

ЛАБОРАТОРНАЯ РАБОТА 1

ФОРМУЛИРОВКА ЗАДАЧИ Используя файлы, сохранить набор структурированных данных и произвести над его элементами некоторые вычисления (выбор/поиск по критериям, сортировку и тому подобные).

ПОЯСНЕНИЯ

В рамках данной лабораторной используем постоянное хранилище данных, выбрав для этого файлы. Можно считать, что создаём некоторое начальное приближение к «базам данных».

Теперь при начале работы, программа должна будет загрузить данные из файла, в котором сохранён массив структурных объектов. Если файл существует и данные в нём корректны (подходят для продолжения работы, обсуждается далее) или отсутствуют (файл есть, но пустой), переходим к меню:

Работа с итоговой программой **должна** происходить следующим образом. При её запуске должен быть предложен выбор действий (некоторый аналог меню в текстовой консоли). Пять из них — общие для всех заданий, и одно — индивидуальное. В виде примера, что может быть показано в консоли при вызове программы:

```
Element item: Game{score, attempts, player}
```

```
Items size: 5
```

```
-----  
Add many items           (1)  
Add one item             (2)  
Print items              (3)  
Remove all               (4)  
Do something             (5)  
Print item by number    (6)  
Exit                    (0)  
-----
```

Your option:

Первая строка приведена просто для примера объяснения, с какими составными объектами предстоит работа. Вторая строка показывает количество уже записанных в файл объектов (в данном примере, объектов типа **Game**, описание типа – чуть ниже). Числами пронумерованы действия, которые должны быть реализованы:

- Последовательное добавление нескольких значений в файл. Сколько этих значений будет — спрашивается у пользователя.
- Добавление только одного объекта в файл.

- Печать в текстовом виде всех сохранённых объектов (печать значений их полей).
- Удаление всех сохранённых элементов из файла.
- Под «**Do something**» как раз обозначено выполнение вычислений из индивидуального задания.
- Вывод элемента сохранённых данных под заданным номером.
- Завершение работы программы.

ВСПОМОГАТЕЛЬНЫЕ ФРАГМЕНТЫ

Далее в примерах рассматривается структура:

```
1 const size_t STR_SZ = 30;
2
3 struct Game
4 {
5     int score;
6     unsigned attempts;
7     char player[STR_SZ + 1];
8 };
```

Типичным подходом к организации сеанса взаимодействия с пользователем является запрос действия от него в бесконечном цикле. И прерывание работы этого цикла по проверке некоторого условия. Общая идея:

```
1 const int stop_code = 0;
2
3 int choose_option();
4
5 int main()
6 {
7     while (true) {
8         option = choose_option();
9
10        // проверки выбранной опции для набора действий
11
12        if (option == stop_code) {
13            break;
14        }
15    }
16 }
```

Однако, куча конструкций **if-else** смотрятся не так упорядоченно, как использование перечислений и конструкции **switch**. В этом случае, можно составить следующий шаблон для данной лабораторной:

```
1 #include <stdio>
2
3 enum class MenuOpt {
4     EXIT, ADD_MANY, ADD_ONE, PRINT, REMOVE, DO
5 };
6
7 void print_menu();
8
9 int main()
10 {
```

```

11  bool stop = false;
12  while (!stop) {
13      print_menu();
14
15      MenuOpt opt;
16      scanf("%d", &opt);
17
18      switch (opt) {
19          case MenuOpt::ADD_MANY:
20              printf("<add many> action is chosen\n");
21              break;
22          case MenuOpt::ADD_ONE:
23              printf("<add open> action is chosen\n");
24              break;
25          case MenuOpt::PRINT:
26              printf("<print> action is chosen\n");
27              break;
28          case MenuOpt::REMOVE:
29              printf("<remove> action is chosen\n");
30              break;
31          case MenuOpt::DO:
32              printf("<specific> action is chosen\n");
33              break;
34          case MenuOpt::EXIT:
35              printf("<exit> action is chosen\n");
36              stop = true;
37              break;
38      }
39  }
40 }
41
42 void print_menu()
43 {
44     printf(
45         "\nElement item: Game{score, attempts}\n"
46         "-----\n"
47         "Add many items (%d)\n"
48         "Add one item   (%d)\n"
49         "Print items    (%d)\n"
50         "Remove all     (%d)\n"
51         "Do something   (%d)\n"
52         "Exit           (%d)\n"
53         "-----\nYour option: ",
54         MenuOpt::ADD_MANY, MenuOpt::ADD_ONE, MenuOpt::PRINT,
55         MenuOpt::REMOVE,   MenuOpt::DO,      MenuOpt::EXIT
56     );
57 }

```

Естественно, это не единственный возможный шаблон. Он демонстрирует базовые принципы: определение перечисления с ограниченным числом вариантов, печать понятного меню, выполнение различных действий в зависимости от выбора. Конечно, в индивидуальном задании ещё добавляется определение самой структуры, замена **printf**ов на вызовы функций, которые будут осуществлять требуемые действия и некоторые другие действия. Но структура будет похожей на показанный пример.

Для организации сохранения информации в файлах в первую очередь нужно продумать и выбрать формат для данных: *текстовый* или *двоичный*. У каждого варианта есть и свои достоинства, и подводные камни. В рамках данной лабораторной рекомендуется полностью сосредоточиться на **текстовом формате**. При таком выборе, файл с данными можно будет просмотреть любым текстовым редактором.

Теперь нужно выбрать способ, как поля составного объекта будут представлены в файле. Здесь можно проявлять фантазию, от представления будет зависеть только сложность загрузки данных из файла в объект структуры. Тем не менее, приведём два относительно простых способа.

Поля в строке, разделённой специальными символами. Условный файл выглядит так:

```
5 || 8 || monkey69
85 || 120 || kingofgames
```

Тогда, чтобы загрузить данные в структурный объект C++, нужен следующий вызов **fscanf**:

```
1 Game obj;
2 fscanf(db_stream, "%d || %u || %30[^\n]", &obj.score, &obj.attempts, obj.player);
```

Минусы способа: не слишком наглядно при просмотре файла в текстовом редакторе.

Каждому полю — отдельная именованная строка. Файл выглядит так:

```
score: 5
attempts: 8
player: monkey69
=====
score: 85
attempts: 120
player: kingofgames
=====
```

Здесь набор из повторяющихся знаков присваивания добавлен в целях визуального разделения отдельных структур. В этом случае, загрузка данных будет чуть подлиннее **fscanf**:

```
1 Game obj;
2 fscanf(db_stream, "%*[^:]: %d", &obj.score);
3 fscanf(db_stream, "%*[^:]: %u", &obj.attempts);
4 fscanf(db_stream, "%*[^:]: %30[^\n]", obj.player);
5 fscanf(db_stream, "%*[^:]: %30[^\n]");
```

В примере комбинация «**%*[^:]**» означает следующее:

- знак «*****» после процента говорит о том, что символы будут выбраны из потока, но не будут сохранены ни в какую переменную;

- в квадратных скобках указано условие, что выбираем все символы **до знака двоеточия**;
- дополнительное двоеточие после закрывающейся квадратной скобки говорит о том, что и этот символ будет выбран, но нигде не сохранён.

Это способ расширенного сканирования входящего набора символа (применимо и для **scanf**, в том числе) для сохранения только тех значений, что реально нам нужны из всей входящей информации. К слову, 4 вызова **fscanf** тоже можно свести к одному:

```
1 Game obj;
2 fscanf(db_stream, "%[^:]: %d%[^:]: %u%[^:]: %30[^\n]*[^\n]",
3        &obj.score, &obj.attempts, obj.player);
```

Не так наглядно, но результат одинаков.

Запись в файл информации в нужном формате в обоих случаях делается тривиально с помощью **fprintf**. Каким путём пойти в конкретном задании — решать вам.

Когда программа получает имя файла, где сохранены данные, хотелось бы проверить, что эти данные корректны, и получить количество сохранённых объектов. Корректность нарушается тогда, когда во время чтения вызовы **fscanf** приводят к ошибке. И эта ошибка не связана с достижением конца файла. И для проверки корректности, и для подсчёта числа структур приходится анализировать файл, поэтому эти оба действия можно сделать за один проход:

```
1 // поскольку в структуре только три поля,
2 // функция *fscanf* вернёт число 3, когда
3 // разбор значений для каждого поля прошёл успешно.
4 const int expected = 3;
5
6 bool success_reading = true;
7 size_t items_count = 0;
8 while ( !feof(db_stream) ) {
9     if (fscanf(db_stream, "...", ...) == expected) {
10         items_count++;
11     }
12 }
13
14 if (ferror(db_stream) != 0) {
15     success_reading = false;
16 }
17
18 // После выполнения кода выше, мы знаем и успешно ли был разобран
19 // файл с данными, и количество структурных объектов в файле.
```

Здесь использована функция **ferror**: она позволяет проверить поток на ошибки чтения из файла. В том числе, в случае **fscanf**, и на ошибки преобразования группы символов в ожидаемое значение (ждали числа, подсунули набор букв и тому подобное). Функция возвращает **нуль**, если ошибок нет, и **ненулевое значение** в противоположном случае. В коде выше показана не конечная функция, а выражена идея. Из-за этого внутри **fscanf** форматная строка и аргументы

отсутствуют.

Ещё пара функций из `<cstdio>`, касающихся файлов, которые могут помочь при разработке индивидуальных программ:

```
(1) int remove(const char *file_name);  
(2) int rename(const char *old_name, const char *new_name);
```

- **(1)**: удаляет файл с заданным именем (имя файла в смысле функции `fopen`, то есть может включать в себя и путь к файлу);
- **(2)**: переименовывает файл.

Обе функции возвращают **нуль**, если операция завершилась успешно. Иначе — **ненулевое значение**.

Как программа будет получать имя файла при начале работы. Чтобы не вводить название файла, с которым будет работать программа, воспользуемся механизмом, называемом **аргументы командной строки**. При запуске любой программы в ОС, ей может быть передано произвольное количество непрерывных символьных групп (т.е. групп символов, не разделённых пробелами). Программа может обработать переданные ей значения, может проигнорировать их. Если запускаем программу в **текстовом режиме работы с ОС** (т.е. через консоль), то передача выглядит следующим образом:

```
|> prog.exe arg another_arg -vvvv ##
```

В примере исполняемому файлу по имени **prog.exe** были переданы 4 группы символов, разделённых пробелами. Эти группы и называются **аргументами командной строки**.

В C++ для получения подобных аргументов в своей программе используется функция **main**. В стандарте языка объявлены две её формы:

1. игнорирование аргументов командной строки. В этом случае список параметров — пуст;
2. получение аргументов командной строки. В этом случае по стандарту функция **main** имеет два параметра: количество аргументов (целое число типа **int** и массив строк (где в каждом элементе содержится отдельный аргумент). По соглашению, первый аргумент принято именовать **argc** (видимо сокращение от «arguments count»), второй же — **argv** (возможно от «argument values»).

Вторая форма в коде может быть представлена двумя способами:

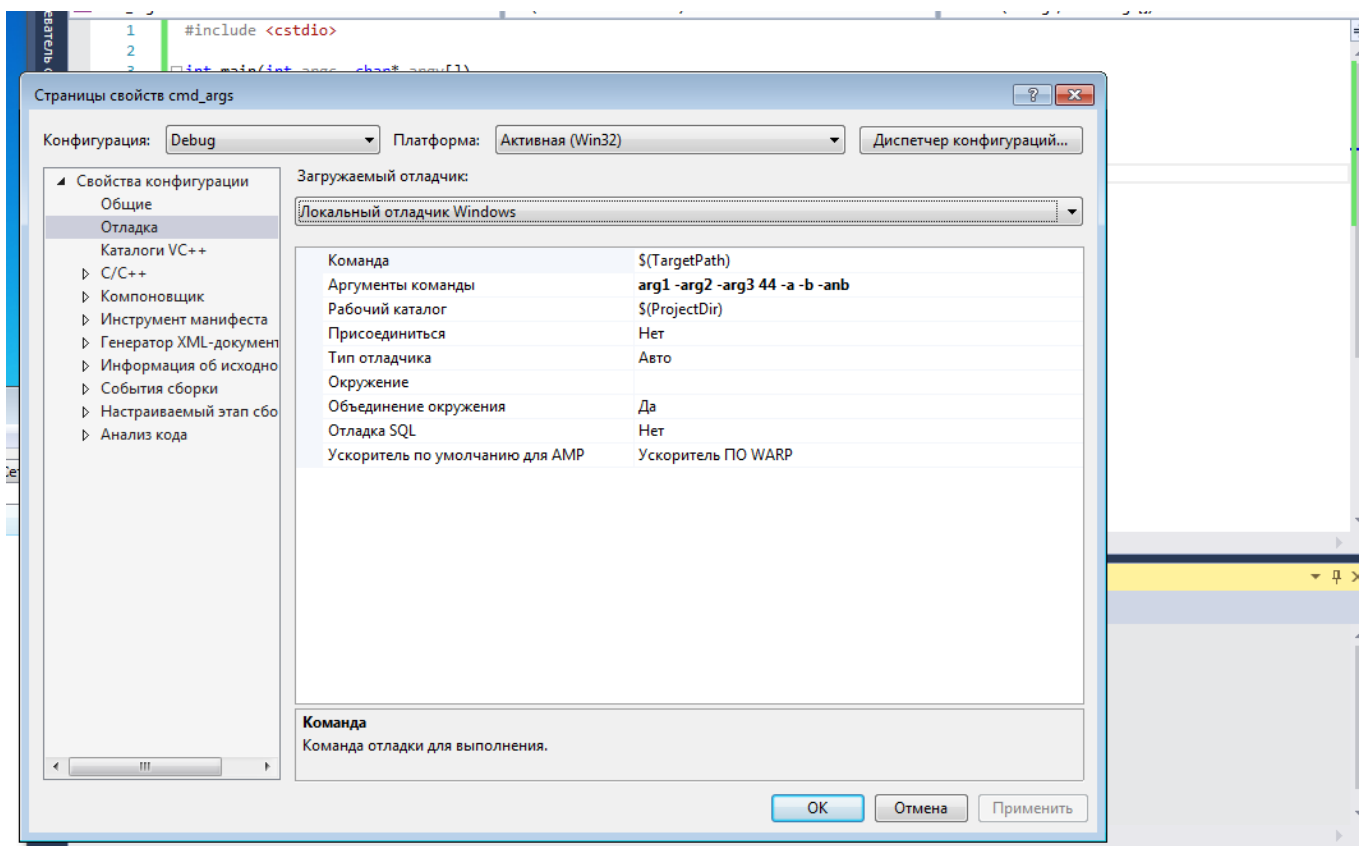
```
1 int main (int argc, char *argv[])  
2 int main (int argc, char **argv)
```

Разница только визуальная, как удобнее смотреть на массив строк.

Кроме переданных программе аргументов командной строки, в массив **argv** на первое место всегда добавляется имя исполняемого файла. Даже если не передан ни один аргумент, этот массив будет не пустым, и параметр **argc** будет равен **единице**. Пример программы, демонстрирующей получение аргументов командной строки:

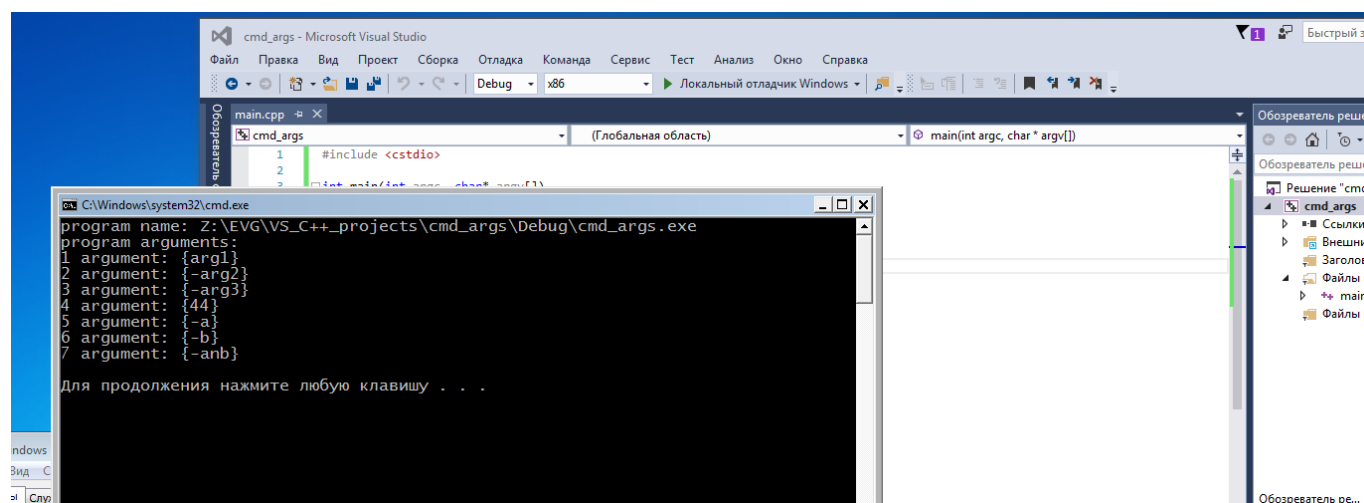
```
1 #include <stdio>
2
3 int main(int argc, char *argv[])
4 {
5     printf("program name: %s\n", argv[0]);
6
7     if (argc > 1) {
8         printf("program arguments:\n");
9         for (int i = 1; i < argc; i++) {
10             printf("%d argument: %s\n", i, argv[i]);
11         }
12         printf("\n");
13     }
14 }
```

Если в IDE в проекте скомпилировать данный пример и начать запускать, то в консоли появится только название программы (скорее всего, увидите полный путь к исполняемому файлу). Для того, чтобы добавить аргументы командной строки, в средах разработки надо изменить свойства проекта. В Visual Studio это делается через меню «Проект» => «Свойства ...» и в открывшемся окне интересуют вкладка «Отладка» (для англ. версии IDE: «Project» => «... Properties» => «Debugging»). Вместо троеточия будет написано имя проекта. Открывшееся окно будет выглядеть примерно так:



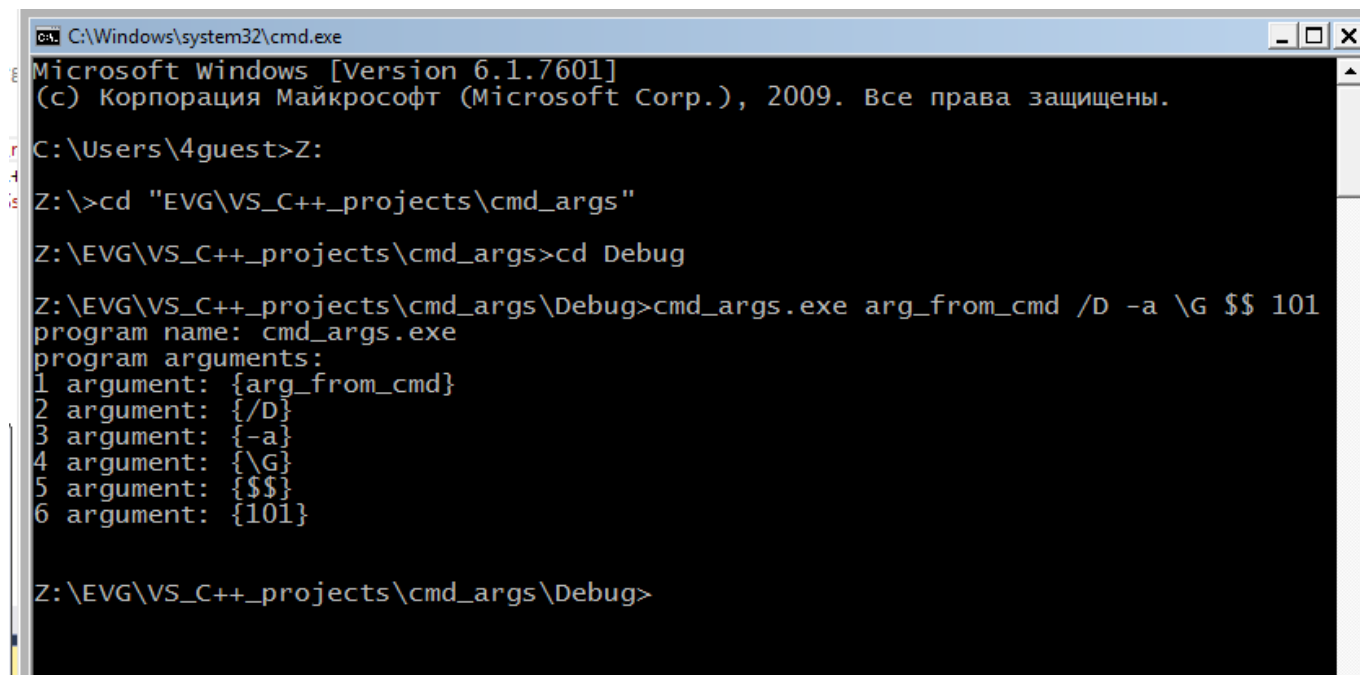
И аргументы задаются в поле «Аргументы команды» просто как текстовая строка. Каждый из них отделяется от другого пробелом. Тут стоит обратить внимание на левый верхний угол, где идёт выбор конфигурации. В средах разработки, как правило, аргументы командной строки задаются отдельно для каждой конфигурации: «Debug», «Release» и другие. В рамках текущего курса все проекты реализовывались для конфигурации «Debug».

Если указанное поле чем-нибудь заполнить, скомпилировать программу и запустить через Visual Studio, то должны увидеть консоль со следующей информацией:



И, кому интересно, на следующем изображении демонстрируются шаги по за-

пуску той же программы через командную строку Windows (**cmd**):



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\4guest>Z:
Z:\>cd "EVG\VS_C++_projects\cmd_args"
Z:\EVG\VS_C++_projects\cmd_args>cd Debug
Z:\EVG\VS_C++_projects\cmd_args\Debug>cmd_args.exe arg_from_cmd /D -a \G $$ 101
program name: cmd_args.exe
program arguments:
1 argument: {arg_from_cmd}
2 argument: {/D}
3 argument: {-a}
4 argument: {\G}
5 argument: {$$}
6 argument: {101}

Z:\EVG\VS_C++_projects\cmd_args\Debug>
```

Аргументы командной строки предоставляют один из способов передачи входных данных (для решения конкретной задачи) без использования явных запросов от пользователя. В рамках данной лабораторной через них будем получать имя исходного файла с данными. Общая схема использования аргументов проста: определяем, какое количество аргументов нам нужно; проверяем, какое количество реально передано при запуске программы; если данных недостаточно, сообщаем о ситуации и прекращаем работу программы.

Кому повезло с предыдущим заданием и попалась сортировка, помните, что для исходного файла с данными её возможно сделать с использованием двух дополнительных файлов (кроме исходного). А как только получен файл с отсортированными объектами, исходный файл удаляем, а полученный переименовываем в исходный.

ТРЕБОВАНИЯ К ПРОГРАММАМ

- Неограниченное использование дополнительных файлов.
- Имя файла с данными передаётся через аргументы командной строки.
- Добавление данных (первое и второе действия из меню выше) должно происходить в конец текущего файла.
- Любой вывод элементов массива, будь то печать всех элементов или печать элементов, найденных по заданию, предполагает вывод всех полей структур. Формат вывода должен обеспечивать удобное восприятие данных. Если элементов, которые должны быть выведены, нет, то программа должна выдавать соответствующее сообщение.

ЗАДАНИЯ

Примечания.

- Не всегда одна запись, описывающая поля нужной структуры, соответствует ровно одному полю структурного типа в программе. Например, «ФИО» можно задавать как три отдельных поля. Обратить внимание на такой момент.
- Для заданий, где присутствуют даты и некоторые вычисления между ними. Идеальный вариант — учитывать реальное количество дней в месяцах и существование високосных годов. В крайнем случае, вначале, в целях упрощения реализации вычислений, можно предполагать, что каждый месяц всегда состоит из 30 дней (год - 12 месяцев - 30 дней в каждом). Однако, все вычисления с датами должны быть *скрыты* в функции таким образом, чтобы переход от упрощенного варианта к идеальному требовал внесения изменений только во внутреннюю *реализацию* этих функций.

1.1. Структура: информация об авиарейсе

Поля:

- номер
- время вылета
- марка самолёта
- длительность полёта
- расстояние

Задача: найти авиарейсы с максимальной и минимальной средней скоростью полёта.

1.2. Структура: информация об абоненте интернет-провайдера

Поля:

- ФИО
- номер договора
- дата подключения (число, месяц, год)
- текущий баланс

Задача: найти всех абонентов с балансом меньше нуля.

1.3. Структура: информация об автомобиле

Поля:

- производитель
- название модели
- год выпуска
- дата регистрации (число, месяц, год)

Задача: ввести конкретную дату и выбрать все автомобили, зарегистрированные не более двух лет назад от неё.

1.4. Структура: информация о численности населения

Поля:

- страна
- численность (в млн. жителей)
- площадь (в тысячах квадратных километров)

Задача: определить страны с максимальной и минимальной плотностью населения.

1.5. Структура: информация о студентах

Поля:

- ФИО
- номер группы
- общее количество экзаменов
- общая сумма оценок за экзамены (оценивание по пятибальной шкале, отсюда есть некое ограничение на максимальное значение этого поля в зависимости от предыдущего)

Задача: ввести некоторое число от 1 до 5 и выбрать всех студентов, чей средний балл превышает заданное значение.

1.6. Структура: информация о студентах

Поля:

- ФИО
- номер группы
- год поступления
- номер курса

Задача: отсортировать заданный массив по году поступления. Направление сортировки (возрастание/убывание годов) — запрашивается у пользователя. Внутри одного года упорядочение идёт по группам в алфавитном порядке.

1.7. Структура: информация об игроках CS:GO

Поля:

- никнейм игрока
- количество игр
- количество поражений
- игровой стаж (в количестве месяцев)

Задача: найти заданное пользователем количество игроков (например - три, пять, два) с максимальным процентом побед. Показать информацию о них в порядке убывания процента побед.

1.8. Структура: информация об университетском курсе

Поля:

- название дисциплины
- длительность (в семестрах, максимальное количество - два семестра)
- число заданий в каждом семестре (может быть различным)
- форма проверки

Задача: найти пять наиболее «лёгких» курсов — тех, у которых среднее число заданий в семестре меньше, чем в остальных.

1.9. Структура: информация об автомобиле

Поля:

- производитель
- название модели
- год выпуска
- стоимость (в условных единицах)

Задача: найти производителя с самой высокой средней стоимостью автомобилей.

1.10. Структура: информация о продуктовом товаре

Поля:

- название
- цена
- дата производства (число, месяц, год)
- срок годности (в неделях)

Задача: задать дату и пороговое значение срока годности. Среди введенных товаров выбрать все, срок годности которых меньше заданного числа.

1.11. Структура: информация о продуктовом товаре

Поля:

- название
- цена
- дата производства (число, месяц, год)
- срок годности (в неделях)

Задача: задать дату и найти все просроченные товары.

1.12. Структура: информация о сотрудниках

Поля:

- ФИО
- должность
- адрес проживания
- трудовой стаж (в месяцах)

Задача: по заданному пользователем набору фамилий вывести информацию обо всех сотрудниках, чьи фамилии совпадают с введённым набором. Продумать, как организовать ввод фамилий для поиска (запрашивать заранее количество или позволять вводить слова до некоторого заданного стоп-слова).

1.13. Структура: информация о сотрудниках

Поля:

- ФИО
- должность
- адрес проживания
- трудовой стаж (в месяцах)
- дата рождения

Задача: упорядочить введённый массив по должности и дате рождения.

1.14. Структура: информация о товарах

Поля:

- название
- производитель
- цена
- поступление на склад (количество месяцев, прошедших с момента поступления)
- скидка (проценты, по умолчанию — нуль)

Задача: задать количество месяцев и размер скидки. Всем товарам, которые поступили ранее, чем заданное количество месяцев, присвоить заданную величину скидки.

1.15. Структура: информация о статистике футбольной команды

Поля:

- название команды
- год выступления
- количество забитых мячей
- количество пропущенных мячей

Задача: упорядочить введенную информацию по годам и эффективности атаки. Годы выступлений идут по возрастанию, внутри каждого года упорядочение идёт по убыванию разности забитых и пропущенных мячей.

1.16. Структура: информация о футбольном игроке

Поля:

- ФИО
- название клуба
- возраст
- амплуа (нападающий, полузащитник, защитник, вратарь)
- количество игр за клуб

Задача: задать амплуа, пороговое значение возраста и количество игр. Выбрать всех игроков, амплуа которых совпадает с заданным, возраст не превышает порогового значения, а количество игр за клуб больше заданного.

1.17. Структура: информация о товарах

Поля:

- название
- производитель
- цена
- количество на складе

Задача: упорядочить введенную информацию по производителю, названию и количеству товара. Упорядочение по производителю и названию идёт в лексикографическом порядке (стандартные функции сравнения строк), по количеству — в порядке убывания.

1.18. Структура: информация об автомобиле

Поля:

- производитель
- название модели
- тип (легковой / грузовой)
- стоимость

Задача: задать тип автомобилей и вычислить среднюю стоимость всех авто заданного типа.

1.19. Структура: информация о приложениях

Поля:

- название приложения
- название разработчика
- рейтинг (аналог оценки в Apple/Google/Windows Store)
- количество установок

Задача: ввести название разработчика и найти среди введенной информации его приложения с максимальным и минимальным рейтингами.

1.20. Структура: информация о приложениях

Поля:

- название приложения
- название разработчика
- рейтинг (аналог оценки в Apple/Google/Windows Store)
- количество установок

Задача: упорядочить введенную информацию по разработчикам (лексикографический порядок), рейтингу (возрастание, от наименее оценённых к наиболее) и количеству установок (убывание, от наиболее загружаемых к наименее).

1.21. Структура: информация о ноутбуке

Поля:

- название модели
- производитель процессора
- производитель видеокарты
- количество оперативной памяти
- количество постоянной памяти (объём жестких дисков)
- цена
- предустановлена ли ОС (критерий да/нет)

Задача: задать диапазоны оперативной и постоянной памяти, а также нужна ли предустановленная операционная система. На основе введенных данных показать все ноутбуки, удовлетворяющие критериям.