A Classy Introduction

PROGRAM ONE

A refresher on classes and pointers and an introduction to Git and groupwork that will get you through the rest of the semester.

John Grzegorczyk

John Grzegorczyk

Outline

This program will be a quite basic program with the primary goal of learning Git and how to work with your groups.

Description

A layered program focused on the learning of the industry-standard technology that is Git. Students will be working in groups of four on the program each building parts of it that will come together to make a whole. The goal is to not only have a working project that you built as a team, but to come away with an understanding of Git that you will need for the rest of the projects you will do this semester. Parts of the program will be dependent on other parts, meaning there will be a need for prioritization as well as shared code to aid in the development in multiple sections.

Requirements

The program will consist of four parts and (ideally) eight files. Those will be subdivided as follows:

Parts

Driver

- Contains a full program flow.
  - All classes are used.
  - All functions can be tested.
  - User has some kind of input.
- Has a natural way to end the program.

Storage class

- Pointer to an array of pointers to the data.
- Data stored is the data class.
  - Examples: parking lot, toy box, galaxy (holds stars).
  - Requirements: be able to read items from a file, be able to print the contents to the screen.

Data class

- Stores some kind of information.
- Data is private but modifiable through function calls.
- Contains at least 1 pointer.

Other class

- Is either top-level or bottom-level.
  - Can be used as a median between the driver and storage.
  - Can be used as a sub-data in the data class.
- Contains at least 1 pointer

Files

- Driver.cpp (or any name of the sorts)
- [storage class].cpp & [storage class].h(pp)
- [data class].cpp & [data class].h(pp)

- [other class].cpp & [other class].h(pp)
  - NOTE: The classes' names and files should match
  - NOTE: The processor can use .h and .hpp files the same
- Makefile
  - NOTE: The Makefile is for your and our convenience
- TEST_CASE.txt
  - One extensive test case is required!
  - Should test:
    - Every possible path of the program
    - Edge cases of classes
- Other files:
  - Save files
  - Other cpp and h/hpp helper files

It is also acceptable to just use the .h(pp) files without the .cpp files, HOWEVER Make will not be able to speed up the compile time if that is the case.

Timeline

Week one

- Program outline
  - Hand out this document
  - Go over the information
  - Can use slides if desirable
- Group decisions
  - Project ideation and selection
  - Program breakdown
  - Project name (not too important, but it's fun)
- Initial commit
  - Project README.md
    - Contains a description of the program design
    - Outline of what the program will do
    - Classes that are needed
  - Class definitions (if there is time)
- Write-up (done and submitted by the group's leader)
  - Needs to explain what the program idea is
  - What classes will be used
  - Functions that will be needed
  - Program flow
  - Individual role breakdown
  - This does not need to have everything
  - Just rough-draft first-thoughts
  - Must be submitted by the end of the night!

Week two

(workday/check-in)

- Progress check
  - What classes are done
  - What functions need work
  - What functions need to be added for the program to work
- First bug testing (if far enough along)
  - Tests individual classes (where possible)
  - Writes drivers in unique branches for testing
  - Note down and create issues for any bugs found
- Change any classes as needed
  - Functions that would help
  - Variables that make life easier
- Building an initial test case for the whole program
  - Fix all minor class bugs first
  - Then try to write a file to test the whole program

Week three

(first tests and issue creation)

- Intensive bug testing
  - Whole program is tested
  - What breaks
  - What doesn't behave as expected
  - What looks wrong

- Writing issues for bugs
    - Open GitHub and go to your repository
    - Go to “issues” and create new
    - Label it based on the bug you found
- Fixing (A good bit of this will also be done out of class)
    - After thorough testing
    - Open an issue and add yourself to it
    - Create a new branch off the dev branch
    - Attempt to figure out what causes the issue
    - Write a fix
    - Verify it is fixed
    - Create a merge request
    - Repeat
- Further testing iterations
    - After fixing every issue or when testing your fix
    - Look for other potential bugs/issues
    - Note them down and repeat this cycle
- New functionality
    - If you have no bugs or very few
    - Add new functions and feature
    - Create new branches to test ideas
    - Try to add a couple new pointer types
    - Extra work won’t be merged if you do not have time to finish

Week four

(final testing, program due at midnight w/ write-up)

- Progress check/bug testing
    - Is everything done?
    - Is it all functional?
- Cleaning code
    - Messy code is tidied and simplified
    - Broken code is fixed/removed
    - Program is finalized and merged/rebased into development
- Group writes a finalized program write-up
    - Contains all information from the first write-up (polished)
    - What did your group try to do?
        - What was successful?
        - What was unsuccessful?
    - What was your process?
    - What would you do differently?
    - What did you learn?
    - What is the expected outcome of the program?
    - How will the TA run/test your program
    - Additional information for us:
        - Thoughts
        - Comments
        - Concerns
        - Questions
        - Everyone should write these down when you think them
        - Put them in a central location for your Leader
        - Your leader will compile all of them
            - Take out duplicates
            - Organize and order
            - Anonymize names/information
    - It doesn’t need to be perfect, but please make it neat.
- Make sure you have a test file for your TA!

Examples

In these we have both examples from the previous year as well as a new example written within the last 3 months for you!

Video Game Library

Previously the required Program 1

Driver

- `VideoGameLibrary` object
  - Initialized with the proper size
- While loop
  - Menu
  - Selection
  - Library function calls
    - Adds games
    - Removes games
    - Displays
    - Save
    - Load
  - Close program (exits the loop)

Video Game Library

Stored all the Video Games in a “library”.

Variables

- Pointer to an array of pointers to `VideoGame` objects.
- Integers
  - Max size (array size)
  - Game count (number of active pointers)

Functions

- Constructor
- Destructor
- Display
  - Full details
  - Titles only
- Add
- Remove
- Resize (could be private, used when the library is “full”)
- Save to file
- Load from file

Video Game

A class that stored data about video games, utilized `Text` objects to store the name, author, and publisher. This was the data class.

Variables

- Pointers to `Text` objects
  - Name
  - Author
  - Publisher
- Integers
  - Year
  - Rating

Functions

- Constructor
- Destructor
- Getters for all variables
- Setters for all variables

Text

A class that handled strings (like `std::string` but built by the student). This was the helper class.

Variables

- Pointer to character array
- Size variable

Functions

- Constructor
- Destructor
- Getter for size
- Getter for the string

Car Lot

A new example (puts the bonus class at the top of the hierarchy).

Driver

- Creates the lot manager object
- Creates a value tracker (double)
- While loop
  - Add lot
  - Destroy lot
  - Optional:
    - User operated parking
      - Park in a lot
      - Leave a lot
    - Game operated parking
      - Run a day
      - Game characters do parking calls
      - Done in a separate function
  - Display lots
  - View current income/expenses
  - Exit program

Lot Manager

John Grzegorczyk

Program One

Variables

- Pointer
  - Array of `ParkingLot` objects

Functions

- Constructor
- Destructor
- Build lot
- Destroy lot
- Get lot (returns a pointer/reference to a lot)
- Display lots (prints the lots as well as vacancy)

Parking Lot

Variables

- Pointer
  - Array of `Car` objects
  - Array of integers (times when the cars parked)

Could also be:

- Pointer (layered lots, I chose this)
  - Array of pointers to arrays of `Car` objects
  - Array of pointers to arrays of integers (parking times)
- String
  - Lot name
- Integers
  - Lot capacity
  - Current Occupancy
- Bonus?
  - Lot location

Functions

- Constructor
- Destructor (cool points if you can make it so the cars are warned before the lot is “torn down” otherwise they get destroyed with the lot)
- Park
- Leave (returns the time spent parked in the lot)

Car

Variables

- Strings
  - License Plate
  - Make
  - Model
  - Owner Name
- Enumeration
  - Color (basically a number that represents a color ID)
- Integers
  - Year

Functions

- Constructor
- Destructor

- Accessor functions

Rubric

The rubric is out of 100 total points, but there are possible bonus points!

Driver

10 points

Classes

60 points

Storage Class

20 points

Data Class

20 points

Extra Class

20 points

Memory Management

20 points

Leaked Memory

10 points

Overwriting Pointers

5 points

Failure to Clean

5 points

Segmentation Faults

10 points

Out of Bounds Calls

6 points

Uninitialized Pointer Calls

2 points

Null Pointer Calls

2 points

Other

10 points

Included All Files

1 point

Test Case

3 points

Makefile

1 point

Formal Write-up

5 points