Discussion slides for week of September 25, 2007
The University of Michigan
Outline

- Recurrence relations
- Recursive functions
- HW2
- Unit testing
- Refactoring
- Pair Programming
- *Extreme Programming*
Recurrence relations

- Recurrence relations are those that are defined in terms of themselves.

\[ S(0) = 0 \]
\[ S(n) = n + S(n - 1) \]

- What does the above function do?
- How about a closed form?
Recurrence relations – closed forms

- Sometimes recurrence relations can be expressed in closed form.

\[
\begin{align*}
S(0) &= 0 \\
S(n) &= n + S(n - 1) \\
S'(n) &= \frac{n(n + 1)}{2}
\end{align*}
\]
Recurrence relations – Fibonacci

- How would we express the Fibonacci sequence as a recurrence relation?
- What about the tribonacci sequence?
Recursive Functions

- Implement recurrence relations
- Need a base case (Or else??)
- Why recursion
  - Better?
  - Faster?
HW2

• Questions 1 and 2 deal with recurrence relations and recursive functions

• Any questions?
Unit testing

- Unit testing is a methodology by which you test the software of your program.
- Tests are *unit* tests because they usually operate on a small, specific part of the system.
- They are organized in classes.
- We can implement an extremely simple version of unit testing with `assert`. 
Unit testing

PowerRaiser.h

class PowerRaiser {
public:
    PowerRaiser( unsigned int base );
    unsigned int getBase() const;
    unsigned int raise( unsigned int power ) const;

private:
    unsigned int base_;  
};
Unit testing

**PowerRaiser.cpp**

```cpp
PowerRaiser::PowerRaiser( unsigned int base ) :
    base_( base )
{
}

unsigned int PowerRaiser::getBase() const {
    return base_;  
}

unsigned int PowerRaiser::raise( unsigned int power ) const {
    if ( 0 == number ) {
        return 1;
    } else {
        return base_ * raise( power - 1 );
    }
}
```
Unit testing

PowerRaiserTest.h

class PowerRaiserTest {
public:
    void runAllTests();
    void testGetBase();
    void testGetPower();
    ...
};
Unit testing

PowerRaiserTest.cpp

```cpp
void PowerRaiserTest::testGetBase() {
    PowerRaiser p( 10 );
    assert( 10 == p.getBase() );
}

void PowerRaiserTest::testGetPower() {
    PowerRaiser p( 3 );
    assert( 1 == p.raise( 0 ) );
    assert( 3 == p.raise( 1 ) );
    assert( 9 == p.raise( 2 ) );
    assert( 4782969 == p.raise( 12 ) );
    assert( 15625 == PowerRaiser( 5 ).raise( 6 ) );
    assert( 1000000 == PowerRaiser( 10 ).raise( 6 ) );
    ...
}

void PowerRaiserTest::testAll() {
    testGetBase();
    testGetPower();
    ...
}
```

Data Structures and Algorithms
Unit testing

- Each time we make a change to the code base, we run all unit tests to make sure that all of the functionality is still there.

- If an error occurs, it signals a bug. We can figure out where it is with our tests, identify it immediately, and correct it.

- Or, if the bug cannot be resolved, we can revert our code (using SVN or CVS, for example) to the prior state.

- Thus, code repositories play a big part in unit testing.

- Tutorial for CVS:
Unit testing

• This *assert*-based unit testing is all right, but it would be nice to have a unit testing facility that **didn't break** on the first error that occurred.

• There are lots of unit testing frameworks for C++; for example CxxTest, CPPUnit, Unit++, Boost Test Libraries, QuickTest...

• Lots for other programming languages, as well.

• It's a good practice. It will save you from a lot of headaches, lower the cost of software development, and – if you know and do it – it will make you a lot more appealing to potential employers!
Unit testing

CxxTest example

// MyTestSuite.h
#include <cxxtest/TestSuite.h>

class MyTestSuite : public CxxTest::TestSuite
{
public:
    void testAddition( void )
    {
        TS_ASSERT( 1 + 1 > 1 );
        TS_ASSERT_EQUALS( 1 + 1, 2 );
    }
};
Refactoring

• Refactoring is a practice of making your code less risky.

• No functionality of your code will change after a refactoring, but it will be a lot cleaner and safer.

• The idea is to make sure all unit tests pass before refactoring, then refactor, then make sure all unit tests pass afterwards. (Otherwise revert the refactoring).

• Refactoring is driven by code smells. If something smells bad in the code, then we need to refactor.
Code duplication – pre-refactor *(bad)*

```cpp
void myFunction( int* ary, unsigned int arySize ) {
    for ( unsigned int i = 0; i < arySize; ++i ) {
        ary[ i ] += 1;
    }

    ... do some stuff here ...

    for ( unsigned int i = 0; i < arySize; ++i ) {
        ary[ i ] += 2;
    }

    ... do some stuff here ...

    if ( myBool ) {
        for ( unsigned int i = 0; i < ( arySize / 2 ); ++i ) {
            ary[ i ] += 5;
        }
    } else {
        for ( unsigned int i = ( arySize / 2 ); i < arySize; ++i ) {
            ary[ i ] -= 5;
        }
    }
}
```
void doIterativeOp(
    int* ary,
    unsigned int startIdx,
    unsigned int endIdx,
    int operand )
{
    for ( unsigned int i = startIdx; i < endIdx; ++i ) {
        ary[ i ] += operand;
    }
}

void myFunction( int* ary, unsigned int arySize ) { 
    doIterativeOp( ary, 0, arySize, 1 );
    ... 
    doIterativeOp( ary, 0, arySize, 2 );
    ...
    if ( myBool ) { 
        doIterativeOp( ary, 0, arySize / 2, 5 );
    } else { 
        doIterativeOp( ary, 0, arySize / 2, -5 );
    }
}
void doIterativeOp(
    int* ary,
    unsigned int startIdx,
    unsigned int endIdx,
    int operand )
{
    for ( unsigned int i = startIdx; i < endIdx; ++i ) {
        ary[ i ] += operand;
    }
}

void myFunction( int* ary, unsigned int arySize ) {
    doIterativeOp( ary, 0, arySize, 1 );
    ...
    doIterativeOp( ary, 0, arySize, 2 );
    ...
    doIterativeOp( ary, 0, arySize / 2, ( myBool ? 5 : -5 ) );
}
Pair Programming

- A discipline in which people program in pairs.
  - Two sets of eyes on the code instead of one reduces [costly] software errors – according to one study, by up to 30%!
  - Good way to transfer knowledge
  - Also a good way to take the edge off of the sometimes boring task of programming.
Extreme Programming!

• Extreme programming (XP) is a discipline that is composed (in part) of:
  − Software unit tests
    • Tests come first in XP!
  − Fast iterations
  − Constant refactoring
  − Pair programming