**Workshop 1: Move and Copy Semantics**

In this workshop, you work with a large dynamically allocated array of C++ Standard Library strings and compare the performance of copy and move operations on that collection.

**Learning Outcomes**

Upon successful completion of this workshop, you will have demonstrated the abilities to:

* retrieve records from a text file using an input file stream object
* count the number of records in a text file
* monitor the time spent on a particular task using the std::chrono library
* implement **copy semantics** for a class with a resource
* implement **move semantics** for a class with a resource
* identify the processing-intensive operations in copy and move assignments

**Submission Policy**

The *in-lab* section is to be completed during your assigned lab section. It is to be completed and submitted by the end of the workshop period. If you attend the lab period and cannot complete the *in-lab* portion of the workshop during that period, ask your instructor for permission to complete the *in-lab* portion after the period. If you do not attend the workshop, you can submit the *in-lab* section along with your *at-home* section (see penalties below). **In order to get credit for the *in-lab* portion, you must be present in the lab.**

The *at-home* portion of the workshop is due on the day that is four days after your scheduled *in-lab* workshop (@ 23:59:59), **even if that day is a holiday**.

All your work (all the files you create or modify) must contain your name, Seneca email, student number and the date of completion (use the following template):

// Name:

// Seneca Student ID:

// Seneca email:

// Date of completion:

//

// I confirm that the content of this file is created by me,

// with the exception of the parts provided to me by my professor.

You are responsible to back up your work regularly.

**Late Submission Penalties**

The workshop can be usbmitted up to **1 (one) day** late (the day that is 5 days after the lab period); submissions received on this day are considered **late** are subject to penalties:

* only *in-lab* portion submitted late (after the end of the lab period): 0 for *in-lab* portion, max 7/10 for the entire workshop.
* only *at-home* portion submited late (more than 4 days after the lab period): 4 for the *at-home* portion, max 7/10 for the entire workshop.
* both *in-lab* **and** *at-home* portions submitted late: max 4/10 for the entire workshop.
* when the submission closes, if the workshop is not complete, the mark for the entire workshop will be 0/10. The workshop is considered complete if there are two separate submissions (*in-lab* submission and *at-home* submission) containing the *in-lab code*, *at-home code* and *reflection*.

The submission is considered closed at the end of the day that is 5 (five) days after the lab period.

***In-Lab***

This workshop consists of three modules:

* w2 (supplied)
* TimedEvents
* Text

Enclose all your source code within the sdds namespace and include the necessary guards in each header file.

**w2 Module (supplied)**

**Do not modify this module!** Look at the code and make sure you understand it.

**TimedEvents Module**

Design and code a class named TimedEvents that manages a **statically** allocated array of record objects. Your class predefines the maximum number of record objects at 7. The **instance variables** for your class should include:

* the number of records currently stored
* the start time for the current event (an object of type std::chrono::steady\_clock::time\_point; see documentation [here](https://en.cppreference.com/w/cpp/chrono/time_point))
* the end time for the current event (an object of type std::chrono::steady\_clock::time\_point)
* an array of records of anonymous structure type (the structure has no name). The structure should contain the following fields:
  + a string with the event name.
  + a string with the predefined units of time
  + the duration of the recorded event (an object of type std::chrono::steady\_clock::duration; see documentation [here](https://en.cppreference.com/w/cpp/chrono/duration))

Your class includes the following member functions:

* a default constructor
* startClock(): a modifier that starts the timer for an event
* stopClock(): a modifier that stops the timer for an event
* recordEvent(): a modifier that receives the address of a C-style null terminated string that holds the name of the event. This function will update the next time-record in the array:
  + stores the parameter into the name attribute
  + stores "nanoseconds" as the units of time
  + calculates and stores the duration of the event (use std::chrono::duration\_cast<std::chrono::nanoseconds>(), see documentation [here](https://en.cppreference.com/w/cpp/chrono/duration/duration_cast))
* a **friend insertion operator** that receives a reference to an std::ostream object and a TimedEvents object. This operator should insert in the first parameter the records from the array in the following format:

Execution Times:

--------------------------

EVENT\_NAME DURATION UNITS

EVENT\_NAME DURATION UNITS

...

--------------------------

The **name** of the event should be a field of size 20, alligned on the left; the **duration** should be a field of size 12, alligned on the right.

Starting and stopping the timer means getting the current time (use std::chrono::steady\_clock::now(); see documentation [here](https://en.cppreference.com/w/cpp/chrono/steady_clock/now)).

**Text Module**

Design and code a class named Text that manages a **dynamically** allocated array of std::strings. Your class keeps track of the number of strings currently stored and defines the following member functions:

* a no-argument default constructor
* a 1-argument constructor that receives the address of a C-style null terminated string containing the name of a file from which this member function populates the current object. This function
  1. reads the file to count the number of records present (the record delimiter should be a single space ' ')
  2. allocates memory for that number records in the array
  3. re-reads the file and loads the records into the array.
* a copy constructor
* a copy assignment operator
* a destructor
* size\_t size() const: a query that returns the number of records stored in the current object.

To review the syntax for reading from a text file using an std::ifstream object see the chapter in your notes entitled [Custom File Operators](https://scs.senecac.on.ca/~BTP200/pages/content/files.html).

**Sample Output**

When the program is started with the command:

w2.exe gutenberg\_shakespeare

the output should look like:

Command Line:

--------------------------

1: w2.exe

2: gutenberg\_shakespeare

--------------------------

0-arg Constructor - a.size = 0 records

1-arg Constructor - b.size = 1293934 records

Copy Constructor - c.size = 1293934 records

Copy Assignment - a.size = 1293934 records

--------------------------

Execution Times:

--------------------------

0-arg Constructor 790 nanoseconds

1-arg Constructor 4377977955 nanoseconds

Copy Constructor 1976590065 nanoseconds

Copy Assignment 2004531426 nanoseconds

Destructor 3478640044 nanoseconds

--------------------------

**Note:** The execution times will be different every time you run the program! Everything else should match.

**Submission (30%)**

To test and demonstrate execution of your program use the same data as shown in the output example above.

Upload your source code to your matrix account. Compile and run your code using the latest version of the g++ compiler (available at /usr/local/gcc/9.1.0/bin/g++) and make sure that everything works properly.

Then, run the following command from your account (replace profname.proflastname with your professor’s Seneca userid):

~profname.proflastname/submit 305XXX\_w2\_lab

and follow the instructions. Replace XXX with the section letter(s) specified by your instructor.

**⚠️Important:** Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

***At-Home***

For this part of the workshop, upgrade the Text class to include a **move constructor** and a **move assignment operator**. No other modules need to be changed.

**Sample Output**

When the program is started with the command:

w2.exe gutenberg\_shakespeare

the output should look like:

Command Line:

--------------------------

1: w2.exe

2: gutenberg\_shakespeare

--------------------------

0-arg Constructor - a.size = 0 records

1-arg Constructor - b.size = 1293934 records

Copy Constructor - c.size = 1293934 records

Copy Assignment - a.size = 1293934 records

Move Constructor - d.size = 1293934 records

Move Assignment - a.size = 1293934 records

--------------------------

Execution Times:

--------------------------

0-arg Constructor 790 nanoseconds

1-arg Constructor 4010433846 nanoseconds

Copy Constructor 2002725409 nanoseconds

Copy Assignment 1926967415 nanoseconds

Move Constructor 790 nanoseconds

Move Assignment 394 nanoseconds

Destructor 3538222832 nanoseconds

--------------------------

**Note:** See that in the sample output above the *move operations* are **many orders of magnitude** faster than the *copy operations*. If your output doesn't have such a significant difference in times, keep working on your implementation (the actual numbers will be different every time you run the application).

**Reflection**

Study your final solution, reread the related parts of the course notes, and make sure that you have understood the concepts covered by this workshop. **This should take no less than 30 minutes of your time.**

Create a **text** file named reflect.txt that contains your detailed description of the topics that you have learned in completing this particular workshop and mention any issues that caused you difficulty and how you solved them. Include in your explanation—**but do not limit it to**—the following points:

* the reason for the significant time difference between the copy and move operations

**Quiz Reflection**

Add a section to reflect.txt called **Quiz X Reflection**. Replace the **X** with the number of the last quiz that you received and list all questions that you answered incorrectly.

Then for each incorrectly answered question write your mistake and the correct answer to that question. If you have missed the last quiz, then write all the questions and their answers.

**Submission (30% for code, 40% for reflection)**

To test and demonstrate execution of your program use the same data as shown in the output example above.

Upload the source code and the reflection file to your matrix account. Compile and run your code using the latest version of the g++ compiler (available at /usr/local/gcc/9.1.0/bin/g++) and make sure that everything works properly.

Then, run the following command from your account (replace profname.proflastname with your professor’s Seneca userid):

~profname.proflastname/submit 305XXX\_w2\_home

and follow the instructions. Replace XXX with the section letter(s) specified by your instructor.

**⚠️Important:** Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.