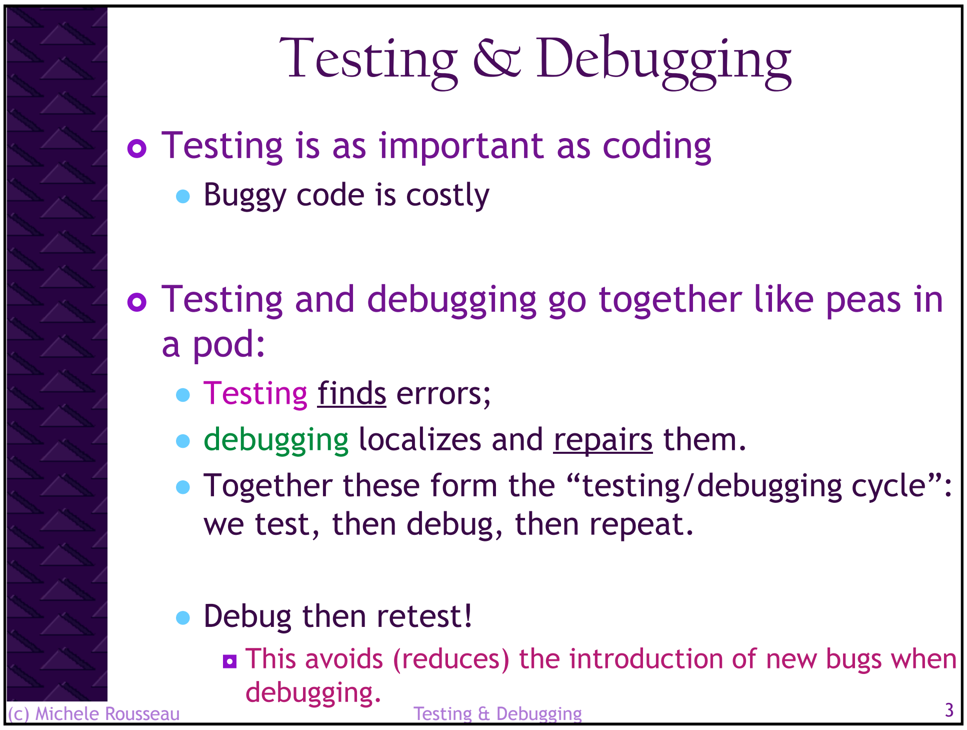
Testing & Debugging

CS1A

- Fixing Compiler Errors
- Fixing Run Time Errors
- Desk Checks
- Creating a Test Plan
  - Black Box & White Box Testing
  - Categories of test values

Testing & Debugging Strategies

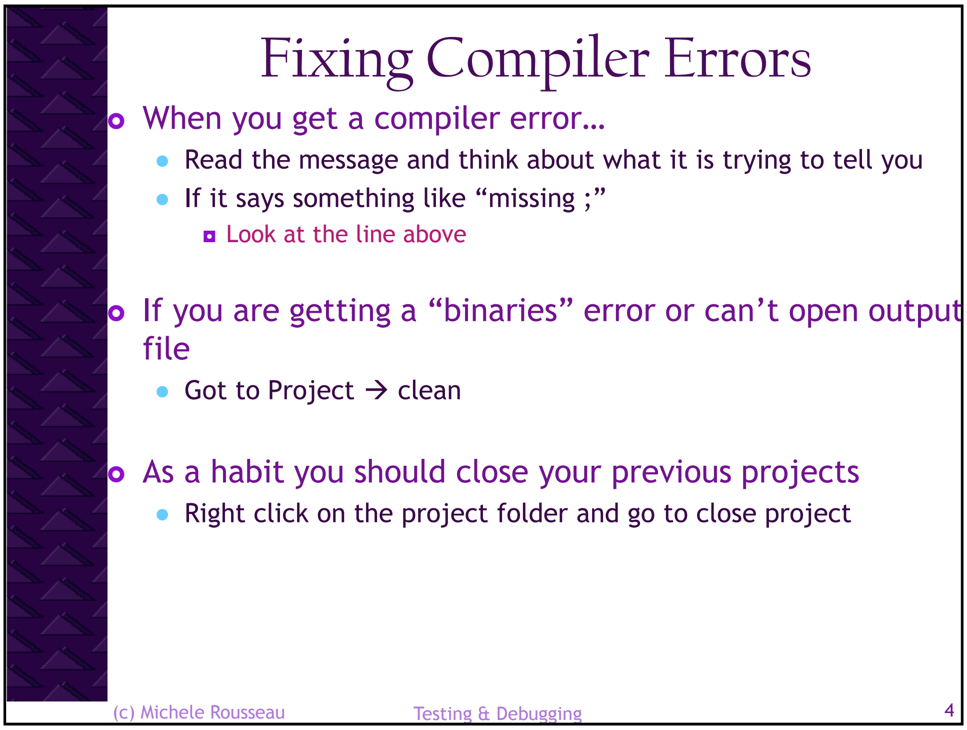
Image: Courtesy of © Christine Jopling



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.

(c) Michele Rousseau Testing & Debugging 3



Fixing Compiler Errors

- When you get a compiler error…
  - 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

(c) Michele Rousseau Testing & Debugging 4

Make Debugging Easier

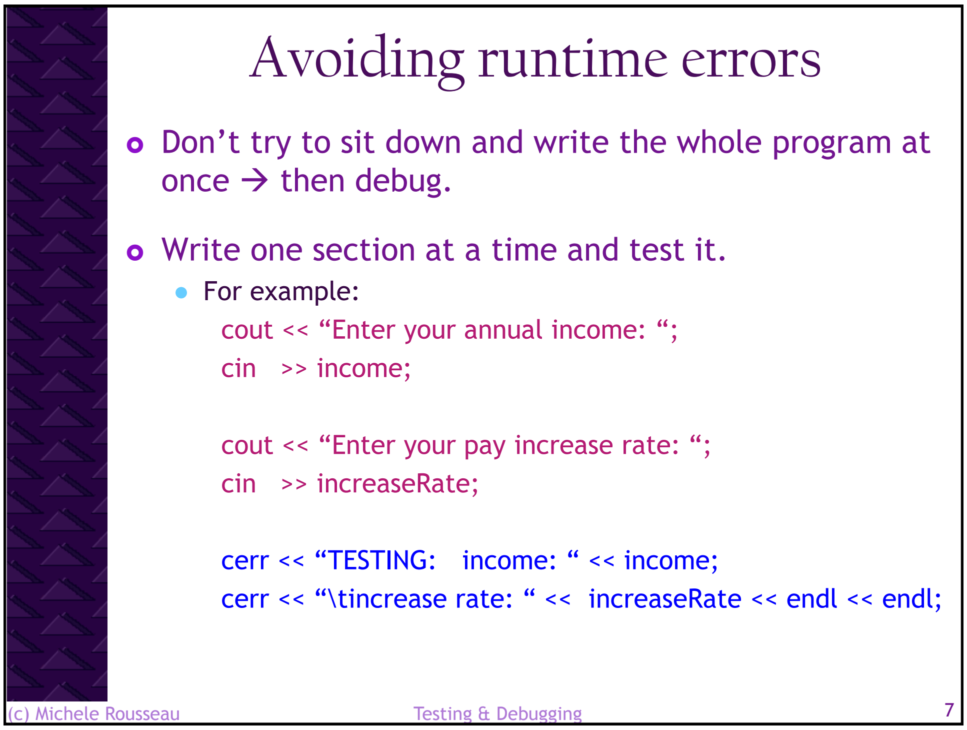
- Using proper style helps
  - Making your code more readable makes it easier to find your errors.
  - Little things like spaces between operators and operands
    - Such as… << and >> operators for cout and cin
- Make sure you name your variables with names that make sense

Fixing Runtime errors

Runtime errors are tricky because the compiler isn't telling you where the error is or what is causing the problem.

The key to fixing runtime problems is first to figure out which part of the code is causing you issues.

- Try to isolate the problem
  - Use ctrl-/ to comment out code and ctrl-/ to uncomment it
- Insert cerr statements and output your variables to make sure they have the values you expect
  - You can delete them after you've found your problem
  - cerr works similarly to cout – except it should flush the output buffer and output in red
- Desk check you code!



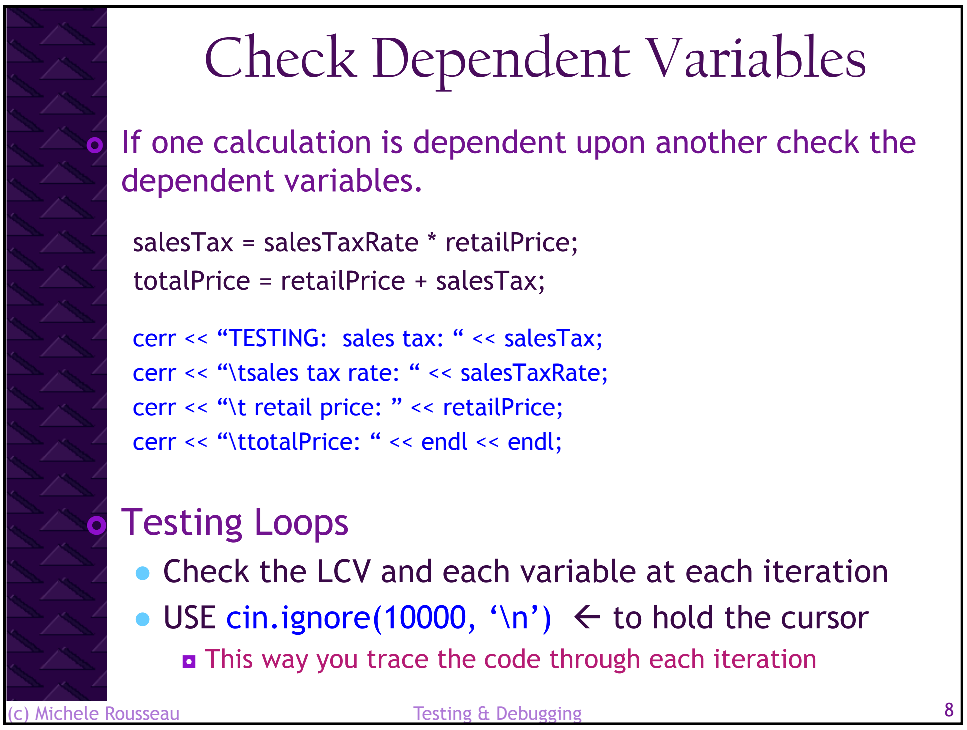
Avoiding runtime errors

- 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;

cerr << “TESTING:   income: “ << income;
cerr << “\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;

cerr << “TESTING:  sales tax: “ << salesTax;
cerr << “\tsales tax rate: “ << salesTaxRate;
cerr << “\t retail price: ” << retailPrice;
cerr << “\ttotalPrice: “ << endl << endl;
```

- Testing Loops
  - Check the LCV and each variable at each iteration
  - USE cin.ignore(10000, ‘\n’) ← 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

Black Box Testing

- Black box testing
  - You don't see the code
  - Test based on the specifications of the program
  - The purpose is to test the functionality of the code
  - Determine the test cases and expected output
  - Compare the actual output to the expected output

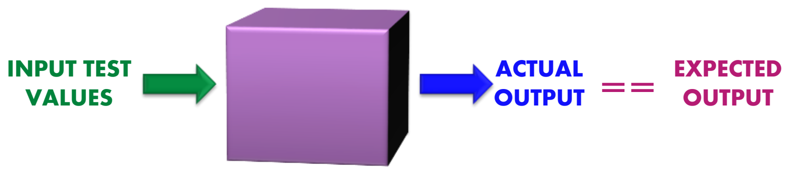


Image: © Michele Rousseau

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 $20 a week for spending money.
  - Inputs:
    - Name, previous amount of money and Amount Spent
    - Name, Pocket Money

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	prevAmt	amtSpent	Name	Pocket Money
Jean Cyr	12.50	23.00	Jean Cyr	9.50
J.R.	23.57	15.00	J.R.	11.43

Also think about boundary tests

- Test expected input
- Test boundary values
  - Try 0s o
  - if you have a selection statement
    - Calculate overtime if hoursworked is > 40
      - ➔ Test these values: 40, 39, 41
- Test for unexpected inputs

Inputs			Expected Outputs	
Name	prevAmt	amtSpent	Name	Pocket Money
Jean Cyr	-10.00	23.00	Jean Cyr	-13.00

White Box Testing

- White Box Testing
  - Look at the code
  - Define you test cases based on the code
  - Testing the Structure of the code
    - Ensure that it meets the design

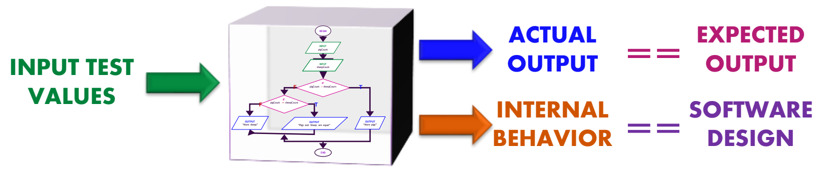
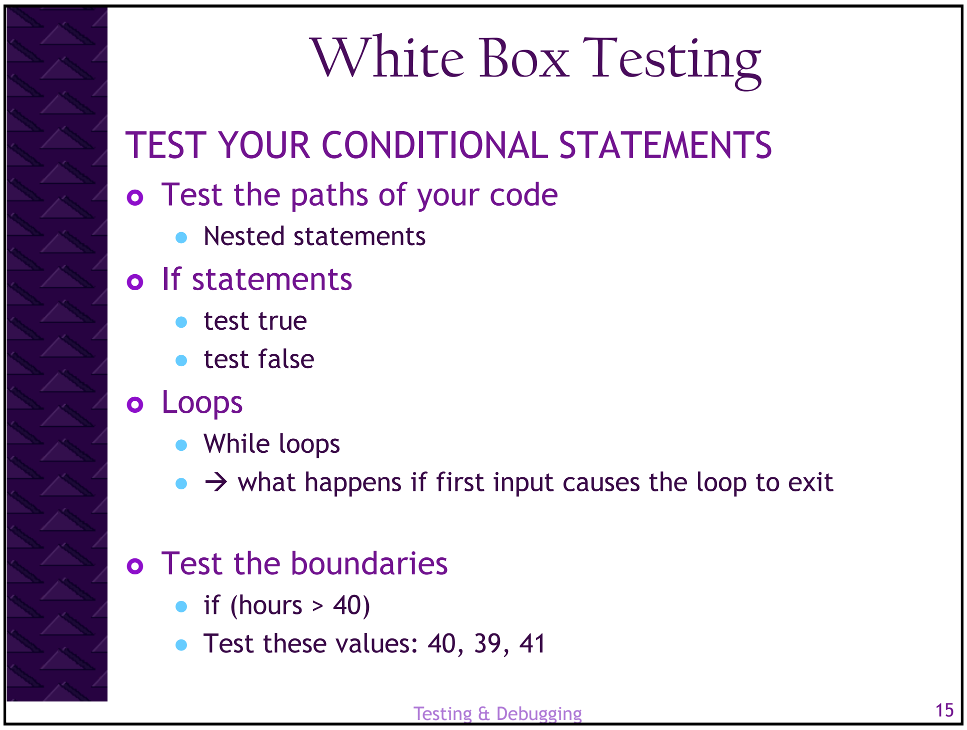


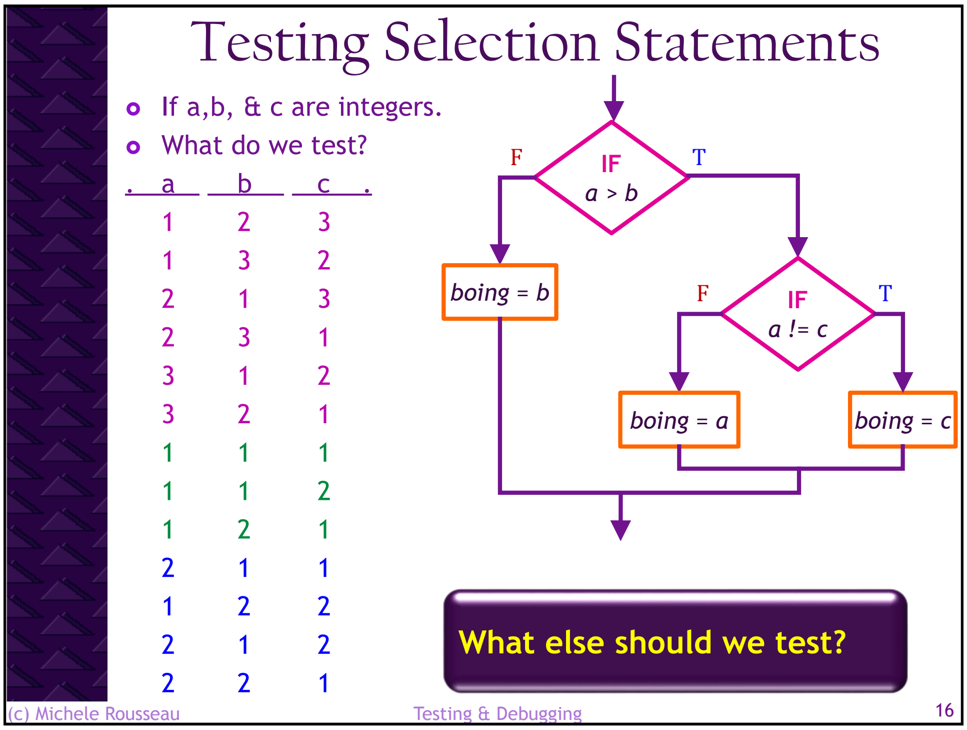
Image: © Michele Rousseau



White Box Testing

TEST YOUR CONDITIONAL STATEMENTS

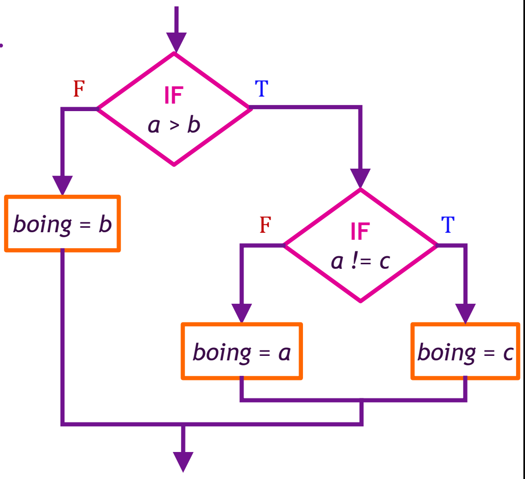
- Test the paths of your code
  - Nested statements
- If statements
  - test true
  - test false
- Loops
  - While loops
  - → what happens if first input causes the loop to exit
- Test the boundaries
  - if (hours > 40)
  - Test these values: 40, 39, 41



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

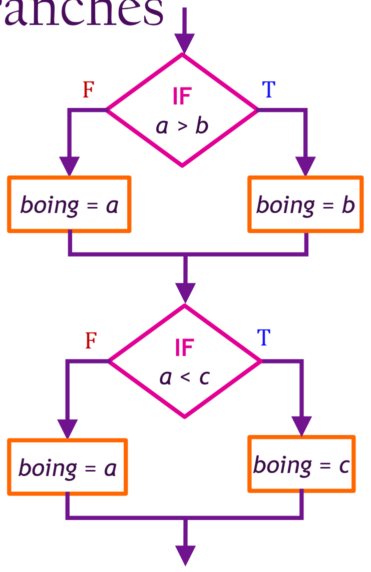


What else should we test?

Testing the Branches

- 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?

Summary

- Black Box Testing
  - Tests 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 Values