Max link

Background on Approaches to Designing a Multithreaded System

<u>Method A</u>

- Method A involves managing memory chunks directly dispatched to child threads.
    - Pros:
        - Threads work on actual memory, reducing latency and enhancing efficiency.
    - Cons:
        - Increased complexity in managing thread safety.

<u>Application</u>:

- Applicable for scenarios where high performance is a priority, outweighing the complexity of implementation.

<u>Method B</u>

- Child threads utilize their memory by communicating with the main thread.
    - Pros:
        - This method is more straightforward, reducing the potential for errors related to manual memory management.
    - Cons:
        - This method may operate more slowly compared to Method A due to additional communication layers.

<u>Application</u>

- Applicable for scenarios where performance is not as important, and quick implementation of a job system is needed, or a system must be built that can be easily expanded without running into thread issues.

Given the balance of simplicity and performance, Method B is selected for this coding project.

General Design Decisions

The code emphasizes a class-based structure for handling various jobs, enhancing organization and scalability. The job management is encapsulated between the job classes only. With this approach, Jobsystem, job, jobworkerthreads all handle the thread mutex management with deques. This method allows the user to code and forget about how the jobs are supposed to be managed. The individual jobs that are programmed – compilejob, parsejob, and outputjob – inherit their properties with polymorphism from the job class. This polymorphism gives these job classes the properties of job like the job id, job type, and the get id function. This encapsulation allows the user easy job creation because they can abstract the id and jobtype functions away. Bitmasking is used to verify whether the worker has any job channels in common with the job. This is the main line of logic: if((queuedJob->m_jobChannels & workerJobChannels) != 0).

The jobsystem uses the singleton pattern where it is a static function so there is only one ever created at class level. The constructor for the job system is made private and there is a createorget() method that is used in main.cpp to kick off the creation of your jobsystem object, with a check during creation that makes sure a jobsystem object has not already been instantiated.

Overview of main.cpp:

Main.cpp instantiates the jobsystem and jobs. The makefile file path is passed in to be passed to compileJob for running "make automated". Passing the filepath of make into a string allows the functionality to be more dynamic for future development by preventing hardcoded lines. Main.cpp creates a queue called stream to hold what compilejob grabs from the terminal. The queue stream is passed by reference to parsingJob to get parsed into an intermediary errors.json file, then outputJob reads from that errors.json file to construct the final output.json file. Main uses busy while loops to handle synchronization of threads by checking the IsJobCompleted bool in the jobsystem class by hardcoded jobID. The while loop is set to run until the thread is IsCompletedStatus returns false for a particular job. As soon as the child thread completes, the IsJobCompleted status returns true and while loop is broken out of and the main thread proceeds. This happens for each job. This solution is quick and easy to implement for syncronization, but should be changed to an await or promise on later implementation to avoid looping for faster thread synchronization.

Main.cpp Code:

```
#include <iostream>
#include <string>
#include <thread>

#include "jobsystem.h"
#include "renderjob.h"
#include "compilejob.h"
#include "parsingjob.h"
#include "outputjob.h"
#include <queue> //for streaming content from compile job to parsing job

//TODO - add promise #include <future> for thread waiting instead of busy whiles?
int main(void){

    //basicThreadControl btc;
    //btc.kickoff();

    //find out how many cores are on the machine
    //unsigned int nCores = std::thread::hardware_concurrency();
    //std::cout << "Number of cores: " << nCores << std::endl;
    std::cout << "Creating Job System" << std::endl;

    JobSystem* js = JobSystem::CreateOrGet();

    std::cout << "Creating Worker Threads" << std::endl;

    //determined by number of cores on machine
    js->CreateWorkerThread("Thread1", 0xFFFFFFFF);
    js->CreateWorkerThread("Thread2", 0xFFFFFFFF);
    js->CreateWorkerThread("Thread3", 0xFFFFFFFF);
    js->CreateWorkerThread("Thread4", 0xFFFFFFFF);
    js->CreateWorkerThread("Thread5", 0xFFFFFFFF);
    js->CreateWorkerThread("Thread6", 0xFFFFFFFF);
    js->CreateWorkerThread("Thread7", 0xFFFFFFFF);
    js->CreateWorkerThread("Thread8", 0xFFFFFFFF);

    std::cout << "Creating Jobs" << std::endl;

    std::vector<Job*> jobs;

    std::queue<std::string> stream;

    std::string directoryToMakefile = "makefile"; //explanatory var for makefile
directory
    //created a compile job
    CompileJob* cjb = new CompileJob(0xFFFFFFFF, 1, stream, directoryToMakefile);
//pass in directory to makefile to run make command in cjb
    jobs.push_back(cjb);
    js->QueueJob(cjb); //queue compile job

    std::cout << "Queuing CJB" << std::endl;

    while(!js->IsJobComplete(0) == 1){} //busy for cjb

    //make parsejob
    ParsingJob* pjb = new ParsingJob(0xFFFFFFFF, 1, stream);
    jobs.push_back(pjb); //add parsing job to jobs vector
    js->QueueJob(pjb); //queue parsing job

    while(!js->IsJobComplete(1) == 1){} //busy for pjb

    //make output job
    OutputJob* ojb = new OutputJob(0xFFFFFFFF, 1); //takes errors.json from data
folder
    jobs.push_back(ojb); //add output job to jobs vector
    js->QueueJob(ojb); //queue output job
    while(!js->IsJobComplete(2) == 1){}

    js->FinishCompletedJobs();
    std::cout << "total jobs completed " << js->totalJobs <<std::endl;
    js->Destroy();

    return 0;

}
```

Code overview for job.h:

Class Job is used with a typical structure where a public virtual function, Execute, is defined and must be overridden in derived classes. This virtual function lets the jobs have a custom implementation of the function, which lets each job do a unique execute function. JobSystem and JobWorkerThread are both friend classes of Job.h, so they can access its properties. This ensures encapsulation where all of the job classes have access to eachother's properites, but main.cpp and the child jobs (compileJob, ParsingJob, and outputJob) do not have access to these properties. This abstracts the code workflow and makes the codebase easier to expand on and protect for future development.

```
class Job{
    friend class JobSystem;
    friend class JobWorkerThread;
    public:
    Job(unsigned long jobChannels = 0xFFFFFFFF, int jobType = -1) :
m_jobChannels(jobChannels), m_jobType(jobType) {
        static int s_nextJobID = 0;
        m_jobID = s_nextJobID++;
    }

    virtual ~Job() {}

    virtual void Execute() = 0; //body set = 0 so purely virtual, abstract class. Must
inherent the execute function
    virtual void JobCompletedCallback() {} //do not have to implement this function
(body not set to zero, so if implemented then does nothing)
                                            //can call if want to.
    int GetUniqueID() const { return m_jobID; } //job type that returns will be a
const so cannot be manipulated

    private:
    int m_jobID = -1;
    int m_jobType = -1;
    unsigned long m_jobChannels = 0xFFFFFFFF; //32 bits, each bit represents a
channel.

};
```

Memory Management and Thread Protection Paradigm

The system employs three deques for organized and efficient memory management. Mutexes ensure atomic operations for thread safety. The job management system in with the use of the job, jobsystem, JobWorkerThread classes. The JobWorkerThread has access to the jobsystem properties because the JobWorkerThread is a friend class of the jobsystem. The jobsystem handles the bulk of the thread management, with the 3 deques being used to move threads from queued status to running status to completed status. Throughout all of these status changes, the jobsystem updates the thread history using mutex safe protocols in the jobsystem class.

Associated Code in jobsystem.h

```
std::vector<JobWorkerThread* > 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 = 0; //oldest still
running job would be lowest number
    mutable std::mutex              m_jobHistoryMutex;
```

Here you can see the deques and mutex for each deque defined in the jobsystem.h file. These mutex are used to mark important lines in the jobsystem.cpp file in the various helper functions where data is being moved between deques so that multiple threads do not conflict.

The associated code in the jobsystem.cpp file can be seen here in each helper function:

<u>ClaimAJob():</u>

in the beginning of ClaimAJob you lock the mutex for the m_jobsrunning queue and m_jobsQueued queue, which signals that these queues will be modified so only one thread should be able to access them at a time to prevent data conflict. Bitmask logic verifies that a job is available and the jobhistory vector mutex is locked, which again signals that this important component will be modified so there should be no access from multiple threads. The claimjob variable is set to queued job and queued job is deleted from the jobsqueued vector. Claimjob is pushed into the jobs running vector, because a job that is claimed by a thread will be ran by that thread. Finally the mutexes are all unlocked so that multithread access may resume and claimjob is returned.

```
m_jobsQueuedMutex.lock();
    m_jobsRunningMutex.lock();

    Job* claimedJob = nullptr;
    std::deque< Job * >::iterator queuedJobIter = m_jobsQueued.begin();
    for( ; queuedJobIter != m_jobsQueued.end(); ++queuedJobIter){
        Job* queuedJob = *queuedJobIter;

        if((queuedJob->m_jobChannels & workerJobChannels) != 0){//bit mask logic
operator not an & op. if 0 then something =1 for some reason
            claimedJob = queuedJob;

            m_jobHistoryMutex.lock(); //locks thread so every other thread is waiting
            m_jobsQueued.erase(queuedJobIter);
            m_jobsRunning.push_back(claimedJob);
            m_jobHistory[claimedJob->m_jobID].m_jobStatus = JOB_STATUS_RUNNING;
            //
            //DEAD LOCK WAS HERE BUT FIXED BY MOVING UNLOCK TO LINE 241
            //
            m_jobHistoryMutex.unlock(); //does not affect thread that grabs first job
            //will get stuck with deadlock unless unlock here
            break;
        }
    }

    m_jobsRunningMutex.unlock();
    m_jobsQueuedMutex.unlock();

    return claimedJob;
```

<u>JobSystem():</u>

The job system constructor reserves space in the job history list with this line

```
    m_jobHistory.reserve(256 * 1024);
```

<u>~JobSystem():</u>

The job system destructor locks the worker threads mutex because it will be modifying the worker threads vector, so it should be the only thread accessing the vector. This vector holds the pointers to these job threads. The destructor gets the number of threads thaht have been made, and tells each workerthread to stop picking up jobs with the m_workerThread[i]→ShutDown()

line. The function then loops through the worker threads and deletes the worker thread objects (the threads are not a concern, just the objects being removed from instantiation). Finally, the workerthread mutex is unlocked.

```
JobSystem::~JobSystem(){//letting threads finish is the nice way to clean up -- even
though can kill
    m_workerThreadsMutex.lock();
    int numWorkerThreads = (int)m_workerThreads.size();
    //we built in functions for letting threads run to completion. Will call those

    //First, tell each worker thread to stop picking up jobs
    for(int i = 0; i < numWorkerThreads; ++i){
        m_workerThreads[i]->ShutDown();
    }

    while(!m_workerThreads.empty()){
        //going through vector of multithreads
        //deleting worker thread objects, but threads are not a concern
        delete m_workerThreads.back();
        m_workerThreads.pop_back();
    }
    m_workerThreadsMutex.unlock();


}
```

<u>CreateorGet():</u>

This function checks to make sure jobsystem does not exist, if it does not then jobsystem is created, but if jobsystem currently exists then the current jobsystem is simply returned. This ensure that there is only ever one jobsystem in play at a time.

```
//ensures theres only one job system created
JobSystem* JobSystem::CreateOrGet(){
    if(!s_jobSystem){//if there isnt a job system, create one
        s_jobSystem = new JobSystem();
    }
    return s_jobSystem; //if theres already one then return it
}
```

Similarly, there is a destroy function for the jobsystem. This destroy function checks if there is a jobsystem, and if a jobsystem exists then the "delete s_jobsystem;" command deletes the jobsystem and the "s_jobsystem = nullptr" command sets the jobsystem to a nullptr for memory safety.

```
//if there is a job system, delete it & set to nullptr
void JobSystem::Destroy(){
    if(s_jobSystem){
        delete s_jobSystem;
        s_jobSystem = nullptr;
    }
}
```

CreateWorkerThread():

The CreateWorkerThread function creates a worker thread by using the constructor in the jobworkerthread class. Then the CreateWorkerThread function locks the workerthreads mutex and pushes the newly created worker to the workerthreads vector. This encapsulation simplifies the duty of this function and the mutex control ensures that there will never be multiple threads interacting with the workerthreads vector at once.

```
void JobSystem::CreateWorkerThread(const char* uniqueName, unsigned long
workerJobChannels){

    JobWorkerThread* newWorker = new JobWorkerThread (uniqueName, workerJobChannels,
this);

    m_workerThreadsMutex.lock();
    m_workerThreads.push_back(newWorker);
    m_workerThreadsMutex.unlock();

    m_workerThreads.back() -> StartUp();

}
```

Similarly, destroyworkerthread() locks the workerthreads mutex so that the workerthreads can be destroyed. This function instantiates a jobWorkerThread pointer called doomedWorker and sets the variable to null pointer. This function iterates through the workerthreads vector and checks the uniqueName of the main thread and of the worker, and if they match then the thread should be destroyed. The doomedWorker thread variable is then set to the contents of the m_workerThreads and the m_workerThreads are erased. Finally, the m_workerThreadMutex is unlocked, and if there is content in the doomedWorker variable, the shutdown function is called on the doomedWorker and the doomedWorker is deleted.

```
void JobSystem::DestroyWorkerThread(const char* uniqueName){
    m_workerThreadsMutex.lock();
    JobWorkerThread* doomedWorker = nullptr;
    std::vector<JobWorkerThread*>::iterator it = m_workerThreads.begin();

    for(; it != m_workerThreads.end(); ++it){
        if(strcmp( (*it)->m_uniqueName, uniqueName) == 0){
            doomedWorker = *it;
            m_workerThreads.erase(it);
            break;
        }
    }



    m_workerThreadsMutex.unlock();

    if( doomedWorker){
        doomedWorker->ShutDown();
        delete doomedWorker;
    }
}
```

QueueJob():

The QueueJob function works with the job history and jobsqueued deque to update update them both, which signals a job has been queued. Respective Mutexes are locked to avoid multiple threads accessing the same data and causing conflicts.

```
void JobSystem::QueueJob(Job* job){//working w/ job histories & job queue
//pushing creaetd job into job queue
//update job entry record to say job is queued
    m_jobsQueuedMutex.lock();

    m_jobHistoryMutex.lock();
    m_jobHistory.emplace_back(JobHistoryEntry(job->m_jobType, JOB_STATUS_QUEUED));
    m_jobHistoryMutex.unlock();

    m_jobsQueued.push_back(job);
    m_jobsQueuedMutex.unlock();

}
```

GetJobStatus():

The GetJobStatus function locks the jobHistory Mutex and sets the jobStatus to JOB_STATUS_NEVER_SEEN. If the job ID is less than the jobHistory list size, then that means that job must be in the jobHistory list. If the job is in the jobHistory list, then its jobstatus is grabbed out of the jobHistory list. The jobHistory mutex is locked and jobStatus is returned.

```
JobStatus JobSystem::GetJobStatus(int jobID) const{//jobstatus is an array that we are
pushing onto an array
//should be indexed so we dont have to search through array
//more complex system where jobs can overwrite previous jobs then this will not be
enough -- would have to search the loop
    m_jobHistoryMutex.lock();
    JobStatus jobStatus = JOB_STATUS_NEVER_SEEN;
    if(jobID < (int) m_jobHistory.size()){ //if(jobID, (int) m_jobHistory.size()){
        jobStatus = (JobStatus) m_jobHistory[jobID].m_jobStatus;
    }

    m_jobHistoryMutex.unlock();

    return jobStatus;
}
```

This getJobStatus() is used in the IsJobComplete function to return a Boolean. This function simply called GetJobStatus & passed in a JobID to GetJobStatus. If the returned job status is equal to “JOB_STATUS_COMPLETED” then the the IsJobComplete function returns true, if the GetJobStatus returned value is not equal then the IsJobComplete function returns false.

```
bool JobSystem::IsJobComplete(int jobID) const{
    //return JobStatus (jobID) == JOB_STATUS_COMPLETED;
    return (GetJobStatus(jobID)) == (JOB_STATUS_COMPLETED);
}
```

FinishCompletedJobs():

This function locks the jobsCompletedMutex and copies all of the objects from m_jobsCompleted to a local deck called jobsCompleted. This copying ensures that m_jobsCompletedMutex can be unlocked as quickly as possible, which frees up thread movement. The local deck jobsCompleted is iterated through and the callbacks are grabbed from the JobCompletedCallback functions (local functions in each job class to signal job complete). The jobHistory mutex is locked so that the jobHistory deque’s status can be set to retired for the jobs. The jobHistory mutex is then unlocked and the job variable is deleted.

```
void JobSystem::FinishCompletedJobs(){

    std::deque<Job*> jobsCompleted;

    m_jobsCompletedMutex.lock();

    jobsCompleted.swap(m_jobsCompleted); //grabs all completed objects in deck and 
copies into local deck
    //allows processing while not keeping queue locked up

    m_jobsCompletedMutex.unlock();

    for(Job* job : jobsCompleted){
        job->JobCompletedCallback(); //grab callback
    m_jobHistoryMutex.lock();
    m_jobHistory[job->m_jobID].m_jobStatus = JOB_STATUS_RETIRED; //update status
    m_jobHistoryMutex.unlock();
    delete job;  //delete the job

    }

}
```

<u>FinishJob():</u>

The finish job function queries the job ID's to only take jobs that are not yet complete by checking each ID with the IsJobComplete Boolean function. The jobStatus for the jobs that are not completed is then checked to ensure that the job queried does not have a "JOB_STATUS_NEVER_SEEN" or "JOB_STATUS_RETIRED" status. If a job does have either of these status then an exception is printed, as this is undefined behavior for this function. However, if the job doesn't have the "JOB_STATUS_NEVER_SEEN" or "JOB_STATUS_RETIRED", then it must have the "JOB_STATUS_RUNNING", which means the job should be finished. For this a variable named thisCompletedJob is created and set to nullptr. The jobsCompletedMutex is locked and iterated through. In the iteration, a variable named someCompletedJob is set to the contents of the current iteration of JobsCompleted. On the condition that the someCompletedJobs→ID matches the jobID, then thisCompletedJob is set to someCompletedJob content, all the content is erased out of the m_jobsCompleted vector, and the loop is broken. The mutex for jobsCompleted is unlocked to resume thread flow. If the thisCompletedJob is a nullptr then an exception is caught, otherwise "thisCompletedJob → JobCompletedCallback();" is called. The jobHistoryMutex is locked so that JobHistory status for the jobs may be set to JOB_STATUS_RETIRED. JobHistoryMutex is unlocked and "thisCompletedJob" is deleted.

```
void JobSystem::FinishJob(int jobID){
    while(!IsJobComplete(jobID)){
        JobStatus jobStatus = GetJobStatus(jobID);
        if(jobStatus == JOB_STATUS_NEVER_SEEN || jobStatus == JOB_STATUS_RETIRED){
            std::cout << "ERROR: Waiting for job(#:" << jobID << ") - no such job in
JobSystem!" << std::endl;
            return;
        }

        m_jobsCompletedMutex.lock();
        Job* thisCompletedJob = nullptr;
        for(auto jcIter = m_jobsCompleted.begin(); jcIter != m_jobsCompleted.end();
++jcIter){
            Job* someCompletedJob = *jcIter;
            if(someCompletedJob->m_jobID == jobID){
                thisCompletedJob = someCompletedJob;
                m_jobsCompleted.erase(jcIter);
                break;
            }
        }
        m_jobsCompletedMutex.unlock();

        if(thisCompletedJob == nullptr){
            std::cout << "Error: Job #" << jobID << " was status complete but not
found in completed list!" << std::endl;

        }

        thisCompletedJob -> JobCompletedCallback();  //find job

        m_jobHistoryMutex.lock();
        m_jobHistory[thisCompletedJob->m_jobID].m_jobStatus = JOB_STATUS_RETIRED;
//update history record
        m_jobHistoryMutex.unlock();
        delete thisCompletedJob; //delete job

    }
}
```

<u>OnJobCompleted():</u>

The OnJobCompleted function locks the jobsCompletedMutex and jobsRunningMutex and creates an iterator variable called runningJobItr to iterate over running jobs. Inside the iteration, if the jobJustExecuted is equal to any of the content in the runningJobItr variable, then:

1. The current job that the runningJobItr is on will be erased from the jobsRunning list
2. jobJustExecuted will be pushed back to the jobsCompleted list
3. The jobHistory will be updated to list the jobJustExecuted status as "Job_Status_Completed"
4. All locked Mutexes will be unlocked

```
void JobSystem::OnJobCompleted(Job* jobJustExecuted){
    totalJobs++; //for testing - not important

    //running on two at same time protect both
    m_jobsCompletedMutex.lock();
    m_jobsRunningMutex.lock();

    std::deque<Job*>::iterator runningJobItr = m_jobsRunning.begin();
    for(; runningJobItr != m_jobsRunning.end(); ++runningJobItr){
        if( jobJustExecuted == * runningJobItr){
            m_jobHistoryMutex.lock();
            m_jobsRunning.erase(runningJobItr);
            m_jobsCompleted.push_back(jobJustExecuted);
            m_jobHistory[jobJustExecuted->m_jobID].m_jobStatus = JOB_STATUS_COMPLETED;
            m_jobHistoryMutex.unlock();
            break;
        }
    }

    m_jobsRunningMutex.unlock();
    m_jobsCompletedMutex.unlock();

    //can add many more things to job system -- this shows how to loop through two
differnet decks grab one erase one
    //swap command very quick in this .cpp file

}
```

JobWorkerThread class:

The jobWorkerThread class has a number of helper functions to have child threads do work, set their channels, shut them down, etc.

JobWorkerThread Constructor/Destructor:

```
JobWorkerThread::JobWorkerThread(const char* uniqueName, unsigned long
workerJobChannels, JobSystem* jobSystem) :
    m_uniqueName(uniqueName),
    m_workerJobChannels(workerJobChannels),
    m_jobSystem(jobSystem){
}

JobWorkerThread::~JobWorkerThread(){
    //If we havent already,
    //signal thread that we should exit as soon as it can (after its current job if
any)
    ShutDown();

    //Block on the thread's main until it has a chance to finish its current job and
exit
    //cleanup process
    m_thread->join();
    delete m_thread;
    m_thread = nullptr;

}
```

JobWorkerThread::Shutdown():

```
void JobWorkerThread::ShutDown(){//stop working
    //m_isStopping can be called by main thread, so needs Mutex protection
    //thread protect this so that data does not get corrupted with reading & writing
at same time
    m_workerStatusMutex.lock();
    m_isStopping = true;
    m_workerStatusMutex.unlock();
}
```

JobWorkerThread::SetWorkerJobChannels:

```
void JobWorkerThread::SetWorkerJobChannels(unsigned long workerJobChannels){
    m_workerStatusMutex.lock();
    m_workerJobChannels = workerJobChannels; //could be set in main thread, so protect
it w/ mutex lock/unlock
    m_workerStatusMutex.unlock();
}
```

These functions all guarantee an encapsulated system of interaction between the main thread and child threads and jobsystem because these functions are all private in the jobWorkerThread class, which means that only the child threads can access themselves and change their properties. JobSystem is a friend class of JobWorkerThread which means that JobSystem can access these functions in its methods, which can be seen for use cases like this in JobSystem:

Shutdown function called on child thread in jobSystem:

```
JobSystem::~JobSystem(){//letting threads finish is the nice way to clean up -- even
though can kill
    m_workerThreadsMutex.lock();
    int numWorkerThreads = (int)m_workerThreads.size();
    //we built in functions for letting threads run to completion. Will call those

    //First, tell each worker thread to stop picking up jobs
    for(int i = 0; i < numWorkerThreads; ++i){
        m_workerThreads[i]->ShutDown();
    }
………CONTINUED……….
```

General Overview of Design Decisions for Child Jobs:

This section covers the different child job implementations – compilejob, parsejob, and outputjob.

Compile Job

CompileJob is responsible for taking the output out of the command line and passing the terminal output off to the parsing job. My program uses a queue in main called stream that is passed into compilejob by reference, as to not make a copy to prevent overhead. The compilejob class execute function takes the path of the make file so that the function can run “make automated” in the terminal. The function then extracts all of the content with a buffer and puts it into a string. The output is then loaded into the stream queue that also exists in main.cpp. Although not implemented to its full extension, this use of a queue stream allows future optimization by letting parsingjob read the contents that compilejob grabs as soon as compilejob starts filling up stream. This method sets the program up for easy optimization for interjob communication, as opposed to reading in content from the data folder.

Associated Code in compileJob for:

- Reading from the terminal and putting in the buffer:

```
std::array< char, 128 > buffer; //buffer to store output
    std::string command = "make -f " + makefile + " automated"; //path to makefile and
command to run makefile based on file path

    //Redirect cerr to cout - easy way to see errors when working with multiple
threads
    //then can capture cout and get all errors
    command.append(" 2>&1"); //redirects stderr to stdout
    //spin up a thread that is going to grab code, compile it, and return a result
    //Terminal tells you there is an error in code, but with threads you cannot tell
where the error is
    //cout to terminal is not thread safe (standard commands like that are not thread
safe)

    //open pipe and run command
    FILE* pipe = popen(command.c_str(), "r"); //opens a pipe and will send something
to it
    //gives terminal to work on, but this terminal is open inside a thread
    //turns command into a c string and sends it to the pipe
    //you can only open pipe to read in this case "r"

    if(!pipe){
        std::cout << "Pipe failed to open" << std::endl;
        return;
    }

    //read till end of process:
    while(fgets(buffer.data(), 128, pipe) != NULL){//reading buffer until get to end
of file - same concept as finishing reading in terminal
        this->output.append(buffer.data()); //append to output string
    }
```

- Loading the buffer info into the stream

```
-   //split output and feed into stream
-       std::stringstream ss(this->output); //create stringstream from output
    string
-       std::string transcribeOut; //string to store words from output string
-       while(std::getline(ss, transcribeOut, '\n')){ //while there are words in
    the stringstream
-           //std::cout << "TranscribeOut: " << transcribeOut << std::endl;
-           streamCjb.push(transcribeOut); //push word to stream
-       }
```

Parsing Job and Output Job

Parsing and Output Jobs manage error parsing and output management, ensuring clear and understandable error and output data. The errors.json is an intermediary file that is stored in the data folder of the project and generated by parsing job. This file stores the errors/warnings as objects.

The output.json is the final output file that is generated by outputJob. This file stores errors/warnings in a nested array structure. This nested array structure is organized as:

```
 [
[
{File 1 error}
],
[
{File 2 error},
{File 2 warning}
]
 ]
```

Where each array in the json file stores object that belong to a specific file. These objects hold errors/warnings belonging to that specific file. This structure makes it intuitive to store multiple errors/warnings across multiple different files in an organized way.

Associated code in parsingJob for parsing the stream:

- the stream is fed into a string variable

```
-   std::string content;
-
-      nlohmann::json errorJsonArray = nlohmann::json::array(); // Initialize a
    JSON array to append errors to json
-      std::vector<std::string> parsedStrings; //vector to store parsed strings
-
-       //START PARSING & WRITING TO JSON FILE
-       // Print and empty the queue
-       while (!streamPjb.empty()) {//take the stream contents & put them into
    string using clang++ as delimiter
-           content += streamPjb.front();
-           streamPjb.pop();
-       }
-
```

- Regex is defined for parsing for either a linker error caused by duplicate mains being detected, a linker error caused by a forward declaration, or a non-linker error. Regex allows parsing that can be easily modified based on simple expressions later, so parsing is easily expandable.

```
-   // Check if it's a linker error
-       std::regex mainLinkerErrorRegex(R"(duplicate symbol '(.*)'
    in:((?:.*\s*)+))"); //REGEX FOR MULT. MAIN DEC. LINKER ERROR PARSING
-       //std::regex forwardDeclarationErrorRegex(R"(Undefined symbols:\s+_(.*),
    referenced from:)"); //REGEX FOR FORWARD DECLARATION ERROR PARSING
-       std::regex forwardDeclarationErrorRegex(R"(Undefined symbols:\s+(.*),
    referenced from:)");
-

-       std::regex errorRegex; // Declare errorRegex here for scope reasons
```

- Checks if error is a linker error based on regex and sets Boolean flag to true for linker error if there is a linker error detected, and uses Boolean in parsing function

```
-   if (std::regex_search(content, mainLinkerErrorRegex) ||
    std::regex_search(content, forwardDeclarationErrorRegex)) {//TODO - boolean not
    being entered, why?
-           isLinkerError = true;
-           //std::cout << "BOOL CHANGED: " << isLinkerError << std::endl;
-       }
-       //isLinkerError = true; //TODO - for testing remove later
-       StringParsing(content, parsedStrings); //parse string for file path, line
    number, column number, error type, and error description
-
```

- Parsing function uses delimiter "./" for nonlinker errors and if a linker error is detected then no delimiter is used.

```
-   void ParsingJob::StringParsing(std::string &content, std::vector<std::string>
    &parsedStrings){
-
-       std::string errorLog = content;
-
-       std::string delimiter = "./";
-       size_t pos = 0;
-
-       if(isLinkerError == 0){//not a linker error so parse ./ delimiter out
-           while ((pos = errorLog.find(delimiter)) != std::string::npos) {
-               std::string token = errorLog.substr(0, pos);
-               parsedStrings.push_back(token);
-               errorLog.erase(0, pos + delimiter.length());
-           }
-       }
-       parsedStrings.push_back(errorLog);
-
-   }
```

This approach of parsing based on a delimiter allows parsingJob to cut out fat from the terminal error/warning so that outputJob has an easier time formatting the final output, but admittedly can be limiting. Fully relying on a library like the nlohmann::json library for parsing the errors/warnings might be a way to optimize this code later on in the development process. After parsing the string is completed, the string is written to a parsedStrings vector. The parsedStrings vector is used to format the intermediary json file errors.json that outputJob reads from. The errors.json is formatted as such depending on if a linkererror was encountered or not:

```cpp
if(isLinkerError == 1) {
    //std::regex errorRegex(R"(duplicate symbol '(.*)' in:((?:.*\s*)+))"); //REGEX FOR LINKER ERRROR PARSING
    if(std::regex_search(content, matches, mainLinkerErrorRegex)){//main linker error parsing
        std::string duplicateSymbol = matches[1].str(); //TODO - differentiate between content from main linker error and forward declaration error
        std::string filePaths = matches[2].str();

        // Split file paths
        std::istringstream stream(filePaths);
        std::string path;
        while (std::getline(stream, path)) {
            nlohmann::json errorJson;
            errorJson["Error Description"] = "duplicate symbol";
            errorJson["Error Detail"] = duplicateSymbol;
            errorJson["File Path"] = path;
            //LINES, COL., SOURCE CODE CONTENT --> NOT INCLUDED FOR LINKER ERRORS, WRITTEN AS NULL IN OUTPUTJOB.CPP
            errorJsonArray.push_back(errorJson);
            }
    }else if(std::regex_search(content, matches, forwardDeclarationErrorRegex)){//forward declaration linker error parsing
        // handling forward declaration error
        //std::string errorDetail = matches[1].str(); // capturing just the undefined symbol name
        nlohmann::json errorJson;
        std::string errorDetail = content;

        errorJson["Error Description"] = "unidentified symbols";
        errorJson["Error Detail"] = errorDetail;
        errorJson["File Path"] = "NULL - not listed for linker error";
        //TODO - Line number and column number added in output.cpp

        errorJsonArray.push_back(errorJson);
    }

}
else {//for non-linker error parsing
```

```
    errorRegex = std::regex(R"((.*):(\d+):(\d+): (\w+): (.+))");
    if (std::regex_search(content, matches, errorRegex)) {
        std::string filePath = matches[1].str();
        int lineNumber = std::stoi(matches[2].str());
        int columnNumber = std::stoi(matches[3].str());
        std::string errorType = matches[4].str();
        std::string errorDescription = matches[5].str();

        nlohmann::json errorJson;
        errorJson["File Path"] = filePath;
        errorJson["Line Number"] = lineNumber;
        errorJson["Column Number"] = columnNumber;
        errorJson["Error Type"] = errorType;
        errorJson["Error Description"] = errorDescription;

        errorJsonArray.push_back(errorJson);
    }
}

    //END OF FILE PARSING & WRITING
    count++;
}
```

This linker error and nonlinker error approach allows the program to parse linker errors and other errors/warnings without erroring out.

Possible improvements for ParsingJob:

In the future, this section can be split up into encapsulated functions for better readability and use. I could even change my outputJob to read from my dequeue stream in main.cpp, which would mean that parsingJob would no longer have to write to json files. This would cut the fat out of this parsing function and allow it to just focus on parsing the content for outputJob to take and write to a json file.

Demonstration of Compile System:

My program uses a makefile to run my code. My job system code is in my code folder and the code I am testing parsing errors from is in my test-code folder. I am in the root directory when I run my makefile, and these folder are located in ./Code and ./test-set respectively. My Program uses a make file designed as such:

```
Make Compile:
        Clang++ -g -std=c++14 ./Code/*.cpp -o output
        ./output
Make Automated:
        Clang++ -g -std=c++14 ./test-code/*.cpp -o test-output
        ./test-output
```

This make file in my root directory ensures that I can run “make automated” to just see the errors/warnings that will be parsed without running my job system, for testing purposes. The “make compile” command runs my job system, and compile job proceeds to run “make automated” and grab the errors/warnings from the terminal, then pass them to parseJob and outputJob.

Here is the terminal output when running “make automated” with errors/warnings:

```
clang++ -g -std=c++14 ./test-code/*.cpp -o test-output
./test-code/test-set-one.cpp:6:4: error: expected expression
   << "test_set_one" << std::endl
   ^
1 error generated.
./test-code/test-set-three.cpp:1:2: error: invalid preprocessing directive
#incl
 ^
./test-code/test-set-three.cpp:4:5: error: use of undeclared identifier 'std'
    std::cout << "test_set_three" << std::endl;
    ^
./test-code/test-set-three.cpp:4:38: error: use of undeclared identifier 'std'
    std::cout << "test_set_three" << std::endl;
                                     ^
3 errors generated.
./test-code/test-set-two.cpp:7:1: error: expected '}'
^
./test-code/test-set-two.cpp:4:20: note: to match this '{'
void test_set_two(){
                   ^
1 error generated.
make: *** [automated] Error 1
```

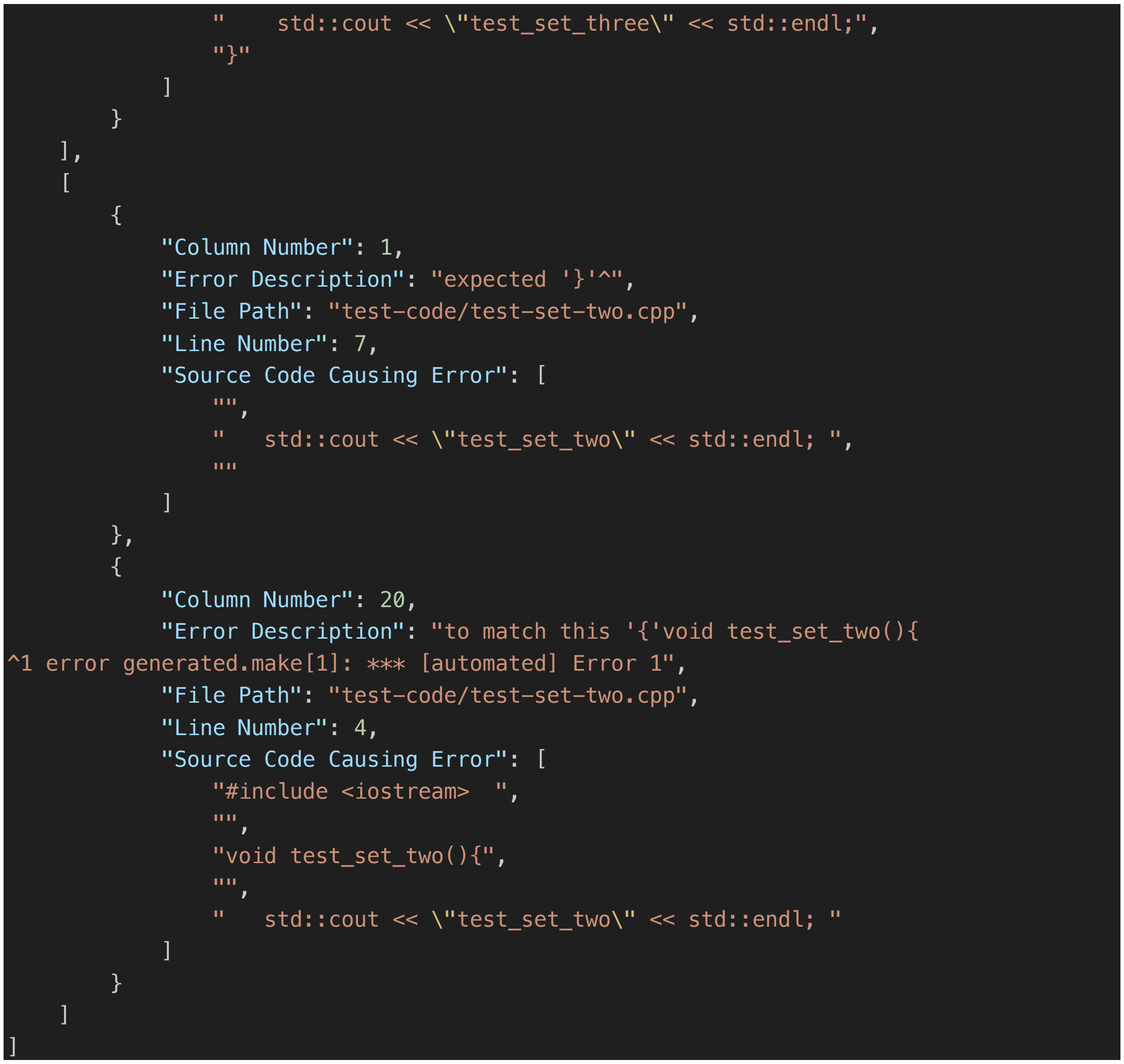
Here is the json output (both errors.json and output.json) when running “make compile” with errors/warnings:

errors.json (intermediary file, solely for outputJob read in):

```
[
    {
        "Column Number": 4,
        "Error Description": "expected expression   << \"test_set_one\" << std::endl
^1 error generated.",
        "Error Type": "error",
        "File Path": "test-code/test-set-one.cpp",
        "Line Number": 6
    },
    {
        "Column Number": 2,
        "Error Description": "invalid preprocessing directive#incl ^",
        "Error Type": "error",
        "File Path": "test-code/test-set-three.cpp",
        "Line Number": 1
    },
    {
        "Column Number": 5,
        "Error Description": "use of undeclared identifier 'std'    std::cout <<
\"test_set_three\" << std::endl;    ^",
        "Error Type": "error",
        "File Path": "test-code/test-set-three.cpp",
        "Line Number": 4
    },
    {
        "Column Number": 38,
        "Error Description": "use of undeclared identifier 'std'    std::cout <<
\"test_set_three\" << std::endl;                                     ^3 errors
generated.",
        "Error Type": "error",
        "File Path": "test-code/test-set-three.cpp",
        "Line Number": 4
    },
    {
        "Column Number": 1,
        "Error Description": "expected '}'^",
        "Error Type": "error",
        "File Path": "test-code/test-set-two.cpp",
        "Line Number": 7
    },
    {
        "Column Number": 20,
        "Error Description": "to match this '{'void test_set_two(){
^1 error generated.make[1]: *** [automated] Error 1",
        "Error Type": "note",
        "File Path": "test-code/test-set-two.cpp",
        "Line Number": 4
    }
]
```

output.json (final output JSON for sorting errors/warn. By file):

```
[
    [
        {
            "Column Number": 4,
            "Error Description": "expected expression   << \"test_set_one\" << 
std::endl   ^1 error generated.",
            "File Path": "test-code/test-set-one.cpp",
            "Line Number": 6,
            "Source Code Causing Error": [
                "",
                "int main(){",
                "   << \"test_set_one\" << std::endl",
                "    //another_function();",
                ""
            ]
        }
    ],
    [
        {
            "Column Number": 2,
            "Error Description": "invalid preprocessing directive#incl ^",
            "File Path": "test-code/test-set-three.cpp",
            "Line Number": 1,
            "Source Code Causing Error": [
                "#incl",
                "#include \"test-set-three.h\"",
                "void test_set_three() {"
            ]
        },
        {
            "Column Number": 5,
            "Error Description": "use of undeclared identifier 'std'    std::cout << 
\"test_set_three\" << std::endl;    ^",
            "File Path": "test-code/test-set-three.cpp",
            "Line Number": 4,
            "Source Code Causing Error": [
                "#include \"test-set-three.h\"",
                "void test_set_three() {",
                "    std::cout << \"test_set_three\" << std::endl;",
                "}"
            ]
        },
        {
            "Column Number": 38,
            "Error Description": "use of undeclared identifier 'std'    std::cout << 
\"test_set_three\" << std::endl;                                     ^3 errors 
generated.",
            "File Path": "test-code/test-set-three.cpp",
            "Line Number": 4,
            "Source Code Causing Error": [
                "#include \"test-set-three.h\"",
                "void test_set_three() {",
```



```
                "    std::cout << \"test_set_three\" << std::endl;",
                "}"
            ]
        }
    ],
    [
        {
            "Column Number": 1,
            "Error Description": "expected '}'^",
            "File Path": "test-code/test-set-two.cpp",
            "Line Number": 7,
            "Source Code Causing Error": [
                "",
                "   std::cout << \"test_set_two\" << std::endl; ",
                ""
            ]
        },
        {
            "Column Number": 20,
            "Error Description": "to match this '{'void test_set_two(){
^1 error generated.make[1]: *** [automated] Error 1",
            "File Path": "test-code/test-set-two.cpp",
            "Line Number": 4,
            "Source Code Causing Error": [
                "#include <iostream>  ",
                "",
                "void test_set_two(){",
                "",
                "   std::cout << \"test_set_two\" << std::endl; "
            ]
        }
    ]
]
```

Here is the terminal output when running “make automated” with no errors/warnings:

```
clang++ -g -std=c++14 ./test-code/*.cpp -o test-output
./test-output
test_set_one
test_set_two
test_set_three
```

Here is the json output (both errors.json and output.json) when running “make compile” with no errors/warnings:

errors.json(intermediary file, solely for outputJob read in): []

the errors.json is left blank because the file is an intermediary file that stores data for the outputjob to read from. When compilejob does not find any errors/warnings, the stream queue is blank so nothing reaches parsingjob which means nothing is written to the errors.json.

output.json(final output JSON for sorting errors/warn. By file):

```
[
    {
        "Error Description": "No errors or warnings - compiled successfully"
    }
]
```

The output.json reads in the content from the errors.json file into an errors json array object. This errors json object is checked for content, and in this case the object is blank, so the output.json writes a "No errors or warnings" statement to the json file.

This code snippet can be seen here:

```
//Generic file path for errors.json
    std::ifstream inputFile("Data/errors.json");
    json errors = json::parse(inputFile);
    inputFile.close();

    //if errors is empty, then write "No errors/warnings" to output.json
    if (errors.empty()) {
        json outputJson = json::array();
        json errorJson;
        errorJson["Error Description"] = "No errors or warnings - compiled
successfully";
        outputJson.push_back(errorJson);

        std::ofstream outputFile("Data/output.json");
        outputFile << outputJson.dump(4); // 4 spaces for pretty printing
        outputFile.close();

        // //delete errors.json file
        //std::remove("Data/errors.json");
        return;
    }
```

Output Explanation

Once outputjob reads in the contents of the errors.json file that parsingjob created, otuputjob stores this content into a json errors object. This errors json object is an array that holds each object, and is iterated through. The error description is checked to determine if the error is a linker error or a non-linker error. If the error is a linker error, then the linkererror function is called, but if the error is a nonlinker error than the nonlinker error function is called. These writing functions have to be different to accommodate the different elements that must be written to the json output file because linker errors have different error attributes compared to most errors. Both functions create a json object and a map. The errors json object is iterated through and contents are assigned to the json object, and then this json object is pushed back to the map. The map is then converted to a json array called output.json. Finally, this array is dumped to a json file.

Here is a code snippet from the nonlinker error section:

```cpp
// Using a map to organize errors by file path.
    std::map<std::string, json> outputMap;

    for (json& error : errors) {
        std::string filePath = error["File Path"];
        int lineNumber = error["Line Number"];
        int columnNumber = error["Column Number"];
        auto surroundingLines = getSurroundingLines(filePath, lineNumber);

        json errorOutput;
        errorOutput["File Path"] = filePath;
        errorOutput["Line Number"] = lineNumber;
        errorOutput["Column Number"] = columnNumber;
        errorOutput["Error Description"] = error["Error Description"];
        errorOutput["Source Code Causing Error"] = surroundingLines;

        // Inserting errorOutput into the map.
        outputMap[filePath].push_back(errorOutput);
    }

    // Convert the map to a json array.
    json outputJson = json::array();
    for (auto& pair : outputMap) {
        outputJson.push_back(pair.second);
    }

    //write JSON to a file
    std::ofstream outputFile("Data/output.json");
    outputFile << outputJson.dump(4); // 4 spaces for pretty printing
    outputFile.close();
```

Overview of Test-Code files:

I set up my files that I am intentionally messing with to cause errors in a file called “test-code”. This file holds a main.cpp file and three other source files called “test-set-one.cpp”, “test-set-two.cpp”, and “test-set-three.cpp”. Having a main.cpp for my test files lets me quickly spin up tests and control what is running when, which gives me finer control over my test cases.

Main.cpp (main.cpp to run the test-set files, NOT main.cpp to run the job system):

```
// main.cpp
#include "test-set-one.h"
#include "test-set-two.h"
#include "test-set-three.h"

int main() {
    test_set_one();
    // Call function from test-set-two.cpp here
    test_set_two();
    // Call function from test-set-three.cpp here
    test_set_three();

    return 0;
}
```

Test-set-# files:
These test files are just dummy files structured as such:

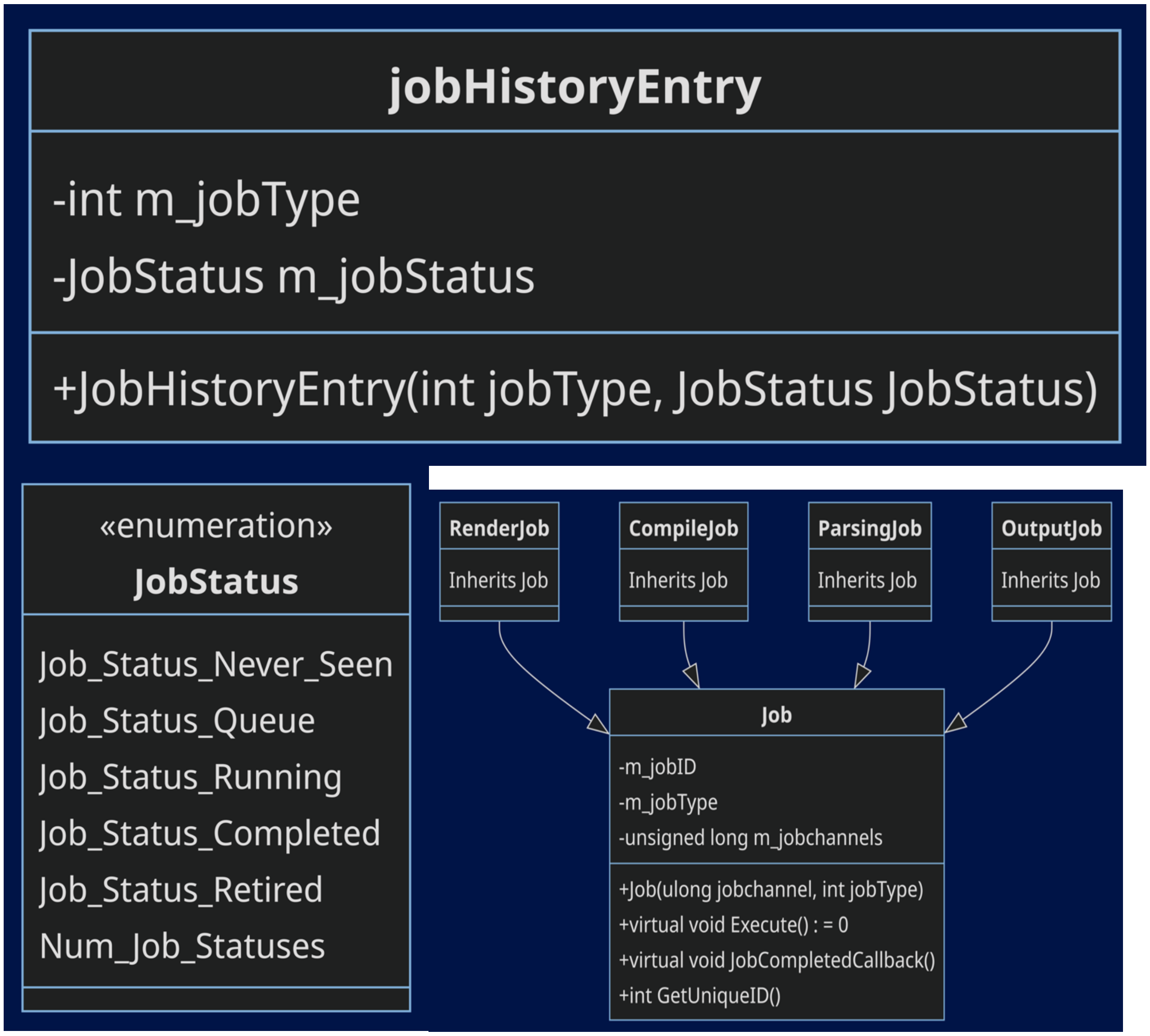
```
#include <iostream>
#include "test-set-three.h"
void test_set_three() {
    std::cout << "test_set_three" << std::endl;
}
```

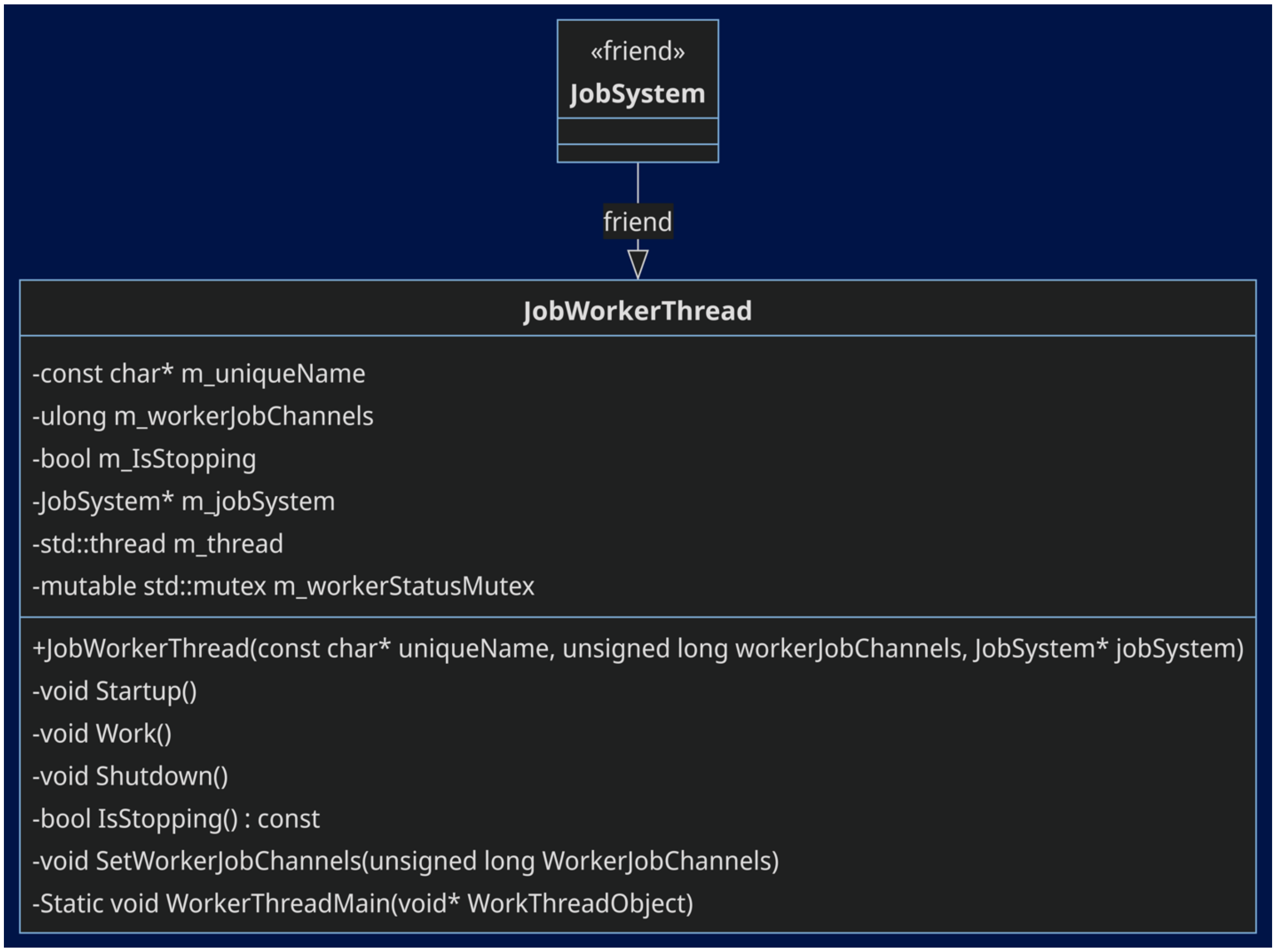
Conclusion

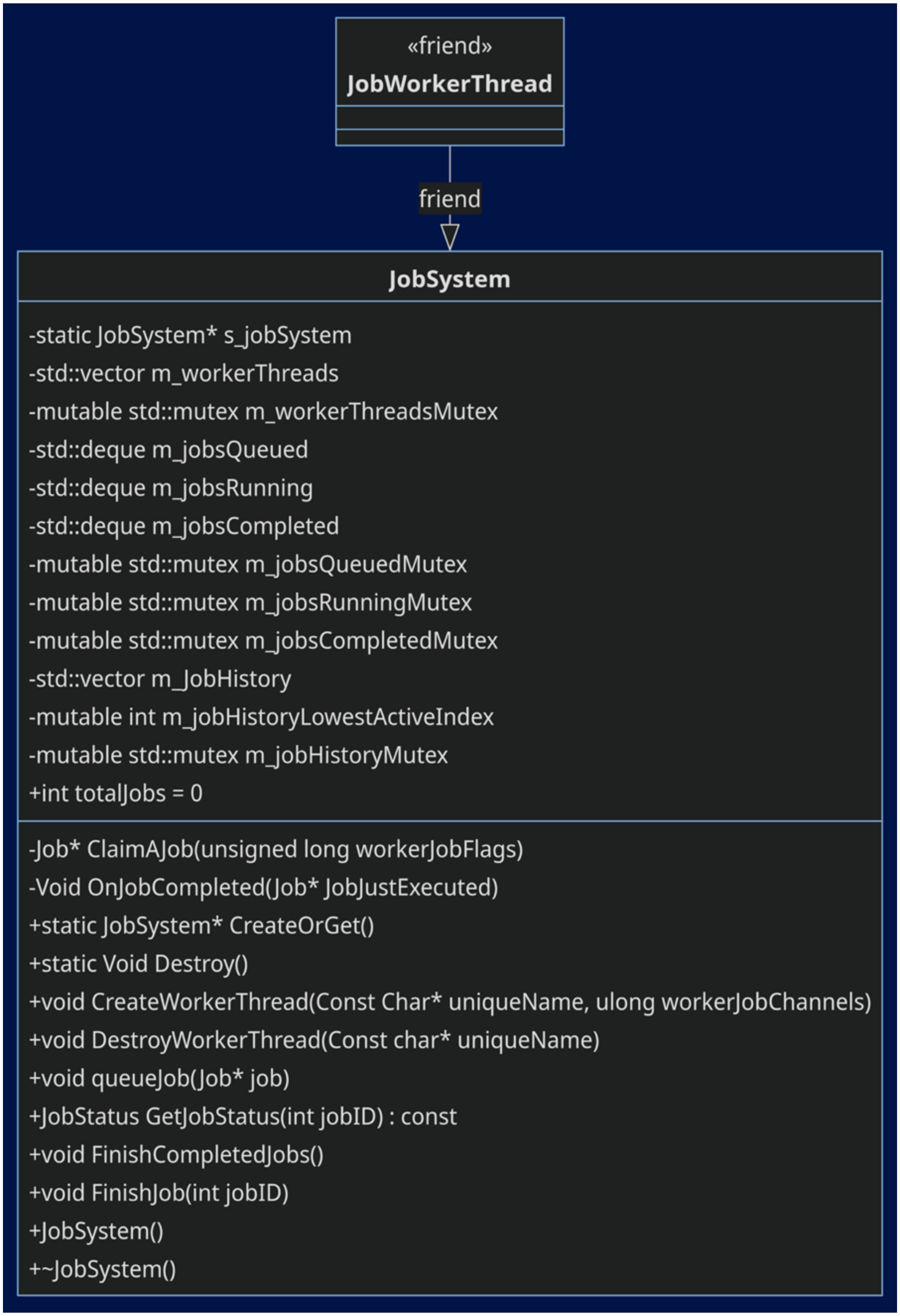
My program encapsulates the job management system with the use of the job, jobsystem, JobWorkerThread classes. The JobWorkerThread has access to the jobsystem properties because the JobWorkerThread is a friend class of the jobsystem. The jobsystem handles the bulk of the thread management, with deques being used to move threads from queued status to running status to completed status. Throughout all of these status changes, the jobsystem updates the thread history using mutex safe protocols in the jobsystem class. This ensures that
The main thread cannot access a data at the same time a child thread is accessing data. The jobWorkerThread class has a number of helper functions to have child threads do work, set their channels, shut them down, etc. This encapsulation keeps the program safe from read write conflicts that cause data to be misinterpreted, and offers a jobsystem that can be easily expanded on in the future.

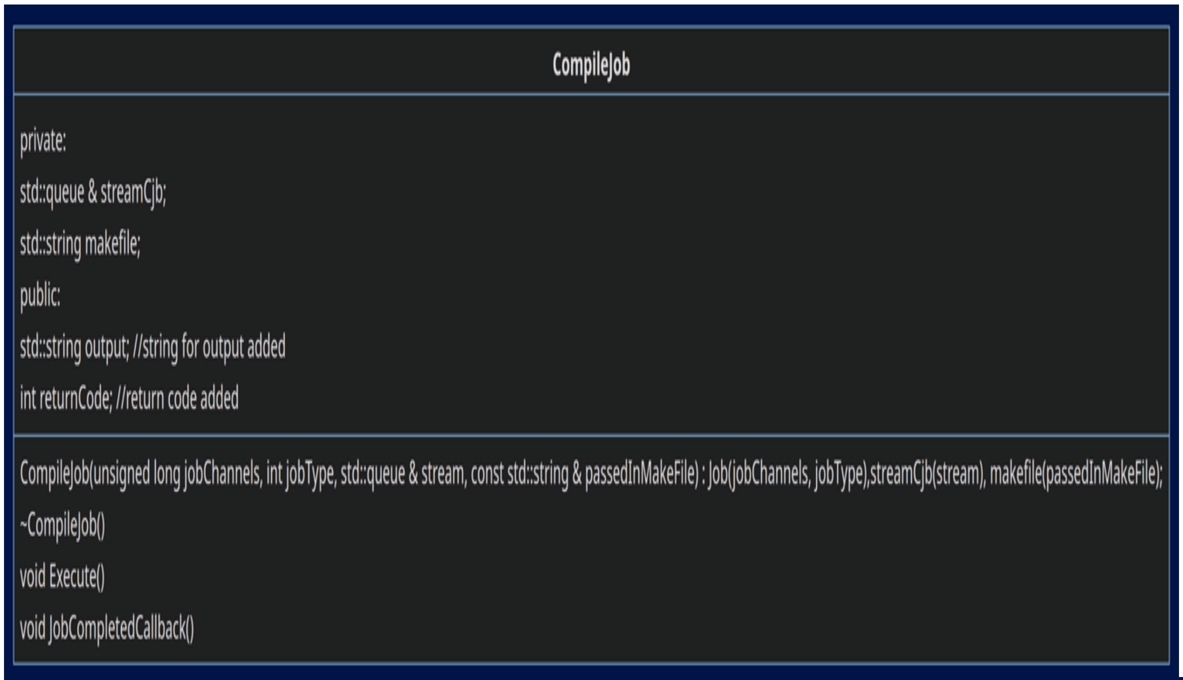
UML Diagrams for the Job System:

NOTE: Render Jobs was not given a UML Diagram as the class is not used in the program, and was only for high volume job testing.


jobHistoryEntry
-int m_jobType
-JobStatus m_jobStatus
+JobHistoryEntry(int jobType, JobStatus JobStatus)
«enumeration»
JobStatus
Job_Status_Never_Seen
Job_Status_Queue
Job_Status_Running
Job_Status_Completed
Job_Status_Retired
Num_Job_Statuses
RenderJob
Inherits Job
CompileJob
Inherits Job
ParsingJob
Inherits Job
OutputJob
Inherits Job
Job
-m_jobID
-m_jobType
-unsigned long m_jobchannels
+Job(ulong jobchannel, int jobType)
+virtual void Execute() : = 0
+virtual void JobCompletedCallback()
+int GetUniqueID()


«friend»
JobSystem
friend
JobWorkerThread
-const char* m_uniqueName
-ulong 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)
-void Startup()
-void Work()
-void Shutdown()
-bool IsStopping() : const
-void SetWorkerJobChannels(unsigned long WorkerJobChannels)
-Static void WorkerThreadMain(void* WorkThreadObject)


«friend»
JobWorkerThread
friend
JobSystem
-static JobSystem* s_jobSystem
-std::vector m_workerThreads
-mutable std::mutex m_workerThreadsMutex
-std::deque m_jobsQueued
-std::deque m_jobsRunning
-std::deque m_jobsCompleted
-mutable std::mutex m_jobsQueuedMutex
-mutable std::mutex m_jobsRunningMutex
-mutable std::mutex m_jobsCompletedMutex
-std::vector m_JobHistory
-mutable int m_jobHistoryLowestActiveIndex
-mutable std::mutex m_jobHistoryMutex
+int totalJobs = 0
-Job* ClaimAJob(unsigned long workerJobFlags)
-Void OnJobCompleted(Job* JobJustExecuted)
+static JobSystem* CreateOrGet()
+static Void Destroy()
+void CreateWorkerThread(Const Char* uniqueName, ulong workerJobChannels)
+void DestroyWorkerThread(Const char* uniqueName)
+void queueJob(Job* job)
+JobStatus GetJobStatus(int jobID) : const
+void FinishCompletedJobs()
+void FinishJob(int jobID)
+JobSystem()
+~JobSystem()



CompileJob

private:

std::queue & streamCjb;

std::string makefile;

public:

std::string output; //string for output added

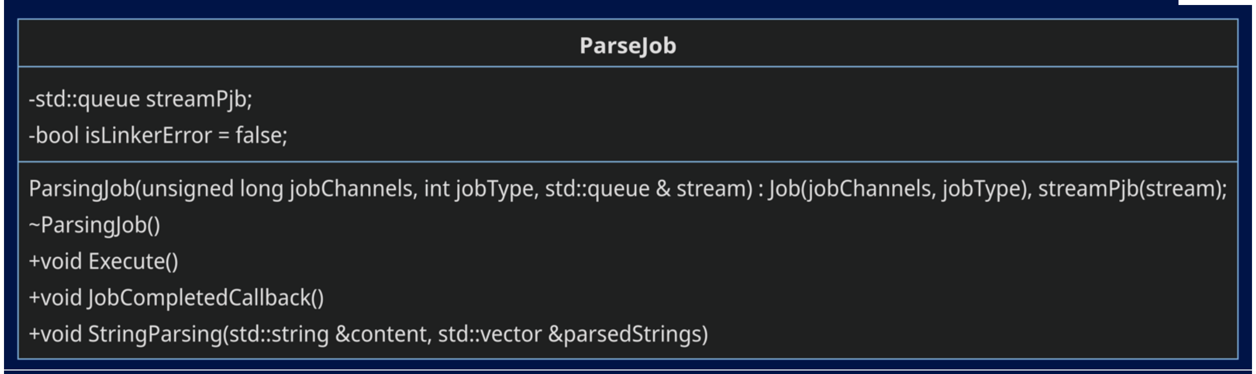
int returnCode; //return code added

CompileJob(unsigned long jobChannels, int jobType, std::queue & stream, const std::string & passedInMakeFile) : Job(jobChannels, jobType),streamCjb(stream), makefile(passedInMakeFile);

~CompileJob()

void Execute()

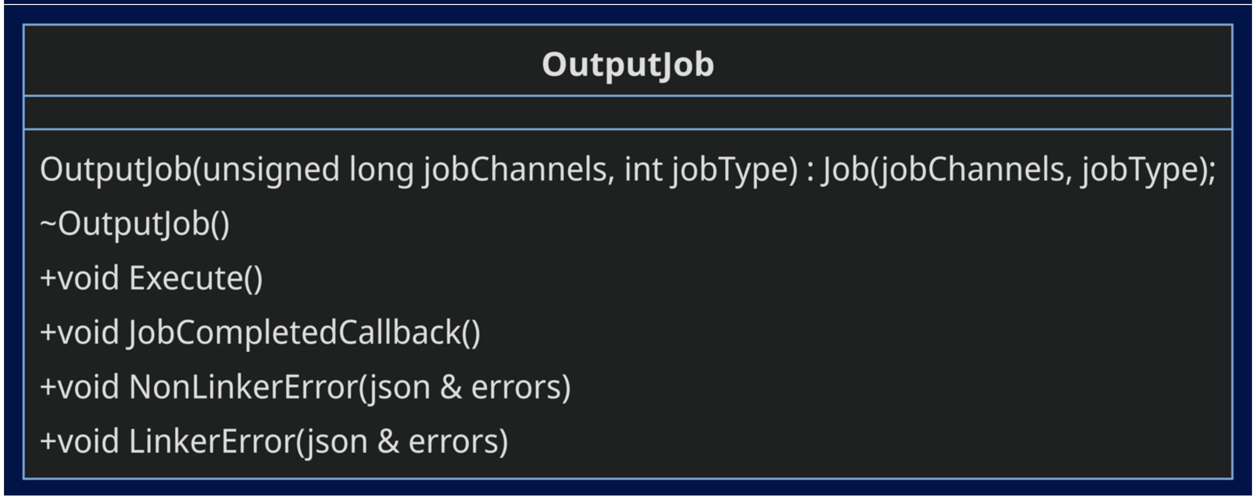
void JobCompletedCallback()



ParseJob

-std::queue streamPjb;
-bool isLinkerError = false;

ParsingJob(unsigned long jobChannels, int jobType, std::queue & stream) : Job(jobChannels, jobType), streamPjb(stream);
~ParsingJob()
+void Execute()
+void JobCompletedCallback()
+void StringParsing(std::string &content, std::vector &parsedStrings)



OutputJob

OutputJob(unsigned long jobChannels, int jobType) : Job(jobChannels, jobType);
~OutputJob()
+void Execute()
+void JobCompletedCallback()
+void NonLinkerError(json & errors)
+void LinkerError(json & errors)