ЛАБОРАТОРНАЯ РАБОТА №5
ПЕРЕГРУЗКА ОПЕРАТОРОВ В ЯЗЫКЕ C++

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:

Целью данной лабораторной работы является изучение перегрузки операторов в языке C++ и использование перегруженных операторов на практике.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:

1. Синтаксис перегрузки

Синтаксис перегрузки операторов очень похож на определение функции с именем operator@, где @ — это идентификатор оператора (например +, -, <<, >>). Рассмотрим простейший пример:

```
1) class Integer
2) {
3) private:
4)     int value;
5) public:
6)     Integer(int i): value(i)
7)     {}
8)     const Integer operator+(const Integer& rv) const {
9)         return (value + rv.value);
10)     }
11) };
```

В данном случае, оператор оформлен как член класса, аргумент определяет значение, находящееся в правой части оператора. Вообще, существует два основных способа перегрузки операторов: глобальные функции, дружественные для класса, или подставляемые функции самого класса. Какой способ, для какого оператора лучше, рассмотрим в конце топика.

В большинстве случаев, операторы (кроме условных) возвращают объект, или ссылку на тип, к которому относятся его

аргументы (если типы разные, то вы сами решаете как интерпретировать результат вычисления оператора).

2. Перегрузка унарных операторов

Рассмотрим примеры перегрузки унарных операторов для определенного выше класса Integer. Заодно определим их в виде дружественных функций и рассмотрим операторы декремента и инкремента:

```
1) class Integer
2) {
3) private:
4)     int value;
5) public:
6)     Integer(int i): value(i)
7)     {}
8)
9)     //унарный +
10)      friend const Integer& operator+(const Integer&
   i);
11)
12)      //унарный -
13)      friend const Integer operator-(const Integer&
   i);
14)
15)      //префиксный инкремент
16)      friend const Integer& operator++(Integer& i);
17)
18)      //постфиксный инкремент
19)      friend const Integer operator++(Integer& i,
   int);
20)
21)      //префиксный декремент
22)      friend const Integer& operator--(Integer& i);
23)
24)      //постфиксный декремент
25)      friend const Integer operator--(Integer& i,
   int);
26)  };
```

```
1) //унарный плюс ничего не делает.
2) const Integer& operator+(const Integer& i) {
3)     return i.value;
4) }
5)
6) const Integer operator-(const Integer& i) {
7)     return Integer(-i.value);
8) }
9)
10) //префиксная версия возвращает значение после
   инкремента
11) const Integer& operator++(Integer& i) {
12)     i.value++;
13)     return i;
14) }
15)
16) //постфиксная версия возвращает значение до
   инкремента
17) const Integer operator++(Integer& i, int) {
18)     Integer oldValue(i.value);
19)     i.value++;
20)     return oldValue;
21) }
22)
23) //префиксная версия возвращает значение после
   декремента
24) const Integer& operator--(Integer& i) {
25)     i.value--;
26)     return i;
27) }
28)
29) //постфиксная версия возвращает значение до
   декремента
30) const Integer operator--(Integer& i, int) {
31)     Integer oldValue(i.value);
32)     i.value--;
33)     return oldValue;
34) };
```

3. Бинарные операторы

Рассмотрим синтаксис перегрузки бинарных операторов. Перегрузим один оператор, который возвращает l-значение, один

условный оператор и один оператор, создающий новое значение (определим их глобально):

```
1) class Integer
2) {
3) private:
4)     int value;
5) public:
6)     Integer(int i): value(i)
7)     {}
8)     friend const Integer operator+(const Integer&
   left, const Integer& right);
9)
10)     friend Integer& operator+=(Integer& left, const
   Integer& right);
11)
12)     friend bool operator==(const Integer& left,
   const Integer& right);
13) };
14)
15) const Integer operator+(const Integer& left, const
   Integer& right) {
16)     return Integer(left.value + right.value);
17) }
18)
19) Integer& operator+=(Integer& left, const Integer&
   right) {
20)     left.value += right.value;
21)     return left;
22) }
23)
24) bool operator==(const Integer& left, const Integer&
   right) {
25)     return left.value == right.value;
26) }
```

Во всех этих примерах операторы перегружаются для одного типа, однако, это необязательно. Можно, к примеру, перегрузить сложение нашего типа Integer и определенного по его подобию Float.

4. Особые операторы

В C++ есть операторы, обладающие специфическим синтаксисом и способом перегрузки. Например оператор индексирования []. Он всегда определяется как член класса и, так как подразумевается поведение индексируемого объекта как массива, то ему следует возвращать ссылку.

Оператор разыменования указателя

Перегрузка этих операторов может быть оправдана для классов умных указателей. Этот оператор обязательно определяется как функция класса, причём на него накладываются некоторые ограничения: он должен возвращать либо объект (или ссылку), либо указатель, позволяющий обратиться к объекту.

Оператор присваивания

Оператор присваивания обязательно определяется в виде функции класса, потому что он неразрывно связан с объектом, находящимся слева от "=". Определение оператора присваивания в глобальном виде сделало бы возможным переопределение стандартного поведения оператора "=". Пример:

```
1) class Integer
2) {
3) private:
4)     int value;
5) public:
6)     Integer(int i): value(i)
7)     {}
8)
9)     Integer& operator=(const Integer& right) {
10)         //проверка на самоприсваивание
11)         if (this == &right) {
12)             return *this;
13)         }
14)         value = right.value;
15)         return *this;
16)     }
17) };
```

5. Неперегружаемые операторы

Некоторые операторы в C++ не перегружаются в принципе. По всей видимости, это сделано из соображений безопасности.

- Оператор выбора члена класса «.".
- Оператор разыменования указателя на член класса ".*"
- В C++ отсутствует оператор возведения в степень (как в Fortran) "**".
- Запрещено определять свои операторы (возможны проблемы с определением приоритетов).
- Нельзя изменять приоритеты операторов

ЗАДАНИЕ

1. Для класса Complex перегрузить операторы присваивания, инкремента, декремента, сравнения, ввода и вывода.

2. Для класса Vector перегрузить операторы присваивания, сравнения, ввода и вывода.