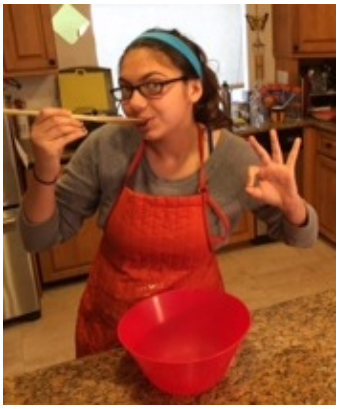
7.10 Unit testing (classes)

Testbenches

Like a chef who tastes food before serving, a class creator should test a class before allowing use. A ***testbench*** is a program whose job is to thoroughly test another program (or portion) via a series of input/output checks known as ***test cases***. ***Unit testing*** means to create and run a testbench for a specific item (or "unit") like a function or a class.



PARTICIPATION ACTIVITY 7.10.1: Unit testing of a class.

Start 2x speed

SampleClass
- Public item1
- Public item2
- Public item3

SampleClassTester program
- Create SampleClass object
- Test public item1
- Test public item2
- Test public item3

User program
- Create SampleClass object
- Use public item 2

Feedback?

The testbench below creates an object, then checks public functions for correctness. Some tests failed.

Features of a good testbench include:

- Automatic checks. Ex: Values are compared, as in `testData.GetNum1() != 100`. For conciseness, only fails are printed.
- Independent test cases. Ex: The test case for GetAverage() assigns new values, vs. relying on earlier values.
- ***100% code coverage***: Every line of code is executed. A good testbench would have more test cases than below.
- Includes not just typical values but also ***border cases***: Unusual or extreme test case values like 0, negative numbers, or large numbers.

Figure 7.10.1: Unit testing of a class.

```
Beginning tests.
   FAILED set/get num2
   FAILED GetAverage for 10, 20
   FAILED GetAverage for -10, 0
Tests complete.
```

```
#include <iostream>
using namespace std;

// Note: This class intentionally has errors

class StatsInfo {
public:
   void SetNum1(int numVal) { num1 = numVal; }
   void SetNum2(int numVal) { num2 = numVal; }
   int GetNum1() const { return num1; }
   int GetNum2() const { return num1; }
   int GetAverage() const;

private:
   int num1;
   int num2;
};

int StatsInfo::GetAverage() const {
   return num1 + num2 / 2;
}
// END StatsInfo class


// TESTBENCH main() for StatsInfo class
int main() {
   StatsInfo testData;

   // Typical testbench tests more thoroughly

   cout << "Beginning tests." << endl;

   // Check set/get num1
   testData.SetNum1(100);
   if (testData.GetNum1() != 100) {
      cout << "   FAILED set/get num1" << endl;
   }

   // Check set/get num2
   testData.SetNum2(50);
   if (testData.GetNum2() != 50) {
      cout << "   FAILED set/get num2" << endl;
   }

   // Check GetAverage()
   testData.SetNum1(10);
   testData.SetNum2(20);
   if (testData.GetAverage() != 15) {
      cout << "   FAILED GetAverage for 10, 20" << endl;
   }

   testData.SetNum1(-10);
   testData.SetNum2(0);
   if (testData.GetAverage() != -5) {
      cout << "   FAILED GetAverage for -10, 0" << endl;
   }

   cout << "Tests complete." << endl;

   return 0;
}
```

Feedback?

Defining a testbench as a friend class (discussed elsewhere) enables direct testing of private member functions

PARTICIPATION ACTIVITY 7.10.2: Unit testing of a class.

1) A class should be tested individually (as a "unit") before use in another program.
   - (•) True
   - ( ) False

   Correct
   The user expects the class to work, and may not even be able to debug the class. Bugs would also be harder to find.

2) Calling every function at least once is a prerequisite for 100% code coverage.
   - (•) True
   - ( ) False

   Correct
   If a testbench doesn't call a function, that function's lines of code can't possibly be executed. Multiple calls may be needed for 100% coverage.

3) If a testbench achieves 100% code coverage and all tests passed, the class must be bug free.
   - ( ) True
   - (•) False

   Correct
   100% code coverage is one goal of testing, but doesn't guarantee correctness. Different input values may yield different behavior for the same line of code.

4) A testbench should test all possible values, to ensure correctness.
   - ( ) True
   - (•) False

   Correct
   Testing all possible values is impossible for nearly any class; too many values exist. Instead, some typical values and some border cases should be tested.

5) A testbench should print a message for each test case that passes and for each that fails.
   - ( ) True
   - (•) False

   Correct
   A testbench may have hundreds of test cases. The tester is concerned about cases that FAIL. To make fails most evident, many programmers recommend only printing the fails.

Feedback?

Regression testing

Regression testing means to retest an item like a class anytime that item is changed; if previously-passed test cases fail, the item has "regressed".

A testbench should be maintained along with the item, to always be usable for regression testing. A testbench may be in a class' file, or in a separate file as in MyClassTest.cpp for a class in MyClass.cpp.

Testbenches may be complex, with thousands of test cases. Various tools support testing, and companies employ *test engineers* who only test other programmers' items. A large percent, like 50% or more, of commercial software development time may go into testing.

PARTICIPATION ACTIVITY 7.10.3: Regression testing.

1) Testbenches are typically disposed of after use.
   - ○ True
   - ◉ False

   Correct
   Testbenches are kept, to be used later as the item being tested is updated.

2) Regression testing means to check if a change to an item caused previously-passed test cases to fail.
   - ◉ True
   - ○ False

   Correct
   Even minor changes to an item may unexpectedly introduce bugs. An automated testbench can easily be run when an item is changed, to detect such "regression".

3) For commercial software, testing consumes a large percentage of time.
   - ◉ True
   - ○ False

   Correct
   Testing that consumes 50% or more is not unheard of. Writing thorough testbenches is a job in itself.

Feedback?

Erroneous unit tests

An erroneous unit test may fail even if the code being tested is correct. A common error is for a programmer to assume that a failing unit test means that the code being tested has a bug. Such an assumption may lead the programmer to spend time trying to "fix" code that is already

correct. Good practice is to inspect the code of a failing unit test before making changes to the code being tested.

Figure 7.10.2: Correct implementation of StatsInfo class.

```
#include <iostream>
using namespace std;

class StatsInfo {
public:
    void SetNum1(int numVal) { num1 = numVal; }
    void SetNum2(int numVal) { num2 = numVal; }
    int GetNum1() const { return num1; }
    int GetNum2() const { return num2; }
    int GetAverage() const;

private:
    int num1;
    int num2;
};

int StatsInfo::GetAverage() const {
    return (num1 + num2) / 2;
}
```

Feedback?

PARTICIPATION ACTIVITY 7.10.4: Erroneous unit test code causes failures even when StatsInfo is correctly implemented.

1 2 3 4 2x speed

```
StatsInfo testData;
testData.SetNum1(20);
testData.SetNum2(30);
if (testData.GetAverage() != 35) {
   cout << "   FAILED GetAverage for 20, 30" << endl;
}
```

Wrong expected value

```
StatsInfo testData;
testData.SetNum1(20);
testData.SetNum1(30);
if (testData.GetAverage() != 25) {
   cout << "   FAILED GetAverage for 20, 30" << endl;
}
```

Test object's data not properly set

Not properly initializing the test object's data is another common error.

Feedback?

PARTICIPATION ACTIVITY 7.10.5: Identifying erroneous test cases.

Assume that StatsInfo is correctly implemented and identify each test case as valid or erroneous.

1) num1 = 1.5, num2 = 3.5, and the expected average = 2.5
   - ○ Valid
   - ◉ Erroneous

Correct

Although the test case makes sense mathematically, the implementation details of StatsInfo are being ignored. StatsInfo uses integer arithmetic and cannot store values 1.5 or 3.5, nor return 2.5 from GetAverage(). So the test case is erroneous.

2) num1 = 33, num2 = 11, and the expected average = 22
   - ◉ Valid
   - ○ Erroneous

Correct

33 and 11 are valid integers, and (33 + 11) / 2 = 22, so the test case is valid.

3) num1 = 101, num2 = 202, and the expected average = 152
   - ○ Valid
   - ◉ Erroneous

Correct

Integer division drops the remainder of the calculation (101 + 202) / 2, so the average is 151, not 152. The expected value is wrong and the test case is erroneous.

Feedback?

Exploring further:

- C++ Unit testing frameworks from accu.org.

CHALLENGE ACTIVITY 7.10.1: Enter the output of the unit tests.

Note: There's always an error.

Jump to level 1

Type the program's output.

```
#include <iostream>
using namespace std;

class Rectangle {
public:
   void SetSize(int heightVal, int widthVal) {
      height = heightVal;
      width = widthVal;
   }
   int GetArea() const;
   int GetPerimeter() const;

private:
   int height;
   int width;
};

int Rectangle::GetArea() const {
   return height * width;
}

int Rectangle::GetPerimeter() const {
   return (height * 2) + (width * 2);
}

int main() {
   Rectangle myRectangle;

   myRectangle.SetSize(1, 1);
   if (myRectangle.GetArea() != 2) {
      cout << "FAILED GetArea() for 1, 1" << endl;
   }
   if (myRectangle.GetPerimeter() != 4) {
      cout << "FAILED GetPerimeter() for 1, 1" << endl;
   }

   myRectangle.SetSize(2, 3);
   if (myRectangle.GetArea() != 8) {
      cout << "FAILED GetArea() for 2, 3" << endl;
   }
   if (myRectangle.GetPerimeter() != 12) {
      cout << "FAILED GetPerimeter() for 2, 3" << endl;
   }

   return 0;
}
```

```
FAILED GetArea
FAILED GetArea
FAILED GetPerim
```

1 2

Check Next **Done**. Click any level to practice more. Completion is preserv

✔ In main(), one or more unit tests are expecting the wrong value. The output helps indicate v
double-checking each expected value is good practice.

Yours

```
FAILED GetArea() for 1, 1
FAILED GetArea() for 2, 3
FAILED GetPerimeter() for 2, 3
```

Expected

```
FAILED GetArea() for 1, 1
FAILED GetArea() for 2, 3
FAILED GetPerimeter() for 2, 3
```

Feedback?

CHALLENGE ACTIVITY 7.10.2: Unit testing of a class.

Write a unit test for addInventory(), which has an error. Call redSweater.addInventory() with parameter sweaterShipment. Print the shown error if the subsequent quantity is incorrect. Sample output for failed unit test given initial quantity is 10 and sweaterShipment is 50:

```
Beginning tests.
   UNIT TEST FAILED: addInventory()
Tests complete.
```

Note: UNIT TEST FAILED is preceded by 3 spaces.

```
27
28 int main() {
29    InventoryTag redSweater;
30    int sweaterShipment;
31    int sweaterInventoryBefore;
32
33    sweaterInventoryBefore = redSweater.getQuantityRemaining();
34    cin >> sweaterShipment;
35
36    cout << "Beginning tests." << endl;
37
38    // FIXME add unit test for addInventory
39
40    /* Your solution goes here  */
41    redSweater.addInventory(sweaterShipment);
42    if(redSweater.getQuantityRemaining()){
43       cout << "   UNIT TEST FAILED: addInventory()" << endl;
44    }
45
46    cout << "Tests complete." << endl;
47
48    return 0;
```

Run

✓ Inventory is 0, shipment input is 25. Testing that quantityRemaining was updated to 25.

Your value `25`

✗ Testing with sweaterShipment input of 25. addInventory updates quantityRemaining.

Output differs. See highlights below. Special character legend

Your output
```
Beginning tests.
   UNIT TEST FAILED: addInventory()↵
Tests complete.
```

Expected output
```
Beginning tests.
Tests complete.
```

✓ Inventory is 25, shipment input is 5. Testing that quantityRemaining remains 25.

Your value `25`

✓ Testing sweaterShipment input of 5. addInventory does not update quantityRemaining.

Your output
```
Beginning tests.
   UNIT TEST FAILED: addInventory()
Tests complete.
```

Feedback?