Лабораторная работа №3

по курсу «Высокоуровневое программирование» (2 семестр)

«Перегрузка операторов»

Оглавление

Цель: приобретение практических навыков и знаний по работе с перегрузкой операторов.

Задачи:

1. Изучить понятие оператора;
2. Выяснить виды и способы перегрузки операторов;
3. Изучить методы и случаи применения перегрузок;
4. Научиться соединять пользовательские объекты с потоками ввода / вывода;
5. Познакомиться с понятием функтора.
6. Научиться применять перегрузку операторов на практике;

Содержание отчёта:

1. Титульный лист;
2. Цель, задачи работы;
3. Формулировка общего задания;
4. Листинги пользовательских функций, классов и основной программы;
5. Результаты работы;
6. Выводы по работе в целом.

Основные теоретические сведения

Перегрузка операторов

В языке C++ присутствует возможность перегружать функции. Перегрузка представляет собой функции имеющие одинаковые имена, но разные типы аргументов и реализацию.

Помимо функций язык программирования C++ позволяет перегружать операторы. Оператор в языке C++ реализован функцией, которая принимает один, ли или два аргумента. Один для унарных операторов, два – для бинарных операторов (тернарный оператор не перегружается). Нельзя перегрузить оператор sizeof, оператор разрешения области видимости «::», оператор выбора члена «.» и указатель, как оператор выбора члена «->».

Правила перегрузки:

- Вы можете перегрузить только существующие операторы. Вы не можете создавать новые или переименовывать существующие. Например, нельзя создать оператор ** для выполнения операции возведения в степень.

- По крайней мере один из операндов перегруженного оператора должен быть пользовательского типа данных. Это означает, что вы не можете перегрузить operator+() для выполнения операции сложения значения типа int со значением типа double. Однако вы можете перегрузить operator+() для выполнения операции сложения значения типа int с объектом пользовательского класса.
- Изначальное количество операндов, поддерживаемых оператором, изменить невозможно. Т.е. с бинарным оператором+ используются только два операнда, с унарным++ — только один.
- Все операторы сохраняют свой приоритет и ассоциативность по умолчанию.

Применение: Вы можете перегрузить оператор + для слияния нескольких объектов вашего класса String или для выполнения операции сложения двух объектов пользовательского класса. Вы можете перегрузить оператор<< для вывода пользовательского класса на экран. Вы можете перегрузить оператор равенства== для сравнения двух объектов класса и т.д. Подобные применения делают перегрузку операторов одной из самых полезных особенностей в языке C++, так как это упрощает процесс работы с классами. Пример встроенной перегрузки - операторы побитовых сдвигов были перегружены в классах потоков, тот самый cin и cout.

Способы перегрузки операторов:

- через дружественные функции;
- через обычные функции;
- через методы класса.

Перегрузка операторов через дружественные функции

Дружественная функция – это функция, которая может получать доступ к закрытым члена класса. Чтобы объявить дружественную функцию, необходимо прописать её прототип с ключевым словом: **friend** в теле класса, а потом реализовать вне класса, словно это обычная функция

```
class Ruble {

public:
	Ruble(unsigned int rub, unsigned int kop) {
		m_rubles = rub;
		m_kopecks = kop;
	}

	void Print() {
		cout << m_rubles << ',' << m_kopecks;
	}

	// объявляем, что функция operator+ является дружественной для класса Ruble
```

```
        friend Ruble operator+(Ruble s_1, Ruble s_2);

private:
        unsigned int m_rubles{};
        unsigned int m_kopecks{};
};

// реализация функции
Ruble operator+(Ruble s_1, Ruble s_2) {

        int rub = s_1.m_rubles + s_2.m_rubles;
        int kop = s_1.m_kopecks + s_2.m_kopecks;

        return Ruble(rub, kop);
}

int main() {

        Ruble sum_1(25, 17);
        Ruble sum_2(17, 12);

        Ruble sum_3 = sum_1 + sum_2; // применение оператора

        sum_3.Print();

        return 0;
}
```

Чтобы выполнить перегрузку оператора необходимо вместо имени функции написать ключевое слово: operator, а затем нужный знак, который будем перегружать. В аргументах необходимо прописать те переменные, которые будут входить в выражение <слагаемое1> + <слагаемое2>. Функция возвращает объект пользовательского класса, который будет создан на основе суммы значений первых двух.

Внутри функции мы имеем доступ непосредственно к приватным полям объекта, это возможно благодаря пометке этой функции, как дружественной для класса. Таким образом можно перегрузить и другие операторы: «-, *, /».

Прибавление числа к классу:

```
// внутри класса
        friend Ruble operator+(Ruble s_1, int num);

// вне класса
Ruble operator+(Ruble s_1, int num) {

        int rub = s_1.m_rubles + num;
        int kop = s_1.m_kopecks;

        return Ruble(rub, kop);
}

// применение оператора
        Ruble sum_3 = sum_1 + 12;
```

Важен порядок аргументов в операторе.

```
Ruble sum_3 = sum_1 + 12;
Ruble sum_3 = 12 + sum_1;
```

Эти две строки не эквивалентны. В первой строке будет вызван наш перегруженный оператор, который принимает сперва класс Ruble, а потом число типа int. А во – второй строчке нужен оператор, который будет принимать сперва число, а потом наш класс, у нас такого оператора нет, поэтому компилятор выдаст нам ошибку.

Перегрузка операторов через обычные функции

Использование дружественной функции для перегрузки операторов удобно тем, что мы имеем прямой доступ ко всем членам класса, с которым работаем. В примере, приведенном выше, дружественная функция перегрузки оператора + имеет прямой доступ к закрытым членам класса Ruble. Однако, если нам не нужен доступ к членам определенного класса, мы можем перегрузить оператор и через обычную функцию. Обратите внимание, в классе должен присутствовать геттеры getRub() и getKop(), с помощью которых мы можем получить доступ к m_rub и m_kop извне класса. Перегрузка оператора + через обычную функцию:

```
class Ruble {

public:
	Ruble(unsigned int rub, unsigned int kop) {
		m_rubles = rub;
		m_kopecks = kop;
	}

	void Print() {
		cout << m_rubles << ',' << m_kopecks;
	}

	unsigned int getRub() {
		return m_rubles;
	}

	unsigned int getKop() {
		return m_kopecks;
	}

private:
	unsigned int m_rubles{};
	unsigned int m_kopecks{};
};

Ruble operator+(Ruble s_1, Ruble s_2) {

	int rub = s_1.getRub() + s_2.getRub();
	int kop = s_1.getKop() + s_2.getKop();

	return Ruble(rub, kop);
}
```

Поскольку принцип перегрузки операторов через обычные и дружественные функции почти идентичен (разница в уровнях/условиях доступа к закрытым членам класса), единственное отличие заключается в том, что в случае с дружественной функцией, её нужно обязательно объявить в классе и определить вне тела класса (или в классе), в то время как обычную функцию достаточно просто определить вне тела класса, без указания дополнительного прототипа функции.

Перегрузка операторов через методы класса

Перегружаемый оператор определяется как метод класса, вместо дружественной функции (Ruble::operator+ вместо friend operator+). Левый параметр из функции перегрузки выбрасывается, вместо него — неявный объект - указатель *this. Внутри тела функции все ссылки на левый параметр могут быть удалены (например, sum_1.m_rub становится m_rub).

Перегрузка оператора + через метод класса:

```
class Ruble {

public:

	Ruble operator+(int num) {

		int rub = m_rubles + num;
		int kop = m_kopecks;

		return Ruble(rub, kop);
	}

	Ruble operator+(Ruble s_1) {

		int rub = m_rubles + s_1.m_rubles;
		int kop = m_kopecks + s_1.m_kopecks;

		return Ruble(rub, kop);
	}

	friend ostream& operator<<(ostream& out, const Ruble& sum);

	friend istream& operator>>(istream& in, Ruble& sum);

private:
	unsigned int m_rubles{};
	unsigned int m_kopecks{};
};
```

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

Перегрузка через обычную или дружественную функции работает для всех типов данных параметров (даже если левый операнд не является объектом класса или является объектом класса, который изменить нельзя).

- 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.

Через метод класса перегрузить оператор << мы не сможем. Потому что при перегрузке через метод класса в качестве левого операнда используется текущий объект. В этом случае левым операндом является объект типа std::ostream или std::ostream и является частью Стандартной библиотеки C++. Поэтому перегрузка оператора << должна осуществляться через дружественную функцию.

Аналогично, хотя мы можем перегрузить operator+(Ruble, int) через метод класса (как мы делали выше), мы не можем перегрузить operator+(int, Ruble) через метод класса, поскольку int теперь является левым операндом, на который указатель *this указывать не может.

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

Перегрузка операторов ввода / вывода

Для классов с множеством переменных-членов, выводить в консоль каждую переменную по отдельности может быть неудобно. В таких случаях считается хорошим вариантом написать отдельную функцию, или метод для вывода, которую можно было бы повторно использовать. Например, функция print() из наших примеров:

```
void Print() {
	cout << m_rubles << ',' << m_kopecks;
}
```

Поскольку метод print() имеет тип void, то его нельзя вызывать в середине стейтмента вывода. Вместо этого стейтмент вывода приходится разбивать на несколько частей (строк).

Не разбивать стейтмент вывода и не запоминать название функции вывода можно, перегрузив оператор вывода <<. Его левым операндом является объект std::cout, а правым — объект пользовательского класса. std::cout фактически является объектом типа std::ostream, его можно использовать параметром out в нашей функции перегрузки (который затем станет ссылкой на std::cout при вызове этого оператора), поэтому перегрузка оператора << выглядит следующим образом:

```
friend ostream& operator<<(ostream& out, const Ruble& sum);

ostream& operator<<(ostream& out, const Ruble& sum) {
	sum.Print();

	return out;
}
```

С перегрузкой арифметических операторов мы вычисляли и возвращали результат по значению. Однако, если вы попытаетесь возвратить std::ostream по значению, то получите ошибку компилятора, так как std::ostream запрещает свое копирование. Возврат ссылки не только предотвращает создание копии std::ostream, но также позволяет нам «связать» стейтменты вывода вместе: std::cout << sum_1 << std::endl;.

Возвращая параметр out в качестве возвращаемого значения выражения (std::cout << sum_1) мы возвращаем std::cout, и вторая часть нашего выражения обрабатывается как std::cout << std::endl;

Второй параметр перегруженного оператора имеет константный тип, а значит объект не будет изменён функцией, то есть мы не сможем у объекта вызвать методы, которые не являются константными. Константный метод обозначается после параметров функции:

```
	void Print() const {
		cout << m_rubles << ',' << m_kopecks;
	}
```

Перегруженный оператор ввода >>:

```
	friend istream& operator>>(istream& in, Ruble& sum);

	istream& operator>>(istream& in, Ruble& sum) {
		in >> sum.m_rubles;
		in >> sum.m_kopecks;

		return in;
	}

	cin >> sum_1;
```

Перегрузка унарных операторов +, – и логического НЕ

Рассмотрим унарные операторы плюс (+), минус (-) и логическое НЕ (!), которые работают с одним операндом.

Оператор «-» возвращает объект Ruble с отрицательным значением m_rubles и m_kopecks. Поскольку этот оператор не изменяет объект класса Ruble, то мы должны сделать функцию перегрузки константной чтобы иметь возможность использовать этот оператор и с константными объектами класса Ruble.

```
	Ruble operator-() const {
		return Ruble(-m_rubles, -m_kopecks);
	}
```

В языке C++ значение 0 обозначает false, а любое другое ненулевое значение обозначает true, поэтому при работе с классами оператор ! будет возвращать true, если значением объекта класса является false, 0 или любое другое значение, заданное как дефолтное (по умолчанию) при инициализации, а в противном случае оператор ! будет возвращать false.

В следующем примере мы добавим перегрузку оператора отрицания для класса ответа на вопрос так, чтобы логическое значение зависело от значения поля класса:

```
class Answer {
public:
	Answer(bool check) {
		m_state = check;
	}

	bool operator!() {
		return !m_state;
	}

private:
	bool m_state{};
};
```

Перегрузка операторов сравнения

Поскольку все операторы сравнения являются бинарными и не изменяют свои левые операнды, то выполнять перегрузку следует через дружественные функции. Например, перегрузим оператор равенства == и оператор неравенства != для класса очков:

```
class Score {
public:
	Score(int score) {
		m_score = score;
	}
	friend bool operator==(const Score &s_1, const Score &s_2);
	friend bool operator!=(const Score &s_1, const Score &s_2);

private:
	int m_score{};
};

bool operator==(const Score &s_1, const Score &s_2) {
	return s_1.m_score == s_2.m_score;
}

bool operator!=(const Score &s_1, const Score &s_2) {
	return !(s_1 == s_2);
}
```

Перегрузка операторов инкремента и декремента

Поскольку операторы инкремента и декремента являются унарными и изменяют свои операнды, то перегрузку следует выполнять через методы класса. Есть две версии операторов инкремента и декремента: версия префикс (например, ++x, --y) и версия постфикс (например, x++, y--). Перегрузка операторов инкремента и декремента версии префикс аналогична перегрузке любых других унарных операторов:

```
class Score {
public:
	Score(int score) {
		m_score = score;
	}

	Score& operator++() {
		++m_score;

		// возвращаем разыменованный указатель на текущий объект класса
		return *this;
	}
private:
	int m_score{};
};
```

Язык C++ использует фиктивную переменную (или «фиктивный параметр») для операторов версии постфикс, который отличает версию постфикс операторов инкремента/декремента от версии префикс.

```
class Score {
public:
	Score(int score) {
		m_score = score;
	}

	Score& operator++() {
		++m_score;

		// возвращаем разыменованный указатель на текущий объект класса
		return *this;
	}

	Score operator++(int) {
		Score tmp(m_score);

		++(*this);

		// возвращаем объект со старым значением
		return tmp;
	}
private:
	int m_score{};
};
```

Перегрузка оператора индексации []

Функция перегрузки оператора [] всегда будет принимать один параметр: значение индекса (элемент массива, к которому требуется доступ) для возврата значения элемента по этому индексу.

```
class IntArray {
public:
	int& operator[] (const int index) {
		return m_array[index];
	}

private:
	int m_array[10]{};
};
```

Теперь всякий раз, когда мы будем использовать оператор индексации ([]) с объектом класса IntArray, компилятор будет возвращать соответствующий элемент массива m_array! Это позволит нам непосредственно как получать, так и присваивать значения элементам m_array.

Перегрузка оператора () и функторы

Перегрузка оператора () используется в реализации функторов (или «функциональных объектов») – классы, которые работают как функции. Преимущество функтора над обычной функцией заключается в том, что функторы могут хранить данные в переменных-членах (поскольку они сами являются классами). Вот пример использования простого функтора:

```
class Accumulator {
public:
	Accumulator() {}

	int operator() (int i) {
		return (m_counter += i);
	}

private:
	int m_counter = 0;
};

int main()
{
	Accumulator accum;
	std::cout << accum(30) << std::endl; // выведется 30
	std::cout << accum(40) << std::endl; // выведется 70

	return 0;
}
```

Обратите внимание, использование класса Accumulator выглядит так же, как и вызов обычной функции, но наш объект класса Accumulator может хранить значение, которое увеличивается.

Такой класс можно реализовать и через обычную функцию со статической локальной переменной, но поскольку функции представлены только одним глобальным экземпляром (т.е. нельзя создать несколько объектов функции), использовать эту функцию мы можем только для выполнения чего-то одного за раз. С помощью функторов мы можем создать любое количество отдельных функциональных объектов, которые нам нужны, и использовать их одновременно.

Задание для всех вариантов

1. Написать для пользовательских классов дружественные функции перегрузки операторов ввода и вывода.
2. Написать перегрузки операторов сравнения двух объектов пользовательского класса, как методов внутри класса.
3. На данном этапе хранение списков данных в программе (пользователи и другие объекты) может быть реализовано в виде простых массивов в главном файле программы.
4. Создать функции взаимодействия пользователя приложения с данными: добавление (используя перегрузки операторов ввода) и удаление, сортировка (с помощью перегрузок операторов сравнения).
5. Совместить созданный функционал с меню.

Контрольные вопросы

1. Что такое оператор в языке C++?
2. Что такое перегрузка операторов?
3. Какие операторы можно перегружать?
4. Как происходит перегрузка операторов?
5. Какие существуют способы перегрузки операторов?
6. Когда, какие способы перегрузки нужно применять?
7. Как выполнить перегрузку операторов ввода/вывода?
8. Как происходит перегрузка унарных операторов?
9. Как выполняется перегрузка операторов сравнения?
10. Перегрузка операторов инкремента и декремента.
11. Чем отличается перегрузка префиксной и постфиксной формы?
12. Как перегрузить оператор индексации?
13. Почему оператор индексации возвращает ссылку, а не значение?
14. Перегрузка оператора «()».
15. Что такое функторы?

Список литературы

1. Курс лекций доцента кафедры ФН1-КФ Пчелинцевой Н.И.
2. Программирование на языке высокого уровня C/C++ [Электронный ресурс]: конспект лекций / – Электрон. текстовые данные. – М.: Московский государственный строительный университет, Ай Пи Эр Медиа, ЭБС АСВ, 2016. – 140 с. – Режим доступа: http://www.iprbookshop.ru/48037.

В начало