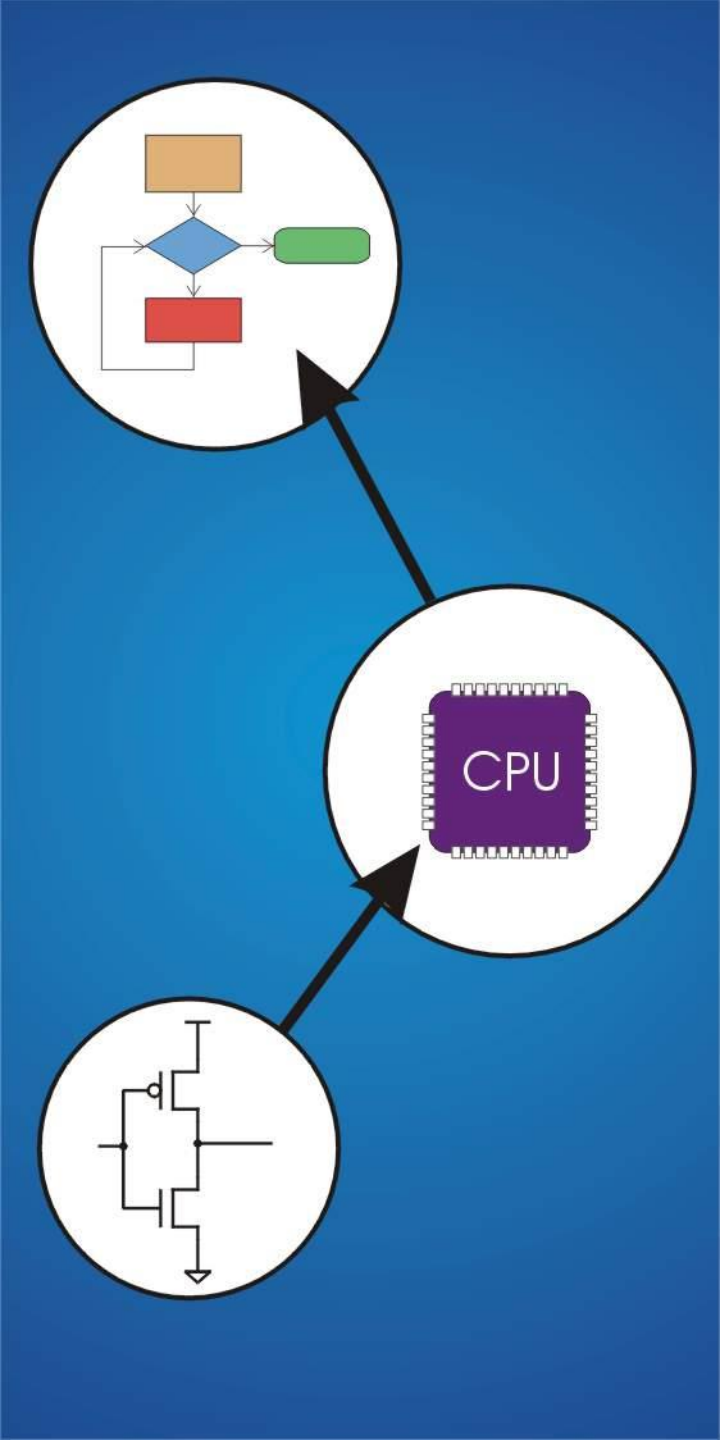


Chapter 15
Debugging

Known, but unfixed errors

"Just ignore errors at this point. There is nothing we can do except to try to keep going."

-A comment in XFS (xfs_vnodeops.c, line 1785)

"Error, skip block and hope for the best."

-A comment in ext3 (namei.c, line 880)

"Note: todo: log error handler"

-A comment in IBM JFS (jfs_logmgr.c, line 222)

-- Excerpt from Prof. Haryadi Gunawi's PhD thesis

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

Syntactic Errors

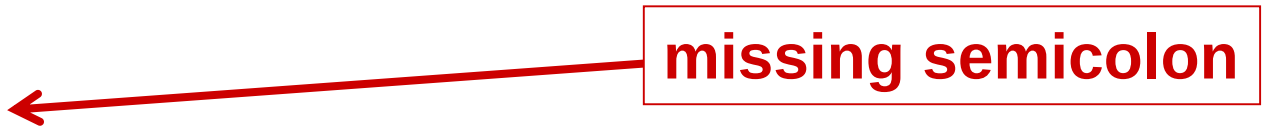
Common errors:

- missing semicolon or brace
- missing variable/function declarations

One mistake can cause an avalanche of errors

- because compiler can't recover and gets confused

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



Semantic Errors

- The real problem starts once the syntax errors are fixed
- Common Errors
  - Missing braces to group statements together
  - Confusing assignment with equality
  - Wrong assumptions about operator precedence, associativity
  - Wrong limits on for-loop counter
  - Uninitialized variables

```
main () {
  int i
  int j;
  for (i = 0; i <= 10; i++)
    j = i * 7;
    printf("%d x 7 = %d\n", 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 many runs of program**

Maybe difficult to fix

- **Have to redesign, may have large impact on program code**

Classic example: Y2K bug

- **only allow 2 digits for year, assuming 19__**

Testing

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

-- Edsger W. Dijkstra

and you need to accept that an exhaustive testing is not possible…

Black-Box Testing

- **assumes nothing about the internals of the program**
- **replies upon input/output specification**
- **is sometimes automated**

White-Box Testing

- **supplements black-box testing**
- **increases test coverage**
- **often uses an assertion**

Debugging Techniques

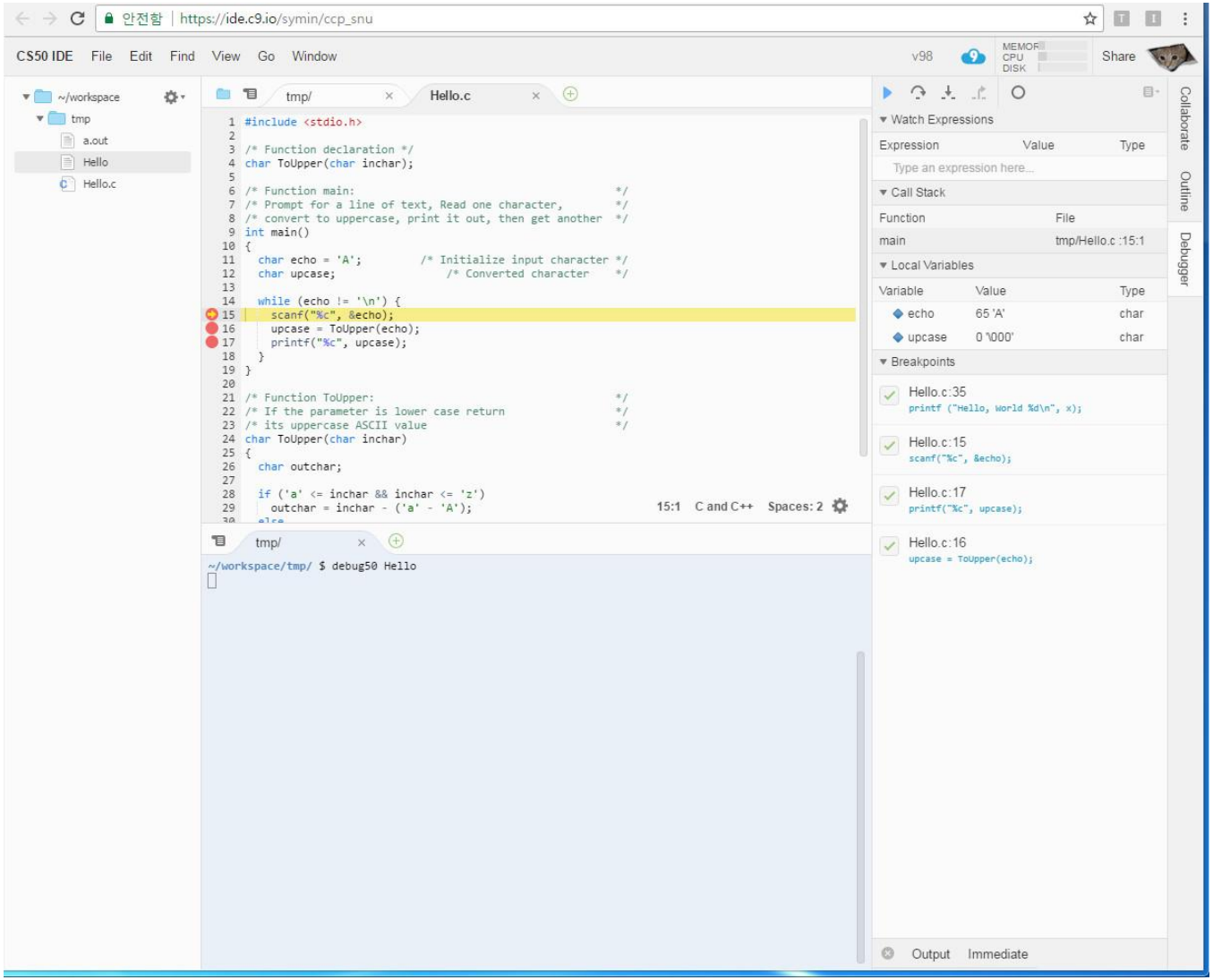
Ad-Hoc

- Insert printf statements to track control flow and values
- Code explicitly checks for values out of expected range, etc.
- Advantage:
  - No special debugging tools needed
- Disadvantages:
  - Requires intimate knowledge of code and expected values
  - Frequent re-compile and execute cycles
  - Inserted code can be buggy

Source-Level Debugger

- Examine and set variable values
- Tracing, breakpoints, single-stepping on source-code statements

Source-Level Debugger



debug web page of cloud 9

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 current function and stop on exit

Source-Level Debugging Techniques

Displaying Values

- Show value consistent with declared type of variable
- Dereference pointers (variables that hold addresses)
  - ➢ See Chapter 17
- Inspect parts of a data structure
  - ➢ See Chapters 17 and 19

Programming for Correctness

- **Accurate Specification**
- **Modular Design**
- **Defensive Programming**
  - **Comment your code**
  - **Adopt a consistent coding style**
  - **Avoid (unsubstantiated) assumptions**
  - **Avoid global variables**
  - **Reply on the compiler (let the compiler generate as many warning messages as possible)**
  - **…**

꼭 기억해야 할 것

- Type of errors
  - Syntactic errors
  - Semantic errors
  - Algorithmic errors
- Testing
  - Black-box testing
  - White-box testing
- Source-level debugger
  - Breakpoints
  - Single-stepping
  - Displaying values
- Programming for correctness
  - Accurate specification
  - Modular design
  - Defensive programming