Lab 2 Analysis: Multithreading

PROGRAM EXECUTION DESCRIPTION (what the program does line by line)
To begin, the program prompts the user to create a JobSystem object. The main function then calls this object's CreateOrGet function which either creates the single instance or gets the single instance of this class. This is called the Singleton pattern and exists to ensure that there is only one JobSystem instance being used at any point in the code. The JobSystem class is a global accessor to the single instance of the real JobSystem.

Now that a JobSystem has been created and has an accessor, the program then prompts the user to create threads. After this, it creates 9 threads which are all able to accept any job available. These threads then pass the uniqueName and workerJobChannels into a function called CreateWorkerThread which essentially passes that into a thread object. That thread object then kicks off and waits for the instruction to start working from the main program. After all 9 of these threads have been created, it creates jobs as well.

The program is currently configured to only create 1 CompileJob and nothing else, as all the tasks are currently run with a single job. This effectively makes the multiple threads useless because only one is necessary. However, the minimum requirement for this class requires only one. If I wanted to utillize the different threads I would create other job objects with other tasks that could be given different parameters and commands. However, for now, a single command and a single job will do. After it pushes all the job objects to the vector of all jobs, it then queues all jobs by looping through the vector of all jobs and and queueing each job individually.

In order to queue a job, the main program tells the JobSystem to run the QueueJob function while giving it a parameter that points to the job it wants to queue. This function begins by attempting to lock a mutex on the shared recourses. This is necessary because the QueueJob function is attempting to add the job to m_jobHistory and m_jobsQueued. These two vectors are shared recourses which means that multiple different threads will be accessing or changing the data in these vectors making it a critical section. If multiple threads are accessing or changing data from these at the same time, it could lead to the data being written or read incorrectly, so it is important that only one thread accesses at once. To do this, before any data is accessed or changed, the thread needs to get a mutex lock on both the m_jobsQueuedMutex and m_jobHistoryMutex. If another thread has it locked, this thread will wait until it is able to lock it. When it is locked, no other thread can then lock the same mutex, because it is already locked. After the job is added to the jobHistory and jobsQueued, the mutexes are unlocked and other threads can access that data.

The jobs have officially been queued, the threads are up and running, and the program then prompts the user to enter a command. The commands include the following.

stop, destroyThreads, destroySystem, destroyJobs, finish, status, start stop:

The stop command simply sets running equal to 0 which stops the while loop, leading to the entire program being executed with a return code of 0.

destroyThreads:

The destroyThreads command calls the destroyAllThreads function in the JobSystem which just calls the DestroyWorkerThread function on each thread. This locks the m_workerThreadsMutex effectively making it so that nothing else can lock that same mutex until it is unlocked. It then creates a pointer to a JobWorkerThread called doomedWorker. After this, it iterates through the list of workerThreads using an iterator and points to each workerThread iteratively using the doomedWorker pointer. It then calls the erase function on each thread effectively destroying each one and getting rid of any memory it was taking up. It then unlocks the mutex and checks to see if the doomedWorker is still doing anything, if so it shuts down and deletes the doomedWorker pointer. This sets m_isStopping equal to true which means it will not pick up any other jobs and is then deleted. Eventually, each thread will have stopped picking up jobs and be destroyed.

destroySystem:

The destroySystem command calls the Destroy function in the JobSystem. This immediately terminates the system and destroys the JobSystem that the accessor pointed to, effectively shutting everything down. It then sets running equal to 0 which terminates the while look for the commands and ends the entire program.

destroyJobs:

The destroyJobs command loops through the list of jobs that the main program created and calls the destructor for each one. This does not stop any threads but simply destroys any jobs even if a thread is actively working on it.

finish:

The finish command calls the FinishCompletedJobs function which immediately creates an empty deque of Job pointers called jobsCompleted. It then locks the mutex for m_jobsCompleted and swaps the m_jobsCompleted vector in the JobSystem with the jobsCompleted vector just created. It then unlocks the mutex for m_jobsCompleted and creates a for each loop that loops through the jobsCompleted vector, which is now populated with all the jobs that have been completed by the JobSystem. For each of the jobs in the jobsCompleted vector, it calls that job's JobCompleteCallback function which essentially is a report function for each job. Each inherited job has its own JobCompleteCallback function but for the purpose of this program, we will analyze the CompileJob's JobCompleteCallback.

JobCompleteCallback:

The intention of this function is to output all the information that was saved from the execution in a json format. This includes all errors and warnings that may have been created while attempting to compile and all of the information that could have come included. It starts by creating an output file called rawOutput.txt that holds the raw output of all compilation. It then closes the file and moves on. After that, it opens an input from rawOutput to read the information and an output to a json file called output.json. It begins parsing through the rawOutput file to find any information that may be useful including the path file, line number, column number, error information, and code that was included in the error. After lots of tedious parsing and outputting information, the function manually outputs everything into the json file through basic writing functions. If it properly functions (which it should), it will output every error/warning of a the first file and its information, then move onto the next file and do the same thing until every error or warning has been parsed and outputted in json format. It then closes the streams and then couts the return code and output of the compiled file.

After the JobCompleteCallback has been called, it locks the m_jobHistoryMutex and proceeds to set the job's status to retired. This is different than being completed as now the callback has been called and it is no longer waiting for anything to be done. The mutex is then unlocked again and then the main program outputs the total number of completed jobs to the terminal.

status:

The status command prompts the user for a job ID. Once given the ID, the program then runs the GetJobStatus on the specific job given. The GetJobStatus function then locks the m_jobHistoryMutex and creates a new JobStatus object called jobStatus and sets it equal to the status of the job that the user provided the function. It is then returned and converted to an int. Depending on the value of this int, the program displays one of 5 statuses, NEVER_SEEN, QUEUED, RUNNING, COMPLETED, or RETIRED.

start:

The start command tells the jobSystem to run the startAllThreads function. This function locks the m_workerThreadsMutex (this should be proof that I understand when and where to use mutexes as this was a function I wrote away from class that required mutex control that I properly included) then loops through the m_workerThreads vector and individually runs the StartUp function of each one. The StartUp function is what actually creates the thread itself. Up to this point the threads were JobWorkerThread objects bur did not actually have a std::thread yet. This function actually creates an std::thread which immediately breaks off from the main program and starts working as the main program continues. This immediately runs the WorkerThreadMain function which then runs the Work function. The work function continuously, with mutex protection, attempts to ClaimAJob. This happens until IsStopping is set to true, which can be done from other command sections. ClaimAJob uses the current thread's channels to loop through the list of queued jobs and find one that matches at least one of its channels. This is done by running a logic and operation between the job's channels and the thread's channels, each being a hex number. If the output is greater than 0, it takes the job. It then executes the job which runs the Execute function of that specific job. Each inherited job has its own Execute function but the CompileJob's function types a specific command into the command line and captures the output. It also redirects the cerr messages to cout messages so that the errors can be captured by a a pipe that then saves the output and return code for later use by the JobCompleteCallback function. After the Execute function is finished, the OnJobCompleted function is called which locks the mutexes for m_jobsCompleted and m_jobsRunning. It then iterates through the m_jobsRunning deque and checks to see if the job that was just completed in the Execute function is equal to the job being pointed to by the iterator in the m_jobsRunning deque. It then locks the m_jobHistoryMutex, erases the job in m_jobsRunning, adds it to m_jobsCompleted, updates its status in m_jobHistory and then unlocks all the mutexes. This effectively updates the status of the job and moves it from the running deque to the completed deque. It then waits for one microsecond to ensure fairness in job claiming as well as making sure that reducing CPU usage.

The program then returns 0 effectively terminating all code and ending the program.

PROS AND CONS OF MUTEXES
Any program that uses threads needs some form of memory control. This is because with multiple threads, multiple functions are being ran at the same time. A problem with this arises when there are critical sections of memory, or when at least two different threads share the same memory area. If, for example, 2 threads are writing to the same file at the same time, then the file could be messed up. If thread one means to write “I like animals” and thread 2 means to write “Dogs are evil” then if they are both writing at the exact same time, the two messages could end up as something like “ID olgisk ea rnei meavlisl” which is a combination of the two messages. To avoid this, memory management is necessary and mutexes were the solution to that problem in this program. Here is what a mutex is:

> A mutex is a syncronization primitive which stands for mutual exclusion used in concurrent programming. It is a primitive that has two functions, locking or unlocking. A function and only lock a mutex if it is currently unlocked. If a function tries to lock a mutex that is already locked it will be blocked, which means it is put to sleep until the mutex is unlocked. A mutex can only be unlocked by the thread or program that has locked it.

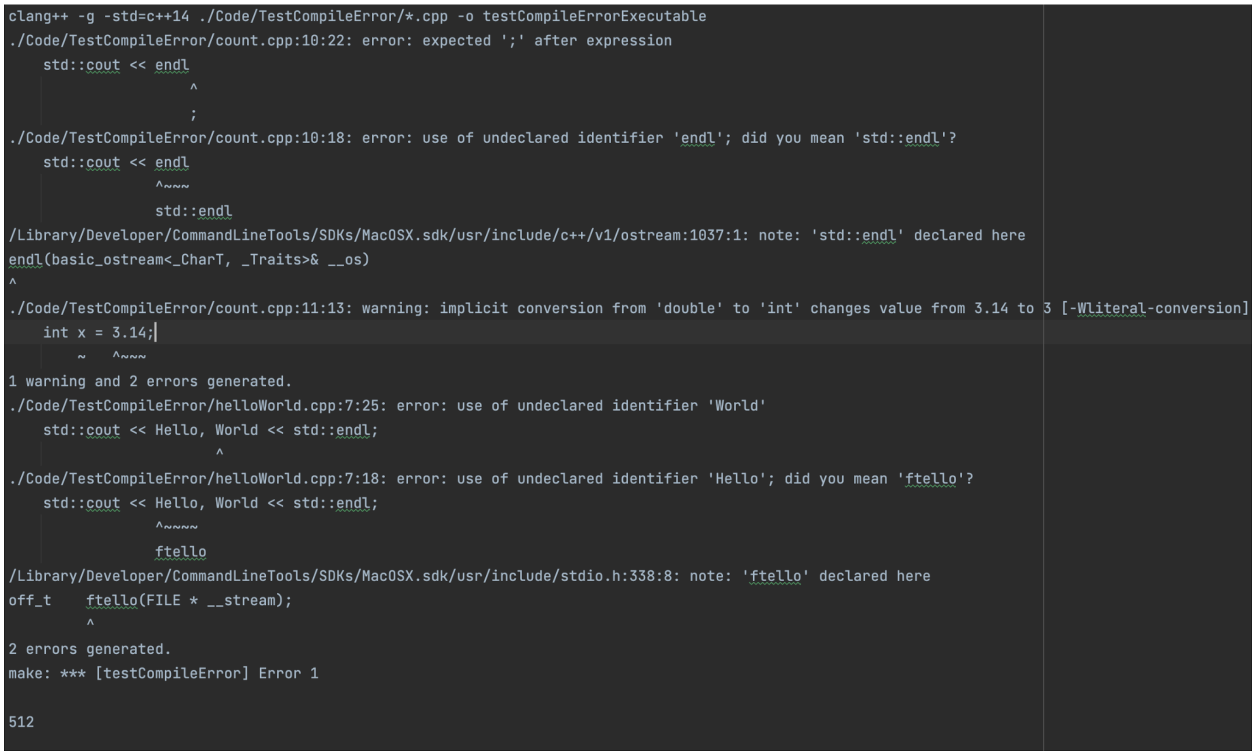
This means that if two threads need to write to a file but both need to lock a mutex first, then whichever thread locks the mutex first will be able to fully write everything it needs to write before the second one is able to do anything. However, if the mutex is never unlocked, the second thread will never be abl to write to the file. Mutexes are good in the sense that they are able to allow only one thread to be working at a specific time or doing a function at a specific time. However, because of the primitive nature of a mutex, it has low flexibility and can easily lead to a deadlock as it has no automatic unlocking mechanism. Here is a list of pros and cons about Mutexes

- Pros
    - Fine grained control: Allows precise and immediate synchronization over critical sections of code
    - Blocking: Prevents busy-waiting
    - Versatility: Allows for signalling between threads as well as exclusive assess to recourses
- Cons
    - Overhead: Each time you lock or unlock a mutex creates overhead
    - Deadlocks: Can easily lead to deadlocks.

Here are some other options for memory synchronization mechanisms
- Counting Semaphores

    - These are Mutexes (which are binary semaphores) that allow a specific amount of threads to lock it at once, no more than a specific count though.
    - Pros
        - Generalization: Semaphores can be used for more complex synchronization patterns, like allowing multiple threads to access a critical section at the same time
        - Versatility: Can be used for both blocking recourses and signalling between threads
        - Resource pools: Can be used to create complex pools of recourse in which multiple different threads are able to access the same memory
    - Cons
        - Very complex: It is hard to code with these and very easy to make mistakes
        - Debugging difficulty: If any problems arise it would be very difficult to find the source
- Atomic operations
    - Atomic operations are synchronization mechanisms that immediately execute a function. It is either entirely completed immediately or not at all, no other threads see any intermediate stages, only the completed stage. It cannot be interrupted by any other thread because it is indivisible.
    - Pros
        - Low Overhead: Have very low overhead because they're implemented directly in hardware
        - No Locking: They allow for lock-free data structures and algorithms, which can be highly efficient
        - Avoiding Deadlocks: Since atomic operations don't involve locking, they avoid deadlock scenarios
    - Cons
        - Limited Applicability: They're suitable for simple operations on basic data types, but not for complex operations or compound data structures.
        - Complexity of Correct Usage: Correctly using atomic operations can be tricky, especially for complex algorithms or data structures.

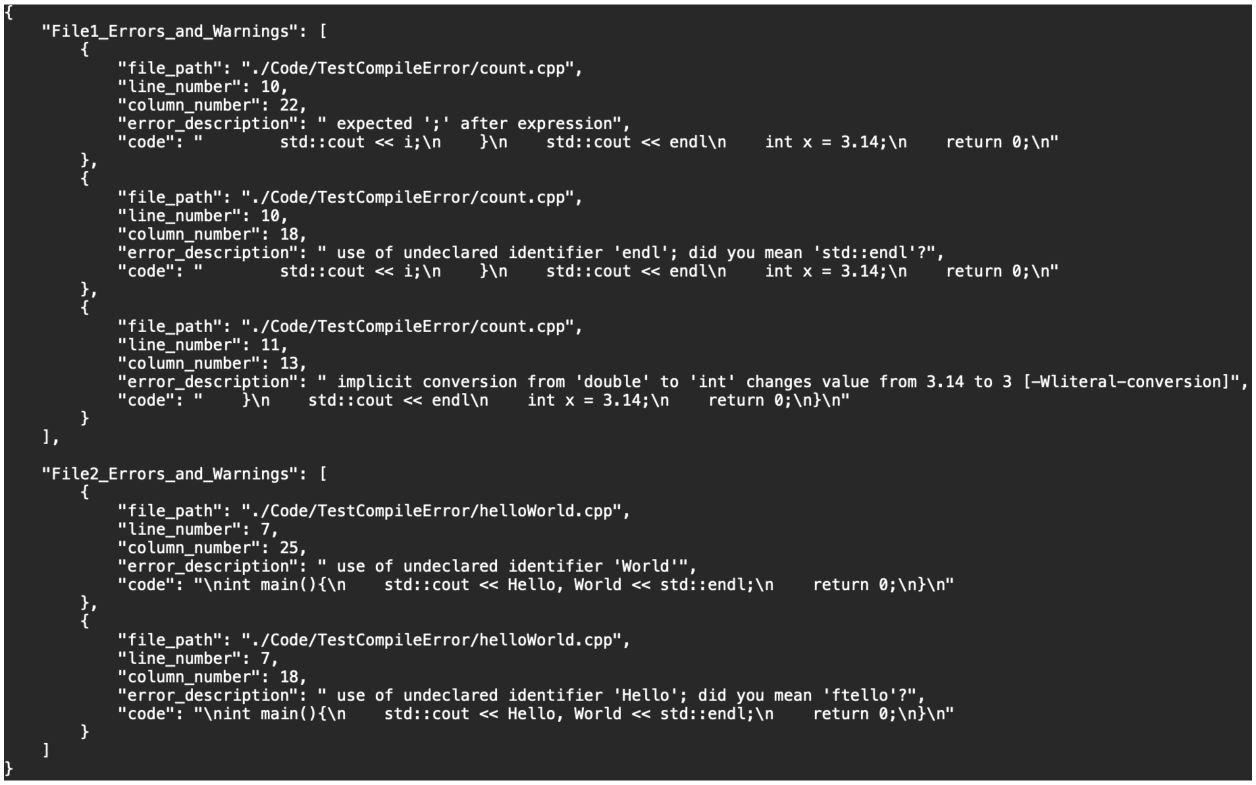
COMPILE SYSTEM DESCRIPTION
The compile system works by using a shell command typed in manually by the coder in the CompileJob class. It takes a command and appends 2>&1 to the end which redirects the cerr output to the cout output. The current command actually executes a rule in the make file that compiles all the source files in the TestCompileError file. It then uses a pipe to type this command into the command line and then reads the output and saves all the information into a string an takes the return code and saves that into a string as well. Eventually when the system is called back, it pastes all this information into a rawOutput.txt file. This could look something like this:



```
clang++ -g -std=c++14 ./Code/TestCompileError/*.cpp -o testCompileErrorExecutable
./Code/TestCompileError/count.cpp:10:22: error: expected ';' after expression
    std::cout << endl
                     ^
                     ;
./Code/TestCompileError/count.cpp:10:18: error: use of undeclared identifier 'endl'; did you mean 'std::endl'?
    std::cout << endl
                 ^~~~
                 std::endl
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:1037:1: note: 'std::endl' declared here
endl(basic_ostream<_CharT, _Traits>& __os)
^
./Code/TestCompileError/count.cpp:11:13: warning: implicit conversion from 'double' to 'int' changes value from 3.14 to 3 [-Wliteral-conversion]
    int x = 3.14;
        ~   ^~~~
1 warning and 2 errors generated.
./Code/TestCompileError/helloWorld.cpp:7:25: error: use of undeclared identifier 'World'
    std::cout << Hello, World << std::endl;
                        ^
./Code/TestCompileError/helloWorld.cpp:7:18: error: use of undeclared identifier 'Hello'; did you mean 'ftello'?
    std::cout << Hello, World << std::endl;
                 ^~~~~
                 ftello
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h:338:8: note: 'ftello' declared here
off_t    ftello(FILE * __stream);
         ^
2 errors generated.
make: *** [testCompileError] Error 1

512
```

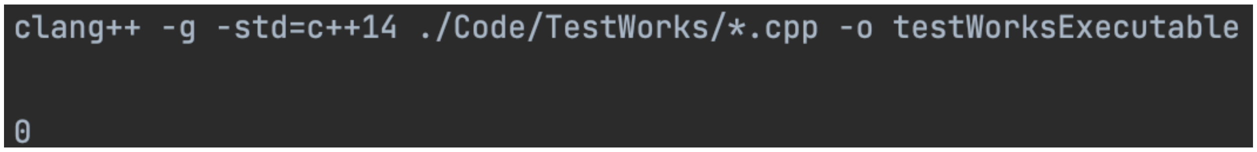
After all the information is pasted into the rawOutput, it is then parsed through by something that reads from that file and turns it into a json. I will not go into extensive detail about how exactly it parses because that would be EXTREMELY repetitive as a lot of it is just tabs and output characters. But the function uses simple fstream read and write commands to create the json file, no other tools. In the end it looks something like this:



```
{
    "File1_Errors_and_Warnings": [
        {
            "file_path": "./Code/TestCompileError/count.cpp",
            "line_number": 10,
            "column_number": 22,
            "error_description": " expected ';' after expression",
            "code": "        std::cout << i;\n    }\n    std::cout << endl\n    int x = 3.14;\n    return 0;\n"
        },
        {
            "file_path": "./Code/TestCompileError/count.cpp",
            "line_number": 10,
            "column_number": 18,
            "error_description": " use of undeclared identifier 'endl'; did you mean 'std::endl'?",
            "code": "        std::cout << i;\n    }\n    std::cout << endl\n    int x = 3.14;\n    return 0;\n"
        },
        {
            "file_path": "./Code/TestCompileError/count.cpp",
            "line_number": 11,
            "column_number": 13,
            "error_description": " implicit conversion from 'double' to 'int' changes value from 3.14 to 3 [-Wliteral-conversion]",
            "code": "    }\n    std::cout << endl\n    int x = 3.14;\n    return 0;\n}\n"
        }
    ],

    "File2_Errors_and_Warnings": [
        {
            "file_path": "./Code/TestCompileError/helloWorld.cpp",
            "line_number": 7,
            "column_number": 25,
            "error_description": " use of undeclared identifier 'World'",
            "code": "\nint main(){\n    std::cout << Hello, World << std::endl;\n    return 0;\n}\n"
        },
        {
            "file_path": "./Code/TestCompileError/helloWorld.cpp",
            "line_number": 7,
            "column_number": 18,
            "error_description": " use of undeclared identifier 'Hello'; did you mean 'ftello'?",
            "code": "\nint main(){\n    std::cout << Hello, World << std::endl;\n    return 0;\n}\n"
        }
    ]
}
```

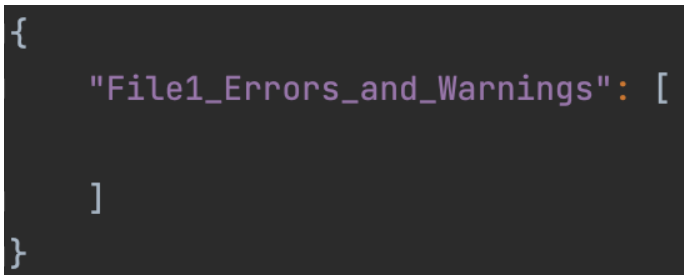
If there are no compile errors, the rawOutput.txt and output.json files are much more concise. Instead of having multiple errors to report, there are no errors and it simply shows the output and return code alone. So for example, it would show this instead:



```
clang++ -g -std=c++14 ./Code/TestWorks/*.cpp -o testWorksExecutable

0
```

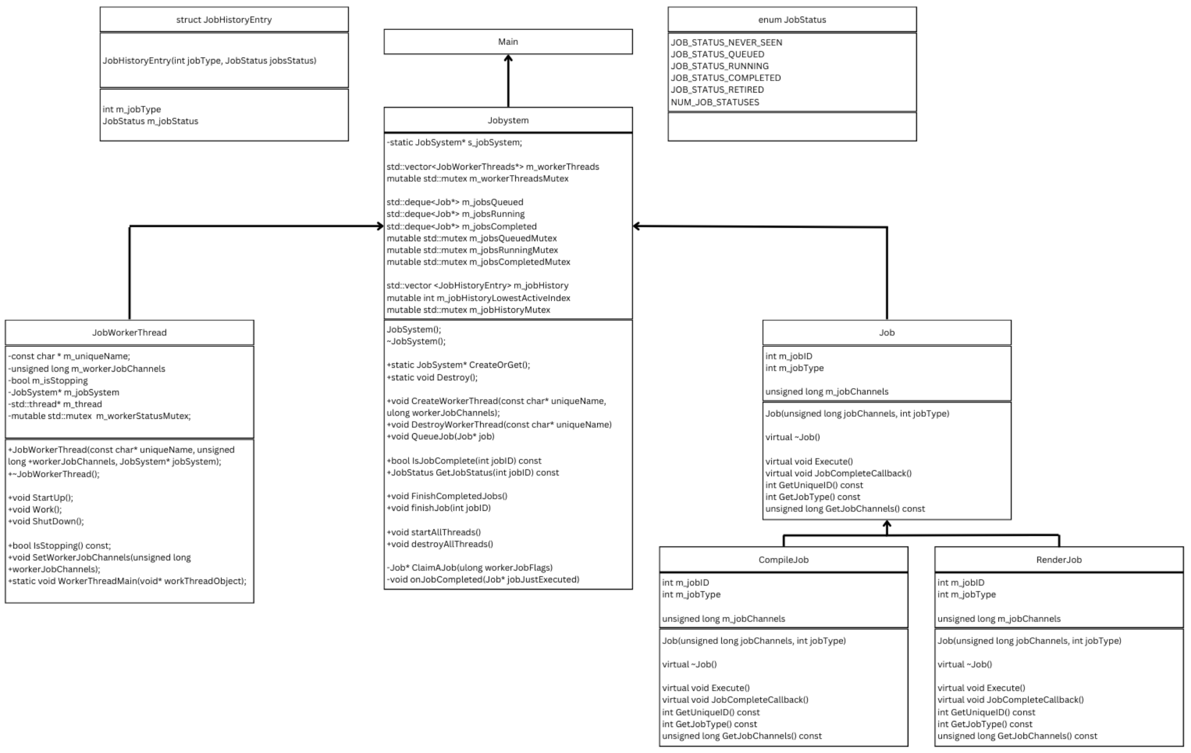
The output.json file is still given any errors that may have occurred though, but because there were none, it is given this insead.



```
{
    "File1_Errors_and_Warnings": [

    ]
}
```

UML DIAGRAM



```
struct JobHistoryEntry

JobHistoryEntry(int jobType, JobStatus jobsStatus)

int m_jobType
JobStatus m_jobStatus
```

```
Main
```

```
enum JobStatus

JOB_STATUS_NEVER_SEEN
JOB_STATUS_QUEUED
JOB_STATUS_RUNNING
JOB_STATUS_COMPLETED
JOB_STATUS_RETIRED
NUM_JOB_STATUSES
```

```
Jobystem

-static JobSystem* s_jobSystem;

std::vector<JobWorkerThreads*> m_workerThreads
mutable std::mutex m_workerThreadsMutex

std::deque<Job*> m_jobsQueued
std::deque<Job*> m_jobsRunning
std::deque<Job*> m_jobsCompleted
mutable std::mutex m_jobsQueuedMutex
mutable std::mutex m_jobsRunningMutex
mutable std::mutex m_jobsCompletedMutex

std::vector <JobHistoryEntry> m_jobHistory
mutable int m_jobHistoryLowestActiveIndex
mutable std::mutex m_jobHistoryMutex

JobSystem();
~JobSystem();

+static JobSystem* CreateOrGet();
+static void Destroy();

+void CreateWorkerThread(const char* uniqueName,
ulong workerJobChannels);
+void DestroyWorkerThread(const char* uniqueName)
+void QueueJob(Job* job)

+bool IsJobComplete(int jobID) const
+JobStatus GetJobStatus(int jobID) const

+void FinishCompletedJobs()
+void finishJob(int jobID)

+void startAllThreads()
+void destroyAllThreads()

-Job* ClaimAJob(ulong workerJobFlags)
-void onJobCompleted(Job* jobJustExecuted)
```

```
JobWorkerThread

-const char * m_uniqueName;
-unsigned long m_workerJobChannels
-bool m_isStopping
-JobSystem* m_jobSystem
-std::thread* m_thread
-mutable std::mutex  m_workerStatusMutex;

+JobWorkerThread(const char* uniqueName, unsigned
long +workerJobChannels, JobSystem* jobSystem);
+~JobWorkerThread();

+void StartUp();
+void Work();
+void ShutDown();

+bool IsStopping() const;
+void SetWorkerJobChannels(unsigned long
+workerJobChannels);
+static void WorkerThreadMain(void* workThreadObject);
```

```
Job

int m_jobID
int m_jobType

unsigned long m_jobChannels

Job(unsigned long jobChannels, int jobType)

virtual ~Job()

virtual void Execute()
virtual void JobCompleteCallback()
int GetUniqueID() const
int GetJobType() const
unsigned long GetJobChannels() const
```

```
CompileJob

int m_jobID
int m_jobType

unsigned long m_jobChannels

Job(unsigned long jobChannels, int jobType)

virtual ~Job()

virtual void Execute()
virtual void JobCompleteCallback()
int GetUniqueID() const
int GetJobType() const
unsigned long GetJobChannels() const
```

```
RenderJob

int m_jobID
int m_jobType

unsigned long m_jobChannels

Job(unsigned long jobChannels, int jobType)

virtual ~Job()

virtual void Execute()
virtual void JobCompleteCallback()
int GetUniqueID() const
int GetJobType() const
unsigned long GetJobChannels() const
```