Topic 8 – Testing

Program testing can be used to show the presence of bugs, but never to show their absence.

– Edgar Dijkstra

Testing & Debugging Strategies

Testing & Debugging

- Testing is as important as coding
  - Buggy code is costly
- Testing and debugging go together like peas in a pod:
  - Testing finds errors;
  - debugging localizes and repairs them.
  - Together these form the "testing/debugging cycle": we test, then debug, then repeat.
  - Debug then retest!
    - This avoids (reduces) the introduction of new bugs when debugging.

Overview

- Fixing compiler errors
- Finding runtime errors
- Drivers, stubs and testing systems
- Black box vs. white box testing
  - Types of values

Fixing Compiler Errors

- When you get a compiler error…
  - Try to read the message and think about what it is trying to tell you
  - If it says something like “missing ;”
    - Look at the line above
- If you are getting a “binaries” error or can’t open output file
  - Got to Project → clean
- As a habit you should close your previous projects
  - Right click on the project folder and go to close project

Make Debugging Easier

- Using proper style helps
  - Making your code more readable makes it easier to find your errors.
  - Little things like spaces between operators and operands
    - Such as… << and >> operators for cout and cin
- Make sure you name your variables with names that make sense
- Insert cout statements and output your variables to make sure they have the values you expect
  - You can delete them after you’ve found your problem
- Try to isolate the problem
  - Use ctrl-/ to comment out code and ctrl-/ to uncomment it

Fixing Runtime Errors

- Occur when the program is executing
  - Incorrect output or
  - System crash – etc..
- Harder to fix
- How to avoid them
- Isolating blocks of code

How to avoid them

Code Incrementally!

- Don't try to sit down and write the whole program at once → then debug.
- Write one section at a time and test it.
  - For example:

```
cout << "Enter your annual income: ";
cin  >> income;

cout << "Enter your pay increase rate: ";
cin  >> increaseRate;

cout << "TESTING:  income: " << income;
cout << "\tincrease rate: " <<  increaseRate << endl << endl;
```

Check Dependent Variables

- If one calculation is dependent upon another check the dependent variables.

```
salesTax    = salesTaxRate * retailPrice;
totalPrice = retailPrice     + salesTax;

cout << "TESTING:  sales tax: " << salesTax;
cout << "\tsales tax rate: " << salesTaxRate;
cout << "\t retail price: " << retailPrice;
cout << "\ttotalPrice: " << endl << endl;
```

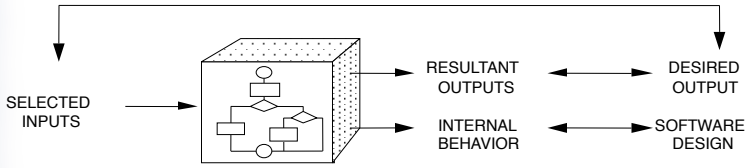
- Testing Loops
  - Check the LCV and each variable at each iteration
  - USE cin.get() ← to hold the cursor
    - This way you trace the code through each iteration

Code Walkthroughs

- Perform a desk check
  - Write down all the variables used in the code segment
    - At this point it is best to walk through every line of code
  - Don't skip steps
  - Don't assume values
    - if there is no value in your variable and you are using it make sure you are initializing it in the code

White Box Testing

- White Box Testing
  - You see the code
  - Test cases based on the structure of the code



White Box Testing

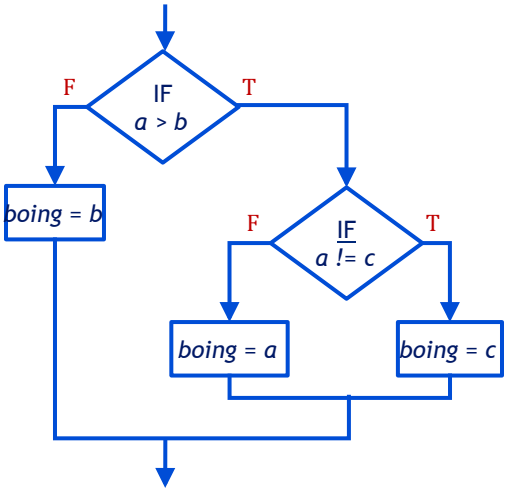
TEST YOUR CONDITIONAL STATEMENTS

- Test the paths of your code
  - Nested statements
- If statements
  - test true
  - test false
- Loops
  - While loops
  - → what happens if first input causes the loop to exit
- Test the boundaries
  - if (hours > 40)
  - Test these values: 40, 39, 41

Testing Selection Statements

- If a,b, & c are integers.
- What do we test?

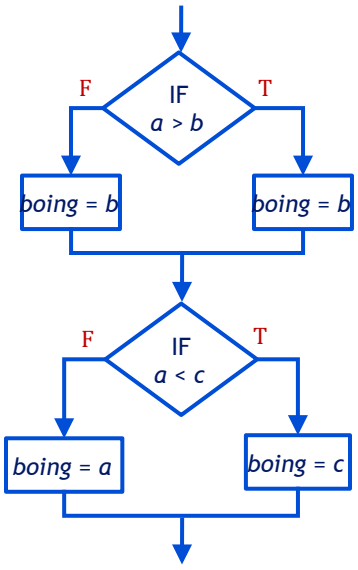
a	b	c
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1
1	1	1
1	1	2
1	2	1
2	1	1
1	2	2
2	1	2
2	2	1



What else should we test?

- If a,b, & c are integers.
- What do we test?

a	b	c
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1
1	1	1
1	1	2
1	2	1
2	1	1
1	2	2
2	1	2
2	2	1



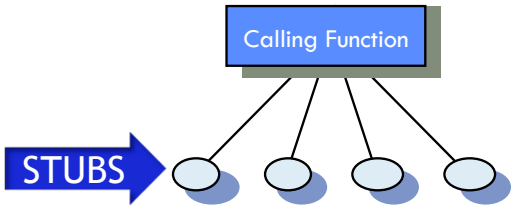
What else should we test?

Testing Stubs and Drivers

- Test stubs
  - Test your Main function to ensure proper values are being sent
- Test drivers
  - Test your functions to ensure proper values are being returned

Test Stubs

- Top-Down Testing Strategy
- Set up dummy functions
  - that you will fill in later
- Just have the parameters set up
  - Test the parameters coming into the function
  - You can add cout statements in the stubs to confirm proper inputs



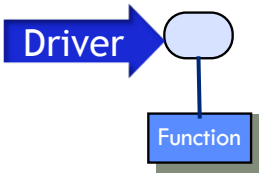
For Example

```
… in main
   AddTwoInts(n1, n2);

int AddTwoInts(int n1, int n2)
{
   cout << "\n In AddTwoInts";
   cout << "\n n1: "<< n1;
   cout << "\n n2: " << n2;
}
```


Test Drivers

- Bottom up testing strategy
- Set up Drivers
  - Pieces of code that test your function
  - Basically they call the function and check the return values



For Example

- … some driver

```
sum = AddTwoInts(4, 5);
cout << "Testing AddTwoInts, n1 = 4, n2 = 5: ";
cout << sum << endl;

sum = AddTwoInts(-3, 25);
cout << "Testing AddTwoInts, n1= -3, n2 = 25: ";
cout << sum << endl;
```

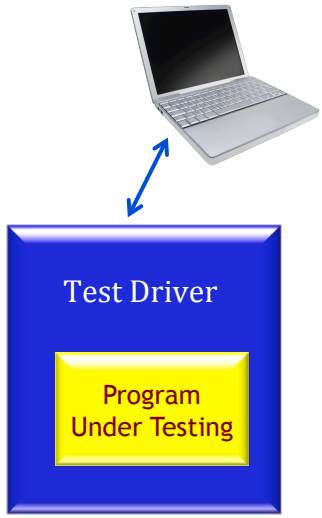
etc...

Testing System

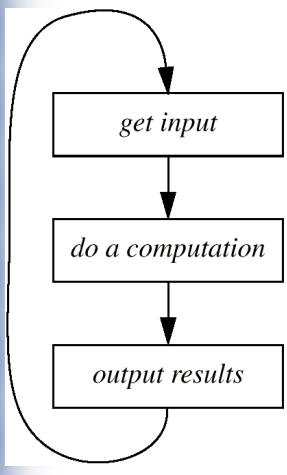
- Test systems allow you to interact with the program (or function) you want to test.
  - The test system will encapsulate the program (or function) to be tested.
- The test system will let you create the version of the program (or function) to be tested and then act as a client (or user) of the program (or function) under testing.
  - The system should provide support for either manual or automated test plan execution.
- There are several industry standard
  - C++ Test, Frameworks such as Boost.Test, CxxTest, GoogleTest, CppUnit, Visual C++, etc

Test System Implementation

- The Test System will be composed of two elements:
  - A Program to be tested.
  - A Test Driver to invoke the Program's functions.
- The Test Driver prompts the user with a menu, gets input from the user, and provides output in the form of counts.



Testing System Implementation



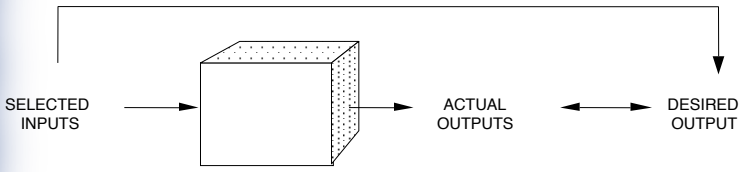
- If we look at what happens when the test is conducted, we see that the following sequence of actions are repeated over and over:
  - Display menu to the user and prompt the user for input.
  - Get the user's input.
  - Perform requested action.
  - Display results.

Use of Testing System

- You will use the testing system during the initial development of your program
  - Find and correct errors
  - Verify compliance with specifications and requirements
- Any time you make future changes in the program
  - Quickly verify that the updated program still comply with specifications and requirements (regression testing)

Black Box Testing

- Black Box Testing
  - You don't see the code
  - Determine the expected outputs
  - Testing based on the specifications for the program ("functional requirements) and the ways that the program will be used ("use cases")



Black Box Testing

- Plans should be written ahead of time
- Testing Inputs and Outputs

- Example,
  - This program calculates how much pocket money a user has left over. The user gets $10 a week for spending money.

  - Inputs:
    - Name, Previous Amount of Money and Amount Spent
    - Name, Pocket Money

Black Box Testing

- Before you run your code
  - Write a **test plan**

- Think about the expected inputs and expected outputs → make a chart

Inputs			Expected Outputs	
Name	Prev Amt	Amt Spent	name	Pocket Money
Jean Cyr	12.50	23.00	Jean Cyr	-0.50
J.R.	23.57	15.00	J.R.	18.57

Test Plan

- A test plan is a document that describes the test cases giving the purpose of each case, the data values to be used, and the expected results
- Test plan design should be carried out concurrently with program development.
- Test plan should include
  - Test cases representing expected inputs and outputs
  - Test cases representing boundaries conditions and exception conditions
  - Clear indication for pass/fail for each test case

Types of Values

- Expected Inputs
- Boundary values
  - Try 0s or if the program have requirements for a selection statement include test for boundary conditions
- Unexpected Inputs

Category	Inputs			Expected Outputs	
	Name	Prev Amt	Amt Spent	Name	Pocket Money
expected	Steve Wu	13.50	23.00	Steve Wu	.50
expected	J.R.	23.57	15.00	J.R.	18.57
unexpected	Jean Cyr	-10.00	23.00	Jean Cyr	-23.00
unexpected	Mary Doe	12.00	-13.00	Mary Do	35.00
boundary	Lisa Covi	0.00	0.00	Lisa Covi	10.00

Summary

- Black Box Testing
  - Testing the functionality of the code
  - Think about what the code should do and write test cases before coding
    - What are the expected inputs and outputs
- White Box Testing
  - Tests the structure of the code
  - Think about your loops and selection statements
    - Do your test cases cover all the statements?
    - Do your test cases follow each branch?
- For all testing have test cases that check…
  - Expected inputs
  - Boundary values
  - Unexpected inputs

Summary

- Black Box Testing
  - Testing the functionality of the code
  - Think about what the code should do and write test cases before coding
    - What are the expected inputs and outputs
- White Box Testing
  - Tests the structure of the code
  - Think about your loops and selection statements
    - Do your test cases cover all the statements?
    - Do your test cases follow each branch?
- For all testing have test cases that check…
  - Expected inputs
  - Boundary values
  - Unexpected inputs

Debugging

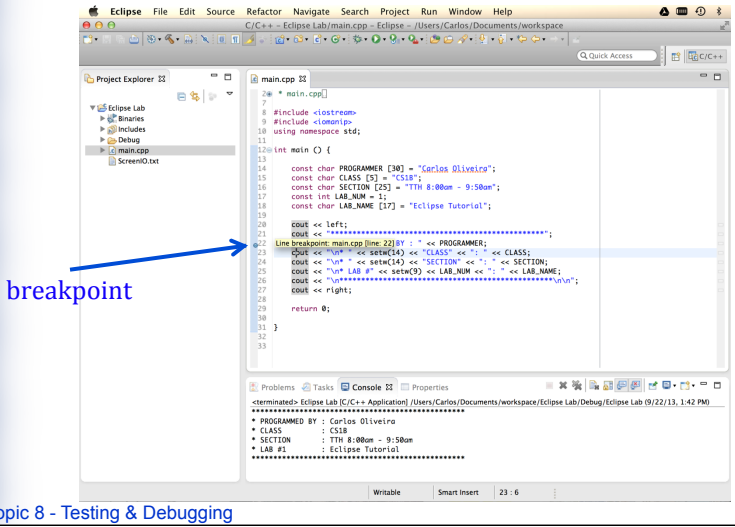
- Print Statements
  - Advantages
    - Easy to add
    - Provide information
    - Track values as you go at strategic points
  - Disadvantages
    - Not always practical
    - Can cause errors
    - Removing can be tedious
- Debugger can help too
  - Can step through code
  - Can add breakponits

Eclipse Debugger

- Eclipse includes a built in debugger supporting multiple advanced debugging capabilities
- Using the debugger you can:
  - Execute your program step by step
    - step over or step into functions
  - Review value of each variable
  - Add execution breakpoints
    - forces the program to stop at specific code line
  - Track error conditions and expressions

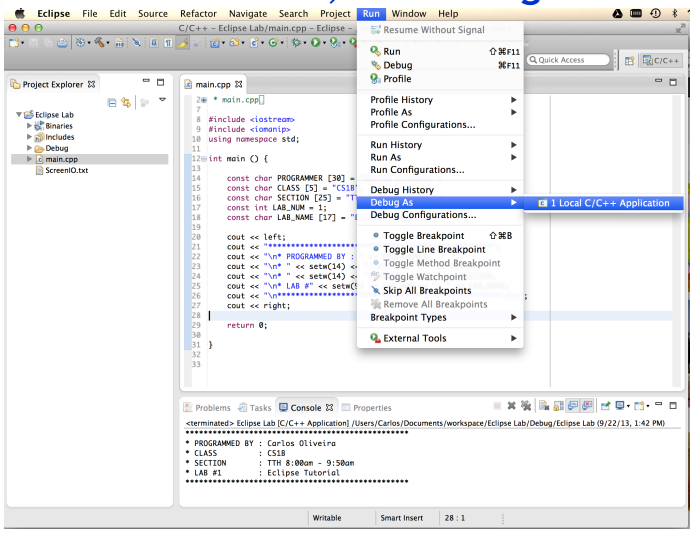
Adding Breakpoints

- Double-click the code line number to add or remove a breakpoint



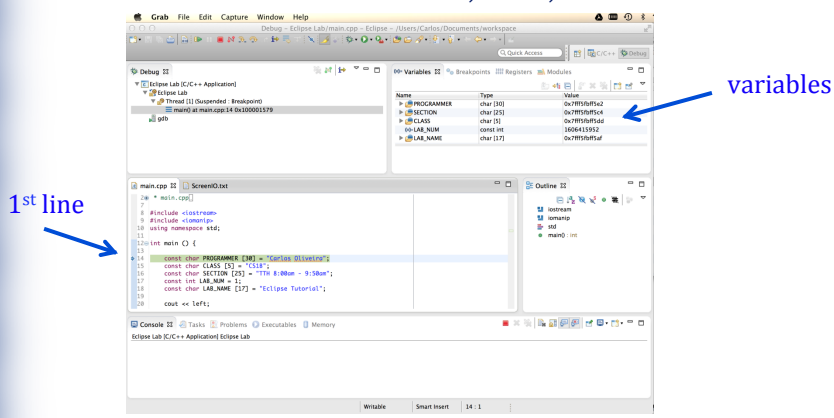
Starting the Debugger

- Instead of **Run As**; use **Debug As**



Running the Debugger

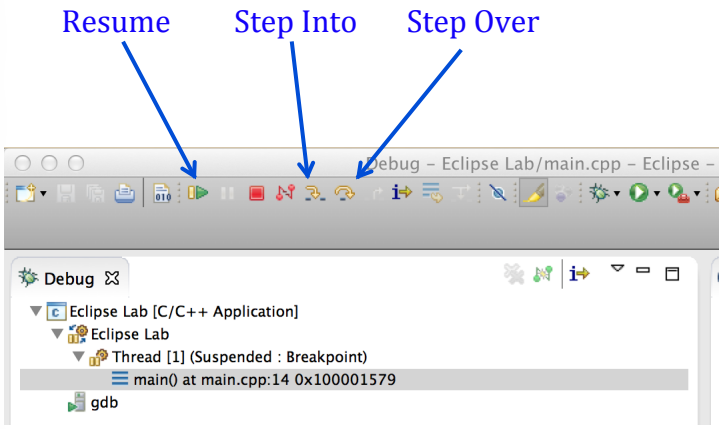
- Once you start the debugger; the debugger will point to the beginning of your program and wait for your command to start executing the program
  - You can examine the variables, stack, etc.



Options for Program Execution

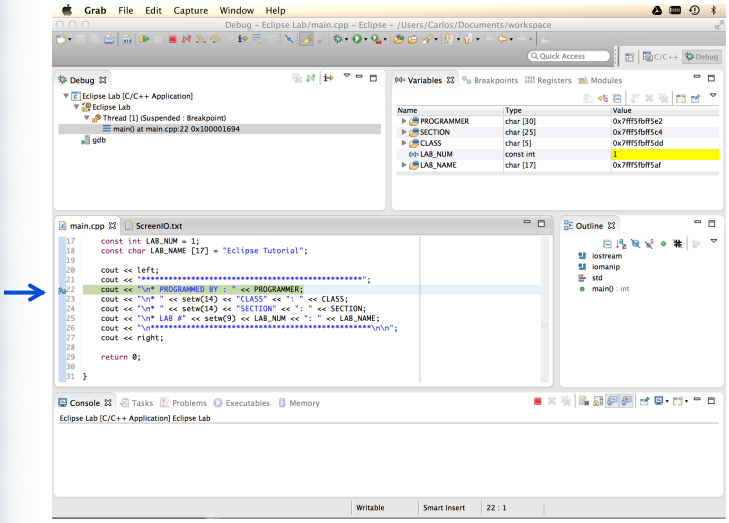
- Once the program is running in the debugger you have different options to execute your program
  1. Using **Resume (F8)** your program will run continuous until find the next breakpoint or terminate
  2. Using **Step Into (F5)** your program will run one statement (line) at the time and the debugger will follow "into" functions part of the statement
  3. Using **Step Over (F6)** your program will run one statement (line) at the time and the debugger will continuous execute any function part of the statement

Option Keys for Execution



Resuming Execution

- Running to the next breaking point - line 22



Final tips

- Check your algorithm first
- Code incrementally
  - Write a function
  - Test and debug
- Divide and conquer (simplify the problem)
- Explain the bug to someone else (or a duck)
- Fix bugs as you find them
  - Fix from top to bottom
- Try to isolate the problem
- Track your common mistakes