15-213

"The course that gives CMU its Zip!"

Debugging
November 9, 2004

Topics

- Defensive programming
- Know your bugs
- Debugging tricks
- Overview of available tools

Defensive Programming

5-20 Bugs per 1000 lines of code (InfoWorld, Oct. 2003)

Programmers must anticipate bugs, even if *your* code is bug-free.

How?

- Check for errors at all possible opportunities: detecting bugs early eases finding the root cause.
- Maintain a clean, modular structure with documented interfaces: *goto's, global variables, long-jumps, clever/obscure macros*, etc. considered ~~harmful~~ dangerous.
- Anticipate common errors: buffer overrun, off-by-one,…
- Consider corner cases: 0/1 loops, empty lists, …
- Provide debugging support in your program: debugging messages, data structure checkers (like the Heap-checker from the `malloc`-lab), print-function for complicated structures, test-case generators, …
- Add redundancy
- Maintain test cases for regression testing: use version control systems (CVS, RCS, BitKeeper, Subversion, …)
- Use all the help you can get: heed compiler warnings, use debuggers, verifyers, IDE's, code generators, high-level tools,…

Assertions

Explicitly state what you expect to be true in your program: invariants, argument ranges, etc.

Assert-macro (ISO9899, ANSI C):

```
#include <assert.h>
```

Generates no tests if "NODEBUG" is defined

```
#define MAX_ARRAY_SIZE 10

void foo(double a[], double b[], int n)
{ int i;
  double *a_ptr = a, *b_ptr = b;
  assert(n > 1 && n <= MAX_ARRAY_SIZE);

  for (i = n; --i;) {
    /* ... */
    a_ptr++;
    /* ... */
    b_ptr++;
  }

  assert(a_ptr == &(a[n]) && b_ptr == &(b[n]));
}
```


Assertions (cont.)

Reasons for using assertions:

- Catch failures early
- Verify & document interfaces
- Express invariants to aid debugging

Debug Messages

Use of cpp-macros and conditional compilation:

```
#ifdef DEBUG
extern int debug_level;
#define DEBUG_PRINT(level, format, args...) \
    { if((level) < debug_level) {\
        fprintf(stderr, "DEBUG_PRINT line=%d in file='%s':\n",\
                __LINE__, __FILE__);\
        fprintf(stderr, format , ## args);}\
    }
#else
#define DEBUG_PRINT(level, format, args...)
#endif


foo(int a, int b) {
  DEBUG_PRINT(0, "foo(a=%d, b=%d) started\n", a, b);
}
```


Add Redundancy

Engineering tradeoff between robustness and performance.

Extreme case Google:

- Data structures have software maintained checksums
- Distributed system (> 10,000 machines): need fail-stop characteristic, handle failures at higher level

Simple Cases:

- Count item and compare to pointer difference (see assertion example)
- Compute simple, inexpensive invariants (for example: the sum of allocated and free memory objects in the heap ought to equal the heap size)

Integrated Development Environment

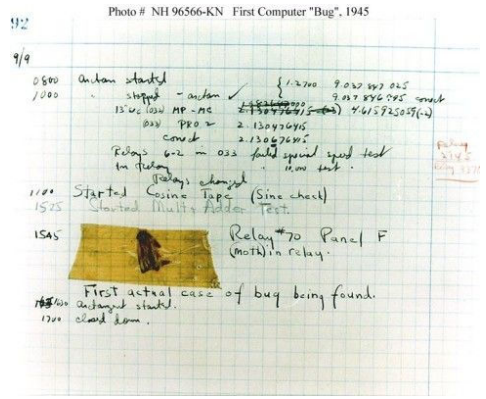
Program-editor (with syntax support), version control system, compiler, debugger, build-system, profiler, graphical user interface, and integration = IDE

- Microsoft Visual-*
- IBM's Eclipse project
- Kdevelop (open source)

- Pro: convenience
- Con: often platform dependent
- No silver bullet

Debugging History

In 1945 G. Hopper found the first "bug" in IBM's *Harvard Mark I*, an electro-mechanical computer:

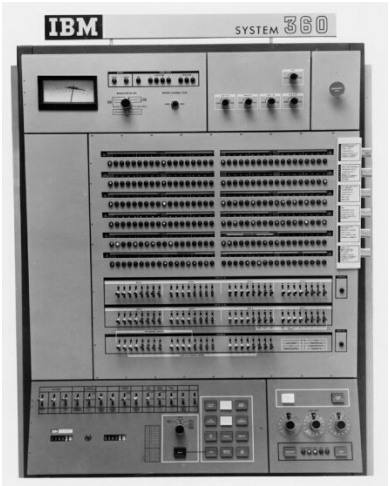


Early Debugging

Use of front pannel switches & lights:

Other tools included:

- Core dumps
- Print statements
- Hardware monitors
- Speakers



Know your Bugs

Common bugs in C-programs

- Pointer bugs
- Dynamic memory allocation / deallocation bugs
- Memory leaks (missing / extra `free()` calls)
- Buffer overflow bugs
- Arrays out of bound errors (off-by-one)
- Exception handling
- Variable scope problems (see linking lecture)
- Race conditions in multi-threaded codes

Other bugs not considered in this class:

- Specification errors
- Performance bugs
- Program logic errors (bad algorithms, data structures, etc.)

Encounter with a Bug

Program produces unexpected result and/or crashes

- Is this behavior reproducible?
- Does it depend on input data?
- Does it change with compilation options? (-g vs. –O2)

First goal: narrow the possible code range that could be responsible for the bug:

- Divide & Conquer
- Simplify the code that shows the bug
- In case of rare/intermittent bugs: try to cause the program to fail more frequently
- Add logging or debugging printouts to pinpoint the approximate location of the failure

GDB (GNU DeBugger)

Basic functionality:

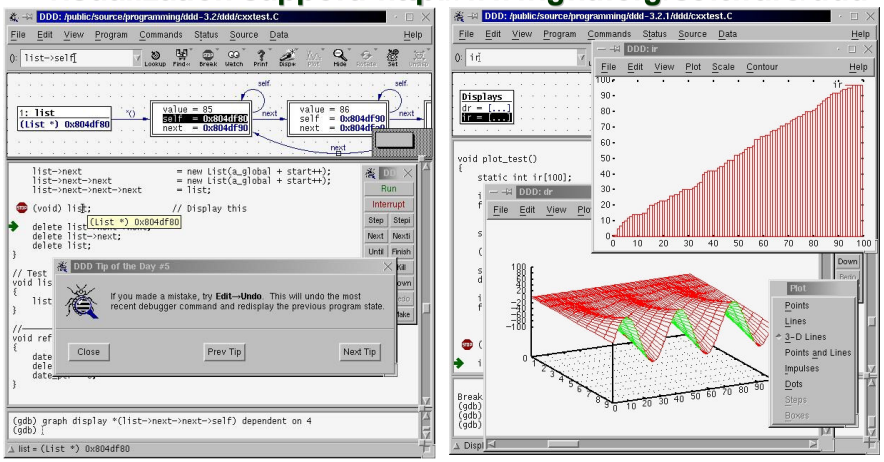
- Can run programs in an observable environment
- Uses `ptrace`-interface to insert breakpoint, single step, inspect & change registers and variables
- Does not require compilation with "-g", but works much better if it has the symbol tables available
- Maintains source line numbers and can inspect source files

- Ability to attach to a running process
- Ability to watch memory locations
- Conditional breakpoints

- Some graphical user interfaces exist (DDD, KDbg, …)

DDD

Graphical front-end to GDB with extended data visualization support: http://www.gnu.org/software/ddd



Annoyingly Frequent Case:

Memory corruption due to an earlier pointer or dynamic memory allocation error: bug cause and effect are separated by 1000's of instructions

- Use GDB to watch the corruption happen:
  - Use conditional breakpoints: `break ... if cond`
  - Set a watchpount: `[r,a]watch expr`
- Use *dog-tags* in your program
- Use a debugging-version of *malloc()*
- Use run-time verification tools

Dogtags

GDB style watch points are frequently too slow to be used in large, complex programs.

```
#ifdef USE_DOG_TAGS
#define DOGTAG(x) int x;
#else
#define DOGTAG(x)
#endif
```

```
struct foobar {
    DOGTAG(dt1);
    int buf[20];
    DOGTAG(dt2);
};
```

- If dogtags are enabled, maintain a list of all allocated dogtags (easier with C++ class objects using the constructor)
- Initialize dogtags to a distinct value (e.g. 0xdeadbeeef)
- Provide function that checks the integrity of the dogtags
- When to call this function?

Dogtags (continued)

Call check funtion near suspect codes by manually inserting calls or (*hack alert*):

```
#ifdef AUTO_WATCH_DOG_TAGS
#define if(expr) if (CHECK_WATCHED_DOG_TAGS,(expr))
#define while(expr) while (CHECK_WATCHED_DOG_TAGS,(expr))
#define switch(expr) switch (CHECK_WATCHED_DOG_TAGS,(expr))
#endif /* AUTO_WATCH_DOG_TAGS */
```

Dynamic Memory Allocation Checker

`malloc()` and friends are a frequent source of trouble therefore there are numerous debugging aids for this problem. The typical functionality include:

- Padding the allocated area with dogtags that are checked when any dynamic memory allocation functions are called or on demand.
- Checking for invalid free() calls (multiple, with bad argument)
- Checking for access to freed memory regions
- Keeping statistics of the heap utilization
- Logging

MALLOC_CHECK_

In recent versions of Linux libc (later than 5.4.23) and GNU libc (2.x), defining `MALLOC_CHECK_` causes extra checks to be enabled (at the expense of lower speed):

- Checks for multiple free() calls
- Overruns by a single byte

Boehm-Weiser Conservative Garbage Collector

Ref: http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Idea: forget about free() calls and try to use garbage collection within C. Has to be conservative.

- Checks for existing pointers to allocated memory regions
- Circular pointers prevent reclaiming
- Assumes that pointers point to first byte (not necessarily true)
- Assumes that pointers are not constructed on the fly

Electric Fence, by Bruce Perens

Ref: http://sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence-2.0.5.tar.gz

Idea: use the virtual memory mechanism to isolate and protect memory regions

- Pro: very fast – uses hardware (page faults) for the testing
- Con: Fairly large memory overhead due to page-size granularity
- Variations of this idea: Wisconsin Wind-Tunnel project – uses ECC bits to get finer granularity (highly platform dependent)

Run Time Memory Checkers

Very powerful tools that use binary translation techniques to instrument the program:

- The program (executable or object files) is disassembled and memory access (or any other operations) are replaced with code that add extra checking
- Generally results in a 2-50x slow-down, depending on the level of checking desired
- Can be used for profiling and performance optimizations

Pixie, Atom, 3rd Degree, Tracepoint

Originally conceived as tool for computer architecture research. Started out as instruction level interpreters then added compilation facilities

- Pixie: MIPS specific
- Shade: Sun specific
- ATOM: Alpha specific
  - 3rd Degree used Atom for debugging and verification purposes
  - Tracepoint tries (unsuccessfully) to commercialize this tool

Valgrind (IA-32, x86 ISA)

Open source software licensed under the GPL (like Linux): http://valgrind.kde.org/index.html

Valgring is a general purpose binary translation infrastructure for the IA-32 instruction set architecture

Tools based on *Valgrind* include:

- *Memcheck* detects memory-management problems
- *Addrcheck* is a lightweight version of Memcheck which does no uninitialised-value checking
- *Cachegrind* is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches in your CPU
- *Helgrind* is a thread debugger which finds data races in multithreaded programs

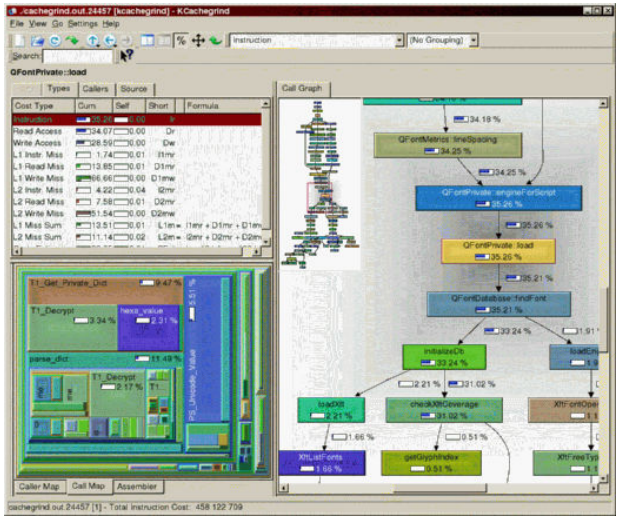
Memcheck

Uses Valgrind to:

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Passing of uninitialised and/or unaddressible memory to system calls
- Mismatched use of `malloc/new/new[]` vs `free/delete/delete []`
- Overlapping `src` and `dst` pointers in `memcpy()` and related functions
- Some misuses of the POSIX pthreads API

KCachegrind

Profiling and cache simulation tool based on Valgrind



Purify

Reed Hastings and Bob Joyce. "*Purify: Fast detection of memory leaks and access errors*" In Proc. 1992 Winter USENIX Conference, pages 125--136, 1992

Commercialized by Rational Software, acquired by IBM

- Binary translation based verification system with high level program development extension (project management)
- Earlier versions used in 15-211 (1997)
- Pro: Very mature, powerful tool
- Con: Costly, limited range of supported platforms
- Commercial competitor: Insure++ from Parasoft

Profiling

Where is your program spending its CPU time?

Profiling is used to find performance bugs and to fine-tune program performance.

Principle approaches:

- Compile time instrumentation (gcc –p …)
- Statistical sampling (DCPI for Alpha based machines)
- Instrumentation via binary translation tools

gcc –pg …

Add instrumentation (counters) at function granularity (calls to `mcount()`)

```
[agn@char src]$ gprof driver gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.03     15.71    15.71       51   308.04   308.04  naive_kernel
 14.11     20.14     4.43 88358912     0.00     0.00  is_alive
  7.58     22.52     2.38       20   119.00  1570.00  run_benchmark
  6.02     24.41     1.89 11044258     0.00     0.00  s_buf1_set
  5.76     26.22     1.81 11044258     0.00     0.00  s_buf_set
  5.67     28.00     1.78       51    34.90    34.90  nofunc8_next_generation
  3.18     29.00     1.00 11044258     0.00     0.00  naive_set
  2.29     29.72     0.72       51    14.12    14.12  s_buf_kernel
  2.10     30.38     0.66 11044258     0.00     0.00  nofunc5_turn_on
. . .
```

Debugging an Entire System?

Debugging kernel level code is hard: mistakes generally crash the system. Real-time constraints prevent seting breakpoint is places like interrupt handlers or I/O drivers.

Alternatives:

- SimOS (Stanford, http://simos.stanford.edu/) defunct
- Vmware: commercial version of SimOS for virtualizing production server, running Windows under Linux or vice versa
- Simics: commercial system level simulation for computer architecture research and system level software development
- User Mode Linux: Run Linux under Linux as a user level process http://user-mode-linux.sourceforge.net/
- Xen: Hypervisor provides virtualized machine for kernel to run on: http://www.cl.cam.ac.uk/Research/SRG/netos/xen/index.html
- Denali Isolation Kernel: virtualized machine and special guest OS: http://denali.cs.washington.edu/pubs/

User Level Linux

User-Mode Linux is a safe, secure way of running Linux versions and Linux processes. Run buggy software, experiment with new Linux kernels or distributions, and poke around in the internals of Linux, all without risking your main Linux setup.