Tommy Hollenberg
CS162 Assign1
Design Document
04/8/2017

Understanding the Problem

The program needs to read in state and county data from a file, calculate and output the following information about the data (either into a file or to the screen, specified by the user):

a) The state with the largest population
b) The county with the largest population
c) The counties with an income above a user-specified amount
d) The average house price for all counties in each state
    - Question: does this mean print avg_house for each county in each state? Or does it mean calculate the population-weighted average house price among all counties for each state?
e) The states sorted alphabetically by name
f) The states sorted by population
g) The counties within states sorted by population
h) The counties within states sorted alphabetically by name

The program will accept metadata as command-line arguments including the number of states to be read and the name of the file to read from. These pieces of information can be entered in any order, but must be preceded by '-s' and '-f' tags, respectively.

The data in the file will be structured with, for each state, a line containing information about the state followed by lines containing information about each of the counties in that state. State lines and county lines will contain the following information:

```
State_name state_population number_of_counties
County_name county_population county_income county_house_price
    number_of_cities city1_name city2_name ... cityN_name
```

The state and county information should be stored in the following structs:

```
struct county {
    string name; //name of county
    string *city; //name of cities in county
    int cities; //number of cities in county
    int population; //total population of county
    float avg_income; //avg household income
    float avg_house; //avg house price
};
```

```
struct state {
    string name; //name of state
    struct county *c; //name of counties
    int counties; //number of counties in state
    int population; //total population of state
};
```

The program must utilize functions exactly matching the following function prototypes:

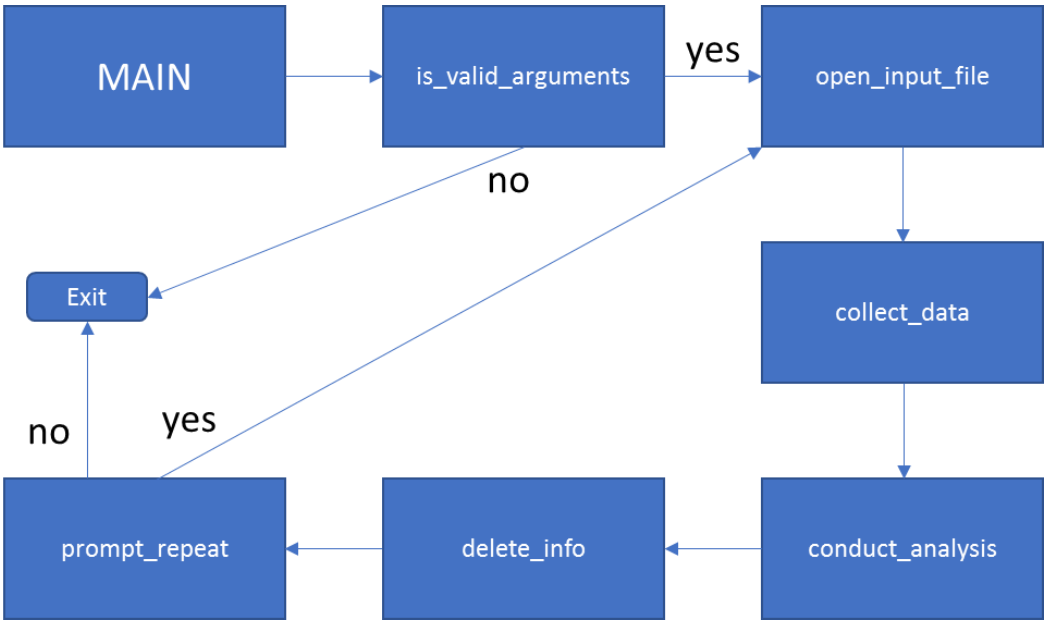
```
bool is_valid_arguments(char*[], int);
state* create_states(int);
void get_state_data(state*, int, ifstream&);
county* create_counties(int);
void get_county_data(county*, int, ifstream&);
void delete_info(state**, int);
```

The struct definitions and function declarations should be stored in a header file called state_facts.h, the function definitions should be stored in a source file called state_facts.cpp, and the main function should be stored in a source file called run_facts.cpp. A Makefile should be used to handle compilation.

The program must check to ensure the supplied command-line arguments are valid, but can simply output an error message and terminate if the number of arguments is incorrect or incorrect options (other than -s and -f) are supplied. However, if the number of states given is not a positive, non-zero integer, the given filename is invalid, or both, the program must recover by re-prompting the user until valid number of states and filename values are given.

Finally, after the report is printed (whether to a file or to the screen), the program must ask the user if they would like to run another report. If so, the program must prompt the user for a new number of states and a new filename, and perform the same operations on the new data.

Devising a Plan




```
bool is_valid_arguments(char *argv[], int argc) {
    if argc isn't 5, or if argv[1] and argv[3] aren't "-s" and "-f",
      or "-f" and "-s", respectively
        Output "Invalid command-line arguments."
        return false
    }

    if argv[1] is "-f" {
        swap argv[1] and argv[3]
        swap argv[2] and argv[4]
    }

    while(argv[2] isn't a nonzero positive integer) {
        Output "Enter a nonzero, positive integer for the number of
          states"
        Read in argv[2]
    }
    return true
}

state* create_states(int n) {
    return address of dynamically allocated length n array of states
}

void get_state_data(state *s, int state_num, ifstream &input) {
    Read in the state name, population, and number of counties from
      the file.
}
```

```
county* create_counties(int n) {
    return address of dynamically allocated length n array of states
}

void get_county_data(county *c, int county_num, ifstream &input) {
    Read in county name, population, average income, average house
      price, and number of cities from the input file.
    Dynamically allocate string array of size [number of cities] and
      assign address to the city member of the county pointed to by c.
    For 0 to number of cities - 1
      Read in city[i] from the file.
}

void delete_info(state **data, int num_states) {
    For each state {
        For each county in that state {
            delete the array of cities
        }
        delete the array of counties
    }
    delete the array of states
    set the pointer in the caller pointed to by data to 0
}

bool is_nonzero_pos_integer(const char *num) {
    nonzero = false
    for each character in num {
        if the character is '0' to '9' inclusive {
            if the character isn't '0'
                nonzero = true
        }
        else return false
    }
    return nonzero
}

void open_input_file(string filename, ifstream &input) {
    Attempt to open file name filename
    While the file isn't opened {
        Output "Error opening file.\nEnter filename: "
        Read in filename from user
        Attempt to open file name filename
    }
}
```

```
void collect_data(state **data, int num_states, ifstream &input) {
    Assign the pointer pointed to by data the return value of a call
      to create_states, passed num_states as an argument
    For each state {
        Call get_state_data, passing in the state array, the current
          state number, and the input filestream object
        Assign the c member of the current state the return value of a
          call to create_counties, passed the counties member of the
          current state as an argument
        For each county {
            Call get_county_data, passing in the c member of the
              current state, the current county number, and the input
              filestream object
        }
    }
}

void prompt_repeat(bool &again, int &num_states, string &filename) {
    Output "Enter any nonzero character to analyze a new data set, 0
      to exit: "
    Read in again from the user
    if again isn't 0 {
        output "Enter the number of states and new filename: "
        Read in num_states and filename from the user
    }
}

void conduct_analysis(state *data, int num_states) {
    Declare double min_income and string filename
    Output "Enter minimum average county income to display: "
    Read in min_income from the user
    Output "Enter name of output file (0 to print to screen): "
    Read in filename from the user

    if filename[0] isn't '0' or filename.length() isn't 1 {
        Declare an output filestream object and open a file named
          [filename]
        Call calc_and_output, passing data, num_states, min_income,
          and the output filestream object
    }
    else call calc_and_output, passing data, num_states, min_income,
      and the cout object
}
```

```
calc_and_output(state *data, int num_states, double min_income,
  ostream &out) {
    Declare pointer to state pointer, s.
    Call set_pointers, passing the address of s, data, and num_states.
    Call output_pop, passing s, num_states, and out.
    Call output_alpha, passing s, num_states, and out.
    Call above_min_income, passing s, num_states, min_income, and out,

    delete s
}

void output_pop(state **s, int num_states, ostream &out) {
    Declare pointer to county pointer, c.
    Declare string, biggest.
    Declare int, big_pop = 0.

    Call iterativeMergeSort, passing s, 0, num_states - 1, and the
      sort_by_pop function.
    Use out to print "\nStates and their Counties ranked by
      population:\n"
    For each state {
        Use out to print the state name.
        Call set_pointers, passing the address of c, the c member of
          the current state, and the counties member of the state.
        Call iterativeMergeSort, passing c, 0, the counties member of
          the current state - 1, and the sort_by_pop function.
        if c[0]->population > big_pop {
            biggest = c[0]->name
            big_pop = c[0]->population
        }
        For each county in the state
            Use out to print the county name
        delete c
    }
    Use out to print "\nLargest state by population: " s[0]->name and
      "\nLargest county by population: " biggest.
}
```

```
void output_alpha(state **s, int num_states, ostream &out) {
    Declare pointer to county pointer, c.
    Call iterativeMergeSort, passing s, 0, num_states – 1, and the
      sort_alphabetic function.
    Use out to print "\nStates and their Counties ranked
      alphabetically:\n"
    For each state {
        Use out to print the state name.
        Call set_pointers, passing the address of c, the c member of
          the current state, and the counties member of the state.
        Call iterativeMergeSort, passing c, 0, the counties member of
          the current state – 1, and the sort_alphabetic function.
        For each county in the state
            Use out to print the county name
        delete c
    }
}

void above_min_income(state **s, int num_states, double min_income,
  ostream &out) {
    Use out to print "\nCounties with average income above "
      min_income ":\n"
    for each state
        for each county in that state
            if the avg_income member is greater than min_income
                Use out to print the county name.
}
```

The remaining functions are templated functions, which means that the compiler will automatically create a function overload for any type that they are called with. Since both state and county structs have member variables name and population, they behave identically for sorting purposes. The use of templated functions saves the time of having to write out two copies of each function, only swapping out all instances of the word "state" with "county."

The set_pointers function creates an array of pointers, and assigns them to each element of a state of county array. This allows addresses to be repeatedly swapped during sorting rather than entire structs.

```
template<typename T>
void set_pointers(T ***p, T *data, int n) {
    Dynamically allocate a length n array of T pointers, and assign
      returned address to the pointer in the caller pointed to by p.
    For each element of *p
        Assign it the address of the corresponding element of data.
}
```

The next two functions perform a merge sort on an array. The logic will not be shown due to time constraints for the submission of this design, but the code has already been written and tested.

```
template<typename T>
void merge(T *A, int l, int m, int r, bool (*comparisonFcn)(T, T)) {
    This function performs the merge operation of the merge sort. The
      subarrays A[l, m] and A[m+1, r] are guaranteed to be sorted
      already. These subarrays are copied into two temp arrays, L and
      R. The first element of L is compared to the first element of R,
      and the one that should come first is copied into A[l]. This
      process is continued, like merging together two sorted stacks of
      in such a way that the resulting stack remains sorted, until one
      stack is exhausted. At this point, the remaining elements of the
      other stack are copied into the remaining elements of A[l, r]

    The comparisonFcn parameter can be passed a pointer to different
      functions to change what characteristic the array will be sorted
      by (for example, by population or alphabetic order).
}

template<typename T>
void iterativeMergeSort(T *A, int l, int r, bool
  (*comparisonFcn)(T, T)) {
    This function performs an iterative implementation of a merge
      sort, calling the merge function, first with subarrays of only
      one element each, and doubling the size after each pass-through,
      until the entire array is sorted.
}
```

These last two functions provide two different comparisons for sorting. The first orders states and counties with higher populations ahead of those with lower populations. In the case of a tie, it places states and counties in alphabetical order, by calling the second function. As expected, the second function orders states and counties by alphabetical order. If the two names are identical up to the end of the shorter of the two, the state or county with the shorter name is placed ahead of the state or county with the longer name.

```
template<typename T>
bool sort_by_pop(T *a, T *b) {
    if a->population > b->population
        return true
    if a->population < b->population
        return false
    return sort_alphabetic(a, b)
}
```

```
template<typename T>
bool sort_alphabetic(T *a, T *b) {
    for each character of the shorter name {
        if a->name[i] < b->name[i]
            return true
        if a->name[i] > b->name[i]
            return false
    }
    return a->name.length() < b->name.length()
}
```

Testing Plan

Command-line argument checking:

Input	Result
state_facts -s 3 -f file.txt	Valid input
state_facts -f file.txt -s 3	Valid input
state_facts -s 1 -f file.txt	Valid input
state_facts -s 0 -f file.txt	"Enter a nonzero, positive integer for the number of states in the file:"
state_facts -s a -f file.txt	"Enter a nonzero, positive integer for the number of states in the file:"
state_facts -s 3 -g file.txt	"Invalid input", terminates
state_facts -s 3 -f file.txt e	"Invalid input", terminates
state_facts -s 3 -f (non-existent file)	"Error opening file.\nEnter filename:"
state_facts -s -f 3 file.txt	"Invalid input", terminates

This program is not expected to handle errors from the file input, so all file input is assumed to be valid, correctly structured, and error free.

Others things to test for:

- The states and counties are sorted correctly, both alphabetically by name and by population.
- The largest state and county by population are correctly identified.
- Only counties above the specified income level are printed.
- The output is correctly printed to the screen or to a file, based on user specification.
- The program loops correctly and allows multiple reports to be run without any memory leaks.