COMP 2404 – Final Project

Due: Friday, December 11 at 11:59:00 pm

1. Goal

For this project, you will design and implement a C++ program that generates a set of reports based on data from Canada's National Graduate Survey (NGS) from the years 2000-2015. Your code will be correctly separated into object design categories, and it will meet every principle of good software engineering that we have covered in class.

The grading of projects in this course will be rubric-based. The **mandatory minimum criteria** for grading is that all the requirements must be implemented, they must meet all the listed constraints, they must execute correctly, and they must print the required reports to the screen. Submissions that are incomplete, or that don't work correctly, or that don't meet all the constraints, or that don't print out resulting reports, will earn a grade of zero.

2. Data Set and Reports

You will use the NGS data provided for you in the `grad.dat` file posted in *cuLearn*. This file contains data on the number of post-secondary graduates, their gender, what degree they earned, and how many of them were employed after graduation, for each province in Canada, and in Canada as a whole, in the years 2000, 2005, 2010, and 2015.

Each record (i.e. each line) in the data file gives us employment statistics for a given year, region, degree, and gender (including the combination of data for "All" genders). That record tells us how many post-secondary students who self-identified as that gender, who were living in that region, and who graduated that year, with that degree, responded to the survey, and how many of those respondents were employed after graduation.

The records are formatted as follows: `<year region degree gender numEmployed numGrads>`
where `year` indicates a graduation year; `region` indicates a region (either a province, or CAN for all of Canada) where graduates lived; `degree` specifies a post-secondary degree, as either College, Bachelor's, Master's, or Doctorate; `gender` tells us whether the numbers in that record are for all genders, or males only, or females only; `numGrads` tells us the total number of post-secondary graduates of that gender who graduated with that degree, during that year, and who lived in that region; and `numEmployed` indicates the subset of that total number who were employed after graduation.

For example, the first record reads: `2000 AB Bachelor's All 6900 7500`
and a later record tells us: `2000 AB Bachelor's Females 4140 4500`

The first record tells us that, in the year 2000, there was a total of 7500 students of all genders who graduated with a Bachelor's degree and who lived in Alberta. Of those 7500, 6900 found employment after graduation. The second record tells us that in the same year in Alberta, 4500 graduates with a Bachelor's degree self-identified as female, and 4140 of these female graduates were employed.

Your program will use the NGS data provided in order to generate a total of five (5) reports, upon request by the user, including the following three (3) reports:

2.1. the employment percentage for each region, by degree, for all years and all genders
   2.1.1. each region will be a row, and each type of degree will be a column
   2.1.2. each cell will show the percentage who were employed in that region, with that degree, for all years and all genders combined, compared to the number of graduates in that region

2.2. the top 3 and bottom 3 regions for percentage of female graduates, for all years and all degrees
   2.2.1. each region will be a row, and there will be one percentage column
   2.2.2. each cell will show the proportion of female graduates, compared to the total number of graduates in that specific region, employed and unemployed combined, for all years and all degrees combined

2.3. the employment proportion for each region, by year, for all degrees and all genders

2.3.1. each region will be a row, and each year will be a column

2.3.2. each cell will show the proportion who were employed in that region, for that specific year, for all degrees and all genders combined, compared to the total who were employed in Canada overall, for that same year

3. Program Requirements

You will implement a program that reads in the graduate data from the `grad.dat` file posted in *cuLearn*, and you will present the user with a menu of possible reports. When the user selects a report, your program will compute the statistics required for the report using the graduate data, and print the information to the screen.

Your program will implement the following requirements:

3.1. **Report requirements**

Your program will allow the user to execute the following reports:

3.1.1. the three (3) reports that are described in section 2, and

3.1.2. two (2) additional reports of your own creation

The reports of your own creation must meet all of the following requirements:

3.1.3. they must represent a significant amount of new logic and new code

3.1.4. they must represent a different way of thinking from the existing reports described in section 2

3.1.5. they cannot be a simple variation of the existing reports

(a) for example, the first report in section 2 produces the employment percentages for all graduates, by region, by degree; it would **not** be acceptable to simply do the same for male graduates or female graduates only, since that would be the same logic as an existing report

3.1.6. they cannot be a simple inversion of the rows and columns of an existing report

3.1.7. your new reports must be thoroughly described in your README file; if your new reports are not explained there, the corresponding code will not be graded

3.2. **Design requirements**

3.2.1. The program will be separated into objects that fit into the control, view, and entity object design categories. You will have one view object, many entity objects, one primary control object that is in charge of the program control flow, and several additional control objects that are responsible for report generation.

3.2.2. You may use the STL `vector` container anywhere you wish in this program. However, the use of STL algorithms is prohibited.

3.2.3. You will design a class to hold each record read in from the data file. The data members must be declared with the correct data type. For example, do not use strings to hold numeric values.

3.2.4. All entity and control objects must be dynamically allocated, with the exception of the primary control object. All dynamically allocated memory must be explicitly deallocated when no longer in use. This deallocation must be implemented manually. Do not use C++11 techniques like shared pointers.

3.3. **User I/O requirements**

3.3.1. Only the view object will read input from the user and print output to the screen. Because the view object should never contain entity or control knowledge, all output must be formatted by the entity and control objects as large strings. The primary control object will then print out these large strings by calling a print string function on the view object.

You may use the `stringstream` class to format data into strings. You can refer to the simple coding example found here.

3.3.2. A menu will be presented to the end user. The user will select an option, and the program will compute and generate the statistics for the corresponding report. Once the report statistics have been printed to the screen, the menu will be displayed again, until the user chooses to exit.

3.3.3. The menu that is presented to the user will **not** be hard-coded, and the different options will not be fixed. As described in a later step, the primary control object will store a collection of report generator objects, one for each type of report. In order to show a menu to the user, the primary control object will query each report generator in its collection, retrieve the name of each report, and present those names to the user as the menu options. When the menu options are shown on the screen, they must be numbered sequentially, starting at 1, with option 0 as the Exit option.

3.4. Control object requirements

3.4.1. The `main()` function will contain only two lines of code: one to declare a primary control object, and the other to call that object's launch function.

3.4.2. In addition to the primary control object that manages the program control flow, your program will implement an inheritance hierarchy of control objects that are responsible for report generation. The base class of this hierarchy will be the abstract `ReportGenerator` class, and there will be one derived, concrete class for each type of report that the user can run. This includes the reports described in section 2, and the new ones that you create, as described in requirement 3.1.

3.4.3. The primary control object will create one instance of each concrete report generator class. It will store these objects in a collection as one of its data members. When the user requests a report, the primary control object will invoke a polymorphic function on the correct report object to do the work.

3.5. Polymorphism requirements

Your program will implement an inheritance hierarchy of report generator classes, which must include polymorphic behaviour, as described below.

You will implement the `ReportGenerator` base class, which will contain, at minimum:

3.5.1. a **`static`** data member that stores the collection of all the data that is read in from the data file
   (a) it is necessary that this member be static, since `ReportGenerator` is an abstract class, and we cannot create any instances of it
   (b) you will need to design a class to contain the information for each individual record
   (c) this data member will be an STL `vector` of record pointers
   (d) this data member will store the *primary collection* of all the NGS data from the data file

3.5.2. several static data members that store *partial collections* of the data, but organized in a way that facilitates the generation of reports; these collections **must** be used when generating the reports
   (a) at minimum, you will define, populate, and traverse for report generation the following: a partial collection organized by year, one organized by region, and one organized by degree
   (b) each partial collection will be defined as a collection of `Property` object pointers (described in a later step); each `Property` object stores a collection that is a subset of the records in the primary data collection; the records in a `Property` object are there because they meet a specific criteria
   (c) each element inside these partial collections will hold the records for one specific *value* of a property; for example:
      (i) one of the properties in the NGS data records is the `year`, because every record represents data for a specific year; so the `ReportGenerator` base class will contain a partial collection organized by year; let's say we call that partial collection `allYears`
      (ii) the NGS data contains statistics for only four different years (2000, 2005, 2010, and 2015), so the `allYears` partial collection will contain exactly four elements, one for each year; the first element of the `allYears` collection will contain all the records for the year 2000; the second element will contain those for the year 2005, and so on for the years 2010 and 2015
      (iii) we will not be not making *copies* of the data records; instead, every element of the `allYears` collection will contain a collection of **pointers** to records that are already in the primary data collection

3.5.3. a static member function to load all the records from the data file into the primary data collection

3.5.4. a data member that stores the name of the report produced by the specific report generator class

3.5.5. a polymorphic `void execute(string& outStr)` that is implemented by each concrete derived class to generate the corresponding report

3.5.6. any additional helper functions that are required for a good OO design

You will implement the report generator concrete classes. Each will be derived from `ReportGenerator`, and there will be one concrete class for each type of report described in section 2 and requirement 3.1.

3.5.7. each concrete class will implement the polymorphic `void execute(string& outStr)` function to compute the statistics required for that particular report

3.5.8. the report results must be formatted into one large string, which will be saved to the `outStr` parameter and returned to the calling function; it is the responsibility of the primary control object to use the view object to print that string to the screen

3.6. **Class template requirements**

The partial collection data members described in requirement 3.5.2 will each be defined as a collection of `Property` object pointers, where `Property` is declared as a class template that is specialized for the data type of that property.

For example, because the `year` property is an integer, the `allYears` collection will contain pointers to `Property` objects that are specialized for the integer data type (so where `<T>` is instantiated to `int`).

The partial collections organized by properties that are strings will store pointers to `Property` objects that are specialized for the string data type.

You will define the `Property` class template to contain:

3.6.1. a value of data type `<T>`, which contains the value for that particular element

(a) for example, the first `Property` element in the partial collection organized by year will have the value `2000` to represent the year 2000

(b) the first `Property` element in in the partial collection organized by region will have the value "`AB`" to represent the province of Alberta

3.6.2. a collection that contains pointers to all the data records that have the specific property value

(a) for example, the first `Property` element in the year partial collection will contain the data records for the year 2000

(b) the first `Property` element in the region partial collection will contain the data records for the province of Alberta

The `Property` class template must overload the following operators:

3.6.3. the addition assignment operator `+=` that adds a new element to the back of the record collection

3.6.4. subscript subscript operator `[]` that returns the record pointer at the given index

NOTE: Your code must use these operators when initializing and accessing the partial collections.

4. Constraints

Your program must comply with all the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

4.1. The code must be written in C++98, and it must compile and execute in the default course VM. It must not require the installation of libraries or packages or any software not already provided in the VM.

4.2. Your program must follow correct encapsulation principles, including the separation of control, view, entity, and collection object functionality.

4.3. Your program must not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.4. Your program must follow basic OO programming conventions, including the following:

4.4.1. Do not use any global variables or any global functions other than `main()`.

4.4.2. Do not use `struct`s. You must use classes instead.

4.4.3. Objects must always be passed by reference, never by value.

4.4.4. Except for simple getter functions, data must be returned using output parameters, and not using the return value.

4.4.5. Existing functions must be reused everywhere possible.

4.4.6. All basic error checking must be performed.

4.4.7. All dynamically allocated memory must be explicitly deallocated.

4.5. All classes must be thoroughly documented in every class definition, as indicated in the course material, section 1.3.

5. Submission

5.1. You will submit in *cuLearn*, before the due date and time, the following:

5.1.1. A UML class diagram (as a PDF file), drawn by you using a drawing package of your choice, that corresponds to the entire program design.

5.1.2. One `tar` or `zip` file that includes:

(a) all source and header files

(b) a Makefile

(c) a README file that includes:

(i) a preamble (program author, purpose, list of source and header files)

(ii) compilation and launching instructions

(iii) a description of the new reports created for requirement 3.1

NOTE: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *cuLearn* will be graded. Submissions that contain incorrect, corrupt, or missing files will receive a grade of zero. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading

The grading of this project will be rubric-based, and the total grade will be out of 60 marks.

6.1. **Minimal criteria**

In order to be graded, your program must meet the following **minimum criteria**:

6.1.1. The program must be implemented in C++98, and it must compile and execute in the default course VM.

6.1.2. A correct Makefile must be provided.

6.1.3. The program must successfully complete an execution that meets all requirements and constraints described in sections 3 and 4.

6.1.4. To be graded, code must be invoked, and the corresponding data must be printed to the screen.

Submissions that do not meet this minimum criteria will not be graded and will earn a grade of zero.

6.2. **Rubric-based grading**

The project will be assessed in accordance with the grading categories listed in section 6.3. Each category will be graded based on a 6-point rubric. Some categories are worth double the weight of other categories, so their value is 12 points each. These will still be assessed according to the 6-point rubric, but the values will be doubled.

The 6-point rubric is defined as follows:

- [6 points] Excellent (100%)
- [5 points] Good (83%)
- [4 points] Satisfactory (67%)
- [3 points] Adequate (50%)
- [2 points] Inadequate (33%)
- [1 points] Poor (17%)
- [0 points] Missing (0%)

6.3. Grading categories

The project will earn an overall grade out of 60 marks. This will be the sum of the grades earned for each of the following categories, based on the 6-point rubric described in section 6.2:

6.3.1. **UML [6 marks]:** This criteria evaluates the UML class diagram corresponding to the design and implementation of your entire program. The diagram must be drawn by you, using a drawing package, and it must be submitted as a PDF file. Other formats will not be graded. Your UML class diagram must comply with the UML format and conventions covered in the course material, section 2.3.

6.3.2. **Object categories and function design [12 marks]:** This criteria evaluates how correctly the functionality of your implementation is separated into control object(s), view object(s), entity objects, and collection object(s), how effectively the functionality of your implementation is separated into modular functions, and how well your object and function design follows the correct software engineering rules of data abstraction and encapsulation.

6.3.3. **Code quality [12 marks]:** This criteria evaluates the overall code quality. This includes, but is not limited to, how well your classes are implemented, how well the code is written, how well the principle of least privilege is applied, the presence of constructors and destructors where appropriate, how well the code complies with all constraints listed in section 4, etc.

6.3.4. **Polymorphic report generation [12 marks]:** This criteria evaluates the effective development and usage of polymorphism in your code, and how it is used in the behaviour of the control objects, as described in the program requirements in section 3.

6.3.5. **Class template [6 marks]:** This criteria evaluates the effective development and usage of the new class template in your code, as described in the program requirements in section 3.

6.3.6. **Innovation [6 marks]:** This criteria evaluates the innovative reports that you add to your program. These reports must be original and substantial. They must represent a different way of thinking from the existing reports, and they must represent a significant amount of original logic and additional code. The new reports will be evaluated based on how much additional code is implemented, and the creativity and the programming sophistication shown in selecting and implementing these reports. New reports that are not described in the README file will not be graded.

6.3.7. **Packaging [6 marks]:** This criteria evaluates the packaging of the code into an archive file. This includes, but is not limited to, the correct separation of the code into header and source files, the presence of complete and correctly formatted Makefile and README files, and the absence of duplicate, additional, and/or unneeded files.