CSE 333 26su

Final Exam

Do not turn the page until you are told to do so.

First Name	**Sample Solution**
Last Name	
UW NetID	

Instructions

- The exam is closed book.
- You are allowed two pages of handwritten notes. Double -sided is ok.
- Silence and put away all cell phones and other mobile or noise-making devices.
- Remove all hats, headphones, smart glasses, watches, and other digital wearables.
- Keep your exam booklet on the table so the people behind you cannot see your answers.
- You will have 1 hour and 55 minutes to complete this exam.

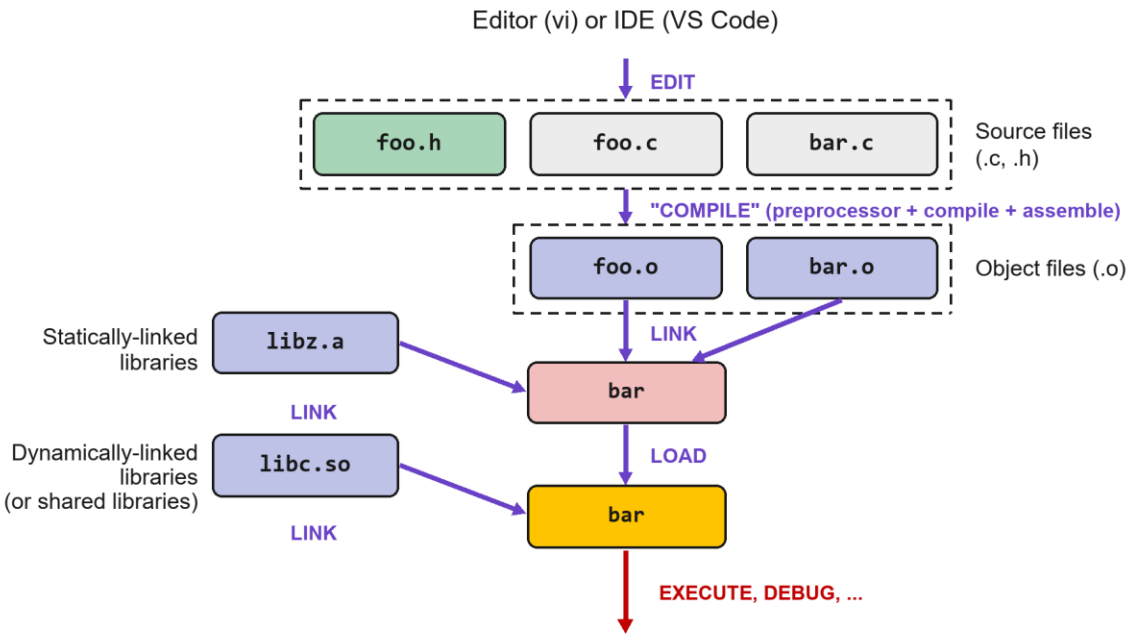
Advice

- Read all questions first and start where you feel the most confident.
- Read questions carefully before starting.
- Skip questions that are taking a long time.
- Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- Relax. If you've been practicing, you got this! If you haven't, you'll learn something new!

Question	1	2	3	4	5	6	7	8*	9*	Total
Possible Points	10	12	16	14	10	8	12	8	10	100

* Questions 8 and 9 will be personalized to each student. We will hand this out once this exam starts.

Question 1. Compilation (10 pts)



Part A. The diagram groups preprocessing, compilation, and assembly into the "COMPILE" phase. Explain what happens in each of these three parts and what is passed to the next part. [6 pts]

- Preprocessing handles #include, #define, and conditional directives, then passes expanded source code to the compiler.
- Compilation checks the program types and translates that source into assembly code.
- The assembler translates the assembly code into a relocatable object file containing machine instructions, symbols, and relocation information.

Part B. Explain what happens during the linking phase. Then, explain the difference shown between a static library and a dynamic/shared library. [4 pts]

The linker combines object files and needed libraries, matches each referenced name with its definition, fixes addresses, and produces the executable. It verifies that function declarations are linked to valid function definitions.

- A static library is copied into the executable at link time.
- A dynamically loaded library (DLL) or shared library stays separate and is loaded when the program starts, so the executable is smaller but needs that library at run time.

Question 2. OSI Model (12 pts)

#	OSI layer name
7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link
1	Physical

Part A. Fill in the name of each numbered layer in the seven-layer OSI model. [7 pts]

Part B. For each layer listed below, explain its primary responsibility. Be specific about what kind of information or delivery the layer provides. [5 pts]

(i) Layer 1

Physical: sends raw bits as electrical, light, or radio signals across one physical link.

(ii) Layer 3

Network: moves packets from one host to another across networks by using addresses and choosing routes.

(iii) Layer 4

Transport: delivers data from one process to another using ports. It splits and rebuilds data; TCP also provides ordered, reliable delivery.

Question 3. Inheritance (16 pts)

The following class hierarchy represents several kinds of performers. The program compiles without error.

```cpp
class Performer {
 public:
    virtual ~Performer() = default;

    void Introduce() {
        std::cout << "Performer::Introduce" << std::endl;
        Act();
    }

    virtual void Act() {
        std::cout << "Performer::Act" << std::endl;
    }

    void Encore() {
        std::cout << "Performer::Encore" << std::endl;
    }
};

class Magician : public Performer {
 public:
    void Introduce() {
        std::cout << "Magician::Introduce" << std::endl;
        Reveal();
    }

    void Act() override {
        std::cout << "Magician::Act" << std::endl;
    }

    void Encore() {
        std::cout << "Magician::Encore" << std::endl;
    }

    virtual void Reveal() {
        std::cout << "Magician::Reveal" << std::endl;
    }
};

class Juggler : public Performer {
 public:
    void Act() override {
        std::cout << "Juggler::Act" << std::endl;
    }
};

class Illusionist : public Magician {
 public:
    void Act() override {
        std::cout << "Illusionist::Act" << std::endl;
        Magician::Act();
    }

    void Encore() {
        std::cout << "Illusionist::Encore" << std::endl;
    }

    void Reveal() override {
        std::cout << "Illusionist::Reveal" << std::endl;
    }
};
```

Part A. Write the complete output, in order, exactly as it is printed. You do NOT need to write code. [8 pts]

```
1  Performer* pp = new Performer();
2  Performer* pm = new Magician();
3  Magician*  mi = new Illusionist();
4  Performer* pj = new Juggler();
5  Magician*  mm = new Magician();
6
7  pp->Introduce();  // (1)
8  pm->Introduce();  // (2)
9  mi->Introduce();  // (3)
10 pm->Encore();     // (4)
11 mi->Encore();     // (5)
12 pj->Act();        // (6)
13 mm->Reveal();     // (7)
14 mi->Act();        // (8)
```

```
(1) Performer::Introduce
    Performer::Act
(2) Performer::Introduce
    Magician::Act
(3) Magician::Introduce
    Illusionist::Reveal
(4) Performer::Encore
(5) Magician::Encore
(6) Juggler::Act
(7) Magician::Reveal
(8) Illusionist::Act
    Magician::Act
```

Part B. Briefly state the difference between static and dynamic dispatch. [4 pts]

Static dispatch applies to non-virtual calls: the compiler selects the method from the expression's declared (static) type. Dynamic dispatch applies to virtual calls: at run time, it selects the most-derived override for the object's actual (dynamic) type. Part C classifies the method named in each numbered statement; for example, (2) statically selects Performer::Introduce(), but the virtual Act() inside it dynamically selects Magician::Act().

Part C. For each statement in Part A, circle either static or dynamic to indicate the type of dispatch. [4 pts]

(1)	◯ **Static**	Dynamic	(5)	◯ **Static**	Dynamic
(2)	◯ **Static**	Dynamic	(6)	Static	◯ **Dynamic**
(3)	◯ **Static**	Dynamic	(7)	Static	◯ **Dynamic**
(4)	◯ **Static**	Dynamic	(8)	Static	◯ **Dynamic**

Question 4. Object Slicing and STL (14 pts)

The code below is used by a museum to keep track of its exhibits. Note the virtual functions.

```
class Exhibit {                                                    // Base class
 public:
    explicit Exhibit(const std::string& name) : name_(name) {}
    virtual ~Exhibit() = default;

    void Rename(const std::string& name) {
        name_ = name;
    }

    virtual void Print() const {                                   // virtual function
        std::cout << "Exhibit: " << name_ << std::endl;
    }

 protected:
    std::string name_;
};

class Artifact : public Exhibit {                                  // Derived class
 public:
    Artifact(const std::string& name, const int year)
            : Exhibit(name), year_(year) {}

    void Print() const override {
        std::cout << "Artifact: " << name_ << " (" << year_ << ")" << std::endl;
    }

 private:
    const int year_;
};
```

To use this code, they create an STL map to keep track of which exhibit is in which room of the museum.

```
int main() {
    Exhibit fossil("Dinosaur Skeleton");
    Artifact moon_rock("Moon Rock", 1969);

    // Map of room # to exhibit
    std::map<int, Exhibit> catalog;
    catalog.emplace(2, fossil);
    catalog.emplace(3, moon_rock);

    // CHECKPOINT

    moon_rock.Rename("Apollo Sample");

    moon_rock.Print();
    catalog.at(3).Print();
}
```

Read the complete program on this page, then answer some questions on the next page.

Part A. What is printed at the end of the program (i.e. after line 16 of main)? [2 pts]

```
Artifact: Apollo Sample (1969)
Exhibit: Moon Rock
```

Part B. Do STL containers store their elements by-reference or by-value? Explain how this relates to "object slicing" and how your previous answers support this claim. (Hint: consider lines 6-8). [3 pts]

By value. The map's value type is Exhibit, so inserting moon_rock copies only its Exhibit part and drops its Artifact part. The map also owns a separate copy, so renaming the original does not rename the map entry. That is why the second line uses Exhibit::Print and the old name.

Part C. One way to fix "object slicing" is to store pointers instead of base-class objects. Explain why. [3 pts]

A pointer can still point to the complete Artifact object even when its pointer type is Exhibit*. The map stores the pointer instead of copying an Exhibit, so virtual Print sees the object's actual type.

Part D. Rewrite the main function using `std::unique_ptr<Type>` and `std::make_unique<Type>`.

You will need to preserve the functionality of main, including to create new objects, write a new map declaration, rename the moon rock Artifact, and to call Print on different objects. [6 pts]

```
int main() {
  std::map<int, std::unique_ptr<Exhibit>> catalog;
  catalog.emplace(2, std::make_unique<Exhibit>("Dinosaur Skeleton"));
  catalog.emplace(3, std::make_unique<Artifact>("Moon Rock", 1969));

  catalog.at(3)->Rename("Apollo Sample");
  catalog.at(2)->Print();
  catalog.at(3)->Print();
}
```

Constructing with std::make_unique directly in emplace is concise and moves the temporary into the map. An existing std::unique_ptr also works if it is explicitly moved, for example catalog.emplace(3, std::move(ptr));. Copying would fail because std::unique_ptr represents exclusive ownership.

Question 5. Templates (10 pts)

The following two files are compiled together. Some functions compile, while others do not compile because of template errors. Read both files before answering the questions.

combine.h

```
#ifndef COMBINE_H_
#define COMBINE_H_

template <typename T>
T CombineLargerFirst(const T& first, const T& second) {
    if (first < second) {
        return second + first;
    }
    return first + second;
}

#endif  // COMBINE_H_
```

combine.cc

```
#include "combine.h"

class Score {
 public:
    explicit Score(int points) : points_(points) {}

    bool operator<(const Score& other) const {
        return points_ < other.points_;
    }

 private:
    int points_;
};

int CombineInts() {
    return CombineLargerFirst(20, 22);                  // Call A
}

double CombineMixed() {
    return CombineLargerFirst(20, 22.5);                // Call B
}

double CombineMixedExplicitly() {
    return CombineLargerFirst<double>(20, 22.5);        // Call C
}

Score CombineScores() {
    return CombineLargerFirst(Score(20), Score(22));    // Call D
}
```

Note that some of these template calls will produce compilation errors. Next, you will explain why.

Part A. Complete the table for Calls A-D. If no template type `T` is selected, explain why. If a type `T` is selected, state whether that function compiles. [4 pts]

Call	type T	Does it compile? Why or why not?
A	int	Yes. int provides both < and +.
B	No single T	No. One argument suggests int and the other suggests double.
C	double	Yes. The explicit type is double, and 20 converts to double.
D	Score	No. Score provides < but does not provide +.

Part B. Call D selects `T = Score`, but its specialization fails. List the operations required from `T`, identify which one `Score` provides, ~~and write only the declaration for the missing operation~~. [4 pts]

```
CombineLargerFirst needs < and +. Score provides <. It is missing:
Score operator+(const Score& other) const;
```

During the exam, I asked students to skip the last step ("and write only the declaration for the missing operation") since it would require memorizing operator overloading syntax. We would not grade on that.

Part C. Templates are nearly always defined in header files. Why? (Hint: think about what phase of compilation they are generated during. What does this mean about their access?) [2 pts]

The compiler creates the needed template version when it sees a call. It must be able to see the complete template definition in that .cc file, so the definition is usually placed in the included header.

Question 6. Concurrent Interleavings (8 pts)

Answer the following questions about which outputs are possible.

```
1  std::string message = "start";            // global variable
2
3  void* Worker(void* argument) {
4      char* suffix = static_cast<char*>(argument);
5
6      std::string snapshot = message;       // R: take a snapshot of our global
7      message = snapshot + suffix;          // W: replace global variable
8
9      std::printf("%s: %s\n", suffix + 1, message.c_str());         // P: print
10     return nullptr;
11 }
12
13 int main() {
14     pthread_t t1, t2;
15
16     char read_suffix[] = "-read";
17     char write_suffix[] = "-write";
18
19     pthread_create(&t1, nullptr, Worker, read_suffix);
20     pthread_create(&t2, nullptr, Worker, write_suffix);
21     pthread_join(t1, nullptr);
22     pthread_join(t2, nullptr);
23
24     std::printf("final: %s\n", message.c_str());                  // P: print
25 }
```

For each candidate below, circle YES if that exact three-line output could be produced by some legal interleaving under the stated execution model. Circle NO if it is impossible.

	Is it possible?	printf(...) output		
(a)	◯ **YES** NO	`read: start-read`	`write: start-write`	`final: start-write`
(b)	YES ◯ **NO**	`read: start-read`	`write: start-write`	`final: start-read-write`
(c)	◯ **YES** NO	`write: start-write`	`read: start-read`	`final: start-read`
(d)	YES ◯ **NO**	`write: start-read`	`read: start-write`	`final: start-write`
(e)	◯ **YES** NO	`read: start-read-write`	`write: start-read-write`	`final: start-read-write`
(f)	YES ◯ **NO**	`read: start-read-write`	`write: start-read-write`	`final: start-write`
(g)	◯ **YES** NO	`write: start-write`	`read: start-write-read`	`final: start-write-read`
(h)	YES ◯ **NO**	`write: start-write-read`	`read: start-read-write`	`final: start-read-write`

Question 7. Execution Models (12 pts)

For each task, pick the execution model that best fits the task: single-threaded, event-driven, multi-threading, and multi-processing. In 1-2 sentences, describe why that execution model makes sense.

Task 1: Simple Server. A server maintains thousands of mostly idle client sockets. When a socket becomes ready, the server reads a short message and sends a short reply. The response message is very short and the server does not need to perform much work on the CPU for each response.

Event-driven.
One event loop can wait for many mostly idle sockets and handle only the sockets that are ready. The short responses do not need worker threads. We are not CPU-bound (i.e. running out of CPU). We are IO-bound (i.e. we are waiting for a bunch of IO requests).

Task 2: File Copy. A command-line program opens one input file, copies its bytes to one output file, closes both files, and exits. It uses the POSIX API and calls read and write in a while loop.

Single-threaded.
This is one ordered stream from one input file to one output file, so a simple read/write loop is enough.

Task 3: Cat Compression. A user uploads 100 cat images. For each image, our program needs to run a compression program that is very CPU-intensive. It compresses each input file and writes the bytes to a different output file. The compression program runs on each file independently.

Multi-threading.
A fixed group of worker threads can compress independent images at the same time on different CPU cores. Each worker writes a separate output, so little data must be shared. If we were single-threaded, we would be bound by CPU.

Task 4: Grading Repositories. CSE 333 students push their code to GitLab. The TAs need to download all of the student repositories from GitLab onto Attu. They write a program that takes in a list of GitLab repositories and clones each one into a separate folder.

Event-driven. The TA's program is mostly IO-bound since we are downloading multiple files and have to wait for the downloads to complete. There is very little CPU-work happening here.

We would accept multi-threading only if you explained that you thought CPU was a concern and gave a good attempt to explain where the CPU limitation was coming from.
if you explained that each git clone

CSE 333 26su

Final Exam

Personalized Section

Please staple this to your other exam papers when turning in the exam.

Name	**First Last**
UW NetID	**StudentId**

This is the personalized section of your final exam. This quarter we have been giving you feedback on your exercise and homework submissions, working with you during office hours, and answering your questions on the discussion board.

Based on those interactions, we have written the following question just for you. It targets a concept that you've gotten feedback on previously in the course. This is a chance for you to show us that you've internalized that feedback and learned.

We wrote a personal message for each student!

Question 8. Personalized Question 1 (8 pts)

Solution: student-specific.

Question 9. Personalized Question 2 (10 pts)

Solution: student-specific.