As programming projects become more complex, we need techniques for making projects manageable. In general, the probability that we can write all the code of the project at once and expect it to just work is very low. And when things do not work, it becomes increasingly difficult to isolate the problem, and in fact, it may be the case that the error is in the interaction of two or more parts of the project that seem to work fine individually. This handout is a work-in-progress intended to give the reader some basic tips on managing the complexity of a large programming project. It is neither exemplary nor exhaustive, but if you follow the tips, it should make managing any project easier.

0. Serious analysis and design up-front reduces implementation time
This may seem obvious, but it is the most important tip. Resist the urge to write any code until you have thought about what you are trying to do and how to get there. The more you know where you are going, the less time it will take to get there. The goal of this step is to break down the problem into small, well-defined pieces.

An exception to this guideline is sometimes you do want to write some prototype code just to see how something works. That's fine as long as you expect to “discard” the actual code and do not try to evolve it into your entire project without serious thought as to how it fits in.

1. Implement the pieces of your design a few at a time
This also may seem obvious. Resist the urge to write all the code at once, and then try to debug it. Solve the pieces a few at a time, making sure that they work by writing one or more driver programs that test the pieces individually. Start with the foundation pieces, then move on to the more refined pieces. Always test the interactions between what you have already written and the new piece by rerunning your previous tests. For example, when writing a new class, start with the constructors, so that you can build objects, then the accessors and output operations that let you see what you have built. Write a driver program that uses just these operations. If the project requires more than one class, start with the simpler one.

Sometimes it is advantageous to write a skeleton of a function without any of the actual implementation. That is, you write the prototype and header of the function, but only a minimal body, usually just a message saying the function isn't implemented, yet, and/or returning some default value. These skeleton functions are called stubs and should be used whenever you know you want a particular function, but want to defer actually implementing it until a later time. This allows you to compile and run a program that uses the function, though of course, the results will not be valid, allowing you to see if the other parts of your program work.

2. Debugging statements should contain real information
A debugging message such as “Entering operator=” or “operator+: rhs operand is empty” is much more informative than messages like “Got here” or “Here 2”. If you are going to write debugging statements, they might as well say something useful.

One way to manage these statements is to use debug flags and a PrintDebug function:

```cpp
const bool DBG_PHASE1 = false;
```
const bool DBG_PHASE2 = true;

void PrintDebug (bool debug, string message)
{
    if (debug)
        cerr << message << endl;
} // end PrintDebug

The PrintDebug function just prints a message if debugging is “on” (represented by the true value) during the call. The flags are used to turn on and off groups of debugging messages. (Otherwise, you would have to find each one and change the call argument from true to false or vice versa.) For example, a default constructor body might have:

    PrintDebug (DBG_PHASE1, "Entering default constructor");
    // Statements to create default object
    PrintDebug (DBG_PHASE1, "Exiting default constructor");

Another tip is to “indent” the debugging messages if they are inside nested control structures. For example,

    PrintDebug (DBG_PHASE2, "Entering operator+");
    if (rightPolyPtr == 0)
        {
            PrintDebug (DBG_PHASE2, " operator+: rhs operand is done");
        }

Now when the program runs, the debugging statements line up as:

    Entering operator+
        operator+: rhs operand is done

and you can see the control structure in the output.

When you are finished with your project, you can set all of the debugging flags to false. A good optimizing compiler should be able to detect that the conditions will be false and that the function will not have any other code executed and optimize away the function calls and perhaps the entire function itself. Later, if there is a newly-discovered problem, or you want to make an enhancement, you can turn the debugging messages back on easily.

3. Start with the simplest test cases: expected and boundaries
   This also may seem obvious, but resist the urge to use a complex test case the first time you run your program. Instead start with the simplest expected test case that will demonstrate the use of the piece you are working on. Once that test passes, go onto to the next simplest test case, etc.

   Generally, the number of interesting test cases can be reduced to the phrase “expected, 0/empty, 1, n/full.” For example, this means that one should test for the case of expected input, no input, and one input. In some cases, there will also be a maximum size input that needs to be tested. A concrete example is
inserting into a doubly-linked circular list. The algorithm is different for lists of 0, 1, and many elements, so you should test for each case.

4. If things get too confusing, start over
Sometimes the right thing to do when you cannot figure out where to start debugging is to just start over writing the program. Usually this happens after you have made a major changes to the design of the project and have forgotten to propagate the changes through all of the existent code. Do this a few times, and you are likely to forget which constructs reflect the latest change and which ones still need to be changed. Throw in some old comments, and you have a mess.

Although, it may seem like a “waste” to discard the work, it really is not. By having written the discarded code, you have figured out what you really want to do. So when you start over, you will be able to complete the project faster, more efficiently, and it will be cleaner than trying to “save” the old work.

5. When all else fails, use a symbolic debugger
A symbolic debugger is a program that allows you to watch the state of your program as it runs and control how your program runs. It allows you to single-step through your code line by line and look at the values of all the variables. It never makes a mistake, so if you don't see the execution you were expecting or don't get the result values you were expecting, then you know something is amiss.

The rest of this handout is a very short tutorial on how to use gdb, a debugger under Linux for program written using g++. First, the source files in the project must be compiled with the -g option. This leaves the symbolic information in the executable. Usually, this is done by adding the option to the compiling command lines of a makefile. For example,

```
testdate.o : testdate.cpp date.h
<TAB>     g++ -Wall -c -g testdate.cpp
```

(Normally, the symbolic information is stripped out of the executable to make it smaller, since the information is not needed by the program and run-time system. When you are done using the debugger, you should take the -g flag out of your makefile commands.)

To use gdb, invoke it with (only) the program's name as its only command-line argument. For example,

```
dh27pc:/home/hwang/cs215/projects > gdb testdate
```

This will start the debugger and load the program. Then the debugger waits for commands. The most common commands (in alphabetical order) are:

- **break <function>** - sets a breakpoint that causes the program to stop running and (re)enter the debugger whenever the function is called. You can also break on line numbers.
- **run** - start running the program. If the program expects command-line arguments, they are given here.
- **continue** - continue running the program after a breakpoint. Usually used if you just want to check something at the breakpoint and then go on rather than single-stepping.
- **next** - execute the next statement of code at the current level. If it is function call, this means to execute the function and return in one step.
- **print <expression>** - print the value of the expression. Usually used to look at the values
of parameters and variables (including array and class elements). Can also be used as a calculator.

- **step** - step into the next statement of code. If it is a function call, this means the next shown step is the first line of the function. If it is not a function call, it has the same effect as **next**. (Note: many “basic” operations are actually function calls, for example, output statement `cout << x`, so make sure you want to step into these operator functions before you use step on these statements.)

- **backtrace** - see what the call stack looks like

- **quit** - exit the debugger. If the program is still running it will ask you if you really want to quit.

- **show <topic>** - the built-in help facility. **show help** will show instructions on how to use it.

There are many others. Consult the man page or using the built-in help facility to learn more.

Here is an example run using a driver program for the Date class. Comments not part of the run are enclosed in square brackets **[like this]**.

dh27pc:/home/hwang/cs215/projects > gdb rationaltest
[Program greeting deleted]
(gdb) break main [set breakpoint at start of program]
[can set at any function name or line number]
Breakpoint 1 at 0x8049600: file rationaltest.cpp, line 79.
(gdb) run [rationaltest does not have any command line arguments]
[if it did, they would be given here]
Starting program: /home/hwang/cs215/projects/rationaltest

Breakpoint 1, main () at rationaltest.cpp:79
79     cerr << "Construction and accessor tests" << endl;
[gdb displays the statement it is about to execute]
(gdb) next [execute next statement - output to cerr]
Construction and accessor tests [output from <<]
80     Rational r1,
(gdb) n [commands can be abbreviated if unique]
81     r2(5),
(gdb) print r1 [look at the value of r1, default constructed Rational]
$1 = {numerator = 0, denominator = 1}
(gdb) n
82     r3(5, 10),
(gdb) p r2 [look at the value of r2, single-argument construction]
$2 = {numerator = 5, denominator = 1}
(gdb) n
83     r4(10, 5),
(gdb) p r3 [look at r3, two-argument construction, note reduction is correct]
$3 = {numerator = 1, denominator = 2}
(gdb) n
84     r5 (0, 24);
(gdb) n
86     TestAccessors (r1);
(gdb) step [step into the function call]
TestAccessors(Rational const&j) (r1=@xbfffeec8) at rationaltest.cpp:19
19     cout << "Numerator: " << r1.GetNumerator();
(gdb) n
22    if (r1.GetNumerator() != 0)

(gdb) [Enter without a command repeats previous command]
25    cout << ", No reciprocal";
(gdb)
26    cout << ", MixedFraction: ";
(gdb)
27    r1.WriteAsMixedFraction(cout);
(gdb)
28    cout << endl;
(gdb)
Numerator: 0, Denominator: 1, Output: 0, No reciprocal, MixedFraction: 0
    [note: no output until flushed by endl]
29 }
(gdb)
main () at rationaltest.cpp:87 [return from TestAccessors]
87    TestAccessors (r2);
(gdb) [note: since current command is n, does not step into function]
Numerator: 5, Denominator: 1, Output: 5, Reciprocal: 1/5, MixedFraction: 5
    [Other calls to TestAccessors deleted]
93    r1 = Rational(-12,10);
(gdb)
94    r2 = Rational(8,-6);
(gdb) s [step into constructor]
Rational (this=0xbfffee98, n=8, d=-6) at rational.cpp:54
54    if (d == 0)
(gdb) n
57    numerator = n;
(gdb)
58    denominator = d;
(gdb)
59    Reduce();
(gdb) s [step into private member function]
Rational::Reduce() (this=0xbfffee98) at rational.cpp:26
26    if (numerator != 0)
(gdb) n
28       if (denominator < 0)  // Transfer sign to numerator
(gdb)
30       numerator = -numerator;
(gdb)
31       denominator = -denominator;
(gdb)
34       bool negative = false;
(gdb)
35       if (numerator < 0)  // Remember it is negative and negate for GCD
(gdb)
37       negative = true;
(gdb)
38       numerator = -numerator;
(gdb)
41       int div = GCD (numerator, denominator);
(gdb) backtrace [show call stack, can be used after a crash]
    [to see where execution stopped]
#0  Rational::Reduce() (this=0xbfffee98) at rational.cpp:41
#1  0x0804a1e5 in Rational (this=0xbfffee98, n=8, d=-6) at rational.cpp:59
GCD(int, int) (m=8, n=6) at rational.cpp:17
17 if ((n <= m) && (m % n == 0))
(gdb) bt [abbreviation for backtrace, note: one more level down]
#0 GCD(int, int) (m=8, n=6) at rational.cpp:17
#1 0x8049f5e in Rational::Reduce() (this=0xbfffee98) at rational.cpp:41
#2 0x804a1e5 in Rational (this=0xbfffee98, n=8, d=-6) at rational.cpp:59
#3 0x80496ff in main () at rationaltest.cpp:94
#4 0x42015704 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) n
19 if (n > m)
(gdb) 21 return GCD(n, m % n);
(gdb) 22 }
[Rest of Reduce execution deleted]
(gdb) Rational (this=0xbfffee98, n=8, d=-6) at rational.cpp:60 [return from Reduce]
60 } // end explicit value constructor
(gdb) main () at rationaltest.cpp:95 [return from constructor]
[Various tests and output deleted]
105 cerr << "Arithmetic tests" << endl;
(gdb) Arithmetic tests
106 TestArithmeticOps (r1, r2);
(gdb) s [step into TestArithmeticOps]
TestArithmeticOps(Rational const&, Rational const&) (r1=@0xbfffeeec8, r2=@0xbfffeeec0)
 at rationaltest.cpp:54
54 cout << r1 << " + " << r2 << " = " << (r1 + r2) << endl;
(gdb) s [step into operator+]
operator+(Rational const&, Rational const&) (lhs=@0xbfffeeec8, rhs=@0xbfffeeec0)
 at rational.cpp:111
111 return Rational ( (gdb) n
114 )
(gdb) n
-6/5 + -4/3 = -38/15 [output from above statement]
TestArithmeticOps(Rational const&, Rational const&) (r1=@0xbfffeeec8, r2=@0xbfffeeec0)
 at rationaltest.cpp:55 [return from operator+]
55 cout << r1 << " - " << r2 << " = " << (r1 - r2) << endl;
(gdb) quit [continue will cause program to run until next breakpoint]
The program is running. Exit anyway? (y or n) y

02/24/08

6 of 6