University of Washington – Computer Science & Engineering
Spring 2026 Instructor: Naomi Alterman 2026-05-04

CSE 333 MIDTERM

Last Name:	Lutionino	
First Name:	Solvatore	
Student ID Number:	333	
Name of person to your Left | Right	Avon	Stringer
All work is my own. I had no prior knowledge of the exam content, nor will I share the content with others in CSE 333 who haven't taken it yet. Violation of these terms could result in a failing grade. **(please sign)**		

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

Instructions

- This quiz contains 12 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last sheet of paper is a reference sheet. *Detach it from the rest of the exam when you start.*
- The exam is closed book. You are allowed one page (US letter, double-sided) of *handwritten* notes.
- Please 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.
- You have 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. If you've been practicing, you got this. If you haven't, you'll learn something now.

Question	1	2	3	4	5	Total
Possible Points	10	18	15	20	1	**64**

(this page left blank for scratch space)

1. Makin' my way downtown (makefiles, 10 pt)

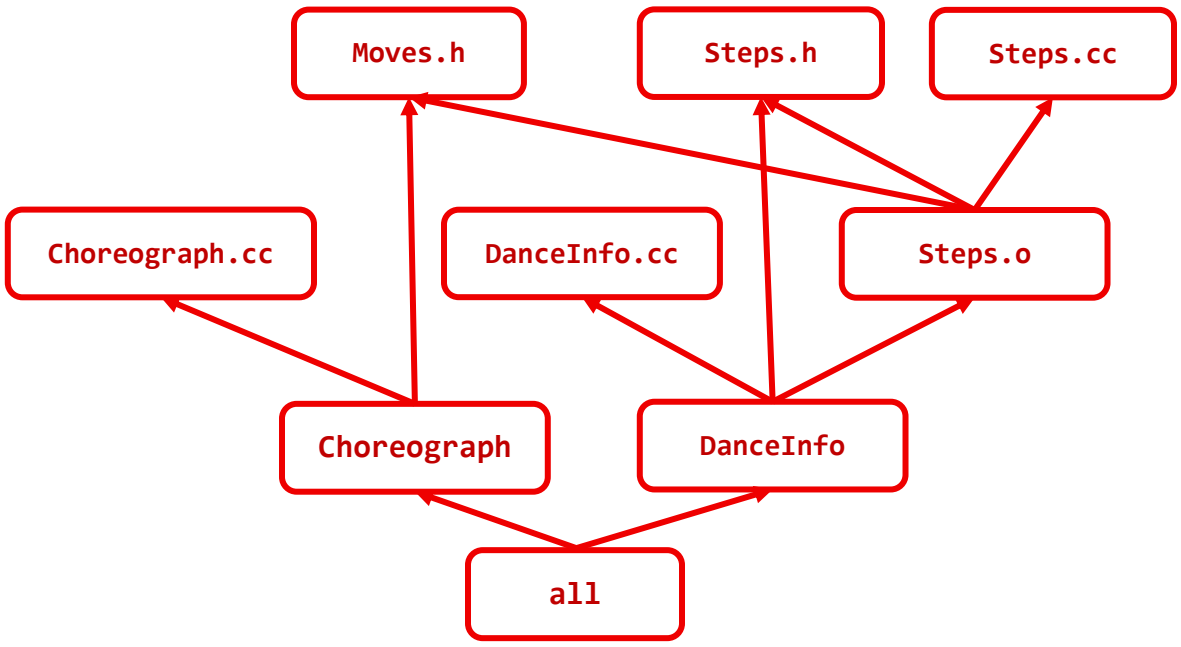
Our friend Jimin is working on an application suite related to choreographing dances. Here's a preview of the source code:

Moves.h	Steps.h	Steps.cc
`struct Wiggle {...};` `struct Shimmy {...};`	`class Steps {...};`	`#include "Steps.h"` `#include "Moves.h"`

Choreograph.cc	DanceInfo.cc

`#include "Moves.h"`
 	
`int main(...) {...}`
	`#include "Steps.h"`
	
	`int main(...) {...}`

The Makefile for this project has targets to build (or rebuild) the `Choreograph` and `DanceInfo` executables, as well as a default target named `all` that builds both. When we run **make all** we end up with the two programs and the object file **Steps.o**. No other files are generated in the process. Draw the dependency diagram for the Makefile, with arrows that point from targets to the predecessors they depend upon. [10 pt]



2. Keep it Classy (C++ Classes, 18pt)

Consider the following C++ program which does compile and execute successfully.

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

class bar {
 public:
  bar() : num_(new int())       { cout << "j"; }
  bar(int a) : num_(new int(a)) { cout << "y"; }
  ~bar()                        { cout << "l"; }

 private:
  int* num_;
};

class baz {
 public:
  baz()                                 { cout << "d"; }
  baz(int a) : morebar_(new bar(a))     { cout << "u"; }
  baz(int a, int b): val_(b)            { cout << "o"; }

 private:
  bar bar_;
  bar* morebar_;
  int val_;
};

class foo {
 public:
  foo()                       { cout << "e"; }
  foo(int a, int b, int c)
      : baz_(a, c), bar_(b)   { cout << "f"; }
  ~foo()                      { cout << "u"; }

 private:
  baz baz_;
  bar bar_;
};

int main() {
  foo f(7, 9, 4);
  return EXIT_SUCCESS;
}
```

(a) Write the output that is produced when the program is executed. [10 pt]

joyfull

Sequence:

1. foo initializes using its 3-int ctor
    a. it initializes foo.baz_ with its 2-int ctor
        i. foo.baz_.bar_'s default ctor is called
            1. an int is allocated
            2. "j" is printed
        ii. foo.baz_.val_ is initialized with the ctor argument
        iii. "o" is printed
    b. It initializes foo.bar_ with its 1-int ctor
        i. an int is allocated
        ii. "y" is printed
    c. "f" is printed
2. The program begins to exit and destructs foo
    a. The ~foo() body is called
        i. "u" is printed
    b. foo.bar_ is destructed
        i. "l" is printed
    c. foo.baz_ is destructed with a default synthesized destructor
        i. foo.baz_.bar_ is destructed
            1. "l" is printed

(b) Unfortunately, the code above is leakin' memory all over the place! How many bytes of memory will be leaked when `main()` exits? Describe a way to fix this problem. [3 pt]

8 bytes are leaked, both from the bar constructor allocating ints on the heap. In the bar destructor we should delete the memory for num_ (and also observe the rule of 3s to define a copy constructor and assignment operator, so we don't end up double-deleting memory).

We should follow the same advice for baz::morebar_ as well, though it's not actively a problem in the above code example.

(c) Suppose we add the following public methods to the class baz:

```
int baz::GetValue() { return val_; }

bool baz::Compare(baz &other_baz) const {
   return val_ > other_baz.GetValue();
}
```

And then try to compile this function:

```
void wump() {
      const baz b1(1, 2);
      baz b2(2, 3);
      cout << b2.GetValue();      // 1
      b1.Compare(b2);             // 2
      b2.Compare(b1);             // 3
}
```

Which, if any, of the three numbered lines in `wump()` could cause a compiler error? Why? [5 pt]

Line 3 will cause a compiler error because it attempts to pass the const object b1 to the non-const argument of baz::Compare

Line 1 is fine because b2 isn't const, so we can call the non-const method GetValue

Line 2 is fine because Compare is a const method, so we can call it on the const object b1. The non-const b2 is passed in as the non-const argument to that method.

3. A lot to process (preprocessor, 15pt)

Suppose we have the following three .h files and one .c file. On the next page, show the output produced by the C preprocessor when it processes file transaction.c with the following command:

```
cpp -E -P -DCOUPON=0.5 transaction.c
```

Remember! The C preprocessor only does string substitution and does not analyze the code it produces for correctness.

utils.h	costs.h

`int getQuantity(int id);`	`#include "utils.h"`
	
	`typedef float f;`
	
	`f getPrice(int id);`

users.h	**transaction.c**

`#include "utils.h"`
 	
`#define ITEM 1000`
 	
`#define USER 2`
 	
`void add_user(int id);`
	`#include "costs.h"`
	
	`#ifndef COUPON`
	`#define COUPON 1.0`
	`#endif`
	
	`#define PRICE(x) (x + 5.0)`
	`#define ITEM_BASE 300`
	`#define ITEM ITEM_BASE + 33`
	
	`int main(int argc, char** argv) {`
	`  int Q = getQuantity(ITEM);`
	`  f cost = Q * getPrice(ITEM);`
	`  printf("cost: %f", PRICE(cost) * COUPON);`
	`}`

Result of `cpp -E -P -DCOUPON=0.5 transaction.c` :

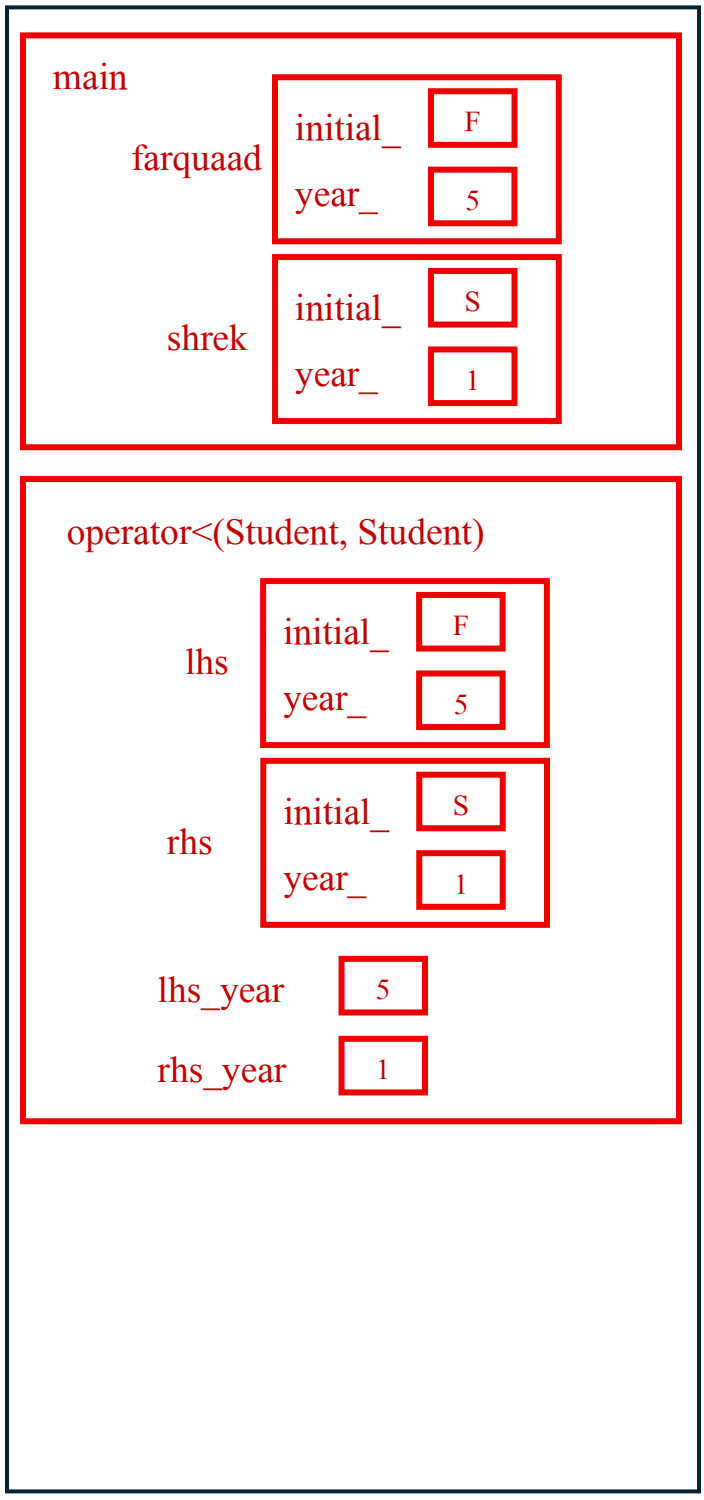
```
int getQuantity(int id);

typedef float f;

f getPrice(int id);

int main(int argc, char** argv) {
  int Q = getQuantity(300 + 33);
  f cost = Q * getPrice(300 + 33);
  printf("cost: %f", (cost + 5.0) * 0.5);
}
```

4. Stacks are like onions; they have layers (memory diagrams, 20 pt)

Consider the following C++ program which compiles and runs successfully:

```
#include <iostream>

using std::cout;
using std::endl;

class Student {
   public:
   Student(char initial, int year)
       : initial_(initial), year_(year) { }

   int GetYear() const { return this->year_; }
   void SetYear(int y) { this->year_ = y; }
   int GetInitial() const { return this->initial_; }

   private:
   char initial_;
   int year_;
};

bool operator<(Student lhs, Student rhs) {
   int lhs_year = lhs.GetYear();
   int rhs_year = rhs.GetYear();
   // PART A
   return lhs_year < rhs_year;
}

int main() {
  Student farquaad('F', 5);
  Student shrek('S', 1);

  // PART B

  if (farquaad < shrek) {
    cout << "Farquaad is in a lower grade than Shrek" << endl;
  } else {
    cout << "Farquaad is NOT in a lower grade than Shrek" << endl;
  }

  return EXIT_SUCCESS;
}
```

(a) Draw a block-and-arrow diagram below to represent the stack of the process when it reaches the line labelled "PART A". Clearly display the names, divisions between, and order of the stack frames. Assume high addresses are at the top of the diagram.
[15 pt]



(b) Suppose on the line indicated with "PART B", we add these lines of code to the program:

```
Student& shrek_twin = shrek;
shrek_twin.SetYear(2);
```

Briefly explain what would change in the memory diagram in Part A and why. [5 pt]

The boxes for the 'shrek' variable in the main stack frame would also be labelled 'shrek_twin' (both labels would be present). The year_ value in that object would hold a '2', and this '2' would also be there in the rhs copy in the operator< stack frame

5. Call me crazy (syscalls, 1 pt)

Make up your own syscall and tell us what it does. Any answer gets a point [1 pt]

int pet(CAT *);

Takes a handle to an active CAT and tries to pet it. Returns 0 on success or -1 on error See errno for cause: could be among other things, ENOTFED, EPLAYING, EBUSY (if being pet by someone else), EDOESNTKNOWYOU. Not all errors are fatal! EAGAIN means the pet was successful and you should actually continue petting the cat.

(this page left blank for scratch space)