CS2141 – Software Development using C/C++

Debugging

Debugging Tips

- Examine the most recent change
  - Error likely in, or exposed by, code most recently added
    - Developing code incrementally and testing along the way
- Debug it now, not later
  - Bug may only reoccur when more difficult, costly, or impossible to debug
- Read before typing
  - "Debug" by avoiding errors; carefully consider impact of changes prior to making them

Debugging Tips cont.

- Make the bug reproducible
  - Hard to track down transient bug; construct input and parameter settings so that bug will appear reliably
- Don't make the same mistake twice
  - If mistake found and corrected, consider if same mistake might appear elsewhere
- Write a log file
  - Log records what happens in program prior to appearance of problem

Debugging Tips cont.

- Use tools
  - `printf()` effective; `gdb/ddd` far faster once you're past the learning curve
- Keep records
  - If bug hard to track down, keep track of what you've tried and what you've learned

Debugging Basics

- You should be able (at some level) to express what you expect the state of your program to be after every statement
- Often state *predicates* on program state; i.e., “If control is here, I expect the following to be true.”

Example

```
#include <stdio.h>

int sum=0, val, num=0;
double ave;

main()
{

while (scanf("%d\n",&val) != EOF) {
    sum += val;
    num++;
  }


if (num > 0) {
    ave = sum/num;

    printf("Average is %f\n", ave);
  }
}
```

- `sum` = 0, `num` = 0
- `sum` should be the total of inputted values, `num` should be total of inputted values
- `ave` should be the floating point average of inputted values

Using gdb

- Compile source with the `-g` switch asserted.
  - In our case, `gcc -ansi -g ave.c`
- *Breakpoint*: line in source code at which debugger will pause execution.
  - At breakpoint, can examine values of relevant components of program state.
    - `break` command sets a breakpoint; `clear` removes the breakpoint.
- Diagnostic `printf()` crude, but effective way of getting a snapshot of program state at a given point.

Using gdb cont.

- Once paused at a breakpoint, use gdb `print`, or `display` to show variable or expression values.
  - `display` will automatically print values when execution halts at breakpoint.
- From a breakpoint, may `step` or `next` to single step the program.
  - `step` stops after next source line is executed
  - `next` similar, but executes functions without stopping.

Using gdb cont.

- Find out where execution is, in terms of function call chain, with `where` command; also shows function argument values
- To make things easier, put the problematic data set in a file named data

```
% a.out < data
Average is 2.000000
```

Quickie post mortem debugging

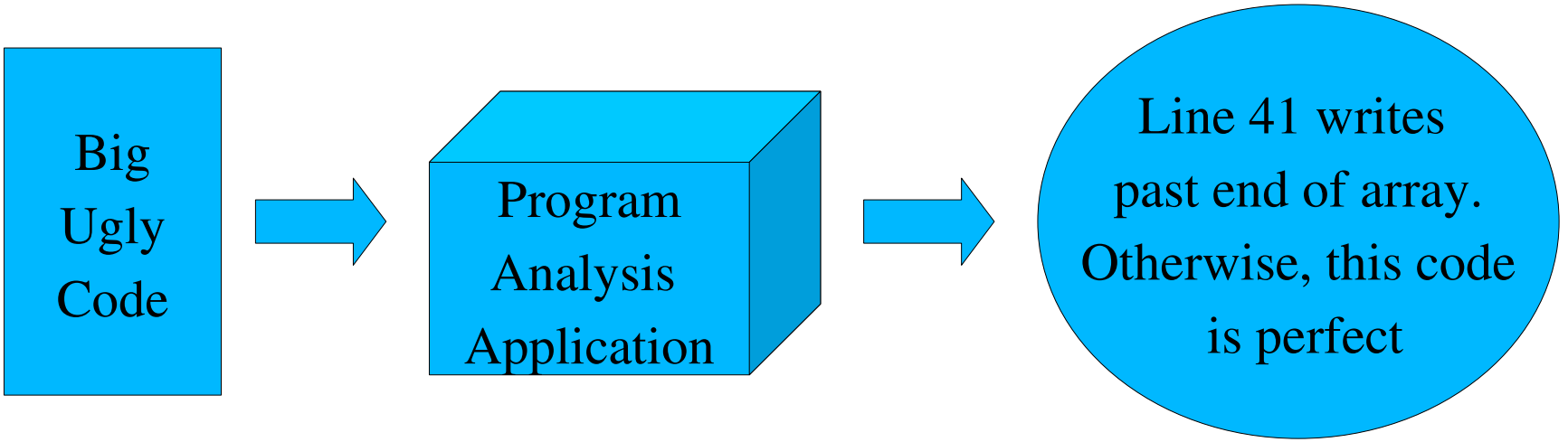
- `gdb ./a.out core`

A GUI for gdb: ddd

- Display values graphically
- Click on a pointer value, graphically display thing pointed to
- Visualize complex linked data structures

Analysis Tools

- Program analysis – process of automatically analyzing the behavior of computer programs
- Ideally:



Program Analysis

- Static analysis performed without executing program
  - Attempts to evaluate *all possible executions*
  - Difficult
    - Hard to determine all possible variable values, paths
    - Computationally complex
  - Static analysis tools tend to have voluminous and speculative output
    - Commonly identify potential problems that don't exist in actual execution, “false positives”
    - But, can identify problems not detectable by dynamic analysis

Program Analysis cont.

- Dynamic analysis performed by executing program
  - Cannot detect all bugs; only those exposed by the execution being analyzed
  - Instrumentation required for analysis
    - Can change the execution
  - Not prone to “false positives”

Splint

- Static C program checker
- Security vulnerabilities and coding mistakes
- http://www.splint.org
-

Splint cont.

- Problems detected
  - Dereferencing a potentially null pointer
  - Using potentially undefined storage or returning storage that is not properly defined
  - Type mismatches, with greater precision and flexibility than provided by C compilers
  - Memory management errors including uses of dangling references and memory leaks
  - Problematic control flow such as likely infinite loops, fall through cases or incomplete switches
  - Buffer overflow vulnerabilities
  - And others

Valgrind

- Debugging and profiling tool
- http://valgrind.org/

Valgrind cont.

- Dynamic analysis
- Several components:
  - Memcheck – memory management problems
    - This is our focus
  - Cachegrind – cache profiler
    - For performance tweaking, find source of cache misses
  - Massif – heap profiler
    - Depict heap usage over time
  - Helgrind – data races in *multithreaded* programs

memcheck

- Use of uninitialized memory
- Read/write
  - of memory after free
  - off end of blocks allocated via malloc
  - inappropriate areas of stack
- Leaks
- Mismatched use of malloc/new/new[] and free/delete/delete[]
- Etc.
- *Does not bounds check on statically allocated arrays*