UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS110 Assignment 1

ShowFlix Simulation

Due date: 16 August 2018, 23h59

Total marks: **80**

1 General instructions

- This assignment should be completed individually.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- If your code does not compile you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence of certain functions or classes).
- Read the entire assignment thoroughly before you start coding.
- Submit all your assignment tasks through the COS110 assignments page (`http://assignments.cs.up.ac.za/`) under the correct task slot.
- To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `http://www.ais.up.ac.za/plagiarism/index.htm`.

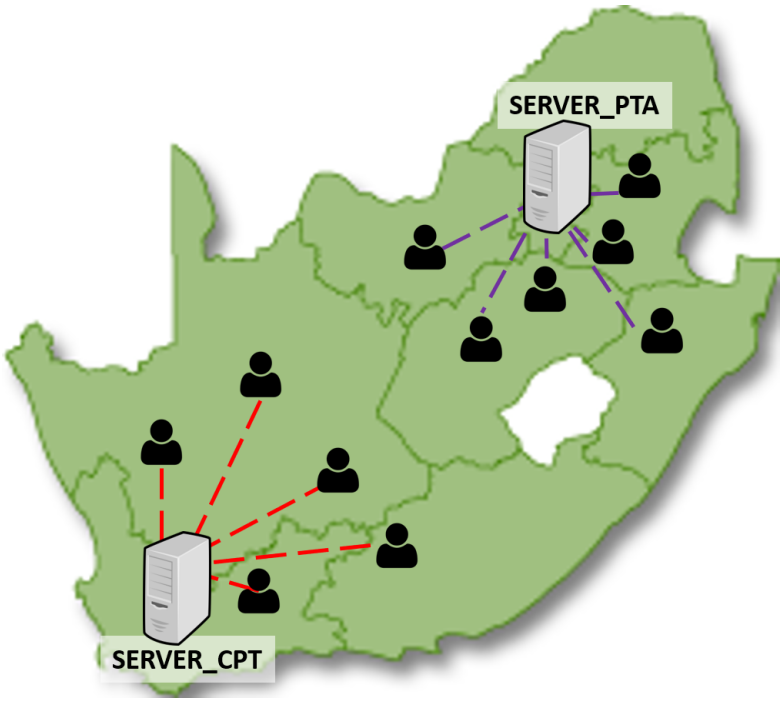
2 Overview

The goal of this assignment is to deepen your understanding and application of C++ classes and dynamic objects through the simulation of a fictional online movie streaming service called ‘ShowFlix’. The scope of this assignment covers chapters 2 to 14.4, with particular attention to chapters 9, 13 and 14.

3 Background

The ShowFlix company wants to simulate its server configuration to see if it will be able to cope with a large number of user download requests. The servers will be placed at different locations across the country, and they will each serve requests in their vicinity. A file with the location and maximum number of requests allowed for each server is provided. User requests will appear randomly and will be served by the closest server to it, provided the server has capacity to deal with that request. If the closest server cannot accommodate that request, it will be sent to the next nearest server until one of the servers can handle the request. If all the servers are at capacity, the user request will simply be dropped.

Figure 1: An example of the server layout for ShowFlix



4 Your task

The assignment is divided into four related tasks wherein you will create 5 classes that interact with one another via public interfaces. Implement the classes in the order given below, and test your code thoroughly to ensure correct operation of your classes before moving to the next tasks. Write a Makefile to make compilation and linking easier.

The classes that you will create for each task are:

1. Location, MovieFile
2. UserRequest
3. Server
4. Simulator

4.1 Task 1: Building Blocks (15 marks)

4.1.1 Location class

The `Location` class contains the `x` and `y` coordinates of a place as its private member variables. Create two files: `Location.h` (class definition) and `Location.cpp` (implementation) that support the following public interface:

Member Functions:

- `Location(int, int)`: constructor that receives the x and y coordinates
- `int getX()`: returns the `x` coordinate
- `int getY()`: returns the `y` coordinate
- `float computeDistance(const Location& loc)`: computes the Euclidean distance between the current and input location objects
- `string toString()`: returns the location coordinates in the following string format: (x,y) (no spaces in between or around the string)

4.1.2 MovieFile class

The `MovieFile` class contains the title, duration and size of a movie file (in MB). Create the definition and implementation files for the `MovieFile` class and implement its public interface as follows:

Member Functions:

- `MovieFile(string, int, int)`: the constructor should receive 3 parameters: the name of a movie file, its duration in minutes, and the file size with a default value of 0
- `MovieFile(const MovieFile& other)`: copy constructor that copies all information from the input object
- `string getName()`: returns the filename
- `int getDuration()`: returns duration of the movie
- `int getFileSize()`: returns file size
- `void setFileSize(int)`: sets file size
- `void appendFileSize(int)`: adds the input value to the current file size. This will be used later to simulate the process of downloading chunks of data at a time

4.1.3 Test and submit your work:

Write a `main` file and test all the functions in your classes. You may use the given `main` file as a starting point for your tests, but make sure to conduct further exhaustive tests. When you are certain that all works as expected, compress the header and implementation files of your two classes (don't include the main file) into a single archive (either .tar.gz, .tar.bz2 or .zip - make sure that there are no folders or subfolders in the archive) and submit it for marking to the appropriate upload slot (**Assignment 1, Task 1**) before the deadline. Note that you do not have to upload a makefile for this task. Also note that the Automated Marking Tool is not a debugger - the number of uploads is limited and should be used wisely.

4.2 Task 2: User Download Request (15 marks)

The `UserRequest` class is used to initiate a download request from the server. The class contains the location of the user requesting to download, and details of the file to be downloaded. Each `UserRequest` has a **unique** ID assigned to it, with the first object assuming a value of 1. The ID should be incremented by 1 for each subsequent request. Create the definition and implementation files for the `UserRequest` class. Define the member variables of the class as you see fit (*Hint:* the functions should give you an idea of the required variables for the class). You will need your implementations from the previous task to complete this task.

Member Variables:

- `UserRequest(Location* loc, int fileIndex)`: This constructs the `UserRequest` object with the given location and index of a file that the user wishes to download
- `Location* getLocation()`: returns the user request location
- `int getMovieIndex()`: returns the index of the movie file as initialized in the constructor
- `int getRequestId()`: returns the unique ID of the user request object
- `bool isComplete()`: returns the status of the user request (i.e. if it has been fulfilled or not)
- `void setComplete(bool)`: sets the status of the user request (i.e. if it has been fulfilled or not)
- `void download(MovieFile* video, int bytes)`: initiates the download of a movie and specifies the bytes (file size) transferred in the initial download. It is only called once to start the download process. A deep copy of the input video should be made and stored in the `UserRequest` object.
- `void download(int data)`: this function is used for subsequent download activity, and simulates the continued receipt of data as the movie is being downloaded. The input signifies **additional** chunk of data (in MB) to be added to the overall file size.
- `int getAmountDownloaded()`: returns the current file size
- `MovieFile* getDownloadedFile()`: returns the downloaded file if it exists (and `null` if it does not).

Remember to properly discard allocated memory when the class ceases to exist.

4.2.1 Test and submit your work:

Write a `main` file and test all the functions in your class. You may use the given `main` file as a starting point for your tests, but make sure to conduct further exhaustive tests.

Files to upload:

- `Location.h`, `Location.cpp`
- `MovieFile.h`, `MovieFile.cpp`
- `UserRequest.h`, `UserRequest.cpp`

When you are certain that all works as expected, compress the files above into a single archive (either .tar.gz, .tar.bz2 or .zip - make sure that there are no folders or subfolders in the archive) and submit it for marking to the appropriate upload

slot (**Assignment 1, Task 2**) before the deadline. Note that you do not have to upload a makefile for this task. Also note that the Automated Marking Tool is not a debugger - the number of uploads is limited and should be used wisely.

4.3 Task 3: Server (30 marks)

This task brings us closer to the heart of the assignment: the 'ShowFlix' simulation. You are required to make use of the `Location`, `MovieFile` and `UserRequest` classes from the previous tasks to implement the `Server` class functionality.

A server is characterized by its name, location, download speed, maximum number of user requests it can handle, and a collection of movies in its storage. Assume that the download speed (MB/s) is consistent throughout all downloads of a specific server. You have been given a class specification file, `Server.h`, that is well-commented which you should implement in `Server.cpp`. Do **not modify** the given `Server.h` file as it will be overwritten on the Automated Marking Tool.

The following provides further information on selected functions:

Member Functions:

- `void assignRequest(UserRequest* userReq)`: makes a **shallow** copy of the `UserRequest` object and adds it to the server's list of requests.
- `void runDownloads()`: the server should loop through the list of its user requests, and download a chunk of data according to the server's download speed. Assume that the function will be **called once per second**. For example, if a `UserRequest` object wishes to download a file of size 135MB, on a server that has a download speed of 20MB/s, that means every time the `runDownloads` function runs, 20MB of data will be added to that `UserRequest` , until it has strictly transferred a total size of 135MB (and not a megabyte more). The function should call the relevant `UserRequest`'s `download` function depending on whether a download has been previously initiated or not. At the end of transferring the chunk of data, the function should check if the `UserRequest` has been fulfilled, and if so, it should set the necessary flag.
- `void dropCompletedRequests()`: The objective of this function is to drop all user requests that have been fulfilled, and remove them from the server's list of connnected users. This will require you to shift the elements in the `requests` array (member variable) to make sure that the elements are stored in contiguous locations (i.e. in sequential order with no empty elements between them).
- `void printInfo()`: This function prints out an abridged version of the server's details. It should print out a single line with the following information: the

name and location of the server, and the number of currently connected users/active requests. Here is an example:
`Server_PTA, Location: (40,70), #Connections: 4`

- `void printDetailedInfo()`: This function prints out a detailed version of the server's details. It should print out the name, location and number of connected user requests in a single line. It should then print out the list of all user requests and the amount of data already downloaded. Indent the user request information (you can use a tab or any number of spaces). See below for an example.

```
Server_PTA, Location: (40,70), #Connections: 4
      Request 1, Downloaded data: 50MB
      Request 2, Downloaded data: 30MB
      Request 3, Downloaded data: 20MB
      Request 4, Downloaded data: 10MB
```

Remember to properly discard allocated memory when the class ceases to exist.

4.3.1 Questions to ponder...

- When a `UserRequest` object is assigned to a server, who is responsible for its deletion? The `Server` class or the `main` function?

4.3.2 Test and submit your work:

Write a `main` file and rigorously test all the functions in your class.

Files to upload:

- `Location.h`, `Location.cpp`
- `MovieFile.h`, `MovieFile.cpp`
- `UserRequest.h`, `UserRequest.cpp`
- `Server.cpp`

When you are certain that all works as expected, compress the files above into a single archive (either .tar.gz, .tar.bz2 or .zip - make sure that there are no folders or subfolders in the archive) and submit it for marking to the appropriate upload slot (**Assignment 1, Task 3**) before the deadline. Note that you do not have to upload a makefile for this task. Also note that the Automated Marking Tool is not a debugger - the number of uploads is limited and should be used wisely.

4.4 Task 4: Simulation (20 marks)

This task combines everything that has been implemented so far to simulate the movie streaming system. The `Simulation` class simulates how randomly generated user requests are assigned to the nearest server based on availability, where at every second (iteration), each server will run its file download functionality, followed by the dropping of fulfilled requests. This is detailed in the `run` function of the class.

You have been given a partially defined class in `Simulator.h`. Create a file called `Simulator.cpp` and implement the public interface as defined in the header file. You are free to add any private members (variables and functions) in your definition and implementation files. Your `Simulator` class will be tested against the **memo versions** of the classes created in tasks 1 to 3.

Below is a description of some of the public functions that you should implement.

Member Functions:

- `Simulator(string serverFile, string movieFile, int seed = 123)`: receives the filenames that contain the server and movie information, and a seed value to seed the random number generator. The constructor should read the files and load the information onto `servers` and `movies` arrays (it can do this directly or through helper functions). See the sub-section below on how these files are formatted.
- `Location* generateRandomLocation()`: generates and returns a random x and y coordinate of a Location object, where the values are between the specified `MIN_LOCATION_VALUE` and `MAX_LOCATION_VALUE` constants.
- `int generateRandomMovieIndex()`: returns a random index between 0 and the number of movie files loaded through the class constructor.
- `void run(int maxDuration, float demandProb = 0.5, bool verbose = true )`: this function implements the behaviour of the simulation, and iterates every second until `maxDuration` is reached. It uses the given demand probability value (`demandProb`) to generate a random user request based on the following probability rule:
  `if rand()< demandProb ? generate new request : do not generate request`

  If a new request was generated, it should be assigned to the closest server based on the Euclidean distance computation. Should that server be full, the request should be passed to the next closest server. If all servers are full, the request should then be discarded with the following message: `"SERVERS FULL: Request _REQUEST_ID dropped"`, where `_REQUEST_ID` is the unique ID of the `UserRequest` object.

The `run` function should thereafter run all server downloads for the current iteration, and then reconcile the system by dropping all fulfilled requests from all servers.

When the `verbose` input parameter is given as `false`, the function should not print out anything to the console.

- `void printServerInfo()`: should print out an abridged version of all server information. The print out should look as follows:

```
-----------------------------------------------------
      Server Report
-----------------------------------------------------
Server_CPT, Location: (50,30), #Connections: 0
Server_PTA, Location: (40,70), #Connections: 4
Server_JNB, Location: (70,20), #Connections: 0
-----------------------------------------------------
```

  The spacing and the length of the dotted lines should be set at free will, but must be there.

4.4.1 Input Data

Server File
The first line in the server file specifies the number of servers that will be used in the simulation, and the rest of the file contains information about those servers, where each line is formatted as follows:
`SERVER_NAME X_COORD Y_COORD DOWNLOAD_SPEED MAX_USERS`

The `X_COORD` and `Y_COORD` values specify the location of the server. The `MAX_USERS` value specifies the maximum number of user requests that the specific server can handle. The other values are self-explanatory.

Movie File
The first line in the movie file specifies the number of movies each server contains. The lines that follow provide details of the movies, formatted as follows:
`MOVIE_DURATION FILE_SIZE MOVIE_TITLE`

The `MOVIE_DURATION` specifies the length of the movie in minutes, while `FILE_SIZE` specifies the size of the movie in megabytes (MB). The `MOVIE_TITLE` variable is the name of the movie which can contain multiple words.

4.4.2 Test and submit your work:

Write a `main` file and test all the functions in your class. You may use the given `main` file as a starting point for your tests, but make sure to conduct further exhaustive tests.

Files to upload:

- `Simulator.cpp` and `Simulator.h` only

When you are certain that all works as expected, compress `Simulator.h` and `Simulator.cpp` into a single archive (either .tar.gz, .tar.bz2 or .zip - make sure that there are no folders or subfolders in the archive) and submit it for marking to the appropriate upload slot (**Assignment 1, Task 4**) before the deadline. Note that you do not have to upload a makefile for this task. Also note that the Automated Marking Tool is not a debugger - the number of uploads is limited and should be used wisely.

5 Implementation Considerations

- Read the instructions carefully, and make sure you implement every function as stated.
- For every **new** operator, there should be a corresponding **delete** operator. Make sure that every dynamic object is being released.
- Let `cout` be your debugging friend! When initially coding, have simple, but descriptive `cout` statements that help you track every object that's being created and deleted (think constructors and destructors). Remember to remove these additional print outs when you are done. Alternatively, you can make use of available C++ debugging tools.
- All queries regarding the assignment should be **posted** on the **ClickUp online discussion forum**. Questions sent through other means (e.g. cos110queries@cs.up.ac.za or lecturer email addresses) run the risk of **not** being answered. Post your question under the relevant task thread. Make sure to post as little code as possible so you don't expose yourself to being plagiarised. Your wording in your queries should demonstrate that you have attempted something, and that you need clarification. Take note, **this is not a debugging service**.
- The deadline for asking questions on the forum (and getting replies from the staff) is **14 August 2018, 18:00**. So make sure you start the assignment in time.
- **Take heart, be strong and enjoy the beauty of programming!**

```
The End
```