



**Классы.
Дружественные
функции. Перегрузка
операторов.**

Друзья классов

- Дружественная функция – это функция, которая, не являясь компонентом класса, имеет доступ к его защищенным и собственным компонентам.
- Такая функция должна быть описана в теле класса со спецификатором `friend`

Пример использова ния friend - функции

```
class pair
{
private:
    int x,y;
    friend void set(pair*,int,int);
public:

    pair(int x1,int y1){x = x1; y = y1;}

    int sum(void){return (x+y);}

};
void set(pair *p,int x1,int y1)
{
    p->x = x1;
    p->y = y1;
}
```

Пример использова ния friend - функции

```
int main(void)
{
    pair A(5,6);
    pair B(7,8);

    cout<<A.sum()<<endl;
    cout<<B.sum()<<endl;

    set(&A,9,10);
    set(&B,11,12);

    cout<<A.sum()<<endl;
    cout<<B.sum()<<endl;

    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

11
15
19
23

friend - функции

-Функция `set` описана в классе `pair` как дружественная и определена как **обычная глобальная** функция (вне класса, без указания его имени, без операции `::` и без спецификатора `friend`).

-Дружественная функция при вызове не получает указателя **this**.

-Объекты класса должны передаваться дружественной функции только через **параметр**.

По **ссылке**, по **значению**, по **адресу**.

friend -
функции

Итак, **дружественная функция**:

- **не может быть компонентной** функцией того класса, по отношению к которому определяется как дружественная;
- может быть **глобальной** функцией;
- может быть **компонентной** функцией **другого** ранее определенного **класса**.

friend – “МЕТОДЫ”

```
class B;  
  
class A  
{  
private:  
    string name;  
public:  
    void f(B&);  
    A(const string& name){this->name = name;};  
};  
  
class B  
{  
private:  
    friend void A::f(B&);  
    string name;  
public:  
    B(const string& name){this->name = name;};  
};
```

friend – “МЕТОДЫ”

```
void A::f(B& b)
{
    cout << "Метод-компонент " << this->name;
    cout << " Друг " << b.name;
}
int main()
{
    setlocale(LC_ALL, "Russian");
    A a("class A");
    B b("class B");
    a.f(b);
};
```


friend –
“методы”

```
C:\WINDOWS\system32\cmd.exe
Метод-компонент class A Друг class B
Для продолжения нажмите любую клавишу . . .
```

В этом примере **класс A** при помощи своего метода **void A::f()** **получает доступ** к закрытым полям **класса B**.

friend - функции

-Может быть дружественной по отношению к нескольким классам.

Например:

```
// предварительное неполное определение  
класса
```

```
class CL2;
```

```
class CL1 {friend void f(CL1,CL2); ... };
```

```
class CL2 {friend void f(CL1,CL2); ... };
```

В этом примере функция имеет доступ к компонентам классов CL1 и CL2.

Friend - классы

- Класс может **быть дружественным** другому классу.
- Это означает, что **все методы** класса являются дружественными для другого класса.
- Дружественный класс должен **быть определен вне тела** класса, «предоставляющего дружбу».

Friend - классы

Например:

```
class X2{friend class X1; . . .};
```

```
class X1 { . . . void f1(. . .);
```

```
void f2(. . .); . . . };
```

- В этом примере функции f1 и f2 класса X1 являются друзьями класса X2, хотя они описываются без спецификатора friend.

Friend - классы

Рассмотрим класс `point` – точка в n-мерном пространстве и дружественный ему класс `vector`

В классе `vector` определим метод для определения квадратичной длины вектора, которая вычисляется как **сумма квадратов** его координат.

Friend - классы

```
#include <iostream>
using std::cout;

class point
{
    friend class vector;
private:
    int N;
    double *x;
public:
    point(int n, double d = 0.0);
    ~point();
};
```

Friend - классы

```
point::point(int n, double d)
{
    this->N = n;
    this->x = new double[N];
    for(int i = 0; i < N; i++)
    {
        this->x[i] = d;
    };
};

point::~~point()
{
    delete[] this->x;
};
```

Friend - классы

```
class vector
{
private:
    double *xv;
    int N;
public:
    vector(const point&,const point&);
    ~vector();
    double SquareLen();
};
```


Friend - классы

```
vector::vector(const point& begin, const point& end)
{
    this->N = begin.N;
    this->xv = new double[N];
    for(int i = 0; i < N; i++)
    {
        xv[i] = end.x[i] - begin.x[i];
    };
};

vector::~~vector()
{
    delete[] this->xv;
};
```

Friend - классы

```
double vector::SquareLen()  
{  
    double res = 0.0;  
    for(int i = 0; i < N; i++)  
    {  
        res += xv[i]*xv[i];  
    };  
    return res;  
};
```

Friend - классы

```
int main()
{
    point A(2,4.0);
    point B(2,2.0);

    vector V(A,B);

    cout << V.SquareLen();

    return 0;
} // Будет выведено – 8.
```

ПЕРЕГРУЗКА ОПЕРАТОРОВ

- В языке C++ определены множества операций над переменными стандартных типов, такие как +, *, / и т.д. Каждую операцию можно применить к операндам определенного типа.

Как быть, если необходимо, чтобы операторы +, *, / и т.д. могли **совершать действия** над объектами классов?

Есть решение – необходимо **использовать перегрузку операторов**.

ПЕРЕГРУЗКА ОПЕРАТОРОВ

Перегрузка операторов позволяет определить действия, которые будет выполнять оператор.

Перегрузка подразумевает создание функции, название которой содержит слово **operator** и символ перегружаемого оператора.

Функция оператора может быть определена как **член класса**, либо вне класса, возможно, как **дружественная функция**.

ПЕРЕГРУЗКА ОПЕРАТОРОВ

Определение оператора-функции имеет следующий синтаксис:

```
тип operator “знак оператора” (параметры)
{
    Действия...
    return тип();
}
```

ПЕРЕГРУЗКА УНАРНЫХ ОПЕРАТОРОВ

Любой унарный оператор[⊕] может быть определен двумя способами:

- как **компонентная функция без параметров**
- как **глобальная (возможно, дружественная) функция с одним параметром.**
- В первом случае выражение $\oplus Z$ означает вызов `Z.operator⊕()`, во втором – вызов `operator⊕(Z)`

Два случая перегрузки унарных операторов

Где `class&` - передача значения по ссылке

А) как **компонентная** функция

`тип operator` “знак оператора” (`void`)

```
{  
    Действия...  
}
```

Б) как **глобальная** функция

`тип operator` “знак оператора” (`class&`)

```
{  
    Действия...  
    return тип();  
}
```


ПЕРЕГРУЗКА УНАРНЫХ ОПЕРАТОРОВ

Унарные операции **инкремента** ++ и **декремента** – существуют в двух формах: **префиксной** и **постфиксной**

Они определены следующим образом:

– префиксная форма:

`operator++();`

`operator--();`

– постфиксная форма:

`operator++(int);`

`operator--(int);`

ПЕРЕГРУЗКА УНАРНЫХ ОПЕРАТОРОВ

```
#include <iostream>
using std::cout;
class person
{
    int age;

public:
    person(int age){this->age = age;}

    void HowOld()
    {
        cout << this->age;
    }
    //префиксная форма
    void operator++()
    {
        ++ this->age;
    }
};
```

ПЕРЕГРУЗКА УНАРНЫХ ОПЕРАТОРОВ

```
#include <iostream>
using std::cout;
class person
{
    int age;
public:
    person(int age){this->age = age;}

    void HowOld(){cout << this->age;}
    //префиксная форма
    friend void operator++(person& p);
};
void operator++(person& p)
{
    ++p.age;
};
```

ПЕРЕГРУЗКА УНАРНЫХ ОПЕРАТОРОВ

```
int main()
{
    class person bob(21);
    ++bob;

    //Выводит 22
    bob.HowOld();
    return 0;
};
```

```

class number
{
    int x;
public:
    int get() {return this->x;}

    number(const int& x){this->x = x;}
    //постфиксная форма
    number operator++(int)
    {
        number tmp = *this;
        this->x++;
        return tmp;
    }
};

```

```

int main()
{
    number a(11);
    cout<<(a++).get() << endl;
    cout << a.get() << endl;
    return 0;
}

```

C:\WINDOWS\system32\cmd.exe

```

11
12

```

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

Любая бинарная операция \oplus может быть определена двумя способами:

-как **компонентная** функция с одним параметром

-как **глобальная** функция с двумя параметрами.

В первом случае $x \oplus y$ означает вызов $x.operator^{\oplus}(y)$, во втором – вызов $operator^{\oplus}(x,y)$.

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

- При перегрузке операторов стоит учитывать приоритет и ассоциативность.
- *Приоритет* операторов задает порядок операций в выражениях, содержащих более одного оператора.
- *Ассоциативность* операторов указывает, будет ли операнд в выражении, содержащем несколько операторов с одинаковым приоритетом, группироваться по левому краю или по правому.
- (порядок выполнения)

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

- К примеру оператор `<<` имеет высокий приоритет и ассоциативен слева направо.
- Поэтому при перегрузке оператора `<<` для работы с потоковыми объектами возвращаем ссылку на поток.

`cout << A << B << C;`

1 - cout

2 - cout

3 - cout

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

Допустим, у нас есть класс `point` – точка в двумерном пространстве. Хотим обращаться к координатам точки через `operator[]`.

хотим сделать так, чтобы была возможность выводить в консоль координаты точки привычным образом. (`cout << point`)

Для этого перегрузим `operator[]` как нестатическую **компонентную функцию** класса

И `operator<<` как **дружественную глобальную функцию**

```
class point
{
private:
    int x;
    int y;
public:
    point(int x = 0, int y = 0);

    //перегрузка оператора индексации
    int& operator[](const char& coord);

    //перегрузка оператора << для работы с потоковым классом вывода
    //Левый операнд - объект потокового класса, правый - class point
    friend ostream& operator<<(ostream& stream, const point& p);
};
```

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

```
1 //конструктор класса
2 point::point(int x, int y)
3 {
4     this->x = x;
5     this->y = y;
6 };
7 //перегрузка оператора индексации
8 //как нестатической компонентной функции
9 int& point::operator[](const char& coord)
10 {
11     if (coord == 'x')
12     {
13         return this->x;
14     }
15     else if(coord == 'y')
16     {
17         return this->y;
18     };
19 };
```

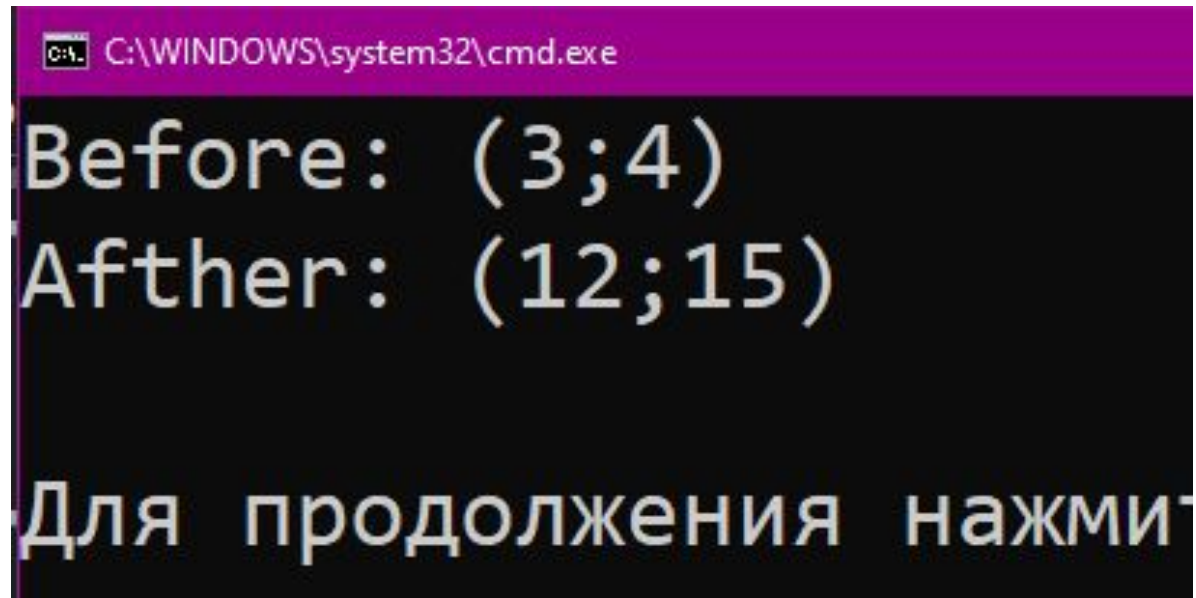
```
ostream& operator<<(ostream& stream, const point& p)
{
    stream << '(' <<p.x<< '; ' <<p.y<< ')';
    return stream;
};
```

```
int main()
{
    point a(3,4);
    cout << "Before: " << a << endl;

    a['x'] = 12;
    a['y'] = 15;

    cout << "After: " << a << endl;

    return 0;
}
```



```
C:\WINDOWS\system32\cmd.exe
Before: (3;4)
After: (12;15)
Для продолжения нажми
```

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

Перегрузка **operator=** - оператора присваивания.

Операция отличается тремя особенностями:

- Оператор **не наследуется**;
- Оператор **определен по умолчанию** для каждого класса в качестве операции побайтного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева;
- Операция может перегружаться **только** как **функция компонент** класса.

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

Когда следует перегружать оператор присваивания?

В большинстве случаев перегрузка не требуется.

НО бывают случаи, когда побайтное копирование нежелательно. (использование оператора копирования без перегрузки – побайтное копирование).

Например, если некоторый класс содержит **указатели на динамическую область памяти**, то после поразрядного копирования выйдет так, что два экземпляра будут владеть одним указателем.

Что вызывает некоторые **проблемы**.

```

class Array
{
    int size;
    int* mas;
public:
    Array(const int& size)
    {
        this->size = size;
        this->mas = new int[size];
    };
    ~Array()
    {
        delete[] this->mas;
    }

    int& operator[](const int& index)
    {
        return *(this->mas + index);
    };

    int Size() const
    {
        return this->size;
    };
};

```

```

int main()
{
    //Массив на 10 элементов
    Array a(10);
    //Второй массив
    Array b(5);
    cout << "Pointer b.mas = ";
    cout << &b[0] << endl;
    b = a;
    cout << "Afther: ";
    cout << &a[0] << ' ' << &b[0];
}

```

C:\WINDOWS\system32\cmd.exe

```

Pointer b.mas = 0x15bec0
Afther: 0x15be70 0x15be70

```

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

-После побайтного копирования `b.mas` и `a.mas` указывают на одну область памяти.

-Память, которая выделилась под массив `b` не будет очищена.

-По завершении программы у объектов вызовутся деструкторы, что означает очищение одной и той же области памяти два раза.

-может вызвать крах программы.

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

Выход – перегрузить оператор

присваивания

```
    Array& operator=(const Array& a);  
};
```

```
Array& Array::operator=(const Array& a)
```

```
{  
    delete this->mas;  
    this->size = a.size;  
    this->mas = new int[a.size];  
    for (int i = 0; i < a.size; i++)  
    {  
        this->mas[i] = a.mas[i];  
    };  
    return *this;  
};
```

```
C:\WINDOWS\system32\cmd.exe
```

```
Pointer b.mas = 0x8cbec0
```

```
After: 0x8cbe70 0x8cbf00
```

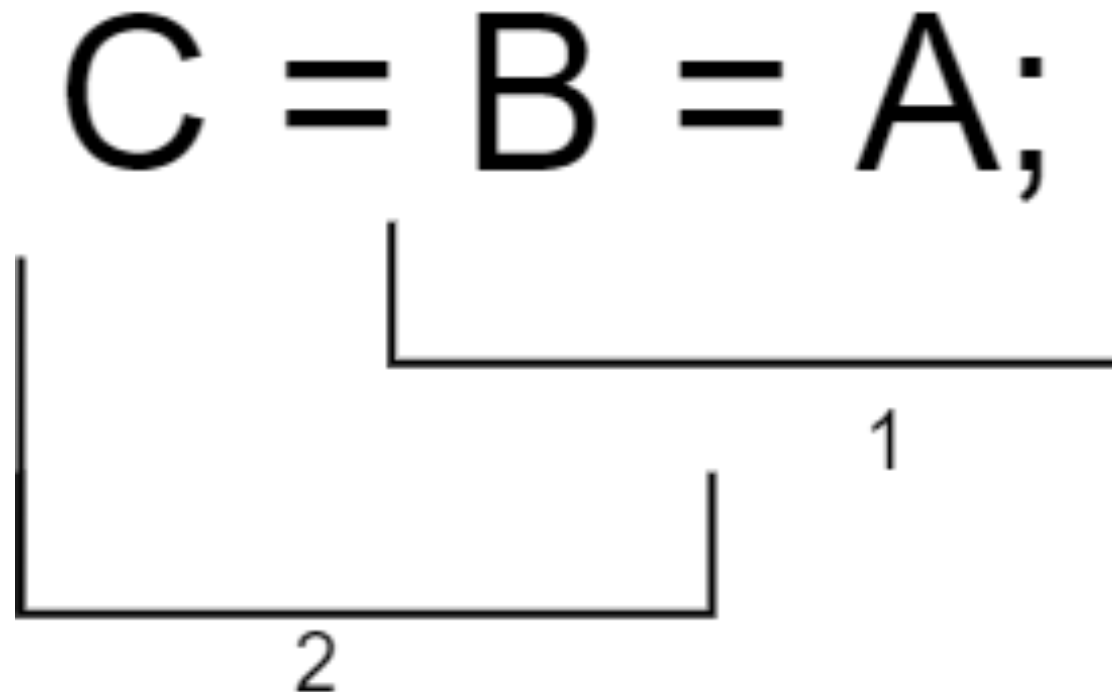
```
Для продолжения нажмите люб
```

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

```
int main()  
{  
    //Два массива  
    Array b(12);  
    Array c(7);  
    //Массив на 10 элементов  
    Array a(10);  
    for (int i = 0; i < a.Size(); i++)  
    {  
        a[i] = i;  
    }  
    //переопределенный operator=  
    // возвращает ссылку на объект  
    c = b = a;  
}
```

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ

Оператор присваивания (operator=) **ассоциативен справа налево**. (порядок чтения). И имеет **низкий приоритет**.



Тут – первым действием выполняется $V = A$ – копируется информация, **возвращается ссылка** на объект V класса `Array`, далее выполняется $C = V$.

Лабораторная работа №3. Дружественные функции и классы. Перегрузка операторов.

Создать класс `Pair`(пара чисел). Пара должна быть представлена двумя полями: типа `int` для первого числа и типа `double` для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. Реализовать:

- Вычитание пар чисел
- Добавление константы к паре (увеличивается первое число, если константа целая, второе, если константа вещественная).

```
class Pair
{
private:
    int x;
    double y;

public:
    //Конструктор без параметров.
    Pair()
    {
        this->x = 0;
        this->y = 0.;
    };

    //Конструктор с параметрами
    Pair(const int& x, const double& y)
    {
        this->x = x;
        this->y = y;
    };
};
```

```
//Добавление вещественной константы к паре.
Pair operator+(const double& y) const
{
    return Pair(this->x, this->y + y);
}

//Добавление целой константы к паре.
Pair operator+(const int& x) const
{
    return Pair(this->x + x, this->y);
}

//Вычитание пар чисел
Pair operator-(const Pair& p) const
{
    return Pair(this->x - p.x, this->y - p.y);
};
```

```
//Добавление константы к паре должно быть коммутативно
//Хотим, чтобы левый операнд был вещественный или целочисленный
//Поэтому, используем перегрузку при помощи friend функции
friend Pair operator+(const int& x,const Pair& p)
{
    return p + x;
};

friend Pair operator+(const double& y,const Pair& p)
{
    return p + y;
};

//Перегрузка оператора << для работы с потоковым классом
friend std::ostream& operator<<(std::ostream& stream, const Pair& p)
{
    stream <<'('<< p.x << ':' << p.y<<')';
    return stream;
};
```

```

int main()
{
    setlocale(LC_ALL, "Russian");
    Pair a(3,5);
    Pair b(2,8);

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "Вычитание пар чисел: " << a - b << endl;

    cout << "Добавление int константы 5 к a: " << endl;

    cout << "a + 5 = " << a + 5 << endl;
    cout << "5 + a = " << 5 + a << endl;

    cout << "Добавление double константы 2.0 к a: "
    cout << "a + 2.0 = " << a + 2.0 << endl;
    cout << "2.0 + a = " << 2.0 + a << endl;

    return 0;
}

```

```

C:\WINDOWS\system32\cmd.exe
a = (3:5)
b = (2:8)
Вычитание пар чисел: (1:-3)
Добавление int константы 5 к a:
a + 5 = (8:5)
5 + a = (8:5)
Добавление double константы 2.0 к a:
a + 2.0 = (3:7)
2.0 + a = (3:7)

```

Ответы на вопросы

1. Для чего используются дружественные функции и классы?

- Чтобы предоставить доступ к `private` – полям класса, методам другого класса, который является дружественным.
- Чтобы предоставить доступ к `private`-полям глобальным не компонентным функциям.

Ответы на вопросы

2. Сформулируйте правила описания и особенности дружественных функций.

- Дружественная функция объявляется внутри класса с ключевым словом `friend`.

- Поскольку дружественная функция не является компонентной (ей не передается указатель `this`), то необходимо, чтобы она принимала в качестве параметра объект класса по ссылке, по значению или по адресу.

- Дружественная функция может быть дружественной сразу к нескольким классам.

- На неё не распространяются спецификаторы доступа, поэтому то место, где она описана в классе, неважно.

Ответы на вопросы

3. Каким образом можно перегрузить унарные операции.

-Как компонентные нестатические функции класса.

тип `operator` “знак оператора”(void);

-Как обычная глобальная не компонентная функция, которая также может быть дружественная классу.

тип `operator` “знак оператора”(class A), где class A – передача объекта класса.

- Унарные операторы перегружаются чаще всего как методы класса.

Ответы на вопросы

4. Сколько операндов должна иметь унарная функция-операция, определяемая внутри класса.

- Унарная операция по определению работает с одним операндом. Раз она перегружается как компонентная нестатическая функция, то она не должна принимать параметров. (неявно принимает указатель *this*).

5. Сколько операндов должна иметь унарная функция-операция, определяемая вне класса.

- Т.к. оператор перегружается как глобальная функция, то параметр *this* ему не передается, следовательно необходимо явно передавать объект класса.

Ответы на вопросы

6. Сколько операндов должна иметь бинарная функция-операция, определяемая внутри класса?

- Если оператор перегружается как компонентная функция, то левым операндом по умолчанию является объект класса – *this*. Правым операндом является тот объект, что передается в качестве параметра в перегружаемый оператор.

Например: `A operator+(const int& value);` - тут для класса `A` перегружается оператор сложения как метод класса. →
`a.operator+(5) ≡ a + 5.` где `a` – объект класса.

Ответ: один.

Ответы на вопросы

7. Сколько операндов должна иметь бинарная функция-операция, определяемая вне класса?

- Если оператор перегружается как не компонентная функция, чаще всего дружественная классу. То указатель *this* не передается, поэтому, необходимо явно передавать объект класса в качестве параметра. Также необходимо передавать объект другого класса, с которым должен взаимодействовать исходный класс посредством оператора.

Например:

Из лабораторной работы №3: -

```
friend Pair operator+(const double& y, const Pair& p).
```

- Существуют бинарные операторы, перегрузка которых вне класса невозможна: это “->”, “[]”, “()”, “=”

Ответ: два

Ответы на вопросы

8. Чем отличается перегрузка префиксных и постфиксных унарных операций.

-Префиксные и постфиксные операции по сути являются версией одного оператора в разных формах. Если при перегрузке префиксного оператора не нужно передавать никаких параметров, то при перегрузке постфиксного оператора необходимо передать незначащий параметр *int*. – Чтобы объяснить компилятору разницу.

Также эти операторы могут отличаться по типу возвращаемого значения. Допустим, если префиксный оператор (инкремента или декремента) модифицирует какое-либо информационное поле, а затем возвращает ссылку на объект этого класса, то постфиксный оператор должен сохранить состояние объекта класса во временную переменную, затем модифицировать поле класса, затем вернуть копию предыдущего состояния. – Это накладывает некоторые ограничения на использование постфиксных операторов, т.к. они не позволяют взаимодействовать напрямую с объектом класса.

Ответы на вопросы

9. Каким образом можно перегрузить операцию присваивания.

- Оператор присваивания можно перегрузить только как нестатическую компонентную функцию класса.

10. Что должна возвращать операция присваивания?

- Ссылку на объект класса, в который происходит копирование (левый операнд). Это нужно для реализации многочисленного присваивания.

Например: $a = b = c$; Где a , b и c – Объекты одного класса.

Ответы на вопросы

11. Каким образом можно перегрузить операции ввода-вывода?

- Для того, чтобы обеспечить взаимодействие пользовательского класса и потокового класса (левым операндом является объект потокового класса, правым операндом является объект пользовательского класса), необходимо перегрузить оператор<< или оператор>> как дружественную функцию. С двумя параметрами – первый: объект класса `std::ostream` или `std::istream`, второй: объект пользовательского класса.

Например: `friend std::ostream& operator<<(std::ostream& stream, const Pair& p)`

- Тут `operator <<` объявлен в классе как дружественная функция.

ОТВЕТЫ НА ВОПРОСЫ

12. В программе описан класс

```
class Student  
{  
    ... Student& operator++();  
    ....
```

}; и определен объект этого класса Student s; Выполняется операция ++s; Каким образом, компилятор будет воспринимать вызов функции-операции?

-Как вызов метода класса: s.operator++();

ОТВЕТЫ НА ВОПРОСЫ

13. В программе описан класс

```
class Student {  
... friend Student& operator ++( Student&);  
.... };
```

и определен объект этого класса Student s; Выполняется операция ++s; Каким образом, компилятор будет воспринимать вызов функции-операции?

-Как вызов глобальной функции: operator++(s);

Ответы на вопросы

14. В программе описан класс class Student

```
{
```

```
...
```

```
bool operator<(Student &P);
```

```
....
```

```
};
```

и определены объекты этого класса Student a,b;

Выполняется операция cout<<a<b;

Каким образом, компилятор будет воспринимать вызов функции-операции?

- Приоритет оператора << выше, чем у оператора<. Программа просто не скомпилируется.
- Если выполнялась бы операция cout<<(a<b); То компилятор это воспринял бы как вызов метода класса a.operator<(b);

Ответы на вопросы

15. В программе описан класс class Student

```
{
```

```
...
```

```
friend bool operator >(const Person&, Person&)
```

```
.....
```

```
};
```

и определены объекты этого класса Student a,b;

Выполняется операция cout<<a>b;

Каким образом, компилятор будет воспринимать вызов функции-операции?

Приоритет у оператора << выше, программа не будет работать.

Если выполняется операция cout << (a>b); то компилятор это будет воспринимать как вызов глобальной функции: cout << operator>(a, b);