Software Testing

Before we begin

- Project
  - Design feedback on the way.
  - Things to note:
    - More than 1 goal configuration
    - Solution is sequence of actions that take you from start to goal.
  - Clock problem due Oct 16.

Before we begin

- Constructors, Destructors, operator=
  - Lots of problems in Lab.

Constructor

- A constructor for a class is called when an object of that class is created:
  - Local / Stack Based
    - `Foo F (3, 4, "Joe");`
  - On Free Store
    - `Foo *Fptr (new Foo ((3, 4, "Joe"));`

Constructor

- Copy Constructor
  - Initializes an object based on the contents of another object of the same type.

```
Date (const Date &D) :
 d (D.d), m (D.m), y (D.y) {}
```

  - Object has access to non-public members of objects of the same class

Assignment operator

- operator=
  - Called when an assignment is made
  - Copies all relevant data from object assigner to assignee.
  - Must be declared as a class member
  - Should check for self-assignment!

Destructor

- A constructor for a class is called when the memory of an object of that class is reclaimed:
  - A global (static) object is reclaimed when the program terminates.
  - A local (automatic) object is reclaimed when the function terminates (stack is popped).
  - A dynamically allocated object is reclaimed when someone invokes the `delete` operator on it.
  - Like Java `finalize`

Constructor

- Important safety tips:
  - Always provide a default constructor
  - If your constructors perform any non-trivial work (e.g. memory allocation), should define the full suite of:
    - Constructors
    - Copy constructor
    - operator=

operator<<

- Writes a class object to an output stream

```
friend ostream& operator<< (ostream
  &os, const Foo &F)

Foo f;
cout << "My foo looks like" << f <<
  "and my g looks like" << g;
```

operator<<

```
class Foo
{
private:
     int foo1, foo2, foo3;
     …
}

friend ostream& operator<< (ostream &os, const Foo &F)
{
   os << "{" << F.foo1 << "," << F.foo2 << "," <<
  F.foo3 << "}";

   return os;
}
```

Before we begin

- Questions

Plan for this week

- Today: testing 1
- Monday: Testing 2 / Return exam 1
- Tuesday: Start on Templates

Software Testing

- From the software testing FAQ
  - TESTING means "quality control"
  - QUALITY CONTROL measures the quality of a product
  - QUALITY ASSURANCE measures the quality of processes used to create a quality product.
- No such thing as bug-free code!

Software testing

- Some definitions:
  - Error – Improper action of a programmer
  - Fault – The result of an error (improper logic).
  - Failure – Improper action of an executing program due to a fault.

Software testing

- Programmer writes:

```
char *foo = 0;
strcpy (foo, "I smell pointer problems");
```

  - This is an error
- The fault is that strcpy accesses a null pointer.
  - Many faults in C++ are pointer problems
  - Bad logic are problems too
- The failure:

```
Segmentation fault (Core dumped)
```

Software Development Cycle

- Gather Requirements
  - Find out what the user needs
- System Analysis
  - Express these needs formally in system terms
- Design
  - Design a high level solution
- Implementation
  - Turn solution into code
- Testing
  - Verify that the solution works
- Maintenance
  - Iterate the cycle

Software Testing

- When to test
  - Incrementally during implementation phase
    - Assure each unit or class meets design and functional specs
  - Limited testing of overall system during implementation
  - Formal system test during testing phase (after implementation is complete)
    - Alpha / Beta Testing
    - Tests program requirements

Software Testing

- Levels of testing
  - Unit testing
    - individual classes
  - Integration Testing
    - Assembly of one or more classes
  - System Test
    - System as a whole

Software Testing

- Types of Testing
  - Formal Verification
    - Reduce program to logical assertions and "prove" mathematically that the program is correct
  - Empirical Testing
    - Generate Test cases to see where errors exist.
    - Most testing that you will do will be empirical

Software Testing

- Empirical Testing
  - White Box testing
    - Assumes access to the code
    - Test all program flows
    - Covers all statements and conditions
  - Black Box Testing
    - Assumes no access to code or knowledge of implementation
    - Test cases generated based on requirements
    - Test valid and invalid input
    - Follow the contract

Programming by Contract

- Introduced by Bertrand Meyer, the creator of Eiffel.
- Creates a contract between the software developer and software user
  - Every feature, or method, starts with a *precondition* that must be satisfied by the consumer of the routine.
  - each feature ends with *postconditions* which the supplier guarantees to be true (if and only if the preconditions were met).
  - each class has an *invariant* which must be satisfied after any changes to the object represented by the class.

Programming by Contract

```
SomeClass::someFunction ( AnotherClass *fillMeWithData
  )
{
  // check any preconditions here
  preCondition ( fillMeWithData );

  // non-NULL check
  // do your stuff to add the functionality here ...

  // check post conditions
  postCondition ( fillMeWithData->hasData() ); // did
   we do what we said
  postCondition ( checkInvariant() ); // class
   invariant check required
}
```

Assertions

- Debugging mechanism to test condition at any point in the code
  - If condition is false, the program aborts and dumps core.
  - Useful for testing preconditions, postconditions and invariant checks.

Assertions

```
#include <cassert>

void foo (int *p)
{
   // At this point p should not be null
   assert (p != 0);
   …
}
```

Assertions

```
// constructor
//
// Preconditions:
// last & first are not empty (emptyString)
// age is not negative
//
Person::Person( string last, string first, int age, string
   firstJobName ): lastName(last), firstName(first),
   currentAge(age), currentJob(0)
{
   assert( last != emptyString );
   assert( first != emptyString );
   assert( age >= 0 );
   if ( firstJobName != noJob ) {
        currentJob = new Job( firstJobName );
   }
}
```

Questions?

Unit Testing

- A unit test tests at a "unit" (in C++, class) level.
- Why test classes individually?
  - Limit the scope of testing
  - Easier to generate test cases
  - Bugs found earlier (before integration) are easier to fix.

Unit Testing

- Black box approach
  - Must rely on functional specs and contracts
  - Supply inputs, check outputs.
    - Call methods with well chosen parameters
    - Call methods in various orders
    - Check object state via access methods.

Unit Testing

- White Box Approach
  - You have the code, look directly at execution,
    - Use debugger to set data member values
    - Use debugger to get data member values
    - Use debugger to check flow of execution

Unit Testing

- About writing test cases (black box)
  - A test case should be able to...
    - ...run completely by itself, without any human input. Unit testing is about automation.
    - ...determine by itself whether the function it is testing has passed or failed, without a human interpreting the results.
    - ...run in isolation, separate from any other test cases (even if they test the same functions). Each test case is an island.
  - This is what `try` does for lab submissions!

Unit Testing

- About writing test cases
  - **Testing for success**
    - **Method gives expected results on good input**
  - **Testing for failure**
    - **Methods should fail on bad input**
  - **Test the contract!**

Unit Testing

- Questions?

Quick Homework

- Here's a list class
- Come up with some black box test cases for it.

Summary

- Software Testing
- Error / Fault / Failure
- Level of Testing
- White Box vs. Black Box
- Unit Testing
- Questions?
  - Have a good weekend.