6.5 Unit testing (functions)

Testing is the process of checking whether a program behaves correctly. Testing a large program can be hard because bugs may appear anywhere in the program, and multiple bugs may interact. Good practice is to test small parts of the program individually, before testing the entire program, which can more readily support finding and fixing bugs. ***Unit testing*** is the process of individually testing a small part or unit of a program, typically a function. A unit test is typically conducted by creating a ***testbench***, a.k.a. test harness, which is a separate program whose sole purpose is to check that a function returns correct output values for a variety of input values. Each unique set of input values is known as a ***test vector***.

Consider a function HrMinToMin() that converts time specified in hours and minutes to total minutes. The figure below shows a test harness that tests that function. The harness supplies various input vectors like (0,0), (0,1), (0,99), (1,0), etc.

Figure 6.5.1: Test harness for the function HrMinToMin().

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

// Function converts hrs/min to min
int HrMinToMin(int origHours, int origMinutes) {
   int totMinutes; // Resulting minutes

   totMinutes = (origHours * 60) + origMinutes;

   return origMinutes;
}

int main() {

   cout << "Testing started" << endl;

   cout << "0:0, expecting 0, got: "    << HrMinToMin(0, 0)  <<
endl;
   cout << "0:1, expecting 1, got: "    << HrMinToMin(0, 1)  <<
endl;
   cout << "0:99, expecting 99, got: "  << HrMinToMin(0, 99) <<
endl;
   cout << "1:0, expecting 60, got: "   << HrMinToMin(1, 0)  <<
endl;
   cout << "5:0, expecting 300, got: "  << HrMinToMin(5, 0)  <<
endl;
   cout << "2:30, expecting 150, got: " << HrMinToMin(2, 30) <<
endl;
   // Many more test vectors would be typical...

   cout << "Testing completed" << endl;

   return 0;
}
```

```
Testing started
0:0, expecting 0, got: 0
0:1, expecting 1, got: 1
0:99, expecting 99, got: 99
1:0, expecting 60, got: 0
5:0, expecting 300, got: 0
2:30, expecting 150, got:
30
Testing completed
```

Feedback?

Manually examining the program's printed output reveals that the function works for the first several vectors, but fails on the next several vectors, highlighted with colored background. Examining the output, one may note that the output minutes is the same as the input minutes; examining the code indeed leads to noticing that parameter origMinutes is being returned rather than variable totMinutes. Returning totMinutes and rerunning the test harness yields correct results.

Each bug a programmer encounters can improve a programmer by teaching him/her to program differently, just like getting hit a few times by an opening door teaches a person not to stand near a closed door.

PARTICIPATION ACTIVITY 6.5.1: Unit testing.

1) A test harness involves temporarily modifying an existing program to test a particular function within that program.
   - ○ True
   - ○ False

2) Unit testing means to modify function inputs in small steps known as units.
   - ○ True
   - ○ False

Feedback?

Manually examining a program's printed output is cumbersome and error prone. A better test harness would only print a message for incorrect output. The language provides a compact way to print an error message when an expression evaluates to false. assert() is a macro (similar to a function) that prints an error message and exits the program if assert()'s input expression is false. The error message includes the current line number and the expression (a nifty trick enabled by using a macro rather than an actual function; details are beyond our scope). Using assert requires first including the cassert library, part of the standard library, as shown below.

Figure 6.5.2: Test harness with assert for the function HrMinToMin().

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

double HrMinToMin(int origHours, int origMinutes) {
   int totMinutes;  // Resulting minutes

   totMinutes = (origHours * 60) + origMinutes;

   return origMinutes;
}

int main() {

   cout << "Testing started" << endl;

   assert(HrMinToMin(0, 0)  == 0);
   assert(HrMinToMin(0, 1)  == 1);
   assert(HrMinToMin(0, 99) == 99);
   assert(HrMinToMin(1, 0)  == 60);
   assert(HrMinToMin(5, 0)  == 300);
   assert(HrMinToMin(2, 30) == 150);
   // Many more test vectors would be typical...

   cout << "Testing completed" << endl;

   return 0;
}
```

```
Testing started
Assertion failed: (HrMinToMin(1, 0) == 60), function main, file main.cpp, line 20.
```

Feedback?

Using branches for unit tests

If you have studied branches, you may recognize that each print statement in main() could be replaced by an if statement like:

```
if ( HrMinToMin(0, 0) != 0 ) {
   cout << "0:0, expecting 0, got: " << HrMinToMin(0, 0) << endl;
}
```

But the assert is more compact.

assert() enables compact readable test harnesses, and also eases the task of examining the program's output for correctness; a program without detected errors would simply output "Testing started" followed by "Testing completed".

A programmer should choose test vectors that thoroughly exercise a function. Ideally the programmer would test all possible input values for a function, but such testing is simply not practical due to the large number of possibilities -- a function with one integer input has over 4

billion possible input values, for example. Good test vectors include a number of normal cases that represent a rich variety of typical input values. For a function with two integer inputs as above, variety might include mixing small and large numbers, having the first number large and the second small (and vice-versa), including some 0 values, etc. Good test vectors also include ***border cases*** that represent fringe scenarios. For example, border cases for the above function might include inputs 0 and 0, inputs 0 and a huge number like 9999999 (and vice-versa), two huge numbers, a negative number, two negative numbers, etc. The programmer tries to think of any extreme (or "weird") inputs that might cause the function to fail. For a simple function with a few integer inputs, a typical test harness might have dozens of test vectors. For brevity, the above examples had far fewer test vectors than typical.

PARTICIPATION ACTIVITY 6.5.2: Assertions and test cases.

1) Using assert() is a preferred way to test a function.
   - True
   - False

2) For function, border cases might include 0, a very large negative number, and a very large positive number.
   - True
   - False

3) For a function with three integer inputs, about 3-5 test vectors is likely sufficient for testing purposes.
   - True
   - False

4) A good programmer takes the time to test all possible input values for a function.
   - True
   - False

Feedback?

Exploring further:

- assert reference page from cplusplus.com

CHALLENGE ACTIVITY 6.5.1: Unit testing.

Add two more statements to main() to test inputs 3 and -1. Use print statements similar to the existing one (don't use assert).

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

// Function returns origNum cubed
int CubeNum(int origNum) {
   return origNum * origNum * origNum;
}

int main() {

   cout << "Testing started" << endl;

   cout << "2, expecting 8, got: " << CubeNum(2) << endl;

   /* Your solution goes here  */
   cout << "3, expecting 27, got: " << CubeNum(3) << endl;
   cout << "-1, expecting -1, got: " << CubeNum(-1) << endl;

   cout << "Testing completed" << endl;

   return 0;
```

Run ✓ All tests passed

✓ Testing 3 and -1 unit tests

Your output

```
3, expecting 27, got: 27
-1, expecting -1, got: -1
```

✓ Testing all

Your output

```
Testing started
2, expecting 8, got: 8
3, expecting 27, got: 27
-1, expecting -1, got: -1
Testing completed
```

Feedback?