

## 1. Goal

For this project, you will design and implement a C++ program that generates a set of reports based on data from Statistics Canada's *Other livestock*<sup>1</sup> census, for the years 2011 and 2016. Your code will be correctly separated into object design categories, it will be designed for easy extensibility to additional categories of data, and it will meet every principle of good software engineering that we have covered in class.

The **mandatory minimum criteria** for grading this project is that all the requirements must be implemented, the code must meet all the listed constraints, it must execute correctly, and it 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 Statistics Canada census data provided for you in the `farms.dat` file posted in *cuLearn*. This file contains data on nine types of animals living on farms in Canada, including the number of farms that host the animals, and the total number of animals living on those farms, for each region (province or territory) and agricultural sub-region in Canada, for the census years 2011 and 2016.

Statistics Canada's *Other livestock* census category includes nine types of animals: horses/ponies, goats, llamas/alpacas, rabbits, bison/buffalo, elk/wapiti, domestic deer, wild boars, and mink.

Each record (i.e. each line) in the census data file gives us farm and animal statistics for a given year, region, agricultural sub-region, and type of animal. That record tells us how many farms reported data for that type of animal, in that sub-region, for that year, and it tells us how many animals were reported by those farms.

The records are formatted as follows: `<year region subRegion animalType numFarms numAnimals>` where `year` indicates a census year; `region` indicates either a province, or a territory, or "CAN" for all of Canada, where the farms and animals are located; `subRegion` specifies the agricultural sub-region for those farms and animals; `animalType` tells us which type of animal was counted for that record; `numFarms` tells us the number of farms that reported hosting animals of that type in that sub-region, for that year; and `numAnimals` indicates the total number of animals of that type reported as living on those farms.

For example, the first record reads: `2011 CAN All Horses-Ponies 47454 392340`  
and a later record tells us: `2016 ON Eastern-Ontario-Region Llamas-Alpacas 143 944`

The first record tells us that, in the census year 2011, there were 47,454 farms that reported they were hosting horses and ponies in Canada overall. Altogether, those farms reported that a total of 392,340 horses and ponies lived on those farms. The second record tells us that, in 2016, in the Ontario sub-region of Eastern Ontario, there were 143 farms that reported having llamas and/or alpacas, and altogether, they reported a total of 944 llamas and/or alpacas living on those farms.

Your program will use the census data provided in order to generate the following three (3) reports, when requested by the user:

- 2.1. a report listing the regional percentage breakdown of farms that hosted animals of each type in 2016
  - 2.1.1. each row will show data for a region (province or territory), excluding Canada
  - 2.1.2. each column will represent a type of animal, in addition to one last column showing the region's percentage over all animals
  - 2.1.3. each cell will show the percentage breakdown, for each region, of farms hosting that specific type of animal, over all regions

---

<sup>1</sup>Statistics Canada, [Table 32-10-0427-01 Other livestock on census day](#).

- 2.1.4. the cells of the last column will show the percentage breakdown, for each region, of farms hosting all types of animals combined, over all regions
  - 2.1.5. the report rows will be ordered in descending order by regional percentage of all animal types, as shown in the last column
  - 2.1.6. each column will add up to 100%
  - 2.1.7. do not include Canada as a region in your calculations
  - 2.1.8. do not include individual sub-regions in your calculations; you must use the numbers for the "All" sub-regions only
  - 2.1.9. do **not** assume that the numbers for each region add up to the numbers for Canada; these are real-life statistics, and like most statistics, they contain omissions; to ensure that your columns add up to 100%, you must compute the correct number for the denominator for calculating the percentages
- 2.2. a report listing the animal type percentage breakdown for Canada, and for each animal type, the percentage change from 2011 to 2016
- 2.2.1. each row will show data for an animal type
  - 2.2.2. there will be three columns:
    - (a) the percentage of animals of that type in Canada, over all animal types, in 2011
    - (b) the percentage of animals of that type in Canada, over all animal types, in 2016
    - (c) the change in percentage for each animal type, from 2011 to 2016
  - 2.2.3. for example, horses and ponies represented 21.7% of all *Other livestock* animals in Canada in 2011, and 20.5% in 2016; this indicates a change of -1.2%
  - 2.2.4. the first two columns will each add up to 100%
  - 2.2.5. the rows will be ordered in ascending order by percentage change, as shown in the last column
  - 2.2.6. you must show the positive or negative indicator (+/-) for the percentage change
  - 2.2.7. you will use only Canada as a region in your calculations
- 2.3. a report listing the sub-region, within each region, that hosted the highest number of horses and ponies in 2016
- 2.3.1. each row will show data for a region (province or territory), excluding Canada
  - 2.3.2. there will be three columns:
    - (a) the region
    - (b) the sub-region, within that region, that hosted the highest number of horses and ponies in 2016
    - (c) the number of horses and ponies in that sub-region in 2016
  - 2.3.3. the rows will be ordered in descending order by number of animals

### 3. Program Requirements

You will implement a program that reads in the census data from the `farms.dat` file posted in *cuLearn*, and you will present the user with a menu of the three possible reports. When the user selects an option, your program will compute the statistics required for the corresponding report, using the census data. It will print the results to the screen, and save them to a text file unique to that report.

Your program will implement the following requirements:

#### 3.1. Design and implementation requirements

- 3.1.1. The program will be separated into objects that fit into the control, view, entity, and collection 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, several additional control objects that are responsible for report generation, and several objects of one collection class.



- 3.1.2. Your design must be fully scalable and extensible. This means that it must support any number and any value of years, regions, and animal types. You must not hard-code any of these values, or even make any assumptions about how many values there could be.
- 3.1.3. You will use the STL `vector` container in some portions of this program, where indicated in the instructions. Only some member functions of the `vector` class will be permitted, again indicated in the instructions. The basic member functions, such as `size()`, `push_back()`, `at()`, and the overloaded subscript operator are always permitted. **No other STL containers, and no STL algorithms, are permitted in this program.**
- 3.1.4. **DO NOT** use the STL `map` container anywhere in this program. We will be implementing our own version of a map, which will be used to generate the reports.
- 3.1.5. You can use the `stringstream` class and the `iomanip` library as needed.
- 3.1.6. 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.
- 3.1.7. 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, and in the correct portion of the program.
- 3.1.8. Do not use C library functions. You must use their C++ equivalent.
- 3.1.9. Do not use C++11 techniques like shared pointers, [for-each](#) loops, or [range-based](#) loops.

## 2. User I/O requirements

- 3.2.1. Report results must be computed and generated *on demand* only, when the end user requests the report. They must not be computed in advance.
- 3.2.2. 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 results have been printed to the screen and saved to a file, the menu will be displayed again, until the user chooses zero (0) to exit.

## 3.3. Control object requirements

- 3.3.1. 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.
- 3.3.2. 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, as described in section 2.
- 3.3.3. The primary control object will create one instance of each concrete report generator class. It will store these objects in a STL `vector` collection of **ReportGenerator pointers**, 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.  
**NOTE:** Remember that polymorphism only works with base class pointers. If you invoke a function on an object, or on a derived class pointer, there is no dynamic binding, therefore no polymorphism.

## 3.4. Report generator base class 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:

- 3.4.1. a **static** data member that stores the collection of all the records read in from the census 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 a STL `vector` of record pointers
  - (d) this data member will store the *primary collection* of all the census data

- 3.4.2. several `static` data members that store different *maps* of the data, organized in a way that facilitates the generation of reports; these maps **must** be populated when loading the census data from the data file, and they **must** be used to retrieve the data records when generating the reports
- (a) you will define, populate, and traverse for report generation the following three maps: a year map to organize records for each census year, a region map to organize records by region, and an animal map to organize records by animal type
  - (b) each map will be defined as an object of the `Map` class template (described in a later step), as follows:
    - (i) each map will contain a collection of keys, and a collection of values
    - (ii) the keys collection will contain the values that match the kind of map; for example, the keys for the year map will be the integers 2011 and 2016, the keys for the region map will be the region names, as strings ("ON", "QC", "AB", etc.), and the keys for the animal map will be the animal types, as strings
    - (iii) the keys collection will be stored as a STL `vector` of the data type of the keys (integer or string)
    - (iv) the values will be stored as a STL `vector` of STL `vectors` of record pointers
    - (v) the map's values collection will contain a collection of records for each key in the keys collection
    - (vi) the element (the collection of records) contained at a given index of the values collection will correspond to the key contained at the **same** index of the keys collection
  - (c) for example:
    - (i) the census data contains statistics for only two different years (2011 and 2016), so the year map's keys collection will contain exactly two elements: the integers 2011 and 2016.
    - (ii) the year map's values collection will *also* contain two elements: the first element will be a collection of all records for the year 2011, and the second element will be a collection of all records for the year 2016
    - (iii) we can use the year map to find all the records for 2016 by first searching the keys collection to find the index where the value 2016 is stored, then retrieving the records collection at the same index in the values collection
    - (iv) note that we will not be making any *copies* of the data records; instead, every element of every map's values collection will consist of a collection of **pointers** to records that are already in the primary data collection
- 3.4.3. one or more `static` member functions to load all the records from the census data file into the primary data collection, and to populate the three maps described above
- 3.4.4. a pure virtual `void compute()` member function that is implemented by each derived, concrete class to generate the corresponding report
- 3.4.5. any additional helper functions that are required for a good OO design and implementation, including clean up functions

### 3.5. Report generator derived classes requirements

You will implement a concrete report generator class, derived from the `ReportGenerator` base class, for each type of report described in section 2. Each concrete class will contain, at minimum, the following:

- 3.5.1. a data member to store the report results, as a dynamically allocated `ReportData` object (described in a later step); this object will be initialized to store the results as a collection of report rows, stored in a specific order (ascending or descending), by using a behaviour class (also described later) that polymorphically performs comparisons between data when adding a new report row
- 3.5.2. a constructor that creates the `ReportData` object with a comparison behaviour object as parameter
- 3.5.3. a destructor, as required



- 3.5.4. a polymorphic `void compute()` function that computes the results for that particular report; the concrete report generator classes will **not** use the primary collection of records to access the census data; instead, they **MUST** use the maps created in an earlier step; specifically:
  - (a) the report described in 2.1 must retrieve records from the *region map* to do its computations
  - (b) the report described in 2.2 must retrieve records from the *year map*
  - (c) the report described in 2.3 must retrieve records from the *animal map*
- 3.5.5. a `formatData()` member function that takes the computed statistics, and uses them to populate the `ReportData` data member; you may design this member function to take the required parameters
- 3.5.6. a `printReport()` member function that uses the `ReportData` data member to both print the report results to the screen, and save them to a text file unique to that report
- 3.5.7. any additional helper functions that are required for a good OO design, or that are necessary to break up the code for an elegant, modular design

### 3.6. Map class template requirements

You will implement a `Map` class template that stores a collection of keys and a collection of values. Each value corresponds to one key, and each value is itself a collection of data record pointers. The overall structure of this class will model a *parallel array* structure, where a key at a given index in the keys collection corresponds to a value at the **same** index in the values collection.

The `Map` class template will be parameterized for the key data type, which we will denote as `<T>`. For the year map, for example, the keys will be integers. For the region and animal maps, the keys will be strings.

The `Map` class will contain the following data members:

- 3.6.1. a data member for the keys collection, stored as a STL `vector` of elements of type `<T>`
- 3.6.2. a data member for the values collection, with each element corresponding to the key at the same index in the keys collection; the values collection will be stored as a STL `vector` whose elements are STL `vectors` that contain record pointers
- 3.6.3. the number of elements in the keys collection will always equal the number of elements in the values collection
- 3.6.4. for example, in the *region map*:
  - (a) the keys for the region map will be the collection of regions (provinces and territories), stored as strings: "AB", "ON", "QC", etc.
  - (b) the element in the values collection corresponding to the key "AB" will be the collection of all the data records for the province of Alberta; the element corresponding to the key "ON" will be the collection of all the records for the province of Ontario; the element corresponding to the key "QC" will be the collection of all the records for the province of Quebec; and so on
- 3.6.5. for another example, in the *year map*:
  - (a) there will be two elements in the keys collection: one element will be the integer 2011, and the other will be 2016
  - (b) there will also be two elements in the values collection: one element will be the collection of all the records for 2011, and the other element will be the collection of all the 2016 records
- 3.6.6. it's important to note, once more, that all the elements in the values collection will be collections of pointers to existing records in the primary data collection; no copies of record objects will be made anywhere in this program

The `Map` class will contain the following member functions:

- 3.6.7. an `add(T key, Record* rec)` member function that adds the given record to the collection corresponding to the given key; if the given `key` is already in the keys collection, the `rec` parameter is added to the element of the values collection that corresponds to that key; if the `key` is not in the keys collection, it is added to the back of the keys collection, and the `rec` parameter is added to a new collection that is added to the back of the values collection
- 3.6.8. a getter function for the size of the keys collection

- 3.6.9. a getter function for the keys collection, which is returned using a reference
- 3.6.10. an overloaded subscript operator that takes a key type `<T>` as parameter, searches the keys collection to find the key matching the parameter, and returns a reference to the element of the values collection corresponding to that key
  - (a) for example, all the records for the province of New Brunswick will be returned if a calling function uses the expression `regionMap["NB"]`, and all records from the year 2011 will be returned by `yearMap[2011]`
  - (b) your code **must** use this operator to access the records when computing the report results
  - (c) you will use exception handling to deal with the case where the given key is not found

### 3.7. Report data class template requirements

The `ReportData` class template will contain the results for one report, organized as an ordered collection of keys and formatted strings that each represent one row in the report. Each report generator concrete object will contain a `ReportData` object, populate it with the formatted results of the report, and print the `ReportData` object to the screen.

All reports, including the ones for this project and all future reports, must be able to use this class. Therefore, the class must be generic enough to support multiple ways of ordering the report rows. This will be accomplished with a simple implementation of the Strategy design pattern, with comparison behaviour classes used for keeping the report rows in the correct order.

The `ReportData` class template will be parameterized for type `<T>`, which will be the data type of the keys used to order the report rows.

You will begin by implementing the `ReportRow` class template, as a *nested class* inside the `ReportData` class. We did an example of a nested class in the coding examples, specifically in the linked list class.

The `ReportRow` class will contain the following:

- 3.7.1. a data member that represents the key for the report row, stored as a data member of type `<T>`
  - (a) for example, the report described in 2.1 must be sorted in descending order by regional percentage over all animal types; since the value by which the report rows will be ordered is the regional percentage, that number will be the key for each corresponding report row; for report 2.1, the key will be a `float`, since that's how this program stores and prints out percentages
- 3.7.2. a data member for the report row, stored as a string
  - (a) for example, if report 2.1 indicates that Saskatchewan hosted 14.1% of all animals, the corresponding `ReportRow` object would have a key of 14.1, and a report row containing the formatted results for Saskatchewan: SK 15.0 7.5 14.6 8.2 31.0 31.8 13.7 0.0 0.0 14.1
  - (b) the report row will be a long string, already formatted by the report generator object with the correct spacing, ready to be printed out
- 3.7.3. a default constructor that takes a key and a row string as parameters, and initializes the correct data members
  - (a) for simplicity, and *for this class and this constructor only*, you may place the constructor implementation inside the class definition

**NOTE:** You will define this class exactly as the `Node` class in the linked list coding example. The `ReportRow` class will be defined in the **private** area of the `ReportData` class, and the `ReportRow` data members will be declared as public, so that `ReportData` can access them.

The `ReportData` class will contain the following data members:

- 3.7.4. a collection of report rows, stored as a STL vector of `ReportRow` objects, parameterized for the same data type `<T>` as `ReportData`; the elements of this collection will be in the correct order at all times
- 3.7.5. a pointer to the behaviour class object that will be used to compare the key of a new report row with the keys of the existing rows, in order to insert the new row into its correct position in the report row collection; this data member will be stored as a `CompareBehaviour` pointer (defined in a later step)



The `ReportData` class will contain the following member functions:

- 3.7.6. a default constructor that takes a `CompareBehaviour` pointer and initializes the corresponding data member
- 3.7.7. a destructor, as required
- 3.7.8. an `add(T key, string row)` member function that does the following:
  - (a) traverse the collection of report rows, using the comparison behaviour object to compare each row's key to the parameter `key`, to find the insertion point of the new report row contained in the parameter `row`
    - (i) your code **must** use a STL iterator that starts at the beginning of the report row collection, and explicitly advances through the collection; do NOT use a for-each loop, or a range-based loop, and do NOT use the subscript operator instead
  - (b) once the insertion point is found, use the STL vector's `insert()` member function to insert, into the correct position of the report row collection, a new `ReportRow` object containing the key and row indicated in the parameters `key` and `row`
- 3.7.9. an overloaded stream insertion operator that prints out to the output stream all the rows of the report (not the keys, as these are already formatted into the report row)  
**NOTE:** Your report generator concrete objects must use this operator once to print the report data to the screen, and a second time to print it to the text file.

### 3.8. Comparison behaviour classes requirements

Your program will implement an inheritance hierarchy of comparison behaviour class templates, which must include polymorphic behaviour.

There will be an abstract `CompareBehaviour` class that contains a pure virtual `bool compare(T, T)` member function. You will also implement two concrete derived classes that provide an implementation of this comparison function. In the `AscBehaviour` class, the comparison member function will compare the two parameters for ascending (increasing) order, and in the `DescBehaviour` class, it will compare them for descending (decreasing) order.

Your program **must** create these concrete objects where required, and it must use them to keep the rows in the `ReportData` objects in the order prescribed for the corresponding report.

## 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), except where explicitly permitted in the instructions, and only exactly as explicitly permitted.
- 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 `structs`. 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) the posted data file
  - (c) a Makefile
  - (d) a README file that includes:
    - (i) a preamble (program author, purpose, list of source and header files)
    - (ii) compilation and launching instructions

**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. Evaluation

6.1. **Marking components:**

- 10  UML class diagram
- 10  Report generator abstract class
- 50  Report generator concrete classes
- 10  Map class
- 20  Report data and comparison behaviour classes

6.2. **Execution requirements:**

- 6.2.1. all marking components must be called and execute successfully in order to earn marks
- 6.2.2. all data handled must be printed to the screen for marking components to earn marks

6.3. **Deductions:**

- 6.3.1. Packaging errors: There will be deductions for missing packaging files (including the Makefile, the README, and/or the data file), for additional files and/or directories submitted, for missing documentation, and for the incorrect separation and bundling of code into files.
- 6.3.2. Major design and programming errors: There will be deductions for poor design choices, for poor use of programming conventions and/or bad style, for memory leaks and/or valgrind errors, for insufficient or incorrect distribution of functionality across objects of the appropriate design categories and into modular member functions, for components that violate the Constraints listed and/or the principle of least privilege, or for components that use prohibited functions, classes, or algorithms.
- 6.3.3. Execution errors: Components that cannot be tested because they don't compile or don't execute in the provided course VM, or that are not used in the code, or where data is not printed to the screen, will earn zero.