COSC 292 (Advanced Programming 2)  
Assignment 3

**Assignment due:** Monday, April 15, 2024, by 10:00 pm

**Late submissions:** The marks for late assignments will be as follows:

* Less than 24 hours late: original mark \* 0.75
* 24 hours or more late: a mark of 0 will be given and no feedback will be provided

**Assignment specifications:** This assignment must be completed using C. Submissions should follow good programming practices, including proper documentation.

This assignment should be completed in groups of 2 (or individually **only** if you cannot find a partner). Both students will receive the same mark. Email [michael.grzesina@saskpolytech.ca](mailto:michael.grzesina@saskpolytech.ca) with your partner by Friday, April 5, 2024, at 10:00 pm. Partners may be assigned if not enough groups are chosen by the students.

Any plagiarism or other academic misconduct, including submitting code generated by AI systems, will result in a mark of 0 for all offending parties and possible further consequences. Remember: share concepts, not code!

Create a project named ***username1username2*cosc292a3**, where ***username1*** and ***username2*** are your SaskPolytech usernames.

# Assignment specifications

The purpose of this assignment is to practice several concepts learned in class such as:

* Creating and populating structs
* Working with arrays of structs
* Serialization of a struct
* Saving date to a file and retrieving data from a file

Your client, Ada’s Used Vehicles, would like you to create an application that will store information about vehicles. A Vehicle struct will store vehicle information, and a Garage struct will store an array of vehicles. Structs are define in a header file called **structs.h** as follows (add proper documentation as needed):

#ifndef STRUCTS\_H

#define STRUCTS\_H

#include <stdio.h>

typedef unsigned char BYTE;

#define VIN\_SIZE 18

#define MAKE\_SIZE 25

#define MODEL\_SIZE 25

typedef struct

{

// All char arrays are null-terminated strings.

BYTE VIN[VIN\_SIZE];

BYTE Make[MAKE\_SIZE];

BYTE Model[MODEL\_SIZE];

BYTE\* Description;

}Vehicle;

typedef struct

{

Vehicle\*\* Vehicles;

unsigned short NumVehicles;

}Garage;

Vehicle\* createVehicle(BYTE bVin[], BYTE bMake[], BYTE bModel[], BYTE\* bDescription);

void addVehicleToGarage(Garage\* g, Vehicle\* vPtr);

void displayVehicle(Vehicle v);

void displayGarage(Garage g);

BYTE\* serializeVehicle(Vehicle\* vPtr);

void writeGarageToFile(Garage g, FILE\* filePtr);

void readGarageFromFile(Garage\* g, FILE\* filePtr);

#endif // !STRUCTS\_H

## Part A – Creating and working with Vehicles

createVehicle – dynamically allocate a Vehicle struct. Populate with attributes passed in. NOTE – the description is a dynamically allocated null-terminated string of the exact size required. String functions may be used to code this function.

addVehicleToGarage – Note that the garage stores Vehicle pointers, not the structs themselves. Given a garage and a vehicle pointer, add the pointer to the garage Vehicles array. A deep copy of the vehicles is not required – just add the pointer passed in. The Vehicles array is a dynamically allocated array of Vehicle pointers. It must be expanded to add a new pointer. NumVehicles must also be incremented.

displayVehicle – Display the attributes of a vehicle to the console window.

displayGarage – For each vehicle in the garage, display its contents.

Test code (in **program.c**) – add appropriate documentation and memory handling as needed:

#define MAX\_DESCRIPTION 255

void enterVehicles(Garage\* g)

{

BYTE bVin[VIN\_SIZE];

BYTE bMake[MAKE\_SIZE];

BYTE bModel[MODEL\_SIZE];

BYTE bDescription[MAX\_DESCRIPTION];

for (int i = 0; i < 3; i++)

{

printf("Enter a VIN: ");

gets\_s(bVin, VIN\_SIZE);

printf("Enter a make: ");

gets\_s(bMake, MAKE\_SIZE);

printf("Enter a model: ");

gets\_s(bModel, MODEL\_SIZE);

printf("Enter a description: ");

gets\_s(bDescription, MAX\_DESCRIPTION);

Vehicle\* temp = createVehicle(bVin, bMake, bModel, bDescription);

addVehicleToGarage(g, temp);

}

}

int main(int argc, char\*\* argv)

{

Garage g = { NULL, 0 };

enterVehicles(&g);

displayGarage(g);

return EXIT\_SUCCESS;

}

## Part B – File I/O

Create a separate header file called **fileio.h**. Note that these functions are similar but not necessarily identical to the last assignment. Add appropriate documentation as needed.

#ifndef FILEIO\_H

#define FILEIO\_H

#include <stdio.h>

typedef unsigned char BYTE;

FILE\* openFile(const char\* fileName, const char\* fileMode);

int readFile(FILE\* filePtr, BYTE\* data, int bytesToRead);

int writeFile(FILE\* filePtr, BYTE\* data, int bytesToWrite);

#endif // !FILEIO\_H

openFile – same as previous assignment

readFile – This function takes a buffer that is already allocated (**data**). The function will read into this buffer. It also takes a parameter indicating how many bytes to read from **filePtr**. It does not allocate any memory. The function still returns an error code.

writeFile – Given a buffer, write its **data** to the **filePtr**. **bytesToWrite** are the number of bytes to write out.

## Part C – Serialization of a struct

What is serialization and why is it needed? To serialize a struct means to convert its current state to a byte stream in such a way that the byte stream can be reverted back into a copy of the struct. Serialization is required for writing some structs (and objects) to file or to pass the data within a struct across a network.

Consider an instance of the Vehicle struct:

typedef struct

{

// All char arrays are null-terminated strings.

BYTE VIN[VIN\_SIZE];

BYTE Make[MAKE\_SIZE];

BYTE Model[MODEL\_SIZE];

BYTE\* Description;

}Vehicle;

The **VIN**, **Make**, and **Model** are all stored within the struct instance. However, the **Description** is simply a pointer (4 or 8 bytes, depending whether you are working on an x86 or x64 platform). The actual **Description** is stored at a different location in memory.

The Vehicle itself may be on a stack frame or on the heap depending on how it was created. The description is on the heap.

A diagram of a memory

Description automatically generated

A naïve approach to write the vehicle to file would be to pass its address to the **writeFile** function and simply write 72 bytes (size of **myVehicle** assuming 4-byte pointers). Although this would store any primitives or static arrays (**VIN**, **Make**, and **Model**) just fine, the description would be written out as a 4-byte pointer. The actual string would not be written out to file. The address itself would be meaningless when the file is loaded again.

What is the solution? Serialize your struct. Convert it into a single stream of bytes. For example, a BYTE array could be allocated that is the size of VIN\_SIZE + MAKE\_SIZE + MODEL\_SIZE + strlen(Description string) + 1. The data would be copied from myVehicle to the byte stream in order:

A black rectangle with white text

Description automatically generated

Now the entire byte stream can be written to file. One problem, though – how long is the description string? When we read the data from the file, we won’t know its length. The solution is to write the size of the description field into the BYTE array before writing to file. The size could be stored in an **unsigned short**:

A rectangular black and white rectangle with black text

Description automatically generated

Now when we read from file, we can read the **VIN**, **Make**, and **Model** (defined sizes) first. Then we read the two-byte **unsigned short**. Next, we can read in the description (now that we know its exact size). Following this would be the next vehicle in the file.

### Functions

serializeVehicle – Given a pointer to a vehicle, dynamically allocate a byte array to store the VIN, Make, Model, size, and description string. Copy the data from the struct to the BYTE array. Return the array. This will be a helper method called from **writeGarageToFile**.

writeGarageToFile – Firstly, the number of vehicles should be written into the first two bytes of the file. Then loop through the vehicles. Each vehicle should be serialized and then written to file one at a time.

readGarageFromFile – Read the number of vehicles in the file and assign to the garage. Loop through the number of vehicles. For each vehicle, read in the VIN, Make, and Model. Then read in the size. Next, read in the description.

NOTE: This does not address all issues regarding serialization. Some other issues are:

* Endianness – A struct serialized on a little-endian platform will not be directly readable on a big-endian platform.
* Data type sizes – Data type sizes are defined as minimums. A **short** may be two bytes on one system and four bytes on another.
* Struct byte alignment could be an issue if the struct is serialized as a single block of memory. Different platforms utilize different byte alignment schemes.

Test code – alter as needed:

Test writing to file:

void TestWriteToFile(Garage g)

{

FILE\* filePtr = openFile("vehicles.vhs", "wb");

writeGarageToFile(g, filePtr);

fclose(filePtr);

}

int main(int argc, char\*\* argv)

{

Garage g = { NULL, 0 };

enterVehicles(&g);

TestWriteToFile(g);

Test reading from file:

void TestReadFromFile(Garage\* g)

{

FILE\* filePtr = openFile("vehicles.vhs", "rb");

readGarageFromFile(g, filePtr);

fclose(filePtr);

}

int main(int argc, char\*\* argv)

{

Garage g = { NULL, 0 };

TestReadFromFile(&g);

displayGarage(g);

# Marks breakdown

* **structs.h** properly set up – 1 mark
* In **structs.c**:
  + **createVehicle** – 5 marks
  + **addVehicleToGarage** – 5 marks
  + **displayVehicle** – 2 marks
  + **displayGarage** – 2 marks
* **fileio.h** properly set up – 1 mark
* In **fileio.c**:
  + **openFile** – 1 mark
  + **readFile** – 3 marks
  + **writeFile** – 3 marks
* Serialization:
  + **serializeVehicle** – 7 marks
  + **writeGarageToFile** – 5 marks
  + **readGarageToFile** – 8 marks
* Other:
  + Adequate test code – 3 marks
  + Comments, etc. – 2 marks
  + Free memory when done – 2 marks
* Marks may be adjusted by possible deductions

**Total marks: 50**

# Submission instructions

* Ensure that your entire project is in your ***username1username2*cosc292a3** folder.
* Right-click on your ***username1username2*cosc292a3** folder and choose **Send to > Compressed (zipped) folder** to create a zip file
* Submit the zip file to the online course under **Assessments > Dropbox > Assignment 3**.