CS 162
Intro to Programming II

Debugging
Programming Errors

• Syntax errors
  – Misuse of C++ language
  – How are they caught?

• Logic errors
  – Doesn’t perform task correctly (aka. bugs)
  – How are they caught?

• Runtime errors
  – Stops your program from running
  – How are they caught?
Syntax Error Examples

• Missing main function
• Use of identifier not declared
• Misspelled Words
• Forget a Semicolon
• Forget Required Keyword
• Missing quote, curly brace, and parenthesis
• Use of single quotes instead of double
Logic Error Examples

• Poorly written programs
  – Add instead of subtract (incorrect operation)
  – Using last two digits for date
  – Same error message for different errors
  – Program that never ends
  – Add one to the largest integer (could be syntax)
Runtime Error Examples

• Open a file that doesn’t exist
• Segmentation fault
  – Infinite loop that eats memory
  – Divide by variable that is zero
Debugging Process

• The key is to localize the problem

• You need to find where the problem starts

• Fix one error at a time, usually starting with the first reported compiler error

• Fix one error at a time
Localizing the Error

• Syntax:
  – **READ compiler errors** (pay attention to line #)
  – Use **google** to search for error

• Logic/Runtime
  – Use **std::cout** to find where the code is breaking
    • Print variable values
    • Print indicator messages
  – **Trace** through the code
  – **Comment** out suspicious code
Error Handling

• What can we do to prevent these errors?
  – Overflow
  – Divide by zero

• Overflow-
  – Check if full before adding anything
  – You can’t put 13 eggs in a carton made for a dozen

• Divide by zero-
  – Check if zero before using
  – Check input and don’t accept a 0
Decomposition

• Helps localize by working on small portions
  – For each new portion of code, you know where the error is!

• Techniques you can use to segment your code:
  – Use stubs as placeholders
  – Use drivers to fill in unwritten sections for testing
Stubs

• After you do decomposition you start with high level structure
• For each future function you write a stub
• Stub- placeholder to demonstrate flow of control
  – May be just an output statement
  – As program is developed it may return ‘sample’ data to test other functions
• Eventually replaced by full function
do {
    cout << "Please enter 1 or 2 or 9 to exit \n";
    cin >> x;
    switch( x ) {
    case 1:
        cout <<"Option one" << endl;
        break;
    case 2:
        cout <<"Option two" << endl;
        break;
    case 9:
        cout <<"Bye Bye!" << endl;
        break;
    default:
        cout << "Please enter a valid number" << endl;
    }
} while(x != 9);
Drivers

• Similar in purpose to the second type of stub
• From the other direction
  – Stub ‘simulates’ internal operation of program
  – Driver provides input and receives output to test the program
  – May do minimal input to get test data from user

For examples see pages 385-387 of the text.
Fundamental Rule for Testing

• Decomposition is important
• Still need for thorough testing!
• Every function should be tested in a program in which every other function in that program has already been fully tested and debugged.