Chapter 15
Debugging
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
- mis-spelled type in declaration

One mistake can cause an avalanche of errors
- because compiler can't recover and gets confused

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

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

```c
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__
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

```
#include <stdio.h>

int AllSum(int n);

int main()
{
    int in; /* Input value */
    int sum; /* Value of 1+2+3+...+n */
    do {
        printf("Input a number: ");
        scanf("%d", &in);
        if (in > 0) {
            sum = AllSum(in);
            printf("The AllSum of %d is %d\n", in, sum);
        }
    } while (in > 0);

    int AllSum(int n)
    {
        int i; /* Iteration count */
        int result; /* Result to be returned */
        for (i=1; i<=n; i++) /* This loop calculates sum */
            result = result + i;
    }
```

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 current function and stop on exit

LC-3 software also provides a similar capability