Unit testing

Unit testing

- Every function (near enough) should have at least one test.
- Code that is not tested does not work.
- Keep your tests forever. You'll be pleased when you need them again.

A testing framework for C++

- Unit testing revolution occurred in late 1990's.
- C++ is too old to have decent testing support.
- You need to choose a testing framework:
  - Boost unit test framework (`http://www.boost.org`),
  - `cppunit`,
  - Google test,
  - ...
  - We'll use a simple one of our own. Its built into `FMLib6.zip`.

Macros defined by `testing.h`

- `ASSERT`
- `ASSERT_APPROX_EQUAL`
- `DEBUG_PRINT`
- `INFO`
- `TEST`

The preprocessor for C allows you to define macros to avoid repetitive typing. Generally using macros is a bad idea because they lead to a language within a language. We'll use block caps for all macros.
Using macros looks pretty much the same as calling C++ functions . . . unless you look closely.

The ASSERT macro

```
ASSERT( test );
```

- This checks whether `test` is `true` and throws an error if it is not true.
- To speed up performance of a real system, all `ASSERT` checks are skipped when running the release build.
- This needs to be a macro because it prints out the line where the assertion failed and the test is eliminated in the release build. You can't do this with functions.

```
double safeSqrt( double x ) {
    ASSERT( x>=0 );
    return sqrt(x);
}
```

The ASSERT_APPROX_EQUAL macro

- Tests if two double values are nearly equal.
- Throws an error if they are not sufficiently close.
- Needs to be a macro for the same reasons as ASSERT.

```
void testNormInv() {
    ASSERT_APPROX_EQUAL( norminv( 0.975 ), 1.96, 0.01 );
}
```

The INFO macro

- Prints out a message together with the file name and line number.
- Allows you to build messages using <<.

```
double priceOptionByMonteCarlo( int numScenarios ) {
    if (numScenarios>1000000) {
        INFO(
        "Embarking upon a calculation with "
        <<numScenarios<<
        " scenarios");
    }
    ... /* length calculation goes here */ ...
}
```

The DEBUG_PRINT macro

- Prints out a message so long as:
  - You are running the debug build.
  - You have enabled debug by calling `setDebugEnabled(true)`.
- Allows you to build messages using <<.
- This stops `DEBUG_PRINT` slowing down release code.
- It stops you being overwhelmed with `DEBUG_PRINT` messages.

```
double max( double a, double b ) {
    DEBUG_PRINT( "Entering max("<<a<<", "<<b<<")");
    double ret = a>b ? a : b;
    DEBUG_PRINT( "Returning "<<ret);
    return ret;
}
```

The TEST macro

- Prints out that it is about to run a test.
- Runs the test.
- Indicates whether the test passed.
- Needs to be a macro to print out the name of the function automatically.

```
void testMatlib() {
    TEST( testNormInv );
    TEST( testNormCdf );
}
```

This looks a bit like a function call, but in reality you can't pass functions around like this in normal C++. This is typical of macros (and why we will try to avoid using them).

Using testing.h - part 1

In each of your files #include "testing.h".
For each test that you want to perform, write a function whose name begins test.

```
#include "testing.h"

... /*non testing code here*/ ...

static void testNormCdf() {
    ASSERT_APPROX_EQUAL( normcdf( 1.96 ), 0.975, 0.001 );
}

static void testNormInv() {
    ASSERT_APPROX_EQUAL( norminv( 0.975 ), 1.96, 0.01 );
}
```

Using testing.h - part 2

- In each file, write a single function which calls all the other test functions in turn. Use the TEST macro.
- Name this function after the .cpp file.
- Declare this function in the header.

In matlib.cpp

```
void testMatlib() {
    TEST( testNormInv );
    TEST( testNormCdf );
}
```

In matlib.h

```
void testMatlib();
```

Using testing.h - part 3

- In your main method, you should call all the test functions defined in the header files.

In matlib.cpp

```
int main() {
    testMatlib();
    ... /* run other tests */ ...
}
```

Insert DEBUG_PRINT where necessary

- Write lots of `DEBUG_PRINT` statements to help you follow what is going on. They won't be called until you call `setDebugEnabled( true )`.

```
void testMatlib() {
    // switch on the DEBUG_PRINT statements
    setDebugEnabled(true);
    TEST( testNormInv );
    setDebugEnabled(false);
    // switch them off again
    TEST( testNormCdf );
}
```

What have we gained?

- We no longer need to keep writing `main` methods and creating new projects
- We have a record of all the tests performed.
- Whenever we change our code we can retest immediately.
- We know that our code always works! (So long as we have enough tests).
- We have useful `DEBUG_PRINT` statements that will help us figure-out what is going on if we find a bug in future.

`testing.h` itself

- The file `testing.h` contains the definition of all the macros.
- It looks ugly and confusing.
- Writing macros is not advisable unless you are writing development tools so I won't explain how it works.
- Writing macros isn't recommended unless you need their special features. This really only applies to writing development tools like a testing framework.

Other frameworks are available

- We've rolled together a basic logging framework and testing framework.
- Lots of libraries you can use instead for both.
- Cool features like pie charts of how many tests are passing/failing, sending log messages to a database etc..
- Ask your boss which to use.

Test driven development

Write the tests first!

- No danger of you being too lazy to write them
- Forces you think about what problem you are actually trying to solve
- Tests your tests! They should fail to begin with.