

## ЛАБОРАТОРНАЯ РАБОТА № 25 ПЕРЕГРУЗКА ОПЕРАЦИЙ

### ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Одним из наиболее мощных средств языка C++ является возможность перегрузки операций. С ее помощью можно сделать программу более понятной и логичной, например, если в классе **CTime** перегрузить операцию «-», то вместо малопонятной строки

```
int dT = Time1.SecBetween(Time2); // Разность двух времен в сек.
```

можно будет написать

```
int dT = Time1 - Time2;
```

Чтобы перегрузить операцию, необходимо написать функцию и дать ей **специальное имя**, которое должно состоять из ключевого слова **operator**, после которого записывается **обозначение перегружаемой операции**. Например, имя функции **operator+** можно использовать для перегрузки операции сложения.

Перегружать операции можно только тогда, когда хотя бы один из операндов является объектом класса, созданного или используемого пользователем, указателем или ссылкой на него. **Перегружать операции только для стандартных типов нельзя.**

Перегружать можно любые операции, кроме **., .\*, ?:, ::, #, sizeof**. Обозначения собственных операций вводить нельзя.

Составить функцию с перегруженной операцией можно тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс.

Все операции делятся по числу операндов на унарные (**++, --, унарный -, !**, и др.) и бинарные (**+, бинарный -, >, ==** и др.). В зависимости от этого, а также от того, является ли перегружаемая операция членом класса или нет, она будет иметь разное число аргументов.

### Перегрузка бинарных операций

Бинарные операции имеют по два операнда, т.е. перегрузить операцию можно тогда, когда хотя бы один из операндов имеет нестандартный тип. Перегруженную функцию-оператор можно сделать членом класса (тогда у нее будет один аргумент) или дружественной функцией (тогда у нее будет два аргумента).

Рассмотрим перегрузку операций на примере класса **MPoint**, предназначенного для описания объектов-точек на плоскости. Класс должен хранить координаты точки (**x** и **y**) и методы для задания, модификации и считывания координат. Кроме того, хорошо было бы иметь функции, позволяющие работать с точками, например, складывать две точки, определять расстояние между ними, сравнивать на совпадение, сдвигать точки и т.д.

```
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;}
float BegDistance() {return sqrt((float)X*X + Y*Y);}
};

```

Если мы хотим складывать точки, то можно написать глобальную функцию, в которую передавать в качестве параметров два объекта типа **MPoint**, и возвращать тоже объект этого класса.

```

MPoint PSumm(MPoint a, MPoint b)
{
    MPoint tmp; // объявляем временный объект
    tmp.SetX(a.GetX() + b.GetX()); // складываем координ. X точек a и b
    tmp.SetY(a.GetY() + b.GetY()); // складываем координ. Y точек a и b
    return tmp; // возвращаем этот объект
}

```

Теперь можно найти сумму точек:

```

int main()
{
    . . . . .
    MPoint p1, p2, p3;

    p1.SetXY(10, 20);
    p2.SetXY(15, 35);
    p3 = PSumm(p1, p2); // у точки p3 координаты стали: 25, 55
    . . . . .
}

```

Чтобы вместо последней строки можно было написать

```
p3 = p1 + p2; // у точки p3 координаты стали: 25, 55
```

нужно перегрузить **операцию +**. Для этого у функции **PSumm** нужно просто поменять название на **operator+**.

```

MPoint operator+(MPoint a, MPoint b)
{
    MPoint tmp; // объявляем временный объект
    tmp.SetX(a.GetX() + b.GetX()); // складываем координ. X точек a и b
    tmp.SetY(a.GetY() + b.GetY()); // складываем координ. Y точек a и b
    return tmp; // возвращаем этот объект
}

```

Теперь эту функцию можно вызывать двумя способами:

```
p3 = operator+(p1, p2); // Через имя функции - «operator+»
или   p3 = p1 + p2;      // С помощью знака операции - «+»
```

Второй вариант записи гораздо понятнее, правда сама написанная нами функция выглядит немного сложно, т.к. в обычной глобальной функции запрещен доступ к закрытым членам класса. Но функцию можно упростить, если **сделать ее дружественной** классу **MPoint**. Для этого нужно вставить в класс ее объявление (**только заголовок**) с ключевым словом **friend**.

```

class MPoint
{
    int X, Y;
public:
    . . . . .
    friend MPoint operator+(MPoint a, MPoint b);
    . . . . .
}

```

Этим самым мы разрешаем функции **operator+** доступ к закрытым членам класса. Теперь ее можно упростить:

```
MPoint operator+(MPoint a, MPoint b) // Это глобальная функция
{
    // НЕ член класса MPoint, но
    MPoint tmp;           // она дружелюбна ему,
    tmp.X = a.X + b.X;    // поэтому в ней разрешен доступ
    tmp.Y = a.Y + b.Y;    // к закрытым членам класса
    return tmp;
}
```

Вызываться функция будет так же:

```
p3 = p1 + p2;
```

Аналогично можно перегрузить и другие бинарные операции. При создании функции-оператора важно правильно определить количество и тип аргументов, а также тип возвращаемого значения. Для этого нужно понять, для чего и как эта операция будет использоваться. Проще всего сначала создать обычную функцию, а затем изменить ее имя на символ соответствующей операции.

Перегрузим операцию «—» так, чтобы можно было с ее помощью определять расстояние между двумя точками. Сделаем эту функцию глобальной и дружелюбной, следовательно, она должна иметь два аргумента, а возвращаемое значение пусть будет типа **float**.

```
float operator-(MPoint a, MPoint b) //
{
    float f;
    int ix = a.X - b.X;
    int iy = a.Y - b.Y;

    f = sqrt((float)ix * ix + iy * iy); // Расстояние между точками
    return f;
}
```

Так же просто можно перегрузить операцию, которая сравнивает точки на совпадение, или операцию, которая сравнивает точки по расстоянию от начала координат:

```
bool operator==(MPoint a, MPoint b) // Совпадают ли точки
{
    return (a.X == b.X && a.Y == b.Y);
}
bool operator>(MPoint a, MPoint b) // а дальше от начала координат?
{
    return (a.BegDistance() > b.BegDistance());
}
```

Не забываем объявить их дружелюбными:

```
class MPoint
{
    . . . . .
public:
    . . .
    friend MPoint operator+(MPoint a, MPoint b);
    friend float operator-(MPoint a, MPoint b);
    friend bool operator==(MPoint a, MPoint b);
    friend bool operator>(MPoint a, MPoint b);
    . . . . .
}
```

Использовать перегруженные операции можно так:

```
int main()
```

```

{
    . . . . .
    MPoint p1, p2;
    float f;

    p1.SetXY(10, 20);
    p2.SetXY(15, 35);
    f = a - b; // Расстояние между точками
    cout << "Расстояние между точками = " << f << endl;
    if (p1 == p2) // Если точки совпадают
    {
        cout << "Точки совпадают" << endl;
    }
    if (p1 > p2) // Если p1 дальше от начала координат
    {
        cout << " p1 дальше от начала координат" << endl;
    }
    . . . . .
}

```

Можно перегружать бинарные операции и по-другому: сделать их членом класса. В этом случае не надо объявлять дружественные функции, нужно просто добавить в класс соответствующие методы. Например, перегрузим операцию сложения, для этого сначала создадим метод класса с именем **Summ**, а потом поменяем его.

```

class MPoint
{
    . . . . .
public:
    . . . . .
    MPoint Summ(MPoint b) // Сложение точек
    {
        MPoint tmp; // Временная переменная
        tmp.X = X + b.X; // складываем координ. X текущей точки и точки b
        tmp.Y = Y + b.Y; // складываем координ. Y текущей точки и точки b
        return tmp;
    }
}

```

Теперь можно найти сумму точек:

```

int main()
{
    . . . . .
    MPoint p1, p2, p3;

    p1.SetXY(10, 20);
    p2.SetXY(15, 35);
    p3 = p1.Sum(p2); // Координаты точки p1 складываем с
                     // координатами точки p2 и результат присваиваем p3.
                     // у точки p3 координаты стали: 25, 55
                     // p1 и p2 не изменились
    . . . . .
}

```

Здесь метод класса (**Summ**) вызывается для объекта **p1**, а вторая точка (**p2**) передается в качестве аргумента. Чтобы создать из нее функцию-оператор, нужно заменить имя **Summ** на **operator+**.

```

class MPoint
{
    . . . . .
public:
    . . . . .
    MPoint operator+(MPoint b) // Сложение точек
    {
        MPoint tmp;          // Временная переменная
        tmp.X = X + b.X; // складываем координ. X текущей точки и точки b
        tmp.Y = Y + b.Y; // складываем координ. Y текущей точки и точки b
        return tmp;
    }
}

```

Чтобы сложить две точки можно явно вызвать функцию **operator+** :

```

int main()
{
    MPoint p1, p2, p3;
    p1.SetXY(10, 20);
    p2.SetXY(15, 35);
    p3 = p1.operator+(p2); // у точки p3 координаты стали: 25, 55
                          // или более красивый и понятный вариант:
    p3 = p1 + p2; // у точки p3 координаты стали: 25, 55
}

```

Таким образом, перегруженную бинарную операцию можно сделать членом класса (как **Summ**) или дружественной функцией (как **PSumm**). В зависимости от этого у нее будет разное число аргументов (как у функций **Summ** и **PSumm** в вышеописанных примерах). Чтобы создать функцию-оператор, нужно заменить имя функции в выбранном варианте на **operator+**.

Разница в объявлении и вызове перегруженной бинарной операции:

	Вариант 1	Вариант 2
	Оператор-функция – член класса	Оператор-функция – глобальная дружественная
Определение	<pre> MPoint operator+(MPoint b) {     MPoint tmp;     tmp.X = X + b.X;     tmp.Y = Y + b.Y;     return tmp; } </pre>	<pre> MPoint operator+(MPoint a, MPoint b) {     MPoint tmp;     tmp.X = a.X + b.X;     tmp.Y = a.Y + b.Y;     return tmp; } </pre>
Вызов	<p><code>p3 = p1.operator+(p2);</code></p> <p>или</p> <p><code>p3 = p1 + p2;</code></p> 	<p><code>p3 = operator+(p1, p2);</code></p> <p>или</p> <p><code>p3 = p1 + p2;</code></p> 

Вызов с помощью знака операции в обоих случаях выглядит совершенно одинаково.

Во всех предыдущих примерах значение операндов не менялось. Но в языке C++ есть операции, в которых изменяется и сам операнд, например **+=**, **-=**, **\*=**, **/=** и т.д.

Перегрузим операцию **+=** как метод класса. В отличие от операции **+**, где значения операндов не менялись, в операции **+=** к первому операнду добавляется значение второго операнда, например, так: `p1 += p2`. Если использовать обычный метод класса, то вызов его будет следующим образом: `p1.Add(p2)`, т.е. у функции должен быть один аргумент типа **MPoint**. Результат операции ничему не присваивается,

следовательно, возвращаемого значения нет. Соответственно функция будет выглядеть так:

```
class MPoint
{
    . . . . .
public:
    . . .
    void Add(MPoint b)
    {
        x += b.x;
        y += b.y;
    }
    . . . . .
}
```

Заменяем имя функции `Add` на `operator+=` и получаем перегруженную операцию:

```
class MPoint
{
    . . . . .
public:
    . . .
    void operator+=(MPoint b)
    {
        x += b.x;
        y += b.y;
    }
}
```

Использовать перегруженную операцию можно так:

```
int main()
{
    MPoint p1, p2;
    p1.SetXY(10, 20);
    p2.SetXY(15, 35);
    p1 += p2; // у точки p1 координаты стали: 25, 55
}
```

Если использовать дружественную функцию, то ее вызов будет выглядеть по-другому: `Add1(p1, p2)`, т.е. у этой функции должно быть два аргумента типа `MPoint`. Результат ничему не присваивается, следовательно, возвращаемого значения нет.

Но **первый операнд должен измениться**, следовательно, в функцию нужно передавать аргумент не по значению, а **по ссылке** (`MPoint&`).

Соответственно функция будет выглядеть так:

```
class MPoint
{
    . . . . .
public:
    . . .
    friend void Add1(MPoint& a, MPoint& b);
}
void Add1(MPoint& a, MPoint& b)
{
    a.x += b.x;
    a.y += b.y;
}
```

Заменяем имя функции `Add1` на `operator+=` и получаем перегруженную операцию:

```
class MPoint
{
    . . . . .
public:
    . . .
    friend void operator+=(MPoint& a, MPoint& b);
}
void operator+=(MPoint& a, MPoint& b)
{
    a.x += b.x;
    a.y += b.y;
}
```

Используется эта перегруженная операция аналогично.

У бинарной операции один из операндов может быть стандартного типа, например, для умножения координат точки на некоторое число (масштабирование). Для этого перегрузим операцию `*=`.

```
class MPoint
{
    . . . . .
public:
    . . .
    void operator*=(int k)
    {
        x *= k;
        y *= k;
    }
    . . . . .
}
int main()
{
    . . . . .
    MPoint p1;

    p1.SetXY(3, 5);
    p1 *= 10; // у точки p1 координаты стали: 30, 50
    . . . . .
}
```

Одну и ту же операцию можно перегружать несколько раз, но эти перегруженные функции должны отличаться аргументами:

```
class MPoint
{
    . . . . .
public:
    . . .
    friend MPoint operator+(MPoint a, MPoint b);
    friend MPoint operator+(MPoint a, int k);
    . . . . .
}
MPoint operator+(MPoint a, int k)
{
    MPoint tmp;
    tmp.X = a.X + k;
    tmp.Y = a.Y + k;
```

```

    return tmp;
}

Можно добавить вторую функцию, чтобы обеспечить коммутативность сложения:
MPoint operator+(int k, MPoint a)
{
    return a + k;
}

int main()
{
    . . . . .
    MPoint p1, p2;

    p1.SetXY(3, 5);
    p2 = p1 + 10; // у точки p2 координаты стали: 13, 15
    . . . . .
    p2 = 100 + p1; // у точки p2 координаты стали: 113, 115
    . . . . .
}

```

### Перегрузка унарных операций

Унарные операции имеют по одному операнду, следовательно, для того, чтобы перегрузить операцию, ее **операнд** должен быть **нестандартного типа**. Перегруженную оператор–функцию можно сделать членом класса (тогда у нее не будет аргументов) или дружественной функцией (тогда у нее будет один аргумент).

Перегрузим операцию *унарный минус*. В отличие от операции *бинарный минус*, в которой вычисляется разность двух чисел, операция *унарный минус* должна менять знак числа и присваивать это значение другой переменной, например, **a = -b**. Если **b** было равно **3**, то **a** станет равным **-3**, а значение операнда **b** не изменится. Аналогично должна вести себя и подобная операция для класса **MPoint**. Перегрузим для примера эту операцию как член класса **MPoint**:

```

class MPoint
{
    . . . . .
    MPoint operator-();           // унарный минус
};

MPoint MPoint::operator-() // функция – член класса MPoint
{
    // унарный минус
    MPoint tmp;
    tmp.X = -X;
    tmp.Y = -Y;
    return tmp;
}

int main()
{
    . . . . .
    MPoint p1(3, 5), p2;
    p2 = -p1; // у точки p2 координаты стали: -3, -5
    . . . . .
}

```

Свои особенности имеет перегрузка операций **++** и **--**. Как известно, эти операции имеют две формы: префиксную и постфиксную (**++x** и **x++**). Чтобы их различить при



перегрузке соответствующих операций для **постфиксной формы** вводят дополнительный аргумент, который не используется, но по нему компилятор отличает одну форму от другой. Для функций-операторов, создаваемых как члены класса, эти две формы будут определяться так:

```
operator++() // префиксная форма
```

и

```
operator++(int NotUsed) // постфиксная форма,
```

а для дружественных функций так:

```
operator++(MPoint& c) // префиксная форма
```

и

```
operator++(MPoint& c, int NotUsed) // постфиксная форма.
```

Перегрузим операцию **++** для класса **MPoint** так, чтобы префиксная форма увеличивала на **1** координату **X**, а постфиксная – координату **Y**. Но прежде необходимо разобраться с типом возвращаемого значения. Здесь все зависит от того, как мы собираемся эту операцию использовать. Если только так:

```
x++;
```

```
++y;
```

то возвращаемое значение не используется, а если так:

```
x = y++;
```

```
z = ++y;
```

то возвращаемое значение присваивается переменной типа **MPoint**, следовательно, возвращаемое значение должно быть этого типа (или ссылкой на него).

```
class MPoint
```

```
{
```

```
    . . .
```

```
    MPoint& operator++(); // префиксный
```

```
    MPoint& operator++(int NotUsed); // постфиксный
```

```
};
```

```
MPoint& MPoint::operator++() // функция – член класса MPoint
```

```
{ // префиксный
```

```
    X++;
```

```
    return *this; // Возвращаем ссылку на самого себя
```

```
}
```

```
MPoint& MPoint::operator++(int NotUsed) // функция – член класса  
// MPoint – постфиксный
```

```
    Y++;
```

```
    return *this; // Возвращаем ссылку на самого себя
```

```
}
```

Теперь можно использовать эти операции:

```
int main()
```

```
{
```

```
    . . . . .
```

```
    MPoint p1(1,5), p2, p3;
```

```
    ++p1; // p1 стало равным 2, 5
```

```
    p1++; // p1 стало равным 2,6
```

```
// или так:
```

```
    p2 = ++p1; // p1 стало равным 3,6; p2 стало равным 3,6
```

```
    p3 = p1++; // p1 стало равным 3,7; p3 стало равным 3,7
```

```
    . . . . .
```

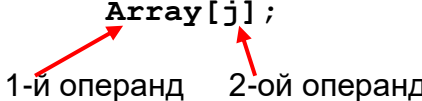
```
}
```

Перегруженные операции **++** (**--**), и префиксная, и постфиксная, всегда выполняются **до операции присваивания**, в отличие от операций **++** (**--**) для стандартных типов!

## Перегрузка операции индексирования

Операция индексирования `[ ]` обычно перегружается, когда класс содержит множество каких-либо элементов, для которых индексирование имеет смысл. Операция индексирования должна возвращать ссылку на элемент, содержащийся в множестве. Например, в классе **Строка** содержится массив символов, и индексирование поможет обращаться к конкретному символу строки.

Операция индексирования – бинарная, ее первый операнд – сам объект, к которому применяется операция индексирования, а второй операнд – индекс:

`Array[j];`  
  
1-й операнд    2-ой операнд

Операцию индексирования проще делать членом класса. Обычно эту операцию нужно использовать как слева, так и справа от знака присваивания. Индекс может быть целого типа, тогда применение операции выглядит так:

```
char c = s[j];  
s[i] = 'w';
```

где `s` – это **объект класса Строка**. В первом случае операция `[ ]` возвращает *i*-тый символ, хранящийся в строке, т.е. тип возвращаемого значения операции должен быть **char** (тип хранимых данных). Но чтобы использовать операцию `[ ]` слева от знака присваивания, возвращать значение нужно обязательно **по ссылке**, т.е. тип возвращаемого значения для класса **Строка** должен быть ссылкой на **char**.

Для класса строка перегрузка операции индексирования выглядит так:

```
class MString // Класс "Строка"  
{  
private:  
    int len;    // Длина строки  
    char* Str;  // Указатель на массив, где будет храниться строка  
public:  
    MString() {Str = 0; len = 0;} // Конструктор без параметров  
    MString(const char* s);      // Конструктор с параметром строкой  
    ~MString();                  // Деструктор  
    . . . . .  
    char& operator[](int i); // Возвращает ссылку на i-тый символ в Str  
};
```

В перегружаемой операции `[ ]` проверяем, не выходит ли индекс за пределы массива, и возвращаем *i*-тый элемент массива. Что делать в случае, если индекс вышел за пределы, решается в зависимости от задачи и требований. Лучше всего вызывать исключительную ситуацию, хотя можно просто возвращать ссылку на первый (или последний) элемент.

```
char& MString::operator[](int i)  
{  
    if ((i>=0) && (i<Len))  
        return Str[i];  
    throw 5; // вызываем исключительную ситуацию  
}
```

Чтобы не проверять индекс на отрицательность, лучше его тип сделать **unsigned int**.

```
char& MString::operator[](unsigned int i)  
{  
    if (i<Len)  
        return Str[i];  
    throw 5; // вызываем исключительную ситуацию  
}
```

В классе Треугольник тоже может понадобиться операция индексирования, для того, чтобы с ее помощью обращаться к вершинам треугольника. Объявим в классе массив из объектов класса MPoint, в котором будем хранить данные вершин треугольника.

```
class MTriangle
{
private:
    MPoint Angles[3]; // Массив из трех точек - вершин треугольника
public:
    . . . . .
    void SetAngle(unsigned int i, MPoint p) // Задаем i-тую вершину
    {
        if (i < 3) Angles[i] = p;
    }
    MPoint GetAngle(unsigned int i) // Возвращаем i-тую вершину
    {
        if (i < 3) return Angles[i];
        return Angles[0];
    }
    . . . . .
};
```

Функция **SetAngle()** задает вершину треугольника точкой (объектом класса **MPoint**), в ней проверяем индекс на корректность и копируем поля точки в **i**-тый элемент массива. Функция **GetAngle()** возвращает **i**-тую вершину треугольника, если индекс корректный, а если нет – вершину с индексом **0**. (хотя **0**-вая вершина ничем не лучше других). Лучше всего вызывать исключительную ситуацию.

Возможность перегрузки операции индексирования позволит вместо этих двух функций написать одну:

```
class MTriangle
{
private:
    MPoint Angles[3]; // Массив из трех точек - вершин треугольника
public:
    . . . . .
    MPoint& operator[] (unsigned int i) // Возвращаем ссылку
    {                                     // на i-тую вершину
        if (i < 3) return Angles[i];
        return Angles[0];
    }
    . . . . .
};
```

Теперь можно задавать и читать данные вершин треугольника так:

```
int main()
{
    MPoint p1(1,5), p2(5, 5), p3(5, 1);
    MTriangle t1;

    t1[0] = p1; // Задаем вершины значениями точек
    t1[1] = p2;
    t1[2] = p3;
    for(unsigned int i=0; i<3; i++)
    {
        p1 = t1[i]; // Читаем i-тую вершину
        cout << p1.GetX() << ", " << p1.GetY() << endl;
    }
}
```

```

for(unsigned int i=0; i<3; i++)
    t1[i] = MPoint(i, i*i); // Задаем вершины с помощью конструктора
for(unsigned int i=0; i<3; i++)
    cout << t1[i].GetX() << ", " << t1[i].GetY() << endl;
}

```

Операция `[]` возвращает ссылку на объект класса **MPoint**, поэтому для него можно вызвать метод, например **GetX()**:

`t1[i].GetX()` // возвращает координату X i-той вершины треугольника

При перегрузке операции `[]` в качестве индекса не обязательно должна выступать переменная целого типа, например чтобы построить класс-массив с именованными строками, в качестве индекса можно использовать тип **char\*** или класс **Строка**. Это позволит обращаться к элементам массива по именам.

### Перегрузка операции присваивания

Операция присваивания есть у каждого класса и по умолчанию она осуществляет побайтное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. **Если класс содержит элементы, под которые динамически выделяется память, операцию присваивания необходимо перегрузить** так, чтобы копирование проходило корректно. Чтобы правильно определить аргумент и тип возвращаемого значения, рассмотрим, как используется операция, на примере класса *Строка*:

```
s1 = s2;
```

Операция бинарная, левый операнд – сам объект, в который копируются значения из другого объекта. Поэтому в качестве аргумента должен выступать объект такого же типа или ссылка на него. Возвращаемого значения нет, поэтому можно объявить операцию присваивания так:

```
void operator=(MString& s);
```

Но, чтобы можно было использовать цепочку операторов присваивания, как принято в языке C++:

```
s1 = s2 = s3 = s4;
```

необходимо возвращать значение объекта, куда проводилось копирование, а лучше – ссылку на него. В качестве аргумента лучше передавать константную ссылку

```

class MString // Класс "Строка"
{
    . . . . .
    MString& operator=(const MString& s); // Операция присваивания
};

```

Саму функцию можно определить так:

```

MString& MString::operator=(const MString& s)
{
    if (this == &s) return *this; // Если копирование самого в себя -
                                // не копируем
    if (Len != s.Len) // Если длина другая
    {
        Len = s.Len; // Устанавливаем новую длину строки
        if (Str) // Если память уже выделялась
        {
            delete[] Str; // освобождаем память
            Str = 0;
        }
        Str = new char[Len + 1]; // Выделяем новую
    }
    for (int i=0; i<Len; i++) // Копируем символы

```

```

        Str[i] = s.Str[i];
    return *this; // Возвращаем ссылку на сам объект
}

```

Операцию присваивания можно определять только как метод класса. Она не наследуется.

### Перегрузка операции преобразования типа

Операция преобразования (называемая также операцией приведения) используется для преобразования объекта одного класса в объект другого класса или в переменную стандартного типа. Формат операции:

```
operator type();
```

где **type** – имя нового типа. При перегрузке операции приведения не указывают тип возвращаемого значения, т.к. это и есть тот тип, к которому должен быть преобразован объект. Перегружаемая операция приведения типа может быть использована для преобразования объектов определенных пользователем типов в переменные стандартных типов или в объекты других определенных пользователем типов, например для класса *Строка* можно создать операцию преобразования в тип *char\** или *const char\**:

```

class MString // Класс "Строка"
{
    . . . . .
    operator const char*() {return Str;} // Операция преобразования
                                        // в const char*
};

```

Теперь в тех местах, где требуется тип **char\*** можно подставлять и объекты класса **MString**. При этом объект этого типа будет автоматически преобразовываться в тип **char\***:

```

MString ms1("ABCD");
char s1[50];
strcpy(s1, ms1); // ms1 неявно преобразуется в const char*

```

Можно перегрузить операции преобразования в тип **int** и **float**:

```

class MString // Класс "Строка"
{
    . . . . .
    operator int();           // Операция преобразования в int
    operator float();        // Операция преобразования в float
};

```

Тогда в тексте программы можно будет писать так:

```

MString s1("ABCD");
int i = s2; // строка преобразуется в целое число

```

Обратные преобразования (из других типов в тип **MString**) можно делать, определяя соответствующие конструкторы в классе **MString**.

```

class MString // Класс "Строка"
{
    . . . . .
    MString() {Str = 0; Len = 0;} // Конструктор без параметров
    MString(const char* s);       // Конструктор с параметром
    строкой
    MString(const MString& s);     // Конструктор копий
    MString(const char c);        // Конструктор с параметром const
    char
    MString(int i);               // Конструктор с параметром int
    MString(float f);            // Конструктор с параметром float
    ~MString();                  // Деструктор
    . . . . .
}

```

```
};
```

Тогда в тексте программы можно будет писать так:

```
MString s1;  
s1 = MString("x = ") + MString(12); // Преобразование char* в  
                                     // MString целого числа в MString и слияние строк  
printf("%s", s1); // будет выведен текст "x = 12"  
MString s2("y = ");  
int i = 125;  
s2 += i; // Преобразование целого числа в MString  
printf("%s", s1); // будет выведен текст "y = 125"
```

## Нетрадиционная перегрузка операций

При перегрузке операций обычно стараются сохранить их смысл, хотя можно, например, перегрузить операцию **+** так, чтобы она выполняла не сложение, а вычитание объектов класса. Однако смысл такой перегрузки весьма сомнителен. Тем не менее, в некоторых случаях перегруженные операции выполняют совсем нетрадиционные действия.

Примером такой перегрузки являются операции сдвига (**>>** и **<<**) широко используемые для ввода/вывода информации. Операции, позволяющие работать с потоками ввода-вывода, определены в заголовочном файле **iostream**, который включает в себя классы **istream** и **ostream**, а также прототипы функций **operator>>()** и **operator<<()**. Кроме того, здесь же определены объекты **cin** и **cout**.

Объект **cin** соответствует стандартному потоку ввода. По умолчанию этот поток ассоциируется со стандартным устройством ввода — обычно клавиатурой. Объект **cout** соответствует стандартному потоку вывода. По умолчанию этот поток ассоциируется со стандартным устройством вывода — обычно дисплеем.

В классе **ostream** перегружены операции **<<** для вставки информации в поток вывода, поэтому она называется операцией вставки. Операция **<<** перегружена для применения со всеми базовыми типами C++: **char**, **short**, **int**, **float** и т.д. Синтаксис использования операции очень простой:

```
cout << значение;
```

Это значит, что **значение** будет вставлено в поток вывода (выведено на дисплей), например:

```
cout << "Это пример"; // Строка выводится на дисплей
```

Операция вывода перегружена так, что в одной строке можно выводить несколько значений разного типа:

```
#include <iostream.h>  
MPoint p1(10, 20);  
cout << "x = " << p1.GetX() << ", y = " << p1.GetY() << "\n";
```

На дисплее будет выведено:

```
x = 10, y = 20
```

В классе **istream** перегружены операции **>>** для извлечения информации из потока ввода, поэтому она называется операцией извлечения из потока.

Синтаксис использования операции:

```
cin >> переменная;
```

Это значит, что значение из потока ввода (с клавиатуры) будет присвоено переменной, например:

```
cin >> i; // вводится значение и присваивается i
```

Может быть несколько переменных:

```
cin >> i >> j >> k; // вводятся последовательно значения в i, j, k
```

Операция **>>** перегружена для применения со всеми базовыми типами C++ и автоматически определяет тип вводимых данных. Ввод осуществляется до первого неверного символа, а при вводе строк — до первого пробела.

Операции **<<** и **>>** можно перегрузить и для своих классов, например для класса **MPoint**. Тогда в программе можно будет выводить значение точки так:

```

MPoint p1;
. . .
cout << p1;

```

Здесь cout – это первый операнд, он имеет тип **ostream&**, а p1 – второй операнд, его тип **MPoint**. Можно объявить глобальную оператор-функцию:

```

std::ostream& operator<<(std::ostream& os, MPoint& p)
{
    os << "x = " << p.GetX() << ", y = " << p.GetY();
    return os;
}

```

Тогда в программе можно будет выводить значение точки так:

```

MPoint p1(10, 20);
cout << p1 << endl;

```

На дисплее будет выведено:

```

x = 10, y = 20

```

## Пример доработки класса

Рассмотрим процесс доработки класса на следующем примере:

### Задание:

Доработать класс Время и программу, иллюстрирующую возможности данного класса, – перегрузить операции:

a++;	время увеличилось на 1 секунду
a--;	время уменьшилось на 1 секунду
a += k;	время увеличилось на k секунд
a -= k;	время уменьшилось на k секунд
b = a + k;	время b стало больше времени a на k секунд
b = a - k;	время b стало меньше времени a на k секунд
k = a - b;	число секунд между временами a и b
a < b	сравнение двух времен на <
a > b	сравнение двух времен на >
a == b	сравнение двух времен на ==
a = b	время a стало равным времени b

В программе должны быть как минимум два объекта данного класса, окна ввода времен, вывода времен и кнопки, по которым вызываются функции, иллюстрирующие возможности этих объектов.

Разработанный ранее класс **CTime** выглядел так:

```

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)
        {Hour = 0; Min = 0; Sec = 0; SetTime(h, m, s);}
    CTime(const char* str) {SetTime(str);}

    void SetHour(BYTE h);           // Установка часов
    void SetMin(BYTE m);           // Установка минут
    void SetSec(BYTE s);           // Установка секунд
    void SetTime(BYTE h, BYTE m, BYTE s)
        {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
GetTime(char* str) // читать время строкой
void Print(); // Вывод на дисплей
};

```

Используя перегрузку операций можно значительно улучшить наш класс, сделать его более понятным и удобным в использовании. Добавляемые функции требуют особого подхода, так как в них проводятся **сравнения времен** и **арифметические действия над временами**. Чтобы упростить их, лучше было бы хранить время не в часах, минутах и секундах, а в секундах, прошедших с начала суток. Для осуществления такого подхода, напомним функции, которые переводят час, минуту и секунду в число секунд, прошедших с начала суток, и обратную функцию, которая переводит число секунд, прошедших с начала суток в час, минуту и секунду.

Число секунд с начала суток может лежать в пределах от 0 до  $24 * 3600 - 1$ , поэтому для его передачи в функцию и из функции нужна переменная типа **unsigned int**.

```

void CTime::SecToTime(unsigned int s)
{
    // s - число секунд, прошедших с начала суток
    Hour = s / 3600;
    Min = (s % 3600) / 60;
    Sec = (s % 3600) % 60;
}
unsigned int CTime::TimeToSec()
{
    unsigned int s = (unsigned int)Hour * 3600 + Min * 60 + Sec;
    return s; // число секунд, прошедших с начала суток
}

```

Эти функции должны использоваться только в методах класса, поэтому помещаем их объявление в закрытую часть класса.

Функции, выполняющие сравнения и арифметические действия легко реализуются с помощью этих функций. Сначала пишем функции – члены класса, сравнивающие время, хранящееся в классе, для которого вызываются данные методы, со временами, передаваемыми в функцию в качестве аргументов. Для сравнения переводим оба времени в число секунд, прошедших с начала суток, и сравниваем уже просто два числа.

```

class CTime
{
private:
    . . . . .
    void SecToTime(unsigned int s); // Секунды в час:мин:сек
    unsigned int TimeToSec(); // Час:мин:сек в секунды
public:
    . . . . .
    bool operator>(CTime& t1);
    bool operator<(CTime& t1);
    bool operator==(CTime& t1);
};

bool CTime::operator>(CTime& t1)
{
    return TimeToSec() > t1.TimeToSec();
}

```



```

bool CTime::operator<(CTime& t1)
{
    return TimeToSec() < t1.TimeToSec();
}
bool CTime::operator==(CTime& t1)
{
    return TimeToSec() == t1.TimeToSec();
}
Теперь их вызов будет более привычным:
int main()
{
    CTime Time1, Time2;
    . . . . .
    if (Time1 > Time2)    printf("Time1 > Time2");
    if (Time1 < Time2)    printf("Time1 < Time2");
    if (Time1 == Time2)   printf("Time1 == Time2");
}

```

Операцию  $k = a - b$ , вычисляющую число секунд между временами  $a$  и  $b$ , реализуем с помощью **операции** `-` (тоже член класса):

```

class CTime
{
    . . . . .
    int operator-(CTime& t1); // разница между временами в секундах
};
int CTime::operator-(CTime& t1)    // -
{
    if (TimeToSec() > t1.TimeToSec())
        return TimeToSec() - t1.TimeToSec();
    return t1.TimeToSec() - TimeToSec();
}

```

Ее использование:

```

int main()
{
    CTime Time1, Time2;
    . . . . .
    // разность времен Time1 и Time2
    int dT = Time1 - Time2;
}

```

Чтобы перегрузить операции `+` и `-`, вычисляющие время, отстоящее от исходного на заданное число секунд, создаем оператор-функции `operator+` и `operator-` :

```

class CTime
{
    . . . . .
    void operator+(int s);
    void operator-(int s);

};
CTime CTime::operator+(int s)    // operator +
{
    CTime tmp;
    int sNew = TimeToSec() + s;
    if (sNew >= 0 && sNew < 24 * 3600)
        tmp.SecToTime(sNew);
    return tmp;
}

```

Аналогично можно перегрузить и операцию `-`, а можно определить ее через уже написанную функцию `operator+`

```
class CTime
{
    . . . . .
    void operator-(int s) {operator+(-s);}
};
```

Теперь перегруженные операции можно использовать в таком виде:

```
Time2 = Time1 + d; // d - число секунд
```

```
Time2 = Time1 - d; // d - число секунд
```

Две перегруженные **операции** – различаются типом аргумента (у первого – `CTime&`, у второго – `int`) и имеют совершенно разный смысл и применение.

Операции `+=` и `-=` перегружаем аналогично:

```
class CTime
{
    . . . . .
    void operator +=(int s);
    void operator -=(int s) {operator+=(-s);}
};
void CTime::operator+=(int s) // operator +=
{
    int sNew = TimeToSec() + s;
    if (sNew >= 0 && sNew < 24 * 3600)
        SecToTime(sNew);
}
```

А операции `++` и `--` перегружаем, используя только что написанные операции. Чтобы результат выполнения операции можно было использовать для присваивания или как-то еще, делаем у функции возвращаемое значение типа `CTime` и возвращаем сам объект через разыменованный указатель `this` (можно возвращать ссылку на объект):

```
class CTime
{
    . . . . .
    CTime operator ++(int) {operator+=(1); return *this;}
    CTime operator --(int) {operator+=(-1); return *this;}
};
```

Операцию присваивания перегружать нет необходимости, т.к. в данном классе память динамически не выделяется, следовательно, побайтное копирование, которое проводит операция присваивания по умолчанию, нас вполне устраивает.

Если при тестировании используются потоки ввода/вывода, можно перегрузить **операцию** `<<` для вывода объекта класса `CTime` в поток, чтобы можно было писать так:

```
CTime t1("12:23:34");
cout << t1; // Вывод на дисплей времени в виде: 12:23:34
```

Для этого переделываем ранее написанную функцию `Print()` в **операцию** `<<`

```
void CTime::Print() // Был метод вывода на дисплей
{
    char s[10];
    GetTime(s);
    cout << s;
}
std::ostream& operator<<(std::ostream& os, CTime& t)
{
    // Новый метод вывода на дисплей
    char s[10];
```

```

    t.GetTime(s);
    os << s;
    return os;
}

```

Тогда на дисплей информацию можно будет выводить проще:

```

for(int i=0; i<6; i++)
    cout << times[i] << endl;

```

**Окончательно получаем класс Время:**

```

typedef unsigned char BYTE;

class CTime
{
private:
    BYTE Hour;
    BYTE Min;
    BYTE Sec;
    void SecToTime(unsigned int s);
    unsigned int TimeToSec();
public:
    CTime() {Hour = 0; Min = 0; Sec = 0;} // Конструктор по умолчанию
    CTime(BYTE h, BYTE m, BYTE s){Hour=h;Min=m;Sec=s; SetTime(h,m,s);}
    CTime(const char* str) {SetTime(str);}

    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)
        {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

    bool operator>(CTime& t1); // Сравнение на >
    bool operator<(CTime& t1); // Сравнение на <
    bool operator==(CTime& t1); // Сравнение на ==
    int  operator-(CTime& t1); // Разница между временами в секундах
    CTime operator+(int s); // Время, отстоящее на s секунд
    CTime operator-(int s) // Время, меньшее на s секунд
        {return operator+(-s);}
    void operator +=(int s); // Увеличить время на s секунд
    void operator -=(int s) {operator+=(-s);} // Уменьшить на s секунд
    CTime operator ++(int) // Увеличить на 1 сек.
        {operator+=(1); return *this;}
    CTime operator --(int) // Уменьшить на 1 сек.
        {operator+=(-1); return *this;}
};

//----- методы -----
void CTime::SecToTime(unsigned int s)
{
    // s - число секунд, прошедших с начала суток
    Hour = s / 3600;
    Min = (s % 3600) / 60;
}

```

```

    Sec = (s % 3600) % 60;
}
unsigned int CTime::TimeToSec()
{
    int s = (int)Hour * 3600 + Min * 60 + Sec;
    return s;    // число секунд, прошедших с начала суток
}
void CTime::SetTime(const char* str)
{
    // преобразуем строку "12:34:56" в час, мин и сек
    BYTE h, m, s;
    if(str[2] == ':' && str[5] == ':') // Если символы 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 = 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
}
bool CTime::operator>(CTime& t1)
{
    return TimeToSec() > t1.TimeToSec();
}
bool CTime::operator<(CTime& t1)
{
    return TimeToSec() < t1.TimeToSec();
}
bool CTime::operator==(CTime& t1)
{
    return TimeToSec() == t1.TimeToSec();
}
CTime CTime::operator+(int s)
{
    CTime tmp;
    int sNew = TimeToSec() + s;
    if (sNew >= 0 && sNew < 24 * 3600)
        tmp.SetTime(sNew);
    return tmp;
}

```

```

}
int CTime:: operator-(CTime& t1)
{
    if (TimeToSec() > t1.TimeToSec())
        return TimeToSec() - t1.TimeToSec();
    return t1.TimeToSec() - TimeToSec();
}
void CTime::operator+=(int s)
{
    int sNew = TimeToSec() + s;
    if (sNew >= 0 && sNew < 24 * 3600)
        SecToTime(sNew);
}
//----- Конец методов -----
std::ostream& operator<<(std::ostream& os, CTime& t)
{
    char s[10];
    t.GetTime(s);
    os << s;
    return os;
}
//----- Конец методов -----

```

Тестирование класса:

```

#include <stdio.h>
#include <conio.h>
#include <cstdlib>
#include <time.h>
#include <locale.h>

int main()
{
    CTime times[6];
    char s1[10];
    int w;

    srand(time(0));
    setlocale(0, "Russian");
    for (;;) // Бесконечный цикл
    {
        printf("\n\nМеню:\n");
        printf("0 - выход из программы\n");
        printf("1 - заполнить случайными числами: \n");
        printf("2 - заполнить по возрастанию: \n");
        printf("3 - просмотр: \n");
        printf("4 - сравнить: \n");
        printf("5 - добавить сек: \n");
        printf("6 - считать разности: \n");
        . . . . . и т.п.

        scanf_s(" %d", &w);

        switch (w)
        {
            case 0:    // выход

```

```

    return 0;
case 1:    // случайными числами
    for(int i=0; i<6; i++)
    {
        times[i].SetHour((float)rand()/RAND_MAX * 24);
        times[i].SetMin((float)rand()/RAND_MAX * 60);
        times[i].SetSec((float)rand()/RAND_MAX * 60);
    }
    for(int i=0; i<6; i++)
    {
        times[i].GetTime(s1);
        printf("%s\n", s1);
    }
    break;
case 2:    // по возрастанию
    for(int i=0; i<6; i++)
    {
        times[i].SetHour(i);
        times[i].SetMin(i*5);
        times[i].SetSec(i*10);
    }
    for(int i=0; i<6; i++)
    {
        times[i].GetTime(s1);
        printf("%s\n", s1);
    }
    break;
case 3:    // просмотр
    for(int i=0; i<6; i++)
    {
        times[i].GetTime(s1);
        printf("%s\n", s1);
    }
    break;
case 4:    // сравнить
    for(int i=0; i<5; i++)
    {
        if (times[i] > times[i+1])
            printf("times[%d] > times[%d]\n", i, i+1);
        if (times[i] < times[i+1])
            printf("times[%d] < times[%d]\n", i, i+1);
        if (times[i] == times[i+1])
            printf("times[%d] == times[%d]\n", i, i+1);
    }
    break;
case 5:
. . . . . и т.д.
    }
}
return 0;
}

```

## ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

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

### Задания

По заданию преподавателя доработать класс, созданный в предыдущей лабораторной работе, и программу, иллюстрирующую возможности данного класса. Необходимо продемонстрировать работу со всеми методами и перегруженными операциями класса. Вариант задания выдает преподаватель.

*Для получения максимального балла в программе необходимо:*

- сделать цикл для организации повторных вычислений и меню,
- перегрузить операцию вывода ( $<<$ ) параметров объекта класса на дисплей

### Варианты заданий

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

Доработать класс **Аквариум** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

- $a * k$ ; увеличение размеров аквариума в  $k$  раз
- $a /= k$ ; уменьшение размеров аквариума в  $k$  раз
- $a += k$ ; долить заданный объем жидкости ( $k$ ) в аквариум  $a$  (естественно, не превышая объема аквариума);
- $a -= k$ ; вылить заданный объем жидкости ( $k$ ) из аквариума  $a$  (естественно, не больше, чем было жидкости к этому моменту);
- $k = a - b$ ; вычисление разности объемов жидкостей двух аквариумов;
- $a << b$ ; перелить весь объем жидкости из одного аквариума в другой (естественно, не превышая объема аквариума);
- $a < b$  ( $a > b$ ) сравнение двух аквариумов по объему налитой жидкости на  $<$  ( $>$ )
- $a == b$  сравнение двух аквариумов по объему налитой жидкости на  $==$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

#### Вариант 2. Колесо со спицами

Доработать класс **Колесо со спицами** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

- $a * k$ ; умножение размеров колеса на некоторое число
- $a / k$ ; деление размеров колеса на некоторое число
- $k = a / b$ ; вычисление отношения площадей двух колес
- $k = a - b$ ; вычисление разности площадей двух колес
- $k = a + b$ ; вычисление суммы площадей двух колес
- $a < b$  ( $a > b$ ) сравнение двух колес (по площади) на  $<$  ( $>$ )
- $a == b$  сравнение двух колес (по площади) на  $==$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
- задавать / изменять параметры выбранного объекта вручную;

- выполнять все перегруженные операции для выбранных объектов;  
Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 3. Равносторонний шестиугольник

Доработать класс **Равносторонний шестиугольник** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a * = k;$	увеличение размера шестиугольника в $k$ раз
$a /= k;$	уменьшение размера шестиугольника в $k$ раз
$k = a / b;$	вычисление отношения площадей двух шестиугольников
$k = a - b;$	вычисление разности площадей двух шестиугольников
$k = a + b;$	вычисление суммы площадей двух шестиугольников
$a < b$ ( $a > b$ )	сравнение двух шестиугольников (по площади) на $<$ ( $>$ )
$a == b$	сравнение двух шестиугольников (по площади) на $=$
$a[i]$	чтение значения $i$ -той вершины шестиугольника

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
- задавать / изменять параметры выбранного объекта вручную;
- выполнять все перегруженные операции для выбранных объектов;  
Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 4. Обыкновенная дробь

Доработать класс **Обыкновенная дробь** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$k = a / b;$	деление двух дробей
$k = a * b;$	умножение двух дробей
$k = a - b;$	разность двух дробей
$k = a + b;$	сумма двух дробей
$a < b$ ( $a > b$ )	сравнение двух дробей на $<$ ( $>$ )
$a == b$	сравнение двух дробей на $=$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
- задавать / изменять параметры выбранного объекта вручную;
- выполнять все перегруженные операции для выбранных объектов;  
Все проверки корректности ввода должны проводиться в методах класса.

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

Доработать класс **Бочка** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a * = k;$	увеличение размеров бочки в $k$ раз
$a /= k;$	уменьшение размеров бочки в $k$ раз
$a += k;$	долить заданный объем жидкости $k$ в бочку $a$ (естественно, не превышая объема аквариума);
$a -= k;$	вылить заданный объем жидкости $k$ из бочки $a$ (естественно, не больше, чем было жидкости к этому моменту);
$k = a - b;$	вычисление разности объемов жидкостей двух бочек;
$a < b$ ( $a > b$ )	сравнение двух бочек по объему налитой жидкости на $<$ ( $>$ )



$a = b$  сравнение двух бочек по объему налитой жидкости на  $==$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;

- задавать / изменять параметры выбранного объекта вручную;

- выполнять все перегруженные операции для выбранных объектов;

Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 6. Лампочка

Доработать класс **Лампочка** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$k = a / b$ ; отношение рассеиваемых мощностей двух лампочек

$k = a - b$ ; вычисление разности рассеиваемых мощностей двух лампочек

$a < b$  сравнение двух лампочек (по рассеиваемой мощности) на  $<$

$a > b$  сравнение двух лампочек (по рассеиваемой мощности) на  $>$

$a == b$  сравнение двух лампочек (по рассеиваемой мощности) на  $==$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;

- задавать / изменять параметры выбранного объекта вручную;

- выполнять все перегруженные операции для выбранных объектов;

Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 7 Пирамида

Доработать класс **Пирамида** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a *= k$ ; увеличение размеров пирамиды в  $k$  раз

$a /= k$ ; уменьшение размеров пирамиды в  $k$  раз

$k = a / b$ ; вычисление отношения объемов двух пирамид

$k = a - b$ ; вычисление разности объемов двух пирамид

$k = a + b$ ; вычисление суммы объемов двух пирамид

$a < b$  ( $a > b$ ) сравнение двух пирамид (по объему) на  $<$  ( $>$ )

$a == b$  сравнение двух пирамид (по объему) на  $==$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;

- задавать / изменять параметры выбранного объекта вручную;

- выполнять все перегруженные операции для выбранных объектов;

Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 8. Комплексное число

Доработать класс **Комплексное число** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a += b$ ; комплексное число  $a$  увеличилась на  $b$  ( $b$  тоже комплексное число)

$a -= b$ ; комплексное число  $a$  уменьшилась на  $b$  ( $b$  тоже комплексное число)

$c = a + b$ ; сумма комплексных чисел ( $a$  также операции  $-$ ,  $*$ ,  $/$ )

$a < b$  сравнение двух комплексных чисел на  $<$

$a > b$	сравнение двух комплексных чисел на $>$
$a == b$	сравнение двух комплексных чисел на $==$
$a = b$	комплексное число $a$ стало равным $b$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 9. Треугольник

Доработать класс **Треугольник** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a++$	сдвиг треугольника вправо на 1 пиксел
$a--$	сдвиг треугольника влево на 1 пиксел
$++a$	сдвиг треугольника вверх на 1 пиксел
$--a$	сдвиг треугольника вниз на 1 пиксел
$k = a / b;$	вычисление отношения площадей двух треугольников
$k = a - b;$	вычисление разности площадей двух треугольников
$k = a + b;$	вычисление суммы площадей двух треугольников
$a < b$ ( $a > b$ )	сравнение двух треугольников (по площади) на $<$ ( $>$ )
$a == b$	сравнение двух треугольников (по площади) на $=$
$a[i]$	обращение к $i$ -той вершине треугольника

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 10. Цилиндр

Доработать класс **Цилиндр** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a *= k;$	увеличение размеров цилиндра в $k$ раз
$a /= k;$	уменьшение размеров цилиндра в $k$ раз
$k = a / b;$	вычисление отношения объемов двух цилиндров
$k = a - b;$	вычисление разности объемов двух цилиндров
$k = a + b;$	вычисление суммы объемов двух цилиндров
$a < b$ ( $a > b$ )	сравнение двух цилиндров (по объему) на $<$ ( $>$ )
$a == b$	сравнение двух цилиндров (по объему) на $==$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 11. Произвольный четырехугольник

Доработать класс **Четырехугольник** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

<code>a++</code>	сдвиг четырехугольника вправо на 1 пиксел
<code>a--</code>	сдвиг четырехугольника влево на 1 пиксел
<code>++a</code>	сдвиг четырехугольника вверх на 1 пиксел
<code>--a</code>	сдвиг четырехугольника вниз на 1 пиксел
<code>k = a / b;</code>	вычисление отношения периметров двух четырехугольников
<code>k = a - b;</code>	вычисление разности периметров двух четырехугольников
<code>k = a + b;</code>	вычисление суммы периметров двух четырехугольников
<code>a &lt; b (a &gt; b)</code>	сравнение двух четырехугольников (по периметру) на <code>&lt;</code> и <code>&gt;</code>
<code>a == b</code>	сравнение двух четырехугольников (по периметру) на <code>=</code>
<code>a[i]</code>	обращение к i-той вершине четырехугольника

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 12. Трапеция

Доработать класс **Трапеция** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

<code>a++</code>	сдвиг трапеции вправо на 1 пиксел
<code>a--</code>	сдвиг трапеции влево на 1 пиксел
<code>++a</code>	сдвиг трапеции вверх на 1 пиксел
<code>--a</code>	сдвиг трапеции вниз на 1 пиксел
<code>k = a / b;</code>	вычисление отношения площадей двух трапеций
<code>k = a - b;</code>	вычисление разности площадей двух трапеций
<code>k = a + b;</code>	вычисление суммы площадей двух трапеций
<code>a &lt; b (a &gt; b)</code>	сравнение двух трапеций (по площади) на <code>&lt;</code> ( <code>&gt;</code> )
<code>a == b</code>	сравнение двух трапеций (по площади) на <code>=</code>
<code>a[i]</code>	чтение значения i-той вершины трапеции

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 13. Ромб

Доработать класс **Ромб** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

<code>a++</code>	сдвиг ромба вправо на 1 пиксел
<code>a--</code>	сдвиг ромба влево на 1 пиксел
<code>++a</code>	сдвиг ромба вверх на 1 пиксел
<code>--a</code>	сдвиг ромба вниз на 1 пиксел
<code>k = a / b;</code>	вычисление отношения площадей двух ромбов

$k = a - b$ ;	вычисление разности площадей двух ромбов
$k = a + b$ ;	вычисление суммы площадей двух ромбов
$a < b$ ( $a > b$ )	сравнение двух ромбов (по площади) на $<$ ( $>$ )
$a == b$	сравнение двух ромбов (по площади) на $=$
$a[i]$	чтение значения $i$ -той вершины ромба

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

#### Вариант 14. Параллелограмм

Доработать класс **Параллелограмм** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a++$	сдвиг параллелограмма вправо на 1 пиксел
$a--$	сдвиг параллелограмма влево на 1 пиксел
$++a$	сдвиг параллелограмма вверх на 1 пиксел
$--a$	сдвиг параллелограмма вниз на 1 пиксел
$k = a / b$ ;	вычисление отношения площадей двух параллелограммов
$k = a - b$ ;	вычисление разности площадей двух параллелограммов
$k = a + b$ ;	вычисление суммы площадей двух параллелограммов
$a < b$ ( $a > b$ )	сравнение двух параллелограммов (по площади) на $<$ ( $>$ )
$a == b$	сравнение двух параллелограммов (по площади) на $=$
$a[i]$	чтение значения $i$ -той вершины параллелограмма

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

#### Вариант 15. Конус

Доработать класс **Конус** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a *= k$ ;	увеличение размеров конуса в $k$ раз
$a /= k$ ;	уменьшение размеров конуса в $k$ раз
$k = a / b$ ;	вычисление отношения объемов двух конусов
$k = a - b$ ;	вычисление разности объемов двух конусов
$k = a + b$ ;	вычисление суммы объемов двух конусов
$a < b$ ( $a > b$ )	сравнение двух конусов (по объему) на $<$ ( $>$ )
$a == b$	сравнение двух конусов (по объему) на $=$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
- задавать / изменять параметры выбранного объекта вручную;
- выполнять все перегруженные операции для выбранных объектов;

Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 16. Резистор

Доработать класс **Резистор** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$k = a / b;$	отношение сопротивлений двух резисторов
$k = a - b;$	разность сопротивлений двух резисторов
$a < b$	сравнение двух резисторов (по сопротивлению) на $<$
$a > b$	сравнение двух резисторов (по сопротивлению) на $>$
$a == b$	сравнение двух резисторов (по сопротивлению) на $=$
$c = a + b;$	сложение последовательно включенных резисторов
$c = a    b;$	сложение параллельно включенных резисторов

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 17. Равносторонний пятиугольник

Доработать класс **Равносторонний пятиугольник** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

$a *= k;$	увеличение размера пятиугольника в $k$ раз
$a /= k;$	уменьшение размера пятиугольника в $k$ раз
$k = a / b;$	вычисление отношения площадей двух пятиугольников
$k = a - b;$	вычисление разности площадей двух пятиугольников
$k = a + b;$	вычисление суммы площадей двух пятиугольников
$a < b$ ( $a > b$ )	сравнение двух пятиугольников (по площади) на $<$ ( $>$ )
$a == b$	сравнение двух пятиугольников (по площади) на $=$
$a[i]$	чтение значения $i$ -той вершины пятиугольника

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### Вариант 18. Трехмерный вектор

Доработать класс **Трехмерный вектор** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

операция $[]$	задавать и читать компоненты вектора;
$a = b + c;$	сложение векторов;
$a = b - c;$	вычитание векторов;
$f = b * c;$	скалярное произведение векторов;
$a *= k;$	вектор увеличился в $k$ раз
$a /= k;$	вектор уменьшился в $k$ раз
$a = b$	вектор $a$ стал равен вектору $b$

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### **Вариант 19. Логический элемент И**

Доработать класс **Логический элемент И** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

- [ ] задать (читать) значение на i-том входе
- a < b сравнение логических элементов (по значению выходного сигнала) на <
- a > b сравнение логических элементов (по значению выходного сигнала) на >
- a == b сравнение логических элементов (по значению выходного сигнала) на ==
- a = b параметры элемента a стали равны параметрам элемента b
- a << "1 0 0 1" задать значения всех входных сигналов строкой

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.

### **Вариант 20. Логический элемент ИЛИ**

Доработать класс **Логический элемент ИЛИ** и программу, иллюстрирующую возможности данного класса. Для этого необходимо перегрузить операции:

- [ ] задать (читать) значение на i-том входе
- a < b сравнение логических элементов (по значению выходного сигнала) на <
- a > b сравнение логических элементов (по значению выходного сигнала) на >
- a == b сравнение логических элементов (по значению выходного сигнала) на ==
- a = b параметры элемента a стали равны параметрам элемента b
- a << "1 0 0 1" задать значения всех входных сигналов строкой

В программе необходимо создать динамический массив объектов данного класса, для чего иметь возможность задавать размер массива. Программа должна иметь возможность:

- задавать параметры всех объектов автоматически по возрастанию с некоторым шагом;
  - задавать / изменять параметры выбранного объекта вручную;
  - выполнять все перегруженные операции для выбранных объектов;
- Все проверки корректности ввода должны проводиться в методах класса.