# White-Box Testing

Steven Zeil

August 3, 2013

## Contents

**Styles of Testing**

Testing traditionally can be conducted in three styles

- *Black-Box* testing
  - Try to choose "smart" tests based on the requirements, without looking at the code.
- *White-Box* testing
  - Try to choose "smart" tests based on the structure of the code, with minimal reference to the requirements.
- *Random* testing
  - Try to use directed random selection to choose tests that are "representative" of how the program will be used.

. . . . . .

We've already looked at black-box techniques. In this lesson we take up white-box testing.

**Complementary Strategies**

Black-box and white-box testing are complementary.

- projects should combine elements of each
- Black-box tests
  - can be designed earlier
  - are better at catching errors of omission and errors in design
- White-box tests
  - must await the development of the code
  - are better at catching errors of commission and errors after design

. . . . . .

**Combining Black- and White-Box Testing**

Best to

- Do BB testing first
- Measure how well your BB tests did at WB coverage
- Add tests as needed to achieve your WB goals

. . . . . .



# 1 Statement Coverage

**Statement Coverage**

*statement coverage* is the selection of tests so that every statement has been executed at least once.

- the most basic of all white-box methods
- does not have to happen in a single test – as long as one test in the suite executes a statement, it's covered

. . . . . .

**Example: statement coverage**

**Question:** How would you test this function so that every statement is covered?

```
template <typename T>
int seqSearch(const T list[], int listLength,
              T searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}
```

. . . . . .

**Answer:** You will have to run at least two tests, because you can't execute both `return` statements in one call.

- Need at least one test where *searchItem* is somewhere in *list*
  - Will cover the for loop header, the if statement, and the first return
- Need at least one test where *searchItem* is not in *list*
  - Covers the second return statement

This does *not* mean that you are limited to just two tests. But statement coverage will require that you have at least these two.

It's also worth noting that you would probably have gotten both of these during black-box testing as functional cases.

- But, if you missed one during BB testing, the WB requirement would serve as a reminder.

**100% Coverage may not be Possible**

Can we cover this statement with a test?

```
int BidderCollection::add (const Bidder& value)
//  Adds this bidder
//Pre: getSize() < getMaxSize()
{
  if (size < MaxSize)
    {
     addToEnd (elements, size, value);
     return size - 1;
    }
  else
    {
     cerr << "BidderCollection::add - collection is full" << endl;
     exit(1);
    }
}
```

- During unit test, certainly.
- During an integration or system test, probably not.
  - If the program calling this function allows execution of that statement, it's almost certainly a bug in that program.

– Typical of code inserted as "defensive programming"

. . . . . .

## 1.1 Monitoring Statement Coverage

**Monitoring Statement Coverage**

We can check whether we have covered statements by adding debugging output.

- add a unique output to each block of "straight line" code

```
#define COVER cerr << "Block " << __FILE__ << ":" << __LINE__ << endl
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
    COVER;
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            {
             COVER;
             return loc;
            }

    COVER;
    return -1;
}
```

However, g++ and other compilers offer tools that can automate this process.

. . . . . .

**Monitoring Statement Coverage with `gcov`**

We can go a long way by simply being *aware* of the idea of statement coverage when we develop our tests.

- Recognize that we want tests to cover every branch
- But as programs get more complicated, even experienced testers do a poor job of stmt coverage
  - Automatic coverage tools can help

. . . . . .

**gcov**

- Coverage tools need to "understand" the control flow of your code
  - typically have to duplicate much of the work of a compiler
  - This makes them often complicated and expensive.
- It's a lot easier to do this if the compiler provides support in the first place
  - The gcc/g++ suite includes a coverage tool, `gcov`

. . . . . .

**Example: Unit Testing the Array Search Functions**

- We will look at doing a unit test of the three search functions in `arrayUtils.h`
- First, we need a test driver. We can go with either of two styles.
  - In one, `gcovDemo.cpp`, we write a main program that reads data from a text stream (e.g., standard in), uses that data to construct arrays, and invokes each function on those arrays, printing the results of each.
  - Alternatively, `gcovDemo2.cpp` uses no external input but instead contains code to construct the data we need, then `asserts` that the search functions produce the expected results.
- For this example we'll use the first style.

. . . . . .

**Compiling for gcov**

To use `gcov`, we compile with special options

- `-fprofile-arcs -ftest-coverage`
- In Unix, add these to the compilation commands or add them to the C++ options in a `makefile` such as this one: `makefile`.
  - In Code::Blocks,
    - add these to "`Project -> Build Options -> Compiler Settings -> Other options`", and
    - going to "`Project -> Build Options -> Linker Settings`" and, under `Link libraries`, add the `libgcov.a` file found inside the Code::Blocks installation directory, probably under `MingW\lib\gcc\mingw32\3.4.5` (the version number at the end may vary).



- When the code has been compiled, in addition to the usual files there will be several files with endings like `.gcno` These hold data on where the statements and branches in our code are.

.....

**Running Tests with gcov**

- Run your tests normally.
- As you test, a `*.gcda` file will accumulate data on which statements have been covered.

.....

**Viewing Your Report**

- Run `gcov` *`mainProgram`*
  - the immediate output will be a report on the percentages of statements covered in each source code file.
  - Also creates a detailed report for each source code file. e.g.,

```
    -:   69:template <typename T>
    -:   70:int seqSearch(const T list[], int listLength, T searchItem)
    -:   71:
    1:   72:    int loc;
    -:   73:
    2:   74:    for (loc = 0; loc < listLength; loc++)
    2:   75:        if (list[loc] == searchItem)
    1:   76:            return loc;
    -:   77:
#####:   78:    return -1;
    -:   79:
```

.....

Code::Blocks users on Windows will find `gcov` in the same directory as the `g++` executables. The easiest way to handle this is to start a Windows `cmd` window, cd to the directory where Code::Blocks has placed your compiled code, then do

*`pathToExecutables`*`\gcov` *`mainProgram`*

**Interpreting the gcov Report**

```
    -:   69:template <typename T>
    -:   70:int seqSearch(const T list[], int listLength, T searchItem)
    -:   71:
    1:   72:    int loc;
    -:   73:
    2:   74:    for (loc = 0; loc < listLength; loc++)
    2:   75:        if (list[loc] == searchItem)
    1:   76:            return loc;
    -:   77:
#####:   78:    return -1;
    -:   79:
```

- Lists number of times each statement has been executed
- Lists #### if a statement has never been executed
  - Focus on these as you choose additional tests

. . . . . .

**Resetting the Report Data**

- Delete the .da files to reset all counts to zero.
  - e.g., if you change or remove tests from your test set

. . . . . .

## 2 Branch Coverage

**Branch Coverage**

*Branch coverage* is a requirement that, for each branch in the program (e.g., if statements, loops), each branch have been executed at least once during testing.

- (It is sometimes also described as saying that each branch condition must have been true at least once and false at least once during testing.)

. . . . . .

**Similarity to Statement Coverage**

How is branch coverage different from statement coverage? If you have covered every statement in

```
if (x > 0)
   y = x;
else
   y = -x;
```

or in

```
while (z > 0)
   --z;
```

have you not covered all the branches as well?

. . . . . .

**Difference from Statement Coverage**

Only difference occurs when branches are "empty":

```
y = x;
if (x < 0)
   y = -x;
```

or in

```
while (z-- > 0);
```

In these cases branch coverage would require tests in which x and z are positive. Statement coverage would not.

. . . . . .

**Example: branch coverage**

**Question:** How would you test this function so that every branch is covered?

```
template <typename T>
int seqSearch(const T list[], int listLength,
              T searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}
```

. . . . . .

**Answer:** If we start with our statement coverage tests:

- One test where *searchItem* is somewhere in *list*
- One test where *searchItem* is not in *list*

we are not guaranteed that we would cover the "empty else" part of the `if` statement.

- Need at least one test where the `list[loc] != searchItem`
  - Could be combined with either of the first two tests, either
    - *searchItem* is somewhere in *list* but not in position 0, or
    - *searchItem* is not in *list* and *listLength* $\geq 1$

. . . . . .

## 2.1 Monitoring Branch Coverage with `gcov`

**Awareness**

Again, just being *aware* of the idea of branch coverage can help guide our tests.

- Recognize that we want tests to cover every branch
- But as programs get more complicated, even experienced testers do a poor job of stmt/branch coverage
  - Automatic coverage tools can help

. . . . . .

**gcov Does Branch Coverage**

`gcov` can report on branches taken.

Just add new options to the `gcov` command:

```
gcov -b -c mainProgram
```

. . . . . .

**Reading gcov Branch Info**

- gcov reports
  - Number of times each function call successfully returned
  - # of times a branch was *executed* (really how many times the branch condition was evaluated)
  - and # times each branch was *taken*
    - For branch coverage, this is the relevant figure

. . . . . .



**Example: gcov Branch Coverage report**

```
        -:   84:template <typename T>
        -:   85:int seqOrderedSearch(const T list[], int listLength, T searchItem)
        -:   86:
        1:   87:    int loc = 0;
        -:   88:
        1:   89:    while (loc < listLength && list[loc] < searchItem)
branch  0 taken 0
call    1 returns 1
branch  2 taken 0
branch  3 taken 1
        -:   90:
    #####:   91:       ++loc;
branch  0 never executed
        -:   92:
        1:   93:    if (loc < listLength && list[loc] == searchItem)
branch  0 taken 0
call    1 returns 1
branch  2 taken 0
        1:   94:       return loc;
branch  0 taken 1
        -:   95:    else
    #####:   96:       return -1;
        -:   97:
```

.....

**Report Organization**

- Report is organized by *basic blocks*, straight-line sequences of code terminated by a branch or a call
- Hard to map to specific source code constructs
  - lowest-numbered branch is often the leftmost condition
  - Fact of life that compilers insert branches and calls that are often invisible to us

.....



**What, Really, is a Branch?**

- A "branch" is anything that causes the code to not continue on in straight-line fashion
  - The branch listed right after an "if" is the "branch" that jumps around the "then" part to go to the "else" part.
- && and || operators introduce their own branches
- Other branches may be hidden
  - Contributed by calls to inline functions
  - Or just a branch generated by the compiler's choice of code generation

. . . . . .

## 3 Loop Coverage

**Loop Coverage**

Various definitions of *loop coverage* exist.
One of the more useful suggests that a loop is covered

- if in at least one test the body was executed 0 times, and
- if in some test the body was executed exactly once, and
- if in some test the body was executed more than once.

A good idea to keep in mind, but harder to monitor.

. . . . . .

## 4 Final Thought: Combining Black and White-Box Testing

**Final Thought: Combining Black and White-Box Testing**

Best to

- Do BB testing first
- Measure how well your BB tests did at WB coverage
- Add tests as needed to achieve your WB goals

. . . . . .



## 5 arrayUtils.h

```
#ifndef ARRAYUTILS_H
#define ARRAYUTILS_H



//  Add to the end
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter
template <typename T>
void addToEnd (T* array, int& size, T value)
{
   array[size] = value;
   ++size;
}



// Add value into array[index], shifting all elements already in positions
//    index..size-1 up one, to make room.
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

template <typename T>
void addElement (T* array, int& size, int index, T value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= index) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  array[index] = value;
  ++size;
```

```
}


// Assume the elements of the array are already in order
// Find the position where value could be added to keep
//    everything in order, and insert it there.
// Return the position where it was inserted
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

template <typename T>
int addInOrder (T* array, int& size, T value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= 0 && value < array[toBeMoved]) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  array[toBeMoved+1] = value;
  ++size;
  return toBeMoved+1;
}


// Search an array for a given value, returning the index where
//    found or -1 if not found.
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
```



```
}


// Search an ordered array for a given value, returning the index where
//    found or -1 if not found.
template <typename T>
int seqOrderedSearch(const T list[], int listLength, T searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
      {
       ++loc;
      }
    if (loc < listLength && list[loc] == searchItem)
       return loc;
    else
       return -1;
}


// Removes an element from the indicated position in the array, moving
// all elements in higher positions down one to fill in the gap.
template <typename T>
void removeElement (T* array, int& size, int index)
{
  int toBeMoved = index + 1;
  while (toBeMoved < size) {
    array[toBeMoved] = array[toBeMoved+1];
    ++toBeMoved;
  }
  --size;
}



// Search an ordered array for a given value, returning the index where
//    found or -1 if not found.
template <typename T>
int binarySearch(const T list[], int listLength, T searchItem)
```

```
{
    int first = 0;
    int last = listLength - 1;
    int mid;

    bool found = false;

    while (first <= last && !found)
    {
        mid = (first + last) / 2;

        if (list[mid] == searchItem)
            found = true;
        else
            if (searchItem < list[mid])
                last = mid - 1;
            else
                first = mid + 1;
    }

    if (found)
        return mid;
    else
        return -1;
}




#endif
```

## 6 gcovDemo.cpp

```
#include <cassert>
#include <iostream>
#include <sstream>
#include <string>

#include "arrayUtils.h"
```

```
using namespace std;



// Unit test driver for array search functions




int main(int argc, char** argv)
{
  // Repeatedly reads tests from cin
  // Each test consists of a line containing one or more words.
  // The first word is one that we want to search for. The
  // remaining words are placed into an array and represent the collection
  // we will search through.

  string line;
  getline (cin, line);
  while (cin)
    {
      istringstream in (line);
      cout << line << endl;
      string toSearchFor;
      in >> toSearchFor;
      int nWords = 0;
      string words[100];
      while (in >> words[nWords])
    ++nWords;

      cout << seqSearch (words, nWords, toSearchFor)
       << " "
       << seqOrderedSearch (words, nWords, toSearchFor)
       << " "
       << binarySearch (words, nWords, toSearchFor)
       << endl;

      getline (cin, line);
```



```
    }

  return 0;
}
```

## 7 gcovDemo2.cpp

```
#include <cassert>
#include <iostream>
#include <string>

#include "arrayUtils.h"

using namespace std;



// Unit test driver for sequential search function



void test1()
{
  cout << "test 1" << endl;
  int arr1[] = {1, 2, 3};
  int k = seqSearch (arr1, 3, 1);
  assert (k == 0);
  k = seqOrderedSearch (arr1, 3, 1);
  assert (k == 0);
  k = binarySearch (arr1, 3, 1);
  assert (k == 0);
}



void test2()
{
  cout << "test 2" << endl;
  string arr2[] = {"abc", "def", "ghi"};
  string key = "xyz";
```

```
  int k = seqSearch (arr2, 3, key);
  assert (k == -1);
  k = seqOrderedSearch (arr2, 3, key);
  assert (k == -1);
  k = binarySearch (arr2, 3, key);
  assert (k == -1);
}


int main(int argc, char** argv)
{
  test1();
  test2();

  return 0;
}
```

## 8 makefile

```
#############################################################################
# Macro definitions for "standard" C and C++ compilations
#
# Edit the next 5 definitions. After that, "make" will
#   build the program.
#
#  Define special compilation flags for C++ compilation. These may change when
#  we're done testing and debugging.
CPPFLAGS=-g -O0 -fprofile-arcs -ftest-coverage
#
#  Define special compilation flags for C compilation. These may change when
#  we're done testing and debugging.
CFLAGS=-g
#
#  What is the name of the program you want to create?  (See below for notes
#     on using this makefile to generate multiple programs.)
TARGET=gcovDemo.exe
#
#  List the object code files to be produced by compilation. Normally this
#  list will include one ".o" file for each C++ file (with names ending in
#  ".cpp", ".cc" or ".C"), and/or each C file (with names ending in ".c").
```

```
#  Do NOT list .h files. For example, if you are building a program from
#  source files foo.c, foo.h, bar.cpp, baz.cc, and bam.h you would use
#      OBJS1=foo.o bar.o baz.o
OBJS=gcovDemo.o
#
#  What program should be used to link this program? If the program is
#  even partly C++, use g++.  If it is entirely C, use gcc.
LINK=g++ $(CPPFLAGS)
#LINK=gcc $(CFLAGS)
#
#  Define special linkage flags.  Usually, these are used to include
#  special libraries of code, e.g., -lm to add the library of mathematical
#  routines such as sqrt, sin, cos, etc.
LFLAGS=-lm
#
#
#
#  In most cases, you should not change anything below this line.
#
#  The following is "boilerplate" to set up the standard compilation
#  commands:
#
.SUFFIXES:
.SUFFIXES: .d .o .h .c .cc .C .cpp
.c.o: ; $(CC) $(CFLAGS) -MMD -c $*.c
.cc.o: ; $(CPP) $(CPPFLAGS) -MMD -c $*.cc
.C.o: ; $(CPP) $(CPPFLAGS) -MMD -c $*.C
.cpp.o: ; $(CPP) $(CPPFLAGS) -MMD -c $*.cpp

CC=gcc
CPP=g++

%.d: %.c
touch $@
%.d: %.cc
touch $@
%.d: %.C
touch $@
%.d: %.cpp
touch $@
```



```
DEPENDENCIES = $(OBJS:.o=.d)

#
# Targets:
#
all: $(TARGET)

$(TARGET): $(OBJS)
$(LINK) $(FLAGS) -o $(TARGET) $(OBJS) $(LFLAGS)

clean:
-rm -f $(TARGET) $(OBJS) $(DEPENDENCIES) make.dep *.bb *.bbg *.gc* *.da gmon.out


make.dep: $(DEPENDENCIES)
-cat $(DEPENDENCIES) > make.dep

include make.dep
```