, когда встречается объявление локального объекта.

```
//Пример №3. Использование конструктора
#include<iostream>
#include <windows.h>
using namespace std;
```

```

class Game {
    int number;
    char* name;
public:
    Game() { //первый конструктор
        name = new char[60];
        strcpy_s(name, 59, "Слова из слова");
        number = 0;
    }
    Game(const char* n) { //второй конструктор
        name = new char[60];
        strcpy_s(name, 59, n);
        number = 50;
    }
    Game(const char* n, int x) { //третий конструктор
        name = new char[60];
        strcpy_s(name, 59, n);
        number = x;
    }
    Game(int* y) { //четвертый конструктор
        name = new char[60];
        strcpy_s(name, 59, "Wood blocks");
        number = *y;
    }
    void print();
};

void Game::print() { cout << "Номер игры \"" << name << "\" равен =
" << number << endl; }

int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int a = 10, *b;
    b = &a;
    Game game; //вызов конструктора без параметров
    Game game1( "Скайдом"); //вызов конструктора с параметром
    типа "const char*"
    Game game2("Bubble hit", 100); //вызов конструктора с двумя
    параметрами
    Game game3(b); //вызов конструктора с параметром типа "int*"
    game.print();
    game1.print();
    game2.print();
    game3.print();
    return 0;
}

```

**Результаты выполнения программы представляются в следующем виде:**

```
Номер игры "Слова из слова" равен = 0
Номер игры "Скайдом" равен = 50
Номер игры "Bubble hit" равен = 100
Номер игры "Wood blocks" равен = 10
```

Конструктор может содержать значения аргументов по умолчанию. Задание в конструкторе аргументов по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если при вызове конструктора не указаны никакие значения. **Конструктор, у которого все аргументы заданы по умолчанию, называется конструктором по умолчанию, т. е. конструктором, который можно вызывать без указания каких-либо аргументов. Для каждого класса может существовать только один конструктор по умолчанию.**

```
//Пример №4. Пример использования конструктора по умолчанию
#include <windows.h>
#include<iostream>
using namespace std;
class Time {
public:
    Time(int = 0, int = 0, int = 0); //конструктор с параметрами
    по умолчанию
    void setTime(int, int, int); //конструктор с параметрами
    void printStandart(); //печать времени в стандартном формате
private:
    int hour; //0-23
    int minute; //0-59
    int second; //0-59
};
//Конструктор Time инициализирует элементы класса полученными
аргументами
Time::Time(int hr, int min, int sec) {
    setTime(hr, min, sec);
}
//Установка нового времени. Выполнение проверки корректности
значений данных. Установка неправильных значений на 0
void Time::setTime(int h, int m, int s) {
    hour = (h >= 0 && h < 24) ? h : 0;
    minute = (m >= 0 && m < 60) ? m : 0;
    second = (s >= 0 && s < 60) ? s : 0;
}
// Печать времени в стандартном формате
void Time::printStandart() {
    cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
    << ":" << (minute < 10 ? "0" : "") << minute
    << ":" << (second < 10 ? "0" : "") << second
    << (hour < 12 ? "AM" : "PM");
}
```



```

int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    Time t1, //все аргументы определены по умолчанию
        t2(2), //минуты и секунды определены по умолчанию
        t3(21, 34), //секунды определены по умолчанию
        t4(12, 25, 42), //все значения указаны
        t5(27, 74, 99); //указаны все неправильные значения
    cout << "Все аргументы по умолчанию ";
    t1.printStandart();
    cout << "\nЧасы заданы; минуты и секунды по умолчанию ";
    t2.printStandart();
    cout << "\nЧасы и минуты заданы; секунды по умолчанию ";
    t3.printStandart();
    cout << "\nЧасы, минуты и секунды заданы: ";
    t4.printStandart();
    cout << "\nВсе значения заданы неверно: ";
    t5.printStandart(); cout << endl;
    return 0;
}

```

Результаты работы программы:

```

Все аргументы по умолчанию 12:00:00AM
Часы заданы; минуты и секунды по умолчанию 2:00:00AM
Часы и минуты заданы; секунды по умолчанию 9:34:00PM
Часы, минуты и секунды заданы: 12:25:42PM
Все значения заданы неверно: 12:00:00AM

```

Конструктор имеет следующие отличительные особенности:

- конструктор всегда вызывается при создании нового объекта, т.е. когда под объект отводится память и когда объект инициализируется;
- конструктор может определяться разработчиком или создаваться по умолчанию;
- конструктор не может быть вызван как обычный метод у объекта. Он вызывается явно компилятором при создании объекта и неявно при выполнении оператора *new* при выделении памяти под объект;
- конструктор всегда имеет то же имя, что и класс, в котором он определен;
- конструктор никогда не должен возвращать значения;
- конструктор не наследуется.

Чтобы передать аргумент конструктору, необходимо связать этот аргумент с объектом при объявлении объекта. C++ поддерживает два способа реализации такого связывания. Вот как выглядит первый способ.

```

Time time = Time (101); //конструктор с одним параметром

```

В этом объявлении создается объект с именем *time*, которой передается значение 101. Но эта форма используется редко, поскольку второй способ имеет более короткую запись и удобнее для использования. Во втором способе аргумент должен следовать за именем объекта и заключаться в круглые скобки. Например, следующая инструкция эквивалентна предыдущему объявлению,

```
Time time (101); //конструктор с одним параметром
```

Это самый распространенный способ объявления параметризованных объектов. Общий формат передачи аргументов конструкторам.

```
тип_класса имя_переменной(список_аргументов);
```

Здесь элемент *список\_аргументов* представляет собой список разделенных запятыми аргументов, передаваемых конструктору. В общем случае, если конструктор принимает только один аргумент, для инициализации объекта можно использовать либо вариант *ob(x)*, либо вариант *ob=x*. При создании конструктора с одним аргументом неявно создается преобразование из типа этого аргумента в тип этого класса.

### **КОПИРУЮЩИЙ КОНСТРУКТОР (КОНСТРУКТОР КОПИИ)**

Конструктор копии вызывается в случае, когда один объект инициализирует другой. Рассмотрим следующую программу, демонстрирующую использование копирующего конструктора.

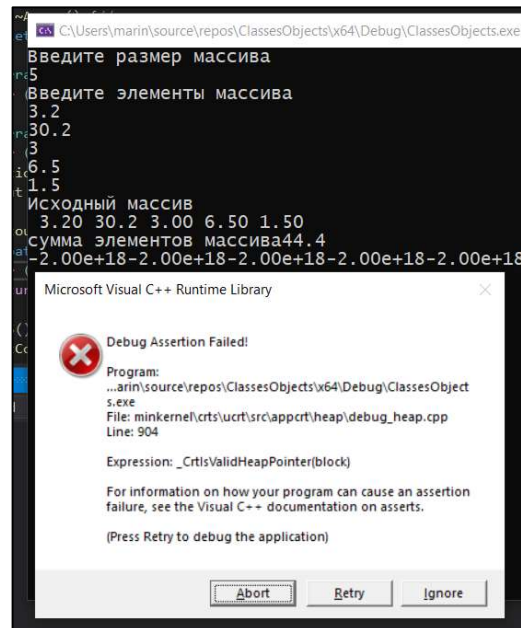
```
//Пример №5. Использование конструктора копий
#include <windows.h>
#include<iostream>
#include<iomanip>
using namespace std;
class Array {
public:
    float* array;//указатель на массив
    int cardinality;//количество элементов массива
    Array(int size = 10);//конструктор с одним параметром
    ~Array();//деструктор
    void input();//функция ввода элементов массива
    void display();//функция вывода элементов массива
};
float countSum(Array obj);//функция вычисления суммы элементов массива
Array::Array(int size) { //реализация конструктора
    cardinality = size;
    array = new float[cardinality];
}
Array::~~Array() { //реализация деструктора
```

```

        delete[] array;
    }
    void Array::input()    { //реализация метода ввода значений
элементов массива
        for (int i = 0; i < cardinality; i++) cin >> array[i];
    }
    void Array::display() { //реализация метода вывода на экран
элементов массива
        for (int i = 0; i < cardinality; i++) cout << setw(5)<<
setprecision(3)
        << setiosflags(ios::showpoint) << array[i];
        cout << endl;
    }
    float countSum(Array obj) { //функция вычисления суммы элементов
массива
        float sum = 0;
        for (int i = 0; i < obj.cardinality; i++)sum +=
obj.array[i];
        return sum;
    }
    int main() {
        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
        float sum;
        int size;
        cout << "Введите размер массива" << endl;
        cin >> size;
        Array ob(size); //объявление объекта типа Array
        cout << "Введите элементы массива" << endl;
        ob.input(); //вызов метода input()
        cout << "Исходный массив" << endl;
        ob.display(); //вызов метода display()
        sum = countSum(ob); //вызов метода countSum()
        cout << "сумма элементов массива" << setw(4) << sum<< endl;
        ob.display(); //при вызове метода display() возникает ошибка
        return 0;
    }

```

**Результаты работы программы:**



В функцию *countSum()* передается значение типа *Array*. Даже если вызванный метод ничего не будет выполнять, произойдет ошибка, связанная с динамическим выделением и освобождением памяти. Параметр метода *countSum()* является локальным в теле этого метода. Любой локальный объект конструируется тогда, когда встречается его объявление, и разрушается, когда блок, в котором он описан, прекращает существование. После завершения метода объект прекращает существовать в результате вызова деструктора объекта *ob*.

В функции *main()* выполняются следующие действия:

- создание *Array ob(size)*; конструирует новый объект *ob*. Конструктор объекта *ob* выделяет (динамически) память под массив *array* с помощью оператора *new*;
- вызывается функция *countSum(ob)*;
- значение объекта *ob* копируется из функции *main* в стек метода *countSum()*;
- копия объекта *ob* содержит указатель на ту же динамическую память (указатель на динамическую память в объекте-оригинале и объекте-копии имеет одинаковые значения);
- функция *countSum()* завершается;
- вызывается деструктор для копии объекта *ob*, который разрушает динамически выделенную память под массив *array*;
- указатель в оригинале объекта адресуется несуществующую удаленную память.

Если в приведенном примере изменить сигнатуру метода *countSum(Array ob)* на *countSum(Array &ob)*, то ошибка будет устранена. При необходимости можно оставить и исходное объявление функции. В этом случае надо устранить ошибку в самом классе *Array*. Когда объект *ob*

копируется из функции *main()* в метод *countSum()*, должен вызываться конструктор копирования. Общий вид конструктора копирования имеет следующий вид:

```
имя_класса (const имя_класса &obj) { // тело конструктора }
```

Элемент *obj* означает ссылку на объект, которая используется для инициализации.

Так как в рассмотренном классе такого конструктора нет, то вызывается конструктор, заданный по умолчанию. Этот конструктор строит точную копию всех данных объекта *ob*, что и приводит к ошибке. Если в классе *Array* задать явно конструктор копирования, например,

```
Array(const Array& ob) {  
    cardinality = ob.cardinality;  
    array = new float[cardinality];  
    for (int i = 0; i < cardinality; i++) array[i] =  
ob.array[i];  
}
```

то ошибка будет устранена. Таким образом, если в конструкторе некоторого класса осуществляется динамическое выделение памяти, то такой класс **должен** иметь соответствующий конструктор копирования, а также деструктор (для корректного освобождения памяти).

Такие языки, как *Java* и *C#*, не имеют конструкторов копии, поскольку ни в одном из них не создаются побитовые копии объектов. Дело в том, что *Java* и *C#* самостоятельно динамически выделяют память для всех объектов, а программист оперирует этими объектами исключительно через ссылки. Поэтому при передаче объектов в качестве параметров функции или при возврате их из функций в копиях объектов нет никакой необходимости.

Тот факт, что ни *Java*, ни *C#* не нуждаются в конструкторах копии, делает эти языки в этой части проще, но работа с объектами исключительно посредством ссылок (а не напрямую, как в *C++*) налагает ограничения на тип операций, которые может выполнять программист. Более того, такое использование объектных ссылок в *Java* и *C#* не позволяет точно определить, когда объект будет разрушен. В *C++* же объект всегда разрушается при выходе из области видимости. Язык *C++* предоставляет программисту полный контроль над ситуациями, складывающимися в программе, поэтому он несколько сложнее, чем *Java* и *C#*.

## ДЕСТРУКТОР

Противоположные действия, по отношению к действиям конструктора, выполняет метод-деструктор или метод-разрушитель, который уничтожает объект. Деструктор может вызываться явно или неявно. Неявный вызов

деструктора связан с прекращением существования объекта из-за завершения его области видимости. Явное уничтожение объекта требует явного вызова деструктора, как и любого другого метода класса. Деструктор имеет то же имя, что и класс, но перед именем деструктора записывается знак тильда «~» (лат. *titulus* — подпись, надпись) — название типографских знаков в виде волнистой черты.

**Деструктор не может иметь аргументы, возвращать значение и наследоваться.** Во многих случаях при разрушении объекту необходимо выполнить некоторое действие или даже некоторую последовательность действий. Локальные объекты создаются при входе в блок, в котором они определены, и разрушаются при выходе из него. Глобальные объекты разрушаются при завершении программы.

```
//Пример №6. Пример использования деструктора
#include <windows.h>
#include<iostream>
using namespace std;
class DandyGame {
    int number;
    char* name;
public:
    DandyGame(const char* n); //объявление конструктора
    ~DandyGame(); //объявление деструктора
    void showName(void);
};

DandyGame::DandyGame(const char* n) { //второй конструктор
    name = new char[60];
    strcpy_s(name, 59, n);
    number = 50;
    cout << "Работает конструктор" << endl;
}

void DandyGame::showName(void) { //определение Методы
    cout << "имя игры: \"" << name << "\"<< endl;
}

DandyGame::~~DandyGame() { //определение деструктора
    delete name;
    cout << "Работает деструктор" << endl;
}

int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    DandyGame game1("Super Mario Bros"); //инициализация объекта
game1
    DandyGame game2("La formica"); //инициализация объекта game2
```

```

    game1.showName(); //вызов метода showName() класса Game для
объекта game1
    game2.showName(); //вызов метода showName() класса Game для
объекта game2
    return 0;
}

```

Результат работы программы:

```

Работает конструктор
Работает конструктор
имя игры: "Super Mario Bros"
имя игры: "La formica"
Работает деструктор
Работает деструктор

```

**Подобно конструкторам деструкторы не возвращают значений, следовательно, в их объявлениях отсутствует тип возвращаемого значения. В отличие от конструкторов, деструкторы не могут иметь параметров.** Не существует средств передачи аргументов объекту, который разрушается. Если же у вас возникнет ситуация, когда при вызове деструктора объекту необходимо получить доступ к некоторым данным, определяемым только во время выполнения программы, создайте для этой цели специальную переменную. Затем непосредственно перед разрушением объекта установите эту переменную равной нужному значению.

//Пример №7. Использование конструкторов и деструктора. Разработан класс вектор (одномерный динамический массив). Методы класса: конструкторы, деструктор и несколько методов, выполняющих преобразование в массиве, например, нахождения максимального элемента

```

#include <windows.h>
#include <iostream>
#include <iomanip>
using namespace std;
class Vector {
    float* arr; //вектор (одномерный массив)
    int size; //размерность вектора
public:
    Vector(); //конструктор без параметров
    Vector(int, float*); //конструктор с двумя параметрами
    ~Vector(); //деструктор
    void set(); //метод инициализация вектора
    void print(); //вывод вектора на экран
    float findMax(); //нахождения максимума в векторе
};

```

```

Vector::Vector() : size(0), arr(0){} //конструктор без параметров
Vector::Vector(int s, float* a) {} //конструктор с двумя
параметрами

```



```

        size = s;//инициализация размерности
        arr = new float[size];//выделение памяти под вектор
        for (int i = 0; i < size; i++) *(arr + i) = *(a + i);
    }
    Vector::~Vector() { delete[] arr; }//деструктор
    void Vector::set() { //инициализация вектора
        if (!arr) {
            cout << "не выделена память под вектор" << endl;
            cout << "введите размерность вектора" << endl;
            cin >> size;
            arr = new float[size];
        }
        cout << "Введите элементы в вектор" << endl;
        for (int i = 0; i < size; i++) cin >> *(arr + i);
    }
    void Vector::print() { //функция вывода вектора на экран
        if (!arr) {
            cout << "вектор пустой" << endl;
            return;
        }
        for (int i = 0; i < size; i++) cout << setw(4)<<*(arr + i);
        cout << endl;
    }
    float Vector::findMax() { //функция нахождения максимума в
    векторе
        float max = *arr;//инициализация переменной первым
    элементом массива
        for (int i = 0; i < size; i++)
            if (max < *(arr + i))
                max = *(arr + i);
        return max;
    }
    int main() {
        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
        float max;//переменная для хранения максимального элемента
        Vector v1;//вызывается конструктор без параметров
        float arr[] = { 2.5,1.6,6.7,4.8,5.6 };//объявление и
    инициализация массива
        Vector v2((sizeof(arr) / sizeof(int)), arr);//вызов
    конструктора с параметрами
        v1.set();//метод инициализирует объект v1
        cout << "Вектор v1" << endl;
        v1.print();//метод выводит на экран содержимое объекта v1
        max = v1.findMax();//метод возвращает максимальный элемент
    вектора
        cout << "Максимальное число в векторе 1= " << max << endl;
        cout << "Вектор v2" << endl;
        v2.print();//метод выводит на экран содержимое объекта v2
    }
}

```



```

        max = v2.findMax();//метод возвращает максимальный элемент
вектора
        cout << "Максимальное число в векторе 2= " << max << endl;
        return 0;
}

```

Результаты работы программы:

```

не выделена память под вектор
введите размерность вектора
6
Введите элементы в вектор
3.2
3.5
9.8
1.2
2.7
0.6
Вектор v1
3.2 3.5 9.8 1.2 2.7 0.6
Максимальное число в векторе 1= 9.8
Вектор v2
2.5 1.6 6.7 4.8 5.6
Максимальное число в векторе 2= 6.7

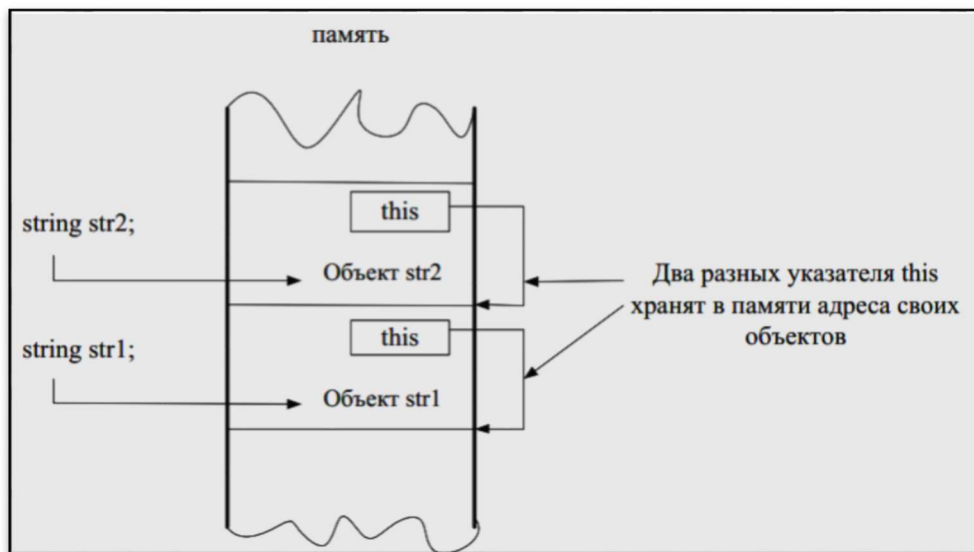
```



## УКАЗАТЕЛЬ THIS

При каждом вызове метода класса ему автоматически передается указатель, именуемый ключевым словом *this*, который указывает на объект, для которого вызывается этот метод. Указатель *this* — это неявный параметр, принимаемый всеми методами-элементами класса. Следовательно, в любом методе-элементе класса указатель *this* можно использовать для ссылки на вызывающий объект.

Каждый объект имеет скрытый от пользователя указатель *this*. Иначе это можно объяснить так. Когда объявляется объект, под него выделяется память. В памяти есть специальное поле, содержащее скрытый указатель, который адресует начало выделенной под объект памяти. Получить значение указателя в компонентах-методах класса можно с помощью ключевого слова *this*.



Для любого объекта класса, например, класса *Game*, указатель *this* неявно объявлен так:

```
Game *const this;
```

### Основные свойства и правила использования указателя *this*:

- каждый новый объект имеет свой скрытый указатель *this*;
- указатель *this* указывает на начало своего объекта в памяти компьютера;
- указатель *this* не надо дополнительно объявлять;
- указатель *this* передается как скрытый аргумент во все нестатические (т. е. не имеющие спецификатора *static*) компоненты-методы;
- указатель *this* является локальной переменной, которая недоступна за пределами объекта;
- можно обращаться к указателю *this* непосредственно в виде *this* или *\*this*.

```
//Пример №8. Пример использования указателя this
#include <iostream>
#include <string>
#include <windows.h>
using namespace std;
class Account {
    int numberOfAccount;
public:
    void setNumber(int numberOfAccount) { this->numberOfAccount = numberOfAccount; }
    int getNumber() { return this->numberOfAccount; } //тоже самое, что return numberOfAccount
};
```

```
int main() {  
    Account firstAccount;  
    firstAccount.setNumber(100);  
    cout << firstAccount.getNumber();  
    return 0;  
}
```

## ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ STATIC ПЕРЕМЕННЫЕ

Локальная *static*-переменная сохраняет свое значение между вызовами функции. Если к локальной переменной применен модификатор *static*, то для нее выделяется постоянная область памяти практически так же, как и для глобальной переменной. Это позволяет статической переменной поддерживать ее значение между вызовами функций (другими словами, в отличие от обычной локальной переменной, значение *static*-переменной не теряется при выходе из функции). **Ключевое различие между статической локальной и глобальной переменными состоит в том, что статическая локальная переменная известна только блоку, в котором она объявлена, а глобальная известна всему файлу.** Таким образом, статическую локальную переменную в некоторой степени можно назвать глобальной переменной, которая имеет ограниченную область видимости. Чтобы объявить статическую переменную, достаточно предварить ее тип ключевым словом *static*. Например, при выполнении этой инструкции переменная *count* объявляется статической.

```
static int count;
```

Статической переменной можно присвоить некоторое начальное значение. Например, в этой инструкции переменной *count* присваивается начальное значение 200:

```
static int count = 200;
```

**Локальные *static*-переменные инициализируются только однажды, в начале выполнения программы, а не при каждом входе в функцию, в которой они объявлены.** Возможность использования статических локальных переменных важна для создания независимых функций, поскольку существуют такие типы функций, которые должны сохранять значения между их вызовами. Если бы статические переменные не были предусмотрены в C++, то пришлось бы использовать вместо них глобальные, что открыло бы путь для всевозможных побочных эффектов глобальной видимости.

Глобальная *static*-переменная известна только файлу, в котором она объявлена. Если модификатор *static* применен к глобальной переменной, то компилятор создаст глобальную переменную, которая будет известна только

файлу, в котором она объявлена. Это означает, что, хотя эта переменная является глобальной, другие функции в других файлах не имеют о ней "ни малейшего понятия" и не могут изменить ее содержимое. Поэтому она и не может стать "жертвой" несанкционированных изменений. Следовательно, для особых ситуаций, когда локальная статичность оказывается бессильной, можно создать небольшой файл, который будет содержать лишь функции, использующие глобальные *static*-переменные, отдельно скомпилировать этот файл и работать с ним, не опасаясь вреда от побочных эффектов "всеобщей глобальности".

Несмотря на то, что глобальные *static*-переменные по-прежнему допустимы и широко используются в C++-коде, стандарт C++ возражает против их применения. Для управления доступом к глобальным переменным рекомендуется другой подход, который заключается в использовании пространств имен.

Пространство имен (*namespace*) — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью исключения конфликтов имен, которые могут возникнуть, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут получить доступ к элементам двумя вариантами:

- используя полное имя идентификатора, например `std::vector<std::string> vec;`

- используя объявление *using* для отдельного идентификатора (*using* `std::string;`) или директивы *using* для всех идентификаторов в пространстве имен (*using namespace* `std;`).

## STATIC-ДАННЫЕ КЛАССА

Компоненты-данные могут быть объявлены с модификатором (спецификатором) класса памяти *static*. *Static*-компонента совместно используется всеми объектами этого класса и хранится в одном месте. Статическая компонента класса должна быть явно определена в контексте файла. **Основные правила использования статических компонент:**

- статический компонент класса совместно используется всеми объектами этого класса, т. е. он использует одну область памяти;
- статические компоненты не являются частью объектов класса;
- объявление статических компонент-данных в классе не является их определением. Они должны быть явно определены в контексте файла;
- к статической компоненте *count* класса *Level* можно обращаться *Level::count* независимо от объектов этого класса, а также при помощи операций «.» (точка) и «->» (стрелка) при использовании объектов этого класса;

- статическая компонента существует даже при отсутствии объектов этого класса;
- статические компоненты можно инициализировать, как и другие глобальные объекты, только в файле, в котором они объявлены.

## STATIC И CONST КОМПОНЕНТЫ-МЕТОДЫ КЛАССА

В языке C++ компоненты-методы могут использоваться с модификаторами *static* и *const*. Любой метод класса имеет явный и неявный список параметров. Неявные параметры можно представить как список параметров, доступных через указатель *this*. Статический метод не может обращаться к указателю *this*, а метод *const* не может изменять неявные параметры.

### Основные свойства и правила использования *static* функций:

- статические методы не имеют указателя *this*;
- в классе не могут быть объявлены два метода с одинаковыми сигнатурами, но, чтобы при этом один метод был статическим, а другой нет;
- статические методы не могут быть виртуальными.

```
//Пример №9. Использование статических методов-элементов класса
#include <windows.h>
#include <iostream>
using namespace std;
class Box {
public:
    static int objectCount;
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
        // Increase every time object is created
        objectCount++;
    }
    double Volume(){return length * breadth * height;}
    static int getCount() {return objectCount;}
private:
    double length;// Length of a box
    double breadth;// Breadth of a box
    double height;// Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void) {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    // Print total number of objects before creating object.
```

```

        cout << "Первоначальное количество объектов: " <<
Box::getCount() << endl;
        Box Box1(3.3, 1.2, 1.5); // Declare box1
        Box Box2(8.5, 6.0, 2.0); // Declare box2
        // Print total number of objects after creating object.
        cout<<"Конечное количество объектов: " <<Box::getCount()<<
endl;
        return 0;
}

```

Результаты работы программы:

```

Первоначальное количество объектов: 0
Constructor called.
Constructor called.
Конечное количество объектов: 2

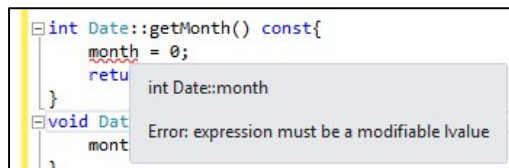
```

```

//Пример №10. Применение константных методов класса
class Date{
public:
    Date(int mn, int dy, int yr);
    int getMonth() const; //A read-only function
    void setMonth(int mn); //A write function; can't be const
private:
    int month;
};
int Date::getMonth() const{
    //month = 0;
    return month; //Doesn't modify anything
}
void Date::setMonth(int mn){
    month = mn; //Modifies data member
}
int main(){
    Date MyDate(7, 4, 2015);
    MyDate.setMonth(4);
    cout<<MyDate.getMonth();
}

```

Метод *setMonth()* может изменить значение поля *month*, а константный метод *getMonth()* не может этого сделать. Если со стороны метода *getMonth()* будет предпринята попытка изменить поле *month*, то компилятор выдаст сообщение об ошибке:



```

int Date::getMonth() const{
    month = 0;
    return month;
}
void Date::setMonth(int mn){
    month = mn;
}

```

Error: expression must be a modifiable lvalue

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем разница между *struct*, *class*, *union*?
2. Что такое указатель *this*? Приведите пример использования этого указателя.

3. Какова основная форма конструктора копирования и назовите ситуации, в которых он вызывается?
4. Когда вызывается конструктор?
5. Когда вызывается деструктор?
6. Можно ли конструктор и деструктор перегрузить?
7. Приведите пример использования константных и статических данных и методов.
8. Приведите пример использования локальных и глобальных статических переменных.
9. В чем различие передачи аргумента по ссылке и по значению?
10. Создается ли конструктор по умолчанию, если в классе создан один конструктор с параметрами?

### **ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы по индивидуальному заданию.
5. Написать, отладить и проверить корректность работы созданной программы.
6. Написать электронный отчет.

**Отчет должен быть оформлен по стандарту БГУИР ([Стандарт предприятия СТП 01-2017 "Дипломные проекты \(работы\). Общие требования"](#)) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теория по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания
6. скриншоты выполнения индивидуального задания
7. выводы по лабораторной работе

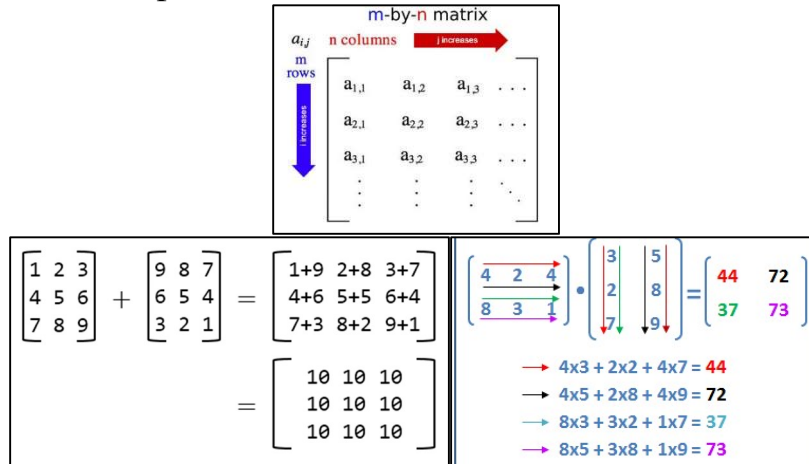
**В РАМКАХ ВСЕГО КУРСА ООП ВСЕ ЛАБОРАТОРНЫЕ РАБОТЫ ДОЛЖНЫ ХРАНИТЬСЯ В ОДНОМ РЕШЕНИИ (SOLUTION), В КОТОРОМ ДОЛЖНЫ БЫТЬ СОЗДАНЫ ОТДЕЛЬНЫЕ ПРОЕКТЫ (PROJECTS) ДЛЯ КАЖДОЙ ЛАБОРАТОРНОЙ РАБОТЫ.**

### **ВАРИАНТЫ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ №1.**

**Напишите программу, которая позволяет выбрать номер лабораторной работы и выполняет действия, указанные в задании.**

1. Создать класс *Matrix*, в котором реализовать методы для работы с матрицами: перемножение матриц, сложение матриц. Память под матрицы в конструкторах выделять динамически. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор, который должен освобождать память, выделенную под матрицы.

В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

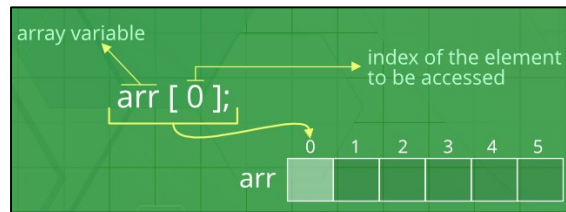


2. Создать класс *Employee*. Класс должен включать поле *int* для хранения идентификационного номера сотрудника и поле *float* для хранения величины его оклада. В классе *Employee* должны быть так же включены поля типа класса *Date* и перечисления *Etype*. Поле типа *Date* использовать для хранения даты приема сотрудника на работу. Поле типа *Etype* использовать для хранения статуса сотрудника: лаборант, секретарь, менеджер и т.д. Разработать методы *getSmth()* и *putSmth()*, предназначенные соответственно для ввода и отображения различной информации о сотруднике. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Написать программу, которая будет запрашивать у пользователя и выводить на экран сведения о сотрудниках. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

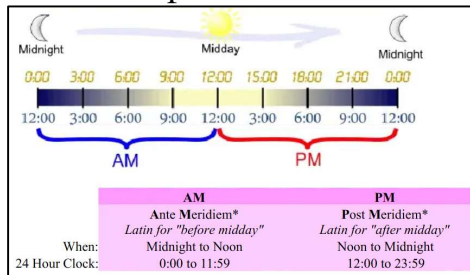


3. Создать класс *Array*, в котором реализовать методы для работы с одномерными массивами: получить пересечение элементов массивов, получить объединение элементов массивов. Память под массивы выделять в конструкторе динамически. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Деструктор должен освобождать динамическую память, выделенную под массивы. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

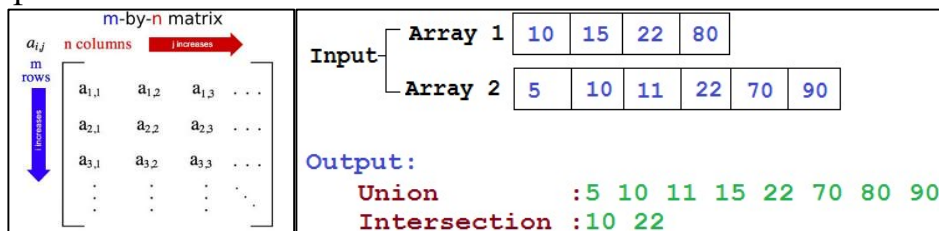




4. Создать класс *Time* с полями: час (0–23), минуты (0–59), секунды (0–59). В классе реализовать конструкторы (с параметрами, без параметров, конструктор копирования), деструкторы, методы установки времени, получения значения часов, минут, секунд, а также методы печати: печать по шаблону «16 часов 18 минут 3 секунды» и «04:18:03 p.m.(a.m.)». Методы установки полей класса должны проверять корректность задаваемых параметров. В деструкторе выводить сообщение о его срабатывании. В программе создать различные варианты срабатывания деструктора: 1. самостоятельно удалить указатель на объект или объект; 2. объект выходит из области видимости; 3. объект принадлежит классу, деструктор которого вызывается. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

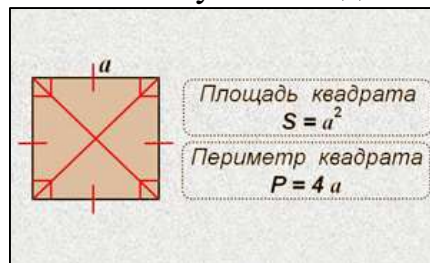


5. Создать класс *Matrix*, в котором реализовать методы для работы с двумерными массивами: получить пересечение элементов массивов; получить объединение элементов массивов. Память под массивы выделять динамически. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Деструктор должен освобождать динамическую память, выделенную под массивы. В программе создать различные варианты срабатывания деструктора: 1. самостоятельно удалить указатель на объект или объект; 2. объект выходит из области видимости; 3. объект принадлежит классу, деструктор которого вызывается. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

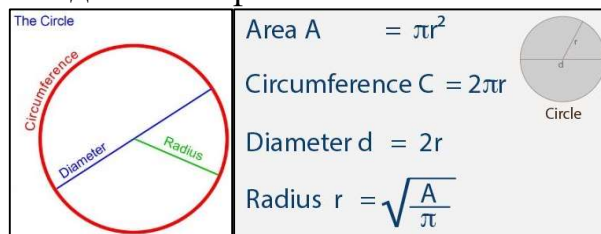


6. Создать класс *Employee*, одно из полей которого хранит «порядковый номер» созданного объекта, т.е. для первого созданного объекта значение этого поля равно 1, для второго – 2 и т. д. Для того чтобы создать такое поле, необходимо иметь еще одно поле, в котором будет храниться количество созданных объектов. Каждый раз при создании нового объекта, конструктор может получить значение этого поля и в соответствии с ним назначить объекту индивидуальный порядковый номер. В классе разработать метод, который будет выводить на экран свой порядковый номер, например: «Мой порядковый номер: 2». В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Предусмотреть метод для записи полученных данных в файл.

7. Создать класс *Square*. Поле класса хранит длину стороны квадрата. Методы-элементы класса возвращают площадь, периметр, устанавливают поля и возвращают значения полей класса, выводят данные о полях класса на экран. Методы-элементы установки полей класса должны проверять корректность задаваемых параметров. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

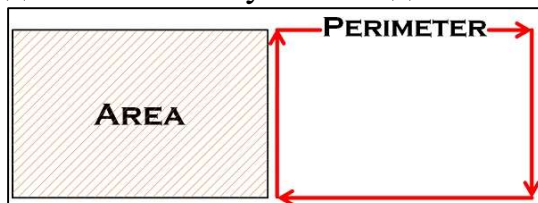


8. Создать класс *Circle*. Поле класса хранит радиус окружности. Методы-элементы класса возвращают площадь, длину окружности, устанавливают радиус окружности и возвращают значения радиуса окружности, выводят данные о полях класса на экран. Методы-элементы установки полей класса должны проверять корректность задаваемых параметров. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

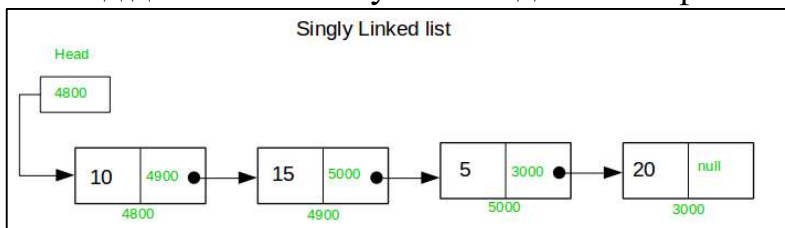


9. Создать класс *Rectangle*. Поля класса: высота и ширина прямоугольника. Методы-элементы класса: вычисление площади, периметра, установление данных класса и возвращение их значений, вывод данных на

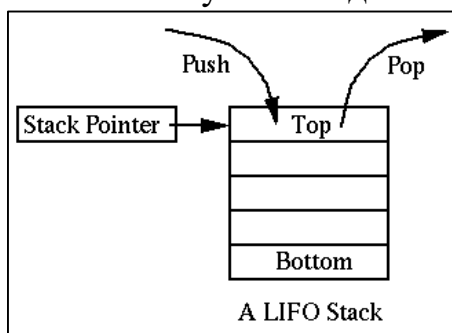
экран. Методы-элементы установки полей класса должны проверять корректность задаваемых параметров. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.



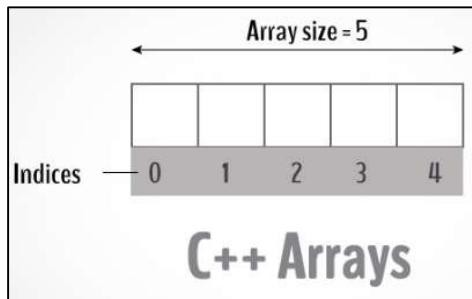
10. Создать два класса *SingleLinkedListInt* и *SingleLinkedListChar* для хранения данных соответствующего типа. Методы-элементы добавляют элемент в список, удаляют элемент из списка, выводят на экран элементы списка, находят элемент в списке. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.



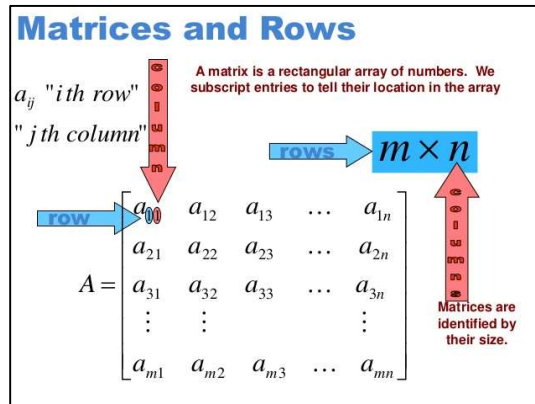
11. Создать класс типа *Stack*. Методы-элементы класса добавляют элемент в стек, удаляют элемент из стека, выводят на экран элементы стека, ищут наибольший элемент в стеке. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.



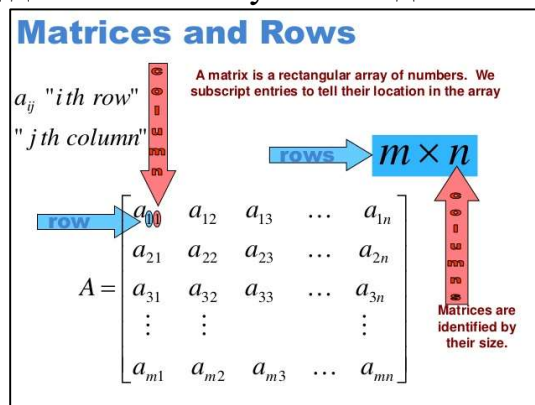
12. Создать класс, в котором реализуются методы для работы с одномерными массивами: метод поиска максимального числа в массиве; метод поиска минимального числа в массиве; метод, меняющий местами минимальное и максимальное значения в массивах. Память под массивы выделять динамически. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Деструктор должен освобождать память, выделенную под массивы. Предусмотреть метод для записи полученных данных в файл.



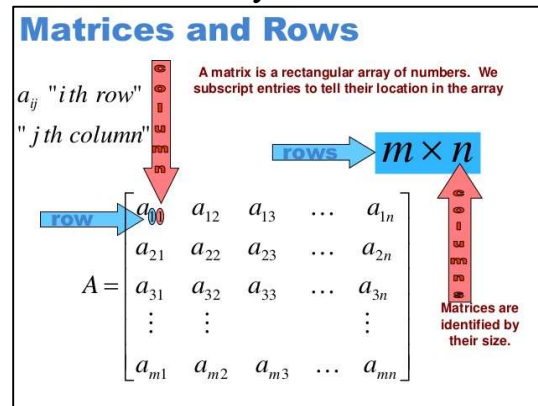
13. Создать класс, в котором реализовать методы для работы с двумерными массивами: метод, обнуляющий строку и столбец с минимальным элементом; метод, который в каждой строке массива первый найденный нечетный элемент меняет местами с первым элементом этой строки. Память под массивы выделять динамически. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Деструктор должен освобождать динамическую память, выделенную под массивы. Предусмотреть метод для записи полученных данных в файл.



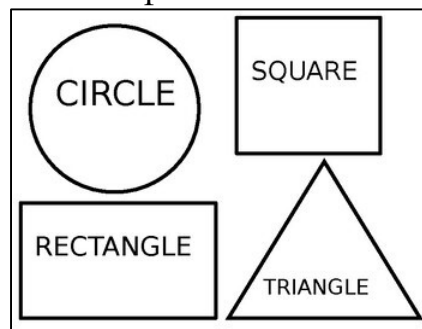
14. Создать класс, в котором надо реализовать методы для работы с двумерными массивами: метод поиска минимального элемента ниже главной диагонали; метод поиска максимального элемента выше главной диагонали. Память под массивы выделять динамически. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Деструктор должен освобождать память, выделенную под массивы. Предусмотреть метод для записи полученных данных в файл.



15. Создать класс, в котором требуется реализовать методы для работы с матрицами: метод находит минимальное число в матрице, максимальное число и меняет их местами; метод выполняет обмен элементов первой и второй, третьей и четвертой и т. д. строк матрицы. Память под матрицы выделять динамически. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Деструктор должен освобождать динамическую память, выделенную под матрицы. Предусмотреть метод для записи полученных данных в файл.



16. Построить систему классов для описания плоских геометрических фигур: круг, квадрат, прямоугольник. Предусмотреть методы для создания объектов, перемещения их на плоскости, изменения размеров фигур. Написать программу, демонстрирующую работу с этими классами. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.



17. Построить описание класса, содержащего информацию о почтовом адресе. Предусмотреть возможность отдельного изменения составных частей адреса, создания и уничтожения объектов этого класса. Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.

Sender's address

---



---



---

stamp

ZIP/Postal Code of Recipient

Recipient's address

**Room #** \_\_\_\_\_

**Street** \_\_\_\_\_ \*

**Moscow**

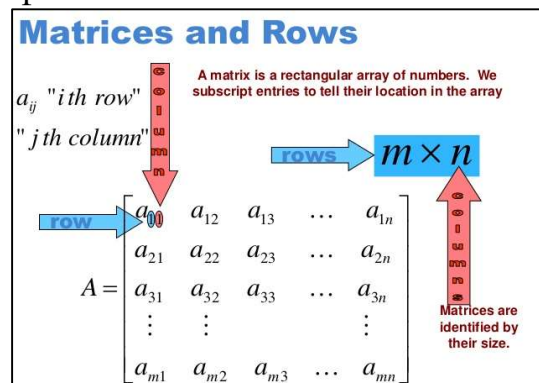
**101000 Russia**

Recipient's name: Jane Doe

Recipient's department: \_\_\_\_\_

\* write the address of your building. Make sure to confirm the address with your coordinator: responsible person at the department.

18. Составить описание класса, обеспечивающего представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывод на экран подматрицы любого размера и всей матрицы. Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.



19. Описать класс «библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям. Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Предусмотреть метод для записи полученных данных в файл.





20. Создать класс, описывающий понятие работник, со свойствами: ФИО, стаж, часовая заработная плата, количество отработанных часов. Реализовать ввод данных работника с клавиатуры. Рассчитать с помощью методов класса заработную плату за отработанное время, и премию, размер которой определяется в зависимости от стажа работника (при стаже до 1 года 0%, до 3 лет 5%, до 5 лет 8%, свыше 5 лет 15%). С помощью метода печати, реализовать вывод информации о работнике на экран. Предусмотреть метод для записи в файл данных о работнике. Предусмотреть метод для записи полученных данных в файл.



21. Создать класс, описывающий понятие работник, со свойствами: ФИО, стаж, часовая заработная плата, количество отработанных часов, налоговые льготы. Составить программу, которая создает массив объектов разработанного класса и по введенной месячной зарплате, вычисляет подоходный налог каждого из работников компании. Создать методы, которые в массиве работников найдут работника с максимальной заработной платой и работника с минимальным подоходным налогом. Предусмотреть метод для записи полученных данных в файл.



22. Создать класс *Tile* (плитка), который будет содержать поля *brand*, *height*, *weight*, *price*, *count* и методы работы с данными класса. Создать массив объектов разработанного класса, заполнить объекты данными. Посчитать общую стоимость хранящейся плитки на складе. В класс добавить необходимый набор полей и методов (минимум два поля и два метода) на свое усмотрение. Использовать конструктор с параметрами, конструктор без

параметров, конструктор копирования. Предусмотреть метод для записи полученных данных в файл.



23. Создать класс зоомагазин. В классе должны быть следующие поля: наименование магазина, массив животных (вид, цена, количество). Включить в состав класса необходимый минимум методов, обеспечивающий полноценное функционирование объектов указанного класса: конструкторы (по умолчанию, с параметрами, копирования); деструктор; предоставить возможность вводить данные с клавиатуры и выводить их на экран. Предусмотреть метод для записи полученных данных в файл.

