

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

ОБЪЕКТНО -ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Урок 3

Константные методы,
explicit конструктора.
Перегрузка операторов

Содержание

1. Константный метод.....	4
1.1. Синтаксис объявления.....	6
1.2. Особенности указателя this в константном методе.....	7
1.3. Константы — члены класса.....	9
1.4. Примеры использования.....	10
2. Объявление конструктора с использованием ключевого слова explicit	16
2.1. Примеры ситуации, иллюстрирующие неявное создание объекта	16
2.2. Ключевое слово explicit и его использование.....	20
2.2.1. Объявление конструктора с использованием ключевого слова explicit	20
3. Необходимость использования перегрузки операторов	26
3.1. Примеры кода (реализация классов через обычные методы члены типа Sum, Mult и так далее)	26

3.2. Логичность использования стандартных символов (+, — , >,< и т. д.)	30
4. Перегрузка операторов.....	34
4.1. Общие понятия перегрузки операторов.....	34
4.1.1. Классификация операторов на основании количества операндов (бинарные, унарные, триадный).....	34
4.1.2. Определение перегрузки операторов	35
4.1.3. Различные виды перегрузки (метод-член, функция-друг, глобальная функция)	38
4.2. Примеры перегрузки операторов.....	43
4.2.1. Перегрузка арифметических операторов	43
4.2.2. Перегрузка операторов сравнения и логических операторов.....	48
4.2.3. Возврат по ссылке. Перегрузка операторов <<, >>	49
4.2.4. Перегрузка оператора присваивания	53
4.2.5. Как выбирать способ перегрузки.....	61
4.2.6. Пример класса с перегрузкой операторов (динамический массив Array)	64
5. Резюме.....	71
6. Домашнее задание.....	83
7. Терминология	86

Материалы урока прикреплены к данному PDF-файлу. Для доступа к ним, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Константный метод

Мы знаем, что в языке C++ есть константы, которые (в отличие от многих других языков программирования) очень часто применяются, например,

```
const int size = 3;  
const string s{ "first day"};
```

Также часто применяются константные указатели и константные массивы:

```
const char* s{ "Next day"};  
const int days{29,30, 31};
```

Константными могут быть и объекты классов:

```
class Date  
{  
    int month;  
    int day;  
    int year;  
};  
  
const Date electionDay{ 11, 03, 2020 };
```

Константные объекты и ссылки на константные объекты часто появляются в тексте программ, как параметры методов класса, например, параметры конструктора копирования

```
Date(const Date& date)
```

Ключевое слово `const` при определении объекта указывает, что объект является неизменяемым, и что любая попытка изменения этого объекта будет ошибкой. Значение константному объекту не может присваиваться, следовательно, такой объект может получить значение только путем инициализации. Мы также, никаким образом не можем изменить значение константного объекта. Ни прямым присвоением, ни посредством функций класса.

```
electionDay.setYear(2024); // ошибка!
```

Таким образом, все что можно сделать с константным объектом, это — получить значения его элементов (`get...`) или вывести их на экран (`write...`). С константными объектами могут происходить непонятные, на первый взгляд, события.

```
int day = electionDay.getDay(); // ошибка!
```

`getDay()` — это метод класса `Date`, который просто возвращает значения дня даты.

```
int getDay()  
{  
    return day;  
}
```

Этот метод не изменяет никакие данные в объекте `electionDay...` Но компилятор-то при обработке вызова `electionDay.getDay()` ничего об этом не знает! И, на всякий случай, не разрешает вызывать этот метод для константного объекта, чтобы предупредить возможное его изменение.

1.1. Синтаксис объявления

Однако, из этой ситуации есть простой выход. Следует методы, которые не изменяют значение объекта (а значит могут вызываться и для констант), пометить специальным образом:

```
int getDay() const
{
    return day;
}
```

Ключевое слово `const` — после закрывающей круглой скобки в заголовке функции, — обозначает, что функция не изменяет значения объекта. Причем, компилятор проверяет, соответствует ли это объявление действительности.

Такая функция называется константной функцией-членом класса или константным методом. Константные методы (и только они) могут вызываться для константных объектов. Метод указывается как константный и в прототипе, и в определении метода.

Следует заметить, что в классе можно перегрузить функцию таким образом, чтобы иметь константную и не-константную версии этой функции одновременно:

```
double getValue();
double getValue() const;
```

Необходимость в двух таких версиях бывает при перегрузке операторов (которые мы рассмотрим позже), где нужны одновременно и неконстантная перегрузка для

чтения и записи объектов и константная перегрузка для константных объектов.

Конструкторы и деструкторы не могут объявляться как `const`. Действительно, назначение этих специальных методов — неперменное изменение объекта (создание или уничтожение), поэтому константными они быть не могут.

1.2. Особенности указателя `this` в константном методе

Рассмотрим небольшой пример (код программы — в папке `Lesson03\les03_00_01`):

```
#include <iostream>
#include <stdio.h>

class Date
{
    private:
        int day;
        int month;
        int year;

    public:
        void setDay(int value)
        {
            day = value;
        }

        int getDay() const
        {
            return day;
        }
};
```

```
int main()
{
    Date aDate;
    aDate.setDay(10);
    std::cout << aDate.getDay() << std::endl;

    getchar();
    return 0;
}
```

При обработке оператора `aDate.setDay(10);` компилятор воспринимает объект `aDate` как скрытый параметр и преобразует вызов `setDay(10)` в вызов функции `setDay(&aDate, 10)`. Соответственно, сам метод `setDate` преобразуется компилятором в функцию с двумя параметрами:

```
void setDay(Date* const this, int value)
{
    this->day = value;
}
```

При компиляции обычного (нестатического) метода класса, компилятор неявно добавляет к списку параметров неявный параметр `*this`. Указатель `*this` — это скрытый указатель, содержащий адрес того объекта, который вызывает метод класса.

Ключевое слово `const`, указанное после закрывающей скобки заголовка функции, например, `int getDay() const`, не имеет отношения к значению, которое возвращается функцией. Оно относится только к скрытому указателю `this` используемому в функции, и обозначает, что то,

на что указывает `this` (то есть, данные объекта) — не может быть изменено.

Если функция объявлена с константным указателем `this`, то в теле функции запрещается изменять данные объектов класса. При попытке в коде изменить объект, будет возникать ошибка компиляции (см. рис. 1).

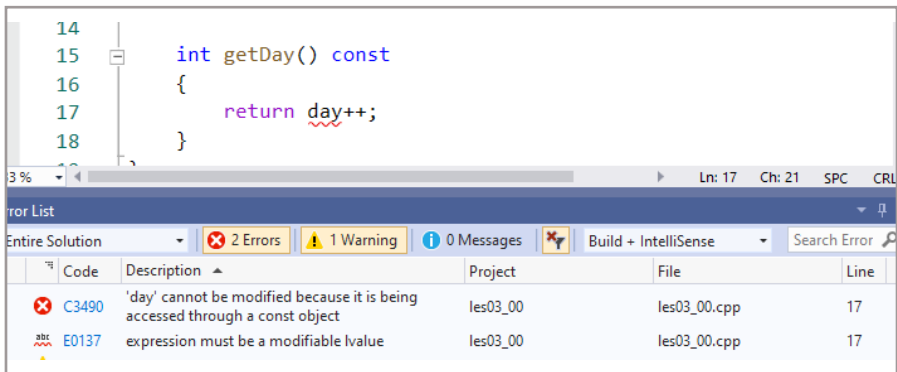


Рисунок 1. Ошибка компиляции константного метода

При попытке изменения переменной-члена класса `day` в константной функции компилятор выдает сообщение об ошибке: «переменная `day` не может быть изменена, поскольку является элементом константного объекта».

1.3. Константы — члены класса

Поля класса, являющиеся константами, а также поля, являющиеся ссылками, в коде класса не могут получать значения нигде, кроме момента инициализации. При этом они должны инициализироваться только с помощью инициализаторов (код программы — в папке `Lesson03\les03_00_02`).

```

class Date
{
    private:
        const int baseYear;
        int& currentYear;
        int day;
        int month;
        int year;
    public:
        Date(int currYear) : baseYear{ 2000 },
                           currentYear(currYear)
}

```

Константа `baseYear` и ссылка `currentYear` получают значения в списке инициализации конструктора класса.

1.4. Примеры использования

Рассмотрим следующую программу, в которой есть класс `Account` и два объекта этого класса: неконстантный `account1` и константный `account2` (код программы — в папке `Lesson03\les03_01`).

В этот код специально внесены ошибки — для демонстрации возможных проблем, возникающих при использовании (или не использовании) константных методов

```

#include <conio.h>
using namespace std;

class Account
{
    private:
        double sum;
        const double rate;

```

```

public:
    Account(double Rate, double Sum)
    {
        // Ошибка C2789
        rate = Rate; // "Account::rate": требуется
                     // инициализация объекта
                     // типа класса, квалифицированного
                     // как const
                     // Ошибка C2166
                     // левостороннее значение
                     // указывает на объект – /7
                     // константу(const)

        sum = Sum;
    }

    double getRate() const
    {
        return rate;
    }

    double getIncome()
    {
        return sum * rate / 100;
    }

    double getIncome() const
    {
        return sum * rate / 100;
    }

    double getSum()
    {
        return sum;
    }

    double setSum() const
    {
        // Ошибка C3490

```

```

        sum += getIncome(); // "sum" не может быть изменен,
                               // поскольку доступ к нему
                               // осуществляется через
                               // константный объект
    return sum;
}
};

int main()
{
    Account account1(5, 2000);
    const Account account2(8, 5000);

    account1.getRate();
    account2.getRate();    // все OK

    account1.getSum();
    // Ошибка C2662
    account2.getSum(); // double Account::getSum(void) :
                       // невозможно преобразовать
                       // указатель "this"
                       // из "const Account" в "Account &"

    account1.getIncome();
    account2.getIncome();

    account1.setSum();
    account2.setSum();

    _getch();
    return 0;
}

```

При запуске программы видим ошибки компиляции:

- `rate = Rate;` — при попытке задать значение константному полю класса при помощи оператора присваивания;

- `sum += getIncome();` — при попытке изменить значение поля класса в константном методе;
- `account2.getSum();` — при попытке вызвать неконстантный метод для константного объекта.

Исправляем ошибки (код программы — в папке *Lesson03\les03_02*):

```
#include <iostream>
#include <conio.h>
using namespace std;
class Account
{
private:
    double sum;
    const double rate;

public:
    Account(double Rate, double Sum) : rate{ Rate }
    {
        this->sum = Sum;
    }

    double getRate() const
    {
        return rate;
    }

    double getIncome() // перегрузка: неконстантный метод
    {
        return sum * rate / 100;
    }

    double getIncome() const // перегрузка: такой же,
                             // но константный метод
    {
```

```

        return sum * rate / 100;
    }

    double getSum() const
    {
        return sum;
    }

    double setSum()
    {
        sum += getIncome();
        return sum;
    }
};

int main()
{
    Account account1(5, 2000);
    const Account account2(8, 5000); // константный объект

    account1.getRate();
    account2.getRate();

    account1.getSum();
    account2.getSum();

    account1.getIncome(); // вызывается неконстантный
                          // метод double getIncome()
    account2.getIncome(); // вызывается константный
                          // метод double getIncome()
                          // const

    account1.setSum();
    _getch();
    return 0;
}

```

Программа компилируется и запускается.

Как правило, в программах следует делать константными методами:

- методы-аксессоры (`getValue() const`);
- методы, выводящие информацию (`printArray() const`).

Все методы, которые выдают какие-либо характеристики объекта, не изменяя при этом значение объекта, должны быть описаны как константные. В противном случае их нельзя будет применить для константных объектов или даже для нормальных объектов, доступ к которым осуществляется через константный указатель или ссылку. Отсутствие `const` в описании таких методов является неправильным стилем программирования на C++

2. Объявление конструктора с использованием ключевого слова `explicit`

2.1. Примеры ситуации, иллюстрирующие неявное создание объекта

В процессе обработки данных часто возникает необходимость преобразования одних типов данных в другие. Преобразование типов может выполняться в следующих случаях:

- присваивание или инициализация переменной значением другого типа данных;

```
double a(10);    // инициализация переменной a (тип double)
                 // константой 10 (тип int)

a = 5;           // присваиваем переменной a (тип double)
                 // константу 5 (тип int)
```

- передача значения в функцию, когда тип передаваемого значения отличается от типа параметра;

```
void doSomething(long number)
{
}

doSomething(5); // передача значения 5 (тип int)
                // в функцию, у которой соответствующий
                // параметр имеет тип long
```


- возврат из функции, когда тип возвращаемого значения в операторе `return` отличается от типа возвращаемого значения, указанного в заголовке функции;

```
float doSomething()  
{  
    return 10.0; // возврат значения 10.0 (тип double)  
                // из функции, которая возвращает  
                // значение типа float  
}
```

- использование бинарного оператора с операндами разных типов;

```
double // операция деления со значениями типов double и int
```

Преобразования типов могут выполняться и для типов данных, разработанных программистом, — преобразования между объектами разных классов, а также преобразования между объектами классов и фундаментальными типами.

Приведем пример классов, для которых такие преобразования были бы естественными и желательными.

Для работы с данными даты и времени было бы удобно разработать следующий набор классов:

- `Date` (*дата*),
- `Time` (*время*),
- `DateTime` (*дата — время*),
- `TimeSpan` (*интервал времени*).

Для представления времени в C++ используется тип `time_t`, значения которого отображают количество

секунд, прошедшее после некоторой базовой временной точки (00:00 1 января 1970 года). В сущности, значения `time_t` — целые числа (`int`).

Таким образом, для комфортной работы с классами даты и времени и использования существующих функций безусловно необходимы преобразования между данными классов `Date`, `Time`, `Date`, `Time`, а также между данными этих классов и данными типа `int`.

Как мы знаем, есть два способа преобразования типов:

- неявное преобразование типов, когда компилятор автоматически конвертирует один тип данных в другой;
- явное преобразование типов, когда используется один из операторов явного преобразования для выполнения конвертации объекта из одного типа данных в другой.

По умолчанию язык C++ понимает конструктор класса с 1 параметром, как оператор неявного преобразования из типа параметра в тип класса.

Рассмотрим пример (код программы — в папке `Lesson03\les03_05`).

```
#include <iostream>
#include <stdio.h>

class Date
{
private:
    int day;
    int month;
    int year;
```

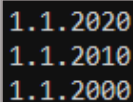
```

public:
    Date(int day, int month, int year)
        : day{ day }, month{ month }, year{ year }
    {}
    Date(int year) : Date(1, 1, year)
    {}
    friend void displayDate(Date date);
};

void displayDate(Date date)
{
    std::cout << date.day << "." << date.month << "." <<
        date.year << std::endl;
}
Date baseDate()
{
    return 2000;
}
int main()
{
    displayDate(2020);
    Date date = 2010;
    displayDate(date);
    Date date2000 = baseDate();
    displayDate(date2000);
    getchar();
    return 0;
}

```

Результат работы программы приведен на рисунке 2.



```

1.1.2020
1.1.2010
1.1.2000

```

Рисунок 2. Неявное создание объектов Date

В приведенном примере определен конструктор `Date` с одним параметром — целым числом:

```
Date(int).
```

Этот конструктор неявно вызывается:

- при вызове функции с типом параметра `Date` и фактическим параметром `int`

```
displayDate(2020);
```

- при выполнении присваивания переменной типа `Date` целого числа

```
Date date = 2010;
```

- при возврате целого числа функцией, которая возвращает `Date`

```
Date date2000 = baseDate()
```

Во всех случаях выполняется неявное преобразование целого числа в объект типа `Date`.

2.2. Ключевое слово `explicit` и его использование

2.2.1. Объявление конструктора с использованием ключевого слова `explicit`

Неявное преобразование целого числа в объект типа `Date` не всегда желательно, а иногда может быть просто ошибкой.

Рассмотрим следующий пример (код программы — в папке `Lesson03\les03_06`).

```
#include <iostream>
#include <conio.h>
using namespace std;

class Array
{
    int size;
    int* array;
public:
    Array(int size = 10);
    ~Array();
    int getSize() const;
    int getValue(int index) const;
    void setValue(int index, int value);
    void display(int index) const;
};

Array::Array(int size)
{
    Array::size = size;
    array = new int[size];
}

Array::~~Array()
{
    delete[] array;
}

int Array::getSize() const
{
    return size;
}

int Array::getValue(int index) const
{
    return array[index];
}
```

```
void Array::setValue(int index, int value)
{
    array[index] = value;
}

void Array::display(int index) const
{
    cout << array[index] << " ";
}

void display(const Array& array)
{
    for (int i = 0; i < array.getSize(); i++)
    {
        array.display(i);
    }
    cout << endl;
}

int main()
{
    cout << "Dynamic integer array" << endl;
    int size = 4;
    Array array(size);

    for (int i = 0; i < size; i++)
    {
        array.setValue(i, size - i);
    }
    display(array);

    cout << "!!!" << endl;
    display(3);

    _getch();
    return 0;
}
```

Результат работы программы приведен на рисунке 3.

```
Dynamic integer array
4 3 2 1
!!!
-842150451 -842150451 -842150451
_
```

Рисунок 3. Неявное создание объекта `Array`

В приведенном примере определен класс `Array` с конструктором `Array(int)`. Конструктор создает пустой динамический массив указанного размера. Глобальная функция `display(const Array& array)` выводит массив на экран.

```
int size = 4;
Array array(size);
display(array);
```

И все бы хорошо. Но... При вызове функции `display(3)` компилятор ожидает параметр типа `Array`, как это указано в заголовке функции `display(const Array& array)`, а получает параметр 3 типа `int`. Как мы знаем, в такой ситуации (передача значения в функцию, когда тип передаваемого значения отличается от типа параметра) происходит преобразование типа значения (3) в тип параметра (`Array`) — если только такое преобразование возможно.

Но оно возможно! — так как имеется конструктор `Array(int)`, и, следовательно, компилятор преобразовывает параметр 3 в объект типа `Array`, — создает объект `Array[3]` посредством вызова конструктора `Array(3)`, в результате чего параметром функции `display(3)` становится пустой

массив `Array[3]`, и на экран выдается содержимое этого пустого массива.

По-видимому, это совсем не то, что ожидалось от `display(3)`...

Вообще-то вызов `display(3)` — это ошибка, никакого смысла в таком вызове функции нет. Но компилятор эту ошибку программиста пропустил, что, безусловно, можно считать недостатком языка программирования.

В C++ этот недостаток исправлен, — можно исключить неявный вызов конструктора, используя ключевое слово `explicit` (написав его перед именем конструктора).

```
explicit Array(int size = 10);
```

`explicit` пишется только в прототипе метода, в коде реализации конструктора его повторять не надо, так и остается

```
Array::Array(int size)
```

Теперь, имея `explicit` в конструкторе, при запуске программы получаем сообщение об ошибке компиляции

```
C2664 'void display(const Array &)':  
cannot convert argument 1  
from 'int' to 'const Array &'
```

Закомментировав (или убрав) ошибочный вызов `// display(3)`, получаем работающую программу без вывода пустого массива на экран.

Использование явного (`explicit`) конструктора препятствует выполнению неявных преобразований. Явные

преобразования (через операторы явного преобразования) будут по-прежнему разрешены. То есть, при необходимости, мы вполне можем написать `display(Array(3));` и получить первоначальный результат.

При `uniform`-инициализации неявная конвертация также будет выполняться

```
// Array array10 = 10;    так нельзя — ошибка компиляции  
Array array10{ 10 };     // а так можно
```

Из всего вышесказанного следует простое и важное правило:

Для предотвращения возможных ошибок, связанных с неявным преобразованием (случайным созданием) объектов класса следует делать конструкторы с 1 (одним) параметром явными, используя ключевое слово `explicit`.

3. Необходимость использования перегрузки операторов

3.1. Примеры кода (реализация классов через обычные методы члены типа `Sum`, `Mult` и так далее)

Человеческая деятельность всегда целенаправленна. Программист пишет программу для решения какой-то задачи.

Программист создает классы в программе для того, чтобы программа была проще, понятней. Тогда в ней будет меньше ошибок, она будет быстрее написана, и, самое главное, — программу будет легче развивать и использовать для решения других задач.

Рассмотрим пример: пусть нам предстоит выполнить ряд геометрических вычислений, связанных с использованием координат точек и векторов на координатной плоскости. Конечно, мы можем хранить координаты в программе, как два массива

```
int count;  
cin >> count;  
double* x = new double[count];  
double* y = new double[count];
```

Но поскольку две координаты (x , y) представляют собой единую сущность — точку на плоскости, вернее и удобнее будет и в программе объединить две координаты в один объект.

Так что создадим и будем использовать класс `Point`

```
class Point
{
private:
    double x;
    double y;
};

int main()
{
    int count;
    cin >> count;
    Point* points = new Point[count];
}
```

Этот класс, однажды созданный, пригодится нам для решения многих задач с точками и векторами на плоскости. Однако, для этого придется предварительно поработать — написать функции, необходимые для решения этих задач. Назовем некоторые из этих функций:

- Вывод точки на экран:

```
void display() const
```

- Чтение точки:

```
void read()
```

- Сравнение двух точек (или векторов):

```
static bool isEqual(const Point& point1, const Point& point2)
```

- Сложение двух векторов:

```
static Point add(const Point& point1, const Point& point2)
```

- Умножение вектора на число:

```
static Point mult(const Point& point, double value)
```

- Расстояние между двумя точками:

```
static double distance(const Point& point1,
                      const Point& point2)
```

- Длина вектора:

```
static double length(const Point& point)
```

Тестовая программа для разработанного класса будет выглядеть следующим образом (код программы — в папке *Lesson03\les03_07*):

```
int main()
{
    Point point1(1, 1);
    Point point2;
    Point point3(1, 1);

    if (Point::isEqual(point1, point3))
    {
        cout << "point1 and point3 are equal" << endl;
    }

    cout << "p1: ";
    point1.display();
    cout << endl;

    cout << "Enter point p2 in format x,y (e.g. 12,10):";
    point2.read();

    cout << "p2: ";
```

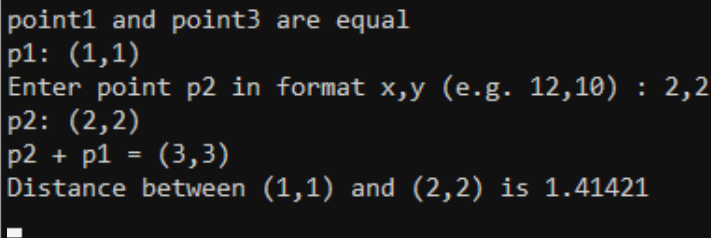
```
point2.display();
cout << endl;

cout << "p2 + p1 = ";
Point::add(point2, point1).display();
cout << endl;

cout << "Distance between ";
point1.display();
cout << " and ";
point2.display();
cout << " is ";
cout << Point::distance(point1, point2);
cout << endl;

_getch();
return 0;
}
```

Результат работы программы приведен на рисунке 4.



```
point1 and point3 are equal
p1: (1,1)
Enter point p2 in format x,y (e.g. 12,10) : 2,2
p2: (2,2)
p2 + p1 = (3,3)
Distance between (1,1) and (2,2) is 1.41421
■
```

Рисунок 4. Выполнение тестовой программы
для класса `Point`

Мы видим, что результаты работы программы выглядят хорошо, а вот о тексте программы этого сказать нельзя.

Дело в том, что задача по своей природе — математическая, а текст программы с точки зрения математика выглядит неуклюже, да и с точки зрения программиста, — текст программы не слишком удобно создавать и редактировать.

3.2. Логичность использования стандартных символов (+, —, >, < и т. д.)

Было бы лучше, если бы при написании программы, работающей с математическими объектами, использовалась запись математических выражений, и в этой записи применялись бы привычные математические операторы и привычные стандартные операторы ввода — вывода).

И такая возможность существует.

К операторам C++ можно применить известный нам аппарат перегрузки функций, и таким образом мы можем определить свои собственные версии операторов, которые будут работать с данными наших классов так, как нам это видится правильным. Примером перегрузки оператора в C++ является оператор `<<`, который используется и как оператор записи в поток, и как оператор сдвига влево.

Приведем тестовую программу с такой же функциональностью, как и в предыдущем примере, но с использованием перегруженных операторов (код программы — в папке `Lesson03\les03_08`):

```
int main()
{
    Point point1(1, 1);
    Point point2;
    Point point3(1, 1);
```

```

if (point1 == point3)
{
    cout << "point1 and point3 are equal" << endl;
}

cout << "p1: ";
point1.display();
cout << endl;

cout << "Enter point p2 in format x,y (e.g. 12,10) :";
point2.read();

cout << "p2: ";
point2.display();
cout << endl;

point3 = point1 + point2;
cout << "p1 + p2 = ";
point3.display();
cout << endl;

cout << "Distance between ";
point1.display();
cout << " and ";
point2.display();
cout << " is ";
cout << point1 % point2 << endl;

cout << "Vector ";
point1.display();
cout << " length is ";
cout << !point1 << endl;

_getch();
return 0;
}

```

Результат работы программы приведен на рисунке 5.

```
point1 and point3 are equal
p1: (1,1)
Enter point p2 in format x,y (e.g. 12,10) : 2,2
p2: (2,2)
p1 + p2 = (3,3)
Distance between (1,1) and (2,2) is 1.41421
Vector (1,1) length is 1.41421
```

Рисунок 5. Выполнение тестовой программы для класса Point с использованием перегруженных операторов

Программа стала короче и интуитивно понятней. Знаки операций `==`, `+` используются привычным образом в понятном контексте:

- `==` — сравнение переменных (объектов `Point`)

```
if (point1 == point3)
```

- `+` — операция сложения значений переменных (объектов `Point`)

```
cout << "p2 + p1 = " << point2 + point1 << endl;
```

В данном примере присутствуют и несколько неожиданные перегрузки операторов:

- `%` — расстояние между точками (объектами `Point`)

```
cout << "Distance between " << point1 << " and "
    << point2 << ": " << point1 % point2 << endl;
```


- `!` — длина вектора (объекта `Point`)

```
cout << "Vector ";  
point1.display();  
cout << " length is ";  
cout << !point1 << endl;
```

Эти варианты перегрузки приведены только для примера того, как не следует поступать.

Таких, непривычных, непонятных конструкций лучше избегать. Возможность перегрузки операторов существует не для запутывания кода, а, наоборот, для придания ему максимально стандартного вида.

4. Перегрузка операторов

4.1. Общие понятия перегрузки операторов

4.1.1. Классификация операторов на основании количества операндов (бинарные, унарные, триадный)

Оператор — это символ (иногда несколько последовательных символов), обозначающий определенную операцию над данными. В C++ имеются следующие группы операторов:

- арифметические (+, -, *, /, %, ++, --);
- операторы отношения (==, !=, >, <, >=, <=);
- логические (&&, ||, !);
- побитовые (&, |, ^, ~, <<, >>);
- операторы присваивания (=, +=, -=, *=, /=, %=, , <<=, >>=, &=, ^=, |=);
- другие операторы (sizeof, ? x : y, ,(запятая), .(точка), ->, cast, &, *).

Данные, с которыми работают, операторы, называются операндами. По количеству операндов операторы C++ делятся на три группы:

- **унарные.** Работают с одним операндом, например, унарный минус (-x) или инкремент (a++). Операнд может быть слева или справа от оператора;
- **бинарные.** Работают с двумя операндами — левым и правым, например сложение (a + b) или присваивание (a += 10);

- **триадные** (*тернарные*). Работают с тремя операндами. В языке C++ есть только один тернарный оператор ($x < y ? x : y$).

Способ реализации перегрузки оператора зависит от того, к какой из этих групп относится данный оператор.

4.1.2. Определение перегрузки операторов

Перегрузка операторов — реализация принципа полиморфизма для действий, выполняемых посредством операторов. Перегруженные операторы имеют одно и то же имя (знак операции), но работают с операндами различных типов.

При разработке класса перегрузка оператора позволяет определить действия, которые будет выполнять данный оператор для объектов этого класса.

Перегрузка подразумевает создание функции, название которой содержит слово **operator** и символ перегружаемого оператора, например, **operator+(Point p1, Point p2)**. Функция оператора может быть определена как член класса, так и вне класса.

Перегрузить можно только те операторы, которые уже определены в C++. Создать новые операторы нельзя.

Приоритет и ассоциативность всех операторов зафиксированы в языке C++ и никак не могут быть изменены посредством перегрузки.

Перегрузка также не может изменить число операндов оператора.

Нельзя перегрузить операнды встроенных типов языка C++.

Каждый оператор перегружается отдельно. Перегрузка, например, оператора `+` не обозначает автоматической перегрузки оператора `+=`.

Если функция оператора определена как отдельная функция и не является членом класса, то количество параметров такой функции совпадает с количеством операндов оператора.

У функции, которая представляет унарный оператор (например, унарный минус: `-x`), будет один параметр: `<T> operator-(T op)`.

У функции, которая представляет бинарный оператор (например, сложение: `x + y`), будет два параметра: `<T> operator+(T op1, T op1)`.

При этом первый операнд (`x`) передается первому параметру функции (`op1`), а второй операнд (`y`) — второму параметру (`op2`).

Хотя бы один из параметров функции оператора должен представлять тип класса.

Рассмотрим пример.

```
#include <iostream>
#include <conio.h>
using namespace std;

class Point
{
public:
    double x;
    double y;

    Point(double x, double y) : x{ x }, y{ y }
    {
    }
}
```

```

void display() const
{
    cout << "(" << x << "," << y << ")";
}
};

Point operator+(const Point& point1, const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}

int main()
{
    Point p1(1,1);
    Point p2(2, 2);
    Point p3 = p1 + p2;
    p3.display();    // (3, 3)

    _getch();
    return 0;
}

```

Оператор `+` перегружен для сложения объектов класса `Point` (векторов на плоскости).

Перегрузка определяется функцией

```
Point operator+(const Point& point1, const Point& point2)
```

Именем функции является `operator+`, что и указывает на назначение функции — перегрузку оператора сложения (`+`).

Функция определена вне класса, имеет два параметра типа `Point`, и возвращаемое значение типа `Point`.

При выполнении операции `p1 + p2` первое слагаемое `p1` становится первым параметром (`const Point& point1`)

функции `operator+`, а второе слагаемое `p2` становится вторым параметром (`const Point& point2`) этой функции.

В теле функции создается новый объект `Point` с координатами, равными сумме координат параметров, и этот объект возвращается в точку вызова. Таким образом реализуется сложение векторов посредством использования оператора `+`: `(p1 + p2)` — вектор, являющийся суммой векторов `p1` и `p2`.

4.1.3. Различные виды перегрузки (метод-член, функция-друг, глобальная функция)

При обработке выражения, содержащего оператор, компилятор выполняет следующие действия:

- если все операнды являются операндами встроенных типов данных, то вызывается соответствующая версия оператора

```
int a = 1;
int b = 2;
int c = a % b; // выполняется оператор % (получение остатка
               // от деления) для целых чисел;
```

- если таких перегрузок оператора нет, то компилятор выдаст ошибку;

```
string a = "1";
string b = "2";
string c = a % b; // ошибка компиляции
```

Компилятор выдает сообщение: *Error C2676 binary ‘%’: ‘std::string’ does not define this operator or a conversion to a type acceptable to the predefined operator.*

- если какой-либо из операндов является объектом класса, то компилятор будет искать версию оператора, которая работает с таким типом данных.

```
Point p1 (1, 1);
double a = 10;
Point p2 = p1 * a; // выполняется оператор *
                  // (умножение вектора на число),
                  // перегруженный в классе Point
```

- если такой перегрузки нет, то компилятор попытается выполнить преобразование пользовательских типов данных во встроенные. Если и это невозможно, — компилятор выдаст ошибку.

```
Point p10(1, 1);
double a = 10;
Point p20 = p10 % a; // ошибка компиляции
```

Ошибка C2679 бинарный "%": не найден оператор, принимающий правый операнд типа "double" (или приемлемое преобразование отсутствует)

В C++ существует три способа перегрузки операторов:

- через дружественные функции;

```
friend Point operator+(const Point& point1,
                      const Point& point2)
{
    return Point(point1.x + point2.x,
                  point1.y + point2.y);
}
```

- через обычные функции;

```
Point operator+(const Point& point1, const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}
```

- через методы класса.

```
Point operator+(const Point& point)
{
    return Point(this->x + point.x, this->y + point.y);
}
```

Некоторые операторы можно перегружать любым из этих способов, некоторые — только строго определенным способом.

Рассмотрим различные варианты перегрузки на примере бинарного оператора `+` (*плюс*) и унарного оператора `-` (*минус*) для класса `Point`. При реализации любого варианта векторы можно будет складывать и выполнять над ними операцию отрицания, например так:

```
Point p1(1,1);
Point p2(2,2);
Point p3 = p1 + p2;
p3.display();      // (3, 3)
(-p1).display();   // (-1,-1)
```

Обычная функция:

```
Point operator+(const Point& point1, const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}
```



```
Point operator-(const Point& point)
{
    return Point(-point.x, -point.y);
}
```

При перегрузке оператора через обычную функцию задаем имя функции в виде `operatorСимволОператора`. Параметры функции представляют операнды. Для бинарной операции: первый параметр — левый операнд, второй параметр — правый операнд. Для унарной операции: единственный параметр — единственный операнд. Возвращаемое значение представляет результат операции.

Недостатком глобальной перегрузки является то, что для ее реализации данные-члены класса должны быть открытыми, а это во многих случаях крайне нежелательно.

Дружественная функция:

```
friend Point operator+(const Point& point1,
                      const Point& point2)
{
    return Point(point1.x + point2.x,
                  point1.y + point2.y);
}

friend Point operator-(const Point& point)
{
    return Point(-point.x, -point.y);
}
```

Перегрузка оператора через дружественную функцию практически тождественна глобальной перегрузке,

но при этом лишена ее главного недостатка. Дружественная функция имеет доступ ко всем полям класса (`x` и `y` в этом примере могут быть закрытыми). Дружественная перегрузка во многих случаях является самым правильным вариантом перегрузки.

Метод класса:

```
Point operator+(const Point& point2)
{
    return Point(this->x + point2.x,
                  this->y + point2.y);
}

Point operator-()
{
    return Point(-this->x, -this->y);
}
```

Как мы видим, синтаксис перегрузки через метод класса отличается от синтаксиса глобальной и дружественной перегрузки.

При перегрузке бинарной операции левым операндом становится неявный объект, на который указывает скрытый указатель `*this`, правым — единственный параметр. При перегрузке унарной операции параметров нет, а операндом становится неявный объект по указателю `*this`.

Перегрузка операторов через методы класса невозможна, если левый операнд не является классом (например, `int`), или это класс, который мы не можем изменить (например, `std::ostream`).

4.2. Примеры перегрузки операторов

4.2.1. Перегрузка арифметических операторов

4.2.1.1. Перегрузка операторов $+$, $-$, $*$ и так далее

Рассмотрим перегрузку операторов на примере класса `Point` (точки или вектора на координатной плоскости).

Бинарные операторы, которые не изменяют левый операнд, лучше всего перегружать через дружественные функции, потому что:

- перегрузка через методы класса требует, чтобы первый параметр обязательно был объектом класса, что не всегда соответствует особенностям операции. Например,

```
Point point1(1, 1);
Point point2 = 10 * point1;
```

операцию `10 * point1` через методы класса перегрузить невозможно, поскольку первый операнд (`10`) — целое число, а не объект `Point`;

- перегрузка через обычную функцию требует, чтобы поля класса были открытыми, что противоречит принципу инкапсуляции ООП.

Иногда полезно иметь возможность выполнять операции, как через использование операторов, так и через вызовы функций с таким же назначением.

Поэтому напомним необходимые функции арифметических операций с векторами, а затем реализуем перегрузку арифметических операндов через вызовы

соответствующих функций. (Код программы — в папке Lesson03\les03_10):

```
// ++функции арифметических операций
static const Point add(const Point& point1,
                      const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}

static const Point subtract(const Point& point1,
                           const Point& point2)
{
    return Point(point1.x - point2.x, point1.y - point2.y);
}

static const Point mult(const Point& point, double value)
{
    return Point(point.x * value, point.y * value);
}

static const Point divide(const Point& point, double value)
{
    return Point(point.x / value, point.y / value);
}
// --функции арифметических операций

// ++операнды арифметических операций
friend const Point operator+(const Point& point1,
                             const Point& point2)
{
    return add(point1, point2);
}

friend const Point operator-(const Point& point1,
                             const Point& point2)
{

```

```

    return subtract(point1, point2);
}

friend const Point operator*(const Point& point,
                             double value)
{
    return mult(point, value);
}

friend const Point operator*(double value,
                             const Point& point)
{
    return mult(point, value);
}

friend const Point operator/(const Point& point,
                             double value)
{
    return divide(point, value);
}
// --операнды арифметических операций

// ++унарный минус
const Point operator-()
{
    return Point(-x, -y);
}
// --унарный минус

```

Унарный минус (один операнд) реализован через метод класса.

Point — это структура, передавать ее в качестве параметра следует по ссылке. А поскольку значение этого параметра функция не изменяет, то ссылка должна быть константной (**const Point& point**).

Возвращаемое значение тоже должно быть константным (`const Point`), иначе компилятор будет воспринимать, как допустимые, выражения, в которых перегружаемая операция будет находиться слева от знака присваивания, например: `point1 + point3 = point2;`.

4.2.1.2. Перегрузка инкремента и декремента

4.2.1.2.1. Цели и задачи перегрузки инкремента и декремента

Операции инкремента (увеличение значения переменной на 1) и декремента (уменьшение на 1) используются при написании программ так часто, что в C++ для них имеются специальные операторы (`++` и `--`). Причем, каждый из этих операторов имеет две версии: префикс и постфикс.

В постфиксной версии (`x++`, `y--`) сначала вычисляется значение выражения, в котором присутствует операнд, а потом уже изменяется значение операнда.

В префиксной версии (`++x`, `--y`), наоборот, сначала изменяется значение операнда, а потом уже вычисляется выражение.

Безусловно, при перегрузке этих операторов особенности постфиксной и префиксной формы должны сохраняться.

4.2.1.2.2. Синтаксис перегрузки и отличия постфиксной и префиксной формы

Операторы инкремента и декремента являются унарными и изменяют значения своих операндов, следовательно, перегрузку нужно выполнять через методы класса.

Перегрузка префиксных операторов инкремента и декремента реализуется так же, как перегрузка любых других унарных операторов:

```
Point& operator++()
{
    ++x; ++y; return *this;
}

Point& operator--()
{
    --x; --y; return *this;
}
```

Для постфиксных операторов в C++ используется специальный «фиктивный параметр». Этот фиктивный целочисленный параметр не используется в коде функции, просто его наличие указывает, что реализуется постфиксная форма операторов инкремента/декремента.

```
const Point operator++(int)
{
    Point point{ x, y }; // создается временный объект
                          // с текущими значениями
    ++(*this);           // выполняется префиксная перегрузка
    return point;        // возвращается временный объект
                          // с текущими значениями
}

const Point operator--(int)
{
    Point point{ x, y };
    --(*this);
    return point;
}
```

Для реализации перегрузки используется префиксный оператор инкремента/декремента, но возвращается первоначальное, еще не измененное значение.

При следующем же обращении к объекту доступ будет уже к присвоенному измененному значению.

4.2.2. Перегрузка операторов сравнения и логических операторов

Перегрузка операторов сравнения подобна перегрузке арифметических операторов, и также осуществляется через дружественную функцию.

```
friend bool operator==(const Point& point1,
                       const Point& point2)
{
    return point1.x == point2.x && point1.y == point2.y;
}

friend bool operator!=(const Point& point1,
                       const Point& point2)
{
    return !(point1.x == point2.x && point1.y == point2.y);
}

friend bool operator>(const Point& point1,
                     const Point& point2)
{
    return length(point1) > length(point2);
}
```

Операторы `==` и `!=` для объектов новых классов почти всегда имеют смысл. Операторы `<`, `>`, `<=`, `>=` — часто такого смысла не имеют и перегружать их тогда не следует.

Логические операторы `!`, `&&`, `||` еще реже имеют для объектов какое-то разумное применение и поэтому перегружаются еще реже.

Перегрузка операторов `&&`, `||` осуществляется через дружественную функцию, аналогично перегрузке арифметических операторов.

Перегрузка оператора `!` подобна перегрузке унарного минуса и осуществляется через функцию-член класса.

4.2.3. Возврат по ссылке. Перегрузка операторов `<<`, `>>`

Некоторые операции с объектами могут иметь форму цепочки (для бинарных операций):

```
операнд1 <оператор> операнд2 <оператор> операнд3 . . .
```

или (для унарных операций):

```
операнд <оператор> операнд <оператор> операнд . . .
```

Примером такой операции является префиксная операция инкремента:

```
int x = 1;
cout << ++(++x) << endl;    // результат: 3
```

Подобным образом должна работать и перегрузка этой операции в классе `Point`:

```
Point point1(1, 1);
cout << ++(++point1) << endl;    // результат: (3,3)
```

Для такой (корректной) реализации необходимо, чтобы результатом операции был тот же объект, который

является объектом операции. То есть, функция перегрузки должна возвращать результат по ссылке (причем, не константной!).

```
Point& operator++()
{
    ++x; ++y; return *this;
}
```

Такая же ситуация возникает при выполнении операций ввода-вывода, например:

```
cout << point1 << " " << point2 << " " << point3 << endl;
```

Приведенный выше оператор выполняет вывод значений нескольких точек (`point1`, `point2`, `point3`) в поток вывода.

Поток (`stream`) — это абстрактный объект, через который программа обменивается данными с внешним миром. Можно представить себе поток, как последовательность символов, которые появляются в одном конце потока и двигаются по мере работы программы к другому концу.

Имеются два типа потоков.

У **потока ввода** источником данных является внешнее устройство (клавиатура, файл), а принимающим объектом является программа.

У **потока вывода**, наоборот, источником является программа, а принимающим объектом — внешнее устройство (монитор, файл, принтер).

В C++ имеется ряд стандартных потоков ввода-вывода, которые предоставляются программе сразу

после ее запуска (при подключении заголовочного файла `<iostream>`). Наиболее часто используемыми являются:

- `cin` — класс, связанный со стандартным вводом (обычно это клавиатура);
- `cout` — класс, связанный со стандартным выводом (обычно это монитор).

Эти классы позволяют работать со всеми встроенными типами данных. Для организации пользовательских операций ввода-вывода необходимы дополнительные средства работы с потоками.

Таковыми средствами являются классы `istream` и `ostream`, — базовые классы ввода-вывода (см. рис. 1.6).

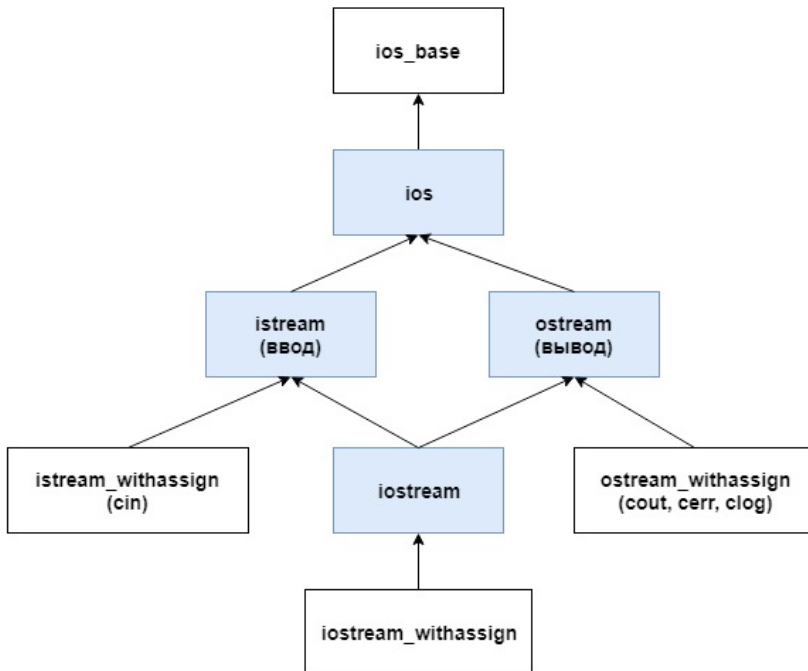


Рисунок 6. Классы ввода-вывода библиотеки `iostream`

Класс `ostream` предназначен для вывода данных и использует переопределенную операцию левого сдвига (`<<`), которую называют операцией помещения в поток.

Класс `istream` предназначен для ввода данных и использует переопределенную операцию правого сдвига (`>>`), которую называют операцией извлечения из потока.

Операции помещения и извлечения допускают цепочку вызовов в одном операторе, так как возвращают значение ссылки на поток

Классы `istream` и `ostream` перегружают операции помещения и извлечения для всех встроенных типов данных. Такая перегрузка позволяет использовать единый синтаксис для ввода и вывода символов, строк, целых и вещественных чисел.

Операции помещения/извлечения можно легко распространить на пользовательские типы данных.

Операция вывода (`cout << point1`) — это бинарная операция, в которой участвуют два операнда:

- первый операнд (`cout`) является ссылкой на поток вывода,
- второй операнд (`point`) является константной ссылкой на объект, который выводится в поток.

Результатом операции (`cout << point1`) должна быть ссылка на поток (`cout`).

```
(cout << point) << point
  |  |
  cout    << point
```

Это необходимо для того, чтобы в этот поток мог быть выведен следующий объект в цепочке вывода.

Поэтому перегрузка операций ввода-вывода должна быть реализована таким образом:

```
// ++операторы ввода-вывода
friend ostream& operator<< (ostream& output, const Point& point)
{
    output << "(" << point.x << "," << point.y << " ";
    return output;
}

friend istream& operator>> (istream& input, Point& point)
{
    input >> point.x;
    input.ignore(1);
    input >> point.y;
    return input;
}
// --операторы ввода-вывода
```

Функция `operator<<` выполняет необходимые действия по выводу элементов объекта `point`, а затем возвращает объект `output` (поток вывода) по ссылке для того, чтобы это возвращенное функцией значение могло стать первым параметром следующего вывода.

Оператор ввода (`>>`) реализуется аналогичным образом.

4.2.4. Перегрузка оператора присваивания

Оператор присваивания (`=`), в отличие от других операторов, создается компилятором в каждом разрабатываемом классе автоматически. Этот оператор по умолчанию выполняет почленное копирование переменных-членов класса.

И если в объекте класса нет ссылок на другие объекты, которые тоже должны быть скопированы, то необходимости в перегрузке оператора присваивания нет. (Класс `Point`, например, не требует перегрузки оператора присваивания).

Однако, если в классе есть указатели на динамическую память, то присваивание по умолчанию будет работать неправильно и потребуется его перегрузить.

Оператор присваивания выполняет почти те же действия, что и конструктор копирования: копирование элементов одного объекта в другой объект. Но есть и некоторые отличия:

- желательно не допускать самоприсваивания (`a = a`);
- нужно возвращать ссылку (указатель) на текущий объект, чтобы была возможность цепочки присваиваний;
- нужно корректно очистить объект, которому присваивается новое значение, для того чтобы не было утечки памяти.

Рассмотрим пример перегрузки оператора присваивания в классе с динамической памятью (*Код программы — в папке Lesson03\les03_11*).

```
#include <iostream>
#include <conio.h>
using namespace std;

class Name
{
    char* firstName;
    char* secondName;
```

```

void setCharArray(char*& dest, const char* source)
{
    int strSize = strlen(source) + 1;
    dest = new char[strSize];
    strcpy_s(dest, strSize, source);
}

void remove()
{
    if (firstName != nullptr)
    {
        delete[] firstName;
    }

    if (secondName != nullptr)
    {
        delete[] secondName;
    }
}

public:
    Name()
    {
        firstName = nullptr;
        secondName = nullptr;
    }

    Name(const char* fName, const char* sName)
    {
        setCharArray(firstName, fName);
        setCharArray(secondName, sName);
    }

    Name(const Name& name)
    {
        setCharArray(firstName, name.firstName);
        setCharArray(secondName, name.secondName);
    }

```

```

~Name()
{
    remove();
}

void write()
{
    cout << firstName << " " << secondName << endl;
}
};

void writeLine(Name name)
{
    name.write();
}

int main()
{
    setlocale(LC_ALL, "");

    char firstName[10] = "John";
    char secondName[10] = "Smith";
    {
        Name name(firstName, secondName);
        cout << "Выполнен конструктор объекта name: ";
        name.write();
    }
    cout << "Выполнен деструктор объекта name" << endl
         << endl;

    {
        Name name(firstName, secondName);
        cout << "Выполнен конструктор объекта name: ";

        writeLine(name);
        cout << "Выполнено копирование объекта name: ";
        name.write();
    }
}

```



```

    }
    cout << "Выполнен деструктор объекта name" << endl << endl;

    Name aName;
    {
        Name name(firstName, secondName);
        cout << "Выполнен конструктор объекта name: ";
        name.write();
        aName = name;
        cout << "Выполнено присваивание объекта aName: ";
        aName.write();
    }
    cout << "Выполнен деструктор объекта name" << endl << endl;
    cout << "Выполняется обращение к объекту aName: ";
    aName.write();

    _getch();
    return 0;
}

```

Результат работы программы приведен на рисунке 7.

При завершении работы программа выбрасывает исключение, как показано на рисунке 8.

```

Выполнен конструктор объекта name: John Smith
Выполнен деструктор объекта name

Выполнен конструктор объекта name: John Smith
Выполнено копирование объекта name: John Smith
Выполнен деструктор объекта name

Выполнен конструктор объекта name: John Smith
Выполнено присваивание объекта aName: John Smith
Выполнен деструктор объекта name

Выполняется обращение к объекту aName: ЭЭЭЭЭЭЭЭЭЭ ЭЭЭЭЭЭЭЭЭЭЭЭЭЭЭЭ

```

Рисунок 7. Выполнение тестовой программы для класса `Name` с использованием динамической памяти



Рисунок 8. Завершение работы тестовой программы для класса `Name`

В программе используется класс `Name`, содержащий указатели на динамические массивы символов.

```
class Name
{
    char* firstName;
    char* secondName;
}
```

В функции `main` происходит создание, обработка и уничтожение объектов этого класса

```
{
    Name name(firstName, secondName); // объект name
                                     // создается
    cout << "Выполнен конструктор объекта name: ";
    writeLine(name); // объект name копируется при
                    // передаче параметра
    cout << "Выполнено копирование объекта name: ";
    name.write();
}
```

```

} // объект name уничтожается так завершается блок,
  // в котором он был определен
cout << "Выполнен деструктор объекта name" << endl << endl;

```

Мы видим, что деструктор класса и операция копирования работают корректно. Но после присваивания объекта другому объекту

```
aName = name;
```

в работе программы возникают ошибки, и завершение программы вообще вызывает исключение (верный признак того, что программа некорректно работает с динамической памятью).

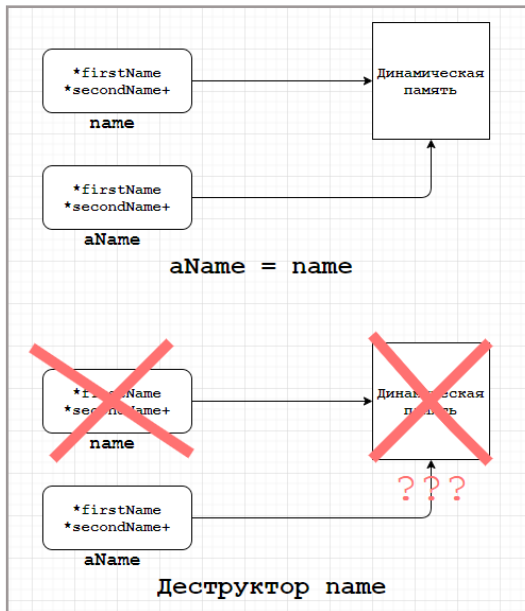


Рисунок 9. Ошибка поэлементного присваивания объектов с указателями на динамическую память

Происходит следующее: после копирования (при присваивании) объекта `name` указатели (`*firstName`, `*secondName`) объектов `name` и `aName` указывают на одну область памяти. А после уничтожения объекта `name` этой памяти больше нет и использование указателей объекта `aName` приводит к ошибке (см. рис. 9).

Поэтому в классе `Name` необходима перегрузка операции присваивания, при выполнении которой будут копироваться и области динамической памяти объектов (код программы — в папке `Lesson03/les03_12`).

```
Name& operator= (const Name& name)
{
    // обходим самокопирование
    if (this == &name)
        return *this;

    // если уже есть значение, то удаляем это значение
    remove();

    // выполняем глубокое копирование
    setCharArray(firstName, name.firstName);
    setCharArray(secondName, name.secondName);

    // Возвращаем текущий объект
    return *this;
}
```

Перегрузка оператора присваивания реализуется через функцию-член класса. Параметром функции является константная ссылка на присваиваемый объект. Возвращаемое значение — ссылка на текущий объект, получивший значение.

Если объект присваивается сам себе (`this == &name`), то просто возвращается указатель на этот объект.

Далее выполняется функция `remove()`, которая проверяет нет ли используемой динамической памяти в объекте, которому присваивается значение, и если есть, то эта память освобождается. Затем выполняется глубокое копирование (создание и копирование динамических массивов) элементов источника в текущий объект (функция `setCharArray`).

Возвращается указатель на обновленный текущий объект (`return *this;`).

Теперь программа выполняется без ошибок (см. рис. 10).

```

Выполнен конструктор объекта name: John Smith
Выполнен деструктор объекта name

Выполнен конструктор объекта name: John Smith
Выполнено копирование объекта name: John Smith
Выполнен деструктор объекта name

Выполнен конструктор объекта name: John Smith
Выполнено присваивание объекта aName: John Smith
Выполнен деструктор объекта name

Выполняется обращение к объекту aName: John Smith

```

Рисунок 10. Выполнение тестовой программы для класса `Name` с перегрузкой операции присваивания

4.2.5. Как выбирать способ перегрузки

При перегрузке операторов нужно учитывать следующие исключения и ограничения:

- Нельзя определить новый оператор, например, `operator**`.

- Нельзя переопределить операторы для операндов стандартных типов.
- Количество операндов, порядок выполнения и ассоциативность операторов изменить нельзя.
- Хотя бы один операнд должен быть типа класса, для которого мы определяем перегрузку.

Операторы, которые перегружать нельзя:

- `?:` (тернарный оператор);
- `::` (доступ к вложенным именам);
- `.` (доступ к полям);
- `.*` (доступ к полям по указателю);
- `sizeof`, `typeid` и операторы каста.

Операторы, которые можно перегрузить только в качестве методов класса:

- `=` (присваивание);
- `->` (доступ к полям по указателю);
- `()` (вызов функции);
- `[]` (доступ по индексу);
- `->*` (доступ к указателю-на-поле по указателю);
- операторы преобразования и управления памятью.

В тех случаях, когда способ перегрузки операторов можно выбирать, нужно обратить внимание на следующие моменты.

При работе с бинарными операторами, которые не изменяют левый операнд (например, `operator+()`), обычно используется перегрузка через обычную или дружественную функцию, поскольку такая перегрузка работает для

всех типов данных параметров (даже если левый операнд не является объектом класса или является объектом класса, который изменить нельзя). Перегрузка через обычную/дружественную функцию имеет дополнительное преимущество «симметрии», так как все операнды становятся явными параметрами (а не как у перегрузки через метод класса, когда левый операнд становится неявным объектом, на который указывает указатель `*this`).

При работе с бинарными операторами, которые изменяют левый операнд (например, `operator+=()`), обычно используется перегрузка через методы класса. В этих случаях левым операндом всегда является объект класса, на который указывает скрытый указатель `*this`.

Унарные операторы обычно тоже перегружаются через методы класса, так как в таком случае параметры не используются вообще.

Итак:

- Для операторов присваивания (`=`), индекса (`[]`), вызова функции (`()`) или выбора члена (`->`) используйте перегрузку через методы класса.
- Для унарных операторов используйте перегрузку через методы класса.
- Для перегрузки бинарных операторов, которые изменяют левый операнд (например, `operator+=()`) используйте перегрузку через методы класса, если это возможно.
- Для перегрузки бинарных операторов, которые не изменяют левый операнд (например, `operator+()`) используйте перегрузку через обычные/дружественные функции.

Желательно также, чтобы назначение и смысл перегруженного оператора был приближен к его стандартным применениям.

4.2.6. Пример класса с перегрузкой операторов (динамический массив Array)

При работе со статическими массивами C++ ощущается недостаток некоторых полезных операций: нет сравнения массивов, нельзя присвоить один массив другому, массив не знает свой размер и т.д.

С помощью перегрузки операторов можно достаточно просто реализовать эти полезные свойства (*код программы — в папке Lesson03\les03_13*).

Array.h: заголовочный файл класса [Array](#)

```
#include <iostream>

class Array {
    friend std::ostream& operator<<(std::ostream&,
                                    const Array&);
    friend std::istream& operator>>(std::istream&, Array&);

public:
    explicit Array(int = 10);
    Array(const Array&);
    ~Array();
    int length() const;

    const Array& operator=(const Array&);
    bool operator==(const Array&) const;
    bool operator!=(const Array& a) const {
        return !(*this == a);
    }
}
```



```

int& operator[](int);
int operator[](int) const;

private:
    int size;
    int* arr;
};

```

В классе предусмотрен конструктор массива с заданием размера массива, конструктор по умолчанию, конструктор копирования.

Перегружены операторы сравнения (`==`, `!=`), присвоения (`=`), индексирования (`[]`), ввода-вывода (`<<`, `>>`).

Для индексирования предусмотрены две версии перегрузки: обычная и константная.

Ввод-вывод реализован через дружественные функции.

Array.cpp: файл реализации класса *Array*

```

#include <iostream>
#include <iomanip>
#include <stdexcept>
#include "Array.h"
using namespace std;

// конструктор (по умолчанию – 10 элементов)
Array::Array(int aSize)
    : size{aSize}, arr{new int[size]{} }
{
}

// конструктор копирования
Array::Array(const Array& a)
    : size{a.size}, arr{new int[size]}
{
}

```

```
    for (int i = 0; i < size; ++i)
    {
        arr[i] = a.arr[i];
    }
}

// деструктор
Array::~Array()
{
    delete[] arr;
}

// размер файла
int Array::length() const
{
    return size;
}

// оператор присвоения
const Array& Array::operator=(const Array& a)
{
    if (&a != this)
    {
        if (size != a.size)
        {
            delete[] arr;
            size = a.size;
            arr = new int[size];
        }
        for (int i = 0; i < size; ++i)
        {
            arr[i] = a.arr[i];
        }
    }

    return *this;
}
```

```

// оператор сравнения
bool Array::operator==(const Array& a) const
{
    if (size != a.size)
    {
        return false;
    }
    for (int i{0}; i < size; ++i) {
        if (arr[i] != a.arr[i]) {
            return false;
        }
    }
    return true;
}

// оператор индексирования
int& Array::operator[](int index)
{
    if (index < 0 || index >= size)
    {
        cout << "Out of range" << endl;
        _getch();
        exit(1);
    }
    return arr[index];
}

// оператор индексирования (get)
int Array::operator[](int index) const
{
    if (index < 0 || index >= size)
    {
        cout << "Out of range" << endl;
        _getch();
        exit(1);
    }
}

```

```

    return arr[index];
}

// оператор ввода
istream& operator>>(istream& input, Array& a)
{
    for (size_t i{0}; i < a.size; ++i) {
        input >> a.arr[i];
    }
    return input;
}

// оператор вывода
ostream& operator<<(ostream& output, const Array& a)
{
    for (int i{0}; i < a.size; ++i) {
        output << a.arr[i] << " ";
    }
    output << endl;
    return output;
}

```

Операторы класса `Array` перегружены стандартными для таких операторов способами.

При выходе индекса за границы массива выдается сообщение “Out of range” (*За пределами массива*) и прекращается работа программы.

`les03_13.cpp`: файл тестовой программы для класса `Array`

```

#include <iostream>
#include <conio.h>
#include "Array.h"
using namespace std;

```

```

int main() {
    Array array1{ 5 };
    Array array2;

    for (int i = 0; i < 5; i++)
    {
        array1[i] = i;
    }

    for (int i = 0; i < 10; i++)
    {
        array2[i] = i + 11;
    }

    cout << "Size of array1: " << array1.length() << endl;
    cout << "Array1: " << array1 << endl;

    cout << "Size of array2: " << array2.length() << endl;
    cout << "Array2: " << array2 << endl;

    cout << "(array1 == array2) ?" << endl;
    if (array1 == array2)
    {
        cout << "array1 == array2" << endl << endl;;
    }
    else
    {
        cout << "array1 != array2" << endl << endl;;
    }

    cout << "Array array3{ array1 }; // copy constructor"
        << endl;
    Array array3{ array1 }; // конструктор копирования
    cout << "Size of array3: " << array3.length() << endl;
    cout << "Array3: " << array3 << endl;

    cout << "array1 = array2; // assignment operator" << endl;

```

```

array1 = array2;           // операция присваивания
cout << "Array1: " << array1;
cout << "Array2: " << array2 << endl;
cout << "array2[5] = 1000;" << endl;
array2[5] = 1000;
cout << "Array2: " << array2 << endl;
cout << "array2[15] = 1000;" << endl;
array2[15] = 1000; // ошибка: индекс за пределами массива

_getch();
return 0;
}

```

На рисунке 1.11 приведен результат работы тестовой программы для класса `Array`.

```

Size of array1: 5
Array1: 0  1  2  3  4

Size of array2: 10
Array2: 11 12 13 14 15 16 17 18 19 20

(array1 == array2) ?
array1 != array2

Array array3{ array1 }; // copy constructor
Size of array3: 5
Array3: 0  1  2  3  4

array1 = array2;           // assignment operator
Array1: 11 12 13 14 15 16 17 18 19 20
Array2: 11 12 13 14 15 16 17 18 19 20

array2[5] = 1000;
Array2: 11 12 13 14 15 1000 17 18 19 20

array2[15] = 1000;
Out of range

```

Рисунок 11. Рисунок 1.11. Выполнение тестовой программы для класса `Array`

5. Резюме

Раздел 1. Константный метод

- Ключевое слово `const` при определении объекта указывает, что объект является неизменяемым, и что любая попытка изменения этого объекта является ошибкой.
- Значение константному объекту не может присваиваться, следовательно, такой объект может получить значение только путем инициализации.
- Вызов метода, который изменяет значение объекта, для константных объектов приводит к ошибке компиляции.
- Ключевое слово `const` — после закрывающей круглой скобки в заголовке функции, например, `int getDay() const`, обозначает, что функция не изменяет значения объекта. Такая функция называется константной функцией-членом класса или константным методом. Константные методы (и только они) могут вызываться для константных объектов.
- Любое изменение объекта в коде константного метода приводит к ошибке компиляции.
- Метод указывается как константный и в прототипе, и в определении метода.
- В классе можно перегрузить функцию таким образом, чтобы иметь константную и неконстантную версии этой функции одновременно. Это имеет смысл и даже необходимо, например, при перегруз-

ке оператора `[]`, где нужны одновременно и неконстантная перегрузка для чтения и записи (`item[2] = 10;`), и константная перегрузка для константных объектов (`cout << itemC[2];`).

- Конструкторы и деструкторы не могут объявляться как `const`.
- Константные поля класса и поля, являющиеся ссылками, должны инициализироваться с помощью инициализаторов этих полей.
- Все методы, которые выдают какие-либо характеристики объекта, не изменяя при этом значение объекта, должны быть описаны как константные. В противном случае их нельзя будет применить для константных объектов или даже для нормальных объектов, доступ к которым осуществляется через константный указатель или ссылку. Отсутствие `const` в описании таких методов является неправильным стилем программирования на C++.

Раздел 2. Объявление конструктора с использованием ключевого слова `explicit`

- Конструктор с одним параметром может вызываться компилятором для выполнения неявного преобразования.
- Например, при вызове функции `displayDate(2020)` — на месте целого значения (2020) автоматически создается объект типа `Date` посредством выполнения конструктора `Date(2020)`.
- В некоторых ситуациях вызов конструктора для неявного преобразования является ошибкой.

Например:

```
Array array(size); // конструктор массива размером size
display(array);    // функция вывода массива
display(3);        // неявно вызывается конструктор array(3)
// и на экран выводится пустой массив из 3-х элементов
```

- Ключевое слово **explicit**, не разрешает неявные преобразования посредством использования конструкторов с одним параметром. Конструктор, объявленный как **explicit**, не может быть использован в неявном преобразовании.
- Объявление конструктора с ключевым словом **explicit** желательно делать всегда для конструкторов, у которых 1(один) параметр.

Раздел 3. Необходимость использования перегрузки операторов

- К операторам C++ можно применить известный нам аппарат перегрузки функций, и таким образом мы можем определить свои собственные версии операторов, которые будут работать с данными наших классов так, как нам это видится правильным.
- Примером перегрузки оператора в C++ является оператор **<<**, который используется и как оператор записи в поток, и как оператор сдвига влево.
- Действия, выполняемые перегруженными операторами, можно реализовать посредством создания и вызовов соответствующих функций, но запись этих действий с помощью операторов часто выглядит понятнее и короче.

- Оператор — это символ (иногда несколько последовательных символов), обозначающий определенную операцию над данными. Данные, с которыми работают, операторы, называются операндами.
- По количеству операндов операторы C++ делятся на три группы:
 - ▷ унарные,
 - ▷ бинарные,
 - ▷ тернарные.
- Унарные операторы работают с одним операндом, например, унарный минус ($-x$) или инкремент ($a++$). Операнд может быть слева или справа от оператора;
- Бинарные операторы работают с двумя операндами — левым и правым, например сложение ($a + b$) или присваивание ($a += 10$);
- Тернарные операторы работают с тремя операндами. В языке C++ есть только один тернарный оператор ($x < y ? x : y$).
- Способ реализации перегрузки оператора зависит от того, к какой из групп (бинарные или унарные) относится данный оператор.
- Перегрузка подразумевает создание функции, название которой содержит слово `operator` и символ перегружаемого оператора, например, `operator+(Point p1, Point p2)`. Функция оператора может быть определена как член класса, так и вне класса.
- Перегрузить можно только те операторы, которые уже определены в C++. Создать новые операторы нельзя.

- Приоритет и ассоциативность всех операторов зафиксированы в языке C++ и никак не могут быть изменены посредством перегрузки.
- Перегрузка также не может изменить число операндов оператора.
- Нельзя перегрузить операнды встроенных типов языка C++.
- Каждый оператор перегружается отдельно. Так, например, перегрузка оператора `+` не обозначает автоматической перегрузки оператора `+=`.
- Если функция оператора определена как отдельная функция и не является членом класса, то количество параметров такой функции совпадает с количеством операндов оператора.
- У функции, которая представляет унарный оператор (например, унарный минус: `-x`), будет один параметр: `<T> operator!(T op)`
- У функции, которая представляет бинарный оператор (например, сложение: `x + y`), будет два параметра: `<T> operator+(T op1, T op1)`.
- При этом первый операнд (`x`) передается первому параметру функции (`op1`), а второй операнд (`y`) — второму параметру (`op2`).
- Хотя бы один из параметров функции оператора должен представлять тип класса.
- В C++ существует три способа перегрузки операторов:
 - ▷ через обычные функции;
 - ▷ через дружественные функции;
 - ▷ через методы класса.

- При перегрузке оператора через обычную функцию задаем имя функции в виде `operatorСимволОператора`. Параметры функции представляют операнды.
- Для бинарной операции: первый параметр — левый операнд, второй параметр — правый операнд.
- Для унарной операции: единственный параметр — единственный операнд. Возвращаемое значение представляет результат операции.
- Недостатком глобальной перегрузки является то, что для ее реализации данные-члены класса должны быть открытыми, а это во многих случаях крайне нежелательно.
- Функция-член класса при перегрузке бинарной операции обращается к левому операнду через неявный указатель `this`, правый операнд — параметр такой функции.
- Перегрузка оператора через дружественную функцию практически тождественна глобальной перегрузке, но при этом лишена ее главного недостатка. Дружественная функция имеет доступ ко всем полям класса. Дружественная перегрузка во многих случаях является самым правильным вариантом перегрузки.
- Синтаксис перегрузки через метод класса отличается от синтаксиса глобальной и дружественной перегрузки.
При перегрузке бинарной операции левым операндом становится неявный объект, на который указывает скрытый указатель `*this`, правым — единственный параметр.

При перегрузке унарной операции параметров нет, а операндом становится неявный объект по указателю `*this`.

- Перегрузка операторов через методы класса невозможна, если левый операнд не является классом (например, `int`) или если это класс, который мы не можем изменить (например, `std::ostream`).
- Параметры-объекты в функцию перегрузки передаются по ссылке или по константной ссылке.
- Возвращаемое значение часто тоже должно быть константным (`const Point`), иначе компилятор будет воспринимать, как допустимые, выражения, в которых перегружаемая операция будет находиться слева от знака присваивания, например: `point1 + point3 = point2`;
- Операторы инкремента и декремента являются унарными и изменяют значения своих операндов, следовательно, перегрузку нужно выполнять через методы класса.
- Перегрузка префиксных операторов инкремента и декремента реализуется так же, как перегрузка любых других унарных операторов
- Для постфиксных операторов в C++ используется специальный «фиктивный параметр»: `operator++(int)`. Этот фиктивный целочисленный параметр не используется в коде функции, просто его наличие указывает, что реализуется постфиксная форма операторов инкремента/декремента.

- Для реализации перегрузки постфиксных операторов используется префиксный оператор инкремента/декремента, но возвращается первоначальное, еще не измененное значение.
При следующем же обращении к объекту доступ будет уже к присвоенному измененному значению.
- Перегрузка операторов сравнения подобна перегрузке арифметических операторов, и также осуществляется через дружественную функцию.
- Операторы `==` и `!=` для объектов новых классов почти всегда имеют смысл. Операторы `<`, `>`, `<=`, `>=` — часто такого смысла не имеют и перегружать их тогда не следует.
- Логические операторы `!`, `&&`, `||` еще реже имеют для объектов какое-то разумное применение и поэтому перегружаются еще реже.
- Перегрузка операторов `&&`, `||` осуществляется через дружественную функцию, аналогично перегрузке арифметических операторов.
- Перегрузка оператора `!` подобна перегрузке унарного минуса и осуществляется через функцию-член класса.
- Некоторые операции с объектами могут иметь форму цепочки (для бинарных операций):

```
операнд1 <оператор> операнд2 <оператор> операнд3 . . .
(cout << a << b<<c)
```

или (для унарных операций):

```
операнд <оператор> операнд <оператор> операнд . . .
++(++x)
```

Для реализации таких операций необходимо, чтобы результатом операции был тот же объект, который является объектом операции.

То есть, функция перегрузки должна возвращать результат по ссылке (причем, не константной!).

- Операция вывода (`cout << object`) — это бинарная операция, в которой участвуют два операнда:
 - ▷ первый операнд (`cout`) является ссылкой на поток вывода,
 - ▷ второй операнд (`object`) является константной ссылкой на объект, который выводится в поток.
 - ▷ Результатом операции (`cout << object`) должна быть ссылка на поток (`cout`).

```
(cout << point) << object
| |
cout      << object
```

Это необходимо для того, чтобы в этот поток мог быть выведен следующий объект в цепочке вывода. Поэтому перегрузка операций ввода-вывода должна быть реализована примерно таким образом:

```
friend ostream& operator<< (ostream& output, const T& object)
{
    output << object.x << " " << object.y << endl;
    return output;
}
```

- Оператор присваивания (`=`), в отличие от других операторов, создается компилятором в каждом разрабатываемом классе автоматически. Этот оператор по умолчанию выполняет почленное копирование переменных-членов класса. И если в объекте класса нет ссылок на другие объекты, которые тоже должны быть скопированы, то необходимости в перегрузке оператора присваивания нет.
- Однако, если в классе есть указатели на динамическую память, то присваивание по умолчанию будет работать неправильно и потребуются его перегрузить.
- Оператор присваивания выполняет почти те же действия, что и конструктор копирования: копирование элементов одного объекта в другой объект. Но есть и некоторые отличия:
 - ▷ желательно не допускать самоприсваивания (`a = a`);
 - ▷ нужно возвращать ссылку (указатель) на текущий объект, чтобы была возможность цепочки присваиваний;
 - ▷ нужно корректно очистить объект, которому присваивается новое значение, для того чтобы не было утечки памяти.
- Перегрузка оператора присваивания реализуется через функцию-член класса. Параметром функции является константная ссылка на присваиваемый объект. Возвращаемое значение — ссылка на текущий объект, получивший значение.
- При перегрузке операторов нужно учитывать следующие исключения и ограничения:

- ▷ Нельзя определить новый оператор, например, `operator**`.
- ▷ Нельзя переопределить операторы для операндов стандартных типов.
- ▷ Количество операндов, порядок выполнения и ассоциативность операторов изменить нельзя.
- ▷ Хотя бы один операнд должен быть типа класса, для которого мы определяем перегрузку.
- Операторы, которые перегружать нельзя:
 - ▷ `?:` (тернарный оператор);
 - ▷ `::` (доступ к вложенным именам);
 - ▷ `.` (доступ к полям);
 - ▷ `.*` (доступ к полям по указателю);
 - ▷ `sizeof`, `typeid` и операторы каста.
- Операторы, которые можно перегрузить только в качестве методов класса:
 - ▷ `=` (присваивание);
 - ▷ `->` (доступ к полям по указателю);
 - ▷ `()` (вызов функции);
 - ▷ `[]` (доступ по индексу);
 - ▷ `->*` (доступ к указателю-на-поле по указателю);
 - ▷ операторы преобразования и управления памятью.
- Бинарный операнд может быть перегружен функцией-членом класса с одним параметром или глобальной функцией с двумя параметрами (один из которых должен быть либо объектом класса, либо ссылкой на объект класса).
- Функции-члены, реализующие перегрузку операторов, не могут быть статическими, так как им требуется обращаться к нестатическим данным класса.

- Оператор индексации (`operator[]`) должен иметь две версии перегрузки: константную (константный метод) и не константную.

6. Домашнее задание

6.1. Создать класс «множество целых чисел»

Множество — это одно из самых базовых понятий в математике.

Множество представляет собой набор каких-либо объектов, которые называются элементами множества. Как правило, каждый элемент в множестве встречается только один раз, и эти элементы не упорядочены.

Часто множество определяется просто перечислением его элементов.

Множество целых чисел A может быть задано, например, так:

$$A = \{3, 8, 46, 5, 11\}$$

А другое множество целых чисел B может быть задано, например, так:

$$B = \{18, 8, 90, 11, 2\}$$

Множество — это набор элементов, коллекция. Другими представителями коллекций в C++ являются массивы, векторы, стеки.

У множества, как и у других видов коллекций, есть свой набор операций.

К таким операциям относятся:

- добавление и удаление элементов в множество

$$\{3, 8, 46, 5, 11\} + 4 = \{3, 8, 46, 5, 11, 4\}$$

$$\{3, 8, 46, 5, 11\} + 3 = \{3, 8, 46, 5, 11\} \text{ — (элементы множества не повторяются!);}$$

- сравнение множеств (множества равны, если они содержат одинаковый набор элементов);
- специальные операции с множествами:
 - ▷ объединение множеств (математическое обозначение: $A \cup B$),

$$\{3, 8, 46, 5, 11\} \cup \{18, 8, 90, 11, 2\} = \{3, 8, 46, 5, 11, 18, 90, 2\}$$

(все элементы A + все элементы B),

- ▷ пересечение множеств (математическое обозначение: $A \cap B$)

$$\{3, 8, 46, 5, 11\} \cap \{18, 8, 90, 11, 2\} = \{8, 11\}$$

(элементы, которые содержатся и в A, и в B),

- ▷ разность множеств (математическое обозначение: $A \setminus B$)

$$\{3, 8, 46, 5, 11\} \setminus \{18, 8, 90, 11, 2\} = \{3, 46, 5\}$$

(элементы, которые содержатся в A, но не содержатся в B),

В создаваемом классе «Множество целых чисел»:

- элементы множества хранить в динамическом массиве;
- каждый элемент множества уникален (элементы не повторяются);
- элементы не упорядочены.

Реализовать методы:

- конструкторы (по умолчанию, с параметрами, копирования);
- деструктор;
- проверка принадлежности элемента множеству.

Реализовать операции:

- добавление элемента в множество (+, +=);
- объединение двух множеств (+, +=);
- удаление элемента (-, -=);
- разность множеств (-, -=);
- пересечение множеств (*, *=);
- присваивание (=);
- сравнения множеств (==);
- потоковый вывод и ввод (<<, >>).

7. Терминология

- `*this`
- `const`
- `explicit`
- `istream`
- `operator`
- `operator!`
- `operator!=`
- `operator()`
- `operator[]`
- `operator+`
- `operator++`
- `operator++(int)`
- `operator+=`
- `operator-`
- `operator--`
- `operator-=`
- `operator<`
- `operator<=`
- `operator=`
- `operator==`
- `operator>`
- `operator>=`
- `operator>>`
- `operator<<`
- `ostream`
- `stream`
- бинарный оператор
- константная функция-член класса
- константный метод

- константный объект
- конструктор копии
- конструктор преобразования
- конструктор с одним аргументом
- неявный вызов конструктора
- оператор вызова функции ()
- оператор преобразования
- перегруженный оператор
- перегрузка оператора дружественной функцией
- перегрузка оператора глобальной функцией
- перегрузка оператора функцией-членом класса
- триадный (тернарный) оператор
- унарный оператор