University of Washington – Computer Science & Engineering

CSE 333 A Autumn 2023 Final: Version A

Last Name:	LastName
First Name:	FirstName
Student ID Number:	1234567
UWNetID:	lastfirst
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)**	*FirstName LastName*

Do not turn the page until 14:30.

Instructions

- This exam contains 15 pages, including this cover page. Put your final answers in the boxes and blanks provided. You may make use of the 'overflow box' on the last page for additional answer space.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed one 3"x5" index card (double-sided) of *handwritten* notes.
- Please silence and put away all cell phones and other mobile or noise-making devices.

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 are here to learn.

Question	1	2	3	4	5	6	Total
Possible Points	25	30	16	26	14	1	112

108 total

22 – Q4 E part 2 was made a bonus

Question 1: Writing robust code. (Continued on next page.)

findrange.cc

```
#include <algorithm> // for min, max
#include <iostream>
#include <vector>

using std::vector;

int FindMaxInt(vector<int> &v) {
  int ourmax = v[0];
  for(auto &e : v) {
    // std::max returns the maximum value of its two arguments
    ourmax = std::max(ourmax, e);
  }
  return ourmax;
}

double FindMaxDouble(vector<double> &v) {
  double ourmax = v[0];
  for(auto &e : v) {
    // std::max returns the maximum value of its two arguments
    ourmax = std::max(ourmax, e);
  }
  return ourmax;
}

int FindMinInt(vector<int> &v) {
  int ourmin = v[0];
  for(auto &e : v) {
    // std::min returns the minimum value of its two arguments
    ourmin = std::min(ourmin, e);
  }
  return ourmin;
}

double FindMinDouble(vector<double> &v) {
  double ourmin = v[0];
  for(auto &e : v) {
    // std::min returns the minimum value of its two arguments
    ourmin = std::min(ourmin, e);
  }
  return ourmin;
}

int main(int argc, char *argv[]) {
  // DO NOT MODIFY START
  vector<int> v1 = {1, 2, 3, 4, 5};
  vector<double> v2 = {1.0, 2.0, 3.0, 4.0, 5.0};
  // DO NOT MODIFY END

  std::cout << "int range: ["
            << FindMinInt(v1) << ", "
            << FindMaxInt(v1) << "]" << std::endl;

  std::cout << "double range: ["
            << FindMinDouble(v2) << ", "
            << FindMaxDouble(v2) << "]" << std::endl;

  return EXIT_SUCCESS;
}
```

Question 1 (cont'd): [25 pts] Systems programming requires your code to be robust and efficient. A good strategy is to write code that properly handles errors, doesn't repeat itself, and minimizes the probability of new errors being introduced when functions you write operate on different data. Use what you have learned in this course to rewrite `findrange.cc` into a smaller program that adheres to best practices we have discussed, uses the same vectors `v1` and `v2` unmodified, and produces the same output. There are many possible rewrites that would result in full marks, but your solution should be smaller than the current implementation and fit into the space below.

```
#include <algorithm>
#include <iostream>
#include <vector>

// Document assumption that v is non-empty.
template<typename T>
T FindMax(const std::vector<T> &v) {
  T ourmax = v[0];
  for(auto &e : v) {
    ourmax = std::max(ourmax, e);
  }
  return ourmax;
}

// Document assumption that v is non-empty.
template<typename T>
T FindMin(const std::vector<T> &v) {
  T ourmin = v[0];
  for(auto &e : v) {
    ourmin = std::min(ourmin, e);
  }
  return ourmin;
}

template<typename T>
void PrintRange(const std::vector<T> &v, const char *type_label) {
  // Avoid crashing on empty vector (document as no-op, or error msg, etc).
  if (v.empty()) return;
  std::cout << type_label << " range: [" << FindMin(v)
            << ", " << FindMax(v) << "]" << std::endl;
}

int main(int argc, char *argv[]) {
  using std::vector;
  // DO NOT MODIFY START
  vector<int> v1 = {1, 2, 3, 4, 5};
  vector<double> v2 = {1.0, 2.0, 3.0, 4.0, 5.0};
  // DO NOT MODIFY END
  vector<int> v3; // empty vector may be possible in future data

  PrintRange(v1, "int");
  PrintRange(v2, "double");
  PrintRange(v3, "int");

  return EXIT_SUCCESS;
}

// // With more knowledge of the STL algorithms it is possible to replace
// // the three utility functions above into the single function below.
// template<typename T>
// void PrintRange(const std::vector<T> &v, const char *type_label) {
//  // Avoid crashing program when vector is empty by either
//  // documenting this case as a no-op or print warning, etc.
//  if (v.empty()) return;
//  const auto [min, max] = std::minmax_element(std::begin(v), std::end(v));
//  std::cout << type_label << " range: ["
//            << *min << ", " << *max << "]" << std::endl;
//}
```

Question 2: Onions, ogres, and inheritance!

Consider the following C++ program, which compiles and executes successfully.

shrek.cc

```
1  #include <iostream>
2  using namespace std;
3
4  class Onion {
5   public:
6    virtual void m1() { cout << "o"; }
7    void m2() { cout << "J"; }
8    void m3() { cout << "k"; }
9  };
10
11 class Ogre : public Onion {
12  public:
13   void m1() { cout << "u"; }
14   virtual void m2() { cout << "g"; }
15   virtual void m3() { cout << "u"; }
16   void m4() { cout << "y"; }
17 };
18
19 class Shrek : public Ogre {
20  public:
21   virtual void m1() { cout << "o"; }
22   void m2() { cout << "!"; }
23   virtual void m3() { cout << "o"; }
24   void m4() { cout << "!"; }
25   void m5() { cout << "n"; }
26 };
27
28 int main() {
29   Shrek s;
30   Ogre o;
31
32   Onion* onion = &o;
33   Ogre* ogre = &s;
34   Shrek* shrek = &s;
35
36   onion->m2();
37   ogre->m3();
38   shrek->m5();
39   ogre->m2();
40
41   onion->m3();
42   onion->m1();
43   shrek->m3();
44   onion->m3();
45   ogre->m4();
46
47   cout << endl;
48 }
49
```

(A) [10pts] What does this program print to the standard output stream when it executes?

```
Jon!kuoky
```

(B) [12pts] Modify the above program by removing and / or adding the virtual keyword in appropriate place(s) so that the modified program prints Jungkook! (including the ! at the end). For each method on the listed line number indicate whether the method should be dynamically dispatched by marking the corresponding box with an 'X'.

Line Number	Method	Virtual	Non-virtual
6	`m1()`		X
7	`m2()`		X
8	`m3()`		X
13	`m1()`	X (either this one)	X (or this one)
14	`m2()`		X
15	`m3()`		X
16	`m4()`	X	
21	`m1()`	X (either this one)	X (or this one)
22	`m2()`	X (either this one)	X (or this one)
23	`m3()`	X (either this one)	X (or this one)
24	`m4()`	X (either this one)	X (or this one)
25	`m5()`	X (either this one)	X (or this one)

(C) [2 pts] Assume we add the virtual keyword to every method and replace the code in main with the given lines below.

```
Onion* onion = new Ogre();
onion->m4();
```

Write the output that is produced when the program is executed. If an error occurs, give a concise description of the problem and the type of error (runtime or compile time).

Compile error - class `Onion` doesn't have `m4()` which is in class `Ogre`

(D) [2 pts] Rewrite the method declaration and / or method definition of `m2()` in class `Onion` so that it is a pure virtual function.

```
virtual void m2() = 0;
```

(E) [4 pts] Suppose that method `m2()` in class `Onion` was correctly rewritten to become a pure virtual function. Indicate for each statement below whether it is true or false by marking the corresponding box with an 'X'

	True	False
Class `Onion` is now an abstract class.	X	
We are unable to create instances of class `Onion`.	X	
A class containing only pure virtual functions can be thought of as a Java interface.	X	
We are unable to execute method `m2()` from class `Onion` without overriding it.	X	

Question 3: NETWORKING

(A) [11 pts] Recall that there are 7 steps of server-side network programming. Fill in the blanks below to describe those 7 steps.

1. Get local IP address and port.

2. Create socket.

3. Bind the socket to local IP address and port.

4. Listen on socket

5. Accept connection from a client.

6. Process that request by reading and writing data on that connection.

7. Close the socket.

(B) [5 pts] For each statement below indicate whether it is true or false using an 'X' in the respective column.

	True	False
TCP is a client/server protocol (one server to one client).	X	
UDP is a client/server protocol (one server to one client).	X	
UDP has error control mechanisms for dropped packets.		X
TCP prioritizes speed compared with UDP.		X
POSIX networking function calls are always blocking.		X

Question 4: Concurrency

Consider the following C++ code which uses pthreads for concurrency.

boundedint.cc

```
#include <pthread.h>
#include <cstdlib>
#include <algorithm>
#include <iostream>
#include <memory>

static const int kTimes = 100;
static unsigned int max_obs = 0;

class BoundedInt {
 public:
  BoundedInt(unsigned int maxv)
    :value_(0),
     max_value_(maxv) {
    pthread_mutex_init(&mtx_,
                       nullptr);
  }

  ~BoundedInt() {
    pthread_mutex_destroy(&mtx_);
  }

  void Increment() {
    //pthread_mutex_lock(&mtx_);
    if (value_ < max_value_) {
      value_ += 1;
    }
    //pthread_mutex_unlock(&mtx_);
  }

  void Decrement() {
    if (value_ > 0) {
      value_ -= 1;
    }
  }

  unsigned int Value() {
    return value_;
  }

 private:
  unsigned int value_;
  unsigned int max_value_;
  mutable pthread_mutex_t mtx_;
}; // class BoundedInt

// file continues in right column
```

```
// file continues here after left column
void* incrementer(void *arg) {
  BoundedInt *b = dynamic_cast<BoundedInt*>(arg);
  for (auto i = 0; i < kTimes; ++i) {
       b->Increment();
  }
  return nullptr;
}

void* decrementer(void *arg) {
  BoundedInt *b = dynamic_cast<BoundedInt*>(arg);
  for (auto i = 0; i < kTimes; ++i) {
       b->Decrement();
  }
  return nullptr;
}

void* max_observer(void *arg) {
  BoundedInt *b = dynamic_cast<BoundedInt*>(arg);
  for (auto i = 0; i < kTimes; ++i) {
       // std::max returns the maximum of its two arguments
       max_obs = std::max(max_obs, b->Value());
  }
  return nullptr;
}


int main(int argc, char *argv[]) {
  BoundedInt a(333);
  pthread_t t1, t2, t3;

  if (pthread_create(&t1, nullptr, incrementer, &a) != 0 ||
      pthread_create(&t2, nullptr, decrementer, &a) != 0 ||
      pthread_create(&t3, nullptr, max_observer, &a) != 0)
  {
    return EXIT_FAILURE;
  }
  if (pthread_join(t1, nullptr) != 0 ||
      pthread_join(t2, nullptr) != 0 ||
      pthread_join(t3, nullptr) != 0) {
    return EXIT_FAILURE;
  }

  std::cout << "Value: " << a.Value() << std::endl;
  std::cout << "Max Observed: " << max_obs << std::endl;

  return EXIT_SUCCESS;
}
```

(Question continued on the next page.)

(A) [6 pts] The program listed in `boundedint.cc` does not currently compile due to three bugs that either misuse a library function or a language feature. Write down the three lines of the current code that are causing the compilation failure. Do not fix the code in the box immediately below. For example, if you think the return statement of `main` causes a compilation failure list the code "`return EXIT_SUCCESS;`" in your answer.

```
1.  BoundedInt *b = dynamic_cast<BoundedInt*>(arg);

2.  BoundedInt *b = dynamic_cast<BoundedInt*>(arg);

3.  BoundedInt *b = dynamic_cast<BoundedInt*>(arg);
```

Rewrite those three lines of code below so that `boundedint.cc` now compiles. Ensure you list the rewritten code lines in the same order you listed them above.

```
1.  BoundedInt *b = reinterpret_cast<BoundedInt*>(arg);

2.  BoundedInt *b = reinterpret_cast<BoundedInt*>(arg);

3.  BoundedInt *b = reinterpret_cast<BoundedInt*>(arg);
```

(B) [5 pts] Assume your program now compiles with your modifications which, importantly, do not change the logic of the program listed in `boundedint.cc`. For each statement below indicate whether it is always true or always false by marking with an 'X' in the respective column. If a statement can sometimes be true and sometimes be false on different program executions, then indicate so by marking with an 'X' in both columns.

	True	False
The printed output will report that `a.Value()` and `max_obs` are equal.	X	X
The printed output will report `a.Value()` equals 0.	X	X
The printed output will report `a.Value()` equals 333 or 100.	X	X
The printed output will report `max_obs` equals 0.	X	X
The printed output will report `max_obs` equals 333 or 100.	X	X

(C) [3 pts] We continue our modifications and next replace the value of `kTimes` in our program to be 1000 (*e.g.,* the corresponding line now reads `static const int kTimes = 1000;`). For each statement below indicate whether it is always true or always false by marking with an 'X' in the respective column. If a statement can sometimes be true and sometimes be false on different program executions, then indicate so by marking with an 'X' in both columns.

	True	False
The printed output will report that `a.Value()` and `max_obs` are equal.	X	X
The printed output will report `a.Value()` with a value greater than 333.		X
The printed output will report `max_obs` with a value greater than 333.		X

(D) [6 pts] Building on our previous modification from part (C) we alter our main function by replacing the function call `pthread_create(&t3, nullptr, max_observer, &a)` with the function call `pthread_create(&t3, nullptr, incrementer, &a)`. For each statement below indicate whether it is always true or always false by marking with an 'X' in the respective column. If a statement can sometimes be true and sometimes be false on different program executions, then indicate so by marking with an 'X' in both columns. [Note: `kTimes = 1000;`]

	True	False
The printed output will report `a.Value()` with a value greater than 333 but less than 667.	X	X
The printed output will report `a.Value()` with a value greater than 2000.		X

(Part D continues on the next page.)

(Part D continued.) If you believe the reported value of `a.Value()` cannot exceed 333 then briefly justify why below. If you believe otherwise, then justify why and also describe the minimum possible modifications you would make to the code to ensure this cannot happen. Pseudocode is fine as we do not expect you to recall exact syntax of library functions that may be helpful.

The reported value of a.Value() can exceed 333. Here is one execution:

1. Thread t3 does not begin to run until threads t1 and t2 complete.
2. Thread t1 calls and returns from b->Increment() 332 times to bring the value from 0 to 332.
3. Thread t1 calls b->Increment() and passes the conditional.
4. Thread t2 calls b->Increment() and passes the conditional.
5. Thread t1 and t2 complete their respective calls to b->Increment() and after they both return the value will now be 334.

To ensure the bound cannot be exceeded we need only uncomment the lines in BoundedInt::Increment() that requires a thread to obtain a mutex that guards the critical section of code. Note that we do not need to add a mutex in BoundedInt::Decrement(), in this particular case, if all we care to do is avoid exceeding the bound since t3 is the only thread that decreases the value and will only ever do so if the current value is greater than 0 (and thus it avoids underflow). However, to ensure overall correctness we *should* also use a mutex around the critical section of code in Bounded::Decrement().

2 pts – Q4 E part 2 was made a bonus

(E) [6 pts] Building on our previous modification from part (C) we instead alter our main function by replacing the function call `pthread_create(&t3, nullptr, max_observer, &a)` with the function call `pthread_create(&t3, nullptr, decrementer, &a)`. For each statement below indicate whether it is always true or always false by marking with an 'X' in the respective column. If a statement can sometimes be true and sometimes be false on different program executions, then indicate so by marking with an 'X' in both columns. [Note: `kTimes = 1000;`]

	True	False
The printed output will report `a.Value()` with a value greater than 333 but less than 667.		X
The printed output will report `a.Value()` with a value between 0 and 2000.	X	X

(Part E continues on the next page.)

(Part E continued.) If you believe the reported value of `a.Value()` cannot exceed 333 then briefly justify why below. If you believe otherwise, then justify why and also describe the minimum possible modifications you would make to the code to ensure this cannot happen. Pseudocode is fine as we do not expect you to recall exact syntax of library functions that may be helpful.

The reported value of a.Value() can exceed 2000, and thus 333, via an execution that includes an underflow (*e.g.,* subtracting less than the current value of an unsigned int). Note that a.Value() cannot be a value greater than 333 but less than 667. Here is one execution:

1. Thread t1 calls b->Increment() which returns bringing the value to 1.
2. Thread t2 calls b->Decrement() and passes the conditional.
3. Thread t3 calls b->Decrement() and passes the conditional.
4. Thread t2 returns from b->Decrement() and the value is now 0.
5. Thread t3 returns from b->Decrement() and the value underflows to the maximum value of an unsigned int.
6. Thread t2 calls and returns from b->Decrement() 332 more times.
7. Thread t1 calls and return from b->Increment() 332 more times.
8. Thread t3 calls and returns from b->Decrement 332 more times. The value is now 332 less than the maximum value of an unsigned int (a.Value() > 2000).

To ensure the bound cannot be exceeded, in this particular case, we *must* obtain a mutex that guards the critical section of code in BoundedInt::Decrement() *should* do the same for BoundedInt::Increment() to ensure correctness.

Bonus of 4 pts if and only if a detailed execution was given that demonstrated an underflow and also listed an appropriate fix. No deductions were applied to this question.

Question 5: Reflection on concepts explored in homework and the course.

These reflection questions only require a brief description.

(A) [4 pts] Homework 4 asked you to follow the hypertext transfer protocol.

Briefly describe one convenient feature of this protocol in sending and receiving requests or responses.

One (of many) reasonable answers:

HTTP is in plain text and human readable. This makes it very convenient during development to create and to understand requests and responses.

Briefly describe one annoyance or difficulty you encountered with this protocol.

One (of many) reasonable answers:

Conversely, since HTTP is in plain text and human readable, our code needed to correctly parse HTTP before it could be processed. It is likely that a protocol without these properties would be easier to support programmatically.

(B) [4 pts] The POSIX API treats all I/O operations similarly whether it is file, network, or other I/O.

Briefly describe one advantage of this design decision.

One (of many) reasonable answers:

A developer need only learn a common API instead of multiple different APIs to handle different types of I/O. For example, if a developer is already familiar with network programming in POSIX then they will likely be able to implement POSIX file I/O using the same principles.

Briefly describe one disadvantage of this design decision.

One (of many) reasonable answers:

Conversely, a common API forces the function signatures / parameters to be generalized to handle all the differences that arise between the different forms of I/O that the API support.

(C) [6 pts] We explored multiple forms of concurrency in this course including using multiple threads of execution and multiple processes.

Briefly describe one scenario where using multiple processes can be more advantageous than using multiple threads of execution within a single process. Justify why this is the case.

One (of many) reasonable answers:

Using a different process for each distinct connection in a client side web browser helps ensure that one rogue website will not crash the entire browser. This is because processes do not share resources whereas in this scenario, if implemented with multiples threads in a single process, a single website may crash the entire process (all threads of execution).

Briefly describe one scenario where using multiple threads of execution within a single process can be more advantageous than using multiple processes. Justify your answer.

One (of many) reasonable answers:

Multiple threads within a single process can share resources with the same virtual address space (e.g., large data structures) and thus can make more efficient use of system resources.

Question 6:

[1 pt; All non-empty answers receive this point] Select one member of the course staff other than the one you chose on the final question of the midterm (though we won't check ☺). Describe or draw an emoji representing that person.

☺

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'.