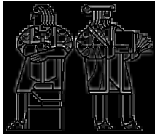


6.170 Lecture 3 Debugging



John Guttag
MIT EECS

Does My Program Work, Why Not

Validation (start in a couple of weeks)

Process designed to uncover problems, increase confidence
Combination of reasoning and test

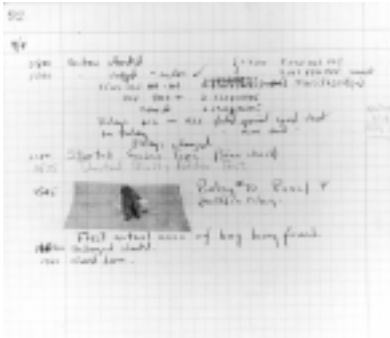
Debugging (Section 9.9 of text)

Ascertaining why a program is not functioning as intended
Function
Performance

Defensive programming (woven throughout term)

Programming to abet validation and debugging
E.g., exploiting compile-time type checking

A Bug: September 9, 1947



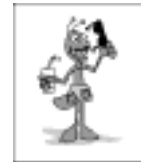
Some Myths About Bugs

Myth: Bugs crawl unbidden into programs

Fact: If there is a bug in your program, you put it there
I.e., you made a mistake

Myth: Bugs breed in programs

Fact: If there are many bugs, you put each of them there
I.e., you made many mistakes



A Matter of Attitude

When you find a bug in your code, feel relieved

Don't feel proud
Be at least a little embarrassed

When you run a program for the first time

Expect it to work
Debugging usually more time consuming than getting it
right in the first place

Debugging

Experience is clearly not the best teacher

Goal is not to eliminate one bug quickly

Goal is to move quickly towards a bug-free program

Key is to be systematic

When testing exposes a problem

Don't run off to the debugger

Study the program text

How could it have produced the result it did?

Is this part of a family of bugs?

How can it be fixed?

Using the debugger is a desperate measure

6.170 **When You Use Debugger**

Decide what you want to accomplish
Decide how you will accomplish it
Know what inputs you plan to use
Otherwise you will waste gobs of time
Know what outputs you expect
Otherwise you will be fooled

✱

© John Chapin/John Guttag Spring 2000 Slide 7

6.170 **The Scientific Method**

Study available data ✱
Test results
Program text
Keeping in mind that you don't understand it
Form a hypothesis consistent with all the data
Design and run a repeatable experiment ✱
With potential to refute hypothesis
Often with useful intermediate results
✱ With expected values
Repeatability often a challenge
Multi-threaded programs
Interactive programs

© John Chapin/John Guttag Spring 2000 Slide 8

6.170 **Designing the Experiment**

✱
Find simplest input that will provoke bug
Usually not the input that revealed existence of bug
Start with data that revealed bug
Keep paring it down
Often leads directly to bug

✱

© John Chapin/John Guttag Spring 2000 Slide 9

6.170 **When You Have a Good Data Set**

Search for bug using simple data
Decide what intermediate states to look at ✱
Predict intermediate values
If necessary, write code to display states
Don't throw this away after finding bug
Don't expect of find bug first time ✱
Consider making check points ✱
Especially when runs take a long time ✱

✱

© John Chapin/John Guttag Spring 2000 Slide 10

6.170 **Some Things to Keep in Mind**

The bug is not where you think it is ✱ ✱
Ask yourself where it cannot be ✱ ✱
Explain why
Try simple things first, e.g., ✱
Reversed order of arguments
✱ Spelling of identifiers
✱ Failure to reinitialize a variable
Same object vs. equal
Deep vs. shallow copy
Make sure that you have correct source code
Recompile everything ✱ ✱
Keep a record of what you've tried ✱

© John Chapin/John Guttag Spring 2000 Slide 11

6.170 **When the Going Gets Tough**

Try binary search ✱
When it gets really tough, get help ✱
We all develop blind spots
There may be something you don't understand
Explaining the problem often helps ✱
Reconsider assumptions
E.g., has the OS changed ✱ ✱
Start working on documentation
Way to approach things from a different angle ✱
Walk away ✱ ✱
Trade latency for efficiency
One reason to start early

© John Chapin/John Guttag Spring 2000 Slide 12

6.170 **When You Find A [sic] Bug**

Ask how it got there

- Careless error
- Misunderstanding
- Language
- Requirements
- Environment
- In attempt to fix something else
- Design vs. coding

Ask if the bug

- Is symptomatic of a larger problem
- Has relatives

© John Chapin/John Guttag Spring 2000 Slide 13