**Testing and Debugging Techniques**

Testing and debugging skills are integral skills that a software developer refines throughout their career.  Testing ensures that a program executes successfully for a well-defined range of values.  Such a program might still crash for values outside this range.  Each program needs to be throughly tested before release to a user community and with each patch to that release.  Compilers identify syntactic errors with respect to the rules of the programming language, but cannot readily identify semantic errors; that is, errors in the meaning or intent of the code.  Walkthroughs and code analysis help identify these errors.

Much of the time and effort involved in ensuring that a program executes correctly for all practical cases is spent on testing and debugging.  Testing ensures that all of the paths through the program envisaged by the designer produce correct results.  Debugging locates those 'bugs' that produce incorrect results.  Over the years, computer scientists have developed sophisticated tools for testing and debugging.  These tools are available in various development environments.  The traditional walkthrough technique simulates instruction-by-instruction stepping of the CPU and its updating of program data in primary memory.

This chapter describes the kinds of errors that are common in source code, introduces testing and debugging techniques and shows how to layout program variables in tabular form to facilitate comprehensive walks through the source code.

Errors

Programming errors are classified into either of two kinds:

* syntactic errors
* semantic errors

Syntactic Errors

Syntactic errors are errors that break the rules of the programming language.  The most common syntactic errors in the C language are:

* missing semi-colon
* unnecessary semi-colon terminator in a #define directive
* undeclared variable name
* mismatched parentheses
* left-side of an assignment expression is not a defined memory location
* return statement is missing

Techniques for identifying syntactic errors include

* reading code statements (walkthroughs)
* compiler error messages (compiler output)
* comparing error messages from different compilers - some are more cryptic than others

Semantic Errors

Semantic errors are errors that fail to implement the intent and meaning of the program designer.  The more common semantic errors are:

* = instead of ==
* iteration without a body (for/while followed by a semi-colon)
* uninitialized variable
* infinite iteration
* incorrect operator order in a compound expression
* dangling else
* off-by-one iteration
* integer division and truncation
* mismatched data types
* & instead of &&

Techniques for identifying semantic errors include:

* vocalization - use your sense of hearing to identify the error (compound conditions)
* intermediate output - printf() statements at critical stages
* walkthrough table
* interactive debugging using
  + Visual Studio IDE - integrated debugger for Windows OSs
  + Eclipse IDE - integrated debugger for Linux OSs
  + gdb - GNU debugger for gcc

Testing Techniques

The two categories of software testing techniques are:

* black box
* white box

Black Box Tests

The simplest type of test is a black box test.  Black box tests are data driven.  We run the executable and treat it as a black box where all internal logic has been hidden from view.  External factors alone determine the success or failure of our tests.  We test against the specifications.  Our tests are input-output driven.

Equivalence Classes

The number of possibilities to be tested in a comprehensive black box test regime is typically too large.  To reduce this number to a manageable set, we introduce equivalence classes.

We create equivalence classes using boundary values.  An equivalence class is a set where testwise any member is as good as any other (for example, i <1, =1...25, >25).

Experts suggest that semantic errors frequently exist at and on boundaries.  We test either side of the boundaries of the equivalence class as well as the boundary itself (for example, i = 0, 1, 2, 17, 24, 25, 26).

We use equivalence classes for both input and output.

White Box Testing

The complementary test to black box tests is a white box test.  White box testing is logic driven.  We treat the program as a glass box with all internal logic visible.  Each white box test is path-oriented.

In white box testing, we execute each possible path through the code at least once.  The number of paths may be too large to test.  To reduce this number and still cover all paths through the code at least once, we prepare flow graphs.

Flow Graphs

A flow graph models the sequences, selections and iterations in the source code.  A flow graph consists of nodes and edges.  Each node represents one or more sequence statements.  Each edge represent the flow of control between two nodes.

Consider the following code.  The flow graph is shown on the right:

|  |  |
| --- | --- |
| // Testing - Flow Graph  // flowGraph.c  #include<stdio.h>  int main(void)  {  int total, value, count;  // Start Node 1 ---  total = 0;  count = 0;  // End Node 1 ---  do {  // Start Node 2 ---  scanf("%d", &value);  // End Node 2 ---  if (value < 0) {  // Start Node 3 ---  total -= value;  count++;  // End Node 3 ---  } else if (value > 0) {  // Start Node 4 ---  total += value;  count++;  // End Node 4 ---  }  // Start Node 5 ---  } while (value != 0);  // End Node 5 ---  if (count > 0)  // Start Node 6 ---  printf("The average value is %.2lf\n",  (double)total/count);  // End Node 6 ---  // Start Node 7 ---  return 0;  } | Icon  Description automatically generated |

Test Criteria

To complete a white box test, we apply the following criteria:

* statement coverage - every elementary statement is executed at least once
* edge coverage - every edge is traversed at least once
* condition coverage - all possible values of the constituents of each compound condition are exercised at least once
* path coverage - all paths from initial node to final node are traversed at least once.
* iteration coverage
  + skip the iteration entirely
  + pass through the iteration once
  + pass through the iteration less than the specified number of times
  + pass through the iteration the specified number of times
  + pass through the iteration once more than the specified number of times
* compound condition coverage
  + break each compound condition into simple conditions