

## ЛАБОРАТОРНАЯ РАБОТА № 24 КОНСТРУКТОРЫ И ДЕКТРУКТОРЫ

### ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

#### Порядок создания и использования класса

В предыдущей лабораторной работе подробно описывалось понятие класс и порядок его создания и использования. **Первый шаг** при разработке класса – объявление класса, которое может включать члены-данные (атрибуты) и члены-функции (методы). Объявление имеет раздел **private**, и члены, объявленные в этом разделе, могут быть доступны только через методы класса. Объявление также содержит раздел **public**, и объявленные в нем члены могут быть непосредственно доступны программе, использующей объекты класса. Как правило, атрибуты записывают в закрытый раздел, а методы — в открытый, поэтому типичное объявление класса имеет следующую форму:

```
class ИмяКласса
{
private:
    // объявления членов-данных
public:
    // объявления членов-функций
};
```

**Второй шаг** при разработке класса — это реализация методов класса. Вместо прототипов в объявление можно включать полное определение функций, однако общепринятая практика состоит в том, чтобы определять функции отдельно, за исключением наиболее простых. В этом случае используют операция “::” для указания того, к какому классу данная функция-член принадлежит, например, пусть класс **MPoint** имеет функцию **BegDistance()**, Определение данной функции-члена класса вне класса будет выглядеть так:

```
int MPoint::BegDistance()
{
    return sqrt(X*X + Y*Y);
}
```

**Для создания объекта** на базе разработанного класса, используется имя класса, как если бы оно было именем типа:

```
MPoint p1; // Объявление объекта p1 класса MPoint
```

Это работает потому, что класс является *типом, определенным пользователем*. Функция-член класса (метод класса), вызывается с использованием объекта класса. Это делается с помощью операции **точка**:

```
cout << p1.BegDistance();
```

Код вызывает метод **BegDistance()**, и, когда код этой функции обращается к членам-данным **X** и **Y** класса, используются значения атрибутов объекта **p1**.

#### Конструкторы

При создании объекта класса **MPoint**, например, при таком объявлении:

```
MPoint p1, p2;
```

в памяти только выделяется место в памяти под поля каждого объекта. , которые заполнены случайными числами, а начальная инициализация не выполняется. Это означает, что значения полей объектов могут быть любыми.

Чтобы присвоить нужные значение полям только что созданного объекта, используется специальный член класса – **конструктор**. Конструктор – это функция-

член класса, **автоматически вызываемая программой при создании экземпляра класса.**

Конструкторы используются для присваивания начальных значений атрибутам объекта, для выделения динамической памяти под внутренние массивы объекта и других целей. **Имя конструктора должно совпадать с именем класса.** Конструктор не имеет возвращаемого значения, даже слово **void** не пишется.

У функции может быть несколько конструкторов, отличающихся количеством или типом аргументов. Конструктор без параметров называется **конструктором по умолчанию**. Конструктор по умолчанию — это конструктор, который используется для создания объекта, когда не предоставлены явные инициализирующие значения, т.е. это конструктор, который применяется для объявлений, подобных показанному ниже:

```
MPoint p1; // используется конструктор по умолчанию
```

Например, для класса **MPoint** конструктор по умолчанию будет выглядеть так:

```
MPoint() {X = 0; Y = 0;}
```

В этом конструкторе атрибутам объекта будут присвоены значения 0.

Если же нужно создать объект с заранее известными значениями, например, чтобы можно было объявлять объекты так:

```
MPoint p2(100, 150);
```

То понадобится конструктор с двумя параметрами, который нужно объявить так:

```
MPoint(int i, int j) {X = i; Y = j;}
```

В этом конструкторе атрибутам объекта будут присвоены значения, указанные при объявлении объекта.

При создании экземпляра класса компилятор сам определяет, какой конструктор вызывать, по числу и типу указанных в объявлении объекта параметров.

```
MPoint p2(100, 150); // в p2 будет установлено: X = 100, Y = 150
```

В целом класс с конструкторами будет выглядеть так:

```
class MPoint
{
private:
    int X, Y;
public:
    MPoint() {X = 0; Y = 0;}
    MPoint(int i, int j) {X=i; Y=j;}
    void SetX(int x) {X = x;}
    void SetY(int y) {Y = y;}
    void SetXY(int x, int y) {X=x; Y=y;}
    int GetX() {return X;}
    int GetY() {return Y;}
    int BegDistance();
    void ShiftX(int dx) {X += dx;}
    void ShiftY(int dy) {Y += dy;}
};
```

← Конструктор по умолчанию

← Конструктор с параметрами

Конструкторы позволяют создавать объекты с заданными начальными значениями:

```
int main()
{
    int d1, d2;
    MPoint p1; // в p1 установлено: x = 0 y = 0
    MPoint p2(100, 150); // в p2 установлено: x = 100 y = 150

    d1 = p1.BegDistance();
    d2 = p2.BegDistance();
}
```

Если никаких конструкторов в классе не создавать, то компилятор сам создает конструктор по умолчанию. Если же объявить хотя бы один конструктор, то компилятор перестанет создавать конструктор по умолчанию. Например, если создать только конструктор с параметрами, то конструктора по умолчанию в классе не будет, и создавать объекты так:

```
MPoint p1; // нужен конструктор по умолчанию
```

будет нельзя.

Конструкторы, как и другие функции в языке C++, могут иметь **аргументы по умолчанию**. Аргумент по умолчанию представляет собой значение, которое используется автоматически, если соответствующий фактический параметр в вызове функции не указан. Например, если функция

```
float Degree (float f, int n); // Возведение f в степень n
```

определена так, что **n** по умолчанию имеет значение **2**, то вызов функции

```
a = Degree(5);
```

означает то же самое, что и

```
a = Degree(5, 2);
```

Чтобы установить значение по умолчанию, нужно прямо в прототипе функции указать нужные значения по умолчанию:

```
float Degree (float f, int n = 2); // Возведение f в степень n
```

В функции со списком аргументов значения по умолчанию должны добавляться в конце. То есть, нельзя предоставить значение по умолчанию некоторому аргументу до тех пор, пока не будут предоставлены значения по умолчанию для всех аргументов, размещенных справа от него:

```
int function_1(int i, int j = 4, int k = 5); // Правильно
int function_2(int i, int j = 6, int k);   // Неправильно
int function_3(int i = 1, int j = 2, int k = 3); // Правильно
```

Теперь функцию `function_3` можно вызывать четырьмя способами:

```
function_3(12, 13, 14); // Заданы все три параметра
function_3(12, 13);     // То же, что и function_3(12, 13, 3);
function_3(12);         // То же, что и function_3(12, 2, 3);
function_3();           // То же, что и function_3(1, 2, 3);
```

Аргументы по умолчанию предназначены для удобства программирования. Например, они позволяют сократить количество конструкторов, методов и перегрузок методов, подлежащих определению.

В классе **MPoint** тоже можно сократить число конструкторов, если объявить конструктор с двумя параметрами так:

```
MPoint(int i = 0, int j = 0) {X = i; Y = j;}
```

В этом случае можно будет создавать объекты тремя способами:

```
MPoint p3(100, 150); // в p3 установлено:  x = 100  y = 150
MPoint p4(100);      // в p4 установлено:  x = 100  y = 0
MPoint p5();         // в p5 установлено:  x = 0   y = 0
```

Чтобы при последнем объявлении не возникло проблем, нужно из объявления класса убрать конструктор

```
MPoint() {X = 0; Y = 0;}
```

т.к. новый конструктор со значениями по умолчанию **0, 0** и будет играть роль конструктора по умолчанию.

Конструкторы вызываются при создании глобальных и локальных объектов, а также при создании статических и динамических массивов объектов. При создании динамических массивов объектов всегда вызываются конструкторы по умолчанию.

## Деструкторы

При уничтожении объекта автоматически вызывается другая функция – деструктор. Имя деструктора состоит из символа '~' (тильда) и имени класса. Деструктор, как и конструктор, не имеет возвращаемого значения, даже слово **void** не пишется. Деструктор не имеет аргументов, поэтому в классе может быть только один. Например, для класса **MPoint** деструктор будет выглядеть так:

```
~MPoint();
```

Деструктор чаще всего используется для освобождения памяти. Если в конструкторе используется операция **new** для выделения памяти, то в деструкторе нужно ее освободить с помощью операции **delete**.

В классе **MPoint** память динамически не выделяется, поэтому деструктор не нужен. Но просто для того, чтобы увидеть, когда вызывается конструктор, определим его, например, так:

```
~MPoint() { cout << "Деструктор MPoint" << endl; }
```

С конструктором и деструктором класс будет выглядеть следующим образом:

```
class MPoint
{
private:
    int X, Y;
public:
    MPoint(int i = 0, int j = 0) {X = i; Y = j;} // Конструктор
    ~MPoint() {cout << "Деструктор MPoint" << endl;} // Деструктор

    void SetX(int x) {X = x;}
    void SetY(int y) {Y = y;}
    void SetXY(int x, int y) {X=x; Y=y;}
    int GetX() {return X;}
    int GetY() {return Y;}
    int BegDistance(){ return sqrt((float)X * X + Y * Y); }
    void ShiftX(int dx) {X += dx;}
    void ShiftY(int dy) {Y += dy;}
};
```

Проиллюстрировать работу конструктора и деструктора можно на следующем примере:

```
int main()
{
    MPoint p1;           // в p1 установлено:  x = 0    y = 0
    MPoint p2(3, 4);     // в p2 установлено:  x = 3    y = 4
    MPoint p3(10);       // в p2 установлено:  x = 10   y = 0

    cout << p1.BegDistance() << endl;
    cout << p2.BegDistance() << endl;
    cout << p3.BegDistance() << endl;

    return 0;
}
```

Результат работы программы:

```
0
5
10
Деструктор MPoint
Деструктор MPoint
Деструктор MPoint
```

Если создается глобальный объект класса, то его деструктор вызывается автоматически при завершении работы программы. Если создается локальный объект

класса, как в приведенном примере, то его деструктор вызывается автоматически, когда выполнение программы покидает блок кода, в котором определен объект. Если объект создается динамически с использованием операции **new**, его деструктор вызывается, когда объект уничтожается с помощью операции **delete**.

Если деструктор в классе не создавать, то компилятор сам создает его, но никаких действий в нем не будет..

### Доработка класса **Время**

В предыдущей лабораторной работе описывался процесс создания класса на следующем примере: **Время суток**. Доработаем класс, добавим в него конструкторы, полезные методы и напишем программу, иллюстрирующую возможности данного класса.

Класс назывался **CTime**, в нем хранилось время суток, поэтому было три атрибута: час, минута и секунда. Каждый параметр ограничен значениями: 0-23 (час) или 0-59 (минута и секунда), следовательно, для хранения каждого из них требуется по одному байту. Так как отрицательными параметры не могут быть, мы использовали тип **unsigned char**. Чтобы не писать каждый раз **unsigned char**, зададим новое имя с помощью оператора **typedef**:

```
typedef unsigned char BYTE;
```

Объявление класса выглядело так:

```
class CTime
{
private:
    BYTE Hour;
    BYTE Min;
    BYTE Sec;
public:
    void SetHour(BYTE h) {if (h < 24) Hour = h;} // Установка часов
    void SetMin(BYTE m)  {if (m < 60) Min = m; } // Установка минут
    void SetSec(BYTE s)  {if (s < 60) Sec = s; } // Установка секунд
    void Set(BYTE h, BYTE m, BYTE s)
    { SetHour(h); SetMin(m); SetSec(s); }

    BYTE GetHour() {return Hour;} // Чтение часов
    BYTE GetMin()  {return Min;}  // Чтение минут
    BYTE GetSec()  {return Sec;}  // Чтение секунд
};
```

При создании объектов класса желательно инициализировать его параметры (час, минута и секунда) некоторыми реальными значениями, для этого объявляем конструктор по умолчанию (в котором задаем начальное время **00:00:00**) и конструктор с параметрами час, минута и секунда. Помещаем члены-данные в закрытую часть класса, а конструкторы в открытую:

```
class CTime
{
private:
    BYTE Hour;
    BYTE Min;
    BYTE Sec;
public:
    CTime() {Hour = 0; Min = 0; Sec = 0;} // Конструктор по умолч.
    CTime(BYTE h, BYTE m, BYTE s); // Конструктор с параметрами
    . . . .
};
```

Чтобы при использовании конструктора с параметрами нельзя было бы установить неверное время, например, такое:

```
CTime t1(100,100,100);
```

применим в нем уже написанный метод установки всех параметров одной функцией, в которой делается проверка на корректность:

```
void Set(BYTE h, BYTE m, BYTE s){SetHour(h); SetMin(m); SetSec(s);}
```

Конструктор с параметрами будет выглядеть так:

```
CTime(BYTE h, BYTE m, BYTE s)
{
    Hour = 0; // Начальные значения на случай, если h, m, или s
    Min = 0;  // окажутся некорректными
    Sec = 0;
    SetTime(h, m, s); // Установка значений с проверкой на корректность
}
```

Начальные значения **Hour**, **Min** и **Sec** задаем равными нулю, чтобы в случае попытки установки некорректных данных обнулить эти значения.

Чтобы задавать время строкой (например, в формате “12:05:23”), напомним функцию – член класса, у которой в качестве аргументов будет строка (т.е. тип аргумента будет **const char\***). В самой функции уже выделим нужные подстроки и преобразуем их в числа. Если строка будет некорректна – менять время не будем. Функция большая, поэтому помещаем объявление в класс:

```
void Set(const char* str);
```

а тело функции напишем вне класса. Тип аргумента – **const char\***, чтобы можно было вызывать эту функцию и с константными строками.

Для выделения подстрок можно использовать различные библиотечные функции, а можно работать непосредственно со строкой. Для этого сначала убеждаемся в том, что символы ':' стоят на своих местах (индексы в строке – 2 и 5). Затем берем из начала строки два первых символа, соответствующие часу, и преобразуем их в число (для этого из кода символа вычитаем код символа '0'). Затем число, полученное из левого символа (обозначающего десятки), умножаем на 10 и суммируем с числом, полученным из правого символа (обозначающего единицы). Аналогично получаем минуты и секунды.

```
void CTime::Set(const char* str)
{ // преобразуем строку в формате "12:34:56" в час, мин и сек
    BYTE h, m, s;
    if (str[2] == ':' && str[5] == ':') // Если символы стоят на
    {                                     // своих местах
        h = (str[0] - '0') * 10 + str[1] - '0';
        m = (str[3] - '0') * 10 + str[4] - '0';
        s = (str[6] - '0') * 10 + str[7] - '0';
        SetTime(h, m, s);
    }
}
```

Для того, чтобы создавать объект класса **CTime** с параметром-строкой, создадим еще один конструктор, в котором используем написанную функцию.

```
CTime(const char* str)
{
    Hour = 0; // Начальные значения на случай, если строка
    Min = 0;  // окажется некорректной
    Sec = 0;
    SetTime(str); // Установка значений строкой
}
```



Теперь в классе три конструктора, поэтому можно создавать объекты тремя способами:

```
CTime t1;  
CTime t2(12, 25, 30);  
CTime t3("23:59:59");
```

Изменять время в объекте тоже можно по-разному:

```
t1.Set("12:00:00");  
t2.Set(5,30,0);  
t3.SetHour(10);  
t3.SetMin(20);  
t3.SetSec(30);
```

Чтобы читать время строкой в формате "**12:05:23**", напишем функцию, формирующую и возвращающую эту строку. Это можно сделать, поместив указатель на строку (**char\* str**) в список аргументов и заполняя переданную в функцию строку нужными данными. Использовать указатель на строку в качестве возвращаемого значения нельзя, так как при завершении функции локальная переменная-строка, объявленная в ней, пропадет.

Строка, которую будем передавать в функцию, должна иметь размер не менее 9 (8 символов плюс конец строки). Каждый символ формируем отдельно: выделяем старший разряд часов, добавляем к нему код символа 0, при этом получаем код символа, стоящий в старшем разряде часов. То же повторяем для младшего разряда часов и далее аналогично для минут и секунд, не забывая разделители (:) и конец строки.

```
void CTime::GetTime(char* str) // читать время строкой в формате  
{                               // "12:05:23"  
    BYTE k = Hour / 10; // Старший разряд часов  
    str[0] = k + '0';   // Код символа  
    k = Hour % 10;      // Младший разряд часов  
    str[1] = k + '0';   // Код символа  
    str[2] = ':';       // Разделитель (:)  
    k = Min / 10;       // Старший разряд минут  
    str[3] = k + '0';   // Код символа  
    k = Min % 10;       // Младший разряд минут  
    str[4] = k + '0';   // Код символа  
    str[5] = ':';       // Разделитель (:)  
    k = Sec / 10;       // Старший разряд секунд  
    str[6] = k + '0';   // Код символа  
    k = Sec % 10;       // Младший разряд секунд  
    str[7] = k + '0';   // Код символа  
    str[8] = 0;         // конец строки  
}
```

Еще полезной будет функция, с помощью которой можно было бы выводить время на дисплей. Используем в ней уже написанный метод **GetTime**, позволяющий записывать время в строку, а потом выведем эту строку на дисплей.

```
void CTime::Print()  
{  
    char s[10]; // Массив для строки со временем  
    GetTime(s); // Читаем в s время из атрибутов объекта  
    cout << s;  // выводим строку на дисплей  
}
```

После доработки получаем класс:

```
class CTime
{
private:
    BYTE Hour;
    BYTE Min;
    BYTE Sec;
public:
    CTime() {Hour = 0; Min = 0; Sec = 0;} // Конструктор по умолчанию
    CTime(BYTE h, BYTE m, BYTE s) // Конструктор с тремя параметрами
        {Hour = 0; Min = 0; Sec = 0; SetTime(h, m, s);}
    CTime(const char* str) // Конструктор с параметром-строкой
    {
        Hour = 0; // Начальные значения на случай, если строка
        Min = 0; // окажется некорректной
        Sec = 0;
        SetTime(str); // Установка значений строкой
    }

    void SetHour(BYTE h) { if (h < 24) Hour = h; } // Установка часов
    void SetMin(BYTE m) { if (m < 60) Min = m; } // Установка минут
    void SetSec(BYTE s) { if (s < 60) Sec = s; } // Установка секунд
    void SetTime(BYTE h, BYTE m, BYTE s) // Установка час., мин., сек.
        {SetHour(h); SetMin(m); SetSec(s);}
    void SetTime(const char* str); // Установка времени строкой

    BYTE GetHour() {return Hour;} // Чтение часов
    BYTE GetMin() {return Min;} // Чтение минут
    BYTE GetSec() {return Sec;} // Чтение секунд
    void GetTime(char* str); // читать время строкой
    void Print(); // Вывод на дисплей
};

// ===== Методы класса =====
void CTime::SetTime(const char* str) // Установка времени строкой
{
    // преобразуем строку типа "12:34:56" в час, мин и сек
    BYTE h, m, s;
    if(str[2] == ':' && str[5] == ':') // Если симв. на своих местах
    {
        h = (str[0] - '0') * 10 + str[1] - '0';
        m = (str[3] - '0') * 10 + str[4] - '0';
        s = (str[6] - '0') * 10 + str[7] - '0';
        SetTime(h, m, s);
    }
}

void CTime::GetTime(char* str) // читать время строкой в формате
{
    // "12:05:23"
    BYTE k;

    k = Hour / 10; // Старший разряд часов
    str[0] = k + '0'; // Код символа
    k = Hour % 10; // Младший разряд часов
    str[1] = k + '0'; // Код символа
    str[2] = ':'; // Разделитель (:)
    k = Min / 10; // Старший разряд минут
```



```

    str[3] = k + '0'; // Код символа
    k = Min % 10; // Младший разряд минут
    str[4] = k + '0'; // Код символа
    str[5] = ':'; // Разделитель (:)
    k = Sec / 10; // Старший разряд секунд
    str[6] = k + '0'; // Код символа
    k = Sec % 10; // Младший разряд секунд
    str[7] = k + '0'; // Код символа
    str[8] = 0; // конец строки
}
void CTime::Print() // Вывод на дисплей
{
    char s[10];
    GetTime(s);
    cout << s;
}

```

Для тестирования класса создадим динамический массив объектов типа **CTime**, затем проверим методы, созданные в классе. Напишем программу с использованием меню, в котором будут пункты, позволяющие задавать размер массива и создавать его, заполнять массив случайными значениями или по порядку, изменять время отдельных элементов массива двумя способами, выводить массив на дисплей.

Напишем функцию, которая будет вызываться в меню, для того, чтобы задавать вручную время отдельных элементов массива (по выбору).

В ней сначала выведем возможные номера элементов, затем запросим номер элемента массива, который нужно задать, далее проверим, правильно ли введен номер элемента, и, если правильно, вводим час, минуту и секунду и задаем время элемента **A[i-1].SetTime(h, m, s);**. Не забываем, что индекс в массиве на 1 меньше номера элемента.

```

void FillArrayHand(CTime *A, int n) // В массиве A размером n
{
    // Задать час., мин., сек. отдельных элементов по выбору
    int h, m, s;
    int i;

    cout << "Установка времени отдельных элементов (1 -:- "
        << n << ")" << endl;
    for(;;)
    {
        cout << "Номер элемента (0 - выход): ";
        cin >> i;
        if (i == 0) return;
        if (i <= n)
        {
            cout << "Час = ";
            cin >> h;
            cout << "Минута = ";
            cin >> m;
            cout << "Секунда = ";
            cin >> s;
            A[i-1].SetTime(h, m, s);
            cout << "Задано время: ";
            A[i-1].Print();
            cout << endl;
        }
    }
}

```

Аналогично будет выглядеть функция, которая будет вызываться в меню, для того, чтобы задавать вручную время отдельных элементов массива (по выбору) строкой.

```
void FillArrayHandStr(CTime *A, int n) // В массиве A размером n
{
    // Задать время строкой для отдельных элементов по выбору
    char s1[10];
    int i;
    cout << "Установка времени отдельных элементов (1 -:- "
        << n << ")" << endl;
    for(;;)
    {
        cout << "Номер элемента (0 - выход): ";
        cin >> i;
        if (i == 0) return;
        if (i <= n)
        {
            cout << "Время (00:00:00) = ";
            cin >> s1;
            A[i-1].SetTime(s1);
            cout << "Задано время: ";
            A[i-1].Print();
            cout << endl;
        }
    }
}
```

В функции, которая заполняет массив случайными значениями, не забываем ограничить диапазон нужными значениями (для часов – 24, а для минут и секунд – 60).

```
void FillArrayRand(CTime *A, int n) // Заполнение массива
{
    // случайными значениями
    BYTE h, m, s;
    for(int i = 0; i < n; i++)
    {
        h = (float)rand() / RAND_MAX * 24;
        m = (float)rand() / RAND_MAX * 60;
        s = (float)rand() / RAND_MAX * 60;
        A[i].SetTime(h, m, s);
    }
}

void FillArrayStep(CTime *A, int n) // Заполнение массива
{
    // значениями с шагом
    BYTE h = 1, m = 2, s = 3;
    for(int i = 0; i < n; i++)
    {
        h += 1;
        m += 2;
        s += 3;
        A[i].SetTime(h % 24, m % 60, s % 60);
    }
}
```

Чтобы продемонстрировать методы класса, напомним две функции вывода данных массива на дисплей. В первой будем отображать время строкой, используя метод класса **Print()**, а во второй – в отдельных столбцах таблицы, используя методы **GetHour()**, **GetMin()** и **GetSec()**. Так как эти методы возвращают переменную типа

**BYTE**, то для правильного ее отображения целым числом необходимо предварительно преобразовать возвращаемое значение в тип **int**.

```
void ViewArray(CTime *A, int n) // Вывод на дисплей всего массива
{
    // строками типа "12:05:23"
    cout << "=====\n";
    cout << "| N |   Время   |\n";
    cout << "|=====|\n";
    for(int i = 0; i < n; i++)
    {
        cout << "| " << setw(2) << i + 1 << " | " << setw(8);
        A[i].Print();
        cout << " |" << endl;
    }
    cout << "=====\n";
}

void ViewArray2(CTime *A, int n) // Вывод на дисплей всего массива
{
    // таблицей с отдельными столбцами
    cout << "=====\n";
    cout << "|      |   Время      |\n";
    cout << "| N |=====|\n";
    cout << "|      | час | мин | сек |\n";
    cout << "|=====|\n";
    for(int i = 0; i < n; i++)
    {
        cout << "| " << setw(2) << i + 1
            << " | " << setw(2) << (int)A[i].GetHour()
            << " | " << setw(2) << (int)A[i].GetMin()
            << " | " << setw(2) << (int)A[i].GetSec()
            << " |" << endl;
    }
    cout << "=====\n";
}
```

Результат вывода с помощью **ViewArray()** и **ViewArray2()**:

N	Время
1	02:04:06
2	03:06:09
3	04:08:12
4	05:10:15
5	06:12:18
6	07:14:21
7	08:16:24
8	09:18:27
9	10:20:30
10	11:22:33

N	Время		
	час	мин	сек
1	2	4	6
2	3	6	9
3	4	8	12
4	5	10	15
5	6	12	18
6	7	14	21
7	8	16	24
8	9	18	27
9	10	20	30
10	11	22	33

Сама программа будет выглядеть так:

```
int main()
{
    CTime* Array;
    int Size = 10;
    char key;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    srand(time(0));

    // Демонстрация работы конструкторов
    CTime t1; // Конструктор по умолчанию
    CTime t2(12,34,56); // Конструктор
    CTime t3("23:59:59"); // Конструктор
    t1.Print();
    cout << endl;
    t2.Print();
    cout << endl;
    t3.Print();
    cout << endl;

    Array = new CTime[Size]; // Выделяем память под массив
    for (;;) // Бесконечный цикл
    {
        cout << "\n Выберите пункт меню:\n";
        cout << " 0 - Выйти из программы\n";
        cout << " 1 - Изменить размеры массива\n";
        cout << " 2 - Задать элементы массива вручную\n";
        cout << " 3 - Задать элементы массива вручную строкой\n";
        cout << " 4 - Задать массив случайными числами\n";
        cout << " 5 - Задать массив по возрастанию\n";
        cout << " 6 - Вывести массив (час:мин:сек)\n";
        cout << " 7 - Вывести массив таблицей\n";

        cin >> key;
        switch (key)
        {
            case '0':
                delete[] Array;
                return 0; // Завершение программы
            case '1':
                delete[] Array;
                cout << "Размер массива = ";
                cin >> Size;
                Array = new CTime[Size]; // Выделяем память под массив
                cout << "Массив создан\n";
                break;
            case '2':
                FillArrayHand(Array, Size);
                ViewArray(Array, Size);
                break;
            case '3':
                FillArrayHandStr(Array, Size);
                ViewArray(Array, Size);
                break;
```

```

    case '4':
        FillArrayRand(Array, Size);
        ViewArray(Array, Size);
        break;
    case '5':
        FillArrayStep(Array, Size);
        ViewArray(Array, Size);
        break;
    case '6':
        ViewArray(Array, Size);
        break;
    case '7':
        ViewArray2(Array, Size);
        break;
    }
}
return 0;
}

```

При запуске программы сразу создается массив объектов класса **CTime** размером **10** элементов. При этом все элементы создаются с помощью конструктора по умолчанию, поэтому они будут иметь значение **00:00:00**.

```

Выберите пункт меню:
0 - Выйти из программы
1 - Изменить размеры массива
2 - Задать элементы массива вручную
3 - Задать элементы массива вручную строкой
4 - Задать массив случайными числами
5 - Задать массив по возрастанию
6 - Вывести массив (час:мин:сек)
7 - Вывести массив таблицей
6
=====
| N |   Время   |
=====
| 1 | 00:00:00 |
| 2 | 00:00:00 |
| 3 | 00:00:00 |
| 4 | 00:00:00 |
| 5 | 00:00:00 |
| 6 | 00:00:00 |
| 7 | 00:00:00 |
| 8 | 00:00:00 |
| 9 | 00:00:00 |
|10 | 00:00:00 |
=====

```

После выбора пункта 5 меню, массив заполнится значениями с шагом. Если далее выбрать пункт 2 и изменить 5-ый элемент,

```

Установка времени отдельных элементов (1 -:- 10)
Номер элемента (0 - выход): 5
Час = 12
Минута = 00
Секунда = 00
Задано время: 12:00:00

```

а затем выбрать пункт меню 3 и изменить 6-ой элемент,

```
Установка времени отдельных элементов (1 :- 10)
Номер элемента (0 - выход): 6
Время (00:00:00) = 13:30:00
Задано время: 13:30:00
```

то массив будет отображаться так:

=====	
N	Время
=====	
1	02:04:06
2	03:06:09
3	04:08:12
4	05:10:15
5	12:00:00
6	13:30:00
7	08:16:24
8	09:18:27
9	10:20:30
10	11:22:33
=====	



## ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

в начале программы **ОБЯЗАТЕЛЬНО** выводить:  
ФИО, группа, номер лаб. работы, номер варианта.

### Задание

Разработать класс и программу, иллюстрирующую возможности данного класса. В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Необходимо продемонстрировать работу **со всеми методами и конструкторами класса**, иллюстрирующие возможности этих объектов. Вариант задания выдает преподаватель.

*Для получения максимального балла в программе необходимо сделать цикл для организации повторных вычислений и меню.*

### Варианты заданий

#### Вариант 1. Аквариум (Параллелепипед с жидкостью)

Разработать класс *Аквариум* и программу, иллюстрирующую возможности данного класса. Класс должен хранить геометрические размеры аквариума (не более 100 пикселей по каждой координате), уровень жидкости, налитый в него, и включать:

*методы*, позволяющие задавать и читать размеры аквариума и уровень налитой жидкости;

*методы*, позволяющие вычислять объем аквариума, объем налитой жидкости, процент заполненности аквариума.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры аквариумов, а также объемы и проценты заполненности.

#### Вариант 2. Колесо со спицами

Разработать класс *Колесо со спицами* и программу, иллюстрирующую возможности данного класса. Класс должен хранить радиус колеса (не более 100), толщину обода (не более половины радиуса), количество спиц (от 4 до 30) и включать:

*методы*, позволяющие задавать и читать параметры колеса;

*методы*, позволяющие вычислять общую площадь колеса, площадь обода, длину окружности колеса.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры колес, а также площади колеса, обода и длины окружностей.

#### Вариант 3. Равносторонний шестиугольник

Разработать класс *Равносторонний шестиугольник* и программу, иллюстрирующую возможности данного класса.

Класс должен хранить координаты (не более 100 пикселей по каждой координате) и размеры шестиугольника и включать:

*методы*, позволяющие задавать и читать размеры шестиугольникам;

*методы*, позволяющие вычислять площадь шестиугольника, длины сторон, периметр.

*метод*, позволяющий перемещать шестиугольник на величину (dx, dy).

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры шестиугольников, а также их площади и периметры.

#### **Вариант 4. Обыкновенная дробь**

Разработать класс *Обыкновенная дробь* и программу, иллюстрирующую возможности данного класса. Класс должен хранить величину числителя и знаменателя и включать:

*методы*, позволяющие задавать и читать значение числителя и знаменателя;  
*методы*, позволяющие задавать и считывать дробь строкой в формате " $\frac{3}{2}$ ";  
*методы*, позволяющие считывать дробь десятичным числом;  
*конструкторы* по умолчанию, и с параметрами.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функции, позволяющие складывать и вычитать две дроби, и продемонстрировать их работу. Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры дробей: числитель, знаменатель, в виде обыкновенной дроби и десятичной дроби.

#### **Вариант 5. Бочка (Цилиндр с жидкостью)**

Разработать класс *бочка* и программу, иллюстрирующую возможности данного класса.

Класс должен хранить геометрические размеры бочки (не более 100 пикселей по каждой координате), уровень жидкости, налитый в нее, и включать:

*методы*, позволяющие задавать и читать размеры бочки и уровень налитой жидкости;

*методы*, позволяющие вычислять объем бочки, объем налитой жидкости, процент заполненности бочки.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры аквариумов, а также объемы и проценты заполненности.

#### **Вариант 6. Лампочка**

Разработать класс *Лампочка* и программу, иллюстрирующую возможности данного класса. Класс должен хранить параметры лампочки: номинальную мощность (не более 500 Вт), номинальное напряжение (не более 1000 В) и включать:

*методы*, позволяющие задавать и читать номинальную мощность, номинальное напряжение;

*методы*, позволяющие вычислять ток и рассеиваемую мощность при рабочем (реально приложенном) напряжении, при превышении номинальной мощности на 10% лампочка перегорает.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры лампочек, а также токи и рассеиваемые мощности при заданном рабочем напряжении.

#### **Вариант 7. Пирамида**

Разработать класс *Пирамида* и программу, иллюстрирующую возможности данного класса.

Класс должен хранить размеры основания (не более 400 пикселей) и высоту пирамиды (не более 400 пикселей) и включать:

*методы*, позволяющие задавать и читать размеры основания и высоту;

*методы*, позволяющие вычислять объем пирамиды, площадь основания, боковой стороны и общую площадь.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры пирамид, а также их объемы, площади основания, боковой стороны и общей площади.

### Вариант 8. Комплексное число

Разработать класс *Комплексное число* и программу, иллюстрирующую возможности данного класса. Класс должен хранить реальную и мнимую части числа (тип float) и включать методы, позволяющие:

- задавать и считывать комплексное число строкой в формате “ $1.3 + i * 12.6$ ”;
- задавать и считывать отдельно реальную и мнимую части;
- вычислять модуль и фазу.

Все проверки корректности должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры комплексных чисел, а также их модули и фазы.

### Вариант 9. Треугольник

Разработать класс *Треугольник* и программу, иллюстрирующую возможности данного класса.

Класс должен хранить координаты вершин треугольника (не более 1000 пикселей по каждой координате) и включать:

- методы*, позволяющие задавать и читать координаты вершины (с индексом i);
- методы*, позволяющие вычислять длины сторон треугольника и периметр.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры треугольников, а также длины их сторон и периметры.

### Вариант 10. Цилиндр

Разработать класс *Цилиндр* и программу, иллюстрирующую возможности данного класса. Класс должен хранить радиус основания (не более 200 пикселей) и высоту цилиндра (не более 400 пикселей) и включать:

- методы*, позволяющие задавать и читать радиус основания и высоту;
- методы*, позволяющие вычислять объем цилиндра, площадь основания, боковой стороны и общую площадь.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры цилиндров, а также их объемы, площади основания, боковой стороны и общей площади.

### Вариант 11. Произвольный четырехугольник

Разработать класс *Четырехугольник* и программу, иллюстрирующую возможности данного класса. Класс должен хранить координаты вершин четырехугольника (не более 99 пикселей по каждой координате) и включать:

- *методы*, позволяющие задавать и читать координаты вершины (с индексом i);
- методы*, позволяющие вычислять длины сторон и периметр четырехугольника.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры четырехугольников, а также их длины сторон и периметры.

### Вариант 12. Трапеция

Разработать класс *Трапеция* и программу, иллюстрирующую возможности данного класса. Класс должен хранить координаты (не более 200 пикселей по каждой координате) и размеры трапеции и включать:

- *методы*, позволяющие задавать и читать координаты и размеры трапеции;
- *методы*, позволяющие вычислять площадь трапеции, длины сторон, периметр.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры трапеций, а также их площади и длины сторон.

### **Вариант 13. Ромб**

Разработать класс *Ромб* и программу, иллюстрирующую возможности данного класса. Класс должен хранить координаты (не более 250 пикселей по каждой координате) и размеры ромба и включать:

*методы*, позволяющие задавать и читать координаты и размеры ромба;

*методы*, позволяющие вычислять площадь ромба, длины сторон, периметр.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры ромбов, а также их площади, длины сторон и периметры.

### **Вариант 14. Параллелограмм**

Разработать класс *Параллелограмм* и программу, иллюстрирующую возможности данного класса.

Класс должен хранить координаты (не более 500 пикселей по каждой координате) и размеры параллелограмма и включать:

*методы*, позволяющие задавать и читать координаты и размеры параллелограмма;

*методы*, позволяющие вычислять площадь параллелограмма, длины сторон, периметр.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры параллелограммов, а также их площади, длины сторон и периметры.

### **Вариант 15. Конус**

Разработать класс *Конус* и программу, иллюстрирующую возможности данного класса. Класс должен хранить радиус основания (не более 300 пикселей) и высоту конуса (не более 500 пикселей) и включать:

*методы*, позволяющие задавать и читать радиус основания и высоту;

*методы*, позволяющие вычислять объем конуса, площадь основания, боковой стороны и общую площадь.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры конусов, а также их объемы, площади основания, боковой стороны и общие площади.

### **Вариант 16. Резистор**

Разработать класс *Резистор* и программу, иллюстрирующую возможности данного класса. Класс должен хранить параметры резистора: сопротивление, макс. рассеиваемую мощность (не более 100 Вт) и включать:

*методы*, позволяющие задавать и читать сопротивление, макс. рассеиваемую мощность;

*методы*, позволяющие вычислять ток и рассеиваемую мощность при рабочем (реально приложенном) напряжении, при превышении макс. мощности на 10% резистор перегорает.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры резисторов, а также токи и рассеиваемые мощности при заданном рабочем напряжении.

### Вариант 17. Равносторонний пятиугольник

Разработать класс *Равносторонний пятиугольник* и программу, иллюстрирующую возможности данного класса.

Класс должен хранить координаты (не более 200 пикселей по каждой координате) и размеры пятиугольника и включать:

*методы*, позволяющие задавать и читать размеры пятиугольникам;

*методы*, позволяющие вычислять площадь пятиугольника, длины сторон, периметр.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры пятиугольников, а также их площади и периметры.

### Вариант 18. Трехмерный вектор

Разработать класс *Трехмерный вектор* и программу, иллюстрирующую возможности данного класса. Класс должен хранить компоненты вектора и включать методы, позволяющие:

– задавать и читать отдельные компоненты вектора (как числа типа float);

– задавать и читать компоненты вектора строкой в формате “**13**; **-23.5**; **0.8**”;

– вычислять модуль.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры векторов, а также их модули.

### Вариант 19. Логический элемент И

Разработать класс *Логический элемент И* и программу, иллюстрирующую возможности данного класса. Класс должен хранить количество входов логического элемента (2, 3, 4 или 8 входов), значения входных сигналов и включать:

– *методы*, позволяющие задавать и читать количество входов логического элемента;

– *методы*, позволяющие задавать значение входного сигнала на i-том входе (0 или 1);

– *методы*, позволяющие задавать значения всех входных сигналов строкой (“1001...”);

– *методы*, позволяющие вычислять сигнал на выходе логического элемента в зависимости от сигналов на входах.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры элементов, а также сигналы на выходах элементов.

### Вариант 20. Логический элемент ИЛИ

Разработать класс *Логический элемент ИЛИ* и программу, иллюстрирующую возможности данного класса. Класс должен хранить количество входов логического элемента (2, 3, 4 или 8 входов), значения входных сигналов и включать:

*методы*, позволяющие задавать и читать количество входов логического элемента;

– *методы*, позволяющие задавать значение входного сигнала на i-том входе (0 или 1);

*методы*, позволяющие задавать значения всех входных сигналов строкой (“1001...”);

*методы*, позволяющие вычислять сигнал на выходе логического элемента в зависимости от сигналов на входах.

Все проверки корректности ввода должны проводиться в методах класса.

Написать функцию, позволяющую выводить массив на дисплей в виде таблицы, в которой отображать параметры элементов, а также сигналы на выходах элементов.