ЛАБОРАТОРНАЯ РАБОТА №5.

Перегрузка операций.

Цель работы: Изучение способов определения арифметических и логических операций над объектами C++, приобретение навыков использования операторных функций в консольных приложениях.

Теоретические сведения.

5.1) Перегрузка операций C++.

Одной из замечательных особенностей языка C++ является возможность переопределения стандартных арифметических или логических действий (например, «+», «−», «=», «<», «==» и т.д.) для объектов того или иного класса. Такое переопределение называют *перегрузкой операций*.

Основная причина, по которой программист решает перегрузить некоторую операцию – это улучшение читаемости программного кода. В качестве классического примера можно рассмотреть операцию конкатенации (объединения) двух текстовых строк. В языке C для объединения двух нуль-терминированных строк используется стандартная функция

```
char* strcat(char* dest, const char* src);
```

которая добавляет содержимое текстовой строки `src` в конец текстовой строки `dest` и возвращает указатель на полученную строку. С помощью этой функции можно «склеить» две строки `str1` и `str2`, например, следующим образом

```
char str1[10] = "Инь";
char str2[20] = "Янь";
strcat(str1, str2);
cout << str1 << endl;
```

С другой стороны, в языке C++ две текстовые строки можно объединить с помощью простой операции «+», например, так

```
string str1 = "Инь";
string str2 = "Янь";
string res = str1 + str2;
cout << res << endl;
```

Легко видеть, что в данном случае программный код более прозрачен и понятен. Это достигается за счет перегрузки операции сложения «+» для объектов класса `string`.

С точки зрения программиста, перегрузка операции означает определение специальной *операторной функции*. Число аргументов этой функции зависит от арности оператора (унарный или бинарный), а также от способа объявления операторной функции. Существует 2 способа объявления операторной функции:

1) как глобальной функции,

2) как метода класса.

Рассмотрим эти способы подробнее.

5.2) Перегрузка через глобальную операторную функцию.

Для определения операторной функции в языке C++ используется ключевое слово `operator`, вслед за которым указывается знак операции (символ "+", "=" и т.д.). В программе могут быть определены несколько таких функций, отличающихся только количеством и типом параметров. Таким образом, один и тот же математический символ может приводить к различным действиям, если он применяется к объектам разных классов.

В общем виде операторная функция может быть определена следующим образом

```
тип_результата operator символ(список_аргументов)
{
        <тело_операторной_функции>
}
```

В качестве `типа_результата` здесь может выступать как встроенный тип данных (int, float, bool и т.д.), так и пользовательский класс (например, `book` или `bookstore`). `Список_аргументов` должен содержать объекты тех классов, для которых перегружается заданная операция. Количество аргументов функции в данном случае точно соответствует арности оператора: для унарных операторов функция должна принимать один аргумент, для бинарных – 2 аргумента. `Символ` оператора может быть одним из следующих

оператор	значение	арность
,	запятая	бинарный
!	логическое НЕ	унарный
!=	неравно	бинарный
%	остаток от деления	бинарный
%=	остаток с присв.	бинарный
&	побитовое ИЛИ	бинарный
&	адрес	унарный
&&	логическое ИЛИ	бинарный
&=	битовое ИЛИ с присв.	бинарный
()	вызов функции	-
()	преобразование типа	унарный
*	умножение	бинарный
*	разымен. указателя	унарный
*=	умножение с присв.	бинарный
+	сложение	бинарный
+	унарный плюс	унарный
++	инкремент	унарный
+=	сложение с присв.	бинарный
-	вычитание	бинарный
-	унарный минус	унарный
--	декремент	унарный
-=	вычитание с присв.	бинарный

оператор	значение	арность
->	доступ к полю	бинарный
/	деление	бинарный
/=	деление с присв.	бинарный
<	меньше	бинарный
<<	сдвиг влево	бинарный
<<=	сдвиг с присв.	бинарный
<=	меньше или равно	бинарный
=	присвоение	бинарный
==	равно	бинарный
>	больше	бинарный
>>	сдвиг вправо	бинарный
>>=	сдвиг с присв.	бинарный
>=	больше или равно	бинарный
[]	элемент массива	-
^	исключающее ИЛИ	бинарный
^=	искл. ИЛИ с присв.	бинарный
|	побитовое ИЛИ	бинарный
|=	побит. ИЛИ с присв.	бинарный
||	логическое ИЛИ	бинарный
~	побитовое НЕ	унарный
delete	удаление из памяти	-
new	размещ. в памяти	-

Существуют следующие ограничения на перегрузку операторов C++

1) Допускается перегружать только существующие операторы. Нельзя определить новый оператор (например, «**»).
2) Запрещается перегружать операторы для встроенных типов данных (int, float, char, …).
3) Перегруженные операции подчиняются стандартным правилам старшинства при использовании в составе сложных выражений.

4) Если некоторая операция имеет две формы - унарную и бинарную (&, *, +, и -), то каждая из этих форм может быть перегружена отдельно.
5) Аргументы операторной функции не могут иметь значений по умолчанию.
6) Перегруженные операторы наследуются производными классами (кроме оператора присваивания operator=).

В качестве примера рассмотрим класс комплексного числа, который содержит поля для хранения вещественной и мнимой части (re и im), конструктор для инициализации полей, и функцию вывода комплексного числа на экран

```
class complex
{
public:
    double re, im;
    complex(double r = 0, double i = 0) : re(r), im(i) {};
    void display() { cout << re << ", " << im << endl; }
};
```

Перегрузим далее операцию сложения двух комплексных чисел с помощью глобальной функции `operator+`. Так как в данном случае перегружается бинарный оператор, функция будет принимать в качестве аргументов два объекта класса `complex`. Результатом сложения также является комплексное число, поэтому возвращаемым значением будет объект класса `complex`

```
complex operator+(complex z1, complex z2)
{
    complex z3;
    z3.re = z1.re + z2.re;
    z3.im = z1.im + z2.im;
    return z3;
}
```

Обратим внимание на то, что внутри функции создается локальный объект с именем z3, полям которого присваиваются значения, вычисленные в соответствии с правилами комплексной арифметики. Этот объект затем возвращается в вызывающую программу. Можно предложить более лаконичный вариант записи той же самой операторной функции, где новый объект создается «на лету», а после инициализации полей сразу возвращается в вызывающую программу.

```
complex operator+(complex z1, complex z2)
{
    return complex(z1.re + z2.re, z1.im + z2.im);
}
```

Рассмотрим теперь пример использования разработанной операторной функции (в любой из описанных выше форм).

```
int main()
{
    complex a(1.2, 3.4);
    complex b(5.6, 7.8);
    complex c;

    c = a + b;
    c.display();
    system("pause");
}
```

С точки зрения компилятора, оператор `c = a + b` в приведенном выше фрагменте кода эквивалентен вызову операторной функции `c = operator+(a, b)`. Полям re и im объекта `c` в итоге будут присвоены значения 6.8 и 11.2, то есть выполнено сложение комплексных чисел.

5.3) Функции – друзья класса.

В рассмотренном выше примере поля re и im класса `complex` объявлены в его открытой (public) части, что, вообще говоря, нарушает инкапсуляцию данных. Однако такой способ объявления был выбран не случайно. Если бы эти поля были объявлены закрытыми (private), прямое обращение к ним в функции `operator+` (в конструкциях вида `z1.re + z2.re`) было бы невозможным. Одним из выходов в данной ситуации может стать использование функций-друзей (friend function).

По определению, дружественная к некоторому классу функция имеет прямой доступ к его закрытым членам – полям и методам. Такие функции позволяют, с одной стороны, обеспечить сохранность данных объекта, с другой стороны, эффективно реализовать доступ к его содержимому. В некоторых случаях это решение является единственным.

Для того, что некоторая функция стала дружественной к заданному классу, она должна быть объявлена внутри этого класса с ключевым словом friend. Для класса комплексных чисел такое объявление может иметь вид

```
class complex
{
private:
    double re, im;
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {};
    void display() { cout << re << ", " << im << endl; }

    friend complex operator+(complex z1, complex z2);
};
```

Несмотря на то, что данное объявление похоже на объявление метода, friend-функция не является методом класса, и остается глобальной (внешней по отношению к классу) функцией. Реализация этой функции внешне ничем не отличается от рассмотренного нами ранее варианта

```
complex operator+(complex z1, complex z2)
{
    return complex(z1.re + z2.re, z1.im + z2.im);
}
```

Однако легко заметить существенное «внутреннее» различие – в последнем случае мы манипулируем значениями закрытых полей данных внутри операторной функции.

5.4) Операторная функция как метод класса.

Операторная функция может быть объявлена не только как глобальная и дружественная, но и как метод класса. В этом случае она объявляется в открытой (public) части класса. Ключевой особенностью такого вида объявления является то, что число параметров операторной функции будет на единицу меньше, чем арность оператора (то есть, для бинарного оператора требуется один аргумент, для унарного оператора –

аргументы отсутствуют). Причина неожиданного «уменьшения» числа аргументов достаточно проста – так как мы имеем дело с методом класса, один из параметров передается неявно, через указатель this.

Перегрузим операцию сложения двух комплексных чисел, используя теперь не глобальную функцию, а метод класса `complex`

```
class complex
{
private:
    double re, im;
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {};
    void display() { cout << re << ", " << im << endl; }

    complex operator+(const complex z);
};
```

Реализация операторного метода в данном случае будет иметь вид

```
complex complex::operator+(const complex z)
{
    return complex(re + z.re, im + z.im);
}
```

Использовать эту функцию можно точно так же, как и описанную ранее глобальную дружественную функцию. Приведенный ниже пример внешне ничем не отличается от примера из п. 5.2.

```
int main()
{
    complex a(1.2, 3.4);
    complex b(5.6, 7.8);
    complex c;

    c = a + b;
    c.display();
    system("pause");
}
```

Однако в данном случае фрагмент кода `c = a + b` будет означать вызов операторного метода вида `c = a.operator+(b)`.

5.5) Пример приложения Visual C++.

Вернемся теперь к нашему программному проекту, использующему классы `book` и `bookstore`. Реализуем некоторые из используемых нами функций как операторные. А именно, для класса `book` реализуем

1) операцию проверки двух книг на эквивалентность (operator==)
2) операцию чтения данных из потока istream (operator>>)
3) операцию записи данных в поток ostream (operator<<)
4) операцию чтения данных из файла ifstream (operator>>)
5) операцию записи данных в файл ofstream (operator<<)

========== файл books.h ==========

```
#ifndef BOOKS_H
#define BOOKS_H

#include <string>
#include <iostream>
#include <fstream>
using namespace std;

class book
{
private:
	string title;
	string authors;
	string publisher;
	int year;
	unsigned int pages;

public:
	book() : title(""), authors(""), publisher(""), year(0), pages(0) {};
	book(string, string, string, int, unsigned int);
	string get_title();

	bool operator ==(book another);
	friend ostream& operator<<(ostream& stream, const book& abook);
	friend istream& operator>>(istream& stream, book& abook);
	friend ofstream& operator<<(ofstream& stream, const book& abook);
	friend ifstream& operator>>(ifstream& stream, book& abook);
};

#endif
```

========== файл books.cpp ==========

```
#include "stdafx.h"
#include "books.h"

using namespace std;

book::book(string ttl, string aut, string pbl, int yer, unsigned int pag)
{
	title = ttl;
	authors = aut;
	publisher = pbl;
	year = yer;
	pages = pag;
}

string book::get_title()
{
	return title;
}

bool book::operator ==(book another)
{
	if (title != another.title) return false;
	if (authors != another.authors) return false;
	if (publisher != another.publisher) return false;
	if (year != another.year) return false;
	if (pages != another.pages) return false;
	return true;
}

ostream& operator<<(ostream& stream, const book& abook)
```

```
{
	stream << "Название: " << abook.title << endl;
	stream << "Автор(ы): " << abook.authors << endl;
	stream << "Издатель: " << abook.publisher << endl;
	stream << "Год:      " << abook.year << endl;
	stream << "Страниц:  " << abook.pages << endl;
	return stream;
}

istream& operator>>(istream& stream, book& abook)
{
	getline(stream, abook.title);
	getline(stream, abook.authors);
	getline(stream, abook.publisher);
	stream >> abook.year;
	stream >> abook.pages;
	stream.get();
	return stream;
}

ofstream& operator<<(ofstream& stream, const book& abook)
{
	stream << abook.title << endl;
	stream << abook.authors << endl;
	stream << abook.publisher << endl;
	stream << abook.year << endl;
	stream << abook.pages << endl;
	return stream;
}

ifstream& operator>>(ifstream& stream, book& abook)
{
	getline(stream, abook.title);
	getline(stream, abook.authors);
	getline(stream, abook.publisher);
	stream >> abook.year;
	stream >> abook.pages;
	stream.get();
	return stream;
}
```

Для класса `bookstore`, в свою очередь, реализуем

1) операторную функцию += вместо метода add_book
2) операторную функцию -= для удаления книги из списка
3) функцию записи в поток содержимого всего класса-контейнера

```
====================== файл bookstore.h ======================

#ifndef BOOKSTORE_H
#define BOOKSTORE_H

#include <string>
#include <fstream>
#include "books.h"

class bookstore
{
private:
	int max_num_books;
	int num_books;
	book *books;
```

```
public:
	bookstore(unsigned int max_nb);
	~bookstore();

	void operator +=(const book &abook);
	void operator -=(const book &abook);
	friend ostream& operator<<(ostream& stream, const bookstore& astore);

	void read_from_file(string filename);
	void write_to_file(string filename);
	void display_all();
	void find_book(string atitle);
};

#endif
```

====================== файл bookstore.cpp ======================

```
#include "stdafx.h"
#include <iostream>
#include "bookstore.h"

using namespace std;

bookstore::bookstore(unsigned int max_nb)
{
	max_num_books = max_nb;
	books = new book[max_num_books];
	num_books = 0;
	cout << "\nВызван конструктор класса bookstore:";
	cout << "\n    выделено объектов - " << max_num_books;
	cout << "\n    загружено книг - " << num_books << endl;
}

bookstore::~bookstore()
{
	max_num_books = 0;
	delete[] books;
	num_books = 0;
	cout << "\nВызван деструктор класса bookstore:";
	cout << "\n    выделенная память освобождена";
}

void bookstore::operator+=(const book &abook)
{
	if (num_books < max_num_books)
	{
		books[num_books] = abook;
		num_books++;
	}
}

void bookstore::operator-=(const book &abook)
{
	if (num_books < 0)
		return;
	int index = -1;
	for (int i = 0; i < num_books; i++)
	    if (books[i] == abook)
			index = i;
	if (index == -1)
		return;

	if (index != num_books - 1)
```

```
			books[index] = books[num_books - 1];
		num_books--;
}

ostream& operator<<(ostream& stream, const bookstore& astore)
{
		stream << "\n ВЕСЬ АССОРТИМЕНТ МАГАЗИНА:\n";
		for (int i = 0; i < astore.num_books; i++)
		{
			stream << "=========== книга " << i+1 << " ============== \n";
			stream << astore.books[i] << endl;
		}
		return stream;
}

void bookstore::read_from_file(string filename)
{
		ifstream infile;
		infile.open(filename);
		if (!infile.is_open())
		{
			cout << "\n\nФайл данных не найден!\n";
			system("pause");
			exit(1);
		}

		int N;
		infile >> N;
		infile.get();

		for (int i = 0; i < N; i++)
		{
			book new_book;
			infile >> new_book;
			this->operator+=(new_book);
		}
		infile.close();

		cout << "\nЗагружены данные из файла " << filename << ":";
		cout << "\n    число загруженных книг - " << num_books << endl;
}

void bookstore::write_to_file(string filename)
{
		ofstream outfile;
		outfile.open(filename);
		outfile << num_books << endl;
		for (int i = 0; i < num_books; i++)
			outfile << books[i];
		outfile.close();

		cout << "\nДанные записаны в файл " << filename << ":";
		cout << "\n    число записанных книг - " << num_books;
}

void bookstore::find_book(string atitle)
{
		cout << "\n\nИщем книгу с названим \"" << atitle << "\"";
		bool found = false;
		for (int i = 0; i < num_books; i++)
		{
			if (books[i].get_title() == atitle)
			{
				found = true;
				cout << "\nНайдена книга:\n";
```

```
                        cout << books[i];
                }
        }
        if (!found)
                cout << "\nКнига с таким названием не найдена!\n";
}
```

`================` основной модуль - lab5.cpp `==================`

```
#include "stdafx.h"
#include <string>
#include "bookstore.h"
#include "books.h"

void main()
{
        setlocale(LC_ALL, "rus");

        bookstore my_store(25);
        my_store.read_from_file("my_books.txt");

        book book1("Дети капитана Гранта", "Верн Ж.", "Эксмо", 2003, 800);
        my_store += book1;

        cout << my_store;
        my_store.find_book("Война и мир");

        system("pause");
}
```

`================================================================`

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №5.

1) Для своего варианта задания реализуйте операторные функции:
   - проверки двух объектов основного класса на эквивалентность (operator==),
   - вывода информации в поток ostream (operator<<),
   - ввода данных из потока istream (operator>>),
   - записи в файловый поток ofstream (operator<<),
   - чтения из файлового потока ifstream (operator>>),
   - добавления нового элемента в класс-контейнер (operator+=).

2) Продемонстрируйте работу операторных функций.

Содержание отчета по лабораторной работе №5.

1) Стандартная «шапка» отчета
2) Цель: формулировка цели работы
3) Теория: краткие сведения о перегрузке операций с примерами (объем 2-3 стр.)
4) Обновленная диаграмма классов.
5) Программный код
   - объявление основного класса и класса-контейнера (с операторными функциями),
   - реализацию одной операторной функции как глобальной, другой – как метода класса.