2a.

Narration in video.

2b.

My reminders project that I developed independently in C++ was part of an iterative development process consisting of a multitude of features and systems that were developed independent of one another and later incorporated together for the final product. One issue I encountered while developing the reminder system was the method `ReminderSystem::Tick()`. This method's job was to loop infinitely while checking if any reminders should be triggered and trigger them if they do. Though debugging, I found out the infinite loop was deadlocking my program's execution. I was able to mitigate the deadlock by making this method run in a different thread of execution, effectively making it run in the background.

Another issue I encountered was linking the command parsing system to the command job system. After a command string was parsed and its command type and parameters stored in a structure, I realized I had no way to know what job this command was supposed to be redirected to. To solve this issue I decided to use a map `std::map<std::string, std::unique_ptr<CommandJob>> commandJobs;` to associate the command jobs to their command titles allowing me to finally link these two independent systems together.

2c.

```
bool CommandProcessor::ProcessCommand(std::string rawCmd) {

  auto findInString = [ & ](const std::string & haystack, char toFind,
    const size_t startIndex = 0) {
    if (startIndex < 0 || startIndex > haystack.size() + 1)
      return std::string::npos;

    for (size_t i = startIndex; i < haystack.size(); i++) {
      if (haystack[i] == toFind)
        return (size_t) i;
    }
```

```
    return std::string::npos;
  };

  auto getSubstringString = [ & ](const std::string & haystack, size_t
start, size_t end) {
    if (start < 0)
      throw std::invalid_argument("start");

    if (end > haystack.size() + 1)
      throw std::invalid_argument("end");

    std::string resultString = "";

    for (size_t i = start; i < end; i++) {
      resultString += haystack[i];
    }

    return resultString;
  };

  if (rawCmd.empty()) // Command string is empty
    return false;

  auto dashChar = findInString(rawCmd, '-', 0);

  if (dashChar == std::string::npos)
    return false; // Could not find the beginning '-'

  auto nameEndingChar = findInString(rawCmd, ' ', dashChar);

  if (nameEndingChar == std::string::npos)
    nameEndingChar = rawCmd.length();

  rawCommands.push_back(rawCmd);

  Command newCmd;
  newCmd.commandTitle = getSubstringString(rawCmd, dashChar + 1,
nameEndingChar);
  std::transform(newCmd.commandTitle.begin(), newCmd.commandTitle.end(),
newCmd.commandTitle.begin(), ::tolower); // Convert the name to lowercase
as the spec defines.
```

```
  newCmd.commandId = commandCount; // Give command a unique identifier
  commandCount++; // Increase command count.

  auto argSearchIndex = nameEndingChar + 1; // Search for args where we
found the name.

  while ((argSearchIndex = findInString(rawCmd, '$', argSearchIndex)) !=
std::string::npos) {

    // Found another argument

    auto argEnd = findInString(rawCmd, ' ', argSearchIndex);

    if (argEnd == std::string::npos) {
      argEnd = rawCmd.size(); // If we can't find the separating space,
this argument is the last characters in the string.
    }

    auto argStart = argSearchIndex + 1; // Go past the $ prefix.

    auto argument = getSubstringString(rawCmd, argStart, argEnd); // Grab
only the argument out of the text, without the $

    newCmd.commandArgs.push_back(argument); // Add the argument to the
array

    argSearchIndex = argStart; // Next iteration start searching after this
argument.
  }

  return ProcessCommand(newCmd); // Pass parsed command to the actual
method that executes the commands function.
}
```

In the method above that features a major algorithm in my program, I have two notable algorithms that function independently of each other and other algorithms, yet form a very important algorithm that parses the inputted command strings received by the program. Firstly, I

wrote the algorithm `findInString` located at the top of the `CommandProcessor::ProcessCommand` which is tasked to return the index or position of any specified character in any given string. The next algorithm featured in this larger algorithm is also located at the top of the `CommandProcessor::ProcessCommand` method. This algorithm, `getSubstringString`, has the job of returning a portion or *substring* of any inputted string based on inputted bounds. Together, both of these algorithms work together to create a massively important algorithm in my program which is `CommandProcessor::ProcessCommand`. These two smaller algorithms are used on multiple strings that are passed to them, to eventually, along with other algorithms in this method, process any inputted command string and parse its' command name and arguments.

2d.

```
class Reminder {
  std::chrono::time_point < std::chrono::steady_clock > startTime, endTime;
  bool running;
  public:

    Reminder();

  std::chrono::duration < double > GetTimeLeft();

  void Start(std::chrono::nanoseconds timeTillDone);

  void Stop(bool didWholeTimeElapse);

  bool IsRunning();

  std::chrono::time_point < std::chrono::steady_clock > GetStartTime();

  std::chrono::time_point < std::chrono::steady_clock > GetEndTime();

  std::string ConvertTimeToString(std::chrono::time_point <
std::chrono::steady_clock > t);

  virtual void OnDone(std::chrono::duration < double > timeElapsed) = 0;
```

```
  virtual std::string GetDescription() = 0;
};
```

Accompanying source file :

```
Reminder::Reminder() {
  running = false;
}

void Reminder::Start(std::chrono::nanoseconds timeTillDone) {
  running = true;
  startTime = std::chrono::steady_clock::now();
  endTime = startTime + timeTillDone;
}

void Reminder::Stop(bool didWholeTimeElapse) {
  running = false;

  if (!didWholeTimeElapse)
    return;
  else {
    this - > OnDone(endTime - startTime);
  }
}

bool Reminder::IsRunning() {
  return running;
}

std::chrono::time_point < std::chrono::steady_clock >
Reminder::GetStartTime() {
  return startTime;
}

std::chrono::time_point < std::chrono::steady_clock >
Reminder::GetEndTime() {
  return endTime;
}

time_t steady_clock_to_time_t(std::chrono::steady_clock::time_point t) {
```

```
  return
std::chrono::system_clock::to_time_t(std::chrono::system_clock::now() +
std::chrono::duration_cast < std::chrono::system_clock::duration > (t -
std::chrono::steady_clock::now()));
}

std::string Reminder::ConvertTimeToString(std::chrono::time_point <
std::chrono::steady_clock > t) {
  std::time_t time = steady_clock_to_time_t(t);
  std::tm timetm = * std::localtime( & time);
  return std::string(std::ctime( & time));
}

std::chrono::duration < double > Reminder::GetTimeLeft() {
  return endTime - std::chrono::steady_clock::now();
}
```

Shown through the `Reminder` class above is one of the most important and most useful abstractions of my program. This abstraction helps me manage the complexity of the entire reminders system and the program as a whole. Every reminder in my program inherits from this base `Reminder` class and makes the addition of new different types of reminders a very efficient and quick process. This base class handles things like initializing reminder data such as when a reminder starts and when it ends, aswell as implementing methods like one to get a human readable string of when the reminder will end. Apart from these utilities, this base class implements the virtual function `OnDone(...)` that allows any reminder class that inherits from this class to implement this function in its' own class to be called whenever the reminder has been triggered. Apart from this classes methods and variables managing the complexity of the program, the reminder system's code is able to simply only reference this base class when needing to interact with any reminders that exists such as text reminders or sound reminders without ever needing to know what type of reminder they are.