Review of Assignments 1 and 2
My Solutions

• Was previously providing solution source code and testing drivers
• Was forced to remove it this week
• Will no longer provide solutions, but will still upload testing drivers
Compile Errors

- A few students had problems with compilation errors
  - Most common mistake was name mismatch
    - e.g. compare vs Compare
  - Some also used C++11 features which is disallowed
    - e.g. static string CelsiusName = “Celsius”;
    - Can’t init non-const members in header
    - Can’t init objects in header
- I take off 5 points for error that I have to fix
  - Start with the first error and change one line to make it go away
  - Can potentially have cascading effects (e.g. mismatch in header leads to mismatch in implementation)
- This is to get you ready for later classes
  - Generally everything is packaged together
    - e.g. in operating systems, you submit the full application instead of just a class
  - But compile errors result in an automatic 0
Common Mistakes

- **Valid member data**
  - The object needs to always be in a valid state
  - Data protection is one of the main features of classes in object oriented programming
  - Whenever you pass data to a class (input, parameters, etc), it needs to check everything and handle invalid data
  - You can use asserts to make sure your class never reaches an invalid state
    - `#include <cassert>`
    - `assert(length => 1 && length <= 39)`
      - Will crash program if it doesn't pass
    - Can make a private void function that tests each of the member variables, then can call this at the end of each of functions in your class
      - If it crashes, you'll be able to track down which function introduced the invalid state
      - Can make tracking things down easier using the macros `__LINE__` and `__PRETTY_FUNCTION__`
Common Mistakes

• Floating-point Precision
  – Floating point numbers are encoded as
    • sign * mantissa * 10^{exponent}
    • e.g. -2500.0 → -1 * 2.5 * 10^{3}
  – This means certain computations will lead to rounding errors
    • Mantissas overflow
      – 52 bits for 64 bit doubles
      – e.g. 1.0 / 3.0
    • Exponents overflow
      – 11 bits for 64 bit doubles
      – e.g. 10^{300} * 10^{300}
  – When working with doubles, you want to
    • Limit number of unnecessary computations
      – Can come across one that leads to overflow (e.g. division by 3)
    • Perform equality checks with a degree of error built in
      – e.g. d1 == d2 → abs(d1 − d2) < 0.0001
  – For more accurate computations, you'll need to use integers
    • Only one type of overflow
    • But harder to represent decimal numbers
Common Mistakes

- **Code Dependencies**
  - Code reuse is highly encouraged as it saves time and avoids common errors
  - However, using non-tested code can lead to an increase in errors
    - I can only test the interface of the class
    - e.g. if you use Set in your constructor and Set doesn't work, it will lead to them both being incorrect
  - Make sure to spend more time testing reused code
    - If you run out of time and believe that there might be a problem, it might be a good idea to break things up....
    - However this will make it harder for to give back partial credit as I won't be able to see the call dependencies
  - Usually, the specification lists things in the best order to implement them...
Common Mistakes

- Crashing
  - Incorrect code is better than code that crashes... at least to a degree
  - If I encounter a crash, it's a flat 10 point deduction
    - Doesn't accumulate with additional crashes
  - Common causes for crashing
    - Invalid memory reference
      - e.g. writing to a NULL pointer
      - Will start running into this in either assignment 4 or 5
    - Infinite loops generating output
      - Crashes because it runs out of memory / disk space
      - This mainly happens in files, but since I redirect I/O when testing, it can happen to your cout calls
    - Creating too many processes / threads
      - Crashes due to memory consumption / hard limits
      - You won't run into it in this class, but you will see it in the future...
Common Mistakes

- **Code Quality**
  - **Structure**
    - Indentation needs to be consistent
    - Code needs to be readable (i.e. not a wall of text)
  - **Naming**
    - Use appropriate variable names for the scope it applies to
      - temp or i is fine for local variables, but not member data
      - Zebra can be a fine name for an animal based class, but is not too good when working with fractions
      - ThisIsAVariableStoringABitmapOfFileLocations is a little long for any situation
    - Need to avoid global variables
      - Can use constant member/class variables instead
    - Need to apply const to appropriate member functions
      - e.g. Compare needs to be const, Set doesn't
      - I haven't taken off points yet, but I will in the future
Testing

• Double check the specification to make sure everything matches
  – I'll generally provide the function prototypes unless it's something obvious (e.g. constructor, operator overloading)

• Make sure to test your code with the provided driver
  – Add your own calls to it to make sure everything gets tested
Testing

- Build small examples to verify your code, this tests two constructors, post increment, equality checking, and the insertion operator
  - Distance d1;
  - d1++;
  - Distance d2(1);
  - cout << d1 == d2;
- Iteratively build on these to test larger ranges of inputs
  - Distance d1;
  - int amt = 1000000;
  - Distance d2(amt);
  - for (int i = 0; i < amt; i++)
    - d1++;
  - cout << d1 == d2;
Testing

- Input and output can be automated
  - For input, you save time by not having to type everything each time you test
  - For output, you can automatically compare your result to a desired value to more quickly find what fails
- Two main ways of doing this
  - With I/O redirection
    - You can pass in a file of inputs and write to an output file
    - Then you can compare your output file with a file of desired output
    - e.g. ./test < inputs > out; diff out correct
  - With stringstream...
Testing

• Input and output can be automated with stringstream
  – Stringstream is a class that lets you build an io stream and then treat it as a string
  – Useful for smaller scale testing as you don’t need a file for each input/output
  – Example:
    • #include <sstream>
    • streambuf *oldCin = cin.rdbuf(); //save cin's buffer
    • ostringstream builder; //output stream to write to cin with
    • istringstream newCin; //new cin buffer
    • cin.rdbuf(newCin.rdbuf());
    • double d1 = 7;
    • char s1 = 'f';
    • Temperature t;
    • builder << d1 << ' ' << s1 << "\n"; //write to the output stream
    • newCin.str(builder.str()); //set cin's buffer to be the output stream
    • t.Input();
    • cout << t.GetDegrees() << endl;
    • cout << t.GetScale() << endl;
    • cin.rdbuf(oldCin); //restore cin's buffer
Readme

• If you run out of time and can't test / implement everything
  – You can document known problems with your code
    • Just make sure I can easily find it
  – If it's something I wouldn't have found with my tests, then I'm not going to take off points for it
  – But if I would have found it anyway, documenting this can potentially save you points
    • e.g. Constructor calls Set, but Set has a problem, I won't mark off the constructor as much
  – This shows that you know what problems exist in your code, and it saves me time trying to figure out what the issue is
  – This is entirely optional
    • But the list of errors also found in the Readme is not