Переход от структур к классам

Цель работы

Цель данной лабораторной работы заключается в освоении навыков:

- создания программных объектов пользовательских типов с использованием классов;
- создания статических библиотек для описания классов;
- ввода-вывода объектов пользовательских типов в файл;
- перегрузки операций потокового ввода-вывода (`<<`, `>>`), операций отношения (`<`, `==`) для пользовательских типов;
- перегрузки конструктора копирования и операции присваивания;
- сортировки массивов объектов пользовательских типов, хранящихся в файле, с использованием перегруженных операций.

Задание

Первая часть

1. Ознакомьтесь с приведенным материалом в (Введении и Пример 1) о переходе от структуры к классу.

2. Разработайте класс "Планета" для планет солнечной системы (4 характеристики планет разного типа приведены в Приложении 1).

   Имя планеты должно иметь тип `char*`. Использование `string` запрещено. Тоже самое касается для полей-строк класса по варианту.

   Память для строк должна выделяться динамически в зависимости от данных (размер выделяемой памяти для поля типа `char*` **не должен задаваться константной**). Размер выделяемой памяти для поля `char*` необходимо определять в зависимости от поступающих данных. Для промежуточного хранения строк с целью считать из файла или с потока `cin` можно воспользоваться массивом `char` с заданной длинной через константу (буфер). Далее на основе определения длинны строки, попавшей в буфер, выделить динамически память для поля `char*`.

   Не забудьте воспользоваться деконструктором, если вы будете использовать динамический массив для строк. Использование деконструктора показано в примере 3.

3. Создайте статическую библиотеку для класса "Планета" и отладить программу, которая создает один объект класса "Планета" и выводит значения его полей на экран. Пример программы приведен в Примере 4.

4. Реализуйте **функции работы с множеством** объектов класса "Планета":

   - чтение БД из файла;
   - запись БД в файл;
   - сортировка БД;
   - добавление нового объекта в БД;
   - удаление объекта из БД;
   - редактирование БД;
   - вывод БД на экран.

   Массив планет и экземпляров класса по варианту должен быть динамическим (**задавать размерность массива константной запрещено**). Размер динамического массива необходимо изменять в зависимости от количества данных.

   Для реализации вышеперечисленных функций, необходимо:

   1. Создать текстовый файл (в блокноте) с данными о планетах солнечной системы и сохранить его в папке проекта. Первоначально, для отладки, введите две записи.

      Обратите внимание на то, что кодировка файла должна быть `ASCII`. Объяснение этого в Приложении 1.

   2. Добавить в программу ввод–вывод объекта класса "Планета" в текстовый файл.

   Функции реализовывать в виде статических методов класса "Планета". Пример использования статических методов представлен в примере 4, а сам процесс перехода к такой декомпозиции - в примере 5.

   Обратите внимание, что в демонстрационном режиме необходимо продемонстрировать **только** все возможности работы **с классом "Планета"**.

Вторая часть

1. Ознакомьтесь с примером 2 перегрузки операции `<<`.
2. Перегрузите конструктор копирования, деструктор и оператор присваивания.
3. Вставьте в конструкторы и деструктор печать типа "Создание (Удаление) ID `n`", где `n` - номер объекта, для которого они вызываются (при реализации данного пункта может быть полезен пример 3).
4. Перегрузите операцию `>>` для класса "Планета" и `ifstream` и прочитайте данные о планетах из файла в массив "Солнечная система" из объектов класса "Планета".

5. Перегрузите операцию `<<` для классов "Планета" и `ofstream` и выведите на экран данные из массива.

Третья часть

1. Перегрузите операции сравнения `<` и `==` для класса "Планета", использовав для этого значение одного из полей.
2. Отсортируйте массив планет солнечной системы, хранящийся в файле, с использованием перегруженных операций.

Четвертая часть

На основе разработанного класса "Планета" выполните задание по варианту (не менее 4 характеристик в классе разного типа). Варианты представлены в приложении 2.

Для класса по варианту организуйте интерфейс взаимодействия пользователя с программой **в виде меню**, позволяющий выполнять следующие действия:

- чтение БД из файла;
- запись БД в файл;
- сортировка БД;
- добавление нового объекта в БД;
- удаление объекта из БД;
- редактирование БД;
- вывод БД на экран.

Обратите внимание, что данный интерфейс используется **только** в интерактивном режиме **для класса по варианту**.

Требования к работе

1. При решении задания лабораторной работы необходимо **создать**
    1. Статическая библиотека с реализацией класса "Планета".
    2. Статическая библиотека с реализацией класса по варианту.
2. Для сдачи лабораторной работы необходимо выполнить все 4 части задания (полная реализация классов "Планета" и по варианту).

Пример 1. Пример перехода от структуры к классу

```
/*
 * Каждая строка файла содержит запись об одном сотруднике. Первая запись в файле
–
 * фактическое число сотрудников. Формат записи:
 * - фамилия (не более 20 позиций),
 * - год рождения (4 позиции),
 * - оклад (не более 8 позиций).
 * Написать программу, которая позволяла бы выводить на экран сведения о
сотрудниках,
 * добавлять и удалять сотрудников из БД, корректировать данные о сотрудниках.
*/

#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

#define l_name  20

struct Man {
    char name[l_name];
    int birth_year;
    float pay;
};

int read_dbase(char *filename, Man *arr, int &n);

int menu();

int menu_f();

void print_dbase(Man *arr, int n);

int write_dbase(char *filename, Man *arr, int n);

/*
int add(Man arr, int n);
int edit(Man* arr, int n);
int remove(Man* arr, int n);
*/

int find(Man *arr, int n, char *name);

int find(Man *arr, int n, int birth_year);

int find(Man *arr, int n, float pay);

void find_man(Man *arr, int n);

//--------------------------------------  Главная функция
int main() {
    const int N = 100;
    Man arr[N];
    char *filename = "dbase.txt";
    int n;
```

```
    //чтение БД в ОП
    if (read_dbase(filename, arr, n)) {
        cout << "Ошибка чтения БД" << endl;
        return 1;
    }
    print_dbase(arr, n);
    while (true) {
        switch (menu()) {
            //case 1: add(arr,n ); break;
            //case 2: remove(arr,n); break;
            case 3:
                find_man(arr, n);
                break;
                //case 4: edit(arr,n); break;
            case 5:
                print_dbase(arr, n);
                break;
            case 6:
                write_dbase(filename, arr, n);
                break;
            case 7:
                return 0;
            default:
                cout << " Недопустимый номер операции" << endl;
                break;
        }
    }
    return 0;
}

////////////////////////////////////////////////////
int menu() {
    cout << " ============== ГЛАВНОЕ МЕНЮ ========================\n";
    cout << "l - добавление сотрудника\t 4 - корректировка сведений" << endl;
    cout << "2 - удаление сотрудника\t\t 5 - вывод базы на экран" << endl;
    cout << "3 - поиск сотрудника\t\t 6 - вывод базы в файл" << endl;
    cout << "\t\t\t\t 7 - выход" << endl;
    cout << "Для выбора операции введите цифру от 1 до 7" << endl;
    int resp;
    cin >> resp;
    cin.clear();
    cin.ignore(10, '\n');
    return resp;
}

// ---------------------------------- Чтение базы из файла
int read_dbase(char *filename, Man *arr, int &n) {
    ifstream fin(filename, ios::in);
    if (!fin) {
        cout << "Нет файла " << filename << endl;
        return 1;
    }
    fin >> n;
    if (n > 100) {
        cout << "Переполнение БД. n= " << n << endl;
        return 1;
```

```
    }
    for (int i = 0; i < n; i++)
        fin >> arr[i].name >> arr[i].birth_year >> arr[i].pay;

    fin.close();
    return 0;
}

//----------------------------------- Вывод базы в файл
int write_dbase(char *filename, Man *arr, int n) {
    ofstream fout(filename, ios::out);
    if (!fout) {
        cout << "Ошибка открытия файла" << endl;
        return 1;
    }
    fout << n;
    for (int i = 0; i < n; i++)
        fout << arr[i].name << ' ' << arr[i].birth_year << ' ' << arr[i].pay <<
endl;

    fout.close();
    return 0;
}

//------------------------------ Вывод базы на экран
void print_dbase(Man *arr, int n) {
    cout << " База Данных " << endl;
    for (int i = 0; i < n; i++)
        cout << setw(3) << i + 1 << ". " << arr[i].name << setw(20 -
strlen(arr[i].name) + 6)
             << arr[i].birth_year << setw(10) << arr[i].pay << endl;
}

//-----------------------------Поиск сотрудника в списке по фамилии
int find(Man *arr, int n, char *name) //возвращает индес элемента с данными о
//сотруднике в БД,реализованной в виде массива
{
    int ind = -1;
    for (int i = 0; i < n; i++)
        if (!strcmp(arr[i].name, name)) {
            cout << arr[i].name << setw(20 - strlen(arr[i].name) + 6)
                 << arr[i].birth_year << setw(10) << arr[i].pay << endl;
            ind = i;
        }
    return ind;
}

//------- Поиск и вывод более старших по возрасту сотрудников
int find(Man *arr, int n, int birth_year) {
    int ind = -1;
    for (int i = 0; i < n; i++)
        if (arr[i].birth_year < birth_year) {
            ind = i;
            cout << arr[i].name << setw(20 - strlen(arr[i].name) + 6)
                 << arr[i].birth_year << setw(10) << arr[i].pay << endl;
        }
```

```
    return ind;
}

//-------- Поиск и вывод сотрудников с окладом, большим чем "pay"
int find(Man *arr, int n, float pay) {
    int ind = -1;
    for (int i = 0; i < n; i++)
        if (arr[i].pay > pay) {
            ind = i;
            cout << arr[i].name << setw(20 - strlen(arr[i].name) + 6)
                 << arr[i].birth_year << setw(10) << arr[i].pay << endl;
        }
    return ind;
}

//--------------------------------------
int menu_f() {
    cout << "\n----------------- ПОИСК -----------------\n";
    cout << "1 - поиск по фамилии  2 - по году рождения\n"
         << "3 - по окладу   4 - конец поиска\n ";
    cout << "Для выбора операции введите число от 1 до 4\n";
    int resp;
    cin >> resp;
    cin.clear();
    cin.ignore(10, '\n');
    return resp;
}

//-------------------------------------  Поиск
void find_man(Man *arr, int n) {
    char buf[l_name];
    int birth_year;
    float pay;

    while (true) {
        switch (menu_f()) {
            case 1:
                cout << "Введите фамилию сотрудника\n";
                cin >> buf;
                if (find(arr, n, buf) < 0)
                    cout << "Сотрудника с фамилией " << buf << " в списке нет\n";
                break;
            case 2:
                cout << "Введите год рождения" << endl;
                cin >> birth_year;
                if (find(arr, n, birth_year) < 0)
                    cout << "В списке нет сотрудников, родившихся до " <<
birth_year << " года\n";
                break;
            case 3:
                cout << "Введите оклад" << endl;
                cin >> pay;
                if (find(arr, n, pay) < 0)
                    cout << "В списке нет сотрудников с окладом, большим " << pay
<< " руб.\n";
                break;
```

```
            case 4:
                return;
            default:
                cout << "Неверный ввод\n";
        }

    }
}
```

В программе, предложенной для решения задачи, при структурном программировании для хранения сведений об одном сотруднике использовалась бы структура `Man`:

```
struct Man {
    char name[l_name];
    int birth_year;
    float pay;
};
```

Начнем с того, что преобразуем эту структуру в класс, так как мы предполагаем, что наш новый тип будет обладать более сложным поведением, чем просто чтение и запись его полей:

```
class Man {
    char name[l_name];
    int birth_year;
    float pay;
};
```

Замечательно. Это у нас здорово получилось! Все поля класса по умолчанию — закрытые (`private`). Так что если клиентская функция `main()` объявит объект `Man man`, а потом попытается обратиться к какому-либо его полю, например: `man.pay = value`, то компилятор быстро пресечет это безобразие, отказавшись компилировать программу. Поэтому в состав класса надо добавить методы доступа к его полям. Эти методы должны быть общедоступными, или открытыми (`public`).

Однако предварительно вглядимся внимательнее в определения полей. В решении задачи на языке Си поле name объявлено как статический массив длиной `l_name`. Это не очень гибкое решение. Мы хотели бы, чтобы наш класс Man можно было использовать в будущем в разных приложениях. Например, если предприятие находится в России, то значение `l_name = 20`, по-видимому, всех устроит, если же приложение создается для некой восточной страны, может потребоваться, скажем, значение `l_name = 200`. Решение состоит в использовании динамического массива символов с требуемой длиной. Поэтому заменим поле `char name[l_name]` на поле `char *pName`. Сразу возникает вопрос: кто и где будет выделять память под этот массив? Вспомним один из принципов ООП: все объекты должны быть самодостаточными, то есть полностью себя обслуживать.

Таким образом, в состав класса необходимо включить метод, который обеспечил бы выделение памяти под указанный динамический массив при создании объекта (переменной типа `Man`). Метод, который автоматически вызывается при создании

экземпляра класса, называется конструктором. Компилятор безошибочно находит этот метод среди прочих методов класса, поскольку его имя всегда совпадает с именем класса.

Парным конструктору является другой метод, называемый деструктором, который автоматически вызывается перед уничтожением объекта. Имя деструктора отличается от имени конструктора только наличием предваряющего символа `~` (тильда).

Ясно, что если в конструкторе была выделена динамическая память, то в деструкторе нужно побеспокоиться об ее освобождении. Напомним, что объект, созданный как локальная переменная в некотором блоке `{ }`, уничтожается, когда при выполнении достигнут конец блока. Если же объект создан с помощью операции `new`, например:

```
Man* pMan = new Man;
```

то для его уничтожения применяется операция `delete`, например: `delete pMan;`.

Итак, наш класс принимает следующий вид:

```
class Man {
public:
    Man(int l_name = 20) { pName = new char[l_name]; } // конструктор
    ~Man() { delete[] pName; }     // деструктор
private:
    char *pName;
    int birth_year;
    float pay;
};
```

Обратим ваше внимание на одну синтаксическую деталь — объявление класса должно обязательно завершаться точкой с запятой (`;`). Если вы забудете это сделать, то получите от компилятора длинный список маловразумительных сообщений о чем угодно, но только не об истинной ошибке.

Рассмотрим теперь одну важную семантическую деталь: в конструкторе класса параметр `l_name` имеет значение по умолчанию (`20`). Если все параметры конструктора имеют значения по умолчанию или если конструктор вовсе не имеет параметров, он называется конструктором по умолчанию. Зачем понадобилось специальное название для такой разновидности конструктора? Разве это не просто удобство для клиента — передать некоторые значения по умолчанию одному из методов класса? Нет! Конструктор — это особый метод, а конструктор по умолчанию имеет несколько специальных областей применения.

Во-первых, такой конструктор используется, если компилятор встречает определение массива объектов, например: `Man man[25];` Здесь объявлен массив из `25` объектов типа `Man`, и каждый объект этого массива при создании вызывает конструктор по умолчанию! Поэтому если вы забудете снабдить класс

конструктором по умолчанию, то вы не сможете объявлять массивы объектов этого класса. Исключение представляют классы, в которых нет ни одного конструктора, так как в таких ситуациях конструктор по умолчанию создается компилятором.

Вернемся к приведенному выше описанию класса. В нем методы класса определены как встроенные (`inline`) функции. При другом способе методы только объявляются внутри класса, а их реализация записывается вне определения класса, как показано ниже:

```
// Man.h (интерфейс класса)
class Man {
public:
    Man(int I_name = 30); // конструктор
    ~Man();   // деструктор
private:
    char *pName;
    int birth_year;
    float pay;
};

// Man.cpp (реализация класса)
#include "Man.h"

Man::Man(int l_name) { pName = new char[l_name]; }

Man::~Man() { delete[] pName; }
```

При внешнем определении метода перед его именем указывается имя класса, за которым следует операция доступа к области видимости `::`. Выбор способа определения метода зависит в основном от его размера: короткие методы можно определить как встроенные, что может привести к более эффективному коду. Впрочем, компилятор все равно сам решит, может он сделать метод встроенным или нет.

Продолжим процесс проектирования интерфейса нашего класса. Какие методы нужно добавить в класс? С какими сигнатурами? На этом этапе очень полезно задаться следующим вопросом: какие обязанности должны быть возложены на класс `Man`?

Первую обязанность мы уже реализовали: объект класса хранит сведения о сотруднике. Чтобы воспользоваться этими сведениями, клиент должен иметь возможность получить эти сведения, изменить их и вывести на экран. Кроме этого, для поиска сотрудника желательно иметь возможность сравнивать его имя с заданным.

Начнем с методов, обеспечивающих доступ к полям класса. Для считывания значений полей добавим методы `GetName()`, `GetBirthYear()`, `GetPay()`. Очевидно, что аргументы здесь не нужны, а возвращаемое значение совпадает с типом поля.

Для записи значений полей добавим методы `SetName()`, `SetBirthYear()`, `SetPay()`. Чтобы определиться с сигнатурой этих методов, надо представить себе, как они будут вызываться клиентом.

Константные методы

Обратите внимание, что заголовки тех методов класса, которые не должны изменять поля класса, снабжены модификатором `const` после списка параметров. Если вы по ошибке попытаетесь в теле метода что-либо присвоить полю класса, компилятор не позволит вам это сделать. Другое достоинство ключевого слова `const` — оно четко показывает сопровождающему программисту намерения разработчика программы. Например, если обнаружено некорректное поведение приложения и выяснено, что "кто-то" портит одно из полей объекта класса, то сопровождающий программист сразу может исключить из списка подозреваемых методы класса, объявленные как `const`. Поэтому использование `const` в объявлениях методов, не изменяющих объект, считается хорошим стилем программирования.

Отладочная печать в конструкторе и деструкторе

Вывод сообщений типа `"Constructor is working"`, `"Destructor is working"` очень помогает на начальном этапе освоения классов. Да и не только на начальном — мы сможем убедиться в этом, когда столкнемся с проблемой локализации неочевидных ошибок в программе.

Перегрузка операций

Любая операция, за исключением `::`, `?:`, `.`, `.*`, определенная в C++, может быть перегружена для созданного вами класса. Это делается с помощью функций специального вида, называемых функциями-операциями (операторными функциями). Общий вид такой функции:

```
возвращаемый_тип operator # (список параметров) { тело функции }
```

где вместо знака `#` ставится знак перегружаемой операции.

Функция-операция может быть реализована либо как функция класса, либо как внешняя (обычно дружественная) функция. В первом случае количество параметров у функции-операции на единицу меньше, так как первым операндом при этом считается сам объект, вызвавший данную операцию. Например, покажем два варианта перегрузки операции сложения для класса `Point`.

Первый вариант — в форме метода класса:

```
class Point {
    double x, y;
public:
    //. . .
    Point operator+(Point &);
};

Point Point::operator+(Point &p) {
    return Point(x + p.x, y + p.y);
}
```

Второй вариант — в форме внешней глобальной функции, причем функция, как правило, объявляется дружественной классу, чтобы иметь доступ к его закрытым элементам:

```
class Point {
    double x, y;
public: //. . .
    friend Point operator+(Point &, Point &);
};

Point operator+(Point &p1, Point &p2) {
    return Point(p1.x + p2.x, p1.y + p2.y);
}
```

Независимо от формы реализации операции `+` мы можем теперь написать:

```
Point p1(0, 2), p2(-1, 5);
Point p3 = p1 + p2;
```

Следует понимать, что, встретив выражение `p1 + p2`, компилятор в случае первой формы перегрузки вызовет метод `p1.operator +(p2)`, а в случае второй формы перегрузки — глобальную функцию operator `+(pl, p2)`.

Результатом выполнения данных операторов будет точка `p3` с координатами `x = -1, y = 7`. Заметим, что для инициализации объекта `p3` будет вызван конструктор копирования по умолчанию, но он нас устраивает, поскольку в классе нет полей-указателей.

Если операция может перегружаться как внешней функцией, так и функцией класса, какую из двух форм следует выбирать? Ответ: используйте перегрузку в форме метода класса, если нет каких-либо причин, препятствующих этому. Например, если первый аргумент (левый операнд) относится к одному из базовых типов (к примеру, `int`), то перегрузка операции возможна только в форме внешней функции.

> Вспомните, что по умолчанию с помощью указателя this в методы класса неявно, скрытым первым параметром, передается адрес объекта, вызвавшего метод.

Пример 2. Перегрузка операции <<

Пример перегрузки операции `<<` для вывода в файл некоторой структуры `element`.

```
// overload.cpp  -  запись  структур в  файл перегруженной
// операцией <<
#include <iostream>
#include <fstream>

using namespace std;
struct element { // Определение некоторой структуры
    int nk, nl;
    float zn;
};

//  Операция-функция,   расширяющая действие операции <<
ofstream &operator<<(ofstream &out, element el) {
    out << ' ' << el.nk << ' ' << el.nl << ' ' << el.zn << '\n';
    return out;
}

int main() {
    const int numbeEl = 5; // Количество структур в массиве
    element arel[numbeEl] = {1, 2, 3.45, 2, 3, 4.56,
                             22, 11, 45.6, 3, 24, 4.33, 3, 6, -5.3};
    // Определяем поток и связываем его с новым файлом abc:
    ofstream filel("abc.txt", ios::app);
    if (!filel) {
        cout << "Неудача при открытии файла abc.\n";
        return 1;
    }
    // Запись в файл abc массива структур:
    for (int i = 0; i < numbeEl; i++)
        filel << arel[i];
    return 0;
}
```

Результат выполнения программы - создание файла с именем `abc.txt` в текущем каталоге и запись в этот файл элементов массива из пяти структур element. Содержимое файла `abc`:

```
 1    2    3.45
 2    3    4.56
 22   11   45.6
 3    24   4.33
 3    6    5.3
```

Пример 3. Статические члены класса

Обратите внимание на то, что счетчик `total` используется для демонстрации жизненного цикла экземпляров класса `gamma`. Также он может быть полезен для отладки того, что вы правильно очищайте память после использования динамического выделения памяти (пара `new` и `delete`).

Поэтому:

- **Не используйте счетчик `total`** для того, чтобы узнать текущее количество элементов в вашей БД.
- **Не используйте значение параметра `id`** для того, чтобы идентифицировать объект в вашей БД. Для идентификации используйте порядковый номер вашего объекта в БД и/или иную информацию, например, название объекта.

```
// overload.cpp  -  запись  структур в  файл перегруженной
// операцией <<
#include <iostream>

using namespace std;

class gamma {
private:
    static int total; //всего объектов
    //(только объявление)
    int id;  //ID текущего объекта
public:
    gamma() //конструктор без аргументов
    {
        total++; //увеличить счетчик объектов
        id = total; //id равен текущему значению total
        cout << "Создание ID " << id << endl;
    }

    ~gamma() //деструктор
    {
        total--;
        cout << "Удаление ID " << id << endl;
    }

    static void showtotal() // статическая функция
    {
        cout << "\nВсего: " << total << endl;
    }

    void showID() // нестатическая функция
    {
        cout << "\nID: " << id << endl;
    }
};

//---------------------------------
int gamma::total = 0;

int main() {
    gamma::showtotal();
    gamma g1;
    g1.showtotal();
    gamma g2;
    gamma g3;
    g3.showtotal();
    g1.showID();
    g2.showID();
    g3.showID();
    cout << "Конец программы" << endl;
    return 0;
}
```

Пример 4. Пример программы, использующей класс Planet

```
#include <iostream>
#include <fstream>
#include "planet.h"

using namespace std;

int read_db(char *, Planet *, const int);

int menu();

void print_db(Planet *, int);

int write_db(char *, Planet *, int);

int find(Planet *, int);

void sort_db(Planet *, int);

const int Size = 12;
const int l_record = 80;

int main() {
    char *file_name = "sunsys.txt";
    Planet planets[Size];
    int n_planet;
    int ind;
    while (true) {
        switch (menu()) {
            case 1:
                n_planet = read_db(file_name, planets, Size);
                break;
            case 2:
                write_db(file_name, planets, n_planet);
                break;
            case 3:
                if ((ind = find(planets, n_planet)) >= 0)
                    planets[ind].edit();
                else
                    cout << "Такой планеты нет" << endl;
                break;
            case 4:
                print_db(planets, n_planet);
                break;
            case 5:
                sort_db(planets, n_planet);
                break;
            case 6:
                return 0;
            default:
                cout << " Неправильный ввод" << endl;
                break;
        }
    }
    return 0;
}
```

Пример 5. Декомпозиция проекта при работе с множествами объектов

Исходная задача

Предположим, что у нас есть класс `Rectangle`, в котором описаны некоторые его свойства и методы взаимодействия с ними:

```
#include <iostream>

class Rectangle {
private:
    int a_, b_;
public:
    Rectangle() {}

    Rectangle(int a, int b) {
        this->a_ = a;
        this->b_ = b;
    }

    int getA() {
        return a_;
    }

    int getB() {
        return b_;
    }

    void setA(int a) {
        this->a_ = a;
    }

    void setB(int b) {
        this->b_ = b;
    }

    void set(int a, int b) {
        setA(a);
        setB(b);
    }
};

int main() {
    Rectangle *rec = new Rectangle(1, 3);
    std::cout << "I know about the rectangle. It has: "
              << std::endl;
    std::cout << " a=" << rec->getA() << std::endl;
    std::cout << " b=" << rec->getB() << std::endl;
    delete rec;
}
```

Обратите внимание, что названия `private` переменных имеют постфикс `_`, чтобы отличать их от общедоступных. Также названий переменных и методов используется верблюжий (`CamelCase`) стиль.

Предположим, что нам поставили следующую **задачу**: обрабатывать в программе множество объектов класса `Rectangle`.

Подход 1: хранение множества объектов в месте его использования

Объявление множества в функции `main`, а его обработчиков - в `main.cpp`

Первое предположение о решении задачи, которое может прийти на ум, может быть желанием хранить нужное нам множество там, где мы с ним будем работать. В нашем случае делать это в функции `main`. В том же файле, где содержится функция `main` (`main.cpp`), расположить функции работы с нашим множеством. Итоговый результат будет таким:

```
#include <iostream>

class Rectangle {
private:
    int a_, b_;
public:
    Rectangle() {}

    Rectangle(int a, int b) {
        this->a_ = a;
        this->b_ = b;
    }

    int getA() {
        return a_;
    }

    int getB() {
        return b_;
    }

    void setA(int a) {
        this->a_ = a;
    }

    void setB(int b) {
        this->b_ = b;
    }

    void set(int a, int b) {
        setA(a);
        setB(b);
    }

};

void showRectangles(Rectangle *arr, int n) {
    for (int i = 0; i < n; i++) {
        std::cout << "I know about the rectangle # " << i
                  << ". It has: " << std::endl;
        std::cout << " a=" << arr[i].getA() << std::endl;
        std::cout << " b=" << arr[i].getB() << std::endl;
    }
}

int calculateAreas(Rectangle *arr, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i].getA() * arr[i].getB();
    }
    return sum;
}

int main() {
    int n = 2;
    Rectangle *arr = new Rectangle[n];
```

```
    arr[0].set(1, 3);
    arr[1].set(4, 6);

    showRectangles(arr, n);
    std::cout << std::endl;
    std::cout << "Sum of the areas of all rectangles: "
              << calculateAreas(arr, n) << std::endl;
    delete[] arr;
}
```

Функция `showRectangles` у нас показывает информацию о всех прямоугольниках в множества, `calculateAreas` - суммирует площади прямоугольников из множества.

А что делать, если в нашей программе мы будем работать не только с объектами класса `Rectangle`, но и с `Circle`, `Triangle`, `Square`? Тогда в обработчики множеств объектов соответствующих классов нам также придется размещать в `main.cpp`. Чем это грозит?

1. Нам придется **придумывать длинные названия функций**, которые выполняют одни и те же действия, но для разных объектов. Например, для расчеты суммы площадей множеств нам придется использовать названия `calculateAreasRectangle`, `calculateAreasTriangle` и т.д. Но в C++ имеется **механизм перегрузки функций**, который позволит нам использовать имя функции `calculateAreas` для разных множеств.

   Перегрузка функций — это механизм C++, благодаря которому функции с разным количеством или типами параметров могут иметь одинаковое имя (идентификатор).

2. Даже если мы применили механизм перегрузки функций, мы сталкиваемся с **ростом объема** `main.cpp`. В лабораторной работе №1 мы познакомились о том, что для декомпозиции проекта можно использовать механизм статических библиотек классов. Значит мы можем унести наши обработчики множеств на уровень статических библиотек и разместить их по соседству с объявлением соответствующих классов или в самих классах.

Объявление множества в функции `main`, а его обработчики сделать методами объектов класса `Rectangle`

Остановимся на том, что функции работы с множествами мы будем размещать в самих классах. Для примера мы продолжим использовать файл `main.cpp`. Для класса `Rectangle` результат наших преобразований будет выглядеть следующим образом:

```
#include <iostream>

class Rectangle {
private:
    int a_, b_;
public:
    Rectangle() {}

    Rectangle(int a, int b) {
        this->a_ = a;
        this->b_ = b;
    }

    int getA() {
        return a_;
    }

    int getB() {
        return b_;
    }

    void setA(int a) {
        this->a_ = a;
    }

    void setB(int b) {
        this->b_ = b;
    }

    void set(int a, int b) {
        setA(a);
        setB(b);
    }

    void showRectangles(Rectangle *arr, int n) {
        for (int i = 0; i < n; i++) {
            std::cout << "I know about the rectangle # " << i
                      << ". It has: " << std::endl;
            std::cout << " a=" << arr[i].getA() << std::endl;
            std::cout << " b=" << arr[i].getB() << std::endl;
        }
    }

    int calculateAreas(Rectangle *arr, int n) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += arr[i].getA() * arr[i].getB();
        }
        return sum;
    }
};

int main() {
    int n = 2;
    Rectangle *arr = new Rectangle[n];
    arr[0].set(1, 3);
```

```
    arr[1].set(4, 6);

    arr[0].showRectangles(arr, n);
    std::cout << std::endl;
    std::cout << "Sum of the areas of all rectangles: "
              << arr[0].calculateAreas(arr, n) << std::endl;
    delete[] arr;
}
```

В этой реализации мы сталкиваемся с несколькими проблемами:

- **Первая проблема** данных методов, о которой мы можем увидеть, это то, что **вызов методов работы с массивами происходит через конкретный объект**. Иными словами, нам для вызова метода работы с массивом нужно обратиться к какому-то объекту из этого массива и вызвать у него требуемый метод.
- **Вторая вытекающая проблема** заключается в том, что методы работы с массивами зачем-то имеют доступу к конкретному экземпляра класса, через который вызываются данные методы. А это может повлечь **проблему безопасной работы с конкретным объектом** в случае внесения каких-то изменений по неосторожности. Пример, где вы можете столкнуться с проблемами безопасного управления данными: сортировка множества.

Решить эти проблемы можно достаточно просто: сделать эти методы статическими методами класса!

Демонстрация использования статических методов представлена в примере 3.

Объявление множества в функции `main`, а его обработчики сделать статическими методами класса `Rectangle`

Результат применения статических методов выглядит следующим образом:

```
#include <iostream>

class Rectangle {
private:
    int a_, b_;
public:
    Rectangle() {}

    Rectangle(int a, int b) {
        this->a_ = a;
        this->b_ = b;
    }

    int getA() {
        return a_;
    }

    int getB() {
        return b_;
    }

    void setA(int a) {
        this->a_ = a;
    }

    void setB(int b) {
        this->b_ = b;
    }

    void set(int a, int b) {
        setA(a);
        setB(b);
    }

    static void showRectangles(Rectangle *arr, int n) {
        for (int i = 0; i < n; i++) {
            std::cout << "I know about the rectangle # " << i
                      << ". It has: " << std::endl;
            std::cout << " a=" << arr[i].getA() << std::endl;
            std::cout << " b=" << arr[i].getB() << std::endl;
        }
    }

    static int calculateAreas(Rectangle *arr, int n) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += arr[i].getA() * arr[i].getB();
        }
        return sum;
    }
};

int main() {
    int n = 2;
    Rectangle *arr = new Rectangle[n];
    arr[0].set(1, 3);
```

```
    arr[1].set(4, 6);

    Rectangle::showRectangles(arr, n);
    std::cout << std::endl;
    std::cout << "Sum of the areas of all rectangles: "
              << Rectangle::calculateAreas(arr, n) << std::endl;
    delete[] arr;
}
```

Теперь вызов любого метода работы с множеством объектов никак не нарушит данные объектов массива.

Мы рассмотрели один из возможных подходов для декомпозиции проекта, в котором происходит обработка множества объектов класса. Этот подход можно применить для выполнения данной лабораторной работы.

Исходные данные

Обратите внимание на то, что кодировка файла должна быть ASCII.

Если ваш файл будет иметь кодировку UTF со спецификацией (UTF with BOM) и в первой строчке вашего файла вы расположите количество элементов в БД, то вы при попытке считать значение количества записей в БД из первой строки получите значение 0!

Name	Diameter	Life	Moons
Mercury	4878	0	0
Venus	12104	0	0
Earth	12774	1	1
Mars	6786	1	2
Jupiter	142796	0	16
Saturn	120000	0	17
Uranus	51108	0	5
Neptune	49600	0	2
Pluto	2280	0	1

Приложение 2. Варианты

Вариант	Тема
1	квартира, как объект для агентства
2	автобус в автопарке
3	анкета для опроса населения
4	компьютер
5	кандидат, участвующий в выборах
6	железнодорожный билет
7	файл на диске
8	книга в библиотеке
9	политическая партия
10	кафедра института
11	автомобиль
12	авиабилет
13	статья в журнале
14	магазин
15	абонент телефонной станции
16	управление каталогом в файловой системе
17	пациент в поликлинике
18	дом, как объект ЖЭКа
19	строительная бригада
20	пищевой набор диеты
21	дорога
22	музыкальный альбом
23	программное обеспечение
24	заявки на выполнение работ

Вариант	Тема
25	товары в магазине
26	участник соревнований
27	фильмотека
28	винотека