Debugging:

how I learned to stop hacking and love my tools

What You Won't Learn

- how to solve all your problems
- the one true way to avoid all bugs
- Many platform specific debugging techniques

Documentation

- learn where the documentation for your system is and use it
- when in doubt, look it up!
  - Internet is your friend, but many cluless people so must filter. Use definitive sources if you can
  - man pages/Info system for Unix systems
  - MSDN for windows
  - javadoc for Java libraries
  - /Developer/Documentation for Mac OS X

Debugging

- Debugging is a black art. Some things to go over, though, so they'll be concrete in our brains:
  - relation to testing
  - why debugging is hard
  - types of bugs
  - process
  - techniques
  - tools
  - avoiding bugs

Debugging and testing

- Testing and debugging go together like peas in a pod:
  - Testing finds errors; debugging localizes and repairs them.
  - Together these form the "testing/debugging cycle": we test, then debug, then repeat.
  - Any debugging should be followed by a reapplication of *all* relevant tests, particularly regression tests. This avoids (reduces) the introduction of new bugs when debugging.
  - Testing and debugging need not be done by the same people (and often should not be).

Why debugging is hard

- There may be no obvious relationship between the external manifestation(s) of an error and its internal cause(s).
- Symptom and cause may be in remote parts of the program.
- Changes (new features, bug fixes) in program may mask (or modify) bugs.
- Symptom may be due to human mistake or misunderstanding that is difficult to trace.
- Bug may be triggered by rare or difficult to reproduce input sequence, program timing (threads) or other external causes.
- Bug may depend on other software/system state, things others did to you systems weeks/months ago.

Designing for Debug/Test

- when you write code think about how you are going to test/debug it
  - lack of thought *always* translates into bugs
- write test cases when you write your code
- if something should be true assert() it
- create functions to help visualize your data
- design for testing/debugging from the start
- test early, test often
- test at abstraction boundaries

Fault Injection

- many bugs only happen in the uncommon case
- make this case more common having switches that cause routines to fail
  - file open, file write, memory allocation, are all good candidates

- Have "test drivers" which test with the uncommon data. If deeply buried, test with a debugger script

Finding and Fixing Bugs

- in order to create quality software you need to find your bugs
  - testing
  - user reports
- the best bugs are those that are always reproducible

Types of bugs

- Types of bugs (gotta love em):
  - Compile time: syntax, spelling, static type mismatch.
    - Usually caught with compiler
  - Design: flawed algorithm.
    - Incorrect outputs
  - Program logic (if/else, loop termination, select case, etc).
    - Incorrect outputs
  - Memory nonsense: null pointers, array bounds, bad types, leaks.
    - Runtime exceptions
  - Interface errors between modules, threads, programs (in particular, with shared resources: sockets, files, memory, etc).
    - Runtime Exceptions
  - Off-nominal conditions: failure of some part of software of underlying machinery (network, etc).
    - Incomplete functionality
  - Deadlocks: multiple processes fighting for a resource.
    - Freeze ups, never ending processes

The ideal debugging process

- A debugging algorithm for software engineers:
  - Identify test case(s) that reliably show existence of fault (when possible)
  - Isolate problem to small fragment(s) of program
  - Correlate incorrect behavior with program logic/code error
  - Change the program (and check for other parts of program where same or similar program logic may also occur)
  - Regression test to verify that the error has really been removed - without inserting new errors
  - Update documentation when appropriate

(Not all these steps need be done by the same person!)

General Advice

- try to understand as much of what is happening as possible
- "it compiles" is **NOT** the same as "it works"
- when in doubt, ask. Then test the answer!
- Error messages are generally just a vague hint and can be misleading.
- Don't always trust the "comments/documents", they can be out-of-date.

What is a Debugger?

"A software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables."

-MSDN

What is a Debugger? (con't)

- A debugger is *not an IDE*
  - Though the two can be integrated, they are separate entities.
- A debugger loads in a program (compiled executable, or interpreted source code) and allows the user to trace through the execution.
- Debuggers typically can do disassembly, stack traces, expression watches, and more.

Why use a Debugger?

- No need for precognition of what the error might be.
- Flexible
  - Allows for "live" error checking – no need to re-write and re-compile when you realize a certain type of error may be occuring
  - Dynamic
  - Can view the entire relevant scope

Why people don't *use a Debugger?*

- With simple errors, may not want to bother with starting up the debugger environment.
  - Obvious error
  - Simple to check using prints/asserts
- Hard-to-use debugger environment
- Error occurs in optimized code
- Changes execution of program (error doesn't occur while running debugger)

Debugging techniques, 1

- Execution tracing
  – running the program
  – print
  – trace utilities
  – single stepping in debugger
  – hand simulation

Debugging techniques, 2

- Interface checking
  - check procedure parameter number/type (if not enforced by compiler) and value
  - *defensive programming*: check inputs/results from other modules
  - documents assumptions about caller/callee relationships in modules, communication protocols, etc
- Assertions: include range constraints or other information with data.
- Skipping code: comment out suspect code, then check if error remains.

Other Functions of a Debugger

- Disassembly (in context and with live data!)
- Execution Tracing/Stack tracing
- Symbol watches

Disassembly

- Most basic form of debugging
- Translating machine code into assembly instructions that are more easily understood by the user.
- Typically implementable as a simple lookup table
- No higher-level information (variable names, etc.)
- Relatively easy to implement.

Execution Tracing

- Follows the program through the execution. Users can step through line-by-line, or use breakpoints.
- Typically allows for “watches” on – registers, memory locations, symbols
- Allows for tracing up the stack of runtime errors (back traces)
- Allows user to trace the causes of unexpected behavior and fix them

Symbol Information

- Problem – a compiler/assembler translates variable names and other symbols into internally consistent memory addresses
- How does a debugger know which location is denoted by a particular symbol?
- We need a "debug" executable.

Debug vs. Release Builds

- Debug builds usually are *not optimized*
- Debug executables contain:
  - program's symbol tables
  - location of the source file
  - line number tags for assembly instructions.
- GCC/GDB allows debugging of optimized code.

Bug hunting with print

- Weak form of debugging, but still common
- How bug hunting with print can be made more useful:
  - print variables other than just those you think suspect.
  - print valuable statements (not just “hi\n”).
  - use exit() to concentrate on a part of a program.
  - move print through a through program to track down a bug.

Debugging with print (continued)

- Building debugging with print into a program (more common/valuable):
  - print messages, variables/test results in useful places throughout program.
  - use a ‘debug’ or ‘debug_level’ global flag to turn debugging messages on or off, or change “levels”
  - possibly use a source file preprocessor (#ifdef) to insert/remove debug statements.
  - Often part of “regression testing” so automated scripts can test output of many things at once.

Finding Reproducible Bugs

- a bug happens when there is a mismatch between what you (someone) *think* is happening and what is *actually* happening
- confirm things you believe are true
- narrow down the causes one by one
- make sure you understand your program state
- keep a log of events and assumptions

Finding Reproducible Bugs

- try explaining what should be happing
  - Verbalization/writing often clarifies muddled thoughts
- have a friend do a quick sanity check
- don't randomly change things, your actions should have a purpose.
  - If you are not willing to check it into CVS with a log that your boss may read, then you are not ready to make that change to the code.
  - Think it through first, both locally and globally.

(semi) irreproducible bugs

- sometimes undesired behavior only happens sporadically
- tracking down these heisenbugs is hard
- the error could be a any level
  - Circuits (e.g. bad ram chip at high memory address)
  - compiler
  - os
  - Linker
  - Irreproducible external "data" and timing

Finding HeisenBugs

- Use good tools like Purify. Most common "Heisenbugs" are memory or thread related.
- try to make the bug reproducible by switching platforms/libraries/compilers
- insert checks for invariants and have the program stop everything when one is violated
- verify each layer with small, simple tests
- find the smallest system which demonstrates the bug
- Test with "canned data", replayed over net if needed.

Timing and Threading Bugs

- ensure the functionality works for a single thread
- if adding a printf() removes the bug it is almost certainly a timing/threading bug or a trashed memory bug
- try using coarse grained locking to narrow down the objects involved
- try keeping an event (transaction) log

Memory Bugs and Buffer Overflow

- Trashing stack/heap causes often difficult to find bugs.
- Manifestation can be far from actual bug.
  - "Free list" information generally stored just after a "malloced" chunck of data. Overwriting may not cause problem until data is "freed", or until something else does a malloc after the free.
  - Stack variables, overwriting past end, changes other variables, sometimes return address. (Buffer overflow)
  - Bad "string" ops notorious, using input data can also be problematic.

An example….

```
void myinit(int startindex, int startvalue, int length, int* vect){
  int i;
  for(i=startindex; i< startindex+length; i++)
     *vect++ = startvalue++;
}
void whattheheck(){    printf("How did I ever get here????\n");   exit(2); }

int main(int argc, char**argv){
  float d;
  int a,b[10],c, i, start,end;
  if(argc != 3) {printf("Usage:%s start, end\n",argv[0]);exit(-1);  }
    start=atoi(argv[1]);   end=atoi(argv[2]);  /* bad style but shorter */
    a=0;      c=0;         d=3.14159;  /* bad style but shorter */
    printf("Initally a %d, c %d, d %f, start %d, end %d\n",a,c,d, start,end);
    myinit(start,start,end,b+start);
    printf("finally a %d, c %d, d %f start %d, end %d \n",a,c,d, start, end);
    if(end>10) b[end-1]=134513723;
    return 0;
}
```

Debugging Techniques

- methodology is key
- knowing about lots of debugging tools helps
- the most critical tool in your arsenal is your brain
- second most important tool is a debugger
  - Core dumps are your friends.. Learn how to use them.
- Memory debugging tools third most important
- Profiler fourth

Tools

- Tracing programs:
  - strace, truss (print out system calls),
  - /bin/bash –X to get a shell script to say what its doing
- Command-line debuggers :
  - gdb (C, C++), jdb (java), “perl -d”
- Random stuff:
  - electric fence or malloc_debug, mtrace, etc (a specialized malloc() for finding memory leaks in C/C++)
  - purify (Part of Rational Suite, a really good memory debugging tools for C/C++ programs)

Debuggers

- allow programs to inspect the state of a running program
- become more important when debugging graphical or complex applications
- jdb and gjdb for java
- gdb for C/C++ programs on unix/Max
- MSVS for Win32
- kdb for the linux kernel

Using gdb

- Compile debugging information into your program:

  `gcc -g <program>`

- Read the manual:
  - `man gdb`
  - in emacs (tex-info):

    `http://www.cslab.vt.edu/manuals/gdb/gdb_toc.html`

    `M-x help <return> i <return>`, arrow to GDB, `<return>`
  - online:

gdb comands

`run/continue/next/step`: start the program, continue running until break, next line (don't step into subroutines), next line (step into subroutines).

`break <name>`: set a break point on the named subroutine. Debugger will halt program execution when subroutine called.

`backtrace`: print a stack trace.

`print <expr>`: execute expression in the current program state, and print results (expression may contain assignments, function calls).

`help`: print the help menu. help <subject> prints a subject menu.

`quit`: exit gdb.

General GDB

- easiest to use inside of emacs (M-x gdb)
- run starts the program
- set args controls the arguments to the program
- breakpoints control where the debugger stops the program (set with C-x space)
- next moves one step forward
- step moves one step forward and also traverses function calls
- continue runs the program until the next breakpoint

General GDB

- p prints a value (can be formated)
  - /x hex
  - /o octal
  - /t binary (t is for two)
  - /f float
  - /u unsigned decimal

General GDB

- bt shows the current backtrace (also where)
- up/down move up down the stack
- f # lets you switch to frame # in the current backtrace
- set var=value
- call allows call of a function (the syntax can get funky)
- jump (dump to a particular line of code)
- Thread # lets you switch to a particular thread

Advanced GDB

- watchpoints let you check if an expression changes
- catchpoints let you know when interesting things like exec calls or library loads happen
- x lets you inspect the contents of memory
- Watchpoints can be quite slow, best to combine with breakpoints.

Advanced GDB

- gcc 3.1 and up provides macro information to gdb if you specify the options -gdwardf2 and -g3 on the command line
- you can debug and already running process with the attach *pid* command
- you can apply a GDB command to all threads with thread apply all
- GDB can be used to debug remote embedded systems (gdbserver, etc..)

The example in gdb

```
void myinit(int startindex, int startvalue, int length, int* vect){
  int i;
  for(i=startindex; i< startindex+length; i++)
     *vect++ = startvalue++;
}
void whattheheck(){    printf("How did I ever get here????\n");   exit(2); }

int main(int argc, char**argv){
  float d;
  int a,b[10],c, i, start,end;
  if(argc != 3) {printf("Usage:%s start, end\n",argv[0]);exit(-1);  }
    start=atoi(argv[1]);   end=atoi(argv[2]);  /* bad style but shorter */
    a=0;      c=0;         d=3.14159;  /* bad style but shorter */
    printf("Initally a %d, c %d, d %f, start %d, end %d\n",a,c,d, start,end);
    myinit(start,start,end,b+start);
    printf("finally a %d, c %d, d %f start %d, end %d \n",a,c,d, start, end);
    if(end>10) b[end-1]=134513723;
    return 0;
}
```

JDB

- Compile your source file with **`-g`** option
  - Produce debugging information
  - For example

    `javac -g rsdimu.java`
- To run jdb,
  - Type “jdb”, use run command to load an executable
    - **`run <class name>`**
    - For example, **`run rsdimu`**
  - OR type “jdb <class name>, then
    - Type “run” to execute the program
  - Type “quit” to exit jdb

Breakpoint

- Make your program stops whenever a certain point in the program is reached
- For example
  - Make a breakpoint in line 6

    `stop at Demo:6`
  - Program stops before execute line 6
  - Allow you to examine code, variables, etc.

```
1 public class Demo
2 {
3    public static void main(...)
4    {
5       System.out.println
6 ("A\n");
7       System.out.println
8 ("B\n");
9       System.out.println
```

Breakpoint

- Add breakpoint
  - `stop at MyClass:<line num>`
  - `stop in java.lang.String.length`
  - `stop in MyClass.<method name>`

- Delete breakpoint
  - `clear (clear all breakpoints)`
  - `clear <breakpoint>`
    - `e.g. clear MyClasss:22`

Step

- Execute one source line, then stop and return to JDB
- Example

```
public void func()
{
  System.out.println
("A\n");
  System.out.println
("B\n");
  System.out.println
("C\n");
```

step

```
public void func()
{
  System.out.println("A\n");
  System.out.println("B\n");
  System.out.println("C\n");
  return 0;
}
```

Next

- Similar to step, but treat function call as one source line
- Example

```
public void func1() {
        println("B\n");
        println("C\n");
}
public void func() {
        println("A\n");
        func1();
        return 0;
}
```

step

next

```
public void func1() {
        println("B\n");
        println("C\n");
}
public void func() {
        println("A\n");
        func1();
        return 0;
}
```

Cont

- Resume continuous execution of the program until either one of the followings
  - Next breakpoint
  - End of program

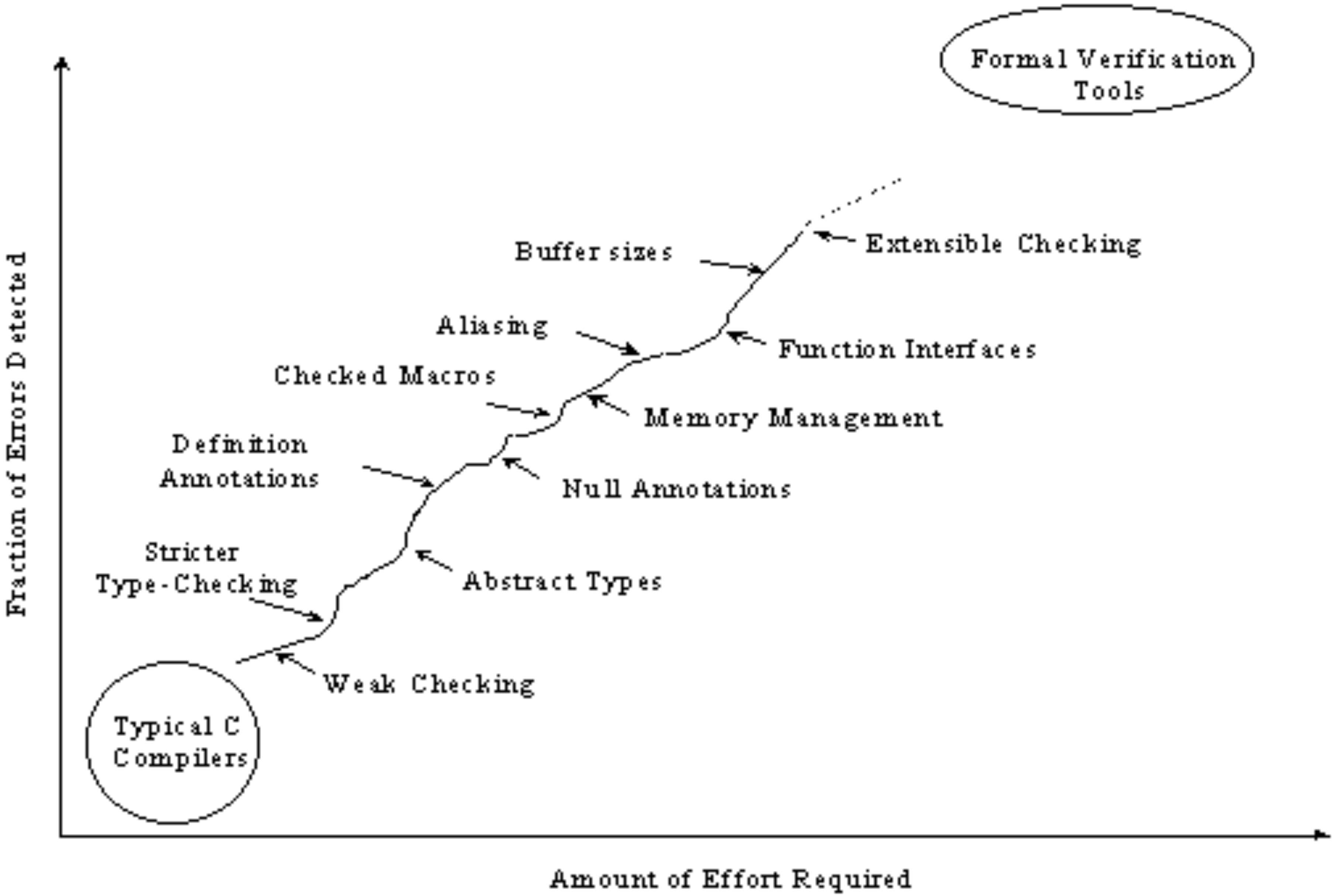
Print

- Print
  - Display the value of an expression
    - `print expression`
      - **print MyClass.myStaticField**
      - **print i + j + k**
      - **print myObj.myMethod()** ***(if myMethod returns a non-null)***
      - **print new java.lang.String("Hello").length()**
- Dump
  - Display all the content of an object
    - `dump <object>`

Analysis Tools

Purpose of Analysis Tools

- Need for a feasible method to catch bugs in large projects. Formal verification techniques require unreasonable effort on large projects.
- Augment traditional debugging techniques without adding unreasonable burden to the development process.



Formal Verification Tools
Extensible Checking
Buffer sizes
Aliasing
Function Interfaces
Checked Macros
Memory Management
Definition Annotations
Null Annotations
Stricter Type-Checking
Abstract Types
Weak Checking
Typical C Compilers
Fraction of Errors Detected
Amount of Effort Required

Two Types of Analysis Tools

- Static Analysis
- Run-time (dynamic) Analysis

Static Analysis

- Examine a program for bugs without running the program.
- Examples:
  - Splint (www.splint.org),
  - PolySpace C Verifier (www.polyspace.com).

Splint

- Open Source Static Analysis Tool developed at U.Va by Professor Dave Evans.
- Based on Lint.

Errors Splint will detect

- Dereferencing a possibly null pointer.
- Using possibly undefined storage or returning storage that is not properly defined.
- Type mismatches, with greater precision and flexibility than provided by C compilers.
- Violations of information hiding.
- Memory management errors including uses of dangling references and memory leaks.
- Dangerous aliasing.

Errors Splint will detect continued…

- Modifications and global variable uses that are inconsistent with specified interfaces.
- Problematic control flow such as likely infinite loops.
- Buffer overflow vulnerabilities.
- Dangerous macro initializations and invocations.
- Violations of customized naming conventions.

What's wrong with this code?

```
void strcpy(char* str1, char* str2)
{
    while (str2 != 0)
    {
        *str1 = *str2;
        str1++;
        str2++;
    }
    str1 = 0; //null terminate the string
}
```

What happens to the stack?

```
void foo()
{
    char buff1[20]; char buff2[40];
    …
    //put some data into buff2
    strcpy(buff1, buff2);
}
```

Secure Programming

- Exploitable bugs such as buffer overflows in software are the most costly bugs.
- Incredibly frequent because they are so hard to catch.
- Analysis tools play a big part in finding and fixing security holes in software.

How does Splint deal with false positives?

- Splint supports annotations to the code that document assumptions the programmer makes about a given piece of code
- These annotations help limit false positives.

Run-time Analysis

- Many bugs cannot be determined at compile time. Run-time tools required to find these bugs.
- Run-time analysis tools work at run-time instead of compile time.
- Example – Purify (www.rational.com).

Purify

- Purify modifies object files at link time.
- After execution, Purify will report bugs such as memory leaks and null dereferences.

Purify continued…

- From the purify manual: “Purify checks every memory access operation, pinpointing where errors occur and providing diagnostic information to help you analyze why the errors occur.”

Types of errors found by Purify

- Reading or writing beyond the bounds of an array.
- Using un-initialized memory.
- Reading or writing freed memory.
- Reading or writing beyond the stack pointer.
- Reading or writing through null pointers.
- Leaking memory and file descriptors.
- Using a pointer whose memory location was just deallocated

Static vs. Run-time Analysis

- Probably good to use both.
- Run-time analysis has fewer false positives, but usually requires that a test harness test all possible control flow paths.

Cons of Analysis Tools

- Add time and effort to the development process.
- Lots of false positives.
- No guarantee of catching every bug.
- However, in a commercial situation, probably worth your time to use these tools.

Other Tools

- Mallocdebug and debugmalloc libraries
- valgrind is a purify-like tool which can really help track down memory corruption (linux only)
- MemProf for memory profiling and leak detection (linux only) www.gnome.org/projects/memprof
- electric fence is a library which helps you find memory errors
- c++filt demangles a mangled c++ name

Memory Leaks

- Memory bugs
  - Memory corruption: dangling refs, buffer overflows
  - Memory leaks
    - Lost objects: unreachable but not freed
    - Useless objects: reachable but not used again

Memory Leaks

- Memory bugs
  - Memory corruption: dangling refs, buffer overflows
  - Memory leaks
    - Lost objects: unreachable but not freed
    - Useless objects: reachable but not used again

Managed Languages

- 80% of new software in Java or C# by 2010 [Gartner] (personally TB does not believe it..)
- Type safety & GC eliminate many bugs

Memory Leaks

- Memory bugs
  - ~~Memory corruption: dangling refs, buffer overflows~~
  - Memory leaks
    - ~~Lost objects: unreachable but not freed~~
    - Useless objects: “reachable” but not used again

Managed Languages

- 80% of new software in Java or C# by 2010 [Gartner]
- Type safety & GC eliminate many bugs

Memory Leaks

Leaks occur in practice in managed languages
[Cork, JRockit, JProbe, LeakBot, .NET Memory Profiler]

- Memory leaks
  - ~~Lost objects: unreachable but not freed~~
  - Useless objects: reachable but not used again

Managed Languages

- 80% of new software in Java or C# by 2010 [Gartner]
- Type safety & GC eliminate many bugs

Other linux Tools

- strace / truss / ktrace let you know what system calls a process is making
- Data Display Debugger (DDD) is good for visualizing your program data www.gnu.org/software/ddd
- gcov lets you see which parts of your code are getting executed
- Profilers (gprof) to see where you are spending "time" which can help with performance logic bugs

Code beautifier

- Improve indentation of your source code for better readability
- source code beautifier in UNIX/Cygwin
  - Indent
  - M-x indent-region in emacs
- Make sure the code beautifier does not change how your code works after beautification!

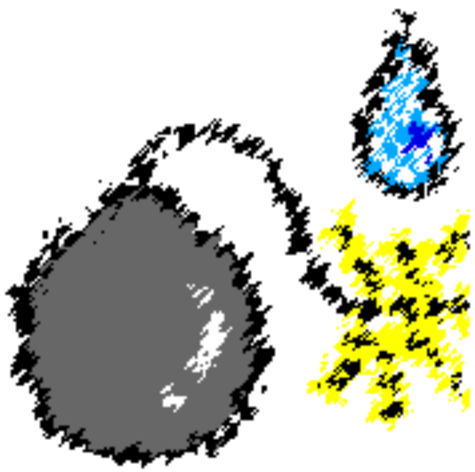
Avoiding bugs in the first place

- Coding style: use clear, consistent style and useful naming standards.
- Document everything, from architecture and interface specification documents to comments on code lines.
- Hold code reviews.
- Program defensively.
- Use/implement exception handling liberally; think constantly about anomalous conditions.
- Be suspicious of cut/paste.
- Consider using an integrated development environment (IDE) with dynamic syntax checking

Code reviews

- Primary programmer(s) for some piece of code presents and explains that code, line by line.
- Audience of programmers experienced in language, code's general domain. Audience may also contain designers, testers, customers and others less versed in code but concerned with quality and consistency.
- Review is a dialogue: audience pushes presenters to reevaluate and rationalize their implementation decisions.
- Extremely useful: reviews often turn up outright errors, inconsistencies, inefficiencies and unconsidered exceptional conditions.
- Also useful in familiarizing a project team with a member's code.

War Stories



Nasty Problems

- overwriting return address on the stack
- overwriting virtual function tables
- compiler bugs - rare but my students and I have a talent for finding them
  - try -O, -O2, -Os
- wrong version of a shared library gets loaded
  - make the runtime linker be verbose

Nasty Problems

- Difference in MS "debug/development" stacks vs performance/runtime stacks
- OS / library bugs
  - create small examples
  - verify preconditions
  - check that postconditions fail
- static initialization problems with C++
- Processor bugs

some fun (simple) bugs

– In java (or whatever):

```
public void foo(int p, int q) {
  int x = p;
  int y = p;
}
```

– In perl:

```
$foo = $cgi->param('foo');
if (!$foo) {
  webDie ("missing parameter foo!");
}
```

– In C:

```
char *glue(char *left, char sep, char *right) {
  char *buf = malloc(sizeof(char) *
                (strlen(left) + 1 strlen(right)));
  sprintf(buf, "%s%c%s", left, sep, right);
  return buf;
}
```

Tricks

- write a custom assert()
- stub routines on which to break
- dump to a file instead of standard out
- make rand() deterministic by controlling the seed
- fight memory corruption with tools that use sentinels, etc. (and if no tools do it yourself)

The Future of Debugging

- better debuggers and programs to help you visualize your programs state
- simple model checkers
- programs keep getting bigger, finding bugs is going to get harder!
- Parallel/distributed debuggers as we move to more parallel/distributed systems.