

**МIНIСТЕРСТВО  ОСВIТИ І НАУКИ  УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

“**КИЇВСЬКИЙ  ПОЛІТЕХНІЧНИЙ  ІНСТИТУТ**

**ІМЕНІ ІГОРЯ СІКОРСЬКОГО”**

**Факультет прикладної математики**

**Кафедра програмного забезпечення комп’ютерних систем**

**Лабораторна робота № 2**

**з дисципліни “ Основи програмування ”**

**тема “ Формат даних CSV ”**

|  |  |  |
| --- | --- | --- |
| **Виконала**  **студентка I курсу**  **групи КП-83**  **Снітко Маріанна Дмитрівна**  **(прізвище, ім’я, по батькові)**  **варіант № 22** |  | **Перевірив**  “**\_\_\_\_” “\_\_\_\_\_\_\_\_\_\_\_\_” 20\_\_\_ р.**  **викладач**  **Гадиняк Руслан Анатолійович**  **(прізвище, ім’я, по батькові)** |

**Київ 2018**

**Мета роботи**

Опанувати формат даних CSV та навчитись його зчитувати і формувати.

Реалізувати модуль універсального списку.

Навчитись збирати проекти з багатьох вихідних файлів за допомогою CMake.

**Постановка завдання**

**UPD-1**: Створити у Google Spreadsheet (або у будь-якому аналозі) таблицю, у яку внести мінімум 5 записів типу Винахідник зі стовпцями (мінімум 4), що містять дані екземплярів цього типу (приклад для типу Студент).

Експортувати\завантажити таблицю як CSV-файл і помістити її у директорію проекту як data.csv.

Описати структуру даних типу Винахідник. Створити у програмі список testList із мінімум 5 довільних екземплярів даного типу.

Створити консольну програму, що дозволяє працювати з CSV файлами.

Опції командного рядка (можуть йти у довільному порядку):

**IN** - (перший вільний аргумент) назва вхідного файлу з CSV

-n **N** - аргумент для додаткової обробки

-o **OUT** - назва вихідного файлу

**UPD-2**. Приклад використання функції getopt() для простого отримання значень опцій програми.

Всі опції не обов'язкові. За замовчуванням:

Якщо нема вхідного файлу - брати дані з testList

Якщо нема аргумента обробки - не обробляти список сутностей.

Якщо нема назви вихідного файлу - виводити дані у консоль (у довільному вигляді)

Якщо задано шлях до файлу з CSV даними, наприклад:

./a.out data.csv

Зчитати дані з файлу у список списків рядків inCsvTable (таблиця рядків, модель пам'яті).

Для цього потрібно реалізувати універсальний список (List, інтерфейс)

Вивести отриману таблицю у консоль (форматування при виводі довільне).

**UPD-4**. На основі таблиці inCsvTable сформувати список items екземплярів типу Винахідник.

Приклад для типу Студент:

**void fillStudentsListFromTable(List \* csvTable, List \* students);**

Якщо задано значення опції -n, наприклад:

./a.out data.csv -n 3

Після формування items Знайти всіх винахідників, у яких більше N патентів. Видалити зі списку лишні елементи. Для цього можна створити функцію:

**void processItems(List \* items, T n);** // T depends on your variant

Вивести модифікований items у консоль.

Якщо задано значення опції -o, наприклад:

./a.out -o out.csv

Перетворити модифікований items у нову таблицю outCsvTable (список списків рядків) і зберегти таблицю у вихідний файл у форматі CSV.

Перевірити коректність формату згенерованого файлу імпортувавши його у Google Spreadsheet (або аналог).

Модулі:

main - головний модуль

list - універсальний список (void \*, інтерфейс (UPD-3))

csv - функції для роботи із форматом даних CSV (інтерфейс модуля).

Створити у корені проекту директорію build.

Додати у корінь проекту файл CMakeLists.txt із директивами для збірки проекту за допомогою CMake.

Використовувати для збірки проекта такі команди:

cd build

cmake ..

make

**Тексти коду програм**

|  |
| --- |
| **csv.c** |
| #include <stdlib.h>  #include <string.h>  #include <stdio.h>  #include "list.h"  void Csv\_addInt(List \*row, int value)  {  char buffer[100];  sprintf(buffer, "%i", value);  char \*bufOnHeap = malloc(strlen(buffer) + 1);  strcpy(bufOnHeap, buffer);  List\_add(row, bufOnHeap);  }  void Csv\_addDouble(List \*row, double value)  {  char buffer[100];  sprintf(buffer, "%lf", value);  char \*bufOnHeap = malloc(strlen(buffer) + 1);  strcpy(bufOnHeap, buffer);  List\_add(row, bufOnHeap);  }  void Csv\_addString(List \*row, const char \*value)  {  char \*bufOnHeap = malloc(strlen(value) + 1);  strcpy(bufOnHeap, value);  List\_add(row, bufOnHeap);  }  int Csv\_int(List \*row, int index)  {  if (index >= 0 && index < row->size)  {  int buff = atoi(row->items[index]);  return buff;  }  else  {  printf("ERROR: invalid index\n");  return -1;  }  }  double Csv\_double(List \*row, int index)  {  if (index >= 0 && index < row->size)  {  float buffer = atof(row->items[index]);  return buffer;  }  else  {  printf("ERROR: invalid index\n");  return -1;  }  }  int Csv\_string(List \*row, int index, char \*buf, int nBuf)  {  if (strlen(List\_get(row, index)) + 1 > nBuf)  {  return -1;  }  else  {  strcpy(buf, List\_get(row, index));  return 0;  }  }  void Csv\_addRow(List \*table, List \*row)  {  List\_add(table, row);  }  List \*Csv\_row(List \*table, int index)  {  return table->items[index];  }  void Csv\_fillTableFromString(List \*csvTable, const char \*csvStr)  {  List \*row = NULL;  char buf[10000];  int bufX = 0;  const char \*p = csvStr;  while (1)  {  if (\*p == ',' || \*p == '\n' || \*p == '\0')  {  buf[bufX] = '\0';  if (bufX == 0 && \*p == '\0' && row == NULL)  {  break;  }  if (row == NULL)  {  row = List\_alloc();  }  char \*bufOnHeap = malloc(strlen(buf) + 1);  strcpy(bufOnHeap, buf);  List\_add(row, bufOnHeap);  if (\*p == '\n' || \*p == '\0')  {  List\_add(csvTable, row);  row = NULL;  }  bufX = 0;  }  else  {  buf[bufX] = \*p;  bufX += 1;  }  if (\*p == '\0')  {  break;  }  p += 1;  }  }  int Csv\_fillStringFromTable(List \*csvTable, char \*buf, int nBuf)  {  buf[0] = '\0';  int buffLength = 0;  for (size\_t i = 0; i < List\_size(csvTable); i++)  {  List \*rowItem = List\_get(csvTable, i);  for (size\_t j = 0; j < List\_size(rowItem); j++)  {  char \*value = List\_get(rowItem, j);  strcat(buf, value);  if (j != List\_size(rowItem) - 1)  {  strcat(buf, ",");  }  buffLength += strlen((char\*)List\_get(rowItem,j));  }  if (i != List\_size(csvTable) - 1)  {  strcat(buf, "\n");  }  }  if (buffLength > nBuf)  return -1;  return 0;  }  char \*Csv\_createStringFromTable(List \*csvTable)  {  char \*newString = malloc(sizeof(char) \* 10000);  newString[0] = '\0';  Csv\_fillStringFromTable(csvTable, newString, 10000);  return newString;  }  void Csv\_clearTable(List \*csvTable)  {  for (size\_t i = 0; i < List\_size(csvTable); i++)  {  List \*temp = List\_get(csvTable, i);  for (size\_t j = 0; j < List\_size(temp); j++)  {  free(List\_get(temp, j));  }  List\_free(List\_get(csvTable, i));  }  } |

|  |
| --- |
| **csv.h** |
| #include "list.h"  void Csv\_addInt (List \*row, int value); // malloc a string and write value in it  void Csv\_addDouble (List \*row, double value); // malloc a str and write value in it  void Csv\_addString (List \*row, char \*value); // malloc a string and copy value  int Csv\_int (List \*row, int index); // get item (str) and convert it to int  double Csv\_double (List \*row, int index); // get str and convert it to double  int Csv\_string (List \*row, int index, char \*buf, int nBuf); // get str & copy it to buf  void Csv\_addRow (List \*table, List \*row); // add row to table as an item  List \*Csv\_row (List \*table, int index); // get item from table as List \*  void Csv\_fillTableFromString (List \*csvTable, const char \*csvStr); // parse csv & fill table  int Csv\_fillStringFromTable (List \*csvTable, char \*buf, int nBuf); // fill buf with csv  char \*Csv\_createStringFromTable (List \*csvTable); // malloc string array and fill it with csv  void Csv\_clearTable (List \*csvTable); // free all table allocated memory |

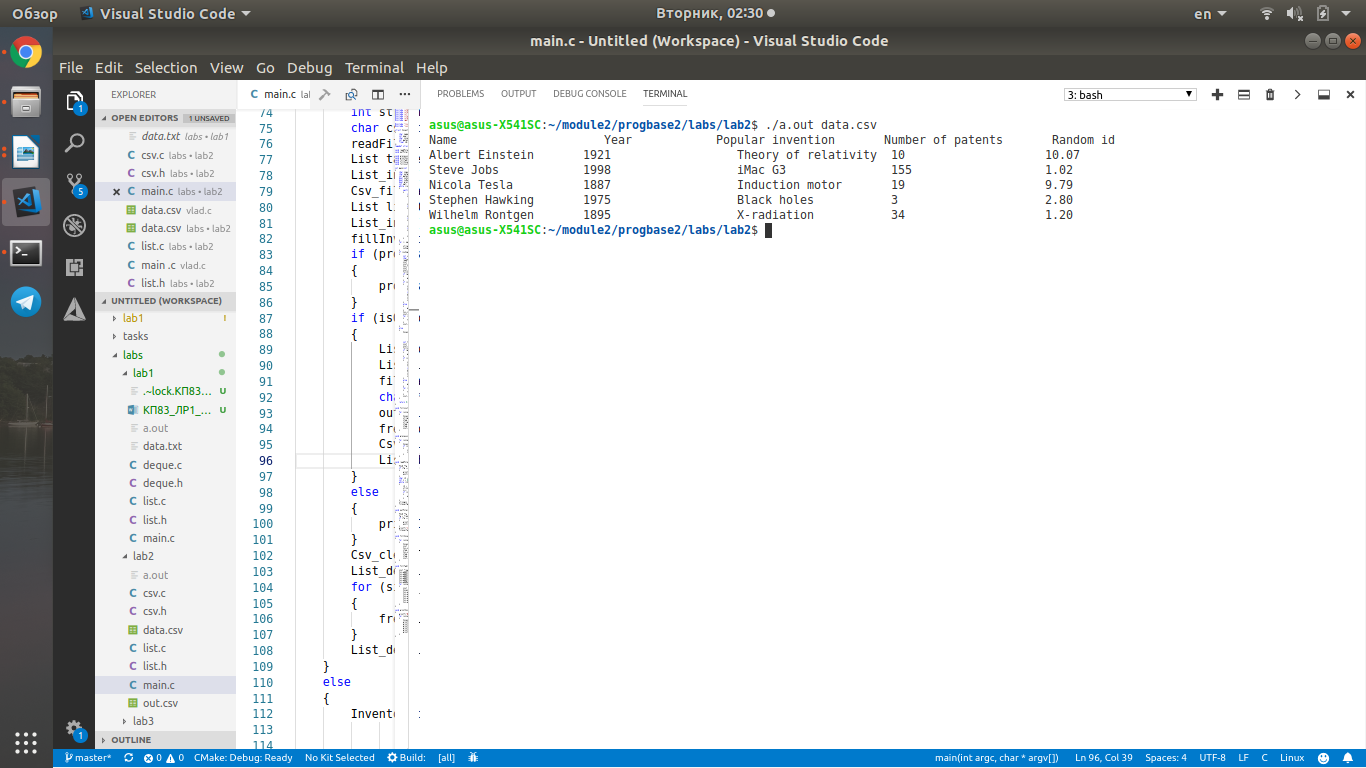
|  |
| --- |
| **list.c** |
| #include <stdlib.h>  #include <stdbool.h>  #include <stdio.h>  #include "list.h"  List \*List\_alloc(void)  {  List \*list = malloc(sizeof(List));  List\_init(list);  return list;  }  void List\_free(List \*self)  {  List\_deinit(self);  free(self);  }  void List\_init(List \*self)  {  self->size = 0;  self->capacity = 4;  self->items = malloc(sizeof(void \*) \* self->capacity);  if (self->items == NULL)  {  fprintf(stderr, "Allocating memory error\n");  abort();  }  }  void List\_deinit(List \*self)  {  free(self->items);  }  size\_t List\_size(List \*self)  {  return self->size;  }  void \*List\_get(List \*self, int index)  {  return self->items[index];  }  void List\_set(List \*self, int index, void \*value)  {  self->items[index] = value;  }  void List\_insert(List \*self, int index, void \*value)  {  if (index > self->size || index < 0)  {  fprintf(stderr, "ERROR: item cannot be inserted\n");  abort();  }  else if (index == self->size)  {  List\_add(self, value);  }  else if (index < self->size)  {  if (self->size + 1 < self->capacity)  {  int newcap = self->capacity + 1;  void \*newitems = realloc(self->items, sizeof(void \*) \* newcap);  if (newitems == NULL)  {  free(self->items);  fprintf(stderr, "Reallocating memory error\n");  abort();  }  self->items = newitems;  self->capacity = newcap;  }  for (size\_t i = self->size; i >= index; i--)  {  self->items[i + 1] = self->items[i - 1];  }  self->items[index] = value;  self->size++;  }  }  void List\_removeAt(List \*self, int index)  {  if (index >= self->size || index < 0)  {  printf("ERROR: Index is bigger than list's size\n");  return;  }  free(self->items[index]);  for (size\_t i = index; i < self->size - 1; i++)  {  self->items[i] = self->items[i + 1];  }  self->size--;  }  void List\_add(List \*self, void \*value)  {  self->items[self->size] = value;  self->size += 1;  if (self->size == self->capacity)  {  int newcap = self->capacity \* 2;  void \*newitems = realloc(self->items, sizeof(void \*) \* newcap);  if (newitems == NULL)  {  free(self->items);  fprintf(stderr, "Reallocating memory error\n");  abort();  }  self->items = newitems;  self->capacity = newcap;  }  }  void List\_remove(List \*self, void \*value)  {  for (size\_t i = 0; i < self->size; i++)  {  if (self->items[i] == value)  {  free(self->items[i]);  for (size\_t j = i; j < self->size; j++)  {  if (j != self->size - 1)  {  self->items[j] = self->items[j + 1];  }  }  self->size--;  return;  }  }  printf("ERROR: value does not exist\n");  }  int List\_indexOf(List \*self, void \*value)  {  for (size\_t i = 0; i < self->size; i++)  {  if (self->items[i] == value)  {  return i;  }  }  printf("ERROR: value does not exist\n");  return -1;  }  bool List\_contains(List \*self, void \*value)  {  for (size\_t i = 0; i < self->size; i++)  {  if (self->items[i] == value)  {  return 1;  }  }  return 0;  }  bool List\_isEmpty(List \*self)  {  if (self->size == 0)  return 1;  else  return 0;  }  void List\_clear(List \*self)  {  self->size = 0;  } |

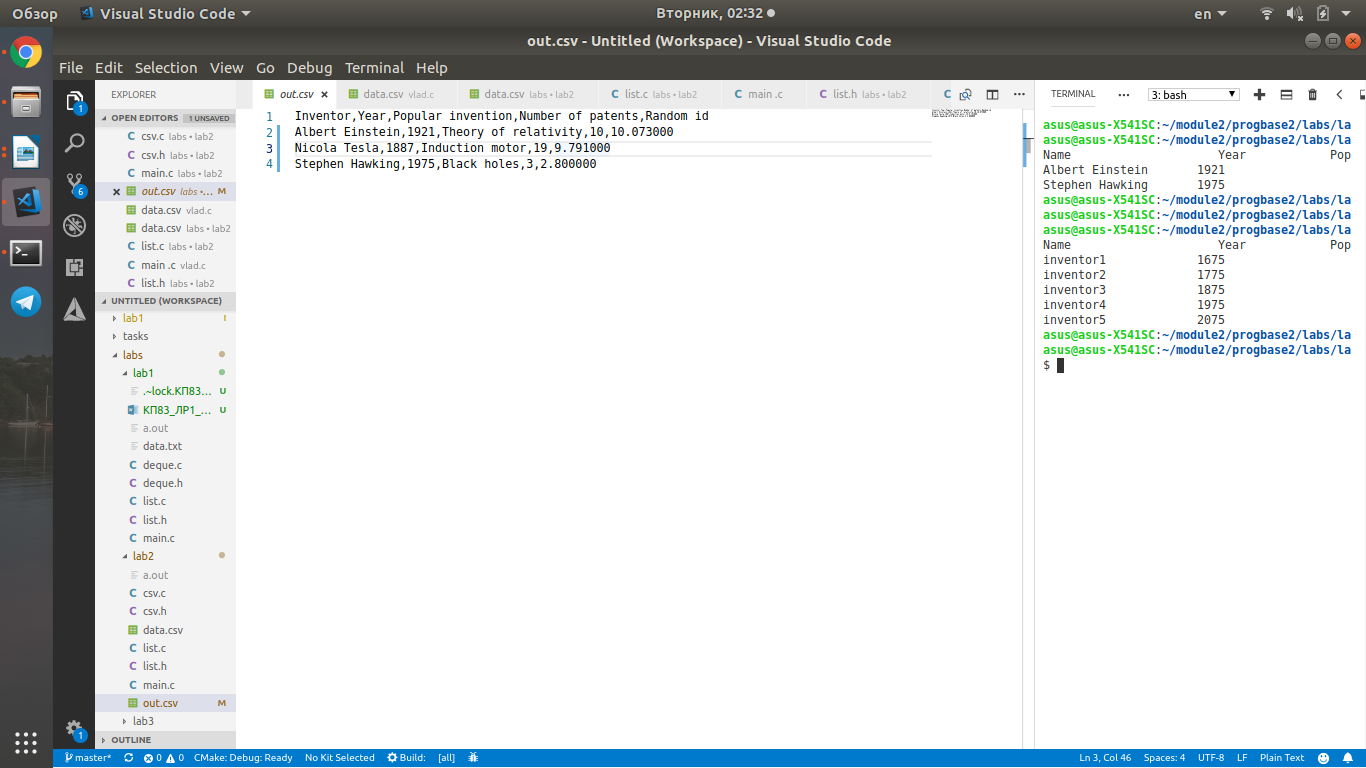
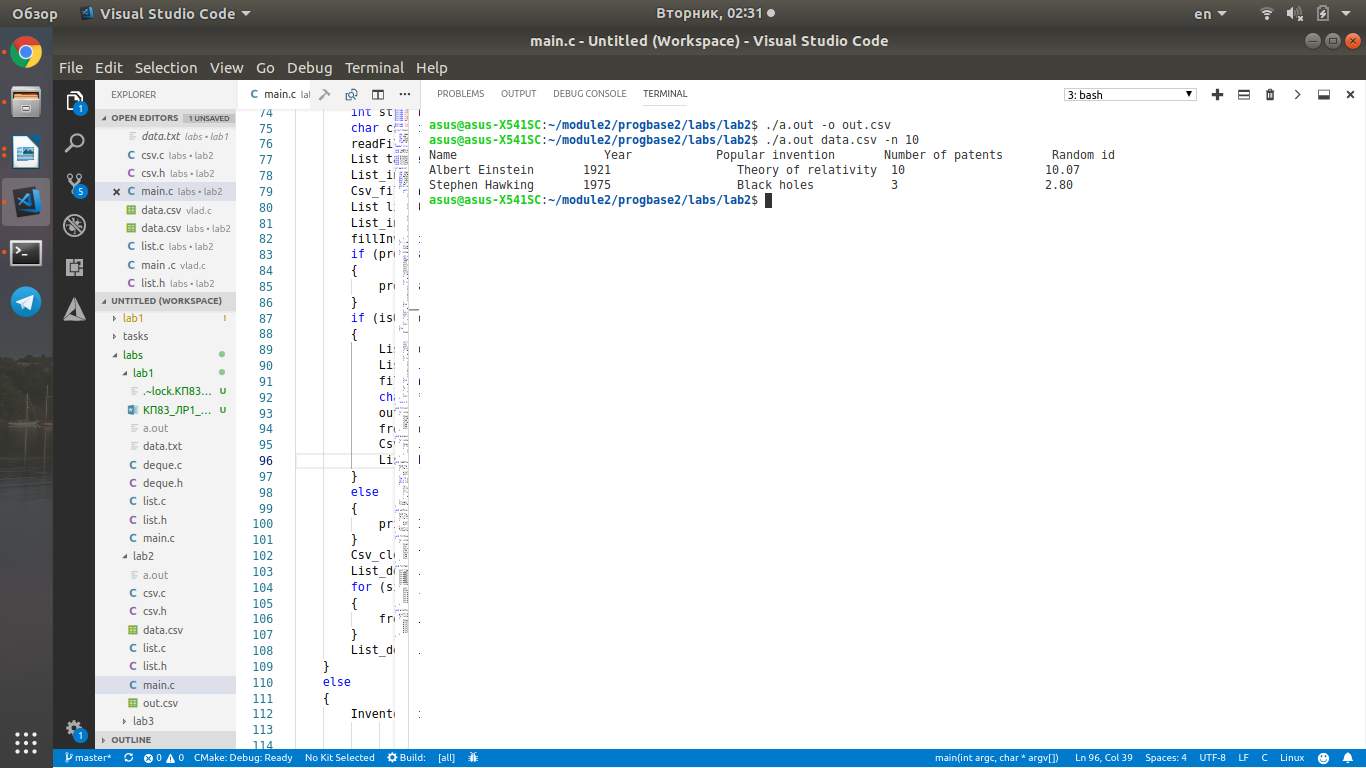
|  |
| --- |
| **list.h** |
| #pragma once  #include <stdlib.h>  #include <stdbool.h>  struct \_\_List  {  void \*\*items;  size\_t size;  size\_t capacity;  };  typedef struct \_\_List List;  List \*List\_alloc (void);  void List\_free (List \*self);  void List\_init (List \*self);  void List\_deinit (List \*self);  size\_t List\_size (List \*self); // return number of items in list  void \*List\_get (List \*self, int index); // return self->items[index]  void List\_set (List \*self, int index, void \*value); // set items[index]  void List\_insert (List \*self, int index, void \*value); // insert, shift right  void List\_removeAt (List \*self, int index); // remove and shift left  void List\_add (List \*self, void \*value); // insert back  void List\_remove (List \*self, void \*value); // remove first by value  int List\_indexOf (List \*self, void \*value); // find index by value  bool List\_contains (List \*self, void \*value); // check by value  bool List\_isEmpty (List \*self); // check if list has any items  void List\_clear (List \*self); // make list empty |

|  |
| --- |
| **main.c** |
| #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <ctype.h>  #include <stdbool.h>  #include <getopt.h>  #include "list.h"  #include "csv.h"  struct Inventor  {  char name[100];  int year;  char invention[100];  int patents;  float id;  };  typedef struct Inventor Inventor;  void readFile(char \*inputFileName, char string[], int len);  void outFile(char \*outFileName, char \*string);  void processListOfStructs(List \*inventors, int patents);  void printInventorsList(List \*inventors);  void fillTableFromListOfStructs(List \*table, List \*inventors);  void fillInventorsListFromTable(List \*table, List \*inventors);  void fillListFromArray(List \*inventors, Inventor array[], int length);  int main(int argc, char \*argv[])  {  char input[20] = "name";  char output[20] = "name";  int patents = 0;  bool isInput = false;  bool isOutput = false;  bool processPatents = false;  int opt;  while ((opt = getopt(argc, argv, "n:o:")) != -1)  {  switch (opt)  {  case 'n':  {  patents = atoi(optarg);  processPatents = true;  break;  }  case 'o':  {  strcpy(output, optarg);  isOutput = true;  break;  }  case ':':  {  printf("option needs a value\n");  break;  }  case '?':  {  printf("unknown option: %c\n", optopt);  break;  }  }  }  for (; optind < argc; optind++)  {  strcpy(input, argv[optind]);  isInput = true;  }  if (isInput == true)  {  int strLength = 1000;  char csvStr[strLength];  readFile(input, csvStr, strLength);  List table;  List\_init(&table);  Csv\_fillTableFromString(&table, csvStr);  List listOfInventors;  List\_init(&listOfInventors);  fillInventorsListFromTable(&table, &listOfInventors);  if (processPatents == true)  {  processListOfStructs(&listOfInventors, patents);  }  if (isOutput == true)  {  List parsedTable;  List\_init(&parsedTable);  fillTableFromListOfStructs(&parsedTable, &listOfInventors);  char \*p = Csv\_createStringFromTable(&parsedTable);  outFile(output, p);  free(p);  Csv\_clearTable(&parsedTable);  List\_deinit(&parsedTable);  }  else  {  printInventorsList(&listOfInventors);  }  Csv\_clearTable(&table);  List\_deinit(&table);  for (size\_t i = 0; i < List\_size(&listOfInventors); i++)  {  free(List\_get(&listOfInventors, i));  }  List\_deinit(&listOfInventors);  }  else  {  Inventor testArr[] = {{"inventor1", 1675, "name 1", 100, 3.1},  {"inventor2", 1775, "name 2", 10, 33.2},  {"inventor3", 1875, "name 3", 20, 55.3},  {"inventor4", 1975, "name 4", 340, 4.4},  {"inventor5", 2075, "name 5", 15, 2.5}};  int testArrLen = 5;  List listOfInventors;  List\_init(&listOfInventors);  fillListFromArray(&listOfInventors, testArr, testArrLen);  if (processPatents == true)  {  processListOfStructs(&listOfInventors, patents);  }  if (isOutput == true)  {  List parsedTable;  List\_init(&parsedTable);  fillTableFromListOfStructs(&parsedTable, &listOfInventors);  char \*p = Csv\_createStringFromTable(&parsedTable);  outFile(output, p);  free(p);  Csv\_clearTable(&parsedTable);  List\_deinit(&parsedTable);  }  else  {  printInventorsList(&listOfInventors);  }  for (size\_t i = 0; i < List\_size(&listOfInventors); i++)  {  free(List\_get(&listOfInventors, i));  }  List\_deinit(&listOfInventors);  }  return 0;  }  void fillListFromArray(List \*inventors, Inventor array[], int length)  {  for (size\_t i = 0; i < length; i++)  {  Inventor \*item = malloc(sizeof(Inventor));  strcpy(item->name, array[i].name);  item->year = array[i].year;  strcpy(item->invention, array[i].invention);  item->patents = array[i].patents;  item->id = array[i].id;  List\_add(inventors, item);  }  }  void processListOfStructs(List \*inventors, int patents)  {  for (size\_t i = 0; i < List\_size(inventors); i++)  {  Inventor \*item = List\_get(inventors, i);  if (item->patents > patents)  {  List\_removeAt(inventors, i);  i -= 1;  }  }  }  void readFile(char \*inputFileName, char string[], int len)  {  FILE \*fp = fopen(inputFileName, "r");  if (fp == NULL)  {  printf("ERROR: file with such name does not exist.\n");  exit(EXIT\_FAILURE);  }  char ch = 0;  int i = 0;  while (ch != EOF)  {  ch = fgetc(fp);  string[i] = ch;  i++;  if (i >= len)  {  abort();  }  }  string[i - 1] = '\0';  fclose(fp);  }  void outFile(char \*outFileName, char \*string)  {  FILE \*fw = NULL;  fw = freopen(outFileName, "w", stdout);  if (fw == NULL)  {  printf("Error opening file %s\n", outFileName);  exit(EXIT\_FAILURE);  }  printf("%s", string);  fclose(fw);  }  void fillInventorsListFromTable(List \*table, List \*inventors)  {  for (size\_t i = 1; i < List\_size(table); i++)  {  Inventor \*item = malloc(sizeof(Inventor));  List \*row = List\_get(table, i);  Csv\_string(row, 0, item->name, 100);  item->year = Csv\_int(row, 1);  Csv\_string(row, 2, item->invention, 100);  item->patents = Csv\_int(row, 3);  item->id = Csv\_double(row, 4);  List\_add(inventors, item);  }  }  void fillTableFromListOfStructs(List \*table, List \*inventors)  {  List \*row = List\_alloc();  Csv\_addString(row, "Inventor");  Csv\_addString(row, "Year");  Csv\_addString(row, "Popular invention");  Csv\_addString(row, "Number of patents");  Csv\_addString(row, "Random id");  Csv\_addRow(table, row);  for (size\_t i = 0; i < List\_size(inventors); i++)  {  List \*row = List\_alloc();  Inventor \*item = List\_get(inventors, i);  Csv\_addString(row, item->name);  Csv\_addInt(row, item->year);  Csv\_addString(row, item->invention);  Csv\_addInt(row, item->patents);  Csv\_addDouble(row, item->id);  Csv\_addRow(table, row);  }  }  void printInventorsList(List \*inventors)  {  printf("Name\t\t\t Year\t\t Popular invention\t Number of patents\t Random id\n");  for (size\_t i = 0; i < List\_size(inventors); i++)  {  Inventor \*item = List\_get(inventors, i);  printf("%s\t\t ", item->name);  printf("%i\t\t ", item->year);  printf("%s\t\t ", item->invention);  printf("%i\t\t ", item->patents);  printf("%.2f\t\t ", item->id);  printf("\n");  }  } |

|  |
| --- |
| CmakeLists.txt |
| cmake\_minimum\_required(VERSION 2.8.9)  project(lab2)  file(GLOB SOURCES "\*.c")  add\_executable(${PROJECT\_NAME} ${SOURCES})  target\_link\_libraries(${PROJECT\_NAME} progbase) |

**Приклади результатів**

****

****

**Висновок**

Протягом виконання цієї лабораторної роботи я ознайомилася із програмуванням на мові С у спеціалізованому редакторі коду.

Більше того, під час виконання лабораторної роботи я опанувала формат даних СSV та навчилася його зчитувати і форматувати. Також мною було засвоєно реалізацію модуля універсального списку різними способами. Окрім цього, я навчилася збирати проекти з багатьох вихідних файлів за допомогою CMake.

Компіляція всього коду відбувалася за допомогою компілятора gcc.

Отже, мета лабораторної роботи досягнена, всі завдання виконані і їх розвʼязання наведені.