University of Washington – Computer Science & Engineering

CSE 333 A & B Autumn 2025 Final: Version A

Last Name:	Studofrasdottír
First Name:	Studentia
Student ID Number:	3
UWNetID:	stdout
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE333 who haven't taken it yet. Violation of these terms could result in a failing grade. **(please sign)**	

Do not turn the page until told to do so.

Instructions

- This exam contains 30 pages, including this cover page. Put your final answers in the boxes and blanks provided. You may make use of the 'overflow box' on page 26 for additional answer space.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed two 5"x8" index cards (double-sided) of *handwritten* notes.
- Please silence and put away all cell phones and other mobile or noise-making devices.
- You have 1 hour and 50 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read *all* questions first and start where you feel the most confident.
- Relax. You've been working hard. You got this.

Question	1	2	3	4	5	6	7	Total
Possible Points	16	30	5	10	10	30	16	**117**

Question 1: S-T-Elf On The Shelf (C++, STL) (16 pt)

Santa's workshop is looking to automate its naughty and nice list using a small C++ program that reads information from standard input. The program should accept input using `cin` to support the following three operations:

- **`<name_of_child> increase`**: increase that child's score by 1
- **`<name_of_child> decrease`**: decrease that child's score by 1
- **`lists`**: prints the nice and naughty lists

Each child begins with a score of 0 the first time they are mentioned. Children with scores >= 0 are considered nice and those with a score < 0 are naughty. Names within each list can be printed in any order.

Suppose the following input is read from `cin`:

```
abby increase
ken decrease
connor increase
sam increase
ken decrease
connor increase
abby decrease
lists
```

After reading this input and reaching the `lists` command the program should produce the following output:

```
nice: abby connor sam
naughty: ken
```

Use `cin`'s extraction operator to read strings from standard input and skip whitespace (e.g., `cin >> s`). You may assume that input operations like reading strings will succeed. Use the provided `print_list(...)` function as part of your solution. You may find one or more STL container types helpful for this task.

Tip: Review the reference documentation that is included at the end of your exam.

Question continues on the next page

Provide your answer in the box below by completing the main function. You should not break your solution into additional functions.

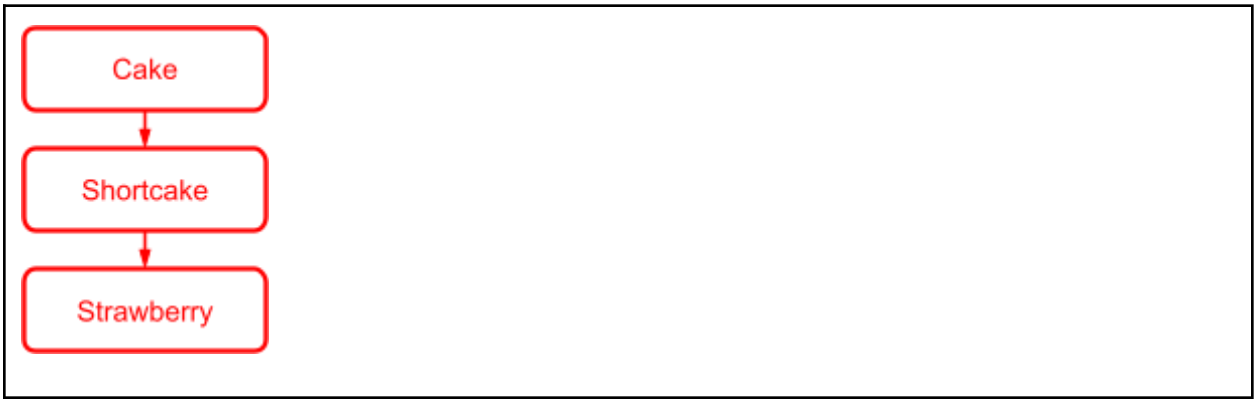
```
// Assume necessary headers are already included…
using namespace std;

void print_list(const char* label, const vector<string> &list) {
  cout << label << ":";
  for (const auto &child : list) {
    cout << " " << child;
  }
  cout << endl;
}

int main(int argc, char **argv) {
  map<string,int> scores;
  string name;

  while (cin >> name) {
    if (name == "lists") {
      vector<string> niceList, naughtyList;

for (const auto &child : scores) {
        if (child.second >= 0) {
          niceList.push_back(child.first);
        } else {
          naughtyList.push_back(child.first);
        }
      }      print_list("nice", niceList);
      print_list("naughty", naughtyList);
    } else {
      string action;
      cin >> action;

      if (action == "increase") {
       scores[name] += 1;
      } else if (action == "decrease") {
       scores[name] -= 1;
     }
    }
  }
  return EXIT_SUCCESS;
}
```

Question 2: The Cake Is a Lie (Inheritance and Dispatch) (30 pt)

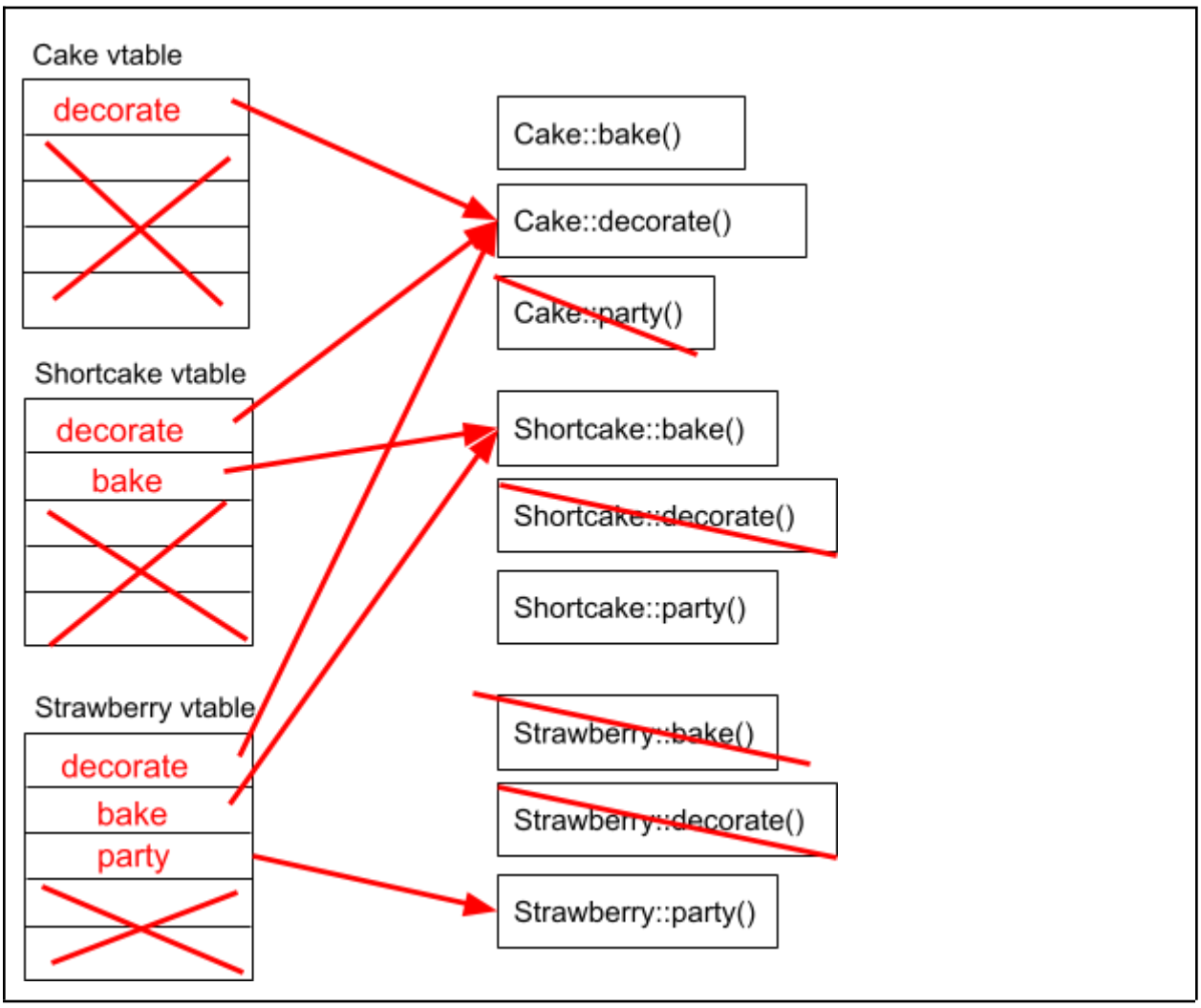
We've decided to bake some cakes, and decided to use class inheritance to make our lives simpler (ha). Consider this header file, "`cake.h`", which you can assume contains no errors:

```
#include <iostream>
using namespace std;

class Cake {
public:
  void bake() {
    cout << "Cake::bake" << endl;
  }
  virtual void decorate() {
    cout << "Cake::decorate" << endl;
  }
};

class Shortcake : public Cake {
public:
  void party() {
    decorate();
    cout << "Shortcake::party" << endl;
  }
  virtual void bake() {
    party();
    cout << "Shortcake::bake" << endl;
  }
};

class Strawberry : public Shortcake {
public:
  virtual void party() {
    bake();
    cout << "Strawberry::party" << endl;
  }
};
```

a) Draw the inheritance hierarchy for the classes defined in this header file



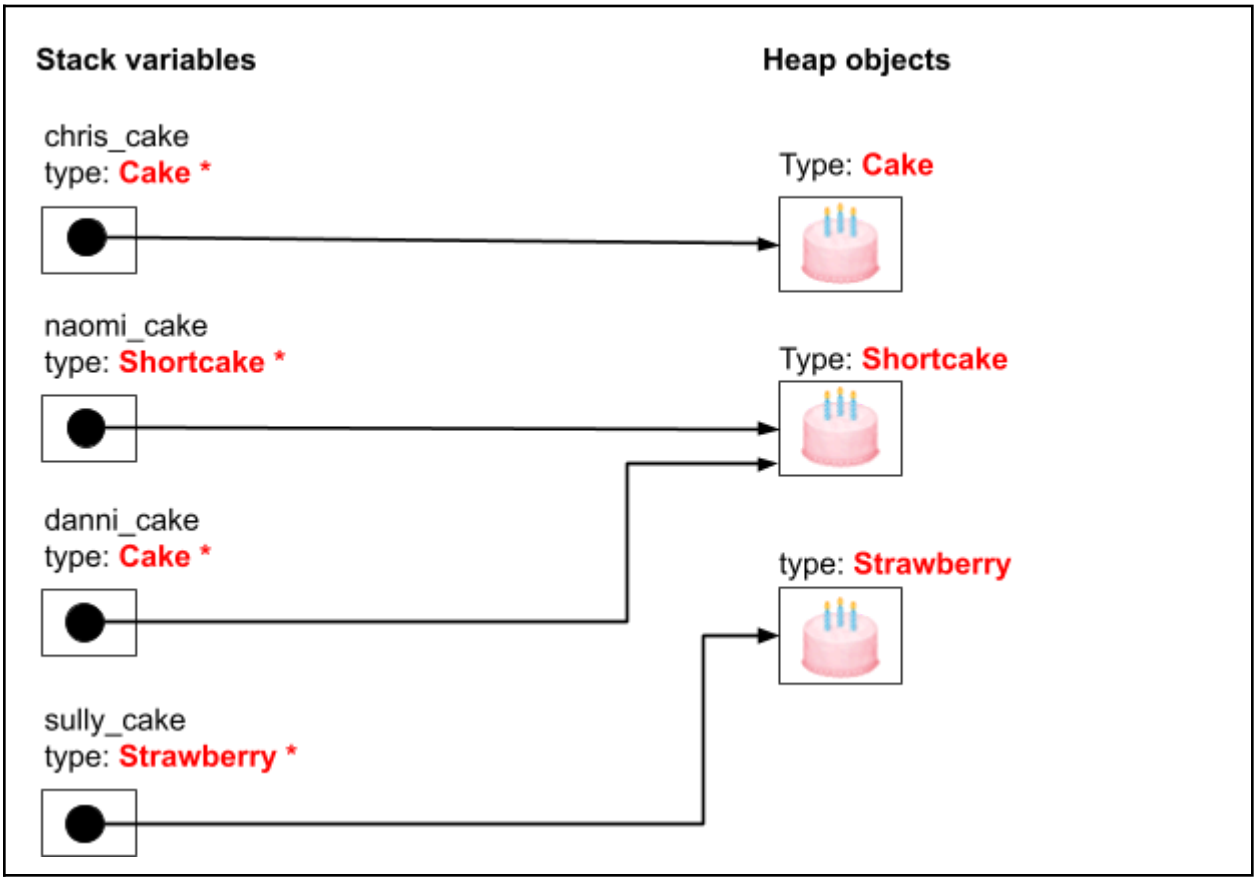
b) Fill out the vtables for each class by writing unqualified function names (no “typename::”) in the table cells and drawing arrows to the code blocks that implement them. Cross out any table cells and blocks that aren’t present in the compiled program.



Now consider the following program, that compiles without errors:

```
#include “cake.h”

int main(int argc, const char **argv) {

  Cake* chris_cake = new Cake();
  Shortcake* naomi_cake = new Shortcake();
  Cake* danni_cake = naomi_cake;
  Strawberry* sully_cake = new Strawberry();

  chris_cake->bake();
  cout << "----" << endl;

  naomi_cake->bake();
  cout << "----" << endl;

  naomi_cake->party();
  cout << "----" << endl;

  danni_cake->decorate();
  cout << "----" << endl;

  sully_cake->bake();
  cout << "----" << endl;

  sully_cake->party();

  return 0;
}
```

c) Complete the diagram below by indicating which type the compiler sees for each diagram element:



d) What does the program output when it executes?

Cake::bake

Cake::decorate
Shortcake::party
Shortcake::bake

Cake::decorate
Shortcake::party

Cake::decorate

Cake::decorate
Shortcake::party *
Shortcake::bake

Cake::decorate
Shortcake::party *
Shortcake::bake
Strawberry::party

* **Explanation:** these calls dispatched to Shortcake::party() rather than Strawberry::party() because they were made within from Shortcake::bake(). From the latter function's point of view, the type of the "this" pointer that party() is implicitly being called on is "Shortcake *", and to "Shortcake"s "party()" is a static dispatch function (virtual is only sticky *down* the inheritance tree)

Question 3: Me, Myself and I-node (Smart pointers) (5pt)

Consider the following structure used to represent a simple file system hierarchy:

```
struct Directory {
  string name_;
  vector<shared_ptr<Directory>> subdirs;
  shared_ptr<Directory> parent;

  Directory(string name) : name_(name) { };
};
```

Now suppose we use it in the following function (assume all necessary includes):

```
void test() {
  shared_ptr<Directory> root(new Directory("root"));
  shared_ptr<Directory> home(new Directory("home"));

  // *** POINT 1 ***

  root->subdirs.push_back(home);
  home->parent = root;

  // *** POINT 2 ***
}
```

a) Write the reference counts for each object on the heap at “POINT 1”

root – 1
home – 1

b) Write the reference counts for each object on the heap at “POINT 2”

root – 2
home – 2

c) Was memory leaked after the above function returned? If so, describe a change we can make to the structure (*not* the test() function) to resolve the issue and briefly describe why this change fixes things. If not, explain why the current design is sufficient.

Yes. Because of the “parent” pointer in “home” there is a cyclic pointer relationship between “root” and “home”. When the shared_ptrs in “test()” go out of scope, they’ll both still have a reference count of “1” and thus won’t be freed.

We can fix this by changing Directory::parent from a shared_ptr<Directory> to a weak_ptr<Directory>. This will break the dependency cycle and cause root’s ref count to hit 0 when “test()” returns, which will cause it to be deleted and in turn cause home’s ref count to hit 0, thus deleting that too.

Question 4: I Cast Magic Missile! (C++ Casting) (10 pt)

Oh boy, you're in trouble now. It turns out this isn't a computer science class; you accidentally signed up for "***Curses, Spells, and Enigmas***" 333: Spell Casting! You come across some ancient C++ magic and are trying to understand exactly how the arcane incantations work.

a) In the code below, **circle** all instances of implicit casting and write out the correct **explicit C++ cast** next to it. Assume all necessary headers and "using" statements have been included. (hint: be careful about things that appear to mutate the type of the data but are *not* casts!)

```
class Spell {
public:
  Spell(const string& magic_word) : magic_word_(magic_word) { }

  string GetMagicWord() const {
   return magic_word_;
  }

private:
  string magic_word_;
};

class Fireball : public Spell {
public:
  Fireball(const string& magic_word, uint32_t heat) :
    Spell(magic_word), heat_(heat) { }

  int GetHeat() const {
   return ~~heat_~~static_cast<int>(heat_);
  }

private:
  uint32_t heat_;
};

// CODE CONTINUES ON NEXT PAGE
```

a) continued from previous page:

```
int main() {
  double x = 67.41;
  Fireball f1("BAM", x static_cast<uint32_t>(x));
  Spell* s = &f1 static_cast<Spell *>(&f1);

  cout << s->GetMagicWord() << " with " << s static_cast<Fireball
*>(s)->GetHeat()
       << "heat!" << endl;

  Fireball* fp = s static_cast<Fireball *>(s);
  Fireball f2("BZZZ", &x reinterpret_cast<uint32_t>(&x));

  cout << f2.GetMagicWord() << " this time with " << f2.GetHeat()
       << "heat???" << endl;
}
```

Notes:

- dynamic_casts are also acceptable for Fireball
- numeric types in the cout<< lines aren't being cast; they're being passed into the overloaded operator<<() for ostreams and their respective types
- It's arguable whether the string literals are being cast to std::strings or not; they're technically being fed as const char*s into the *constructor* of std::string(), but you could make the case this is "casting"

b) Briefly explain why using const-casts are often regarded as dangerous, even for the most skilled of witches and warlocks.

It means that part of the code is assuming some data will not change, while other parts of the code may secretly be changing that data after all! If the compiler performs optimizations based on assuming those values won't change, the program's behavior may break.

Question 5: The best brand of Binary Tree Node (generic) (10 pt)

The following program constructs and destructs a binary tree of three integer values. In the indicated space on the next page rewrite the contents of btnode.h to support types besides integers. For example, bt.cc as currently written should still work with your implementation; however, if the line marked `main_L1` is commented out and the line marked `main_L2` is uncommented and no other changes are made to bt.cc, then the program should construct and then destruct a binary tree of three std::string values when using your version of btnode.h.

Note: This C++ code compiles and runs without error.

bt.cc

```
#include <string>
#include <cstdlib>

#include "btnode.h"

int main() {
  int vals[] = {2, 9, 3};                              // main_L1
  // std::string vals[] = {"two", "nine", "three"};    // main_L2

  auto* left = new BTNode(vals[0]);
  auto* right = new BTNode(vals[1]);
  auto* root = new BTNode(vals[2], left, right);

  delete root;
  delete right;
  delete left;

  return EXIT_SUCCESS;
}
```

btnode.h

```
class BTNode {
 public:
  BTNode(int val)
    : val_(val), left_(nullptr), right_(nullptr) {}
  BTNode(int val, BTNode* const left, BTNode* const right)
    : val_(val), left_(left), right_(right) {}

 private:
  int val_;
  BTNode* left_;
  BTNode* right_;
};
```

Write the complete contents of your version of `btnode.h` in the empty box below.

```
// Potential Solution #1
// - templated class on type of stored value
template<class T>
class BTNode {
 public:
  BTNode(T val) : val_(val), left_(nullptr), right_(nullptr) {}
  BTNode(T val, BTNode<T>* const left, BTNode<T>* const right)
    : val_(val), left_(left), right_(right) {}

 private:
  T val_;
  BTNode<T>* left_;
  BTNode<T>* right_;
};

// Potential Solution #2
// - similar to #1 except non-required (inferrable) parameters omitted
// - see 'class template argument deduction (CTAD)' introduced in C++17
template<class T>
class BTNode {
 public:
  BTNode(T val) : val_(val), left_(nullptr), right_(nullptr) {}
  BTNode(T val, BTNode* const left, BTNode* const right)
    : val_(val), left_(left), right_(right) {}

 private:
  T val_;
  BTNode* left_;
  BTNode* right_;
};
```

Question 6: `accept()`ing Love (Network sockets) (30pt)

Danni's chat program was such a wild success, she decided to open-source the server code. The code itself is bug-free, but many of the comments she added to explain things are factually incorrect (she was mostly just copy-and-pasting Sully's homework…). We labelled the comments we suspect may be incorrect with the text "**[CHECK _]**". You can assume if a comment is not labeled like this, it is accurate.

```
// Assume all necessary headers are included by this file:
#include "necessary_headers.h"

// Given a port number, set up a listener socket for said port.
// Socket is active and has an open connection queue.
// Function returns the socket fd, or a non-positive number on
// failure.
int  GetListenerSocket(char *portnum);

// Given a new client socket, conduct a chat session.
// Return once the chat session is complete (socket disconnects
// or the user sends the message "/quit")
void ChatSession(int c_fd);

// Given a client socket, read a line of text from it and
// return said text as a std::string. Text will be properly
// null-terminated and stripped of newlines.
// On unrecoverable failure or client disconnection, string
// will be empty.
std::string ReadLineFromSocket(int fd);

int main(int argc, char **argv) {
  // Expect the port number as the first command line argument.

  // [CHECK A] We don't have to bounds-check the port, because
  // the OS will take care of that part for us.

  // [CHECK B] Set up our listener socket. This socket shouldn't
  // receive any data, just wait for new connections to show up.
  int listen_fd = GetListenerSocket(argv[1]);
  if (listen_fd <= 0) {
```

```
    return EXIT_FAILURE;
  }

  // [CHECK C] Loop indefinitely, accepting one chat client at a
  // time. Because we're single-threaded, anyone trying to connect
  // while we're in a chat session will get immediately rejected.
  while (1) {
    struct sockaddr_storage caddr;
    socklen_t caddr_len = sizeof(caddr);
    int client_fd = accept(listen_fd,
                     reinterpret_cast<struct sockaddr *>(&caddr),
                     &caddr_len);
    if (client_fd < 0) {
      if (errno == EINTR || errno == EAGAIN ||
          errno == EWOULDBLOCK) {
        continue;
      }
      break;
    }
    ChatSession(client_fd);
  }

  // [CHECK D] Danni is a very good and great girl who
  // expresses love by being fluffy.
  close(listen_fd);
  return EXIT_SUCCESS;
}

int GetListenerSocket(char *portnum) {
  // [CHECK E] We're asking for a TCP socket here because
  // the stream-oriented connection will be faster and more
  // efficient than a bunch of random UDP packets lobbed
  // across the internet
  struct addrinfo hints;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET6;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
```

```
hints.ai_flags |= AI_V4MAPPED;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_canonname = nullptr;
hints.ai_addr = nullptr;
hints.ai_next = nullptr;

// [CHECK F] Ask the OS for a server IP address and other
// socket metadata that supports the hints we listed above,
// possibly getting more than one answer back if there are
// multiple network interfaces, IP versions, or other options.

// [CHECK G] We also don't have to worry about portnum's byte
// ordering, because it's being passed directly to the OS as
// an ASCII string

struct addrinfo *result;
int res = getaddrinfo(nullptr, portnum, &hints, &result);
if (res != 0) {
  return -1;
}

int listen_fd = -1;
for (struct addrinfo *rp = result; rp != nullptr;
     rp = rp->ai_next) {
  // [CHECK H] Make a new socket fd, specifying the protocols
  // we'll be using but nothing else about it (yet). It's not
  // possible for this step to fail.
  listen_fd = socket(rp->ai_family,
                     rp->ai_socktype,
                     rp->ai_protocol);

  int optval = 1;
  setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR,
             &optval, sizeof(optval));

  // [CHECK I] Bind the socket, reserving our port number in the
  // OS for the specific server IP address we list here. At this
  // point, the listener socket still isn't actually letting
```

```
      // clients wait in the queue.
      if (bind(listen_fd, rp->ai_addr, rp->ai_addrlen) == 0) {
        break;
      }

      close(listen_fd);
      listen_fd = -1;
    }

    freeaddrinfo(result);

    if (listen_fd == -1) {
      return listen_fd;
    }

    // [CHECK J] Start listening, auto-closing the socket after
    // 60 seconds of inactivity.
    if (listen(listen_fd, 60) != 0) {
      close(listen_fd);
      return -1;
    }

    return listen_fd;
}

static char readbuf[1024];
std::string ReadLineFromSocket(int fd) {
  std::string response;
  while(1) {
    // [CHECK K] Read a line of text from the socket that is
    // at most 1024 bytes large. Will block and wait if no
    // data is available yet.
    int res = read(fd, readbuf, 1024);
    if (res == 0) {
      // [CHECK L] Client disconnected, return an empty
      // string
      break;
    }
```

```
    if (res == -1) {
      if (errno == EAGAIN || errno == EINTR) {
        continue;
      }
      // [CHECK M] There was an unrecoverable read error,
      // return an empty string
      break;
    }
    readbuf[res - 1] = '\0';
    response = std::string(readbuf);
    break;
  }
  return response;
}

void ChatSession(int c_fd) {
  // [CHECK N] Make sure to include a newline on the message
  // to ensure the buffer flushes
  std::string response = "werr\n";

  // The first line of text received is the username
  // of the person on the other end
  std::string username = ReadLineFromSocket(c_fd);

  // For all subsequent lines, print the username and message
  // to stdout and respond to the client with the message 'werr'.
  // If message is "/quit", don't respond and just close the socket
  // instead.
  while (1) {
    std::string message = ReadLineFromSocket(c_fd);
    if (message.empty() || message == "/quit") {
      break;
    }
    std::cout << "<" << username << "> " << message << std::endl;
    write(c_fd, response.c_str(), response.length());
  }
```

```
    // [CHECK O] Close the socket so, among other reasons, the OS
    // can reuse the file descriptor number for future clients
    close(c_fd);
}
```

a) In the box below, identify which comments were inaccurate and briefly describe how to fix them. Identify each comment by the letter specified by the **[CHECK _]** prefix.

Check C – Clients that try to connect while we're in a session will just sit in the listener socket's wait queue. Fix the comment by saying such folks will have to wait (rather than being rejected)

Check D – To clarify for posterity, Danni is the instructor's cat and this comment is completely accurate. One *could* make the case it should be describing the code below it though.

Check E – TCP has much more bandwidth overhead and is often slower than UDP. This comment should say "We're asking for a TCP socket because it will ensure messages don't get lost or duplicated"

Check H – It is indeed possible for this socket() call to fail, remove that sentence. We don't *need* to add error checking here since subsequent syscalls' error checking will take care of that for us, but it would be good form to add it anyway.

Check J – The "60" is the max length of pending connections in the wait queue, not a timeout value. The comment should say "Start listening, with a max of 60 pending connections"

Check K – read() doesn't read whole lines at a time, it reads whatever data is available (including and moving beyond newline characters). It's a matter of coincidence that messages will come one at a time. The comment should start by saying "Read at most 1024 bytes from the socket"

Check N – Stream socket bytes may be buffered, but not waiting for a newline. Newlines won't flush the contents of a TCP socket. This comment should either be removed or changed to say "Add a newline to display on the other end"

(box continues on next page if you need more space)

a) Continue your answer if you need more space:

(box continues on next page if you need more space)

a) Continue your answer if you need more space:

b) Danni kept the source code to the chat client a secret, so we'll have to write our own. What programming language or libraries do you want to use to write the client? (The only wrong answer is an empty box!)

So many options!!

Question 7: So Threadbare, It's Hard To Process (Concurrency) (16 pt)

Beginning from some positive integer n assume that its value is updated with the following simple logic: when n is even then update `n = n / 2` and when n is odd then update `n = 3*n + 1`. The Collatz conjecture claims that by following these simple update rules, beginning from *any* positive integer, you will *eventually* reach the case when `n == 1`. The conjecture remains unproven and you plan to leverage threads to find a counter example. The program below divides the task across three worker threads: one to update when n is even, another when n is odd, and a third to monitor for `n == 1` and confirm the conjecture holds for the number tested.

The code below gives a complete and functioning main function for your program along with the function declarations used by your three worker threads. Three mutexes are also available for use, if necessary. Your task is to complete the implementation of the three worker functions in the corresponding boxes on the next page. Your program should correctly evaluate the Collatz sequence beginning from the initial value of n in the main function and be free of race conditions while using as few mutexes as possible.

Tip: The reference material at the end of the exam includes the function signatures for locking and unlocking a pthread mutex.

collatz.cc

```
// all necessary includes
static unsigned int n;
static bool searching;
static pthread_mutex_t mutex1;
static pthread_mutex_t mutex2;
static pthread_mutex_t mutex3;

void* EvenUpdater(void*);
void* OddUpdater(void*);
void* ConvergenceObserver(void*);

int main(int argc, char** argv) {
  searching = true; // the search is on
  n = 409284972;    // beginning from this auspicious number

  // assume initialization of any necessary mutexes here
```

```
  pthread_t t_even, t_odd, t_obs;
  if (pthread_create(&t_even, nullptr, &EvenUpdater, nullptr) != 0 ||
      pthread_create(&t_odd, nullptr, &OddUpdater, nullptr) != 0 ||
      pthread_create(&t_obs, nullptr, &ConvergenceObserver, nullptr) != 0) {
    return EXIT_FAILURE;
  }

  if (pthread_join(t_obs, nullptr) != 0 ||
      pthread_join(t_odd, nullptr) != 0 ||
      pthread_join(t_even, nullptr) != 0) {
    return EXIT_FAILURE;
  }

  // assume destruction of any necessary mutexes here

  std::cout << "Collatz conjecture still holds!" << std::endl;

  return EXIT_SUCCESS;
}
```

a) Even Updater (n → n / 2 when n is even)

```
void* EvenUpdater(void*) {

  while (searching) {

    pthread_mutex_lock(&mutex1);
    if (n % 2 == 0) {
      n /= 2;
    }
    pthread_mutex_unlock(&mutex1);


  }

  return nullptr;
}
```

b) Odd Updater (n → 3 * n + 1 when n is odd)

```
void* OddUpdater(void*) {

  while (searching) {

    pthread_mutex_lock(&mutex1);
    if (n % 2 != 0) {
      n = 3 * n + 1;
    }
    pthread_mutex_unlock(&mutex1);

  }

  return nullptr;
}
```

c) Convergence Observer (stop the search when n == 1)

```
void* ConvergenceObserver(void*) {

  while (searching) {

    // sleep(1); // unnecessary to be correct but helpful in 'busy loops'
    if (n == 1) {
      searching = false;
    }



  }

  return nullptr;
}
```

d) How many mutexes were necessary to achieve a correct implementation without race conditions?

```
1 is necessary to lock 'n'
  -  Answering '2' is partial credit if 2nd lock protects 'searching' (not
     strictly necessary in this example).
```

Extra Work (“Overflow Box”):

You may put additional answers here so long as you indicate which question the work belongs to and indicate in the original answer box that you put an answer in the ‘overflow box’.

Reference Material – These Pages NOT Scanned

C++ STL and language features

- If lst is a STL vector, then lst.begin() and lst.end() return iterator values of type vector<...>::iterator. STL lists and sets are similar.

- A STL map is a collection of pair objects. If p is a pair, then p.first and p.second denote its two components. If the pair is stored in a map, then p.first is the key and p.second is the associated value.

- If m is a map, m.begin() and m.end() return iterator values. For a map, these iterators refer to the Pair objects in the map.

- If itr is an iterator, then *itr can be used to reference the item it currently points to, and ++itr will advance it to the next item, if any.

- Some useful operations on STL containers (lists, maps, sets, etc.):
  - c.clear() – remove all elements from c
  - c.size() – return number of elements in c
  - c.empty() – true if number of elements in c is 0, otherwise false

- Additional operations on vectors:
  - c.push_back(x) – copy x to end of c

- Some additional operations on maps:
  - m.insert(x) – add copy of x to m (a key-value pair for a map)
  - m.count(x) – number of elements with key x in m (0 or 1)
  - m.find(item) – iterator pointing to element with key that matches item if found, or m.end() if not found.
  - m[k] can be used to access the value associated with key k. If m[k] is read and has never been accessed before, then a <key,value> Pair is added to the map with k as the key and with a value created by the default constructor for the value type (0 or nullptr for primitive types).

- Some additional operations on sets
  - s.insert(x) – add x to s if not already present
  - s.count(x) – number of copies of x in s (0 or 1)

- You may use the C++11 auto keyword, C++11-style for-loops for iterating through containers, and any other features of standard C++11, but you are not required to do so.

Useful function reference related to pthread mutexes

```
#include <pthread.h>

// Locking a mutex
int pthread_mutex_lock(pthread_mutex_t *mutex);

// Unlocking a mutex
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Useful function reference related to POSIX I/O

```
#include <fcntl.h>

// Opening a file; returns file descriptor, -1 on error
int open(const char *path, int flags, ...
                           /* mode_t mode */ );
```

```
#include <unistd.h>

// Closing a file
int close(int fd);

// Read from a file; returns bytes read, 0 at EOF, -1 on error
ssize_t read(int fd, void buf[count], size_t count);

// Write to a file; returns bytes written, -1 on error
ssize_t write(int fd, const void buf[count], size_t count);
```

```
#include <stdio.h>

// Print a system error message
void perror(const char *s);
```

POSIX Networking

DNS and address-related:

```
int getaddrinfo(const char *restrict node,
                const char *restrict service,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);

struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

Sockets

```
// Create a new socket
int socket(int domain, int type, int protocol);
// domain is AF_INET or AF_INET
// type is SOCK_STREAM for TCP, SOCK_DGRAM for UDP, or
// SOCK_RAW for a "raw" IP socket
// protocol is usually provided for us from getaddrinfo()

// Bind listener sockets to a specific port
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);

// Open the connection queue
int listen(int sockfd, int backlog);

// Accept a waiting connection on a listener socket
// and return a new client socket for that connection
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);

// Connect a client socket to the remote address specified
// by addr
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);

int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

```
int getsockopt(int socket, int level, int option_name,
               void *option_value, socklen_t *option_len);


struct sockaddr {
    sa_family_t     sa_family;      /* Address family */
    char            sa_data[];      /* Socket address */
};

struct sockaddr_storage {

    sa_family_t     ss_family;      /* Address family */

};

typedef /* ... */ socklen_t;

typedef /* ... */ sa_family_t;

struct sockaddr_in {
    sa_family_t     sin_family;     /* AF_INET */
    in_port_t       sin_port;       /* Port number */
    struct in_addr  sin_addr;       /* IPv4 address */
};

struct sockaddr_in6 {
    sa_family_t     sin6_family;    /* AF_INET6 */
    in_port_t       sin6_port;      /* Port number */
    uint32_t        sin6_flowinfo;  /* IPv6 flow info */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t        sin6_scope_id;  /* Set of ifaces */
};

struct in_addr {
    in_addr_t s_addr;
};

struct in6_addr {
    uint8_t   s6_addr[16];
};

typedef uint32_t in_addr_t;

typedef uint16_t in_port_t;
```