# Solution Summary

For almost every project, I create a document like this. I want to document my thoughts, questions, answers, etc so that I can revisit them if necessary. Sure, for production work, the requirements and full design may be put into separate docs; but I typically start with a document like this. It allows me to iterate on design, etc without being bogged down with code details.

Note that the contents of this document other than the Solution Summary are my raw contents. I didn’t pretty them up.

## Implementation Facts/Decisions/Assumptions

* I wanted to illustrate other aspects of coding that are important to me – debug logging and unit testing.
  + Good debug logging is important when supporting code running in the field; it can save the day. I always attempt to log errors/etc. And I attempt to log detailed debug-level info to aid in field diagnosis.
  + In the past, for unit testing C/C++, I’ve used CppUnit and CppTest. I like CppTest better than CppUnit.
  + Since the initial requirements didn’t specify debug logging, I’m assuming that debug logging was not necessary for this coding exercise.
  + In the code, where I’d normally log a message, I’ve outputted the message to stdout and added a TODO for the debug logging. This made the code noisy. I wrote a utility log function to wrap that all up to make the code a bit cleaner.
* I used a git repo to manage code changes. The GitHub repo is <https://github.com/dougbeyer/ComScheduledTask.git> .
* For a very long time now, I’ve used a separate code editor for my editing. It has very powerful template facilities. Every time I start using a new language, I build up extensive templates for that language. I attempt to put “best practices”, etc directly into the templates. C/C++ is no exception. So, when you see things like source code file layout, class layout, template typdefs, etc, that’s all coming from my templates. I don’t have to think about syntax and I certainly don’t type out every line of code. Enter a few characters, hit the space bar, and voila, there’s my file, statement, etc.
* I chose to use TCHAR even though current MSDN reading says to always use widechar. But my templates all use TCHAR. So rather than worry about it, I just continued to use TCHAR. Of course, for production code, I’d use whatever standards Cofense has in place.
* I didn’t spend any time worrying about extensive use of namespaces. I just used the same namespace for all classes.
* ~~I’m bouncing between integer sizes of size\_t and unsigned long. They are NOT the same size. It would have been cleaner to just stick with unsigned long. But most of the STL classes/methods use size\_t. That’s where my use of size\_t came from. I attempted to be careful when manipulating them but obviously this is an area for bugs.~~
* Note that, in general, I prefer functions to return error codes. It’s easy to extend the range of info they return simply by defining more error codes. But, at the same time, I’m inclined to write my functions first using a bool return value to indicate success/failure; it’s easier initially compared to defining return codes. I then refactor as needed if I later determine that a function needs to return more than two possible values.
* I am an advocate of using goto’s within a single function for the sole purpose of avoiding deeply-nested IFs. My goto’s *only* drop down; no weird cases of jumping all over the place.

## Running the code

1. Unzip the source code.
2. Build it using VStudio 2015 Update 3.
   1. TODO – Specify having built cpputest to the correct x86/x64, defining the CPP\_UNIT\_HOME env var, adding this to linker props $(CPP\_UNIT\_HOME)\lib, adding $(CPP\_UNIT\_HOME)\include to compiler props, having built the cppunit\_dll project for both release and debug, ensuring that the release and debug versions of cppunitd\_dll.dll and cppunitd\_dll.lib are in the $(CPP\_UNIT\_HOME)\lib directory.
3. Verify that the Task Scheduler service is running.
   1. TODO – Tell how to do this.
4. Assuming UAC is enabled, administrator access is required to run the apps in order create/open the file mapping.
   1. To run either the producer or consumer app in VStudio, run VStudio as an administrator.
   2. To run either the producer or consumer app as a standalone exe or from a command prompt, their application manifests use runas = administrator. So, UAC will elevate as required.
5. Run the app with the following command line arguments:
   1. Consumer – “Consumer.exe <Num shared buffers> [substring to search for]”. The substring is optional. If the substring is omitted, all strings are processed.
   2. Producer – “Producer.exe <Num shared buffers> <input batch size> <sentence input file path>”. The input batch size is how many sentences to read from the input file in one batch; it’s to prevent resource problems with a HUGE input file.
6. TODO – Describe the command line and GUI interfaces that Windows provides for the Task Scheduler.

# 

# Summary Points of Interest

1. ~~Per Colin McCambridge Colin Mccambridge (CM), debug logging and unit testing are not necessary for this coding exercise.~~
   1. ~~In the code, where I’d normally log an error, I’ve outputted the error to stdout and added a TODO for the debug logging. This makes the code noisy. I wrote a utility log function to wrap that all up to make the code a bit cleaner.~~
   2. ~~Be aware though that it feels very weird to me not be writing debug logging and unit testing at the same time as I’m writing the code. But I’m going to skip that here to maximize time efficiency and to focus on the core coding problem.~~
2. ~~Do I want to talk about my being, in general, not a fan of exceptions? Is it too old school?~~
3. ~~State that I didn’t spend any time worrying about proper namespaces. I just used the same namespace for all classes.~~
4. ~~Comment on that I added the batch size cmd line arg to the Producer app.~~
   1. ~~We were told to favor throughput. File I/O doesn’t help throughput. So, we could just read the entire input file and then process the sentences. However, we were told that there could be any number of sentences in the input file. Hence to avoid potentially running out of memory with a very large number of sentences, we’ll implement a max number of sentences that can be read in one batch. After that, we’ll process the batch. Then go back and read in another batch.~~
5. ~~I used lipsum.com and randomwordgenerator.com to generate random sentences. This was handy to quickly create many sentences.~~
6. ~~Comment on my use of TCHAR. Current MSDN says to always use widechar. But my templates all are using tchar. So rather than worry about it, I just continued to use tchar. Of course, for production code, I’d use whatever standards CrowdStrike has in place.~~
7. ~~Comment on how I prefer to use “goto” within a single function to drop down rather than deeply-nested IFs.~~
8. ~~Make note to run VStudio (or the exe’s) as admins to create or open the file mapping.~~
9. ~~I’m bouncing between integer sizes of size\_t and unsigned long. They are NOT the same size. It would have been cleaner to just stick with unsigned long. But most of the STL classes/methods use size\_t. That’s where my use of size\_t came from. I attempted to be careful when manipulating them but obviously this is an area for bugs.~~
10. ~~Note that, in general, I prefer functions to return error codes. It’s easy to extend the range of info they return simply by defining more error codes. But, at the same time, I’m inclined to write my functions first using a bool return value to indicate success/failure; it’s easier initially compared to defining return codes. I then refactor as needed if I determine that a function needs to return more than two possible values.~~
11. ~~This exercise is made up of two short-lived processes. One of the benefits of using short-lived processes to do tasks is that memory cleanup is not critical – it will all get recovered when the process dies.~~
12. ~~From inspecting the results, it appears that the consumer is not processing the sentences in the exact same order as the producer is writing them. This was not a constraint in the coding exercise instructions.~~
    1. ~~However, it does look like all the produced sentences do get processed. You can verify this by not passing a substring on the consumer command line.~~
13. ~~Note that I implemented the consumer to process every sentence if no substring is passed on the command line.~~
14. ~~For this coding exercise, I have not included any additional info (e.g. version info) in a buffer header.~~

# TODOs

1. ~~Consider dropping debug logging. CM said that was ok. I can leave TODOs where I’d do the logging.~~
2. ~~Per CM, I’m using only ansi strings.~~
   1. ~~TODO – I undef’ed UNICODE and \_UNICODE only to work with spdlog. Since I’ve removed logging, I can remove those undef’s and just use tchar.~~
3. ~~Add log( varargs) function to encapsulate all the COUT and log TODO lines.~~
4. ~~Use const in the various places to make the code more robust!!~~

# Questions

1. Can we assume ansi strings? Widechar?
   1. [CM] ANSI would be fine.
2. Can we assume that we write the strings as we get them from input? Or are you expecting optimizations like sorting the sentences by length, then creating batches by aggregating as many sentences as will fit in one buffer? I wonder if the batching throughput performance would outweigh the sorting and aggregation costs.
   1. NOTE - This question applies whether I end up reading all the sentences from the input file in one read or implement some sort of “size-limited batch read” to protect against an input file that is WAY HUGE.
   2. [CM] We'd like to have the solution scale to arbitrary input file sizes, so doing one read plus a sort is likely to hit resource constraints. On the other hand batching multiple sentences into each buffer is definitely desirable- we'd like to see that sequence of entries format in each buffer.
3. Is an input sentence that is larger than the buffer size invalid? Or must I implement some way of letting a single sentence span buffers?
   1. [CM] Good constraint. We can consider an input sentence that doesn't fit within the buffer format to be invalid. You can handle this (safely) however you'd like- printing an error, just dropping the sentence, etc. It's not necessary to implement buffer spanning.
4. TODO - Must the sentences be processed in the order written?

# Requirements

* 2 applications – one for the producer, one for the consumer.
* The apps use shared memory buffers to communicate.
  + The number of shared buffers is configurable.
  + Each shared buffer is of a fixed length (1024 bytes).
* Producer
  + Reads sentences from an input file. Each line in the file is a sentence.
  + Populates the shared buffers with sentences.
  + Each entry in the shared buffer consists of
    - An unsigned long for the length
    - Followed by the sentence
  + If multiple entries can fit in the buffer, populate the buffer with as many as will fit.
  + The technique used to determine which buffer to populate next is up to me.
  + The producer should print each sentence on a new line to the console.
  + ~~The producer must not crash no matter how an external entity modifies the data in the buffer.~~
  + Will be started before the Consumer and will exit once all sentences are processed by the consumer.
* Consumer
  + Reads the sentences from the shared buffers.
  + Will print to the console all sentences that contain a configurable substring.
  + Must take all necessary precautions to safely process the buffer data. Don’t blindly trust the data.
  + If invalid data is found,
    - The consumer can stop processing that buffer.
    - The consumer must make the buffer available for a potential new sentence from the Producer.
  + The consumer must not crash no matter how an external entity modifies the data in the buffer.
  + The Consumer will be started after the Producer and will not terminate.
* Safe data processing:
  + No valid sentence is lost.
  + Eventually all shared buffers populated with valid sentences are processed.
* Design choices should favor throughput.

# Design

## Assumptions

* They specifically stated the buffer size (i.e. 1024 bytes). Therefore, there’s no reason to make that configurable.
* I’m ignoring any additional buffer header (e.g. version info).

## Binaries

1. Consumer console exe
2. Producer console exe
3. Common.lib – static lib shared by both the producer and consumer. It contains classes like SharedBuffer, SharedBufferMgr, utility functions, etc.

## Classes

1. SharedBuffer
   1. Name (I will define a simple name format (e.g. sharedbufn, where n is the shared buffer number) )
   2. Length (ulong) (1024 bytes)
   3. Buffer (implemented as Windows shared memory)
   4. Lock (probably a named mutex to work across processes) (name it similarly to the SharedBuffer name)
   5. Named event? (when signaled, indicates that it is populated and ready for consumption) (name it similarly to the SharedBuffer name) (If I make the event name identical to the Name, then I can map the SharedBuffer instances in the SharedBufferMgr by name for quick retrieval.
2. SharedBufferMgr
   1. Contains list of SharedBuffer instances
   2. Responsible for creating/opening the shared memory, allocating the proper pointer to each SharedBuffer, etc.
   3. Will abstract the reading and writing of sentences to buffers. This api should take a list of sentences by reference. When writing, the list will provide the sentence data. When reading, it will be populated with the sentences read from the buffer.
      1. Bool write( list<string> & sentences )
         1. Remove sentences that got written. Any sentences left in the list when the call returns must be re-processed by the producer. Or, do I just block till another buffer is available until I’ve written all the sentences?
      2. Bool read ( string eventName (which hopefully will be the SharedBuffer.Name), list<string> & sentences )
   4. Will abstract the algorithm to determine the next buffer to write to.
   5. The constructor will take
      1. The number of SharedBuffer instances to create.
      2. And enum value indicating if it’s the Consumer or Producer instantiating it.
         1. Not sure if this is necessary. I’ll just be sure that each only calls the methods applicable to it.
      3. Should I pass in a callback function from the consumer to provide push notifications of sentences? Or, would it be easier to let the consumer wait on the multiple SharedBuffer events?
3. InputSentences
   1. Encapsulates the reading of the input file.
   2. We were told to favor throughput. File I/O doesn’t help throughput. So, we could just read the entire input file and then process the sentences. However, we were told that there could be any number of sentences in the input file. Hence to avoid potentially running out of memory with a very large number of sentences, we’ll implement a max number of sentences that can be read in one batch. After that, we’ll process the batch. Then go back and read in another batch.
      1. I should make batch size configurable via a cmd line arg.

# Implementation

# Producer

* Main function:
  + Ignore init’ing logging for this coding exercise.
  + Process the cmd line args.
  + Create the shared memory buffers.
  + Read in the input sentences.
  + Write the input sentences to the buffers.
    - Optimize the sentences into batches that fit into a single buffer.
    - Write the batch to a buffer.
    - Write each sentence in the batch to a new line to stdout.
  + When all sentences have been read by the consumer, exit.