Chapter 15
Debugging

Original slides from Gregory Byrd, North Carolina State University
Modified slides by Chris Wilcox, Colorado State University

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Debugging with High Level Languages

- Same goals as low-level debugging
  - Examine and set values in memory
  - Execute portions of program
  - Stop execution when (and where) desired

- Want debugging tools to operate on high-level language constructs
  - Examine and set variables, not memory locations
  - Trace and set breakpoints on statements and function calls, not instructions
  - ...but also want access to low-level tools when needed

Types of Errors

- **Syntactic Errors**
  - Input code is not legal
  - Caught by compiler (or other translation mechanism)

- **Semantic Errors**
  - Legal code, but not what programmer intended
  - Not caught by compiler, because syntax is correct

- **Algorithmic Errors**
  - Problem with the logic of the program
  - Program does what programmer intended, but it doesn't solve the right problem

Common errors:

- missing semicolon or brace
- mis-spelled type in declaration

One mistake can cause an avalanche of errors

```c
main () {
    int i, int j;
    for (i = 0; i <= 10; i++) {
        j = i * 7;
        printf("%d x 7 = %d
", i, j);
    }
}
```

missing semicolon
Semantic Errors

- Common Errors
  - Missing braces to group statements together
  - Confusing assignment with equality
  - Wrong assumptions about precedence/associativity
  - Wrong limits on for-loop counter
  - Uninitialized variables

```c
main () {
  int i, j;
  for (i = 0; i <= 10; i++)
    j = i * 7;
  printf("%d x 7 = %d
", i, j);
}
```

Algorithmic Errors

- Design is wrong, so program does not solve the correct problem
- Difficult to find
  - Program does what we intended
  - Problem might not show up until after many runs
- Maybe difficult to fix
  - May have to redesign
  - May have large impact on program code
- Classic example: Y2K bug
  - Only allow 2 digits for year, assuming 19__

Debugging Techniques

- **Ad-Hoc**
  - Insert printf statements to track control flow and display values
  - Add code to explicitly check for values out of expected range, incorrect branches, etc.
  - Advantage:
    - No special debugging tools needed
  - Disadvantages:
    - Frequent recompile and execute cycles makes this method time-consuming
    - Requires intimate knowledge of code
    - Inserted code can be buggy

Source-Level Debugger

- **Source-Level Debugger**
  - Examine and set variable values
  - Tracing, breakpoints, single-stepping on source-code statements

Main window of Cygwin version of gdb.
### Source-Level Debugging Techniques

- **Breakpoints**
  - Stop when a particular statement is reached
  - Stop at entry or exit of a function
  - **Conditional breakpoints:** Stop if a variable is equal to a specific value, etc.
  - **Watchpoints:** Stop when a variable is set to a specific value

- **Single-Stepping**
  - Execute one statement at a time
  - Step "into" or step "over" function calls
  - **Step into:** next statement is first inside function call
  - **Step over:** execute function without stopping
  - **Step out:** finish executing function, stop on exit

### Displaying Values

- Show value consistent with declared type of variable
- Dereference pointers (variables that hold addresses)
  - See Chapter 16
- Inspect parts of a data structure
  - See Chapters 19