Introduction to Debugging

© Jeff Parker, 2000
Updated March 2001
Goals

This lecture gives some tips for debugging your programs

- We discuss the steps your program goes through
- We discuss the debugging process
- We review some useful features of modern source-level debuggers
- We review some ideas you can use when you don't have access to a source level debugger

No rocket science, but well accepted ideas
Motivation

While we would like to write perfect programs, few of us can

We are imperfect

We inherit code, and our co-workers (instructors) (TFs) are imperfect

We use code for purposes it was never intended (code rot)

There is more to debugging than knowing how to use the debugger.

Debugging is a very useful and transferable art. You will get lots of practice this semester

This lecture will only serve as an introduction
Outline

Types of Bugs
  Compile time errors
  Link Time errors
  Run Time errors
  Logic Errors

Modes of Debugging
  Bench checking
  Printing
    Types of "if (debug)"

Assertions

Into to Java System - How to create a project

Tour of symbolic Debugger
  Anatomy
  Actions
Types of Errors

Compile time errors

```java
void example() {
    System.out.println('Hello, world!');
}
```
Error: Invalid character constant: line 63

Link Errors

"Link Error: cannot find module C0WS.OBJ"
"Class listClass not found in type declaration"
"Method DisplayList(listClass) not found"

Runtime errors

Logical Errors
Runtime Errors

Errors caught by the OS or runtime system

Example: divide by 0

```java
int one = 1; int two = 0; int three = one/two;
```

`java.lang.ArithmeticException: Division by zero. at BinarySearch.main (BinarySearch.java:12:0x7)`

Example: Array bounds excess

```java
int list[] = new int[10]; int pos = 11; list[pos] = 2;
```

`java.lang.ArrayIndexOutOfBoundsException: Index value 11 is out of array bounds 0 and (length - 1) 9 at BinarySearch.main (BinarySearch.java:8:0xc)`

Two common runtime errors that need explanation

Segment Violation - attempt to store to or read from memory you don't own

Bus error - attempt to fetch data from inappropriate address: such as loading long from odd byte address.
Logic Errors

The program compiles, links, and runs to completion, but the output is wrong, or it causes a run-time error.

These are the hardest: the language and runtime system cannot help.

Here are some ideas that can help find logic errors, and some to help prevent them.
How to deal with Bugs

Observe them

Note any unexpected behavior
Be honest with yourself - catch all odd behaviors
To fix bugs we must be able to reproduce them

Know what happens (When I enter '12', it prints 's' and dies)

Locate them - find the error in your logic

Fix them - possible once we have found them

Remember them

There are few new mistakes: most of us have our favorite errors.
Keep track of your bugs and their cure in a notebook. You may find that reviewing these notes helps you find your current problem

Prevent them

We'll review some techniques
Observe them

Reproducibility: If you don't know how to reproduce a bug, and you don't remember what it did when you hit it, you will have a very hard time fixing the problem.

In class, you will be responsible for finding your bugs. It is a good idea to keep careful notes.

You may wish to take an idea from the Real World: the Software Quality Assurance (SQA) group finds bugs and designs tests so that:

- The program with bug fails on the test
- After a fix, the program should work on test
- Test remains in a testbed in case the bug returns
Locate Them

Once you can reproduce the bug, try tweaking the parameters
   It fails when you enter "12", how about "11"? "13"? "21"?

Try to figure out where the happens in the source code
   Is the table valid before you read the file?
   Is if valid after reading the file, but before sorting the data?
   When you have narrowed the search down to a general area, your job
      is much simpler

The symbolic debugger is an excellent microscope. Don't use
it to track down a herd of rhinos: wait until you know
where they are hiding to deploy it
Defensive Behavior

Rather than throwing a program together, consider planning the parts, and testing each component as it is complete.

Good decomposition makes finding errors simpler

Test each interface

Every data structure needs a "display" method

When you read data, from a file or from the user, take the time to echo it back.

Use scaffolding: tear it down when done

Echo the data structure after every change
Bench Checking

Once you have narrowed the problem down to a small section, it is possible to bench check the code ("play the computer") and see what it does. You need to drop preconceptions, and read what you have written - often very difficult.

To jump the gap between what we wanted to say and what we have written, we need feedback:

- print statements
- the debugger
The mighty print statement

The print statement is a powerful low-tech, tool. Can be used even when no symbolic debugger is around

When I write a program, I insert print statements to verify that I have reached a key step, to verify that I have stored the right data, etc.

Many of these statements are scaffolding that will be removed

Some are statements that may stay in some dormant form

```java
final boolean DEBUG = false;
if (DEBUG ) System.out.println("sort");

final int DEBUG_LEVEL = 2;
if (DEBUG_LEVEL >= 3) System.out.println("search")

if (debugOn(level, subsystem)) System.out.println("parse")
```
Assertions

Use prints with discretion: if you have too many, you cannot see the trees for the forest.

Pepper your program with **assertions** - things that you think should be true.

```java
if (size < 1) System.out.println("Warning: size = " + size);
```

Assertions only appear when needed

C and C++ provide a useful facility: the assert macro.

```c
#include <assert.h>
assert(size >= 1)
```

If assertion is false, program prints filename and line #, quits
Finding Your Errors

By validating the data, and printing status, you can narrow down where your problem is.

Very hard to do in non-deterministic situations: rare in this course, all too common in practice.

When you have located the problem, use bench checking to see what is going on.

Keep going until you and the machine differ: focus on where that difference comes from.

The Debugger can help, once you know where to focus the microscope.
Introduction to IDEs

To write a program you must first create and name a project
Select directory
Create project, name it
Select application or applet
Add files

Four major workspaces
Editing
Browsing
Debugging
Output
Sample Editing Environment

The following screen snaps are from MetroWerks v4.0 running on a Macintosh. Your mileage may vary, but your IDE will have similar functions.
Sample Class Browser

Three classes are in scope. We are looking at the class Table

Clicking on function name bring up the function
You can edit in the lower pane
Hierarchy Browser

We won't write projects with complicated class hierarchies. If we did, the following window would be useful.
Debugger Window

windows

Stack: the main program has called function find
Variables: the variables high, low, mid, and s are in scope
Source: the code we are running
Output Window

Output appears in the Console window

```
Insert one
Table
0 one

Find one 1
Insert four

Table
0 four
1 one

Find four 1
Insert two

Table
0 four
1 one
2 two
```
Project Window

You run your project from the Project Window.
This project has 7 source files
Anatomy of Interactive Debugger

Windows
- Call Stack
- Variables
- Source Code
- Breakpoints
  - Red Dot

How to set a breakpoint
- Select a line (dash)
- Click to set (or clear) breakpoint

```
int find(String s)
{
    int low = 0;
    int mid;
    int high = items.size() - 1;
    while (low <= high)
    {
```

```
Debugger Actions

Debuggers provide the following functionality: Metrowerks uses buttons similar to a CD player's

• **Run (to Breakpoint)** - Go until you hit a breakpoint or the end
• **Stop the program** - Stop and allow more actions
• **Kill the program** (Stop and quit debugging)

Next three are Step buttons - Perform the next line of code

• **Step In** - If next line is function call, dive into the function
• **Step Over** - Treat the next line as an indivisible step
• **Step Out** - Exit the current function
Example

```java
int find(String s) {
    int low = 0; int mid; int high = items.size() - 1;
    while (low <= high) {
        mid = (low + high)/2;
        if (0 == s.compareTo((String) items.elementAt(mid)))
            return mid + 1;
        else if (0 < s.compareTo((String) items.elementAt(mid)))
            low = mid;
        else
            high = mid;
    }
    return 0;
}
```
From the console window, we notice that we were able to add the string two, but not to find it. We suspect function find.

If we set a breakpoint in the main after inserting two, but before finding it, we would stop there three times.
Run to breakpoint

After pressing the "run" button, the program runs until the breakpoint. Blue arrow shows where the program is. Note the breakpoint is "before" the source that shares it's line.
Add a new breakpoint

Now we are ready to figure out what happens in the third call to find. Scroll up in the source window to the find function, and place a breakpoint at the start of the function.
Then hit the "run" button again
Variables

We will step through the execution of the function

We can watch the effects of the program on local variables through the Variables window. Some variables are primitives: others are more complex. "this" is a table, which holds a vector called "items". We are looking at the contents of items: there are three elements in the vector. We will be most interested in the values of high, low, and mid.
We single step through the function using the fourth button. The value of **mid** is red, as it has just been updated.
Single Step shows logic in action

It is easy to tell what the program is doing.
We have taken the second branch of the three way comparison.

```java
/*
 * Search for the string in the table held in array "items"
 * Return 0 if it is not there: 1 if it is in
 * the first slot, etc
 */
int find(String s)
{
    int low = 0;
    int mid;
    int high = items.size() - 1;
    while (low <= high)
    {
        mid = (low + high)/2;
        if (0 == s.compareTo((String)items.elementAt(mid)))
            return mid + 1;
        else if (0 < s.compareTo((String)items.elementAt(mid)))
            low = mid;
    }
    return -1;
}
```

When single stepping, you will often see an odd "step" as the blue arrow touches the end of a loop before each new iteration. This is a visualization of the act of testing the loop condition, and is normal.
Watch logic in action

After stepping through a few times, and noticing that the value of mid is stuck at 1, it is clear that there is a problem updating mid. The debugging can help you find your problems: you must fix them.
Using Metrowerks

Find the software

Science Center PCs - StartMenu/Programs/Programming
Science Center Macs - HardDrive/Programming Folder
Church Street PCs - StartMenu/CodeWarrior
Church Street Macs - Launcher/Programming

Running Project

Build
Run
Using Metrowerks Project Window

File/New Project (Project stationary pops up)
Java/Java Application - Name project
File/New - creates new file.
Edit document (or paste a starting example).
Save file. Note that the file name must match the name of the main class (e.g. WarmUp.java)
   Edit/JavaApplication Settings
   Target/Target Settings - set target name
Target/Java Target
   Main Class (Class to start program - e.g. WarmUp)
   Parameters (Contents of args - Command line input)
Metrowerks Debugger

Project/Enable Debugger - left column appears. Click col to set/clear breakpoints
Project/Run becomes Project/Debug - Debugger window pops up

Debugger sub-windows

Variables - select triangle for arrays to open choices
    Double Click on a variable to pop up Data Browser on it
    Double click on a field in an object and open new browser

Stack - Which functions are we in? (Exec::run/WarmUp::main)

Source - Can set/remove breakpoints
    {} - select function/Line - where you are

Icons - Run, Stop, Kill, Step Over, Step Into, Step Out

Debug/view Memory - dump contents of memory. Allows low-level view

Window Menu- can pop up some views

    Browser Catalog Window
    Class Hierarchy Window
    New Class Browser - Can see member functions, etc.
        May wish to pop up one for each class
Using Symantec Cafe

Find the software
   GeneralSoftware/Programming/Symantec Cafe 1.2/Semantec Cafe
Create a project
   Project/New
      Select Directory
      Project Express
      Name it
      Project type (application or applet)
      Add Files
Running Project
   Build
   Run
Symantec Debugger Windows

Window choices
- source
- project
- calls - procedure stack
- data - variables
- thread
- breakpoints - set, clear - click in source
- output
- class Editor
- Hierarch Editor

Java Language
- Class and Interface definitions
Gotchas

Applets need html files (See 8 Queens example)
Applets cannot read files
    To read a file, create an application
The Application runner may not pick up changes to the source code.
    Exit and restart it
Java applications are setup to use arguments from a command line. Take
the time to figure out where these come from in your IDE.
    Roaster uses preferences menu
    Semantec Cafe uses Project/arguments
PC: Put class names in lower case - respect DOS 8.3 naming conventions
Class Path - "Error: Package java.io not found in import" set the class path
    Edit/JavaApplicationSettings/Target/Access Paths
Suggested Reading

There are few books that discuss debugging in detail. Oualline's "Practical C" devotes most of chapter 14 to debugging, though he describes the Free Software Foundation's gbx (part of the GNU project: Gnu is Not Unix). Ideas are the same.

The best guide to your debugger is your system's documentation. Take the time to read through any tutorials or examples before you need it.

Time spent learning to use this tool is a wise investment.