# CIT 5950 Practice Questions

Hello! We know that you want more practice questions for preparing for CIT 5950 Exams, so we have created a collection of Practice problems for you here.

**Note:**

- These questions may vary in difficulty. Some of these questions may be a bit on the easier side, some on the harder side, and some just right.
- There are some topics that are not covered in these practice questions. There was also a request from students about having more C++ coding questions, so we have a lot more of those!
- On the exam we will provide an appendix containing the relevant function descriptions and signatures
- We will try to be more clear in what the question is asking on the exam. We didn't have time to polish all of these questions as much as we would like.
- Ohq is down, multi threaded socket, memory diagram

## C++ Programming - Grayscale

Travis has recently developed a fascination with vintage-style images and can't stop talking about how much better grayscale images look during staff meetings. On his computer, images are currently stored as a 2D vector of pixels, where each pixel contains red, green, and blue channel values. The pixel structure is defined as:

```
struct pixel {
    int red;
    int blue;
    int green;
};
```

Travis now wants to convert all of his images into grayscale using a new format he calls travis_pixel, which stores only a single grayscale value per pixel:

```
struct travis_pixel {
    int grayscale;
};
```

To compute the grayscale value, Travis uses the following formula:

$$grayscale = \frac{red + green + 3 * blue}{3}$$

Given an image represented as a `vector<vector<pixel>>`, write a function to convert it into a `vector<vector<travis_pixel>>` using the formula above.

(Please put your answer onto the next page)

You probably do not need all of this space:

```
vector<vector<travis_pixel>> convert_to_grayscale(
    const vector<vector<pixel>>& image) {

  vector<vector<travis_pixel>> result;
  for (const auto& row : image) {
    vector<travis_pixel> gray_row;
    for (const auto& px : row) {
      int gray_value = px.red + px.green + px.blue * 3;
      gray_row.push_back({gray_value});
    }
    result.push_back(gray_row);
  }
  return result;
}
```

## C++ Programming - Image Compression

Andrew loves compression algorithms. He's working on a simple way to compress binary images or data represented as 2D vectors of 0s and 1s. To reduce memory usage, he wants to compress the image by storing only the indices of the 1s in each row. For example, instead of storing an entire row like [0, 1, 0, 0, 1], he just wants to store [1, 4] — the indices where 1 occurs.

Write a function that takes in a `vector<vector<int>>` representing a binary image and returns a `vector<vector<int>>` where each inner vector contains the indices of 1s in that row.

```
vector<vector<int>> compress_ones(const vector<vector<int>>& binary_image) {


  vector<vector<int>> compressed;
  for (const auto& row : binary_image) {
    vector<int> indices;
    for (int i = 0; i < row.size(); ++i) {
      if (row[i] == 1) {
        indices.push_back(i);
      }
    }
    compressed.push_back(indices);
  }
  return compressed;
}
```

## Club Popularity by Category (C++ Coding)

As part of a campus-wide initiative, Penn is collecting data about which clubs students are interested in joining. Each club falls under a broader category — for example, "Penn AI" under "Technology", or "Penn Dance Collective" under "Arts".

The data has been collected in the following structure:

map<string, map<string, string>> student_club_interests;

Where:

- The key is the student's name (e.g., "Alice").
- The value is an unordered map representing:
  - club category → specific club name (topic).

For example:

```
map<string, map<string, string>> student_club_interests = {
  { "Alice", {
    { "Technology", "Penn AI" },
    { "Arts", "Penn Dance Collective" }
  }},
  { "Bob", {
    { "Technology", "Penn Robotics" },
    { "Arts", "Penn Dance Collective" }
  }},
  { "Carol", {
    { "Technology", "Penn AI" },
    { "Arts", "Penn Music Society" }
  }},
    { "David", {
      { "Technology", "Penn AI" }
  }}
};
```

Your task is to write a function named get_most_popular_topic_per_category:

map<string, string> get_most_popular_club_per_category(
const map<string, map<string, string>>& student_club_interests);

The function receives a structure described on the previous page. The function should return an unordered map where each category is in the result as a key and the value is the most popular club in that category, determined by the number of students interested.

For the example structure provided on the first page, you can assume that the returned value for this function would be:

```
{
  { "Technology", "Penn AI" },
  { "Arts", "Penn Dance Collective" }
}
```

You can assume there are no ties among clubs in the same category and you do not need to handle different letter casing (uppercase vs lowercase)

```
map<string, string>
get_most_popular_club_per_category(
const map<string, map<string, string>>& student_club_interests) {
  // one possible solution

  // Count how many students are interested in each topic
  // per category. Could use unordered_map but space is tight here.
  map<string, map<string, int>> category_topic_counts;

  for (auto& student_entry : student_club_interests) {
      const auto& interests = student_entry.second;
      for (auto& interest_entry : interests) {
        const string& category = interest_entry.first;
        const string& topic = interest_entry.second;
        category_topic_counts[category][topic]++;
      }
  }

  // For each category, find the most popular topic
  map<string, string> result;

  for (const auto& category_entry : category_topic_counts) {
      const string& category = category_entry.first;
      const auto& topic_counts = category_entry.second;

      int max_count = 0;
      string most_popular_topic;

      for (const auto& topic_entry : topic_counts) {
        const string& topic = topic_entry.first;
        int count = topic_entry.second;

        if (count > max_count || (count == max_count && topic <
                                              most_popular_topic)) {
            max_count = count;
          most_popular_topic = topic;
```

```
        }
      }
      result[category] = most_popular_topic;
  }
  return result;
}
```

## Penn LEGO Builders' Club (Threads/Deadlock)

Penn students are collaborating to build complex LEGO structures using different colored bricks. To prevent collisions during multi-threaded construction, each color requires a mutex lock before building with it. There are three common composite structures, each built using two colors:

```
#include <pthread.h>

// these three mutexes are initialized
pthread_mutex_t green_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t red_lock   = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t blue_lock  = PTHREAD_MUTEX_INITIALIZER;

void build_red_green() {
      pthread_mutex_lock(&red_lock);
      pthread_mutex_lock(&green_lock);
      // Build red → green structure
      pthread_mutex_unlock(&green_lock);
      pthread_mutex_unlock(&red_lock);
}

void build_green_blue() {
      pthread_mutex_lock(&green_lock);
      pthread_mutex_lock(&blue_lock);
      // Build green → blue structure
      pthread_mutex_unlock(&blue_lock);
      pthread_mutex_unlock(&green_lock);
}


void build_blue_red() {
      pthread_mutex_lock(&blue_lock);
      pthread_mutex_lock(&red_lock);
      // Build blue → red structure
      pthread_mutex_unlock(&red_lock);
      pthread_mutex_unlock(&blue_lock);
}
```

Assume these three build functions may be called concurrently by different student threads, in any order and with arbitrary interleaving.

This question continues onto the next page

Why don't these functions always work correctly in a multi-threaded environment?
How would you fix the code to solve this problem? Please justify your answer.

This code may deadlock when three threads attempt the following simultaneously:

- Thread A runs build_red_green() → acquires red, waits for green
- Thread B runs build_green_blue() → acquires green, waits for blue
- Thread C runs build_blue_red() → acquires blue, waits for red

Now we have a circular wait:

- Thread A: holds red, waiting on green
- Thread B: holds green, waiting on blue
- Thread C: holds blue, waiting on red

To fix: choose one consistent lock order, e.g.: green > blue > red

```c
#include <pthread.h>

 pthread_mutex_t green_lock = PTHREAD_MUTEX_INITIALIZER;
 pthread_mutex_t red_lock   = PTHREAD_MUTEX_INITIALIZER;
 pthread_mutex_t blue_lock  = PTHREAD_MUTEX_INITIALIZER;

 void build_red_green() {
     pthread_mutex_lock(&green_lock);
      pthread_mutex_lock(&red_lock);
      // Build red → green structure
     pthread_mutex_unlock(&red_lock);
     pthread_mutex_unlock(&green_lock);
 }

 void build_green_blue() {
     pthread_mutex_lock(&green_lock);
      pthread_mutex_lock(&blue_lock);
      // Build green → blue structure
     pthread_mutex_unlock(&blue_lock);
     pthread_mutex_unlock(&green_lock);
 }

 void build_blue_red() {
      pthread_mutex_lock(&blue_lock);
      pthread_mutex_lock(&red_lock);
      // Build blue → red structure
     pthread_mutex_unlock(&red_lock);
     pthread_mutex_unlock(&blue_lock);
 }
```

Alternatively, break the nested structure -

```c
#include <pthread.h>

 pthread_mutex_t green_lock = PTHREAD_MUTEX_INITIALIZER;
 pthread_mutex_t red_lock   = PTHREAD_MUTEX_INITIALIZER;
 pthread_mutex_t blue_lock  = PTHREAD_MUTEX_INITIALIZER;

 void build_red_green() {
      pthread_mutex_lock(&red_lock);
      // Do something with red
     pthread_mutex_unlock(&red_lock);

     pthread_mutex_lock(&green_lock);
      // Do something with green
     pthread_mutex_unlock(&green_lock);
 }

 void build_green_blue() {
     pthread_mutex_lock(&green_lock);
      // Do something with green
      pthread_mutex_unlock(&green_lock);

      pthread_mutex_lock(&blue_lock);
      // Do something with blue
     pthread_mutex_unlock(&blue_lock);
 }

 void build_blue_red() {
      pthread_mutex_lock(&blue_lock);
      // Do something with blue
     pthread_mutex_unlock(&blue_lock);

      pthread_mutex_lock(&red_lock);
      // Do something with red
     pthread_mutex_unlock(&red_lock);
 }
```

## Social Media “Like” Tracker

Pearl is building the backend for a new social media application called *“SnapTalk.”* One of the app's core features is the ability for users to “like” posts made by others. To keep track of this, Pearl uses the following structure in her C++ program:

```
unordered_map<int, unordered_set<string>> likes;
```

In this structure:

- The key (int) represents the post_id (each post has a unique identifier).
- The value is an unordered_set<string> containing the usernames (string) of users who have liked that post.

Each user can like a post at most once. Even if they try to like the same post again, the like count should not increase.

E.g.

```
likes = {
    {1001, {"Alice", "Bob"}},
    {1002, {"Bob", "Charlie", "Alice"}},
    {1003, {"Diana"}}
};
```

- Post 1001 was liked by Alice and Bob.
- Post 1002 was liked by Bob, Charlie, and Alice.
- Post 1003 was liked by Diana.

Note that a user can like multiple posts but should appear only once per post.

The problem continues on the next page:

## Part 1

Write a function like_post that takes the likes structure, a post_id, and a username and adds the user to the set of likes for that post. A new entry should be created if the post doesn't exist yet. If the user already liked the post, nothing should change.

```
void like_post(unordered_map<int, unordered_set<string>>& likes,
               int post_id, string username) {


  likes[post_id].insert(username);



}
```

## Part 2

Write a function unique_users that returns the number of distinct users who have liked any post. Even if a user liked multiple posts, they should be counted only once.

In the example above, unique_users(likes) should return 4 because the users are: Alice, Bob, Charlie, and Diana.

```
int unique_users(
          const unordered_map<int, unordered_set<string>>& likes) {


  // one possible solution
  unordered_set<string> users;

  for (auto like : likes) {
    for (string user: like.second) {
      all_users.insert(user);
    }
  }
  return users.size();

}
```

## Mixed Media Management

Pearl has a full-fledged media library, which now not only stores books, but also DVDs and Magazines. Each item type has some shared and some unique properties.
To capture this, she defines the following structs:

```
struct Book {
      string title;
      string author;
      int pages;
};

struct DVD {
      string title;
      int duration;
};

struct Magazine {
      string title;
      int issue;
};
```

She stores the items in a map, using a unique media_id (int) as the key. Since a media item can be a Book, DVD, or Magazine, she uses std::variant to represent any type of media:

```
map<int, variant<Book, DVD, Magazine>> library;
```

She also wants to track which users have borrowed which items. For this, she uses another structure:

```
unordered_map<int, unordered_set<string>> borrowed_by;
```

This maps a media_id to a set of usernames who have currently borrowed that item (note: multiple users can borrow the same magazine or DVD, but only one user can borrow a book at a time).

This problem continues onto the next page

## Part 1

Write the function borrow_media that takes a media_id, a username, and updates the borrowed_by structure accordingly.

Rules:

- If the item is a Book, only allow the borrow if no one else has borrowed it yet (i.e., the set is empty).
- For DVD or Magazine, multiple users can borrow the same item.

```
void borrow_media(int media_id, const string& username,
          const map<int, variant<Book, DVD, Magazine>>& library,
          unordered_map<int, unordered_set<string>>& borrowed_by) {

  auto it = library.find(media_id);
  if (it == library.end()) return; // media_id not found

  const auto& media = it->second;

  if (holds_alternative<Book>(media)) {
    // Book: only allow borrow if no one has borrowed it yet
    if (borrowed_by[media_id].empty()) {
      borrowed_by[media_id].insert(username);
    }
  } else {
    // DVD or Magazine: allow multiple users
    borrowed_by[media_id].insert(username);
  }
}
```

## Part 2

Write a function list_borrowers that returns the set of all users who have borrowed anything from the library.

```
unordered_set<string> list_borrowers(
    const unordered_map<int, unordered_set<string>>& borrowed_by) {
  unordered_set<string> users;
  for (auto borrower : borrowed_by) {
    for (string user : borrower.second) {
      users.insert(user);
    }
  }
  return users;
}
```

## OH mysterious memory leak

Travis is writing a C++ program to simulate how students remember who helped them in 5950 office hours. Each student remembers a history of TAs who helped them, stored as an array of strings.

To manage this, Travis writes a class:

```
class StudentMemory {
 private:
  string* taNames_;
  int count_;

 public:
  StudentMemory(int size) {
    taNames_ = new string[size];
    count_ = size;
  }

  void setTA(int index, const string& name) {
    if (index >= 0 && index < count_) {
      taNames_[index] = name;
    }
  }

  void printTAs() const {
    for (int i = 0; i < count_; ++i) {
      cout << taNames_[i] << " ";
    }
    cout << endl;
  }

  ~StudentMemory() {
    delete[] taNames_;
  }
};
```

Travis writes a function:

```
void processMemory(StudentMemory m) {
  m.setTA(0, "Travis");
  m.printTAs();
}
```

And in main, he writes:

```
int main() {
  StudentMemory s(2);
  s.setTA(0, "Lobi");
  s.setTA(1, "Pearl");
  processMemory(s);
  s.printTAs();
}
```

This program compiles and runs, but Travis notices strange behavior: sometimes incorrect output, or even a crash. There are 2 problems. The class is missing something and Travis' code that uses it (processMemory or Main) uses it wrong. What are the issues, and how can you fix them?

Please write your answer here:

Problem #1: no copy constructor means shallow copy
When Travis calls processMemory(s), it copies the StudentMemory object by value. But since there's no custom copy constructor, C++ will do a shallow copy, which means that both s and m point to the same taNames array. m.setTA(0, "Travis") modifies the same memory as s. When processMemory(m) ends, m's destructor runs, freeing the shared taNames array. Later, s.printTAs() accesses freed memory, which is undefined behavior.

Problem #2: copying breaks the class
Travis's class manually allocates memory using new[] and frees it in the destructor using delete[]. But when an object of this class is copied/passed by value, C++ does a shallow copy by default, which means both the original and the copy now point to the same memory. When the copy is destroyed, it frees that memory, leaving the original with a dangling pointer. So then if we try to access it again later, it will lead to crashes or corrupted output. One way to fix this is to add a copy constructor to create a new array and copy over the values, and also add a copy assignment operator to safely assign one object to another without sharing memory.

## OHQ is down!

Oh no! OHQ is down, but there are students flooding in to 5950 OH seeking help on the final projects. To resolve this, the TAs have implemented a help queue for CIT 5950. Angie is helping clean up the help queue. Each TA logs students they're helping, grouped by topic. The help queue is stored as:

```
unordered_map<string, vector<string>> helpQueue;
```

Each key is a topic (like "Pthreads" or "PennFAT"), and the value is a vector of student names being helped on that topic.

Here's an example of what it might look like:

```
{
  "Pthreads" → ["Andrew", "Travis", "Hassan"],
  "PennFAT"  → ["Pearl", "Travis"],
  "Signals"  → ["Travis"]
}
```

### Part 1

However, some students rage quit and left. To update the queue, Angie wants to write a function:

```
void removeStudent(unordered_map<string, vector<string>>& helpQueue,
                   const string& name);
```

This function should remove all instances of a given student (ex., "Travis") from every topic in the queue. Your task is to implement the function removeStudent that safely removes all entries matching the given student name across all topics.

You should:

- Use iterators (do not use std::remove!)
- Avoid iterator invalidation errors
- Modify the map in-place (e.g. do not construct a new map, just modify the one passed in)

Here is an example of this function:
Say we want to remove all instances of a student named "Travis". After calling `removeStudent(helpQueue, "Travis");` the helpQueue should now be:

```
{
  "Pthreads" → ["Andrew", "Hassan"],
  "PennFAT" → ["Pearl"],
  "Signals" → []
}
```

Write your implementation on the next page

```
void removeStudent(unordered_map<string,vector<string>>&helpQueue,
                   const string& name) {

  for (auto& pair : helpQueue) {
    vector<string>& students = pair.second;
    for (auto it = students.begin(); it != students.end(); ) {
      if (*it == name) {
        it = students.erase(it);
        // ^ erase returns next valid iterator
      } else {
        ++it;
      }
    }
  }
}
```

## Part 2

After calling removeStudent, Angie realizes some topics have no students left (ex., "Signals"). She wants to clean up the helpQueue by removing any topic where the list of students is empty.

For example, in the earlier example, we want to remove "Signals" → []:

```
{
  "Pthreads" → ["Andrew", "Hassan"],
  "PennFAT" → ["Pearl"],
  "Signals" → []
}
```

Your job is to write this function:

```
void removeEmptyTopics(unordered_map<string, vector<string>>&
                                                    helpQueue);
```

This function should:

- Iterate over the map
- Remove any key-value pair where the value (the student vector) is empty
- Be careful not to invalidate iterators while modifying the map.

```
void
removeEmptyTopics(unordered_map<string,vector<string>>&helpQueue) {
  for (auto it = helpQueue.begin(); it != helpQueue.end(); ) {
    if (it->second.empty()) {
      it = helpQueue.erase(it); ← erase returns next valid iterator
    } else {
      ++it;
    }
  }
```

```
}
```

## countFrequencies

You are given two vectors of strings:

1. words: a list of words that may contain duplicates.
2. tokens: a list of keywords you care about.

Your task is to write a function that counts how many times each word in tokens appears in words, returns a vector<int> where each element corresponds to the frequency of a word in tokens, in the same order

For example if we had the inputs:

vector<string> words =26 {"ok", "hello", "world", "ok", "errors", "world", "world", "errors"};
vector<string> tokens = {"hello", "world", "errors", "case"};

And called our function:
vector<int> frequencies = countFrequencies(words, tokens);

The result would be:
vector<int> frequencies {1, 3, 2, 0};

```
vector<int> countFrequencies(const vector<string>& words,
                             const vector<string>& tokens) {

  // one possible solution

  unordered_map<string, int> freq_map;
  for (string& word : words) {
    freq_map[word]++;
  }

  vector<int> result;

  for (string& token : tokens) {
    result.push_back(freq_map[token]);
  }
  return result;
}
```

## Implement a thread-safe bank account system using POSIX thread

Consider the following code that implements a simple multi threaded bank account system

```
int balance = 0;

pthread_mutex_t mutex;

void* deposit(void* arg) {
  for (int i = 0; i < 5; i++) {
    balance += 100;
    sleep(1);
  }
  return NULL;
}

void* withdraw(void* arg) {
  for (int i = 0; i < 5; i++) {
    pthread_mutex_lock(&mutex);
    balance -= 50;
    pthread_mutex_unlock(&mutex);
    sleep(2);
  }
  return NULL;
}

int main() {
  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL, deposit, NULL);
  pthread_create(&thread2, NULL, withdraw, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

 return 0;
}
```

Part1:

The current implementation is not thread-safe and may produce incorrect results when accessed by multiple threads via a datarace. Please identify which part of the code causes date races, explain why it is unsafe, and rewrite the function(s) that need to be fixed.

```
void* deposit(void* arg) {
  for (int i = 0; i < 5; i++) {

    pthread_mutex_lock(&mutex);

    balance += 100;

    pthread_mutex_unlock(&mutex);
    sleep(1);
  }
  return nullptr;
}
```

Part2:

Additionally, there's a logical flaw. We want to make sure that when we withdraw we only perform the withdrawal if there is enough to subtract from the balance (balance never goes below zero). If the balance is too small, then we wait for it to get bigger before we process the withdrawal. What strategy should we apply to prevent incorrect withdrawals? Describe at a high level how you would change the code to support this change.

This can be achieved by using a pthread_cond_t condition variable.
The thread attempting to withdraw should call pthread_cond_wait() when the balance is too low, and the deposit function should call pthreadcondsignal() after adding money to wake up waiting threads.

## Water Tank

There is a central water tank that provides hot water to multiple households. Each household performs daily activities (e.g., bathing, laundry) that consume varying amounts of water.

In this question:

- Each household is modelled as a thread that periodically uses some water.
- If the tank volume falls below a threshold, no household may draw water until it is refilled.
- A separate refill thread monitors the tank and refills it when needed.
- During a refill, all households must wait until the refill completes.

Assume that the mutex and conditional variable are initialized properly. Each household thread receives its own std::vector<int> representing the amount of water needed for each activity.

Fill in the blanks in refill() and household() to ensure correct synchronization and coordinated access to the water tank (code continues on the next page).

```
#define TANK_THRESHOLD 20
#define TANK_CAPACITY 1000
bool refill_in_progress = false;
pthread_mutex_t tank_mutex;
pthread_cond_t tank_cond;
int current_tank_volume = TANK_CAPACITY;

void *refill(void* arg) {
  while (true) {
    // TODO: acquire relevant locks and wait
    // until refill is actually needed

    pthread_mutex_lock(&tank_mutex);
    while (current_tank_volume > TANK_THRESHOLD) {
      pthread_cond_wait(&tank_cond, &tank_mutex);
    }

    // begin refill
    refill_in_progress = true;
    cout << "Refilling tank..." << endl;
    current_tank_volume = TANK_CAPACITY;
    sleep(2); // simulate refill delay
    cout << "Tank is refilled" << endl;

    // function continues onto next page
    // TODO: mark refill as completed and notify all waiting threads
    refill_in_progress = false;
    pthread_cond_broadcast(&tank_cond);
    pthread_mutex_unlock(&tank_mutex);
```

```
  }
  return nullptr;
}
void *household(void* arg) {
  std::vector<int> activities;
  activities = *(static_cast<std::vector<int>*>(arg));

  for (int water_needed : activities) {
    // TODO: lock and wait if there is a refill is in progress

    pthread_mutex_lock(&tank_mutex);
    while (refill_in_progess) {
      pthread_cond_wait(&tank_cond, &tank_mutex);
    }



    while (current_tank_volume < water_needed) {
      cout << "Not enough water. Requesting refill..." << endl;
      // TODO: signal the refill thread and
      // wait until the refill completes
      pthread_cond_signal(&tank_cond); // wake the refill thread
      pthread_cond_wait(&tank_cond, &tank_mutex);
      // wait for refill to finish

    }

    current_tank_volume -= water_needed; // consume water
    // TODO: release any mutexes held

    pthread_mutex_unlock(&tank_mutex);



    sleep(1); // simulate delay between uses
  }
  return nullptr;
}
```

## Multithreaded Network Socket

Hassan has been working to create a multithreaded server on his computer. One thread focuses on sending messages from his computer to Angie's computer which is on another internet network. Another thread reads messages received from Angie and outputs them to a file called "out.txt". Help Hassan to implement this server by helping complete the code for the two threads to send and receive messages across the network. Assume that ANGIE_IP is a constant string and that the port number for Angie's computer is 10000.

Hassan has set up a socket to send data to Angie's computer via UDP, but is unable to send the input from his terminal to her computer. Help him add this. Use the following function to send data across the network:

As a hint, here is the function signature for sending data across a udp socket

```
int sendto(int socket, char *buffer, int length, int flags,
           struct sockaddr *address, int address_len);
```

Reminder: flags can be set to 0

```
void* send(void* arg) {
    // code to initialize socket
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in addr;

    addr.sin_family = AF_INET;
    addr.sin_port = htons(10000);
    inet_pton(AF_INET, ANGIE_IP, &addr.sin_addr);

    // TODO: Read from the terminal and send to Angie
    // you can do this in a variety of ways, but
    // your code should keep sending data from
    // terminal until there is no more data ("eof")

    array<char, 1024> buf {};
    ssize_t bytes_read = read(STDIN_FILENO, buf.data(), 1024);
    while (bytes_read > 0) {
      sendto(sockfd, buff.data(), bytes_read, 0,
           reinterpret_cast<struct sockaddr*>(&addr), sizeof(addr));
      bytes_read = read(STDIN_FILENO, buf.data(), 1024);
    }

    // another way this could be done:
    string line;
    while (getline(cin, line)) {
```

```
      line += '\n'; // getline removes the newline
      sendto(sockfd, line.data(), line.size(), 0,
           reinterpret_cast<struct sockaddr*>(&addr), sizeof(addr));
    }


    close(sockfd);
}
```

Hassan already wrote the code to write the messages he receives from Angie's computer, however he is not seeing any messages in the file. What is the problem with the following code for the thread to receive data from Angie's computer that Hassan created?

```
void* receive(void* arg) {
    // code to initialize socket
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in addr;

    addr.sin_family = AF_INET;
    addr.sin_port = htons(10000);
    addr.sin_addr.s_addr = INADDR_ANY;

    bind(sockfd, (struct sockaddr*)&server_addr, sizeof(addr));

    // code to read to file
    int out_fd = open("out.txt", ~~O_RDONLY~~ O_WRONLY | O_APPEND);

    char buff[1024];
    while (true) {
        int len = recvfrom(sockfd, buff, 1024, 0, NULL, NULL);
        write(out_fd, buff, len);
    }

    close(out_fd);
    close(sockfd);
}
```

## Threads - Synchronization

Complete the following function which is called by a thread so that it uses a mutex to safely increment a global counter.

```
int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  // assume initialized

void* increment(void* arg) {
    for (int i = 0; i < 1000; ++i) {
        // TODO:

        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);

    }
    return nullptr;
}
```

## Threads - Producer/Consumer

In a producer-consumer setup using a shared buffer, why do we need a pthread_mutex_t (and usually pthread_cond_t) to coordinate between producer and consumer threads?

A pthread_mutex_t ensures that only one thread accesses the shared buffer at a time, preventing race conditions. A pthread_cond_t is used to block a consumer when the buffer is empty and to block a producer when the buffer is full, resuming them only when the condition changes. This prevents busy waiting and ensures proper synchronization.

## Memory Diagram with Threads

Draw the memory diagram for the code below at the point commented “DRAW MEMORY DIAGRAM HERE”. Be sure to be clear about what is on each stack and what is on the heap

```
void* add(void* arg) {
    std::string& referenced_string = *static_cast<std::string*>(arg);
    referenced_string += “ world!”;
    // DRAW MEMORY DIAGRAM HERE
    return NULL;
}

int main() {
    pthread_t thread;
    std::string new_string = “hello”;

    pthread_create(&thread, NULL, add, &new_string);
    pthread_join(thread, NULL);

    return EXIT_SUCCESS;
}
```