ЛАБОРАТОРНАЯ РАБОТА №4.

Композиция классов.

Цель работы: Изучение видов взаимодействия между классами, методов агрегации и композиции классов, приобретение навыков композиции классов в языке C++.

Теоретические сведения.

4.1) Взаимодействие классов. Композиция и агрегация.

В рамках объектно-ориентированного похода изучаемая предметная область описывается в терминах одного или нескольких классов (в общем случае, системы классов). С формальной точки зрения, класс представляет собой «шаблон», который описывает общие свойства и общее поведение всех объектов данного типа. К примеру, программный класс `Автомобиль (car)` может содержать сведения о владельце автомобиля, его регистрационном номере, дате выпуска, типе и цвете кузова, мощности двигателя и т.д., которые хранятся в отдельных полях. Действия автомобиля на дороге также могут быть реализованы в виде методов класса `car`, например

- запустить двигатель,
- набрать скорость,
- остановиться,
- повернуть направо,
- включить свет фар и т.д.

Следующим шагом при разработке класса может стать описание взаимодействия автомобилей между собой (уступить дорогу автомобилю справа, включить сигнал поворота и др.), а также взаимодействие с пешеходами, автоинспекторами, дорожными знаками и разметкой и т.д. Такая архитектура программы позволит довольно просто моделировать движение интенсивных транспортных потоков, включающих сотни и тысячи участников дорожного движения.

Среди всех видов взаимодействия классов между собой в данной работе мы рассмотрим отношения *композиции*. Это отношение возникает в случае, когда один класс содержит объекты другого класса в качестве своей составной части. Иначе это называют отношением «целое-часть» или «has a».

Рассмотрим пример. Известно, что каждый автомобиль содержит двигатель, кузов, четыре колеса и другие элементы. Учтем это при разработке класса `car` и создадим отдельно классы `engine` (двигатель), `body` (кузов), и `wheel` (колесо). Определим класс `engine` следующим образом

```
class engine     // класс двигателя
{
        ...
        float power;   // мощность
        int gear;      // передача
        ...
        void start();  // запустить двигатель
        void stop();   // остановить двигатель
        ...
};
```

Аналогично определим классы body и wheel. Затем определим класс автомобиля так

```
class car          // класс автомобиля
{
       ...
       engine car_engine;  // двигатель
       body car_body;      // кузов
       wheel wheels[4];    // 4 колеса
       ...
};
```

Как видно из приведенного фрагмента, объект car_engine класса engine включен в тело класса car в качестве поля данных. Этот объект описывает двигатель, входящий в состав автомобиля. Таким же способом в тело класса car включен один экземпляр класса body и массив из четырех экземпляров класса wheel. Эти объекты используются для хранения данных о кузове автомобиля и его колесах, соответственно.

Создадим далее экземпляр класса car с именем my_car

```
car my_car;
```

Доступ к компонентам этого сложного объекта производится стандартным способом – с помощью операции «точка». Следующий фрагмент демонстрирует вызов методов классов engine и wheel (только для случая, если поля и методы являются открытыми – public)

```
my_car.car_engine.start();      // запустить двигатель my_car
my_car.wheels[0].turn_right();  // повернуть направо ...
my_car.wheels[1].turn_right();  // .. два передних колеса
```

Как показывает приведенный пример, композиция помогает создавать сложные классы на основе уже существующих, более простых классов. Сложные классы, полученные в результате композиции, часто называют *классами-контейнерами* или *агрегатными классами*. Так, класс car является контейнером для объекта класса engine, объекта класса body и массива из 4-х объектов класса wheel.

Наряду с композицией, для той же цели используется *агрегация* классов. Различие между композицией и агрегацией определяется соотношением между временами жизни экземпляра класса-контейнера и экземпляров содержащегося в нем класса. В случае композиции, объекты включенного в контейнер класса прекращают свое существование после уничтожения их контейнера, в случае агрегации – могут продолжить независимое существование. В рассмотренном примере, удаление из памяти объекта класса car приведет к одновременному уничтожению содержащихся в нем объектов классов engine, body и wheel, поэтому такая связь является композицией. Для композиции характерно включение объектов по значению, для агрегации – по адресу или ссылке.

Проиллюстрируем различие между композицией и агрегацией классов еще одним примером. Рассмотрим отношения между программными классами Вуз, Студент и Факультет. Примем во внимание, что в вузе может обучаться любое количество студентов, причем каждый студент может обучаться в одном или нескольких вузах. Вуз может состоять из одного или нескольких факультетов, но каждый факультет принадлежит только одному вузу. Тогда отношение между классами Вуз и Факультет является композицией, так как при расформировании вуза все факультетов также должны быть автоматически расформированы. С другой стороны, отношение между классами

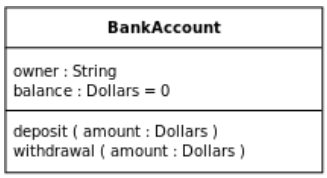
Студент и Вуз является агрегацией, так как студента нельзя удалять при расформировании вуза.

4.2) Диаграмма классов.

Для изображения структуры классов и отношений между ними используют специальные диаграммы. Эти диаграммы демонстрируют все входящие в систему классы, их атрибуты и методы, а также связи классов друг с другом.

Диаграмма классов используется как при разработке общей концепции приложения (его проектировании), так и на этапе перевода концептуальных моделей в программный код. Каждый класс на диаграмме изображается в виде прямоугольника, разделенного на три части. Эти части содержат соответственно

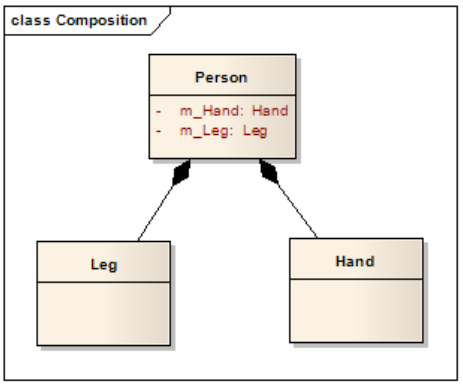
1) Имя класса.
2) Атрибуты класса (его поля).
3) Методы класса.

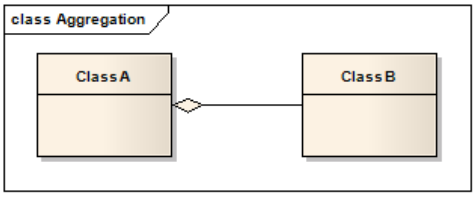


Пример изображения класса на диаграмме показан справа. Класс BankAccount (банковский счет) содержит поля owner (владелец) и balance (баланс), а также методы deposit (пополнить счет) и withdrawal (снять деньги). Тип данных здесь указан после имени поля через двоеточие.

Для каждого элемента класса на диаграмме может быть указан уровень доступа. Для этого перед именем элемента из области public ставится знак «+», из области private – знак «-», из области protected – знак «#».

Отношения на диаграмме изображаются линиями, соединяющими соответствующие классы. Для изображения композиции используют линию, на одном из концов которой располагается ромб со сплошной заливкой. Линия всегда развернута ромбом в сторону класса-контейнера. На рисунке справа показано отношение композиции между классом Person (человек) и классами Hand (рука) и Leg (нога).





Для изображения агрегации рисуют линию, оканчивающуюся полым ромбом (см. рисунок слева).

Для обозначения количества объектов (экземпляров класса), которые могут принимать участие в композиции или агрегации, используется следующая нотация

Обозначение	Число экземпляров
0..1	Нет экземпляров или один экземпляр
1	Ровно один экземпляр
0..*	Ноль и более экземпляров
1..*	Один и более экземпляров

Соответствующие обозначения указывается на обоих концах линии, соединяющей классы.

4.3) Класс-контейнер bookstore.

Вернемся далее к приложению, которое разрабатывалось нами в предыдущих лабораторных работах. На данный момент нами спроектирован и реализован класс book, моделирующий отдельную печатную книгу, а также реализованы некоторые процедуры и функции для работы с книжной коллекцией – динамическим массивом из объектов класса book. А именно, реализованы

1) процедура загрузки коллекции книг из текстового файла,
2) процедура вывода на экран содержимого всей коллекции,
3) функция поиска книги по ее названию.

Предположим теперь, что нам необходимо программно представить ассортимент книг некоторого книжного магазина. С точки зрения покупателя, ассортимент представляет собой информацию обо всех печатных изданиях, имеющихся в наличии в магазине на текущий момент времени. Будем считать, что ассортимент может расширяться за счет поставки новых книг. Предусмотрим для нашей программной модели возможность загрузки данных об ассортименте из файла и записи в файл, вывода всего ассортимента книг на экран, а также поиска необходимой книги по названию.

С точки зрения программиста C++, ассортимент представляет собой коллекцию объектов класса book, к которой добавлены функции файлового ввода/вывода, вывода на экран, поиска и т.д. Для операций над книжной коллекцией разработаем специальный класс-контейнер `bookstore`.

```
class bookstore             // класс книжного магазина
{
private:
    int max_num_books;       // максимальное кол-во книг (вместимость)
    int num_books;           // текущее число книг в магазине
    book *books;             // массив объектов класса book (коллекция книг)

public:
    bookstore(unsigned int max_nb);         // конструктор класса bookstore
    ~bookstore();                           // деструктор класса bookstore

    void add_book(book abook);              // добавить книгу
    void read_from_file(string filename);   // ввести данные из файла
    void write_to_file(string filename);    // записать данные в файл
    void display_all();                     // вывести на экран
    void find_book(string atitle);          // найти книгу по названию
};
```

Указатель `books` в закрытой области этого класса будем использовать для хранения динамического массива объектов `book`. Класс `bookstore` также содержит закрытое поле `max_num_books`, хранящее максимальное число книг в магазине, и закрытое поле `num_books`, показывающее количество книг на данный момент. Предполагается, что значение `num_books` может изменяться по ходу выполнения программы (за счет пополнения ассортимента), тогда как вместимость магазина задается один раз и далее не изменяется.

Рассмотрим далее более подробно методы класса `bookstore`, не обращая пока внимания на конструктор и деструктор (функции `bookstore` и `~bookstore`). В частности, метод `add_book` будет использоваться нами для добавления нового элемента в коллекцию. Этот элемент передается в функцию в качестве параметра.

```
void bookstore::add_book(book abook)
{
	if (num_books < max_num_books)	// можем добавить еще одну книгу?
	{
		books[num_books] = abook;	// заносим книгу в массив
		num_books++;			// увеличиваем счетчик книг
	}
}
```

По условию, массив `books` не может содержать более, чем `max_num_books` элементов, поэтому используем оператор условия для контроля выхода за его пределы.

Рассмотрим теперь метод `read_from_file`, позволяющий загрузить коллекцию целиком из текстового файла. Имя файла передается функции как параметр.

```
void bookstore::read_from_file(string filename)
{
	ifstream infile;
	infile.open(filename);	// открываем файл с заданным именем
	if (!infile.is_open())
	{
		cout << "\n\nФайл данных не найден!" << endl;
		system("pause");
		exit(1);
	}

	int N;
	infile >> N;		// считываем количество книг в файле
	infile.get();		// переход на следующую строку

	for (int i = 0; i < N; i++)
	{
		book new_book;			// создаем новый объект
		new_book.read_from(infile);	// заполняем его данными из файла
		add_book(new_book);		// добавляем в коллекцию
	}
	infile.close();

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

Обратим здесь внимание на внутренний цикл по переменной i, в котором: 1) создается новый объект класса `book`; 2) с помощью метода `read_from` данные об очередной книге загружаются из файла; 3) книга заносится в коллекцию.

Разберем также реализацию метода `find_book`.

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

Легко заметить, что сигнатура функции `find_book` претерпела изменения по сравнению с предыдущим вариантом (см. лаб. работу №3), так как теперь у нас нет необходимости передавать в функцию массив данных и его размер. Так как `find_book` является методом класса `bookstore`, она имеет прямой доступ ко всем его полям, в том числе массиву `books` и переменной `num_books`.

4.4) Конструкторы и деструкторы.

Работа любого класса C++ невозможна без специальных функций, которые называются *конструктором* и *деструктором* класса. Конструктор класса вызывается *неявно* всякий раз, когда новый объект этого класса размещается в оперативной памяти. Задача конструктора – привести объект в корректное начальное состояние. Обычно для этого требуется инициализировать поля объекта некоторыми значениями, однако иногда необходимы и более сложные действия, такие, как выделение памяти для встроенных массивов, вызов конструкторов встроенных объектов и т.д.

Деструктор класса вызывается неявно в момент, когда объект удаляется из памяти. Задача деструктора – выполнить необходимые действия перед уничтожением объекта – освободить используемую внутри объекта динамическую память, закрыть открытые файлы, послать другим объектам сообщения о прекращении своей работы и т.д.

В случае, если программист не определил конструктор и/или деструктор какого-либо класса, он будет создан компилятором автоматически, но при этом не будет выполнять никаких действий, кроме размещения объекта в памяти (конструктор) или удаления из нее (деструктор).

Существует ряд правил, касающихся определения этих методов:

1) конструктор и деструктор всегда объявляются в разделе public;
2) тип возвращаемого значения для них никогда не указывается (в том числе – void);
3) деструктор не может принимать параметров;
4) имя конструктора совпадает с именем класса;
5) имя деструктора совпадает с именем класса, но с приставкой ~ (тильда);
6) в классе допустимо наличие нескольких конструкторов с различающимися параметрами (перегрузка функций), но только один деструктор.

Рассмотрим теперь реализацию конструктора класса `bookstore`, то есть метода `bookstore::bookstore`. Ключевой задачей конструктора будет выделение динамической памяти для массива `books`. Количество элементов в массиве определяется максимальным числом книг и передается в функцию как параметр. Так как в начальный момент времени массив не заполнен, значение `num_books` устанавливается равным нулю.

```
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;
}
```

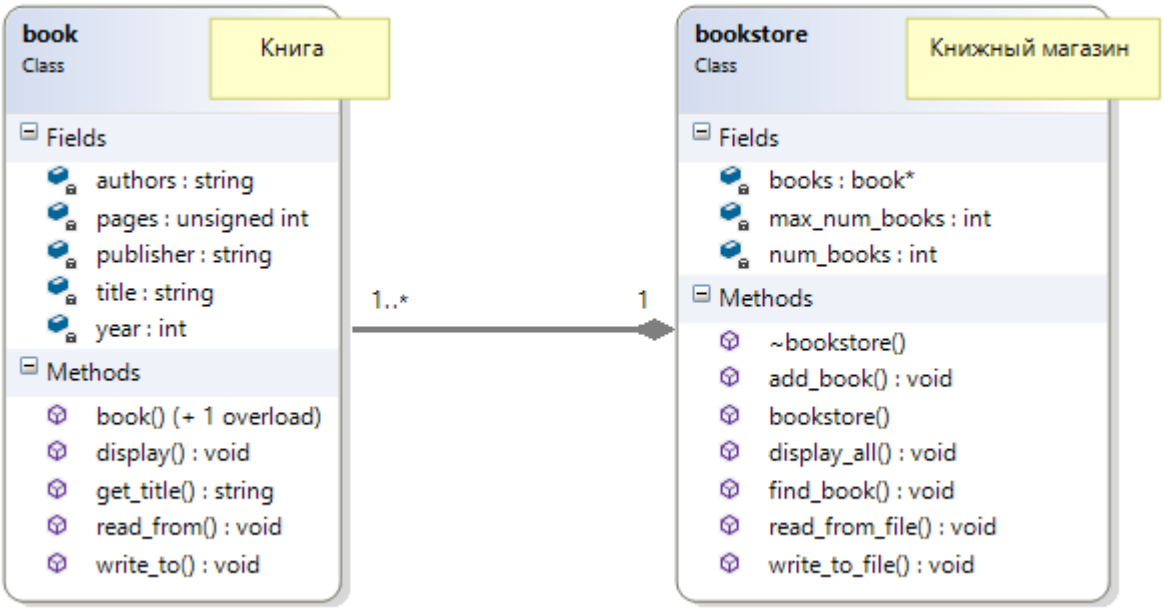
Дополнительная информация выводится на экран для контроля работы конструктора.

В свою очередь, деструктор класса `bookstore` должен освобождать память, выделенную конструктором.

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

4.5) Пример приложения Visual C++.

Рассмотрим теперь целиком программный код проекта, включающего библиотечные модули books (разработан нами ранее) и bookstore, а также основной модуль lab4. Заголовочный файл модуля bookstore содержит объявление класса-контейнера. На рисунке изображена диаграмма классов, соответствующая разработанному приложению.




```
======================== файл books.h ========================
```

```
#ifndef BOOKS_H
#define BOOKS_H

#include <string>
#include <fstream>
using std::string;
using std::ifstream;
using std::ofstream;

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();

	void read_from(ifstream &file);
	void write_to(ofstream &file);
	void display();
};

#endif
```

========== файл books.cpp ==========

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

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;
}

void book::read_from(ifstream &file)
{
	getline(file, title);
	getline(file, authors);
	getline(file, publisher);
	file >> year;
	file >> pages;
	file.get();
}

void book::write_to(ofstream &file)
{
	file << title << endl;
	file << authors << endl;
	file << publisher << endl;
	file << year << endl;
	file << pages << endl;
}

void book::display()
{
	cout << "\n=====================================\n";
	cout << "  Название:     " << title << endl;
	cout << "  Автор(ы):     " << authors << endl;
	cout << "  Издательство: " << publisher << endl;
	cout << "  Год выпуска:  " << year << endl;
	cout << "  Страниц:      " << pages << endl;
}
```

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

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H

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

class bookstore                    // книжный магазин
{
private:
	int max_num_books;             // максимальное кол-во книг в магазине (вместимость)
	int num_books;                 // текущее число книг в магазине
	book *books;                   // массив объектов класса book

public:
	bookstore(unsigned int max_nb);          // конструктор класса bookstore
	~bookstore();                            // деструктор класса bookstore

	void add_book(book abook);               // добавить книгу
	void read_from_file(string filename);    // ввести данные из файла
	void write_to_file(string filename);     // записать данные в файл
	void display_all();                      // вывести на экран
	void find_book(string atitle);           // найти книгу по названию
};

#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::add_book(book abook)
{
	if (num_books < max_num_books)     // можем добавить еще одну книгу?
	{
		books[num_books] = abook;      // заносим книгу в массив
		num_books++;                   // увеличиваем счетчик книг
	}
}
```

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

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

	for (int i = 0; i < N; i++)
	{
		book new_book;
		new_book.read_from(infile);
		add_book(new_book);
	}
	infile.close();

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

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

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

void bookstore::display_all()
{
	cout << "\n\nПОЛНЫЙ АССОРТИМЕНТ КНИЖНОГО МАГАЗИНА \n";
	for (int i = 0; i < num_books; i++)
		books[i].display();
}

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

```
======================== файл lab4.cpp ======================
```

```cpp
#include "stdafx.h"
#include "bookstore.h"

void main()
{
	setlocale(LC_ALL, "rus");
	bookstore my_store(25);

	my_store.read_from_file("my_books.txt");
	my_store.display_all();
	my_store.find_book("Война и мир");

	system("pause");
}
```

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

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №4.

1) Разработайте класс-контейнер, предназначенный для хранения коллекции объектов своей предметной области. Реализуйте следующие методы класса-контейнера:
   - конструктор и деструктор,
   - добавление нового элемента,
   - удаление элемента из коллекции,
   - загрузка из файла и запись в файл,
   - вывод содержимого на экран,
   - поиск данных в коллекции согласно своему индивидуальному заданию.
2) Разместите класс-контейнер в отдельном программном модуле. В заголовочном файле модуля (*.h) поместите объявление класса, в файле реализации (*.cpp) – определения методов класса.
3) В основном модуле программы:
   - создайте экземпляр класса-контейнера,
   - заполните контейнер данными из текстового файла,
   - выведите содержимое контейнера на экран,
   - продемонстрируйте выполнение дополнительных функций из своего задания.

Содержание отчета по лабораторной работе №4.

1) Стандартная «шапка» отчета
2) Цель: формулировка цели работы
3) Теория: краткие сведения о композиции классов с примерами (объем 2-3 стр.)
4) Диаграмма классов с указанием атрибутов, методов, отношений между классами.
5) Программный код
   - объявление класса-контейнера,
   - код конструктора и деструктора класса,
   - код одной из дополнительных функций (любой).