CSC 122 Sections 1 & 2
Time: 75 minutes **Calculator: OK**
Notes: NONE

Directions: Read the problems carefully. *Circle* **all** correct answers to multiple choice questions. Explain "why" if you *circle* FALSE. Fill in **all** blanks (unless you decide a T/F is FALSE). *Circle* the answer you feel most appropriately fills the blank if choices are given. Answer to the best of your ability.

This is a take-home test. You may ask **ME** questions for clarification — *not* each other. Feel free to attach pages to the end of the test if you need to continue a long answer. Please keep in mind that questions that ask you to write code are intended to be done without the aid of the compiler.[1] Take your time, but don't lose track of how much time is left.

Good luck! Have fun!

1) Name at least two benefits that the `class` mechanism give us over using `vector`s with respect to data storage.
   A) `class`es can hold values of different types; `vector`s are limited to holding values of a single base type.
   B) `class`es protect their data in `private` areas; `vector`s allow anyone to walk up and overwrite data with a simple `[]` = combination.
   C) `class`es make us write ten or twenty functions when four would do. `vector`s never made anyone write a function.
   D) `class`es collect small pieces together to more naturally represent real-world data; `vector`s would require they all be the same type or would need to be in parallel.
   E) `class`es couple the data closely with the operations that act on them; `vector`s would have the operations as extra functions polluting our global `namespace`.

2) Given that a `class` *`Rational`* exists and given the `vector`:

```
vector<Rational> values;
```

And given that this `class` has a method to add two *`Rational`* objects:

```
Rational Rational::add(const Rational & b) const;
```

And a [pair of] method(s) to divide a *`Rational`* object by an integer:

```
Rational Rational::divide(long n) const;
Rational Rational::divide(unsigned long n) const;
```

Show code to calculate the average of all the elements in the `vector` *`values`*. (Hint: what is the average of two – or more – rational numbers?)

```
Rational sum(0), avg;

for (vector<Rational>::size_type r = 0; r != values.size(); r++)
{
    sum = sum.add(values[r]);
}

avg = sum.divide(values.size());  // may need to typecast if size_type's are 16-bit
```

3) Before you can open a connection to a file stream, you must do three things:
   A) declare a `vector` of objects to store the file contents.
   B) get from the user the name of their file.
   C) `#include` the library `fstream`.
   D) declare either an `ifstream` or an `ofstream` variable to connect to the file with.
   E) declare a variable to count how many lines we've processed in the file.

[1] I can't keep you from using it, but try to do it on your own first. . . please?! *big sad puppy dog eyes*

4) Once you are ready, you'll need to do two things to make a connection to a user's file:

   A) `clear` the stream variable to make sure it is ready to use.

   B) convert the `string` filename variable to a `c_str` as you pass it to the file connecting method.

   C) pass your `string` filename to the file connecting method as is.

   D) use the `connect` method with respect to your stream variable.

   E) use the open method with respect to your stream variable.

5) Given the following sort function:

```
// sort between start and end indices (inclusive)
void good_bubble(vector<char> & vec,
                 vector<char>::size_type start,
                 vector<char>::size_type end)
{
    vector<char>::size_type loop = start, cur;
    bool done = false;
    while (loop != end && !done)
    {
        done = true;
        for (cur = start; cur != end; ++cur)
        {
            if (vec[cur] < vec[cur+1])
            {
                swap(vec[cur], vec[cur+1]);
                done = false;
            }
        }
        ++loop;
    }
    return;
}
```

, alter it to sort a `vector` of *`Rational`* objects. (Assume that you are given a method named *`less`* in the *`Rational`* `class` which compares its caller and single argument, returning `true` if the caller is less than the argument and `false` otherwise. Its declaration looks like this:

```
bool Rational::less(const Rational & r) const;
```

.) (Please rewrite only the lines that must change. This will make it easier.)

```
void good_bubble(vector<Rational> & vec,
                 vector<Rational>::size_type start,
                 vector<Rational>::size_type end)
. . .
    vector<Rational>::size_type loop = start, cur;
. . .
            if (vec[cur].less(vec[cur+1]))
```

Don't forget to code a *`swap()`* function to go along with your version of the *`good_bubble()`* sorting function:

```
void swap(Rational & a, Rational & b)
{
    Rational c;
    c = a;
    a = b;
    b = c;
    return;
}
```

6) When opening a file connection, make sure the stream connected successfully to the file by:

   A) `clear` the filename `string` inside the checking loop.
   B) checking for `fail` or `!good` or even just `!`.
   C) use ≫ instead of `getline` for the filename the second [and later] time[s].
   D) `clear`ing the stream variable after the checking loop is done.
   E) using `close` and `clear` inside the checking loop.

7) A `class` named *Avg* contains a data member named *Nums* which is a `vector` of `double` values. Show definitions for accessor and mutator methods for this data member. (Hint: Would you need to retrieve ALL of the vector values at once or just one at a time? Thought-provoker: How do you append new values to the *Nums* `vector`? Thought-provoker: Does the programmer using the *Avg* `class` need to know how many numbers are stored in *Nums*?)

```
typedef vector<double>::size_type vcD_sz;

bool Avg::set_num(vcD_sz which, double num)
{
    bool worked = false;
    if (which < Nums.size())
    {
        Nums[which] = num;
        worked = true;
    }
    else if (which == Nums.size())
    {
        worked = add_new_num(num);
    }
    return worked;
}
```

```
bool Avg::add_new_num(double num)
{
    // should we check Nums.max_size()?
    Nums.push_back(num);
    return true;
}

double Avg::get_num(vcD_sz which) const
{
    return (which < Nums.size()
           ? Nums[which] : -42.0);
}

vcD_sz Avg::get_num_nums(void) const
{
    return Nums.size();
}
```

8) What is the type of `getline`'s first argument that it can be used to read from either `cin` or an `ifstream` variable?

   `istream &` is compatible with both the console stream `cin` and any `ifstream` variable.

   What type should you use to allow a function to 'display' to either `cout` or an `ofstream` variable?

   `ostream &` is compatible with both the console stream `cout` and any `ofstream` variable.

9) Of the following prototypes, choose the *best* prototype for a method to input an object of the `class` *Rational* from the user. (Recall that a rational number — or common fraction — has a numerator and denominator: $1/2$, -$4/5$, $7/3$, etc.)

   But **not** $2\frac{1}{3}$ nor -0.80.

   A) `Rational Rational::input(void) const;`
   B) `Rational Rational::input(void);`
   C) `bool Rational::input(void);`
   D) `bool Rational::input(long & numer, long & denom) const;`
   E) `bool Rational::input(Rational & rat) const;`

   Now show an example of a call to the method whose prototype you chose above. (Include any necessary declarations or other statements for it to work.)

```
Rational a;
cout << "Some kind of prompt:  ";
while ( ! a.input() )
{
    // clear, ignore, error message, re-prompt...
}
```

10) What functions/operations can you use to get data from an `ifstream` variable?

≫, `getline`, `peek`, etc.

What functions/operations can you use to put data into an `ofstream` variable?

≪, `setw`, `flush`, etc.

11) A `class`' copy constructor is automatically called in three (3) situations. What/When are they?
   A) when one object is declared and *at the same time* initialized with another object that already exists
   B) when an object is declared and *at the same time* initialized with values chosen by the `main` programmer
   C) when a function accepts an argument using the value mechanism, the formal argument is copy constructed from the actual argument
   D) when a function accepts an argument using the reference mechanism, the actual argument is copy constructed from the formal argument
   E) when a function `return`s its result using the value mechanism, the result given back to the caller is copy constructed from the value of the `return` expression

12) What two new keywords are used to specify where in the program access to the members of a `class` object is available? Explain who (i.e. what parts of the program) can access members of each access type.

Oh. . . access specifier. . .

`private`	only the members (methods; data doesn't access other data) of the `class` itself can access `private` members (data or methods) of a `class`
`public`	any part of the program can access `public` members (methods, typically; data would be silly) of a `class`

13) [✓ TRUE] / FALSE All methods of a `class` can access the `private` data of the `class`.

TRUE / [✓ FALSE] Data which is `public` is generally considered normal when defining a `class`.

[✓ TRUE] / FALSE `private` methods, although rare, can often be useful for tasks the `class` must (or should) manage on its own.

14) Given that a `class` exists named *`Complex`* which has a method to add two *`Complex`* objects, we might call it like this (note the flexibility the caller has):

```
Complex a, b, c;
// fill in a and b
// with values somehow...
c = a.add(b);
```

or even. . .

```
Complex a, b;
// fill in a and b
// with values somehow...
a.add(b).output();
```

Show the definition of a method to add two *`Complex`* objects. (Recall that a complex number is of the form $a \pm b\imath$ where $\imath$ represents $\sqrt{-1}$ — the square root of -1; an imaginary value — hence the funny *i*.) (Hints: Does addition change its operands? What is the result of adding two complex numbers? How many arguments are required to add two complex numbers in a `class` method?) (Thought-provoker: Some might desire to add a *`Complex`* number to a real. . . er. . . `double` value. Can you overload the *add* method for this situation?)

```
// adds the Complex object 'other' to the calling object
// to produce the result which is returned to the caller
const Complex Complex::add(const Complex & other) const
{
    return Complex(real + other.real, imag + other.imag);
}

// adds the 'real' other value to the calling object
// to produce the result which is returned to the caller
const Complex Complex::add(double other) const
{
    return Complex(real + other, imag);
}
```

15) Any/All `classes` should include at least the following methods:
  A) constructors (at *least* default and copy)
  B) constructs (golems, robots, etc.)
  C) accessors & mutators
  D) Mack & Cheesy
  E) input & output

16) What is the member access operator in C++?
The period or dot (`.`) is used to access members of a `class` in C++.

17) What does the term 'calling object' mean, anyway?
The calling object is the object on the left side of the dot (`.`) operator in the call to a `class` method. I.E. *a* is the calling object in the *`Complex`* addition problem (#14).

18) In relation to other parts of a program, where would a `class` *definition* be placed?
Either above the `main` (typically after the `using` directive) or in the interface (`.h`) file of a library.

What about the definitions of [non-`inline`] `class` methods?
Either after the `main` or in the implementation (`.C` or `.cpp`) file of a library.

19) [✓ TRUE] / FALSE In a method to add two *`Rational`* numbers, the calling object acts as the left-hand addend (value to be added). (See the explanation of rational numbers in #9.)

i.e.
$left + right \Rightarrow$
`left.add(right)`

TRUE / [✓ FALSE] The single *`Rational`* argument acts as the left-hand addend.

TRUE / [✓ FALSE] The result of the addition is stored in a new *`Rational`* object — but we still may possibly change either the left- or right- hand objects — or both!

[✓ TRUE] / FALSE This new object is returned from the add function by the return value mechanism.

20) What is required to make a `class` argument 'pass-by-value'?
Nothing — just pass the formal argument with the `class` type and a name and pass the actual argument like normal.
What is required to make a `class` argument 'pass-by-reference'?
As with the built-in types (or `string` — which is a `class`, after all), place an ampersand (&) between the type of the formal argument and its name. The actual argument is passed like normal.

21) Explain (briefly but correctly) the purpose of accessor and mutator methods in a `class`.

accessor	allows caller to view a copy of the value in a `private` data member of the `class` (without direct access, they cannot accidentally change the member's value!)
mutator	allows the caller to perform *controlled* changes to the specified `private` data member of the `class`; contains any error checking and/or validity checking codes necessary to ensure that the `class`' data members are not butchered, garbled, or otherwise messed up/destroyed

22) Why don't `class` input methods typically display a prompt for the user?
They are trying to be as generic/general/re-usable as possible. With a literal prompt, they'd be in the way of interface customization or be too specific to a particular application. With an argument prompt. . . well, Jason just said it would be 'untrue to the zen of `class` programming'. . . or something like that. And with a prompt member (& proper accessor/mutator), we'd be managing data outside of our `class`' 'true nature'. . . whatever that means.
Shouldn't the program always prompt before input?
Yes, but that is the program — not our `class`! We want to be re-used by program after program after program. . . "Don't tie me down!"

Why don't `class` output methods typically print labels and/or spacing?
Similar to the input methods part and prompting — it isn't generic! If I want my time writing this `class` to be rewarded by lots of people downloading it and using it, I've gotta keep myself. . . er. . . it as general and re-usable as I can!
Shouldn't the program always label outputs?
Well, the program should, certainly. But I'm a simple data type and I don't know what kind of program I'm in. I'm not going to presume what kind of labels or even spacing might be appropriate around my data! I'll just print the raw essence of my being to the display and hope the program has set up appropriate labeling and spacing either before or after me if not both. . .

23) Name and describe the three types of constructors briefly.

default	takes no input, fills in data members with sensible values (or purposely erroneous values) when programmer didn't supply any
copy	takes a [constant] reference to the class type as input, fills in data members of the object being constructed with copies of those from the object we have a reference to (via the formal argument)
initialization parameterized ???	takes formal arguments appropriate to initialize some/all data members of the class as input, there may be many versions of this type of constructor — each would take a different number or type(s) of arguments to overload the others

24) Which of the following are benefits that classes give us over the built-in data types?

A) improved initialization of new variables/objects via constructors (most notably the default constructor — built-in types are simply left with garbage bits by default!)

B) data are closely coupled with the operations that work with them

C) allow us to more naturally represent real-world data by collecting together all the little pieces in one place rather than scattered variables each with a part of the whole

D) collected data can be of multiple different types — a built-in typed variable is just the one type (even a string can only collect many values of the type character together!)

E) primitive data security via the private/public keywords and the accessor/mutator methods we provide

25) Other than those you chose above (#15), what methods might/should a *Complex* class include?

A) `Complex add(const Complex & c) const;`

B) `Complex add(double x) const;`

C) `Complex reciprocal(void) const;`

D) `Complex conjugate(void) const;`

E) `double magnitude(void) const;`

26) What is the only real difference between an ADT and a class?

An ADT is a general description that could be used by a programmer to implement it in any programming language. classes are the C++ mechanism for implementing ADTs.

What does ADT stand for, anyway?

Abstract **D**ata **T**ype

27) How is overloading involved with constructors?

Since all constructors have to be named after the class they construct, we must overload them to allow the programmer using our class to create their objects in useful ways: by default, with specific values right away, as a copy of another object, etc.

28) Code the output method for a *Rational* class. (See the explanation in #9 about what a rational number is.) (Hint: Will this method label or space anything? Does it print *anything* but the numerator and denominator?)

```
void Rational::output(void) const
{
    cout << numer << '/' << denom;
    return;
}
```

29) What are the actual differences in structure and class behavior in C++?

A struct has a default of public access for all of its members whereas a class has a default of private access for all of its members.

looks around What? Why are you still here? That's the only difference in the language's eyes! I swear!

Oh, alright! Programmers *tend* to not use many of the object-oriented features of C++ when designing with a struct! They all work, but we just don't use them. . . it's the old C programmer in us crying out in irrational pain. . . *whimper* *sniff* Are you happy now?!

(And it helps when you are moving back and forth between multiple languages on a single project. . . or even multiple projects at once!)

30) Show how to rewrite the following method definition as a single line. (See the explanation in #9 about what a rational number is.)

```
Rational Rational::divide(const Rational & r) const
{
   Rational t;
   t = r.reciprocal();
   t = multiply(t);
   return t;
}
```

```
Rational Rational::divide(const Rational & r) const
{
   return multiply(r.reciprocal());
}
```

31) Code a method to compare two *Rational* objects and tell if the caller is less than the other. (See the explanation in #9 about what a rational number is.) (Hint: You'll want to be as ~~accurate~~ exact as possible.) (Thought-provoker: How long is your method? Is there something you should do to it, then?) (Thought-provoker 2: Will your method change either object it is comparing? Is there something you should do to them, then?)

Either:

```
bool Rational::less(const Rational & other) const
{
    return numer*other.denom < denom*other.numer;
}
```

Or, to `inline` it:

```
class Rational
{
// ...
public:
    bool less(const Rational & other) const
    {
        return numer*other.denom < denom*other.numer;
    }
};
```

32) Is the `inline` keyword needed to `inline` a `class` method? Why/Why not?

NO!

The compiler feels we must be attempting to `inline` the method if we simply define it inside the `class` definition.

33) Why does the mutator of a *Rational* `class` work with both data members simultaneously whereas a *Complex* `class` has separate mutators for its two [numeric] data members? Don't we normally follow that latter tactic? What's up with the *Rational* `class` mutator?!

It is actually not a question of C++ or the mutator itself, but of mathematics. Because of the intimate mathematical relationship between the numerator and denominator of a rational number — the two must be changed in lock-step. Most `class`' [we've seen] don't have data values which are so tied together. For a *Complex* `class` or a *Point2D* `class`, the data members are on separate axes/dimensions, for instance.

34) What *is* a member initialization list with respect to constructors?

When defining the constructor (either as an `inline` method or out-of-line), place a single colon (`:`) after the constructor's argument list followed by a comma separated list of the `class`' data members. After each member's name, place a pair of parentheses. In the parentheses for each member, place an expression with which to initialize the member. After the list is over, you can open the constructor's body. (If the body is empty because all set-up tasks have been completed in the member initialization list, then just close it right away: `{}`. Otherwise, complete set-up tasks before closing the body!)

Give an example of using a member initialization list for a *Rectangle* `class`. (Hint: A rectangle can be defined by its upper right and lower left corner coordinates — two *Point*s.)

```
Rectangle::Rectangle(const Point & low_left, const Point & up_right)
    : lower_left_corner(low_left), upper_right_corner(up_right)
{
}
```

Initializer lists aim to make `class` object construction more efficient. How?

Normally the compiler tries to fill in all `class` data members with their default or 'zero' value *before* the constructor's body executes. However, if an initializer list is present, the compiler will not do this default filling but rather simply perform the initializations the programmer has requested instead. So on many constructors we can place the values we'd normally want (either reasonable defaults or explicitly erroneous values we can detect — the same ones our default constructor would use) before we call the mutator methods inside the body to do any necessary error/validity checking. (And on a default constructor we're done. And on a copy constructor we skip these simple values and simply copy the other object's data values into our members in the list.)

35) In the following code fragment:

i) explain what object(s) is/are being constructed
ii) from what it/they is/are being constructed
iii) which constructor(s) will be used to perform the construction(s)

(See the explanation in #9 about what a rational number is.)

```
void f(Rational a);                 // a by copy from x (below)
void h(Rational & a);               // nothing -- pass by reference

inline
Rational g(void)
{
    Rational t;                     // t by default from nothing
    return t;                       // result of g by copy from t
}

// in some function...maybe main?
{
    Rational x;                     // x by default from nothing
    Rational y(4, 3);               // y by init/param from 4 & 3
    Rational z = y;                 // z by copy from y
    Rational w(x);                  // w by copy from x
    Rational r(-2);                 // r by init/param from -2 (& 1)

    f(x);                           // a is copied from here
    x = g();                        // result of g is called for here,
                                    // but nothing is ctor'd ˜here˜
}
```

In summary:

i) being constructed	**ii) constructed from**	**iii) constructor used**
a	*x*	copy
nothing	N/A	N/A
t	nothing	default
x	nothing	default
y	4 & 3	initialization/parameterized
z	*y*	copy
w	*x*	copy
r	-2 (& 1)	initialization/parameterized
result of *g*	*t*	copy
nothing	N/A	N/A

36) Could a `class` have the following two methods in it and still compile/run fine?

```
void f(void);
void f(void) const;
```

How would the compiler know which one to call?!

Yes, it would work. The compiler can determine which one to call based on whether the calling object needs to remain `constant` or not. i.e. The `constant` version is called when the calling object is a `constant` object, and the non-`const` version is called otherwise.

37) With respect to the *Rational* `class` discussed in #9 — and in particular the *input* method you designed the API for and put to use there — show how you would implement this method. (Thought-provoker: Is there any error checking to be done for a *Rational* number or its input format? If so, show how you should probably implement this, too. . .)

```
// reads a numerator, separator, and denominator from the user
// uses the mutator for proper error checks (we trust the user
// even less than the other programmers!  *grin*)
bool Rational::input(void)
{
    char t;
    long n, d;
    bool okay;
    cin >> n >> t >> d;
    okay = !cin.fail() && t == '/' && set(n, d);
    if (!okay)
    {
        cin.clear(ios_base::failbit);
    }
    return okay;
}

// we must mutate both members at once because changing one will
// affect the other -- mathematically!
bool Rational::set(long n, long d)
{
    bool okay = true;
    long g = gcd(n,d);  // non-class function!
    numer = n/g;
    denom = d/g;
    if (denom < 0)  // nut-jobs!
    {
        denom = -denom;
        numer = -numer;
    }
    else if (denom == 0)  // el-stupido!
    {
        denom = 1;    // ignore the truth
        okay = false; // but report it...
    }
    return okay;
}
```