

Project 3 – Animated Bar Chart

CS 251, Fall 2021

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff. Also, please see copyright at the bottom of this description. Do not post your work for this class publicly, even after the course is over.

Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly.

Early Bonus: If you submit a finished project early (by Wednesday, September 29th at 11:59 pm), you can receive 10% extra credit. In order to receive the early bonus: (1) your submission needs to pass 100% of the tests cases; (2) you may not have any submissions after the early bonus deadline.

Test cases/Submission Limit: Limited submissions (15 submissions max). Most test cases are hidden.

What to submit: (1) bar.h, (2) barchart.h, (3) barchartanimate.h, (4) tests.cpp, (5) application.cpp

IDE: Codio is our new course IDE. It is required that you use it and you may not use repl.it (or any other IDE) anymore. Follow [these instructions](#) to get your account set up in Codio. Once you get logged in, you will be able to see the Project 3 starter code. We have paid for Codio, therefore, we are confident that this will be a dependable option for us. It has a lot of unique features, including allowing instructors to log into your project code at anytime to provide better help. Moving forward, we are using a lot of tools not available on repl.it or Mac/PC local IDEs, so please get comfortable on Codio ASAP. This project uses valgrind and linux terminal commands that will not work elsewhere.

[.pdf starterCode](#)

Project Summary

Animated bar charts spread virally over social media in 2019 because they are a surprisingly simple, yet powerful, way to tell a story about categorical data over time. A quick search of “Bar Graph Racer” or “Animated Bar Graphs” will tell you all about it. Here is one:

The most populous cities in the world from 1500 to 2018

Population (thousands)

City	Region	Population (thousands)
Tokyo	Asia	22,092
New York	North America	16,422
Osaka	Asia	14,389
Paris	Europe	8,467
Los Angeles	North America	8,190
Mexico City	Latin America	8,188
Buenos Aires	Latin America	8,102
London	Europe	7,827
Shanghai	Asia	7,311
Chicago	North America	7,207

1968

by @pratapvardhan; credit @iburnmurdoch

Source: <https://towardsdatascience.com/bar-chart-race-in-python-with-matplotlib-8e687a5c8a41>

Unfortunately, C++ graphics are not super compatible with Linux (virtual machines). And C++ are not easily transportable across different operating systems. Therefore, we will not be using a graphics in this class (much to my own disappointment and countless hours trying to find ways to make it work in various projects). Therefore, we will be creating a text-based animated bar chart. Mimir:

```
U[snanon] >> make run  
./application.exe  
The most populous cities in the world from 1500 to 2018  
Sources: SEDAC; United Nations; Demographia
```

Beijing	672
Vijayanagar	500
Cairo	400
Hangzhou	250
Tabriz	250
Gauda	200
Istanbul	200
Paris	185
Guangzhou	150
Nanjing	147
Cuttack	140
Fez	130

Population (thousands)
Frame: 1500

Here is what it will look on Codio, in dark mode:

```

The most populous cities in the world from 1500 to 2018
Sources: SEDAC; United Nations; Demographia
##### Beijing 673
##### Vijayanagar 499
##### Cairo 398
##### Hangzhou 251
##### Tabriz 244
##### Istanbul 215
##### Gauda 200
##### Paris 187
##### Guangzhou 151
##### Nanjing 147
##### Cuttack 137
##### Fez 130
Population (thousands)
Frame: 1503

```

This handout will walk you through the project step-by-step. However, it is possible that you may be able to complete the project by just writing the .h files using the comments included. The colors you see in this example are on Mimir light mode. Other versions of Linux terminals will have different colors. See details below.

Data Files

There are a number of fascinating data files in the specified format, curated from various sources that we will use to make our Barchart animations. All of these are provided in the starter code.

input file	description	period	data source
cities.txt	<i>most populous cities in the world</i>	1500–2018	John Burn-Murdoch
countries.txt	<i>most populous countries in the world</i>	1950–2100	United Nations
cities-usa.txt	<i>most populous cities in the U.S.</i>	1790–2018	U.S. Census Bureau
brands.txt	<i>most valuable brands in the world</i>	2000–2018	Interbrand
movies.txt	<i>highest-grossing movies in the U.S.</i>	1982–2019	Box Office Mojo

baby-names.txt	<i>most popular baby names in the U.S.</i>	1880–2018	U.S. Social Security
football.txt	<i>the best football clubs in Europe</i>	1960–2019	clubelo.com
endgame.txt	<i>characters in Endgame by screen time</i>	Minute 1–170	Prashant
game-of-thrones.txt	<i>characters in Game of Thrones</i>	S01E01–S8E06	Jeffrey Lancaster

As a canonical example, consider an animated bar chart of the 10 most populous cities in the world, from 1500 to 2018 (shown above). To produce the visualization, you will successively “draw” 519 individual bar charts (one per year of data), with a short pause between each “drawing”. Each bar chart contains the n most populous cities in that year, arranged in descending order of population.

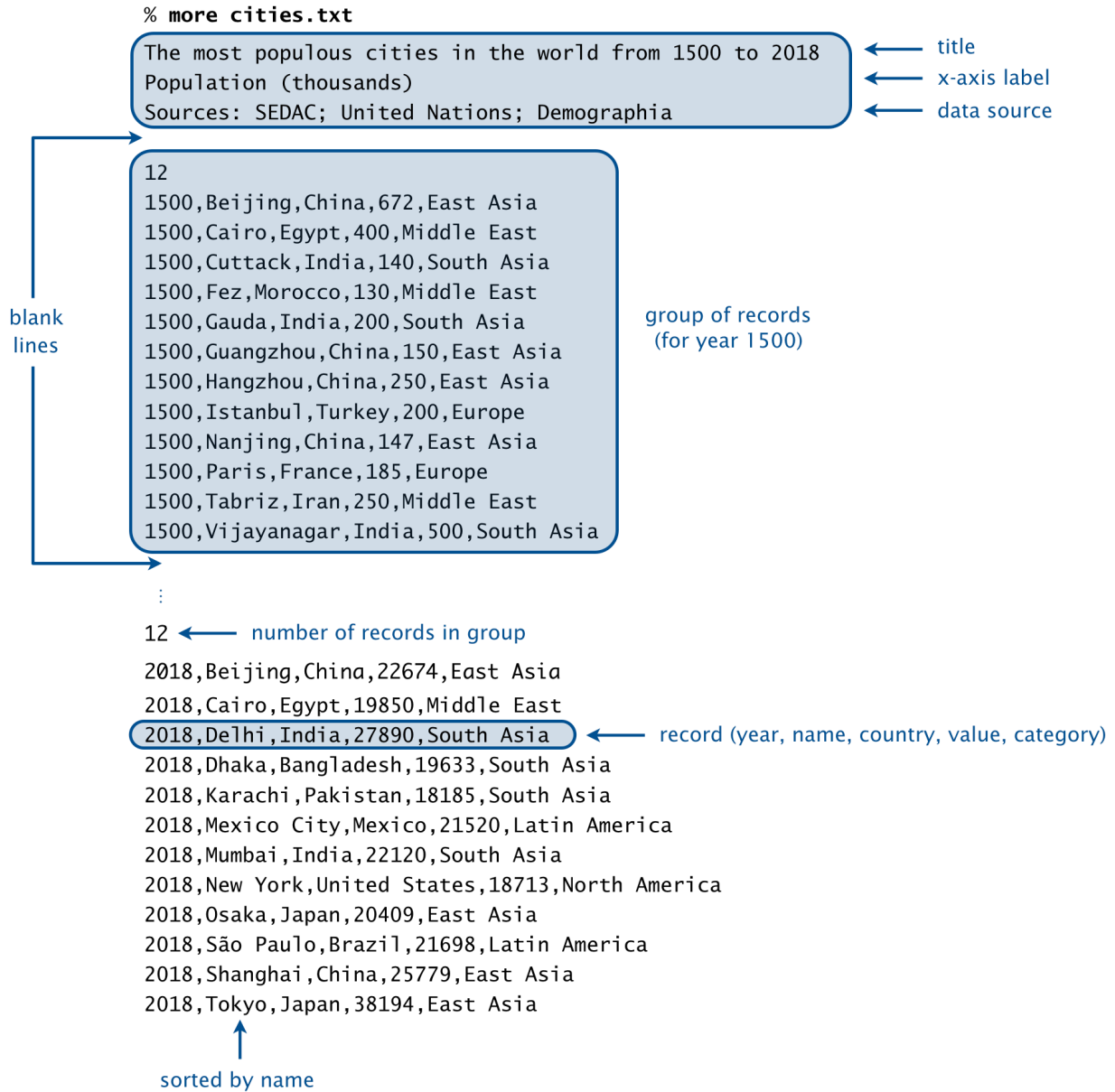


Figure 1 – File format.

File format. A bar chart animation data file is organized as a sequence of lines. The first three lines comprise the *header*:

- The *title*.
- The *x-axis label*.
- The *source* of the data.

Following the header is a blank line, followed by the raw data. Each line (or *record*) consists of 5 fields, separated by commas:

- The *year* or *date* (e.g, 2018).

- The *name* (e.g., Mumbai).
- The associated *country* (e.g., India).
- The *value* (e.g, 22120).
- The *category* (e.g., South Asia).

The value field is an integer; the other fields can be arbitrary strings (except that they can't contain commas or newlines).

Records corresponding to the same year (or time period) are grouped together. A *group* of records consists of an integer *n*, followed by *n* records. Within a group, the records are sorted by name. A blank line separates each group.

Milestone 1 - write `bar.h` and test

The first step is to create a class called *Bar*. A *Bar* aggregates related information (*name*, *value*, and *category*) for use in a bar chart. For example, the first bar drawn in the bar chart represents *name* = *Beijing*, *value* = 672, and *category* = *East Asia*.

Implement the following API:

```
class Bar {
public:
    // default constructor:
    Bar();

    // a second constructor:
    Bar(string name, int value, string category);

    // destructor:
    virtual ~Bar();

    string getName();
    int getValue();
    string getCategory();

    // operators
    bool operator<(const Bar &other);
    bool operator<=(const Bar &other);
    bool operator>(const Bar &other);
    bool operator>=(const Bar &other);

};
```

You will need to decide on and define private member variables for the *Bar* class. The two constructors, destructor, and getters are pretty self-explanatory. For the default constructor, use default values for each type "" and 0.

As you saw in the examples at the top, the bar graph will be plotted in sorted, descending order. Therefore, it will be necessary to sort Bars. In order to sort a list of Bar objects, we need to define how to compare two Bar objects. To do so, we will overload the operators $>$, $>=$, $<$, $<=$. We will use the Bar's value to implement these comparisons. Here is an example of the behavior:

```
Bar b1("a", 1, "cat");
Bar b2("b", 2, "cat");
(b1 < b2) should return true.  (b2 < b1) should return false.
(b1 > b2) should return false.  (b2 > b1) should return true.
```

If you check out the API above for operator $<$, you will see the function signature has an Bar object called other passed in as a parameter. Let's say we write $b1 < b2$. When writing these operators, the current object (this) is the left hand side of the operator (b1, in this example) and the other object is the right hand side of the operator (b2, in this example).

Ready to test?

For this project, your tests are going to be graded. We will learn a testing framework next week and use it for Project 4. Until then, we are going to write our tests in a normal .cpp file. Check out the starter code in **tests.cpp**. You can see two very basic tests written for the default constructor and parameterized constructor. Use this as a guide to write your own tests. You should have multiple tests written for each public member function in every class (as a minimum). The autograder has 100s, 1000s, 10s of thousands tests. Check out the makefile which has different rules for compiling and running the tests vs compiling and running the application. The tests can be run by type (make build_tests, make run_tests). See makefile for details.

Milestone 2 - write barchart.h and test

The next abstraction is a BarChart class. Here is the API:

```

class BarChart {
private:
    Bar* bars; // pointer to a C-style array
    int capacity;
    int size;

public:
    // default constructor:
    BarChart();

    // parameterized constructor:
    BarChart(int n);

    // copy constructor:
    BarChart(const BarChart& other);

    // copy operator=
    BarChart& operator=(const BarChart& other);

    // destructor
    virtual ~BarChart();

    void clear();
    void setFrame(string frame);
    string getFrame();
    bool addBar(string name, int value, string category);
    int getSize();
    Bar& operator[](int i);
    void dump(ostream &output);
    void graph(ostream &output, map<string, string> &colorMap, int top);
};

```

In this abstraction, you are going to allocate space on the heap. You will store one entire frame of the animation inside a C-array of Bar objects. Since you are using a C-array, you will need allocate space based on the n value passed in as a parameter in the parameterized constructor. Additionally, you will see keep track of how many Bars are added using size. This abstraction does not use a dynamic array. Therefore, you can only add n Bars to the BarChart. If a Bar is attempted to be added beyond the capacity, it will be ignored. You may need to add more private member variables. However, you may not use any additional/new C++ containers. This class needs more details, which will be provided below. It is recommended that you write one a time. Write lots of tests for each member function before moving on to the next one. It is recommended that you write the 2 constructors first. Then write each member function, in the order they are listed in the table. Then, write the copy construction, copy operator, and destructor last.

Member Function	How to Call	Details
// default constructor: BarChart();	BarChart bc;	Do not allocate memory for bars. Initialize all private member variables to default values (0 and "").

<code>// parameterized constructor: BarChart(int n);</code>	<code>BarChart bc(10);</code>	Allocate memory for 10 bars. Initialize capacity to 10 and all other private member variables to default values.
<code>// copy constructor: BarChart(const BarChart& other);</code>	<code>BarChart bc(10); BarChart bcCopy(bc);</code>	bcCopy is constructed (new object) and has all the same values of private member variables as bc. See header description and example in ourvector.h.
<code>// copy operator= BarChart& operator=(const BarChart& other</code>	<code>BarChart bc1; BarChart bc2(10); bc1 = bc2;</code>	This allows you set one object equal to a different object (but both objects are both already constructed). See header description and example in ourvector.h.
<code>void clear();</code>	<code>BarChart bc(10); bc.clear();</code>	Frees memory allocated on the heap and sets all private member variables to default values (0, "", nullptr). Called by the user.
<code>// destructor virtual ~BarChart();</code>		Frees memory allocated on the heap. Called automatically when the object goes out of scope.
<code>void setFrame(string frame);</code>	<code>BarChart bc(3); bc.setFrame("1950");</code>	A setter for keeping track of which frame the BarChart is associated with.
<code>string getFrame();</code>	<code>bc.getFrame();</code>	A getter for getting the frame associated with the BarChart object. Using above example, it would return "1950".
<code>bool addBar(string name, int value, string category);</code>	<code>bc.addBar("Chicago", 1020, "US"); bc.addBar("NYC", 1300, "US"); bc.addBar("Paris", 1200, "France");</code>	Adds a Bar to the BarChart. If added successfully, return true. If there the BarChart has reached capacity, do not add the Bar and return false.
<code>int getSize();</code>	<code>int n = bc.getSize();</code>	The line here would return size of 3 since 3 Bars have been added in above example.
<code>Bar& operator[](int i);</code>	<pre>for (int i = 0; i < n; i++) { cout << bc[i].getName() << " "; cout << bc[i].getValue() << " "; cout << bc[i].getCategory(); cout << endl; }</pre>	Using above bc, this would print: Chicago 1020 US NYC 1300 US Paris 1200 France Notice this is in insertion order and not sorted order.
<code>void dump(ostream &output);</code>	Option 1: <code>bc.dump(cout);</code> Option 2: <code>stringstream ss(""); bc.dump(ss); cout << ss.str();</code>	This function is useful for debugging (but is also tested). It prints out the BarChart in text in text using the following format (note it is printed in descending order by value): frame: 1950 NYC 1300 US Paris 1200 France Chicago 1020 US This is what is printed for both option 1 and 2. Please note you should not use any color in the dump function,

		including BLACK (as in, do not do output << BLACK << etc.).
<pre>void graph(ostream &output, map<string, string> &colorMap, int top);</pre>	<pre>string red("\033[1;36m"); string blue("\033[1;33m"); map<string, string> colorMap; colorMap["US"] = red; colorMap["France"] = blue; bc.graph(cout, colorMap, 3);</pre>	<p>This will produce the visualization of the BarChart. colorMap determines how to color each category and top determines how many bars are printed on each frame. See Figure 2 for what is printed for this example. If no color map is provided, it should be printed using the black color. See section titled "Graph Details" for more information on how to translate values into bars. Note the bars are printed in descending order by value. If the colorMap is empty or the category is not found in the colorMap, the bar should be printed with BLACK.</p>

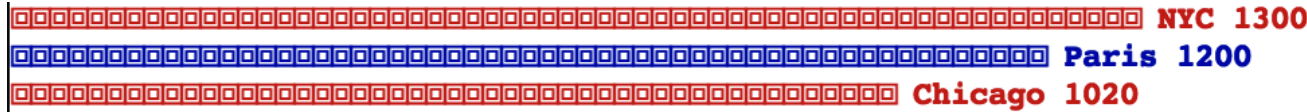


Figure 2 - output of graph for example in table.

Graph Details

Some math is needed for printing each Bar in the BarChart inside the graph function. This section will detail how that should be done.

Top Bar: The bars are printed in sorted, descending order. The top bar (which is the one with the max value), should always be printed with lenMax (set to 60 in the starter code) BOX characters. BOX is a constant defined at the stop, which is the square character you are seeing in the image above. The starter code gives an example of how to do this.

The rest of the Bars: are printed proportionally to the top bar. Here is how you will do that calculation. Let's take Paris as an example. The value is 1200. NYC has the max value. So, we will calculate the percentage Paris's value is with respect to the max value:

$$1200/1300 \approx 0.9231$$

We are then going to multiple that percentage by lenMax (which is 60):

$$(1200/1300) * 60 \approx 0.9231 * 60 \approx 55.385$$

Then, we will take the floor of this calculation or cast to an int, to get 55. As you can see Paris's bar is printed with 55 BOXes and that is how that calculation is determined for each bar (always using the top/max Bar as the denominator of the percentage).

A note on colors: The colors, box character, and clear console character are defined as constants at the top of barchart.h. You will primarily use them in barchartanimate.h (which has access to what is

defined in barchart.h). They are defined in barchart.h so they can be all in one place and some of them will be needed in barchart.h (including BOX and BLACK).

Testing?

Test each member function, as you write it. You should have multiple tests written for each member function. Your **tests.cpp** file will be graded.

Milestone 3 - write barchartanimate.h and test

Finally, you are going to write the last abstraction to complete the Bar Chart Animation functionality. Here is the API:

```
class BarChartAnimate {
private:
    BarChart* barcharts; // pointer to a C-style array
    int size;
    int capacity;
    map<string, string> colorMap;

public:
    // a parameterized constructor:
    BarChartAnimate(string title, string xlabel, string source);
    // destructor:
    virtual ~BarChartAnimate();

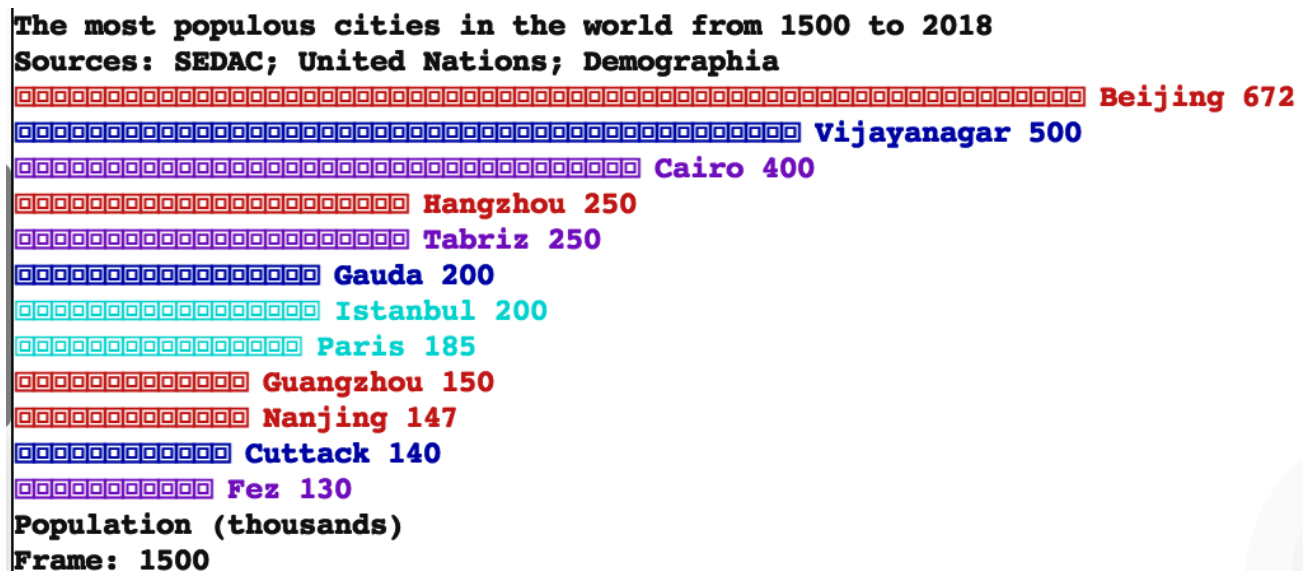
    void addFrame(istream &file);
    void animate(ostream &output, int top, int endIter);
    int getSize();
    BarChart& operator[](int i);
};
```

Member Function	How to Call	Details
// a parameterized constructor: BarChartAnimate(string title, string xlabel, string source);	See application in next step BarChartAnimate bca(title, xlabel, source);	Should allocate memory on heap for barcharts. This will be a dynamic array that expands (by doubling itself) each time it runs out of room to add a new frame. In the constructor, the capacity should be 4. Everything else should be initialized to default values.
virtual ~BarChartAnimate();		Frees memory allocated on the heap. Called automatically when the object goes out of scope.
void addFrame(istream &file);	See application in next step. bca.addFrame(inFile);	The file stream passed in guaranteed to be the format provided above. This explanation is referring to Figure 1 (the file format). Pre condition: when addFrame is called, it assumes that the fstream's first getline

		call will get the empty line above the “group of records” blue box. Post condition: The function should read the file to the end of the last record in the group of records (12 records total, in the example given). The function should build a BarChart using this group of records and add it to the BarChartAnimate object.
<code>void animate(ostream &output, int top, int endIter);</code>	<code>bca.animate(cout, 12, 10);</code> <code>bca.animate(cout, 12, -1);</code>	Animate should produce the final Bar Chart Animation. A few important details are described below in “Animation Details” as well as an image. The parameter top determines the number of the top bars are graphed. The parameter endIter determines how many iterations of the animation are graphed. E.g. the first call will animate the top 12 bars for 10 iterations. The second call will animate the 10 bars for all iterations in the data file.
<code>int getSize();</code>	<code>bca.getSize();</code>	Returns the number of frames added to the BarChartAnimate object.
<code>BarChart& operator[](int i);</code>	<code>for (int i = 0; i < n; i++) { cout << bca[i].getFrame(); cout << endl; }</code>	The [] operator is helpful for debugging and also allows us to grade your abstraction more thoroughly.

Animation Details

See this image for an example of what each frame of your animation should look like in Mimir light mode:



Or in Codio dark mode:

You should do this assignment by filling the color map while you build the BarChartAnimate.

NOTES:

- The colors provided are specific to the Mimir terminal and specific to the “light” mode. If you are in dark mode, your colors will look different. But do not change them as it will make it so you cannot pass the autograder.
- You will have to output CLEARCONSOLE defined in barchart.h to facilitate the animation, your final animation should end with the last frame still showing, so make sure you have this in the correct place. See function header comments for more details.

Milestone 4 – run the application!

The application.cpp file contains the code to run your application. If you built your three .h files according to the specs above, all you need to do is run the application (make build; make run). There are so many data files to try out. So, enjoy!

```
string filename = "cities.txt";
ifstream inFile(filename);
string title;
getline(inFile, title);
string xlabel;
getline(inFile, xlabel);
string source;
getline(inFile, source);

BarChartAnimate bca(title, xlabel, source);

while (!inFile.eof()) {
    bca.addFrame(inFile);
}

bca.animate(cout, 12);
```

Milestone 5 – Creative component

Now that you have finished the final layer of abstraction and seen it in action, it's your turn to use the abstraction! Your job is to create a console user interface for BarChartAnimate, the exact functionality you implement is up to you, but to get started you should modify application.cpp to input the filename used for the animation.

Here are some ideas, but of course you are not limited to these:

- An interface to add/edit frames
- The ability to select a timespan to play
- Change options like playback rate, color scheme

If you would like to add a parameter to a function you already made, make sure you *overload* the function instead of simply adding the parameter. This means creating a function with the same name that has that additional parameter; you must do this in order to still pass the autograder. For instance, if you wanted to add an integer parameter to `animate()`, you would make a *new* function:

```
void animate(ostream &output, int top, int endIter) {  
    /* ... your original animate ... */  
}  
  
void animate(ostream &output, int top, int endIter, int new_parameter) {  
    /* ... your new animate which utilizes new_parameter ... */  
}
```

In order to receive credit for the creative component – you must include instructions on how to run the creative component in the header comment of `application.cpp` AND include a description of what you added.

Requirements

1. You may not change the API provided. You must keep all private and public member functions and variables as they are given to you. If you make changes to these, your code will not pass the test cases. The test cases, from now on, run unit tests on your member functions. However, you may add helper functions and variables.
2. No additional C++ containers are allowed on this assignment. No additional header files are allowed to be included.
3. The implementations must use a C-array, no other data structures are allowed.
4. All sorting must be $O(n \log n)$. You should use built in sorting methods provided in the libraries (look up `algorithm`, `bits/stdc++`, etc.). I hope you never have to write a sorting algorithm again. ☺ In order to pass the autograder, you do have to sort the same way the autograder is sorting in order to handle ties the same. The autograder is calling `sort` and using `greater<>()` as the third parameter, [see here for details](#). Do not use `stable_sort`. Do not sort in ascending order and reverse.
5. You must have a clean `valgrind` report when your code is run. Check out the makefile to run `valgrind` on your code. All memory allocated (call `new`) must be freed (call `delete`). Check out Lecture 037 for an example of clean `valgrind` report.
6. As always, the file may be read exactly once.
7. You need to put effort into testing by writing a “`tests.cpp`” file and submit that for evaluation along with your `.h` files. Check out the Mimir rubric for how this will be graded. You will only get credit if every member function is tested multiple times.
8. Feel free to define additional variables or functions. However, define them as private member variables and functions. No global or static variables.
9. Since you are writing an abstraction, you will notice there are not as many decisions to be made with respect to efficiency (compared to previous projects). However, as always, your code should be writing as efficiently as possible and you will receive penalties to your score if your code is not efficient.

10. Your code should have comments throughout. The starter code includes some comments on functions, which should be replaced or added to with your own comments that full describe the functions. You should also have inline comments on lines that are not self-explanatory and you should have a header comment at the top of the file to describe the class.

Valgrind

Your code must be free of memory errors and memory leaks. To test this, you will need to run valgrind. If your tests are not thorough enough, you might be leaking memory on something that isn't being tested. To run valgrind on your application, type make build, make valgrind. Here is an example of a clean valgrind report (you should have more allocs and frees, this is just an example):

```
==90== Memcheck, a memory error detector
==90== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==90== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==90== Command: ./program.exe
==90==
==90==
==90== HEAP SUMMARY:
==90==    in use at exit: 0 bytes in 0 blocks
==90==   total heap usage: 2 allocs, 2 frees, 112,712 bytes allocated
==90==
==90== All heap blocks were freed -- no leaks are possible
==90==
==90== For counts of detected and suppressed errors, rerun with: -v
==90== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Test passed! No memory leaks detected.
```

You will notice there are no memory leaks and no memory errors.

Citations/Resources

This assignment was inspired by Dr. Kevin Wayne from Princeton University. His version was originally inspired by tweets from [Matt Navarra](#) and [John Burn-Murdoch](#). Thanks to Kai Bonsol for his contributions to improving the autograder.

Copyright 2021 Shanon Reckinger.

This assignment description is protected by [U.S. copyright law](#). Reproduction and distribution of this work, including posting or sharing through any medium, such as to websites like [chegg.com](#) is explicitly prohibited by law and also violates [UIC's Student Disciplinary Policy](#) (A2-c. Unauthorized Collaboration; and A2-e3. Participation in Academically Dishonest Activities: Material Distribution).

Material posted on [any third party](#) sites in violation of this copyright and the website terms will be removed. Your user information will be released to the author.