CISC 3110 Final Review

Rough breakdown of topics:

50% Classes, Objects, and Object-Oriented Programming
10% Linux, g++, make
10% Testing
15% Pointers and dynamic allocation
15% Recursion

Study advice:

- Do exercises in the book (or from other readings).
- Review "Application Activities" questions, below (and from the midterm review)
- Review your solutions to the programming projects.

Note: you should expect most of the exam questions to be familiar. I will use a lot of the questions from the RATs and the application activities, though I may make some small changes or ask them in a different way. You should expect the final, like the midterm and the application activities, to be a mix of different kinds of questions.

1. Suppose we were trying to define an abstract data type for the Tic-Tac-Math board. Which of these would *not* belong to such a definition?

a) A Tic-Tac-Math board is a collection of 9 values, each between 1 and 9.
b) Empty cells on the board are represented as zeros.
c) A Tic-Tac-Math board is *winning* if three values from a "row," "column", or "diagonal" sum to 15.
d) Each value can only appear on the board once.

2. Consider the following abstract data type for a Deck of cards:

> A card must have one of four suits: hearts, diamonds, clubs, spades
> A card must have one of 13 values: two-ten, jack, queen, king, ace.
>
> A Deck *contains* 52 cards
> In a Deck that has been *shuffled*, cards are removed in random order
> You can *deal* the top card in a Deck by removing it

What's the problem:

a) This doesn't include any operations.
b) This doesn't fully specify the values.
c) This doesn't provide any rules of the game.
d) This DOES provide implementation details

3. Here's the `Circle` class from the book (the version from Figure 7.1, p 389):

```
class Circle {
   public:
      double radius = 0.0;

      void setRadius(double r) { radius = r; }
      double getArea() { return 3.14 * pow(radius, 2); }
};
```

Suppose we then declare a `Circle` object:

```
Circle roundThing;
```

Which of the following statements best illustrates the problem with this `Circle` definition?

a) `roundThing.setRadius(47);`
b) `std::cout << roundThing.radius;`
c) `double area = roundThing.getArea();`
d) `roundThing.radius = 0.5;`
e) `delete roundThing;`

4. The next version of `Circle` declares `radius` as a `private double`. What's the best implementation of `setRadius()`? (Assume any needed header files are `#include`-d)

a)
```
void Circle::setRadius(double r) { radius = r; }
```

b)
```
void Circle::setRadius() {
      std::cout << "Please enter the circle's radius ";
      std::cin >> radius;
      std::cout << std::endl;
}
```

c)
```
bool Circle::setRadius(double r) {
      if (r > 0.0) {
            radius = r;
            return true;
      } else {
            radius = 0;
            return false;
      }
}
```

d)
```
void Circle::setRadius(double r) {
      radius = r;
      std::cout << "Radius set; new area is " << getArea(); << std::endl;
}
```

e) `void Circle::setRadius() { radius = sqrt(getArea()/3.14); }`

5. Write a complete class called `Date`. Its public members must include these methods (return types are up to you):

```
setDate(int day, int month, int year);
getDay();
getMonth();
getYear();
```

6. Which is the best default constructor for the `Date` class? (Assume `day`, `month`, `year` are declared as `int` members.)

a) It would be best not to write one at all.

b) `Date::Date() { day = month = year = 0; }`

c) `Date::Date() { day = month = 1; year = 1970; }`

d)
```
Date::Date() {
      std::cout << “Enter values for day, month, and year, separated by space: ”;
      std::cin >> day >> month >> year;
}
```

7. Which is the best destructor for the `Date` class? (Assume `day`, `month`, `year` are declared as `int` members.)

a) It would be best not to write one at all.

b) `Date::~Date() { day = month = year = 0; }`

c) `Date::~Date() { delete day; delete month; delete year; }`

d) `Date::~Date() { std::cout << “Welcome to the end of time.” << std::endl; }`

8. Which of these sets of constructor declarations is illegal?

a)

```
Rectangle();
Rectangle(int width);
Rectangle(int length, int width);
```

b)

```
Rectangle(int width = 1, int length = 1);
Rectangle(std::string name, int width = 1, int length = 1);
```

c)

```
Rectangle(int width = 1);
Rectangle(int perimeter, int width = 37);
```

d)

```
Rectangle(int width, int area);
Rectangle(int width, int length, std::string name);
```

e) None of the above.

9. A private class member function can be called by

a) any other function
b) any other function in the same object
c) only public functions in the same class
d) only private functions in the same class
e) any function in the same class

10. Here is the `SimpleStat` declaration from the book:

```
class SimpleStat {
       private:
              int largest;
              int sum;
              int count;

              bool isNewLargest(int);

       public:
              SimpleStat();
              bool addNumber(int);
              double getAverage();
              int getLargest() { return largest; }
              int getCount() { return count; }
};
```

What's the worst thing that would happen if `isNewLargest()` were declared as `public`?

a) Whoever's using the class would get too much access to class/instance data.
b) Whoever's using the class could be confused about using `isNewLargest()` vs `addNumber()`
c) The implementation of `getLargest()` would no longer work properly.
d) The compiler would complain about "Illegal public function."
e) It makes no difference at all.

11. Reference parameters are so last semester. But *constant* reference parameters are all the rage. What problem do constant reference parameters solve?

a) It can be expensive to copy the objects we're passing to a function.
b) It's risky to have two ways to access a single object.
c) If you want to avoid copying costs, you shouldn't have to risk having your data changed.
d) The compiler can't tell when a function is changing an object's data.
e) I'm tired of people constantly referring to Kanye.

12. So, if constant reference parameters "protect an object when it is passed as an argument," what kind of protection are we talking about? Specifically, suppose Alice has implemented a class called `Hammertime`, and Bob is implementing a function `void stop (const Hammertime& h)`. Which of these correctly (best) describes the situation:

a) Alice is protecting against Bob (or anyone else, for that matter) improperly changing `h`.
b) Alice is protecting against Bob (or anyone else, for that matter), using `Hammertime` in a way she doesn't like.
c) Bob is protecting against bugs in Alice's code messing up his function's behavior.
d) Bob is protecting against his own mistakes changing `h` when he doesn't want to.
e) Bob is protecting against someone using `stop()` to make changes to `h`.

13. In addition to protection with minimum copying, do constant reference parameters offer any other benefits?

a) Nope, not so much.
b) They reduce the chance of memory leaks.
c) They let the compiler know it may be able to optimize the function.
d) They provide a visible guarantee to other programmers that values won't be changed.
e) They're chock full of the vitamins healthy objects need.

14. Suppose we have the class `Polygon` and the class `Point` defined:

```
class Polygon {
   public:
      // several groovy constructors
      Point getCenter();
      double getArea();
      int getNumPoints();
      void shift(double xDelta,
                 double yDelta);
      void rotate(double degrees);
      bool intersects(Polygon p);

   private:
      Point* corners;  // array of points
      int numPoints;
};
```

```
class Point {
  public:
      // several groovy constructors
      double getX();
      double getY();
// move the Point right xDelta and up
yDelta
      void shift(double xDelta,
                 double yDelta);

  private:
      double x;
      double y;
};
```

Write

```
Point(double x, double y);
Point();
Polygon(Point[] arr, int numPoints);
```

Also, write any necessary destructors

Write the `Polygon::shift()` method. This method shifts the `Polygon` `xDelta` units to the right and `yDelta` units up.

15. Consider the `Polygon::intersects()` method. How would this method access the points that make up `p`, the `Polygon` parameter?

a) We need to write a getter method like `Point* getPoints()` for the `Polygon` class.
b) `p.corners[0]` will give us the first `Point` of `p`.
c) It's not necessary; we could just call `p.intersects()` to do the actual computation.
d) It's not possible; the points are declared `private` so they can't be accessed by another object.

16. Remember Tic-Tac-Math?

Define a `struct` that can hold all the values you ask the user for in one turn. Write a `getAPlay()` function that returns one of these `structs`. (Don't worry about validation for the purposes of this question.)

17. Which of these is not a possible type of relationship between two classes?

a) Access
b) Assistance
c) Composition
d) Inheritance
e) Ownership

18 . Testing whether an individual function meets its specification is part of

a) Unit testing
b) Module testing
c) Integration testing
d) System testing
e) Acceptance testing

19.. A radiation therapy machine will incorrectly emit a high-energy beam instead of a low-energy beam if the operator enters a particular (and very unusual) sequence of keystrokes. Two software factors contributing to this behavior are

(1) the code that controls the equipment and the code that processes operator input are not properly synchronized, so experienced operators can make changes too quickly to be processed by the equipment control;

(2) a variable which means "bypass safety checks" if its value is 0 is incremented to make it non-zero —occasionally these increments would overflow to make the value 0, incorrectly bypassing safety checks.

This problem could have been caught with better

a) Unit testing
b) Module testing
c) Integration testing
d) System testing
e) Acceptance testing

20. Suppose I've written the following function:

```
/// return 'E', 'I', or 'S' if the three given lengths represent the sides of
/// an equilateral triangle, an isosceles triangle, or a scalene triangle.
/// Return 'X' if the three lengths do not represent a triangle.

char triangleID(double length1, double length2, double length3);
```

Write a **test set** for this function. Remember that a test set is just a set of test cases; a test case is "the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test."

Hint: Each test case should fit on one line.

21. Now, suppose `triangleID()` is in a module called `triangle`. And suppose your test set from the previous question is in a file called `triangleTest.in`. Write a **main program** that will test `triangleID()`. Assume your executable will be run with the command

```
$ triangleTest < triangleTest.in > triangleTest.out
```

Think especially about what the output of this program should look like—how would it be most helpful to you as the writer of `triangleID()`?

22.

```
public int findLast (int[] array, int arrayLength, int target) {

// Effects: Return the index of the last element in array that equals target.
// If no such element exists, return -1

A     for (int i=arrayLength-1; i > 0; i--) {          B
C           if (array[i] == target) {
D                 return i;
            }
      }
E     return -1;

} // test: array=[2, 3, 5]; target = 2 // Expected = 0
```

(a) Identify the fault.
(b) If possible, identify a test case that does not execute the fault.
(c) If possible, identify a test case that executes the fault, but does not result in an error state.
(d) If possible identify a test case that results in an error, but not a failure. Hint: Don't forget about the program counter.
(e) For the given test case, identify the first error state. Be sure to describe the complete state.
(f) Fix the fault and verify that the given test now produces the expected output.

23. When you're building software, what is the relationship between writing code and testing it?

a) Write some code, get it running "correctly," then apply tests to make sure it's correct.
b) Make a change to some code, then run all tests to make sure the code still passes.
c) There's no essential difference between writing code and writing tests.
d) Develop some tests before you write any code, then write code that passes the tests.
e) These are all viable.

24. The main idea of Test-Driven Development is:

a) Don't write more than a few lines of code before testing your program.
b) Clients present "specifications" for what they want as a set of tests, instead of a long description.
c) The only reason any new code is written is to make a new test pass.
d) Every manager of 5 or more programmers must be a test engineer.
e) Debugging is the most important phase of software development.

25. Why use a test framework like CppUnitLite?

a) If testing isn't automated and easy to do, it won't be done.
b) Frameworks are well-used mechanisms for the tricky language games required by testing.
c) It can run many more different kinds of tests than testing by hand.
d) Without a framework, our tests would be one giant sloppy mess.
e) Both a and b.

26. The best reason to use Test-Driven Development is

a) It provides rapid feedback by encouraging very frequent testing.
b) It dramatically reduces debugging time.
c) It's much easier to test code during development than after development.
d) It encourages a rhythm of alternation between writing new code and polishing existing code.
e) Solving a series of small problems is enjoyable and productive.

27. What is programming by intention?

a) Closing your eyes and wishing very hard that your programming project were done.
b) Pretending that the code you need already exists.
c) Proceeding through a sequence of programming tasks very methodically.
d) Telling your manager that you intend to finish tomorrow.
e) Both c and d

28. You are hired to write a "string calculator," a class with several methods which take various kinds of string parameters and return numeric values. You begin by working on the method

```
class StringCalc {
      public:
            int add(string numbers);
};
```

which takes a string containing 0, 1, or 2 integers (multiple numbers are separated by commas) and returns their sum. For example, `add("27,3")` should return 30. (Assume the string is properly formatted and "valid.")

1) What is the first test you would write? (Use CppUnitLite framework as per reading; no need to write `main()` "boilerplate)
2) Write *just enough* code (implement `StringCalc::add()`) to get this test to pass.

29. When using `gdb`, what's the difference between next and step?

a) There's no difference.
b) Step goes line by line; next may skip over lines that don't do anything.
c) Step goes into functions; next stays in the current function.
d) Next skips to the next function call; step goes strictly to the next line.
e) Step provides more information about program state at each 'step.'

30. When using `gdb`, what's the difference between a breakpoint and a watchpoint?

a) Breakpoints cause the program being debugged to terminate; watchpoints let you choose whether to terminate.
b) Breakpoints are used to detect places where the program is broken; watchpoints just let you observe the program's state.
c) Breakpoints are useful; watchpoints are mostly useless.
d) Breakpoints pause the program at a given line; watchpoints pause the program when a given variable changes.
e) Breakpoints are used to detect memory faults; watchpoints are used for other kinds of faults.

34. To see what functions have been called up to the current point in your program's execution, do a

a) Breakpoint
b) Segmentation fault
c) Core dump
d) Hokey-pokey
e) Backtrace

35. When is it useful to use `gdb` to change the value of a variable in the middle of execution?

a) When you're bored.
b) To try to reproduce a bug by giving a variable an incorrect value.
c) To see if a bug is resolved if a variable gets a correct value.
d) a, b, and c
e) To trigger a watchpoint.

36. I'm writing a class called `Snerp`. I anticipate that many users of this class will need to know how many `Snerp` objects exist at any given time during program execution. What's the best way to provide this behavior?

a) I shouldn't do anything; let the users who need it implement a counter variable.
b) I shouldn't do anything; let the users who need it declare a `friend` (of `Snerp`) function that will access this information
c) I should provide a public static member variable that keeps track of this information.
d) I should provide a private static member variable that keeps track of this information plus a static member function that accesses that variable.
e) I should overload the `new` operator so that new `Snerp` objects know their own "count."

37. When a program is running, when is the most sensible time for static member variables to be initialized (i.e. allocated memory and an initial value)?

a) At around the same time the first object of the class is created.
b) At around the same time `main()` starts running.
c) Exactly when the class's static constructor is invoked.
d) Not until the variable is actually used in code (e.g. passed as a parameter or used in an assignment): if it's never used, no need to initialize it.

38. The book uses the `Length` class in its discussion starting on page 712. The declaration

```
friend Length operator+(Length a, Length b);
```

is added to the `Length` class. What would happen if we also added the declarations below (as well as definitions)?

```
friend double operator+(double a, Length b);
friend double operator+(Length a, double b);
```

a) The compiler will complain that `operator+` has the wrong number of parameters.
b) The compiler will complain that `operator+` has already been defined for the `Length` class.
c) The compiler will issue a warning about "too many `friends`."
d) When the program runs, the wrong operation may be performed if we try to add a `Length` (as first operand) and another value.
e) Nothing bad will happen; this is perfectly fine.

39. Remember our old friend, `Polygon`? It has two private instance members:

```
int numPoints
Point* corners;
```

and the `Point` class also has two private instance members:

```
double x;
double y;
```

Write all necessary copy constructors and `operator=` methods.

40. Speaking of `Point`, which of these operators would be the best addition to the `Point` class? (Be prepared to describe what the behavior of your chosen function should be.)

a) `friend bool operator==(Point a, Point b)`
b) `double operator[](int x)`
c) `friend bool operator<(Point a, Point b)`
d) `Point operator++()`
e) `friend Point operator!(Point a)`

41. A type conversion operator

a) Has no declared return type
b) Takes no explicit parameters
c) Can only convert an object to a built-in type
d) Both a and b
e) All of a, b, and c

42. Which critique of the `Length` class on pages 729–730 is most serious?

a) A file called `Length2.h` shouldn't define a preprocessor constant `LENGTH1_H`.
b) The `operator double()` calculation will produce incorrect values in some cases.
c) It's dangerous for the `double()` and `int()` conversions to produce values with different units.
d) The class is missing a copy constructor.
e) None of the above; the class is fine as written.

43. How can we recognize a convert constructor?

a) Its signature has the keyword `convert` at the end.
b) Its parameters all have built-in types.
c) It has exactly one parameter.
d) We can't; the compiler decides which constructors to use for conversion.
e) We can't; they are only identified at runtime.

44. Suppose the `Length` class has a convert constructor for `int` values (which it does!). In which circumstance would this constructor ***not*** be used?

a)
```
int sum = addLengths(pencil, yardstick); // addLengths returns Length object
```

b)
```
pencil = 8;  // pencil previously declared as a Length object
```

c)
```
Length yardstick = 36;
```

d)
```
Length foo(double x) { return x; }
```

e)
```
testLength(17);  // testLength expects a Length parameter
```

45. Now suppose we have both a `Length` and a `Height` class. We decide we should be able to convert between them, so we write methods

```
Length::operator Height()
Length::Length(Height h)

Height::Height(Length l);
```

And then in our main program, we write

```
Length l(10);
Height h = l;
```

What happens?
(Hint: you will not find the answer in the book. Think about what you know about C++.)

a) The program runs normally; hooray!
b) The program runs, but `h` gets a weird value.
c) The compiler complains about our main program.
d) The compiler complains about the `Length` class.
e) The compiler complains about the `Height` class.

46. You are writing a `NewDate` class for your company. It has lots of nifty new features. But your company also has a bunch of existing code that works with the old `Date` objects, and your technical manager doesn't want to have to change that existing code. What methods should you write in `NewDate` to keep management as happy as possible? Just write the signatures.

(For class discussion: what are the limitations of your solution?)

47. Which of these is an example of aggregation that is *not* composition?

a) A `Person` object `p` contains a `Date` object representing the date of birth of `p`.
b) A `Person` object `p` contains pointers to `Person` objects representing the parents of `p`.
c) A `DynamicList` object contains an array (of `Length` objects) dynamically allocated/de-allocated by constructors and destructors.
d) Both a and b.
e) Both b and c.

48. For each class X below, come up with two classes Y and Z such that

Y HAS-A X
and
Z IS-A X

For example, if X is `Dog`, then

`AnimalShelter` HAS-A `Dog`
`Poodle` IS-A `Dog`

1. Animal
2. Book
3. Polygon
4. File (i.e. a disk file)
5. Hamburger

49. Compared to its base class, a derived class *may* have

a) More data members
b) More member functions
c) Different member functions
d) Fewer member functions
e) All of the above.

50. Following the example on pages 743–744, write a class definition for `Artist`, derived from `Person`, which includes data members representing the artist's agent, primary genre, and country of origin. Don't worry about including header files for other classes. Don't write method definitions.

51. Members which are declared `protected` are

a) Accessible to methods in the class and any derived classes, and classes derived from those, etc.
b) Also protected members of any derived class.
c) Also private members of any derived class.
d) All of the above.
e) None of the above.

52. When constructing of an instance of a derived class, the constructor of the base class is called

a) If/when the programmer indicates it should be called.
b) When the compiler decides it would be optimal.
c) Never.
d) First.
e) Last.

53. The syntax for invoking a particular base constructor is a lot like the syntax for:

a) Member initialization lists, except that class names instead of member names are used.
b) Overloading constructors, except that the name of the base class is used.
c) Array initialization, except that the values inside the `{ }` don't need to be of the same type.
d) The scope operator, except that only one colon is used.
e) Nothing ever before imagined in this world.

54. What will the following program display?

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

class Base {
      public:
            Base() { cout << "Entering the base.\n"; }
            Base(char *str) { cout << "This base is " << str  << ".\n"; }
            ~Base() { cout << "Leaving the base.\n"; }
 };

 class Camp : public Base {
      public:
            Camp() : Base() { cout << "Entering the camp.\n"; }
            Camp(char *str1, char *str2) : Base(str1)
                  { cout << "The camp is " << str2 << ".\n"; }
            ~Camp() { cout << "Leaving the camp.\n"; }
};

int main() {
      Camp outpost("secure", "secluded");
}
```

a)
```
Entering the base.
Entering the camp.
Leaving the camp.
Leaving the base.
```

b)
```
Entering the camp.
Entering the base.
Leaving the base.
Leaving the camp.
```

c)
```
Entering the base.
This base is secure.
Entering the camp.
The camp is secluded.
Leaving the camp.
Leaving the base.
```

d)
```
This base is secure.
The camp is secluded.
Leaving the camp.
Leaving the base.
```

e)
```
The camp is secluded.
This base is secure.
Leaving the camp.
Leaving the base.
```

55. Pick the true one:

a) A member function can be overridden, or overloaded, but not both.
b) Only a base class can contain overloaded methods.
c) A constructor cannot be overridden.
d) Overridden methods change the base class.
e) Overloaded methods are not inherited by derived classes.

56. Consider the class `Hamburger`:

```
class Hamburger {
      protected:
            int calories;

      public:
            Hamburger() { calories = 800; }
            int getCalories() { return calories; }
};
```

Now write a derived class, `Cheeseburger`. The `Cheeseburger` constructor should take a single `int` value, representing how many slices of cheese are on the cheeseburger. The overridden `getCalories()` method should return the total number of calories in the cheeseburger, assuming 80 calories per slice of cheese.