

## ЛАБОРАТОРНАЯ РАБОТА 7

### КАКИЕ ТЕМЫ ЗАТРОНУТЫ

- структуры;
- работа с файлами.

**ФОРМУЛИРОВКА ЗАДАЧИ** С помощью функций потокового ввода/вывода переработать программу из **шестой лабораторной** на использование файлов для хранения. Добавить возможность вывести отдельный объект структуры (из всех сохранённых) по его номеру.

### ПОЯСНЕНИЯ

В шестой лабораторной весь массив структур хранился в оперативной памяти только во время работы программы. В рамках данной лабораторной перейдём к постоянному хранилищу 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:

```
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:

Добавлены две строки:

- **вторая строка** показывает количество уже записанных в файл объектов (в данном примере, объектов типа **Game**);
- в **девятой строке** добавлено новое действие. При его выборе у пользователя запрашивается номер объекта, который хотим вывести на консоль.

### 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 };

```

Для программной реализации подобной задачи в первую очередь нужно выбрать формат для хранения информации: *текстовый* или *двоичный*. У каждого варианта есть и свои достоинства, и подводные камни. В рамках данной лабораторной полностью сосредоточимся на **текстовом формате** хранения данных. При таком выборе, файл с данными можно будет просмотреть любым текстовым редактором.

Теперь нужно выбрать способ, как поля структурного 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 именнованная 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, "%*^[^\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

```

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:

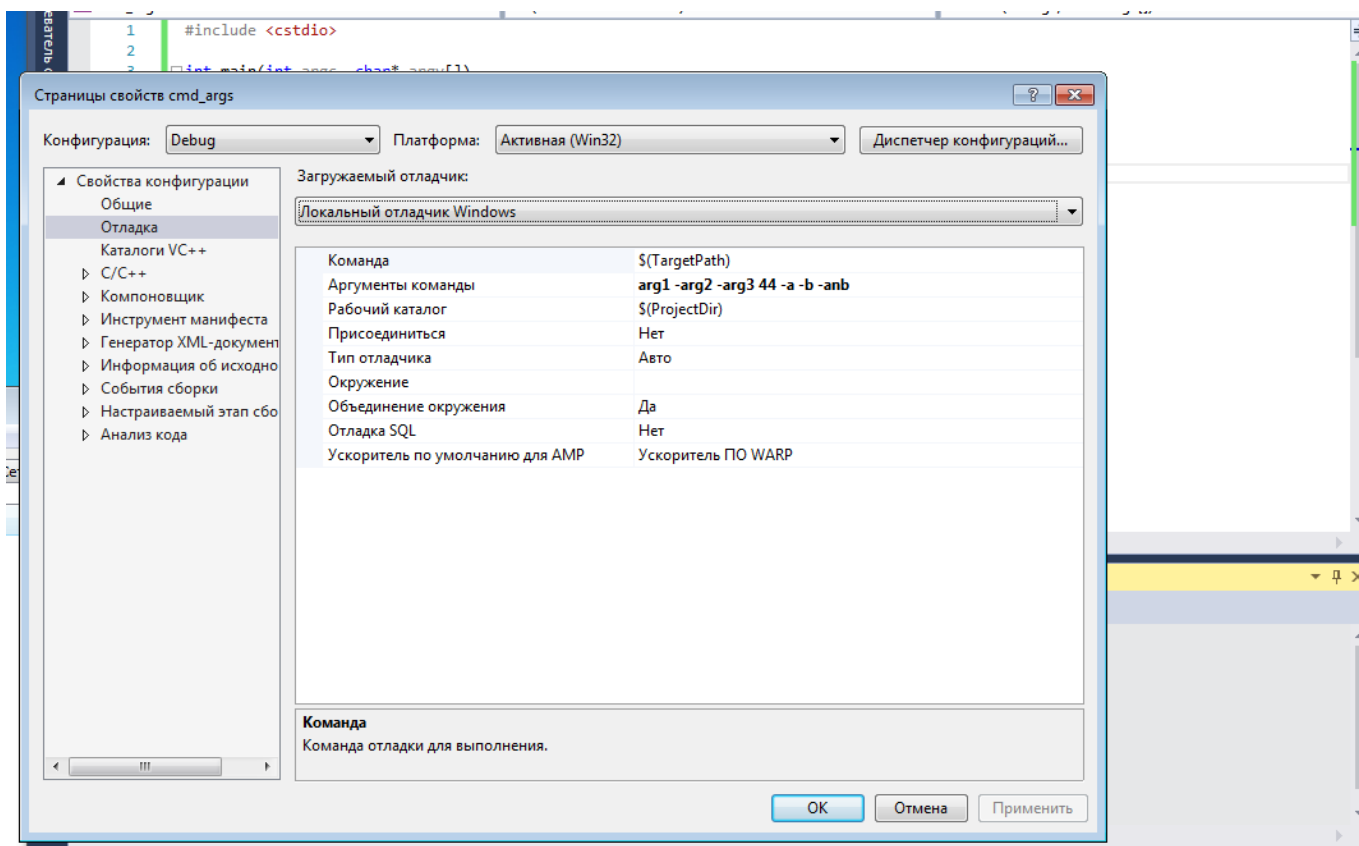
```
1 int main (int argc, char *argv[])
2 int main (int argc, char **argv)
```

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:

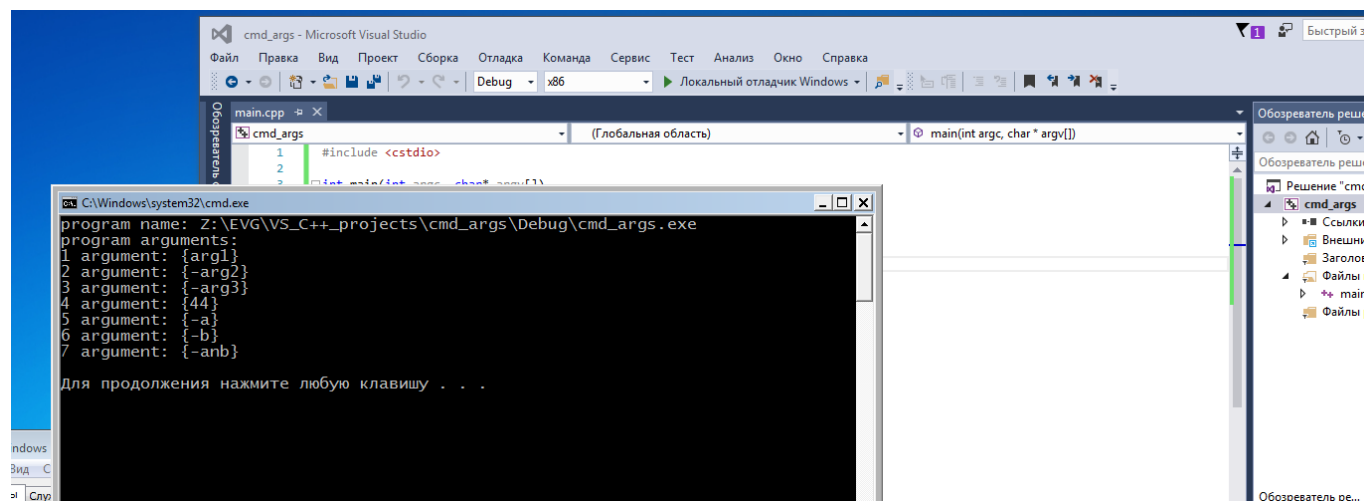
```
1 #include <stdio>
2
3 int main(int argc, char *argv[])
4 {
5     printf("program name: %s\n", argv[0]);
6
7     if (argc > 1) {
8         printf("program arguments:\n");
9         for (int i = 1; i < argc; i++) {
10             printf("%d argument: {%s}\n", i, argv[i]);
11         }
12         printf("\n");
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 интересует вкладка «Отладка» (для англ. версии IDE: «Project» => «... Properties» => «Debugging»). Вместо троеточия будет написано имя проекта. Открывшееся окно будет выглядеть примерно так:



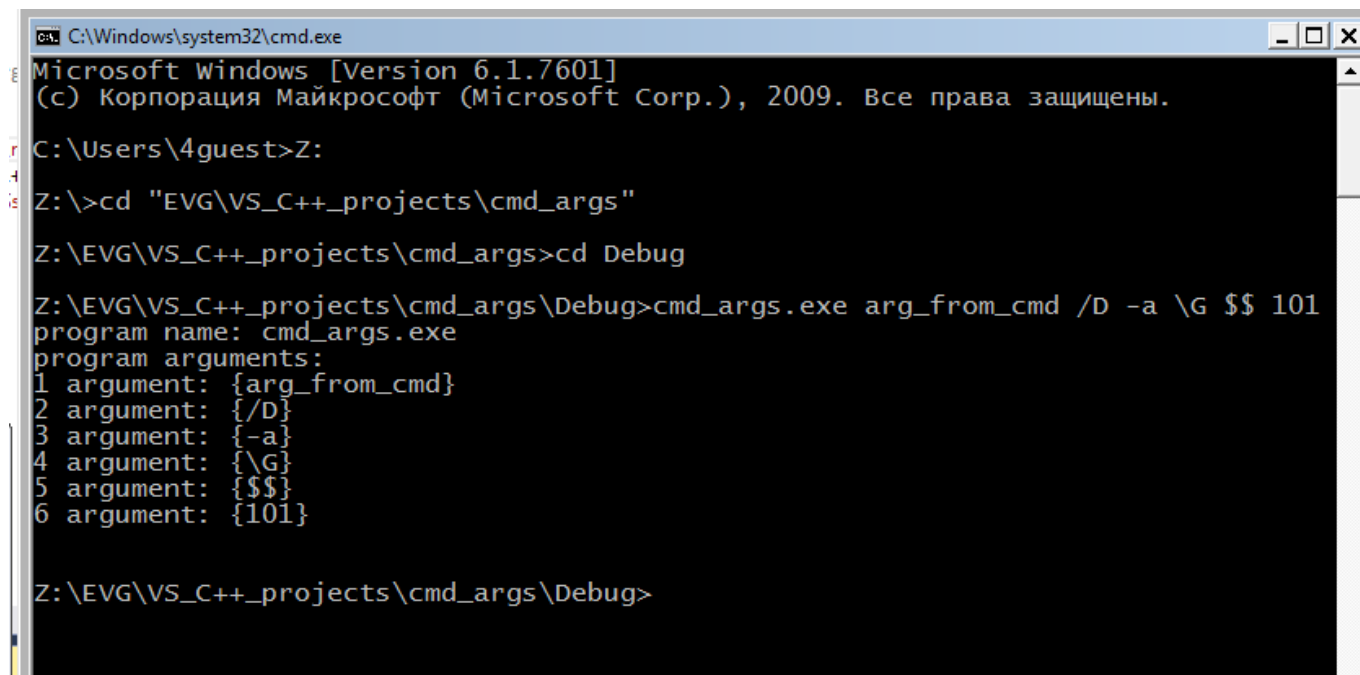
И аргументы задаются в поле «Аргументы команды» просто как текстовая строка. Каждый из них отделяется от другого пробелом. Тут стоит обратить внимание на левый верхний угол, где идёт выбор конфигурации. В средах разработки, как правило, аргументы командной строки задаются отдельно для каждой конфигурации: «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. Если в этом моменте нужны подсказки по организации алгоритма для выполнения нужной сортировки, желательно сообщить об этом преподавателям **до 25 мая**. Хоть на почту, хоть через github.

## ТРЕБОВАНИЯ К ПРОГРАММАМ

- Требования по оформлению/организации кода из предыдущих лабораторных.
- Минимальное использование динамических массивов. Если задача в предыдущей лабораторной состоит в выборе объектов по критериям, то выбранные структуры печатаются в консоли.
- И более строгая формулировка: запрещено считывать **все** структуры из файла в один динамический массив.
- 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. Хороший вариант – выводить в виде таблицы. Если элементов, которые должны быть выведены, нет, то программа должна выдавать соответствующее сообщение.