Better Bug Hunting

Improving the search for bugs in software

Roger Orr
OR/2 Ltd

ACCU Conference - 2007

Overview

- Why do we need to improve?
- The problem solving process
- Tools of the trade
- Making it easier

Why do we need to improve?

- Programming is hard
- Despite the passage of time there is little sign of an end to bugs
- Some estimates are up to 40% of programmer time is spent on debugging

Learning on the job?

- Further studies on programmers find wide variation in their ability to find and fix faults
- Better programmers can be 20 times faster at finding defects
  - Find higher percentage of defects
  - Spend less time on each defect
  - Introduce fewer fresh problems

Learning on the job?

- Further studies on programmers find wide variation in their ability to find and fix faults
- Better programmers can be 20 times faster at finding defects
  - Find higher percentage of defects
  - Spend less time on each defect
  - Introduce fewer fresh problems
- *How can we learn to debug like an expert?*

What is a bug?

- “Anything that causes a user pain”
  - Inconsistent user interfaces
  - Unmet expectations
  - Poor performance
  - Crashes or data corruption
- These are the *symptoms*

What is a bug?

- The software has a *defect*
- This may cause an *infection* in the program
- The infection may result in a *failure*

What is a bug?

- The software has a *defect*
- This may cause an *infection* in the program
- The infection may result in a *failure*

- The word 'bug' is imprecise
  - It can mean any or all of the above!
  - A bug denies my responsibility
  - A bug may not sound serious

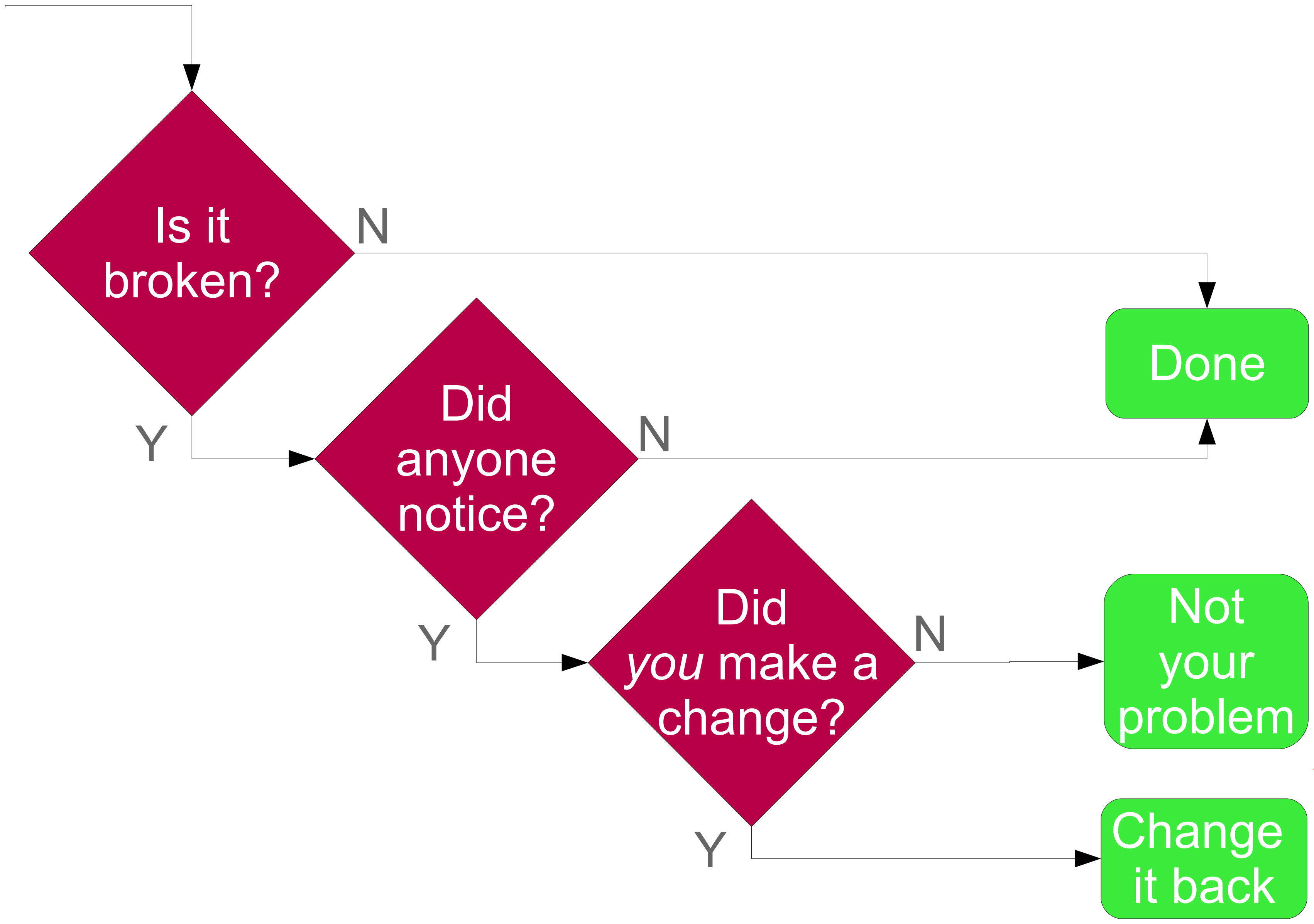
What is a bug?

- Not all failures fall into this pattern.
- Two common alternatives are
  - The software has a widespread *flaw*
  - The software has encountered an *external* problem. (The failure may be inevitable)

Problem Solving

- There are a number of methodologies for *design* and *coding* that most programmers use
- So why do so few programmers have a clear strategy for fixing bugs?

Problem solving flowchart



Problem Solving

- David Agans - www.debuggingrules.com

- Understand the System
- Make it Fail
- Quit Thinking and Look
- Divide and Conquer
- Change One Thing at a Time
- Keep an Audit Trail
- Check the Plug
- Get a Fresh View
- If You Didn't Fix It, It Ain't Fixed

Problem Solving

- Andreas Zeller – Why Programs Fail

- **TRAFFIC**

- **T**rack the problem
- **R**eproduce the failure
- **A**utomate and simplify
- **F**ind infection origins
- **F**ocus on likely origins
- **I**solate the infection chain
- **C**orrect the defect

Problem Solving

- Joe Programmer – at large

- Hope it goes away
- Blame someone else
- Open a debugger and poke around
- Try random changes
- Remove the symptoms
- Forget the whole experience

First Things First

- The most important thing to do when a problem occurs is to capture the program state.
  - What were you doing?
  - How did you get there?
  - Which machine/database/user/etc?
  - What version of the code?
- If you don't capture the state you'd better hope the problem happens again!
- Better programs collect this for you :-)

Why is this first?

- Correlate with other bugs in the bug database – may detect patterns, or find it is already being worked on.
- Some vital clue might be lost - like the 'scene of the crime' in a detective story
- The longer you leave it the more you forget
- Helps with the next step...

Play it again, Sam

- This is a mis-quote – remember 'first things first'
- Try to reliably reproduce the problem
- Ideally make it fail automatically
  - Failing again – or not - may show you something
  - A reliable way to repeat the fault lets you test hypotheses
  - Enables you to prove the fault is fixed
- TDD adds a test that fails – if possible

Reflect

- What other information do you need?
- Can you make a hypothesis?
- What alternatives are there?
- What experiments can you make to confirm or deny your ideas?

How long is the string?

- A fault may be obvious
- If not, try to partition the problem space –
  - Remove code that “can't” be involved.
  - Compare a “good” run with a “bad” run
- Keep all your changes reversible
- With luck you will find the faulty code

May you live in interesting times

- Locating the fault is hard when it is:
  - Not reliably reproducible (or very manual)
  - Non-local (eg C++ memory corruption)
  - Timing related (eg network, threads)
  - Environment related (eg permissions)
  - Not the fault you hypothesised

May you live in interesting times

- Alternatives include
  - Add tracing (or even video recording)
  - Rewrite the code
  - Run under an emulation/sandbox
  - Deliberately perturb the code
  - Use static analysis

So what's wrong?

- You have found the faulty line(s) of code
- Don't break the good cases
  - Duplicate the line, verify old matches new
  - Unit tests with a full set of data
- Prove the fix does remove the problem, and taking the fix out does put the problem back

Dotting the 'i's ...

- What was the fault? – eg syntax, failure to understand an API
- Does the *same* fault exist elsewhere (copy and paste is evil)?
- Does a *similar* fault exist elsewhere?
- Do I need a tool to find other places?

Dotting the 'i's ...

- What was the fault? – eg syntax, failure to understand an API
- Does the *same* fault exist elsewhere (copy and paste is evil)?
- Does a *similar* fault exist elsewhere?
- Do I need a tool to find other places?
- How will I avoid this fault in the future?

... crossing the 't's

- What can I do to make this class of bug:
  - Easier to find?
  - Easier to fix?
  - (More on this later)
- Have I updated the bug database?

The view from above

- What can be factored out from this process?
  - Capturing the state
  - Reducing the search space
  - Finding anomalies
- Use general purpose tools
- Add diagnostic code
- Use the *computer* as a tool to debug

Capturing state (first things first)

- Consider writing a program (or script) to capture environment information:
  - Versions of code, libraries, 3rd party and O/S
  - Environment variables, usernames, permissions
  - Other processes, CPU/memory/disk usage
  - Logs, screen shots
- Nothing gets forgotten
- Consistent format
- Automate, automate, automate...

Capturing state

- Unix core dumps, Windows minidumps
- Can be extrinsic or intrinsic
- May be crash handlers and/or exception handlers
- Think about how you'll use these files
  - Where to save them, how to send them
  - What do you need to unpack them
- Can you do any useful extraction in-place

Capturing state

- Program's 'black box' flight recorder.
- Code within the program may be better equipped to extract information (eg request packets for grid computing)
- Can write a call stack – which can be enough to solve some bugs all by itself.

Stack traces

- Many environments provide stack traces as a matter of course (eg .NET, Java)
- Knowing how you got there dramatically improves fault finding
- Don't lose the stack trace – write a fault handler.
- C++ in many environments needs a little bit of work...this is well worth putting in.

Sample stack trace: g++

Use the `backtrace` function

```
#include <execinfo.h>

void u_sigsegv(int sig, siginfo_t *sigi, void *unused ){
  cerr << "segfault at: " << sigi->si_addr << endl;

  size_t const max_addrs(16);
  void *addrs[ max_addrs ];
  size_t size = backtrace( addrs, max_addrs );
  char ** frames = backtrace_symbols( addrs, size );

  for ( size_t idx = 0; idx != size; ++idx ) {
     cerr << idx << ": [" << addrs[ idx ] << "] "
          << frames[idx] << endl;
  }
  free( frames );
}
```

Sample stack trace: MSVC

Use the dbghelp engine to get stack addresses

```
#include <imagehlp.h>

  // Get current thread context, set up stackFrame

  SymInitialize( GetCurrentProcess(), 0, FALSE );

  // Skip both GetThreadContext and ourself.
  for ( frame = -2; frame < size; ++frame ) {
    if ( ! StackWalk( IMAGE_FILE_MACHINE_I386,
       GetCurrentProcess(), GetCurrentThread(),
       &stackFrame, &context, 0,
       SymFunctionTableAccess, GetModuleBase, 0 ) )
      break;

    if ( frame >= 0 ) {
      array[frame] = (void*)stackFrame.AddrPC.Offset;
    }
  }
  SymCleanup( GetCurrentProcess() );
```

Sample stack trace: MSVC

- The dbghelp engine can also be used to get names using SymGetSymFromAddr()
- You can also obtain access to data types, local variables, etc. - see the Debugging Tools for Windows at: www.microsoft.com/whdc/devtools. (Use custom install to get the SDK)

Capturing state – log files

- Many programs write logs of their execution
- Aim for consistent usage
- Using logging frameworks (eg Log4j) allows for powerful configuration - at a small cost
- Log files should:
  - Be easy to find
  - Include timestamps
  - Be truthful

How to mislead

Command

```
C:>deploy add -p grid_service.tar -a grid_profile.xml
```

Output log

```
Verifying the application package, grid_service.tar.
Verification failed with the following error:
Domain <Application>: The application failed to install.
The target application directory already exists.
```

How to mislead

Command

```
C:>deploy add -p grid_service.tar -a grid_profile.xml
```

Output log

```
Verifying the application package, grid_service.tar.
Verification failed with the following error:
Domain <Application>: The application failed to install.
The target application directory already exists.
```

The actual problem was with temporary directory used for expanding the tar file

How to mislead (2)

Command

```
C:>regtlb32 MyTypeLib.tlb
```

Output log

```
An error occurred! File does not exist.
Win32 Error = 32
```

But actually...

```
"The process cannot access the file because it is being used
by another process"
```

Capturing state – log files

- Many programs write logs of their execution
- Aim for consistent usage
- Using logging frameworks (eg Log4j) allows for powerful configuration - at a small cost
- Log files should:
  - Be easy to find
  - Include timestamps
  - Be truthful
  - **Be helpful**

Capturing state – log files

- Think about the log messages you write
- Distinguish simple information from errors
- Add specifics:
  - Error codes or exception class names
  - File names, arguments to API calls
  - Context of the fault
- It is easy to see which is more help:
  - “file error”
  - “error 2 (file not found) opening c:\config.xml”

Debuggers

- Debuggers don't de-bug – people do

Debuggers

- Debuggers don't de-bug – people do
- Interactive program examination
- Time intensive
- Hard to reproduce a debugging session
- However “given a choice between two languages, choose the one with the better debugger” - Donald Knuth

Debugger tips

- Always approach the debugger with a hypo-thesis to explore.
- Time limit your interactive exploration

Debugger tips

- If you can, always build with symbols
- With Microsoft and g++ can do this even for optimised builds
  - Use Microsoft symbol servers, push for 3rd party
- Do you need to optimise *all* your code?

Debugger tips

- Learn to extend your debugger:
  - Learn how to script it
  - Write functions callable from the debugger
  - Variable expansion (e.g. AUTOEXP.DAT)
  - Can you write and/or read dump files
- Use more than one debugger
  - For example WinDbg command "`!locks`"
- Make use of the debug interface
  - Debuggers do not work by magic!
  - Use the same API to automate tasks

Multi-tasking

- Many debuggers exhibit 'male traits' – they don't multi-task well.
  - Some debuggers are hard to operate with multiple threads and processes
  - Almost all debuggers seriously interfere with the execution of multi-tasking processes
- Can you identify your threads?

Cleverer debuggers

- Some debuggers keep history
  - 'Omniscient' debuggers[1]
  - UndoDB (http://undo-software.com/)
  - Emulators
- War story – 'Atron probe'

1 http://www.ddj.com/dept/debug/184406101?pgno=1

Tools

- The debugger is *a* tool for debugging, not the *only* one.

Tools

- The debugger is *a* tool for debugging, not the *only* one.
- Other tools include:
  - Static analyzers
  - Tracing tools
  - Proxies
  - Stubs

Tools - Static

- Static analysis can identify faults in the source code.
- Always turn on all compiler warnings (and use more than one compiler).
- In my experience there are many false positives – filtering is necessary
- Easier to use a tool from the *start* of the project

Tools - Tracing

- Tracing can be particularly useful across a boundary – helps with dividing your problem space.
- Examples in Web development are obvious
  - FireBug's 'Net' tab shows all the http traffic with timings.
  - Tcpmon a stand-alone tool to do the same.
  - Write your own for special needs (for example, protocol analysis or formatting)

Tools - Tracing

- Tracing inside the process can be useful
- The interface may be that between the program and the OS, for example
  - strace/truss on Unix
  - Apimon/strace/ntTrace on Windows
- Tracing a subset of calls
- Tracing allows post-processing of output
- Can compare 'good' and 'bad' runs
- Whole machine traces can be useful, when possible

Tools - Tracing

- A large number of tracing tools cover access to memory – Purify, valgrind, Developer Partner Studio
  - Act more like a proxy to memory
  - Tend to be very invasive
  - Often fail with larger applications
- Some tracing tools are focused on other areas – for example FileMon from the SysInternals team monitors file access.

Tools - Proxies

- Proxies wrap up part of the system and often provide an instrumented interface
- Some proxies are just used for tracing
- A proxy can perform logic analysis and maintain internal state

Tools - Stubs

- A stub replaces part of the application, usually to provide simpler and/or more deterministic behaviour
- May be produced as an artifact of testing

Another step back

- “Intellectuals solve problems. Geniuses prevent them.”
  (Albert Einstein)
- Prevent bugs
- Anticipate bugs

Bug free design

- The easiest bugs to fix are the ones that were never written.
- Techniques like pair programming, code reviews should be used to reduce the number of bugs.
- However, experience shows that, despite all our efforts, some bugs will *always* get through.

Debug Driven Design

- Design programs to be debugged easily
  - we *are* going to have to debug them
- 'Debug' or 'Test' driven design - what is the difference?
  - Not mutually exclusive but complementary
  - Failed tests still need debugging
  - In my experience TDD helps less with the really 'exciting' bugs
    - related to timing or environment
    - caused by incorrect assumptions

What does a failed test prove?

- The primary purpose for a test is to provide a pass/fail result.
- A failure may tell you nothing about *why*
- Compare these two uses of Junit :
  - fail();
  - assert(“Checking size”, expectSize, obj.size());
- Note: the problem is wider than just unit testing.

DDD

- How can we write programs that help us debug them?
- Reflect on what questions you will need answering to diagnose likely problems
- Can we detect the infection earlier?
- Can we catch the fault?

DDD

- Some elements of debug-driven design
  - Clean interfaces
  - Error handling policy
  - Consistent logging
  - Validate assumptions
  - Debugging hooks

Clean interfaces

- Easier to narrow down the code at fault
  - Eliminate components with valid data
  - Explore interaction at the boundary
  - Proxies/tracing are more effective
  - Perimeter fences can catch problems

Clean interfaces

- Easier to write tests
  - Fewer dependencies, can test in isolation
  - Simpler interactions, less to test
  - Shorter distances between fault and symptom
- The versioning problem – it *will* happen
- Self-describing protocols are safer and more flexible, but may be less efficient

Error handling policy

- What to do in response to an error?
- There are two forces in opposition
- Your customers want ...
  - No loss of data
  - Application to keep going
- Developers want ...
  - To preserve the program state
  - Capture the steps leading to the fault

Consistent logging

- How?
  - Which framework, how to configure
- Why?
  - Does this information add anything to the log?
- What?
  - Specifics – names, error codes, values
- Where?
  - Standard location for log files, tools to access
- Who?
  - User, process, thread, function, line
- When?
  - Timestamps – allow correlation

Validate assumptions

- Programs (and programmers) make many assumptions – also including the program environment and how the code will be used
- Some can be validated
  - Assert (design constraints)
  - Invariant testing
  - Simple self-test for installation problems
  - Test assumptions early and choose a good policy on failure
  - Finding infections beats finding symptoms

Debugging hooks

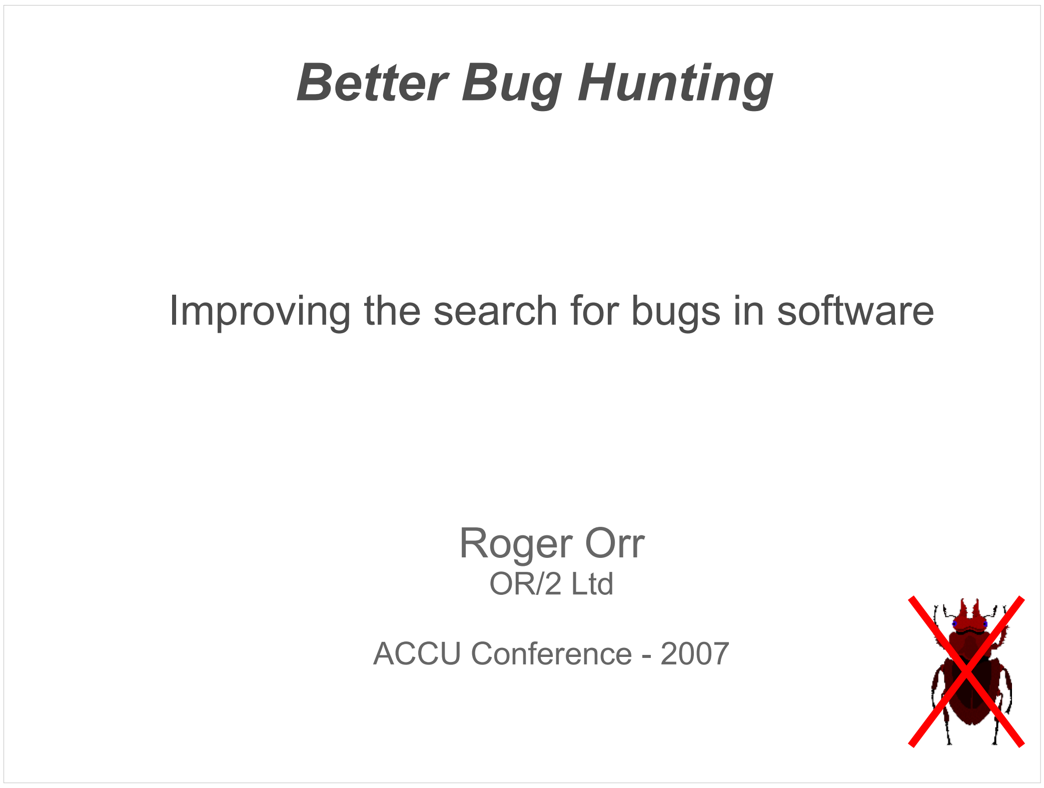
- Many modern integrated circuits have comprehensive debug ports built in
- Many instruction sets have debug control registers
- Software components can do likewise – means you can test with the same code that is in production

Debugging hooks

- Pluggable listeners
- Filters
- In-memory history buffers
- Probes to obtain statistics
- Access to intermediate values
- 'Ping' functionality
- Debug assistance functions

Conclusion

- Bugs are going to occur
- Adopt a strategy for hunting bugs
- Use the right tools
- Design code to be debugged



Better Bug Hunting

Improving the search for bugs in software

Roger Orr
OR/2 Ltd

ACCU Conference - 2007

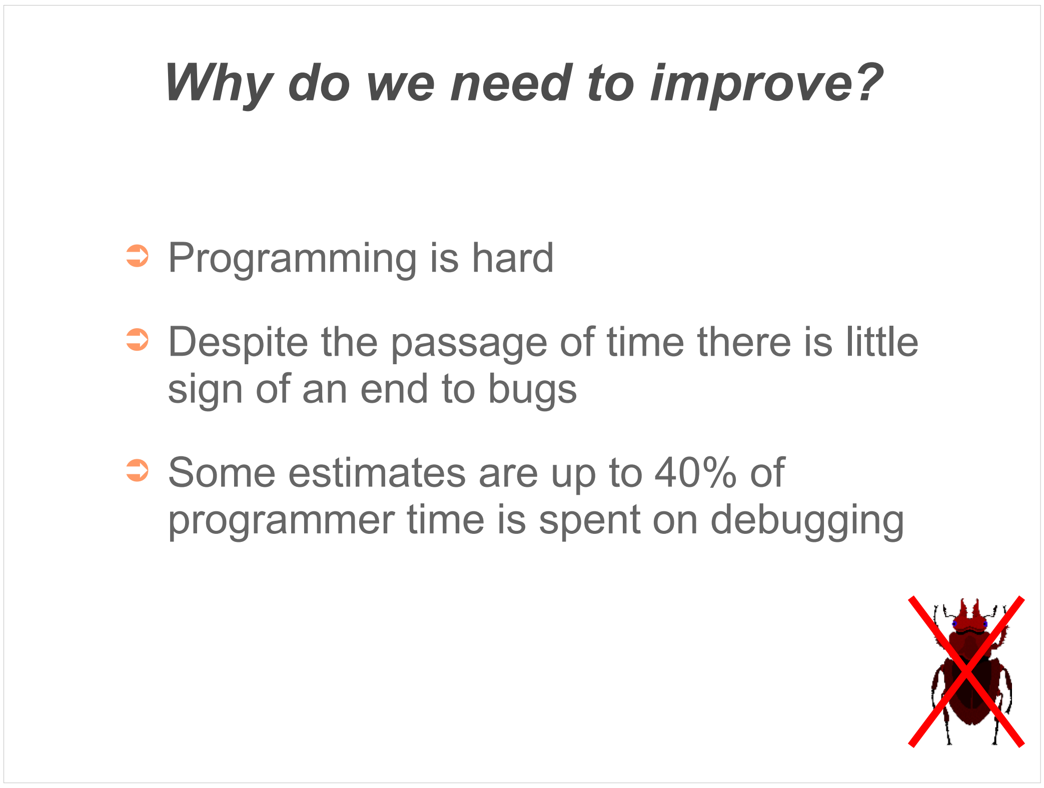
We spend a lot of time as developers searching for bug and removing them from our programs - how can we optimise the hunt?

Why is bug hunting hard, and what makes it seem to be hard to improve? Are there good techniques to follow and bad ones to avoid? Why do some programmers find bugs an order of magnitude faster than other people?

In this session I would like to answer some of these questions with a view to sharing some techniques on how to hunt successfully for bugs.

Overview

- Why do we need to improve?
- The problem solving process
- Tools of the trade
- Making it easier



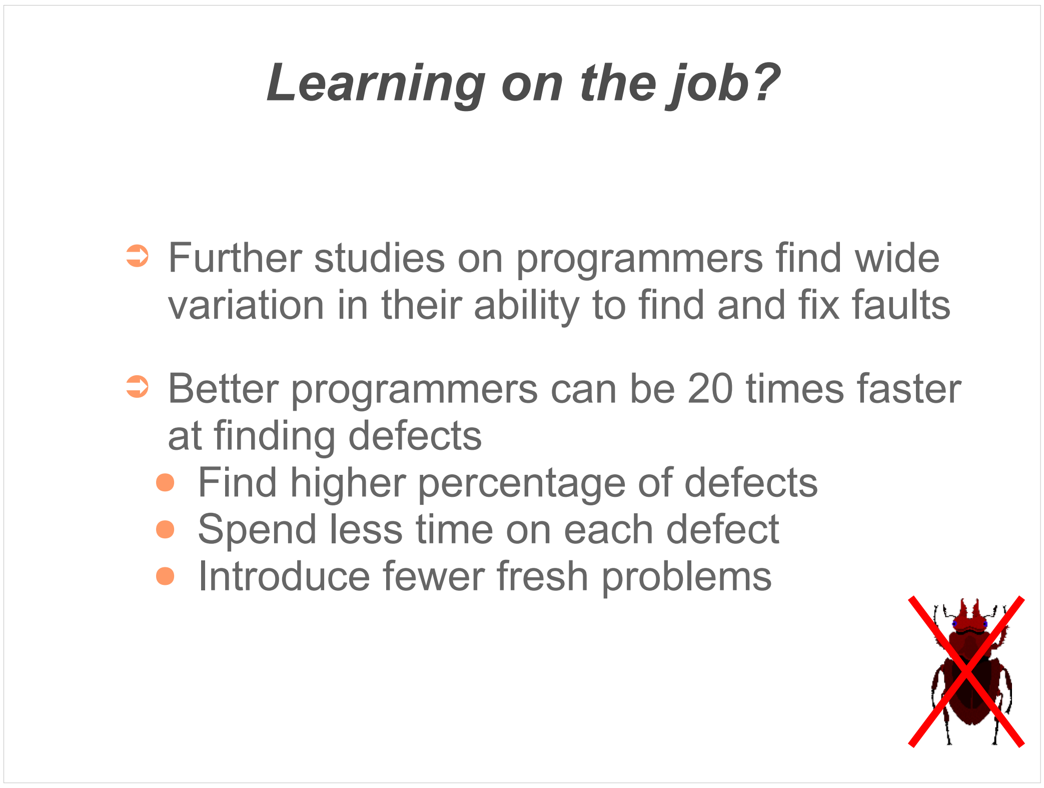
Why do we need to improve?

- Programming is hard
- Despite the passage of time there is little sign of an end to bugs
- Some estimates are up to 40% of programmer time is spent on debugging

Most programmers live in 'bug denial'. We are not expecting to experience problems with our code, and we do not approach the task of fixing them with any kind of structured approach.

We receive far less education on how to fix problems than on how to design and write code.

There are relatively few good books on debugging.



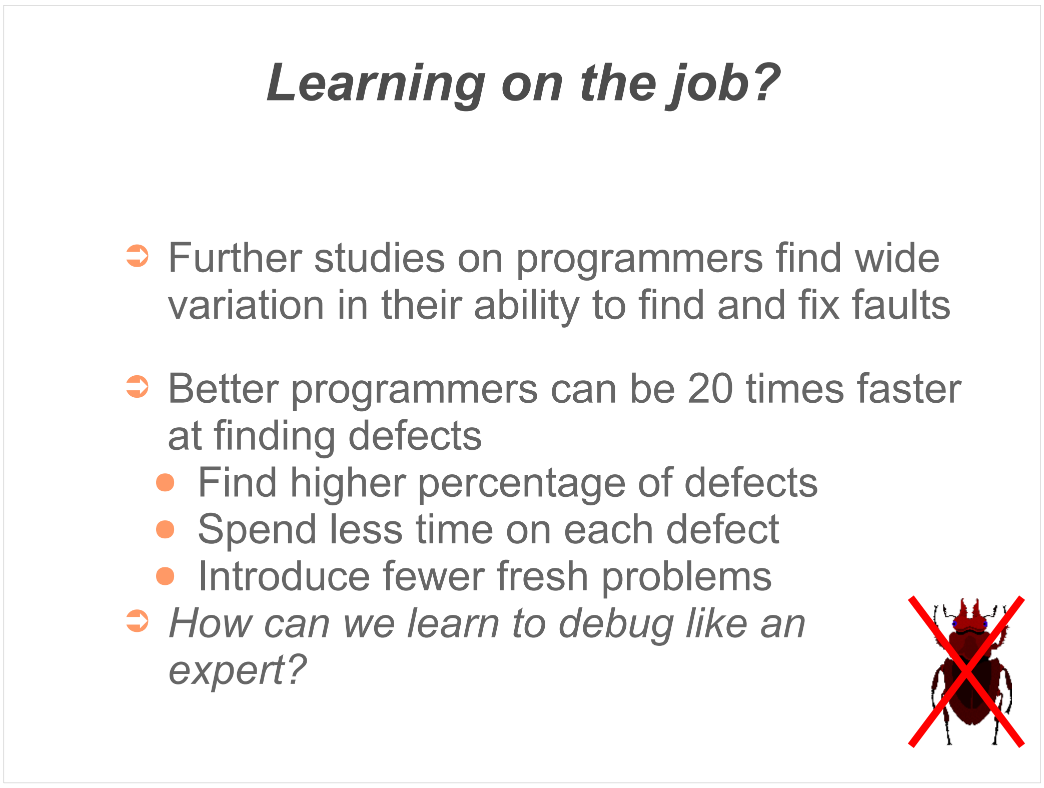
Learning on the job?

- Further studies on programmers find wide variation in their ability to find and fix faults
- Better programmers can be 20 times faster at finding defects
  - Find higher percentage of defects
  - Spend less time on each defect
  - Introduce fewer fresh problems

You might not expect a great deal of difference between the way different programmers approach the task of hunting bugs.

However, the indications are that there is a large difference between the good and bad problem solvers.

This costs projects a lot – what you have doesn't work and programmers fixing faults are not adding new functionality either.

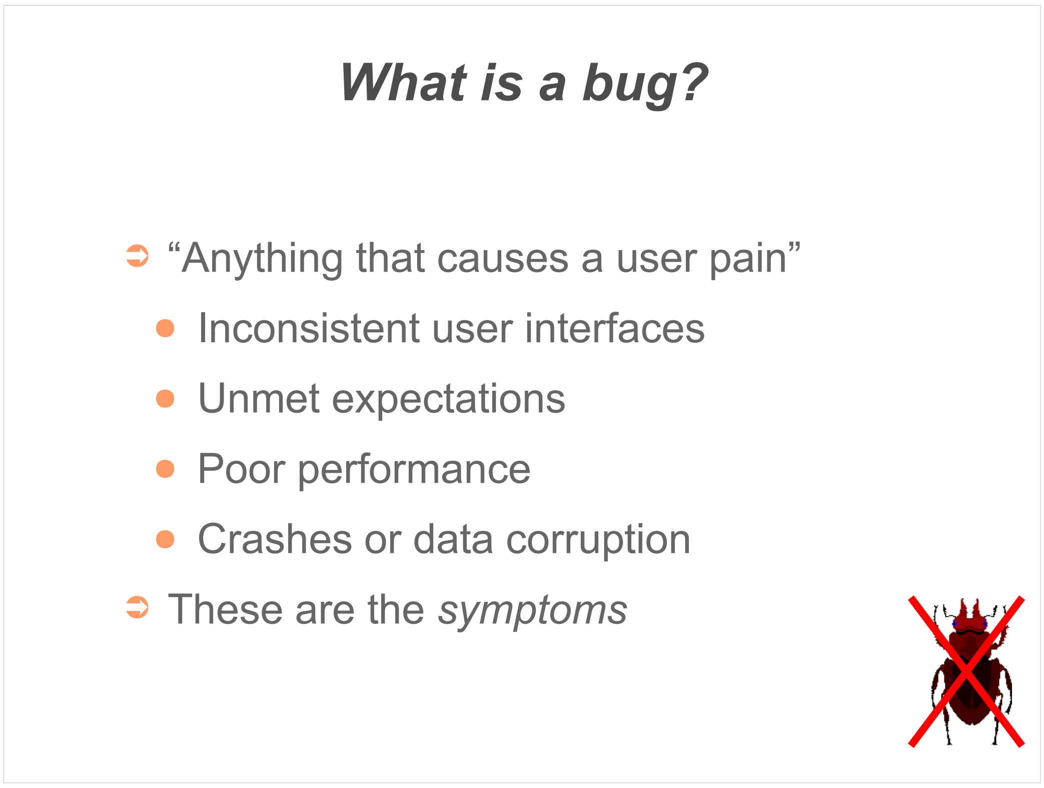


Learning on the job?

- Further studies on programmers find wide variation in their ability to find and fix faults
- Better programmers can be 20 times faster at finding defects
  - Find higher percentage of defects
  - Spend less time on each defect
  - Introduce fewer fresh problems
- *How can we learn to debug like an expert?*

Debugging can seem like a mystery, but much of it can in fact be learned.

It appears that the expert debuggers have learned good problem solving skills, often in an ad-hoc fashion, and these skills can be codified.



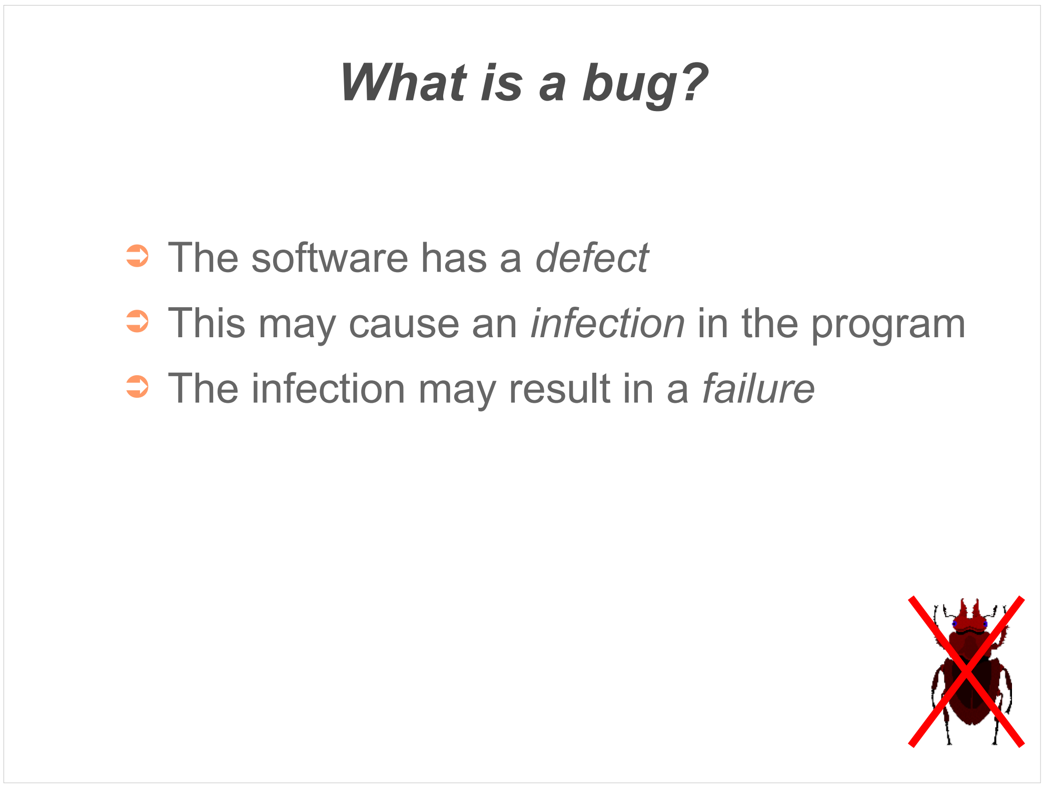
What is a bug?

- “Anything that causes a user pain”
  - Inconsistent user interfaces
  - Unmet expectations
  - Poor performance
  - Crashes or data corruption
- These are the *symptoms*

There are many definitions of a bug – I'm going to follow John Robbins' definition which I like because it:

- is relatively broad
- focuses on the *impact* of bugs.

Most of the time we start with the symptoms and work back from them to find and fix the root cause.



What is a bug?

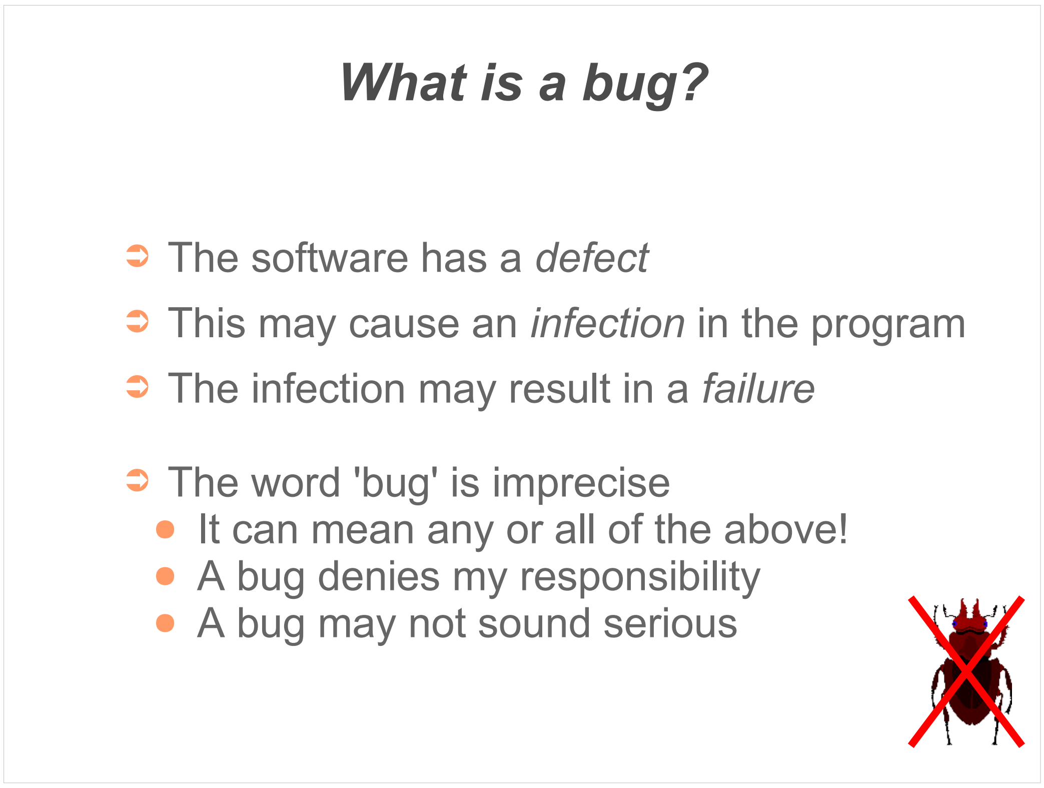
- The software has a *defect*
- This may cause an *infection* in the program
- The infection may result in a *failure*

Notice the data flow.

The root cause is a defect – but not all defects affect the running program.

The infection refers to the invalid state of the program – but not all invalid states cause the program to fail.

The failure is the end result – something unwanted and externally visible has occurred.



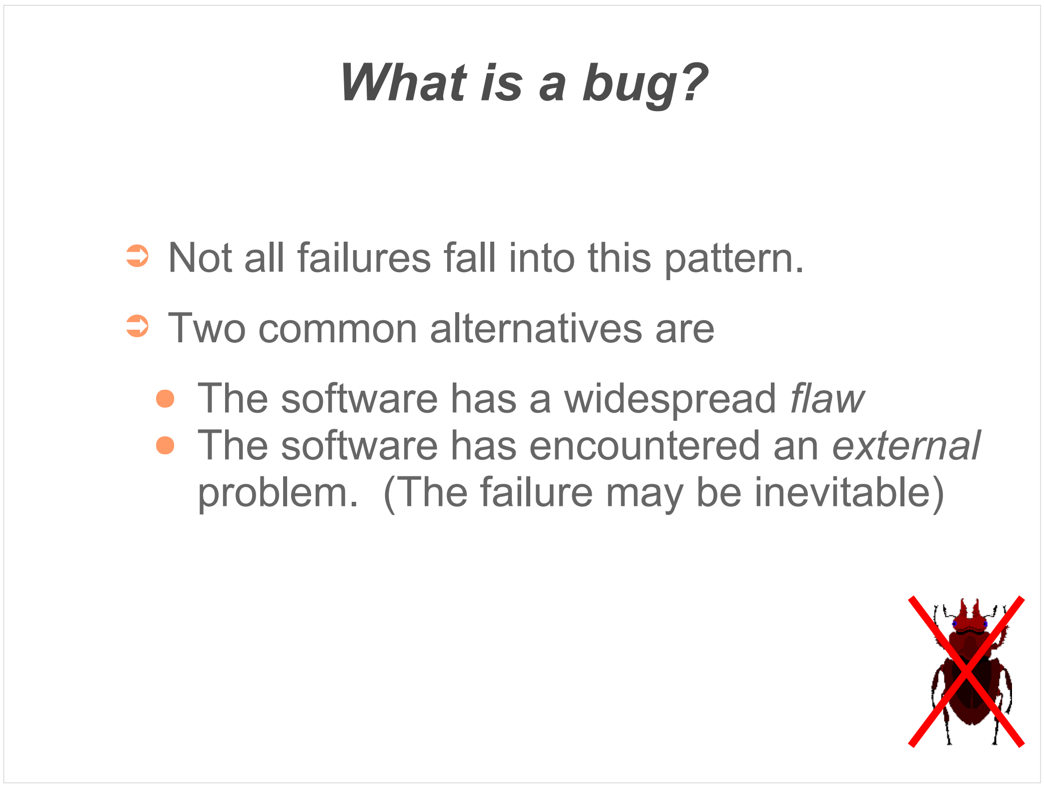
What is a bug?

- The software has a *defect*
- This may cause an *infection* in the program
- The infection may result in a *failure*
- The word 'bug' is imprecise
  - It can mean any or all of the above!
  - A bug denies my responsibility
  - A bug may not sound serious

The three italicised words are not the only names used.

Common synonyms are
- *fault* for *defect*
- *issue* or *problem* for *failure*

Take care over careless use of the word 'bug', it can lead to misunderstandings.



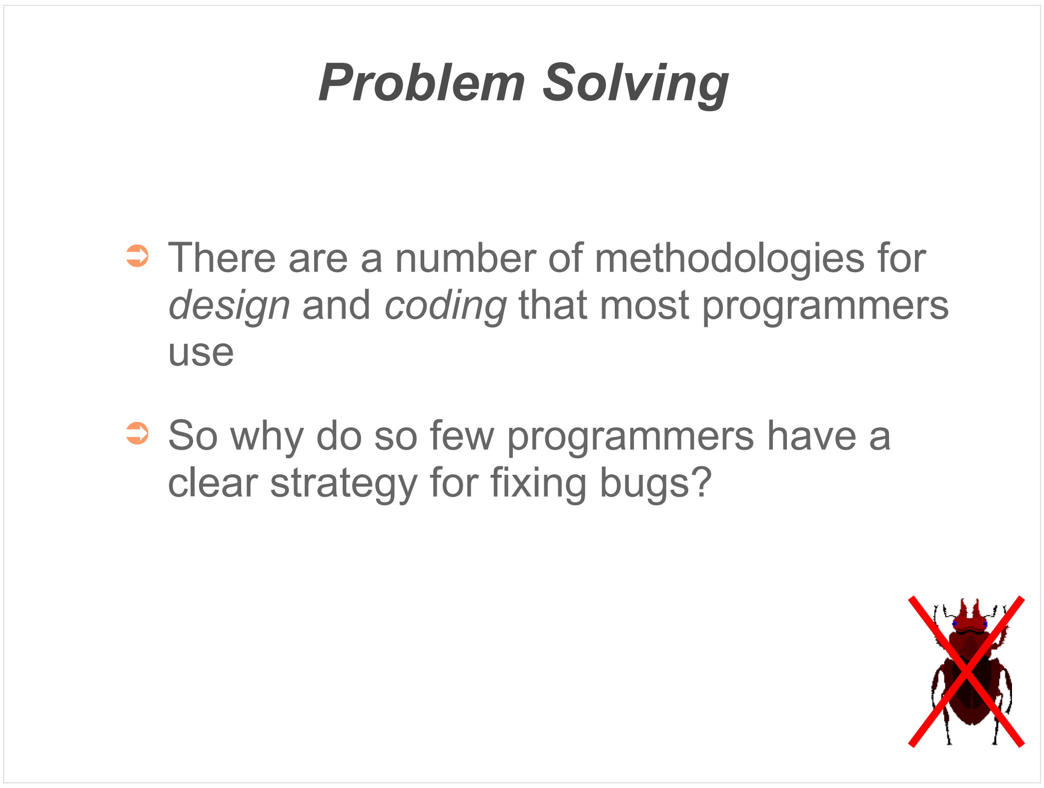
What is a bug?

- Not all failures fall into this pattern.
- Two common alternatives are
  - The software has a widespread *flaw*
  - The software has encountered an *external* problem. (The failure may be inevitable)

It can be tempting – but ultimately unhelpful - to try and categorise all problems with a single mechanism.

- A flaw may require a major rewrite (for example, converting non thread-safe code for multiple threads, or making a single-user program into a multi-user program)

- External problems present various issues. Can the problem be resolved by the program itself; should the problem have been detected earlier (eg installation or startup); or is it simply that the error needs to be reported clearly to the user for remedial action.



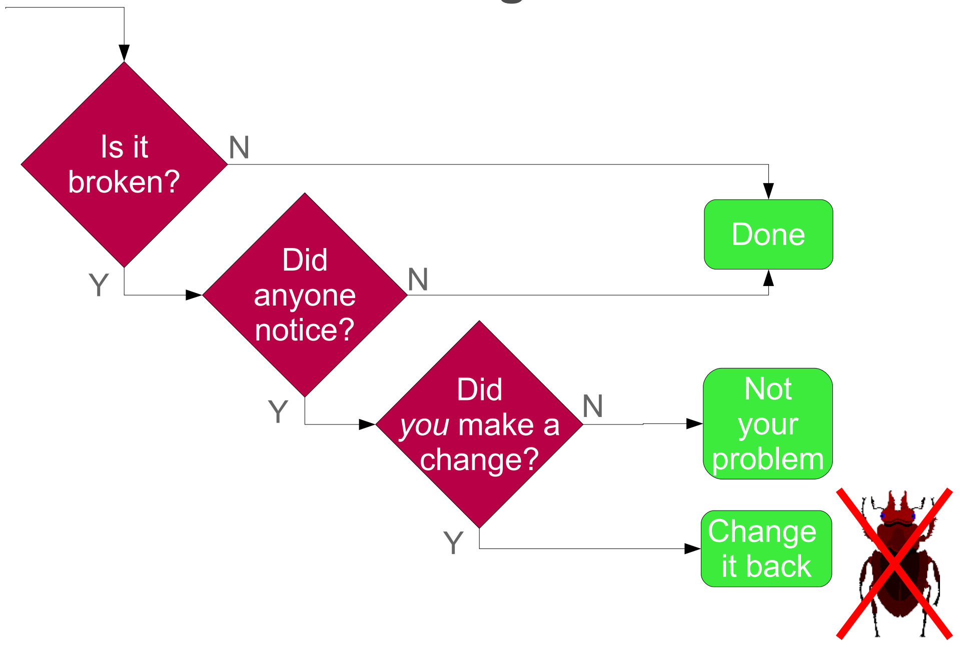
Problem Solving

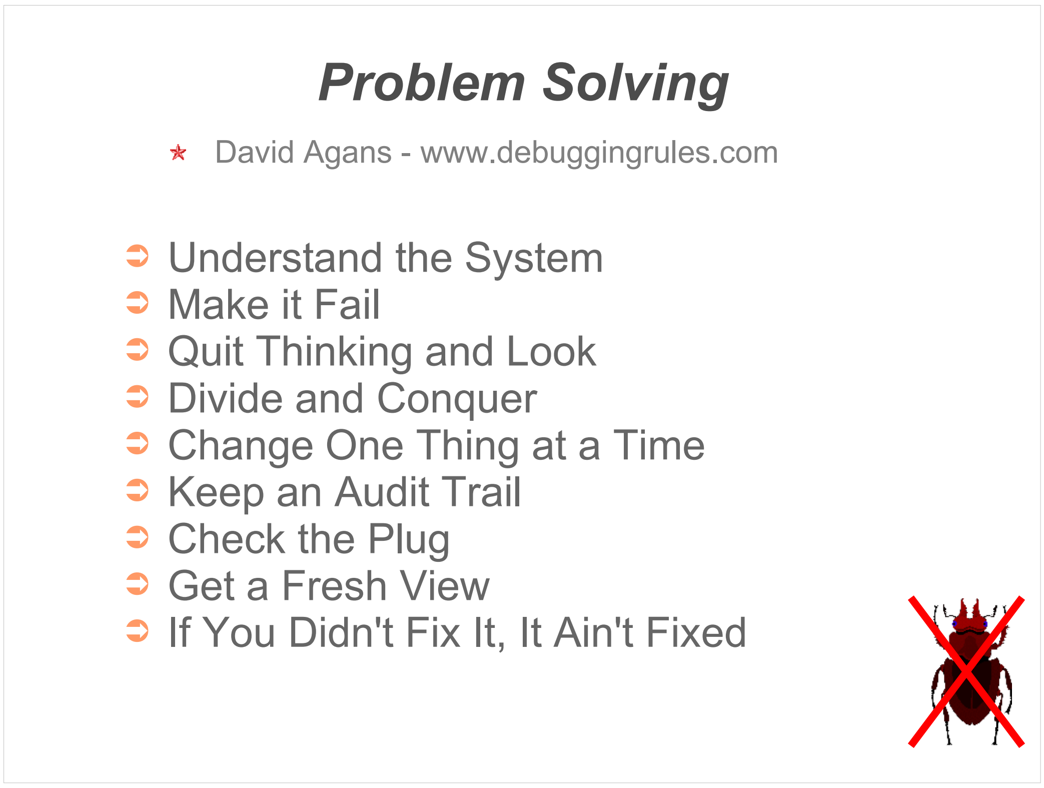
- There are a number of methodologies for *design* and *coding* that most programmers use
- So why do so few programmers have a clear strategy for fixing bugs?

There are numerous books describing various methodologies in use for many of the task of programing; few people just sit at the keyboard and type in their code.

So why, when a problem occurs, do the same programmers just sit at the keyboard and try to find the fault?

Problem solving flowchart





Problem Solving

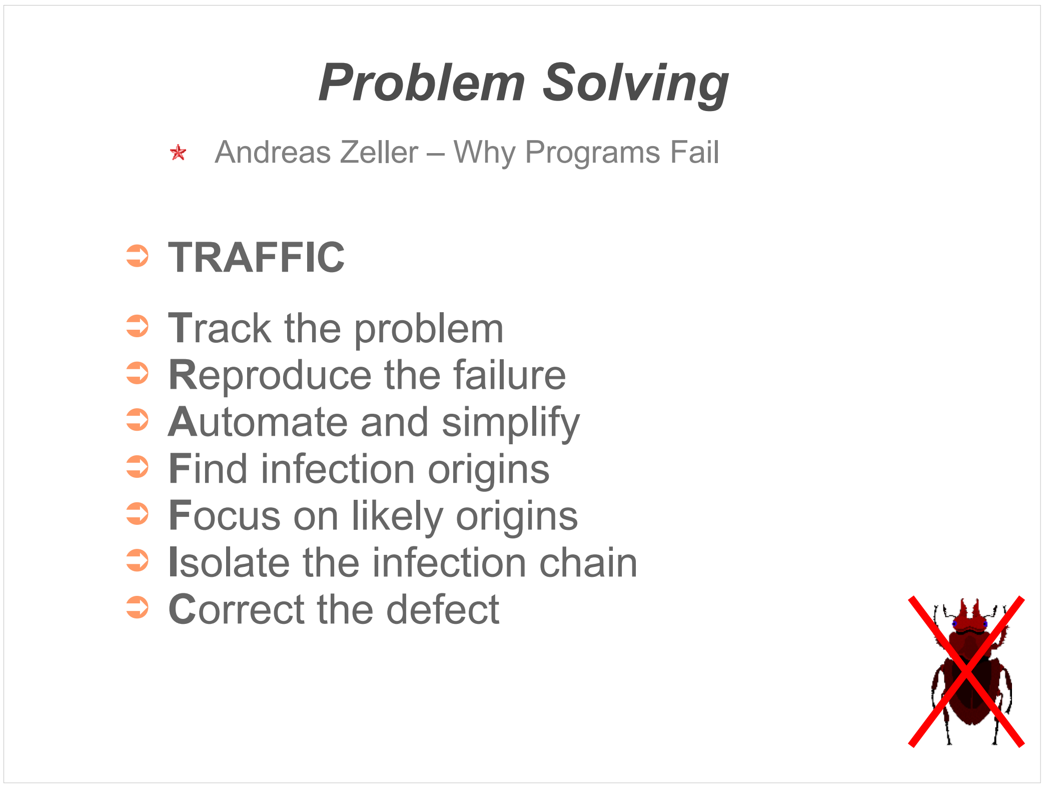
- David Agans - www.debuggingrules.com

- Understand the System
- Make it Fail
- Quit Thinking and Look
- Divide and Conquer
- Change One Thing at a Time
- Keep an Audit Trail
- Check the Plug
- Get a Fresh View
- If You Didn't Fix It, It Ain't Fixed

David Agans book is a fast introduction to debugging by someone who has obviously fixed a number of 'real world' problems.

“This book tells you how to find out what’s wrong with stuff, quick.”

He also provides a downloadable poster listing these nine rules.



Problem Solving

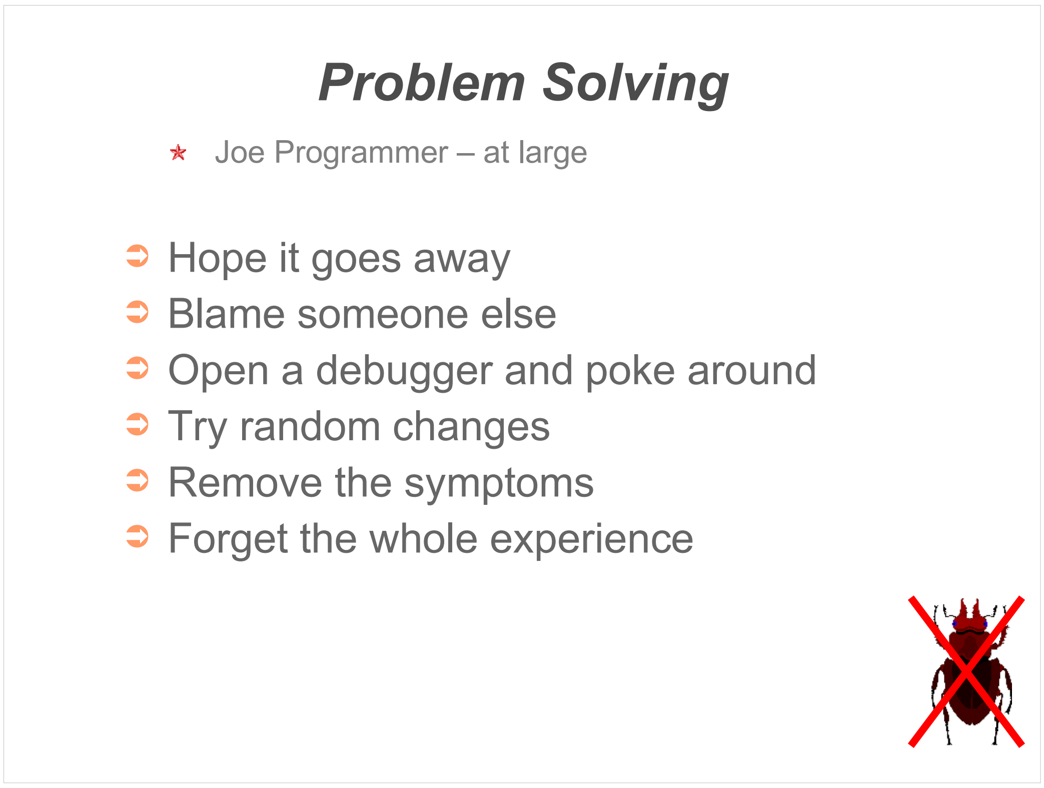
- Andreas Zeller – Why Programs Fail

- **TRAFFIC**

- **T**rack the problem
- **R**eproduce the failure
- **A**utomate and simplify
- **F**ind infection origins
- **F**ocus on likely origins
- **I**solate the infection chain
- **C**orrect the defect

Andreas' book is a relatively academic look at finding the reasons for programs failing. He is keen on the use of tools, and gives examples of various Unix techniques for finding and fixing faults.

He also stresses the need for a systematic approach to problem solving.



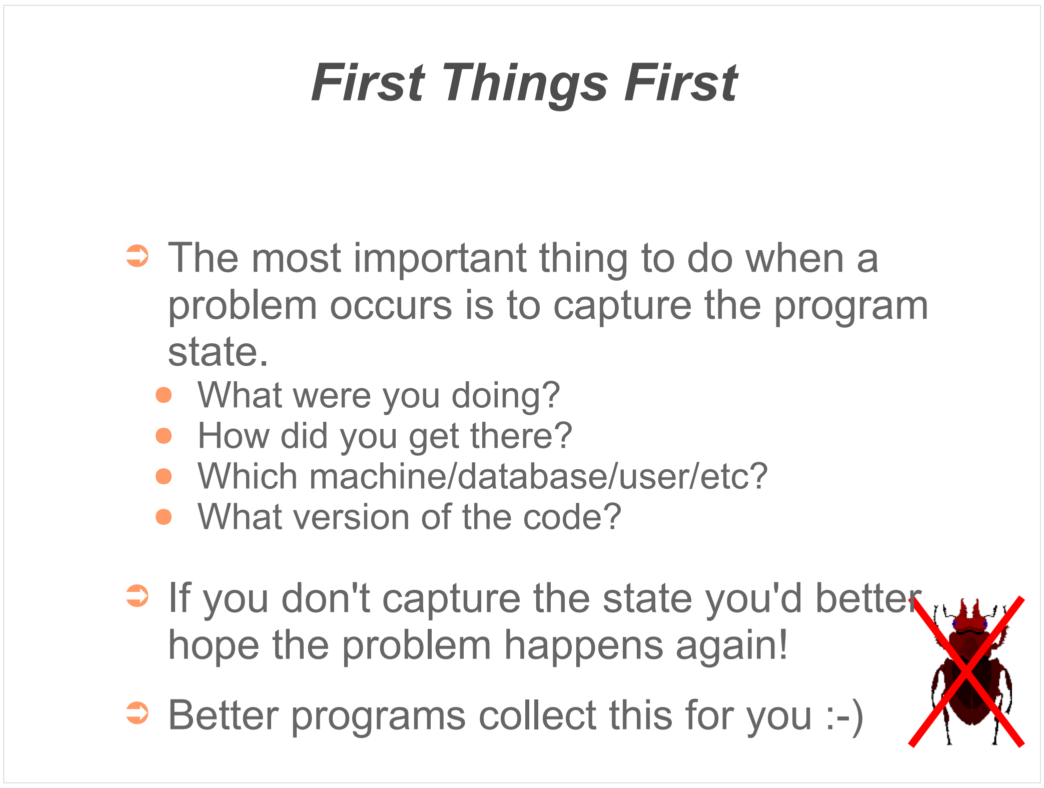
Problem Solving

- Joe Programmer – at large

- Hope it goes away
- Blame someone else
- Open a debugger and poke around
- Try random changes
- Remove the symptoms
- Forget the whole experience

I'm sure you can add your own war stories of the ad-hoc strategies used by your programmer friends.

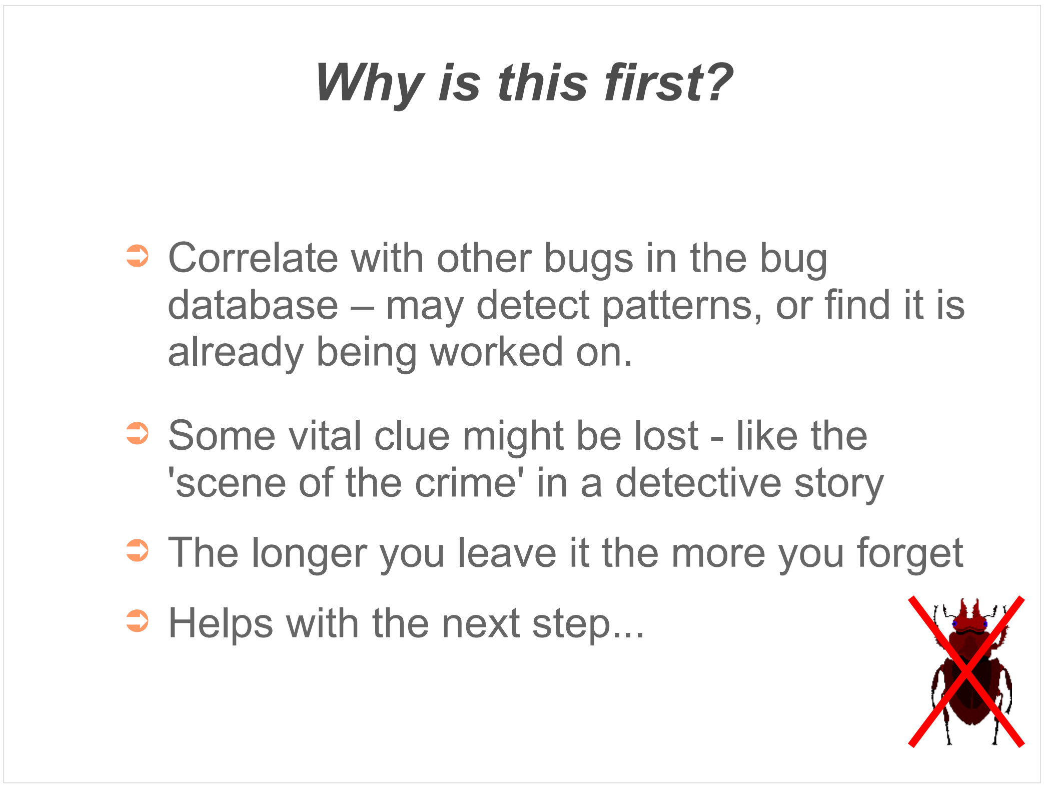
When written down like this it is easy to see how haphazard the technique really is; unfortunately though it seems to remain at the number one slot (at least, based on my own observations in a number of different companies)



First Things First

- The most important thing to do when a problem occurs is to capture the program state.
  - What were you doing?
  - How did you get there?
  - Which machine/database/user/etc?
  - What version of the code?
- If you don't capture the state you'd better hope the problem happens again!
- Better programs collect this for you :-)

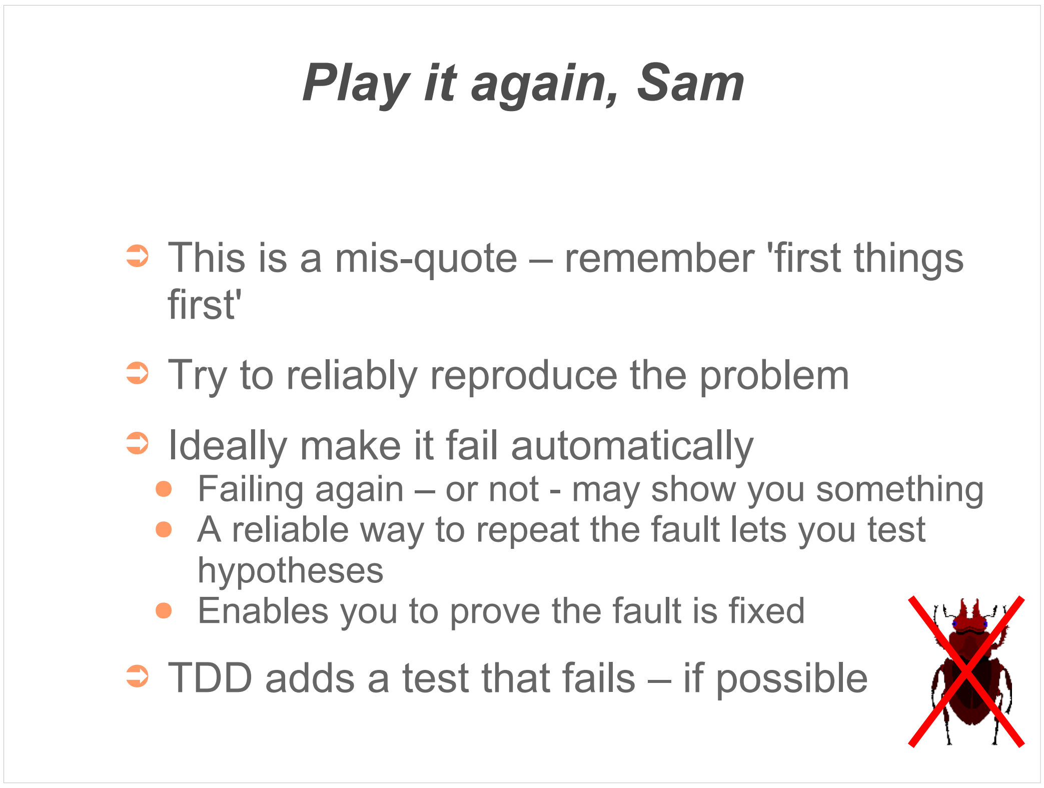
One problem with capturing the state of the program is that you do not know what might be relevant.
For example, a hardware fault on IBM mainframes only affected machines with orange doors (this finish had different electrical characteristics from the others)!
If you have the space, get everything you can think of – hopefully using a script. You can always discard data later, but you may not be able to verify a hypothesis if a vital piece of data is absent.



Why is this first?

- Correlate with other bugs in the bug database – may detect patterns, or find it is already being worked on.
- Some vital clue might be lost - like the 'scene of the crime' in a detective story
- The longer you leave it the more you forget
- Helps with the next step...

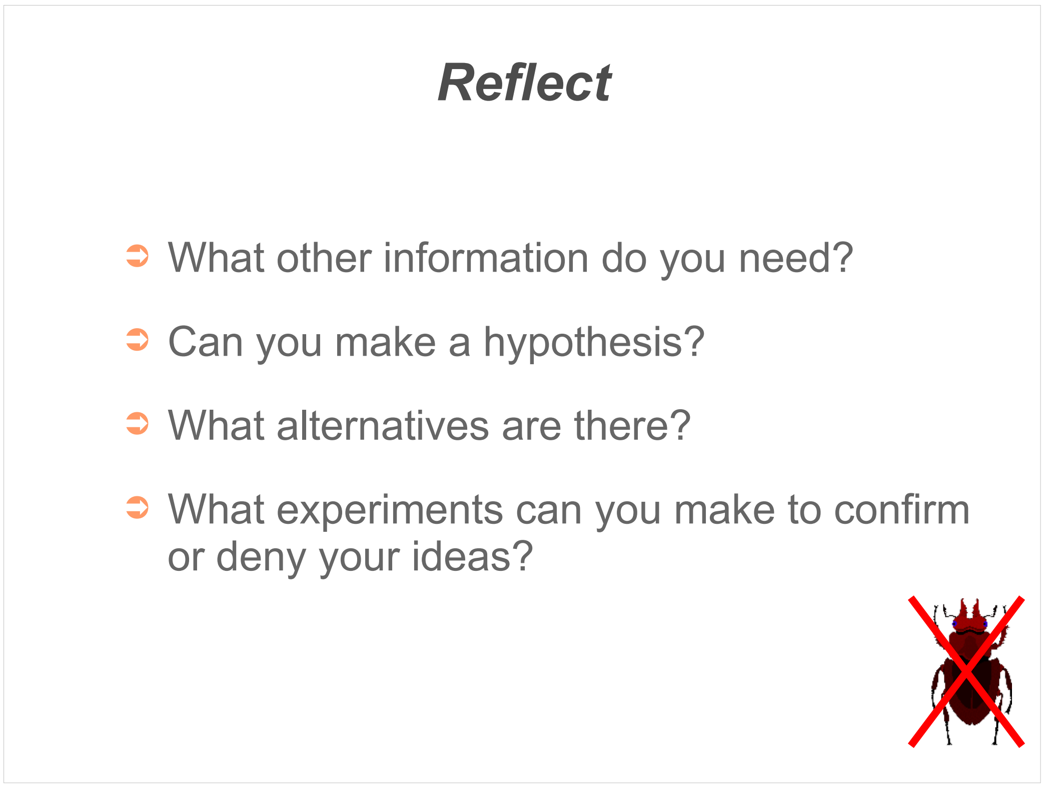
Our memory is frail – and hard to share with other people who may be involved in fixing the problem.
Think of the detective story – the incident team have a checklist of data to be gathered (photos, objects, fingerprints) and each item is dated and cross referenced. Sometimes we need the computing equivalent of the 'Police – do not cross' tape around the faulty machine to keep it from changing before we have extracted what we need.



Play it again, Sam

- This is a mis-quote – remember 'first things first'
- Try to reliably reproduce the problem
- Ideally make it fail automatically
  - Failing again – or not - may show you something
  - A reliable way to repeat the fault lets you test hypotheses
  - Enables you to prove the fault is fixed
- TDD adds a test that fails – if possible

If the problem cannot be reproduced, it has (almost certainly) not been fixed. However, it might be less urgent!
In some cases, reflecting on the symptoms and investigating the code can result in finding a possible cause. However, if you can't reproduce the bug you may not have found the cause of the bug that was experienced, and nor will you necessarily be certain your bug fix is correct.
You may well find that you can deduce (a) side effects that might be in the captured machine state or (b) a way to reproduce the fault more easily.



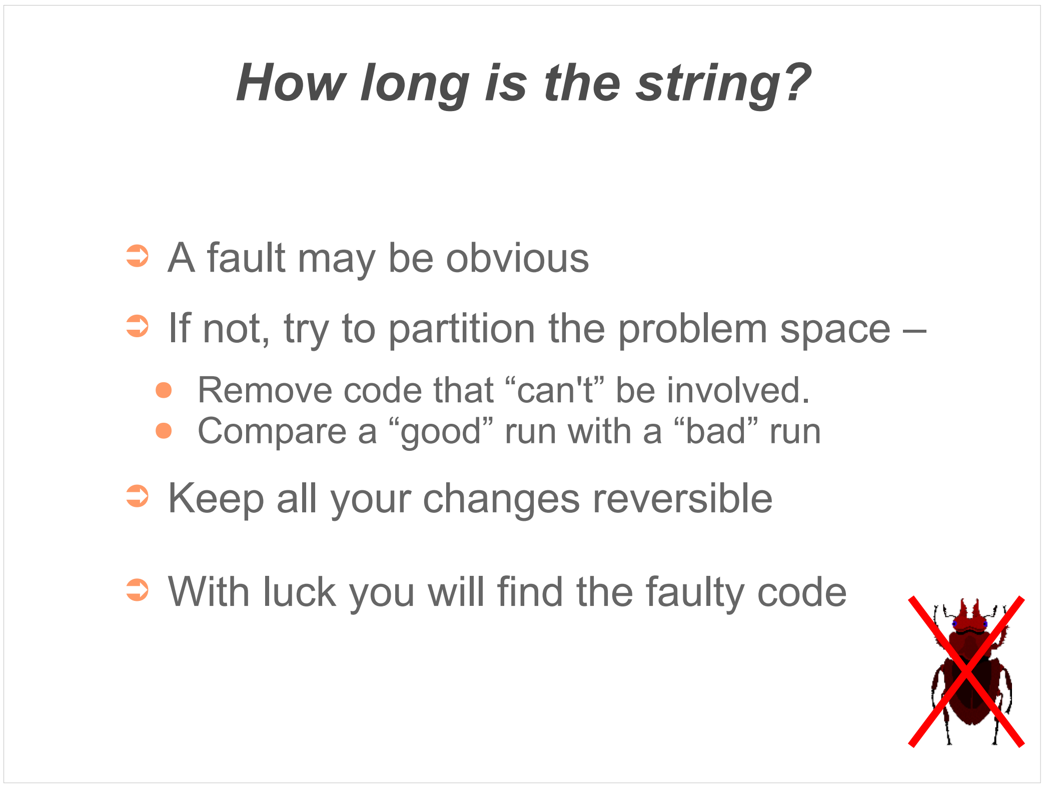
Reflect

- What other information do you need?
- Can you make a hypothesis?
- What alternatives are there?
- What experiments can you make to confirm or deny your ideas?

"Fools rush in where angels fear to tread."

Some bugs can be found quickly, but in many cases a bit of careful thought before acting will repay itself.

Avoid having just a single hypothesis – think creatively about what other possibilities there might be.



How long is the string?

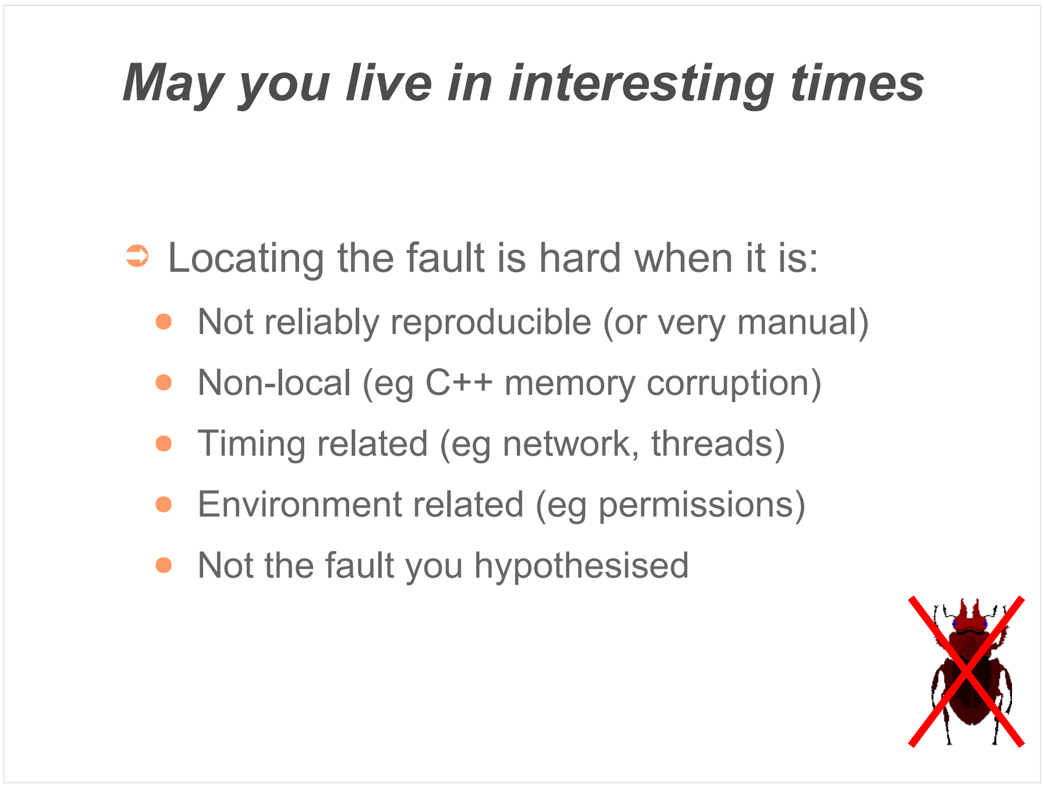
- A fault may be obvious
- If not, try to partition the problem space –
  - Remove code that “can't” be involved.
  - Compare a “good” run with a “bad” run
- Keep all your changes reversible
- With luck you will find the faulty code

For non-trivial bugs you have to reduce the search space.

This is parts of the program execution which do not affect the bug – however you must check that this assumption was correct.

Sometimes stubbing out large parts of the code to prove they are not involved can be helpful.

For multi-process programs can you eliminate entire machines and/or processes from the bug?



May you live in interesting times

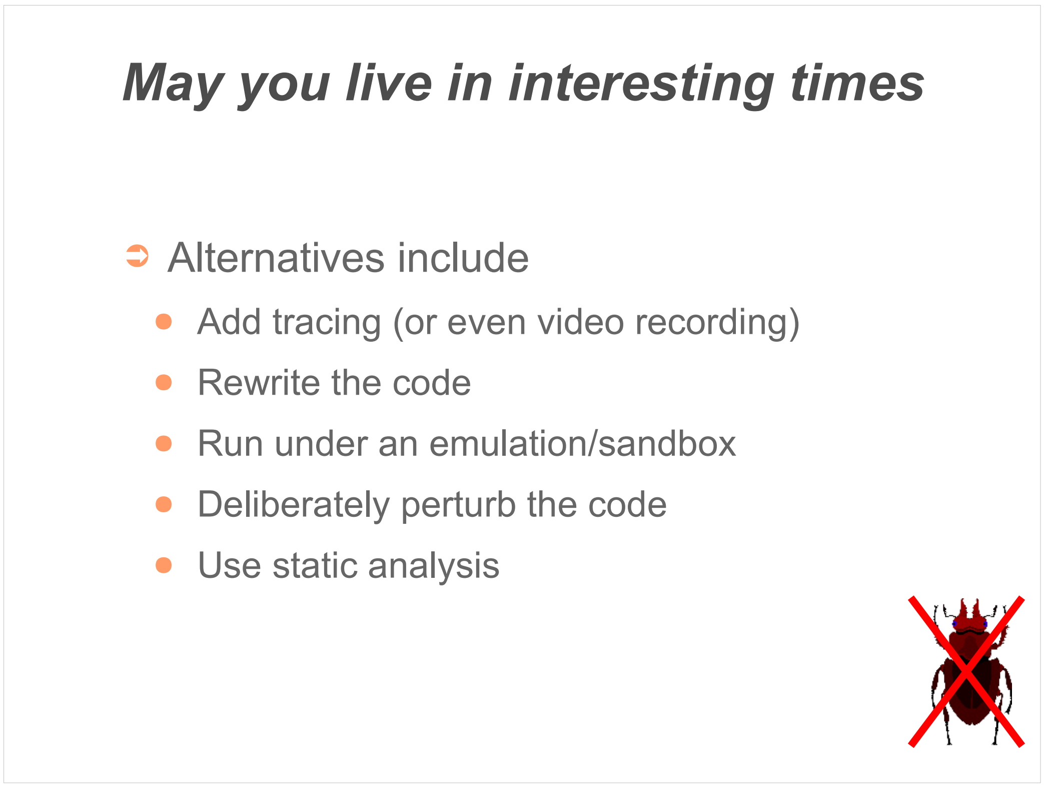
- Locating the fault is hard when it is:
  - Not reliably reproducible (or very manual)
  - Non-local (eg C++ memory corruption)
  - Timing related (eg network, threads)
  - Environment related (eg permissions)
  - Not the fault you hypothesised

A, fortunately small, fraction of bugs are extremely resistant to discovery.

Some of the factors that make problems intractable are shown above.

The computer adage GIGO 'garbage in, garbage out' also applies to bugs – sometimes the bad data covers a long way from the fault before it results in a symptom.

For example, a program I once worked on had two representations of an empty string – a NULL pointer or a pointer to a zero length character array. Almost all the code worked happily with either, but the last part of the program converted the string into an Excel variable and this only worked correctly with one of the two possible empty string types.

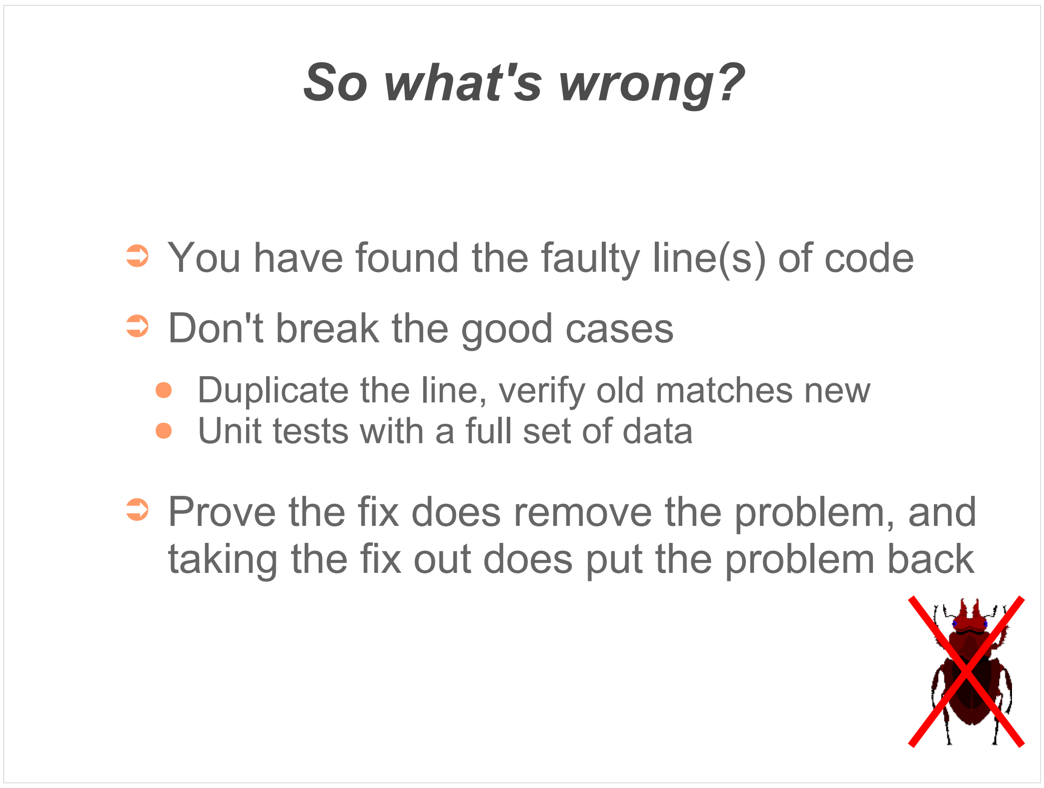


May you live in interesting times

- Alternatives include
  - Add tracing (or even video recording)
  - Rewrite the code
  - Run under an emulation/sandbox
  - Deliberately perturb the code
  - Use static analysis

Have you an 'end stop' for the bug – a guaranteed fixed term solution or work around?

Example policies include a rewrite of the failing code, detecting the bug *after* it occurs and recovering from it or running the program line by line and verifying the state at each step.

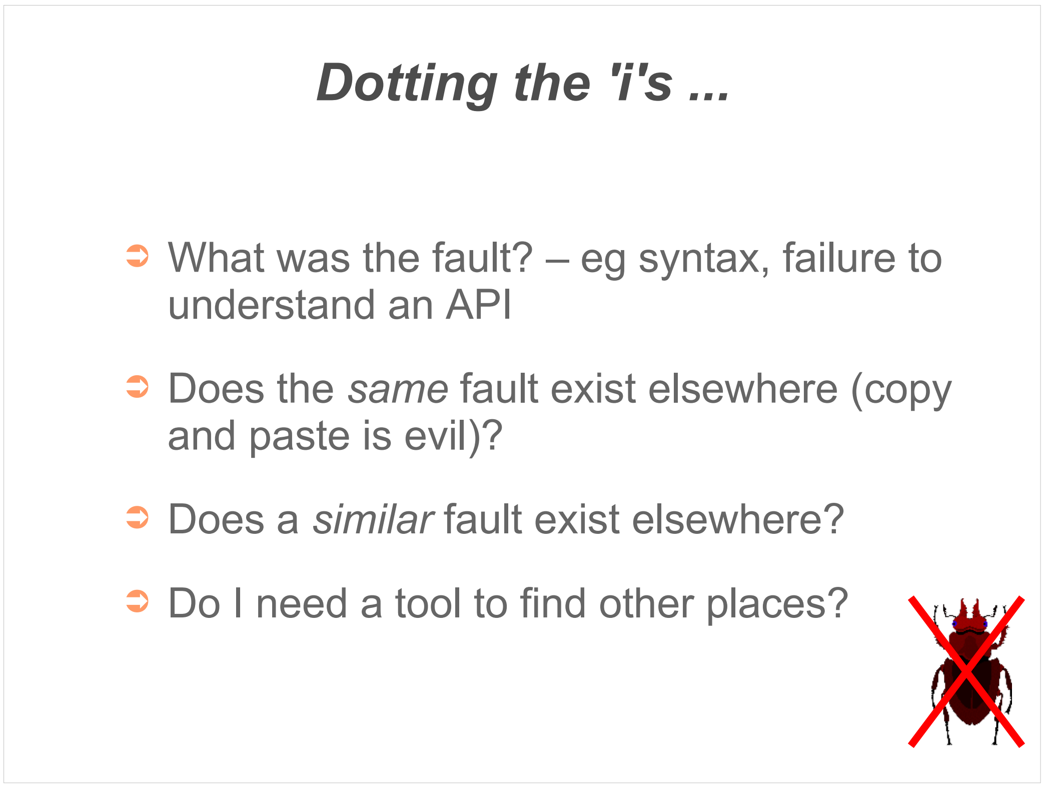


So what's wrong?

- You have found the faulty line(s) of code
- Don't break the good cases
  - Duplicate the line, verify old matches new
  - Unit tests with a full set of data
- Prove the fix does remove the problem, and taking the fix out does put the problem back

Finding the fault in the code is often the key moment, but the sense of euphoria it generates can result in the cure being worse than the disease.

Some surveys have found that 50% of bug fixes introduce fresh bugs – don't be one of these statistics!



Dotting the 'i's ...

- What was the fault? – eg syntax, failure to understand an API
- Does the *same* fault exist elsewhere (copy and paste is evil)?
- Does a *similar* fault exist elsewhere?
- Do I need a tool to find other places?

One thing that separates the expert debuggers from the rest is the ability to learn from their (or other people's) mistakes.

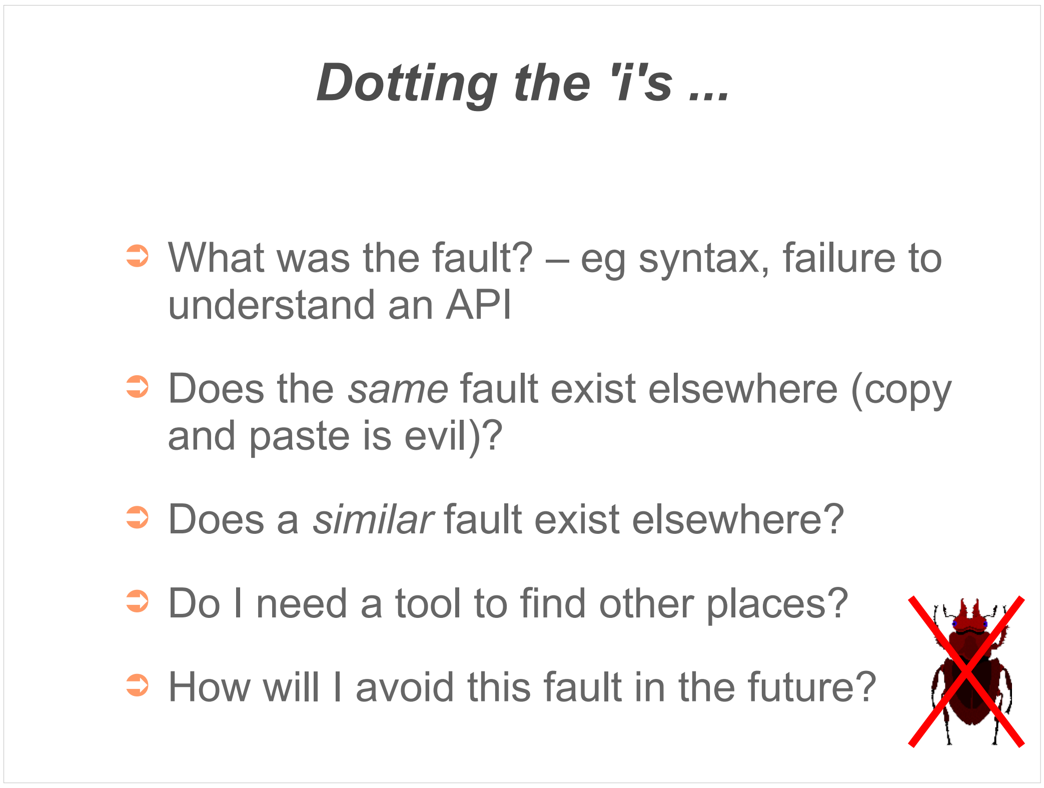
What was the reason for the bug – a simple mistake, a lack of understanding, a misplaced assumption, or what?

Is the same error likely to be repeated elsewhere?

(Of course, given the amount of source code reuse there is a good chance the exactly the same bug is *already* repeated elsewhere!)

Look for it pro-actively, using an automated technique if possible.

In some cases – for example compiler optimiser bugs or a misuse of an API – understanding the fault in detail allows you to write a tool to automatically search for other occurrences (whether actual instances in the binary or potential instances in source code)

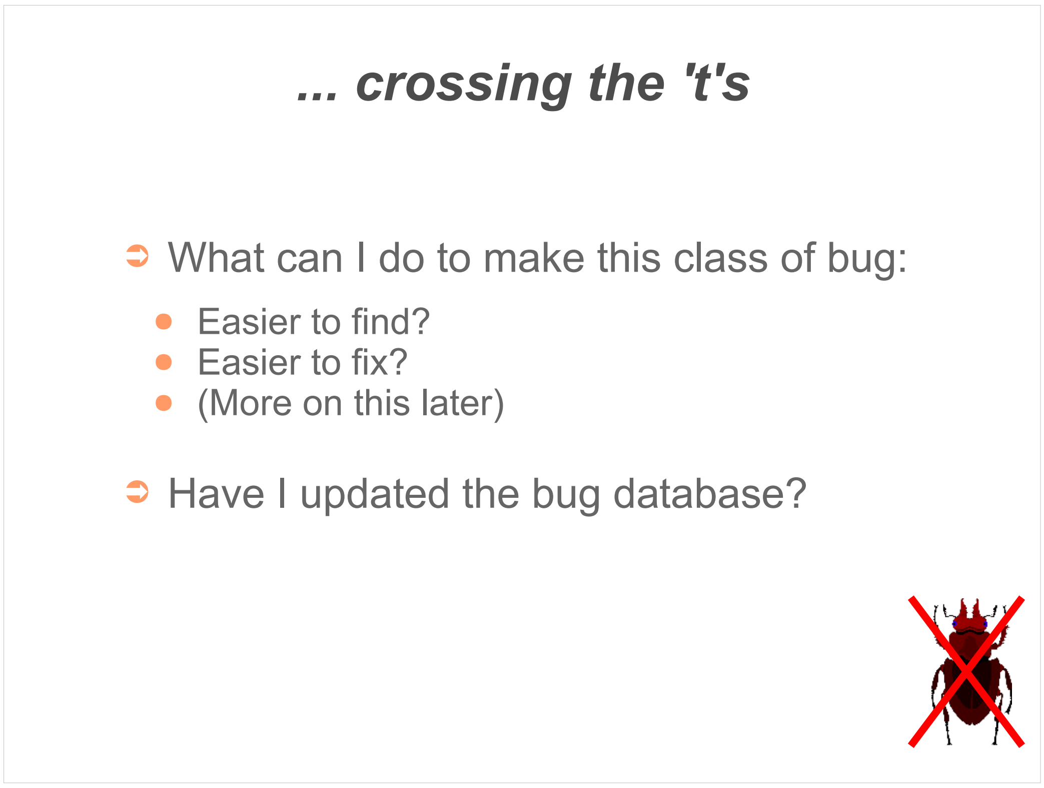


Dotting the 'i's ...

- What was the fault? – eg syntax, failure to understand an API
- Does the *same* fault exist elsewhere (copy and paste is evil)?
- Does a *similar* fault exist elsewhere?
- Do I need a tool to find other places?
- How will I avoid this fault in the future?

The last point above is possibly the most important – have I learned a lesson from the bug I fixed or will I create the same problem again?

What are you own blind spots?



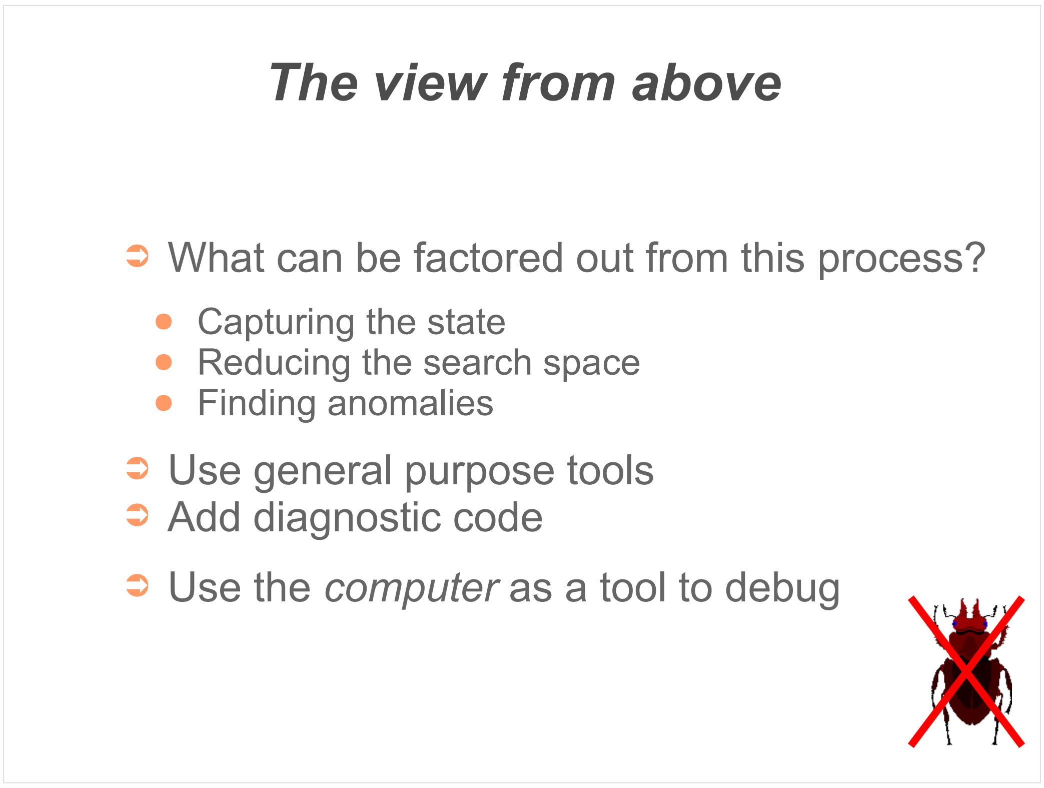
... crossing the 't's

- ➲ What can I do to make this class of bug:
  - Easier to find?
  - Easier to fix?
  - (More on this later)
- ➲ Have I updated the bug database?

Reflect on the process of finding and fixing the bug – is there anything you can do differently to make this task easier?

The bug database has a tendency to become a sink for hopeless causes. There are a number of reasons for this, including lack of filtering on entry and failure to suitably 'age' faults.

If a bug is fixed though, the corresponding entry in the database should be marked (in some cases there may be multiple reports of the same underlying fault). It is of course not yet finally cleared until the testing process is satisfied.



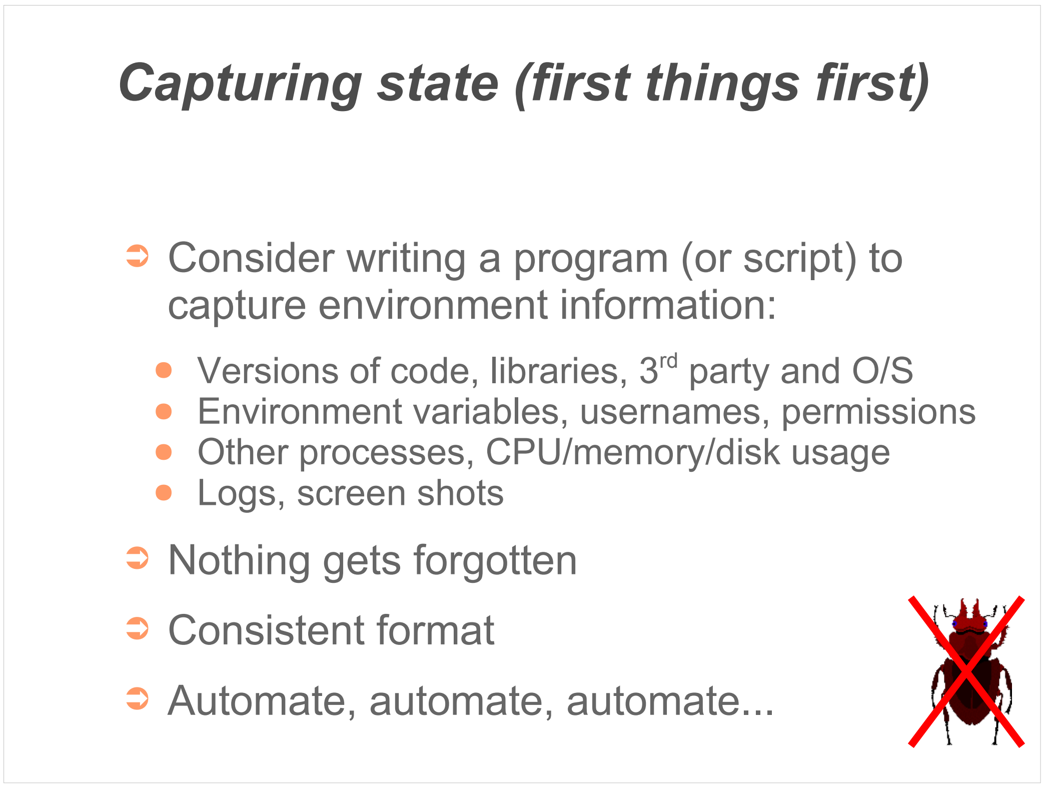
The view from above

- What can be factored out from this process?
  - Capturing the state
  - Reducing the search space
  - Finding anomalies
- Use general purpose tools
- Add diagnostic code
- Use the *computer* as a tool to debug

How can the process described in the last few slides be improved?

One way is to "separate out the things that change from the things that stay the same". Automation is key to improving debugging, but it is vital to try and automate the right parts of the process.

The next section looks at using tools in debugging, both in terms of code included in the program itself and separate programs used to help investigate the faulting program.



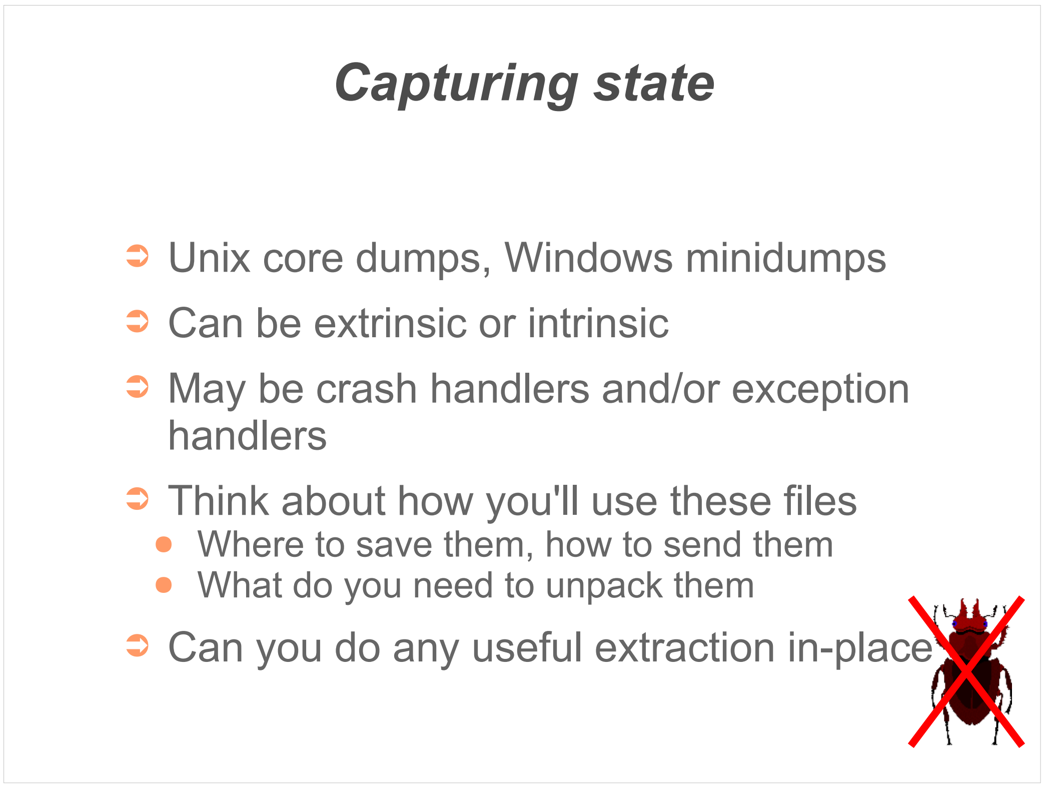
Capturing state (first things first)

- Consider writing a program (or script) to capture environment information:
  - Versions of code, libraries, 3rd party and O/S
  - Environment variables, usernames, permissions
  - Other processes, CPU/memory/disk usage
  - Logs, screen shots
- Nothing gets forgotten
- Consistent format
- Automate, automate, automate...

Capturing the state at the time of the fault is the first, and sometimes the key, part of debugging.

Very many of the things that you need to capture are not specific to the particular program that is faulting, so generic collection programs can be used.

Scripts are typically used at the top level of such programs, to facilitate adding extra data items as they become relevant, although some of the hard work of capturing data may well be delegated to other programs – for example using gcore on Unix to capture a core image of the program.

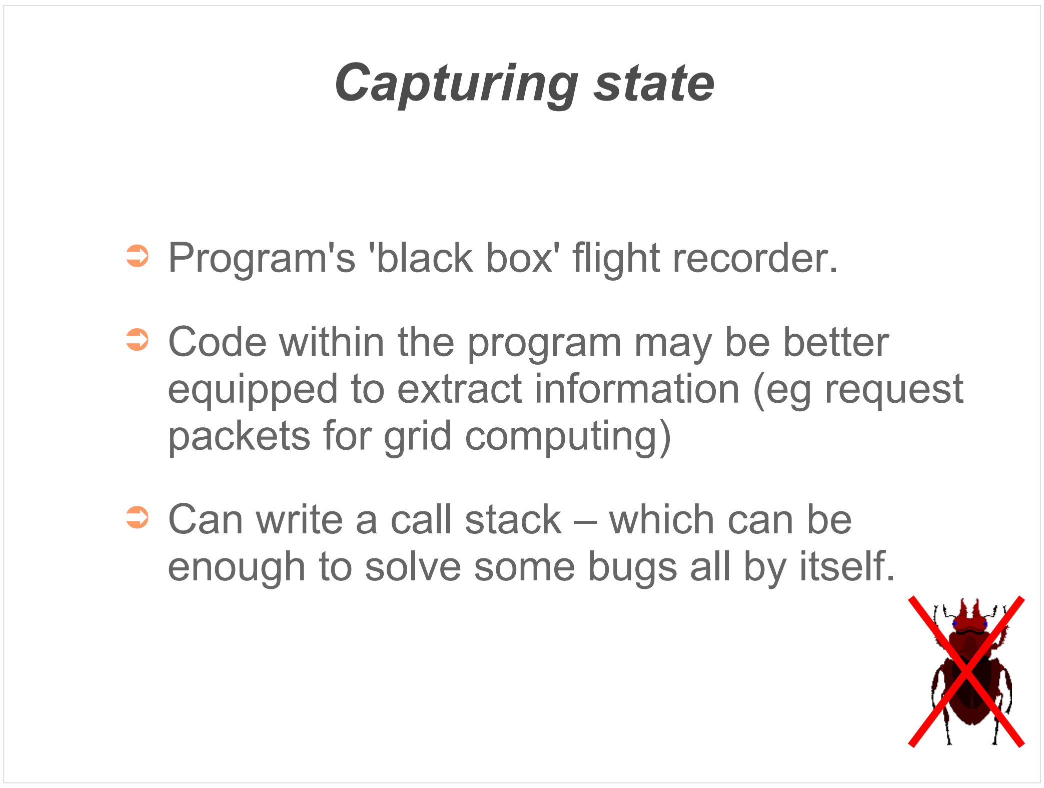


Capturing state

- Unix core dumps, Windows minidumps
- Can be extrinsic or intrinsic
- May be crash handlers and/or exception handlers
- Think about how you'll use these files
  - Where to save them, how to send them
  - What do you need to unpack them
- Can you do any useful extraction in-place

A dump is a great aid to fault finding, as it captures the whole state of the process in a standard form that you can examine later.

Think before hand about the creation and management of these dump files; this will depend a lot upon your application's target audience.

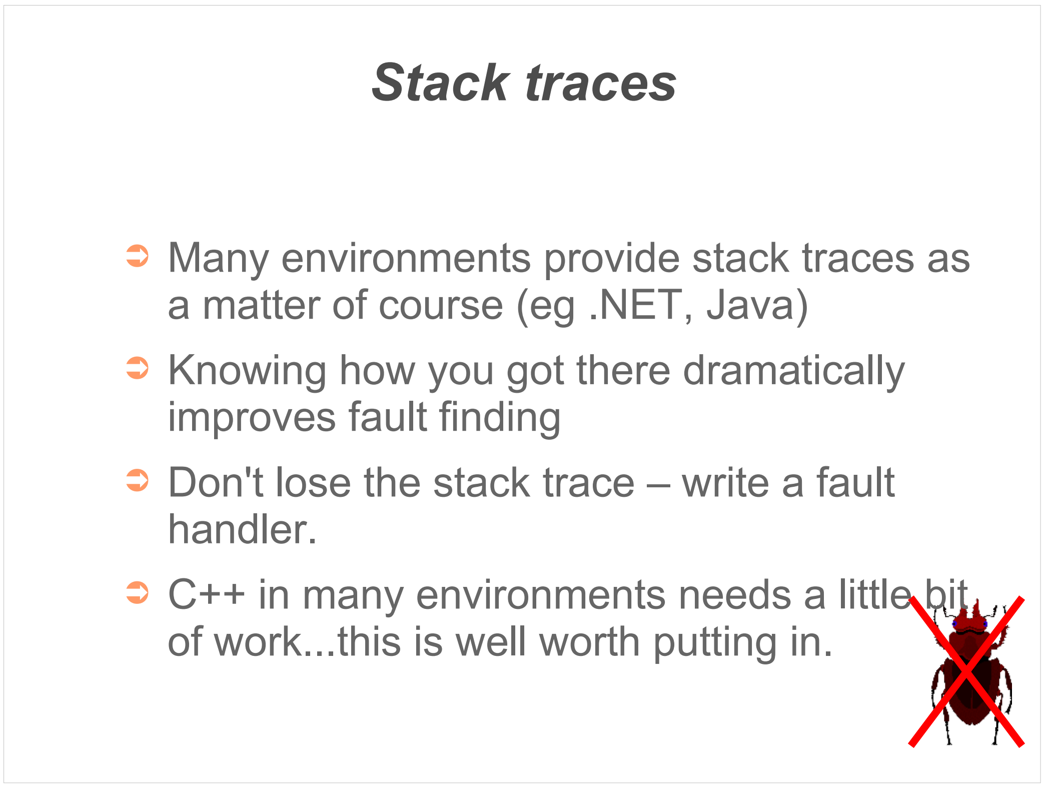


Capturing state

- Program's 'black box' flight recorder.
- Code within the program may be better equipped to extract information (eg request packets for grid computing)
- Can write a call stack – which can be enough to solve some bugs all by itself.

There may be information the program itself can capture for subsequent analysis. This is particularly true of application specific information that is known to the program but hard to infer after the event.

One simple piece of information that is often overlooked is a call stack of the failing location.

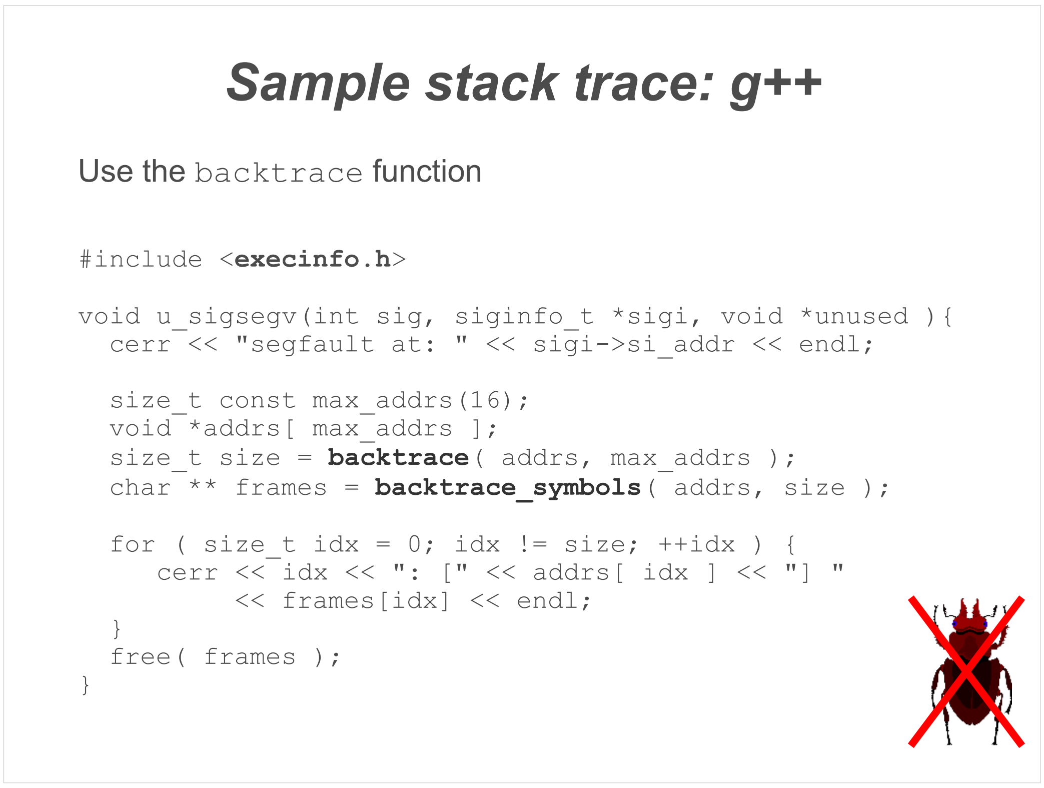


Stack traces

- Many environments provide stack traces as a matter of course (eg .NET, Java)
- Knowing how you got there dramatically improves fault finding
- Don't lose the stack trace – write a fault handler.
- C++ in many environments needs a little bit of work...this is well worth putting in.

A stack trace is the structured programming equivalent of the mythical 'come from' instruction – it lets you understand some of the history of the problem and can make it easier to work back from the symptom to the underlying fault.

Hence many environments make it easy to get a stack trace – but it is still your task to report it somewhere useful.



Sample stack trace: g++

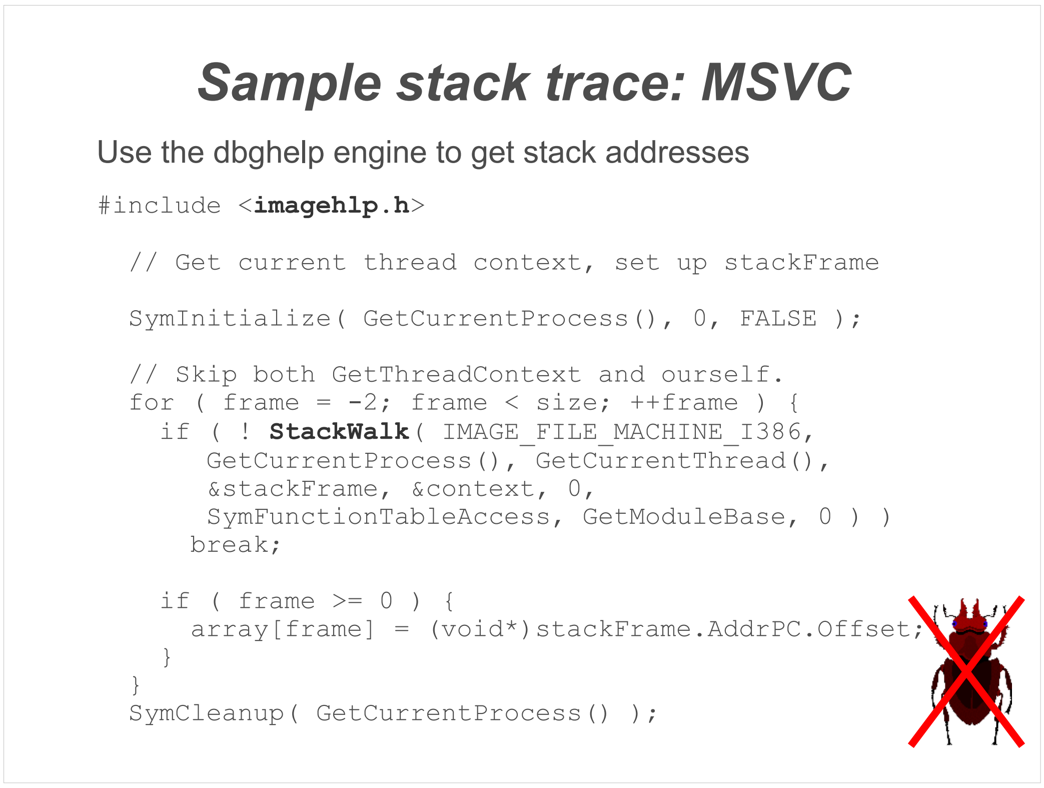
Use the `backtrace` function

```
#include <execinfo.h>

void u_sigsegv(int sig, siginfo_t *sigi, void *unused ){
  cerr << "segfault at: " << sigi->si_addr << endl;

  size_t const max_addrs(16);
  void *addrs[ max_addrs ];
  size_t size = backtrace( addrs, max_addrs );
  char ** frames = backtrace_symbols( addrs, size );

  for ( size_t idx = 0; idx != size; ++idx ) {
     cerr << idx << ": [" << addrs[ idx ] << "] "
          << frames[idx] << endl;
  }
  free( frames );
}
```

A simple Linux fault handler to produce a stack walkback when a segmentation fault occurs.

This can be installed using `sigaction` with the `sa_flags` set to `SA_SIGINFO | SA_ONESHOT`.

Note that we still get a core dump (if enabled), but you may not need it if the call stack alone identifies the problem.



Sample stack trace: MSVC

Use the dbghelp engine to get stack addresses

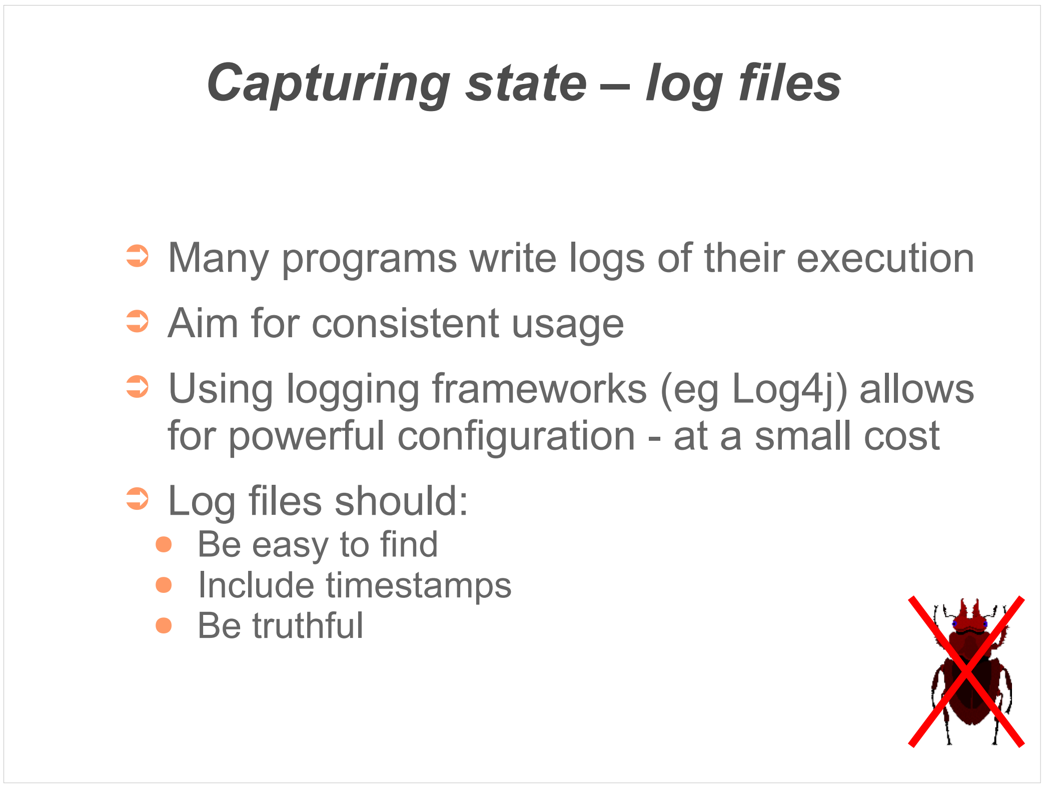
```
#include <imagehlp.h>

  // Get current thread context, set up stackFrame

  SymInitialize( GetCurrentProcess(), 0, FALSE );

  // Skip both GetThreadContext and ourself.
  for ( frame = -2; frame < size; ++frame ) {
    if ( ! StackWalk( IMAGE_FILE_MACHINE_I386,
         GetCurrentProcess(), GetCurrentThread(),
         &stackFrame, &context, 0,
         SymFunctionTableAccess, GetModuleBase, 0 ) )
      break;

    if ( frame >= 0 ) {
      array[frame] = (void*)stackFrame.AddrPC.Offset;
    }
  }
  SymCleanup( GetCurrentProcess() );
```

The dbghelp engine supplied by Microsoft is very useful for getting information about program execution. It provides functions for walking the stack and also for getting symbols from addresses.
Sadly it is not as easy to use as the backtrace functions in g++.

Sample stack trace: MSVC

- The dbghelp engine can also be used to get names using SymGetSymFromAddr()
- You can also obtain access to data types, local variables, etc. - see the Debugging Tools for Windows at: www.microsoft.com/whdc/devtools. (Use custom install to get the SDK)

One powerful feature of dbghelp is the ability to programmatically query data types and variables.
The makes it possible to write tools (or helper functions) that example variables and their values when, for example, producing a stack trace.

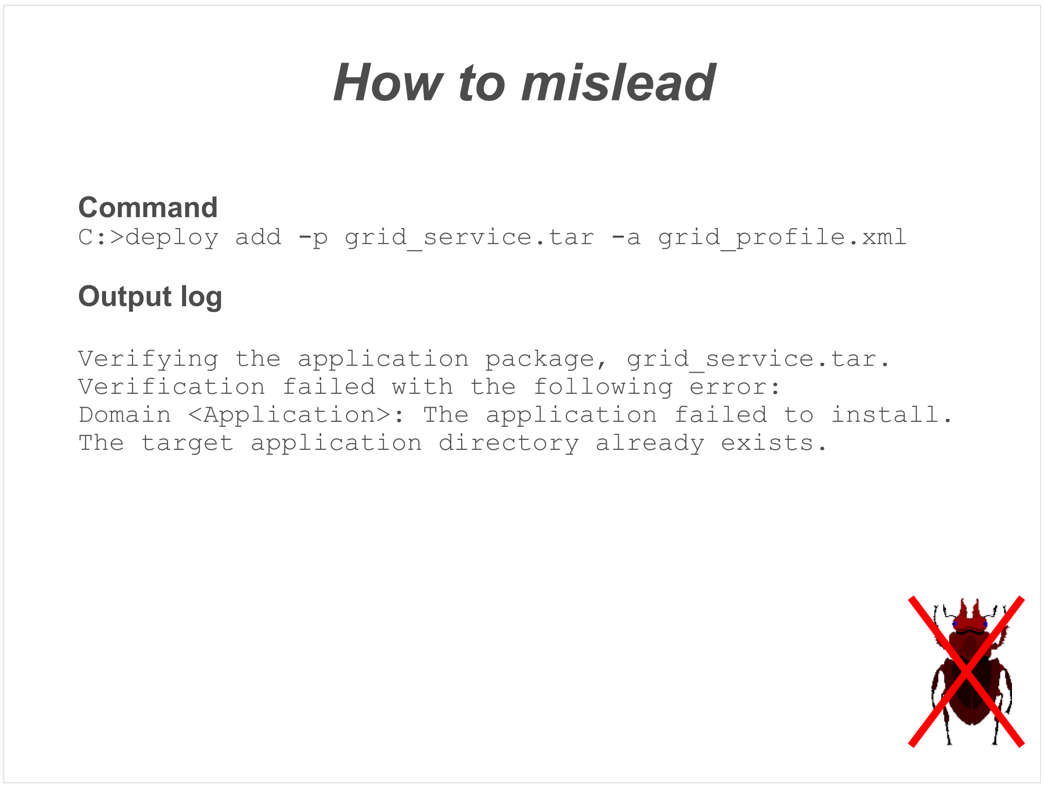


Capturing state – log files

- Many programs write logs of their execution
- Aim for consistent usage
- Using logging frameworks (eg Log4j) allows for powerful configuration - at a small cost
- Log files should:
  - Be easy to find
  - Include timestamps
  - Be truthful

Many programs write log files, but much of what is logged is actually of little help to find and fix bugs.

Remember that the log file is typically read a good time after the code was written and the subtle nuances of the code have been forgotten!



How to mislead

Command

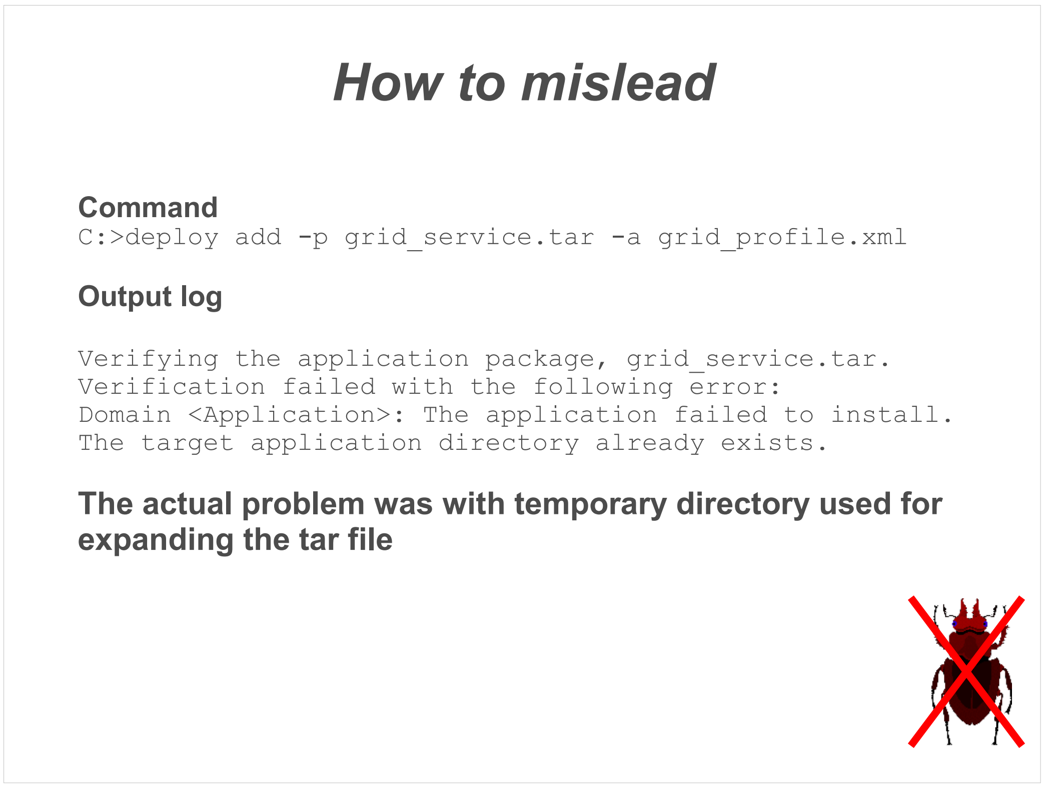
```
C:>deploy add -p grid_service.tar -a grid_profile.xml
```

Output log

```
Verifying the application package, grid_service.tar.
Verification failed with the following error:
Domain <Application>: The application failed to install.
The target application directory already exists.
```

It would appear from the log that the problem is with the target directory on the grid environment.

This was certainly what we tried looking at first.



How to mislead

Command

```
C:>deploy add -p grid_service.tar -a grid_profile.xml
```

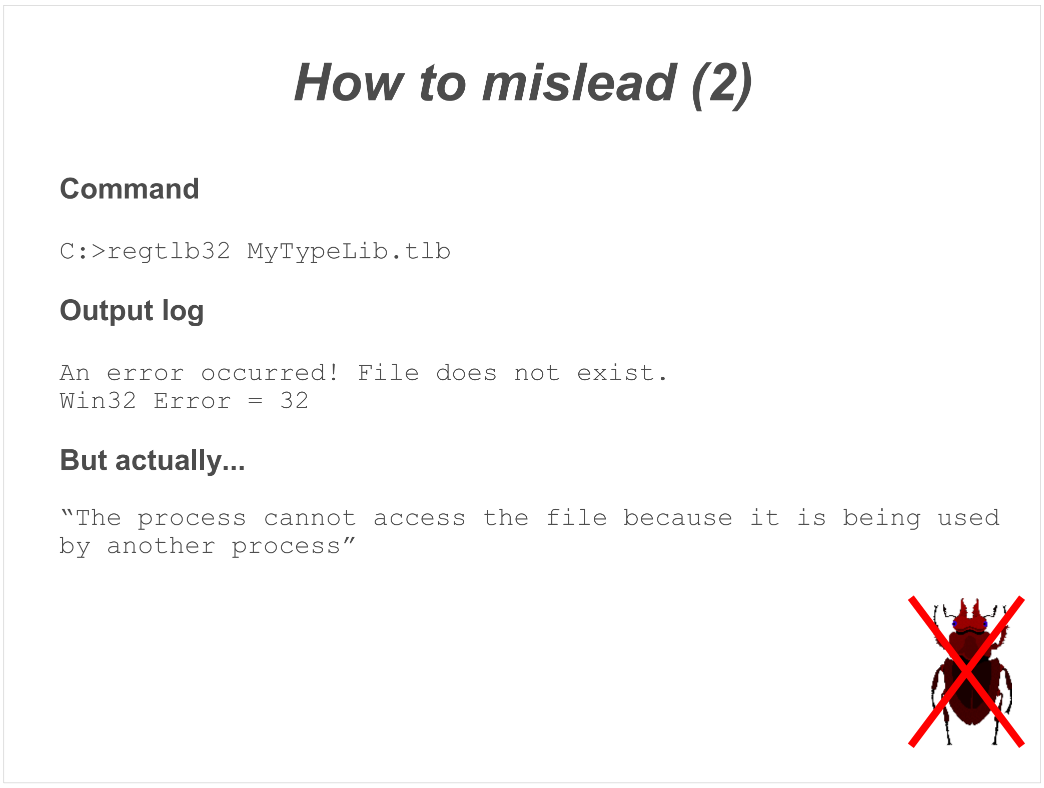
Output log

```
Verifying the application package, grid_service.tar.
Verification failed with the following error:
Domain <Application>: The application failed to install.
The target application directory already exists.
```

The actual problem was with temporary directory used for expanding the tar file

Erroneous or misleading information can be very costly as it can completely throw people off the scent of the real problem.

In this particular case the problem was found with ntTrace; a call to `NtCreateFile` was failing with status 0xc0000035 [183 'Cannot create a file when that file already exists.']



How to mislead (2)

Command

```
C:>regtlb32 MyTypeLib.tlb
```

Output log

```
An error occurred! File does not exist.
Win32 Error = 32
```

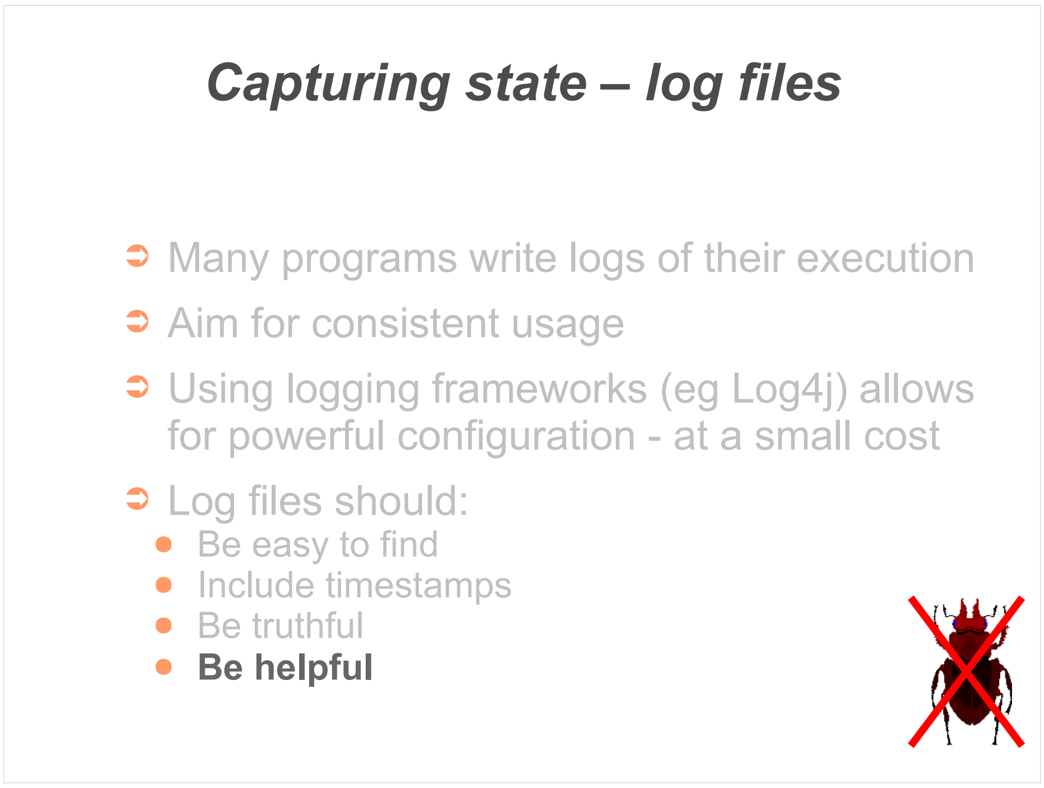
But actually...

```
"The process cannot access the file because it is being used
by another process"
```

At least here the underlying error number is retained. Assuming it is the error number from the actual fault, of course.

It is good, where possible, to give both a user-friendly and a technically more complete error message.

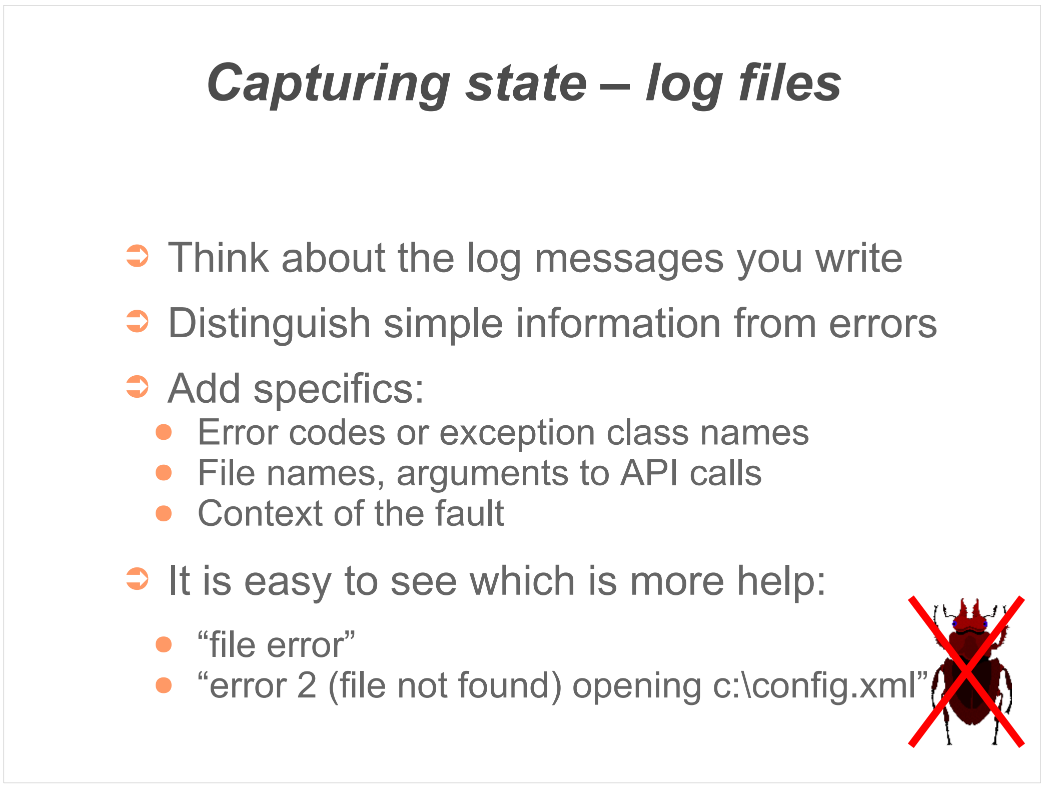
One alternative is to retain the technical information – perhaps in a log file – while giving the user a less technical fault report.



Capturing state – log files

- Many programs write logs of their execution
- Aim for consistent usage
- Using logging frameworks (eg Log4j) allows for powerful configuration - at a small cost
- Log files should:
  - Be easy to find
  - Include timestamps
  - Be truthful
  - **Be helpful**

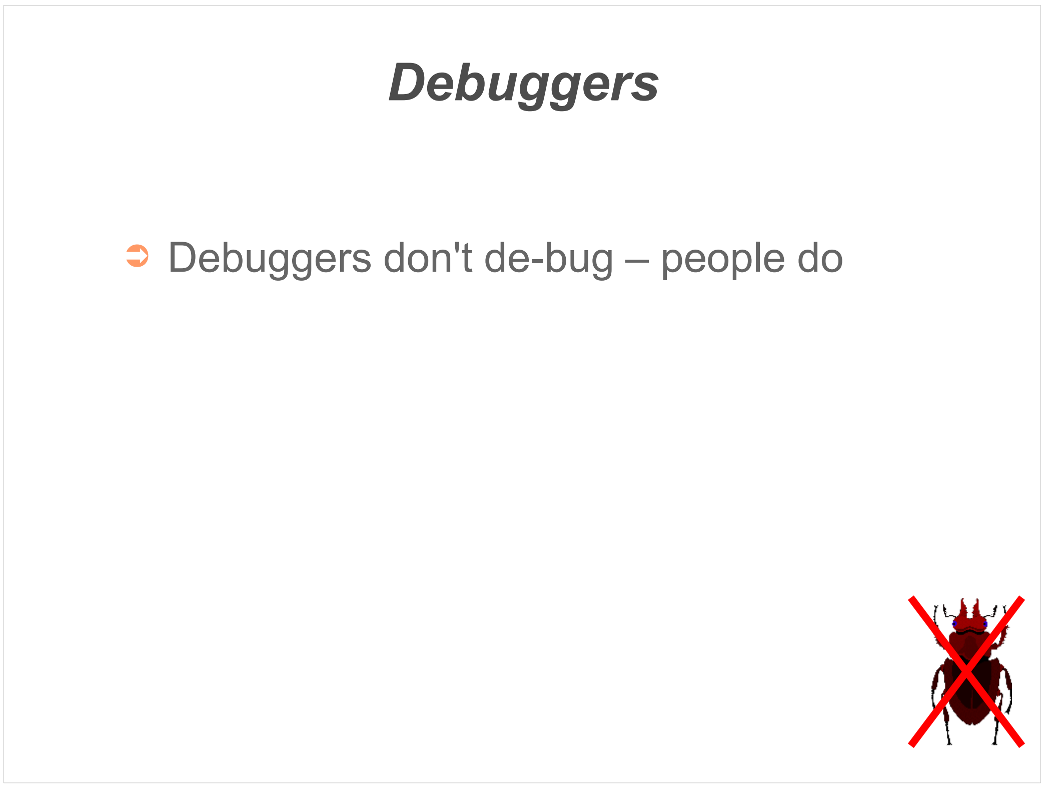
Examine your log files – both in normal running and after a fault. If they don't tell you anything useful why are you collecting them?



Capturing state – log files

- Think about the log messages you write
- Distinguish simple information from errors
- Add specifics:
  - Error codes or exception class names
  - File names, arguments to API calls
  - Context of the fault
- It is easy to see which is more help:
  - “file error”
  - “error 2 (file not found) opening c:\config.xml”

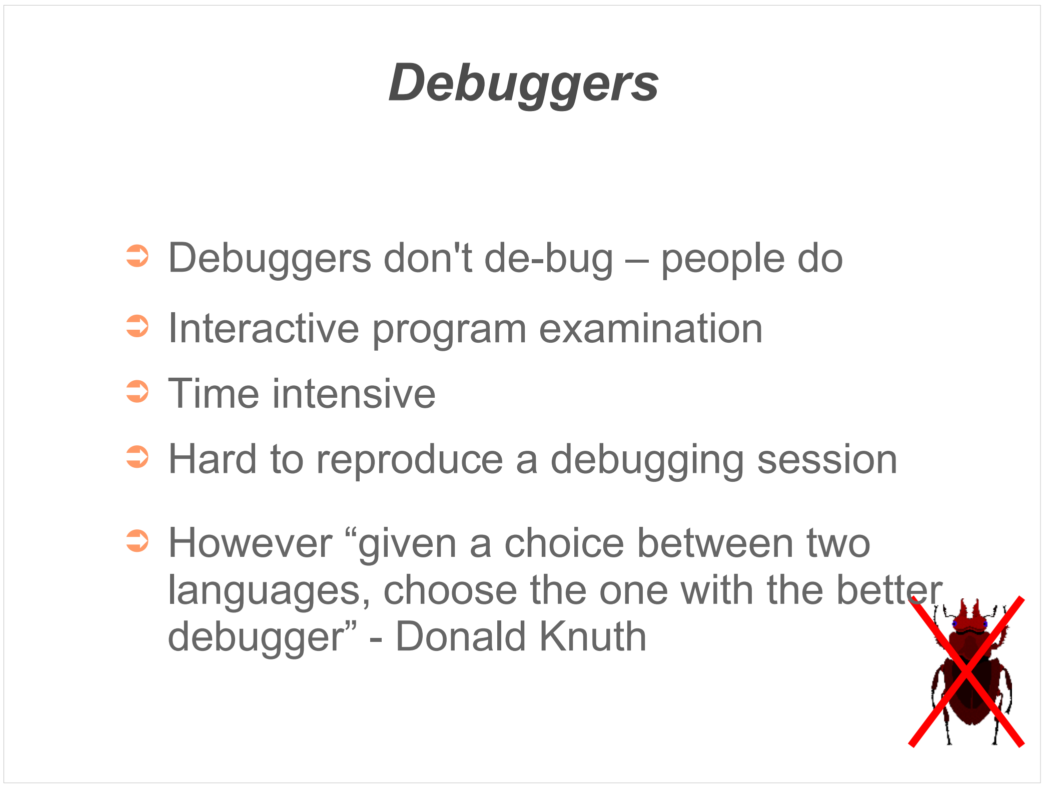
Ideally log files should be understandable without reference to the source code; it should be obvious what is informative text and what is abnormal behaviour.
When you are logging something think about what additional specific information might be useful – it is usually quite simple to add it to the logger.



Debuggers

- Debuggers don't de-bug – people do

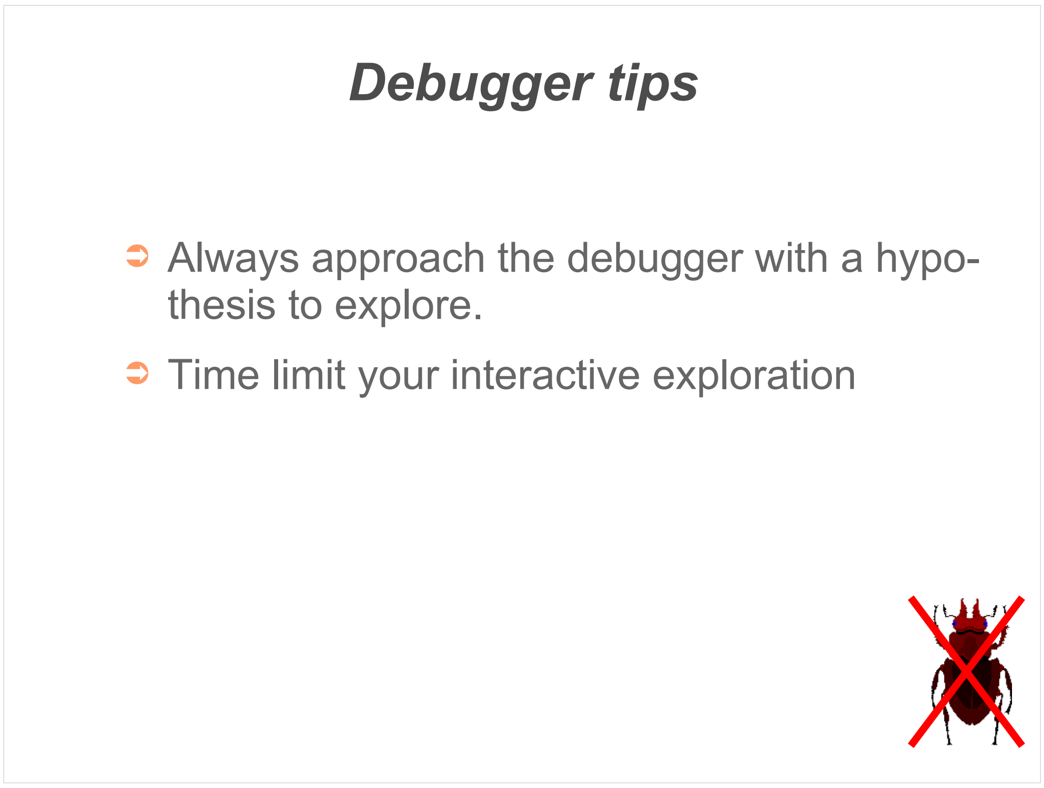
Debuggers can be a very expensive way of consuming valuable developer time. Like fire, they can be a great servant but a bad master.



Debuggers

- Debuggers don't de-bug – people do
- Interactive program examination
- Time intensive
- Hard to reproduce a debugging session
- However “given a choice between two languages, choose the one with the better debugger” - Donald Knuth

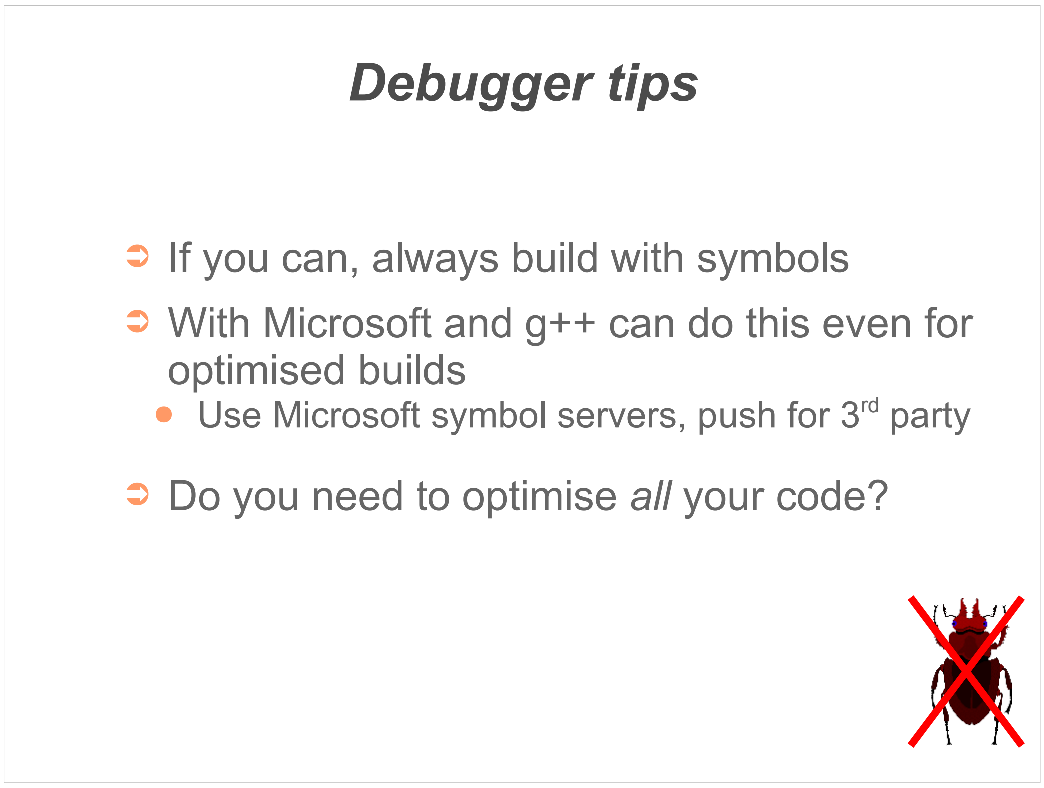
The main problem with most tools called 'debuggers' is they provide, by default, an interactive tool for examining the status of an executing program.
It is typically quite time consuming, especially since if you go *past* the fault and simply find a symptom of it the debugging session may have to be restarted.



Debugger tips

- Always approach the debugger with a hypothesis to explore.
- Time limit your interactive exploration

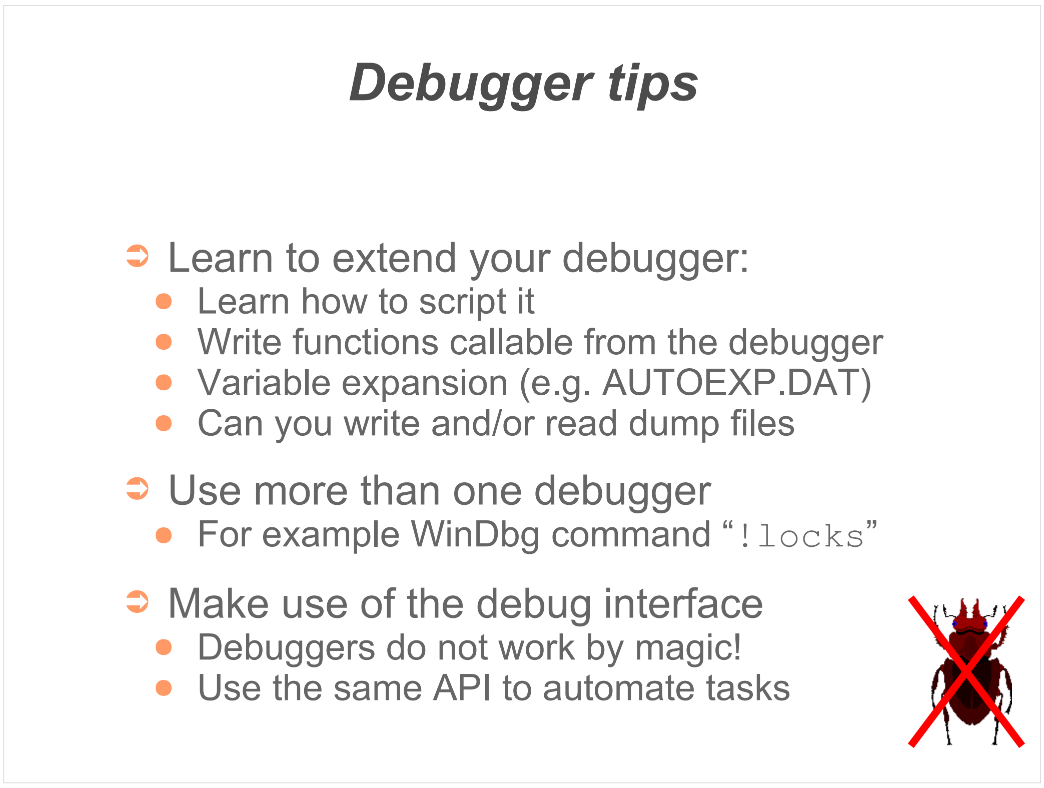
If you come to a debugger with a hypothesis to explore it is likely that the tool will be more useful. Even so, it can be a good discipline to set a time limit for interactive debugging after which you will try other techniques for finding the fault.



Debugger tips

- If you can, always build with symbols
- With Microsoft and g++ can do this even for optimised builds
  - Use Microsoft symbol servers, push for 3rd party
- Do you need to optimise *all* your code?

Debuggers are of most use when (a) they have full symbols for your program and (b) execution flow and use of variables matches your source code.
This can be hard to achieve with fully optimised builds (for compiled languages) – so often people use a special 'debug' build for running under a debugger. This makes debugging production problems harder!
An alternative is to consider varying the optimisation level across your application (where this is supported).



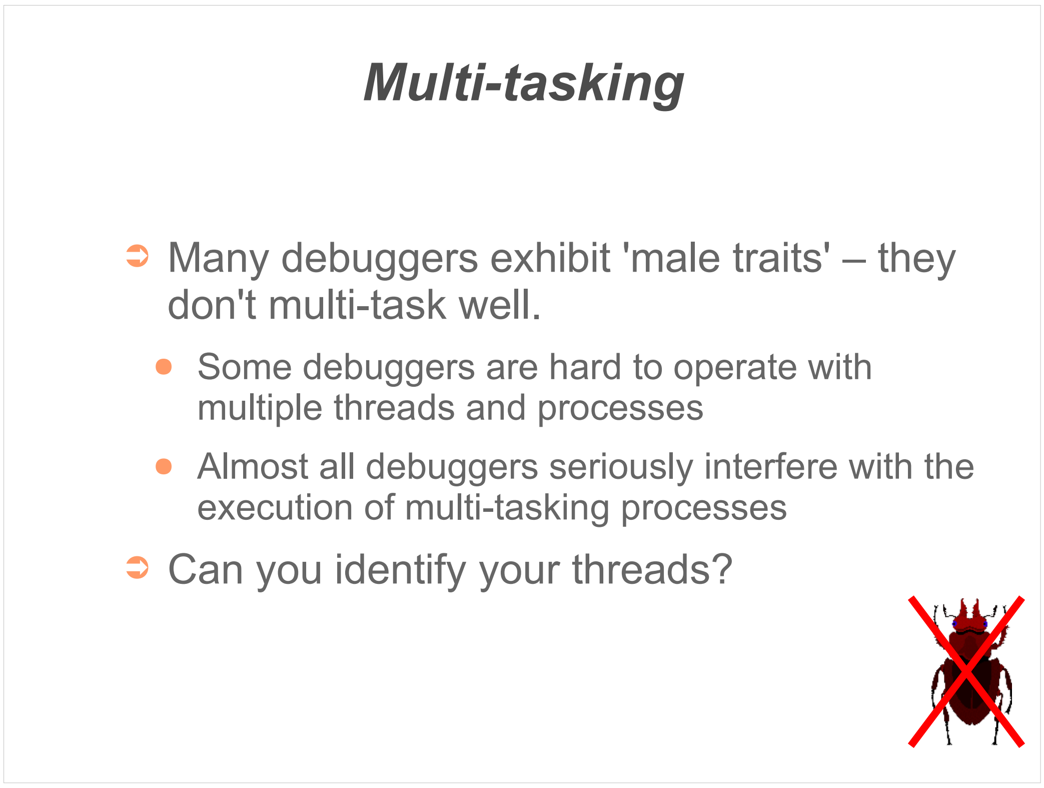
Debugger tips

- Learn to extend your debugger:
  - Learn how to script it
  - Write functions callable from the debugger
  - Variable expansion (e.g. AUTOEXP.DAT)
  - Can you write and/or read dump files
- Use more than one debugger
  - For example WinDbg command "`!locks`"
- Make use of the debug interface
  - Debuggers do not work by magic!
  - Use the same API to automate tasks

Many modern debuggers are extendable, scriptable, or at least configurable. It is worth experimenting with your available debuggers to see what features they offer *before* you are involved in a serious fire-fighting exercise.

Again, most operating systems provide a standard debugging interface used by the system debugger; but this can also be used for simple tools targeting specific functionality.

Example: a tool to log all exceptions using the 'on exception' event from the debug interface.

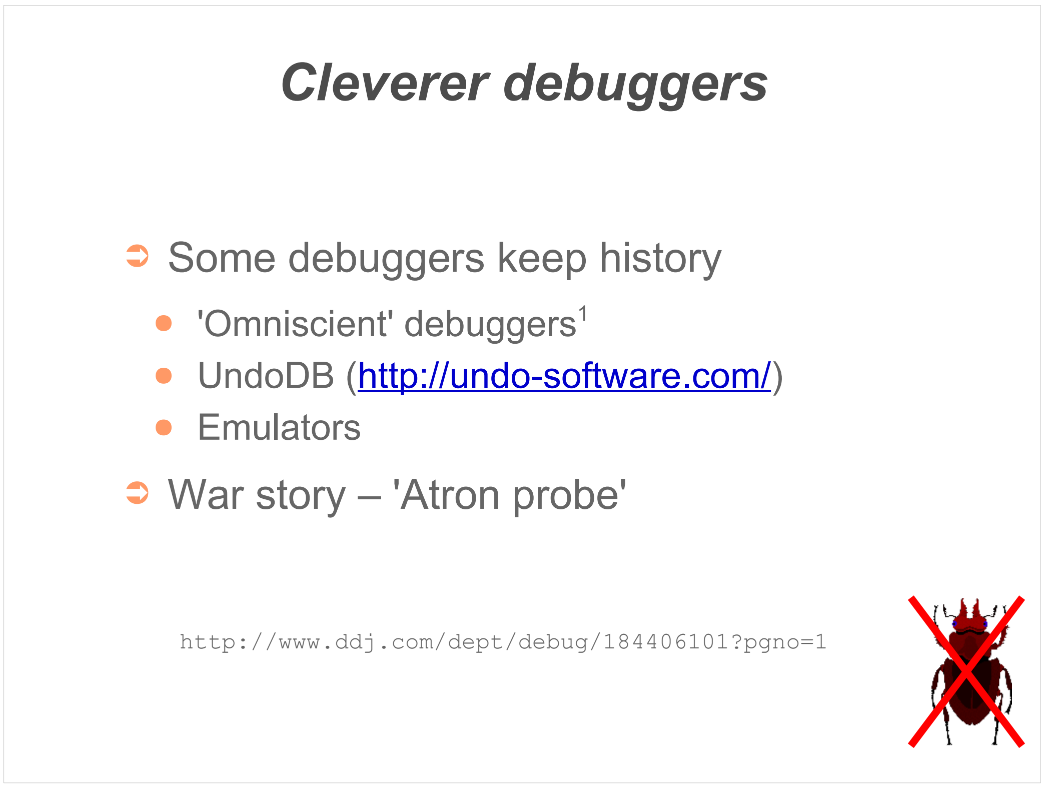


Multi-tasking

- ➲ Many debuggers exhibit 'male traits' – they don't multi-task well.
  - Some debuggers are hard to operate with multiple threads and processes
  - Almost all debuggers seriously interfere with the execution of multi-tasking processes
- ➲ Can you identify your threads?

For example, use the semi-documented method in Windows:

```
typedef struct tagTHREADNAME_INFO
 {
    DWORD dwType; // must be 0x1000
    LPCSTR szName; // pointer to name (in user space)
    DWORD dwThreadID; // thread ID (-1=caller thread)
    DWORD dwFlags; // reserved for future use, use zero
 } THREADNAME_INFO;

 THREADNAME_INFO info;
 info.dwType = 0x1000;
 info.szName = szThreadName;
 info.dwThreadID = dwThreadID;
 info.dwFlags = 0;

 __try {
    RaiseException( 0x406D1388, 0,
       sizeof(info)/sizeof(DWORD), (DWORD*)&info );
 }
 __except(EXCEPTION_CONTINUE_EXECUTION) {
 }
```



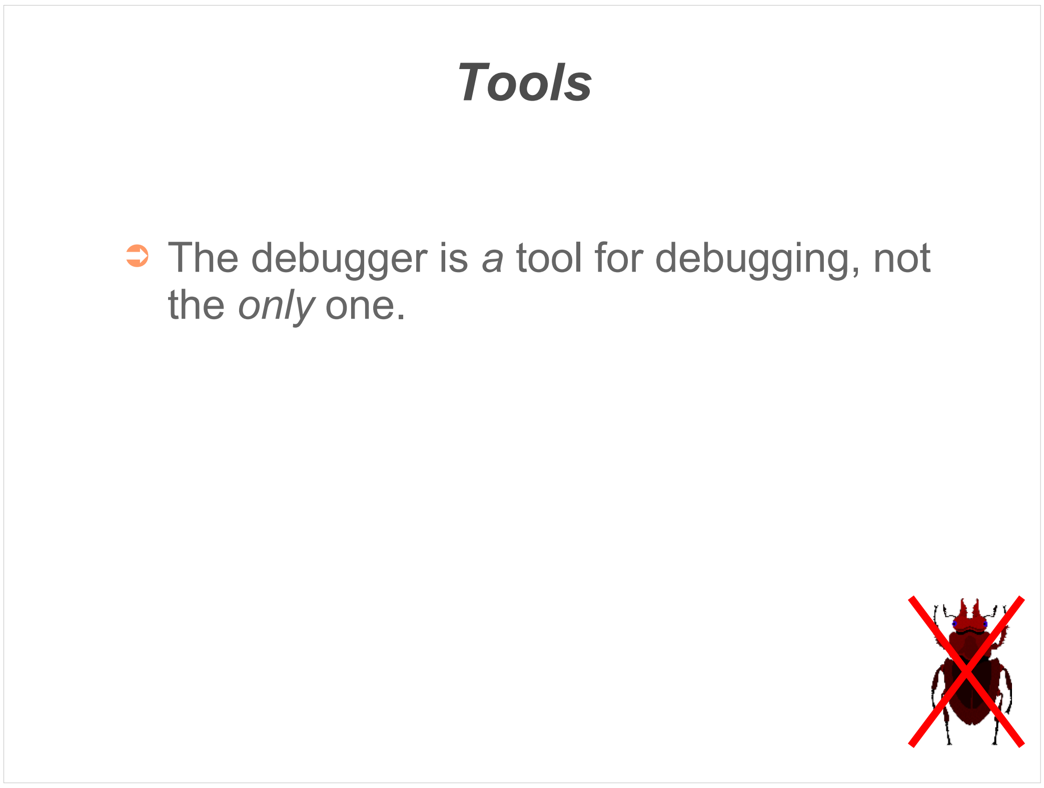
Cleverer debuggers

- Some debuggers keep history
  - 'Omniscient' debuggers[1]
  - UndoDB (http://undo-software.com/)
  - Emulators
- War story – 'Atron probe'

http://www.ddj.com/dept/debug/184406101?pgno=1

The advantage of a debugging with recording is you can *start* with the symptom and work *back* to the fault.

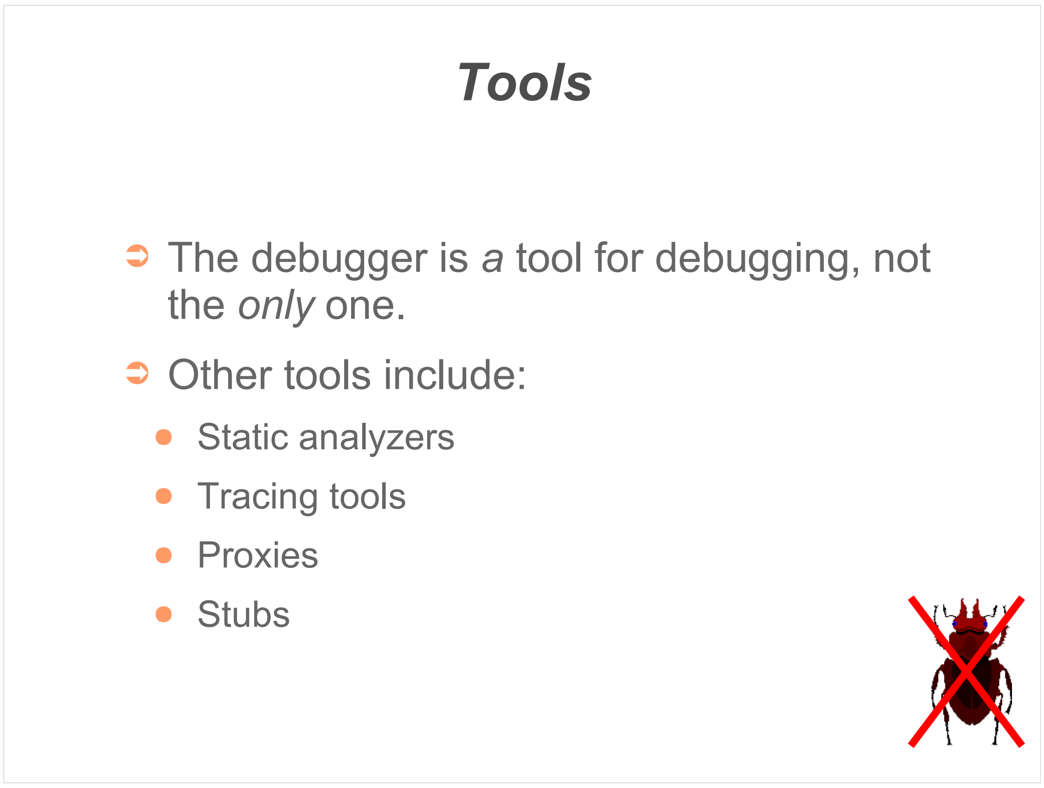
You can sometimes produce some of the same effects with simple debugger scripting, if the number of items you are interested in is fairly limited.



Tools

- The debugger is *a* tool for debugging, not the *only* one.

I personally think the name 'debugger' is a genuine problem as it leads to many people restricting their choice of tools when they are trying to find bugs.



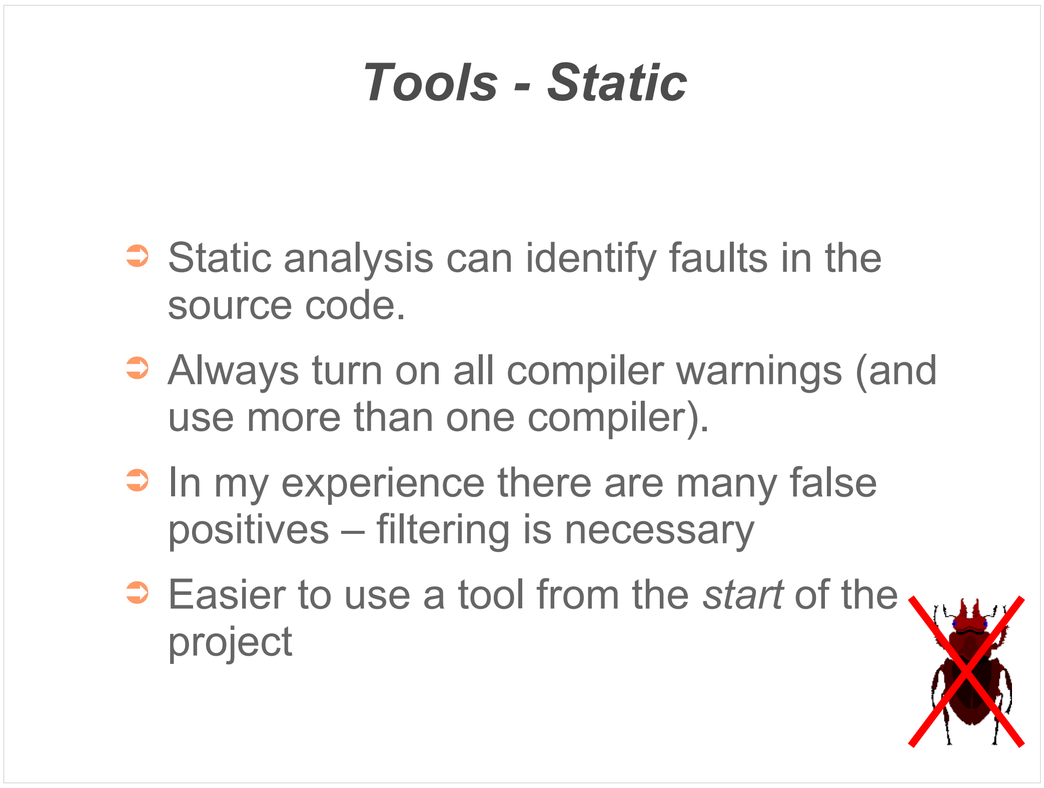
Tools

- The debugger is *a* tool for debugging, not the *only* one.
- Other tools include:
  - Static analyzers
  - Tracing tools
  - Proxies
  - Stubs

Many tools can be used to help debug.
Some analyse the program looking for potential problems which, where successful, removes faults before they create an infection.

Other tools provide alternative ways to inspect, verify or modify the execution of a running program.

Some powerful techniques involve combining multiple tools.

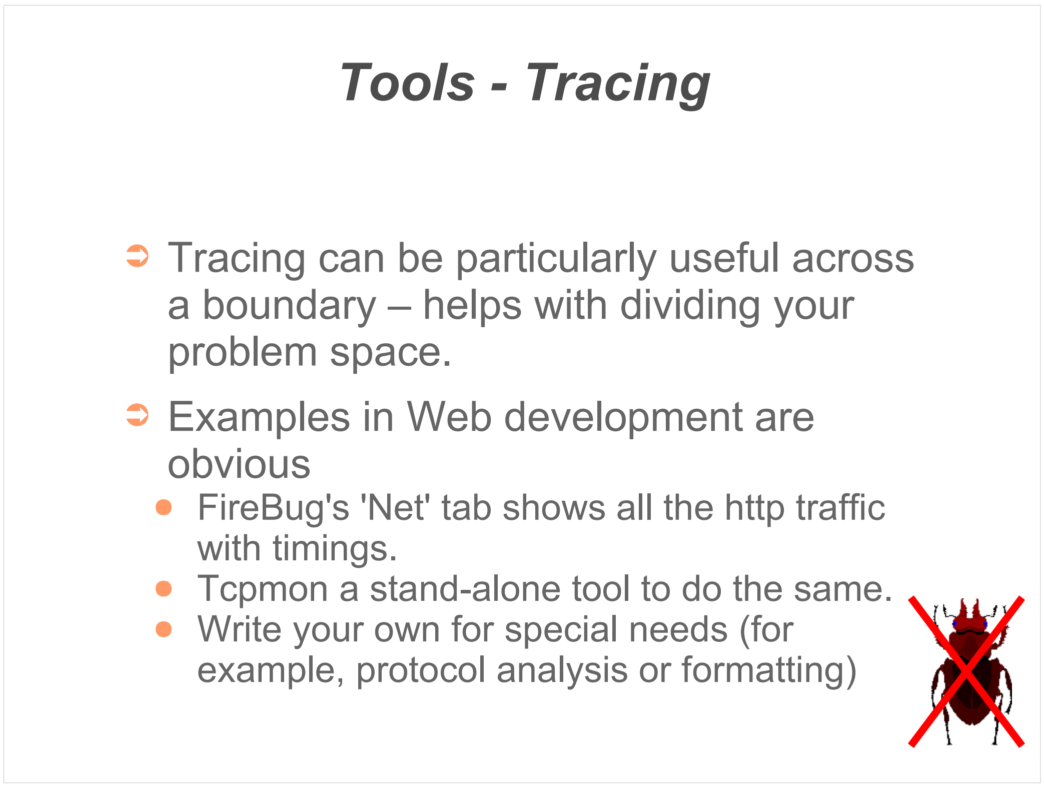


Tools - Static

- Static analysis can identify faults in the source code.
- Always turn on all compiler warnings (and use more than one compiler).
- In my experience there are many false positives – filtering is necessary
- Easier to use a tool from the *start* of the project

The main drawback with static analysis seems to be the number of potential problems that turn out on closer inspection to be perfectly OK.
Tools that allow flexible configuring of their sensitivity are more likely to be useful.

I continue to be slightly puzzled by the observation that the programmers who most need to pay attention to compiler warnings seem to be the ones most likely to ignore them.



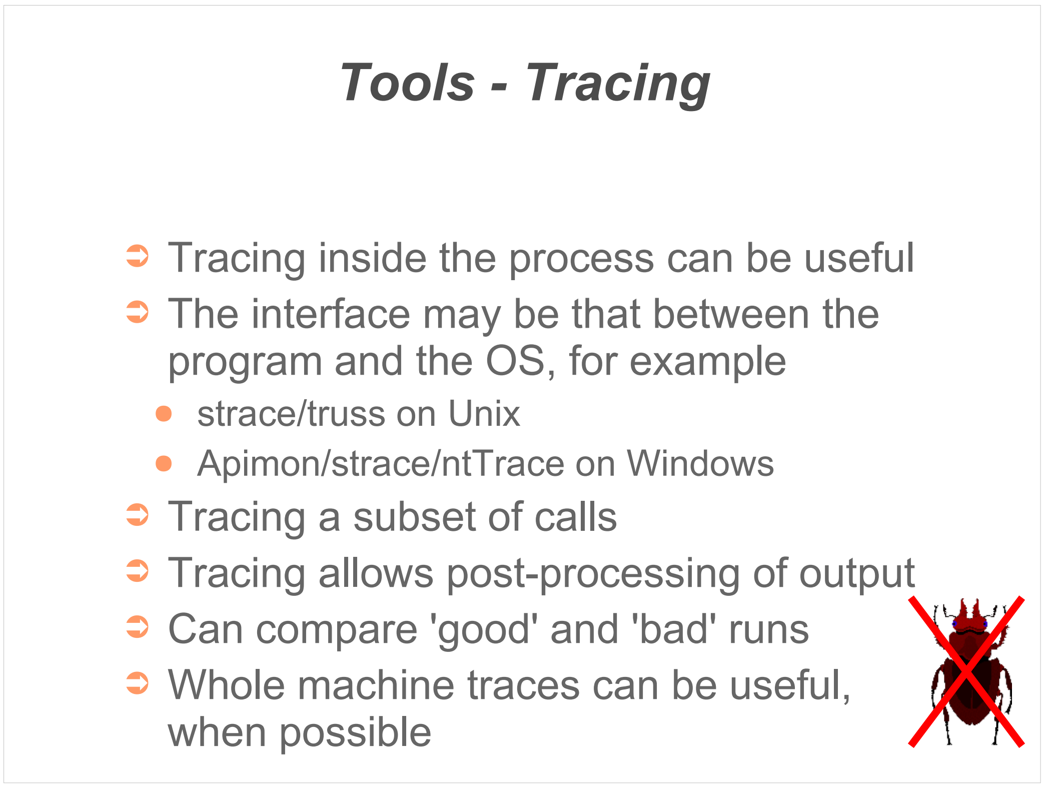
Tools - Tracing

- Tracing can be particularly useful across a boundary – helps with dividing your problem space.
- Examples in Web development are obvious
  - FireBug's 'Net' tab shows all the http traffic with timings.
  - Tcpmon a stand-alone tool to do the same.
  - Write your own for special needs (for example, protocol analysis or formatting)

Tracing lets you look for infection – incorrect/abnormal/unusual data – the occurred prior to the symptom.

Wherever your application has a clean interface between processes consider *what* you might learn from tracing flow across that interface and *how* you can capture this data.

Some tracing simply logs the actual bytes transferred, more advanced tracing may add state management or protocol disassembly.

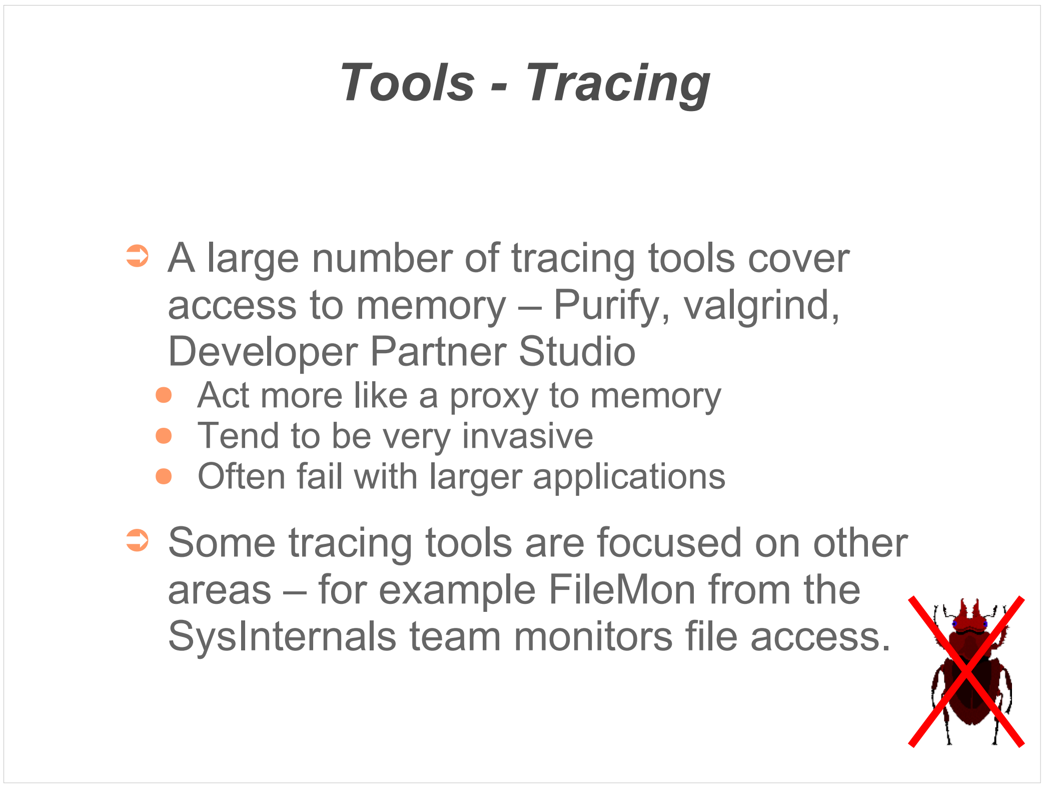


Tools - Tracing

- Tracing inside the process can be useful
- The interface may be that between the program and the OS, for example
  - strace/truss on Unix
  - Apimon/strace/ntTrace on Windows
- Tracing a subset of calls
- Tracing allows post-processing of output
- Can compare 'good' and 'bad' runs
- Whole machine traces can be useful, when possible

Internal interfaces are also a fruitful place to use tracing.
One interface common to all processes on the machine is the interaction with the operating system – there are typically a range of potential tools dealing with this interface.

Filtering of trace information can be done by the tool itself, or by post-processing the trace files. Graphical tools that let you home in on a particular item and then expand the context can be easy to use, or `grep` and `sed` can do the same job from the command line!

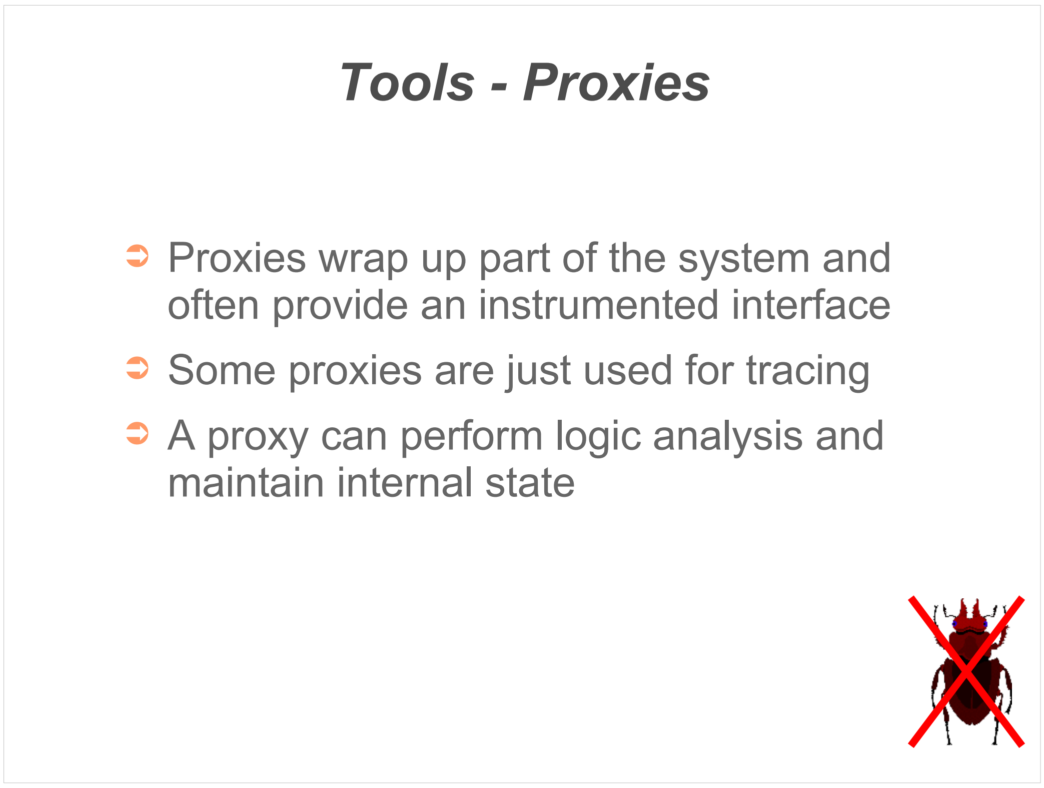


Tools - Tracing

- A large number of tracing tools cover access to memory – Purify, valgrind, Developer Partner Studio
  - Act more like a proxy to memory
  - Tend to be very invasive
  - Often fail with larger applications
- Some tracing tools are focused on other areas – for example FileMon from the SysInternals team monitors file access.

Memory is a key resource, especially for languages such as C++ that contain pointers.

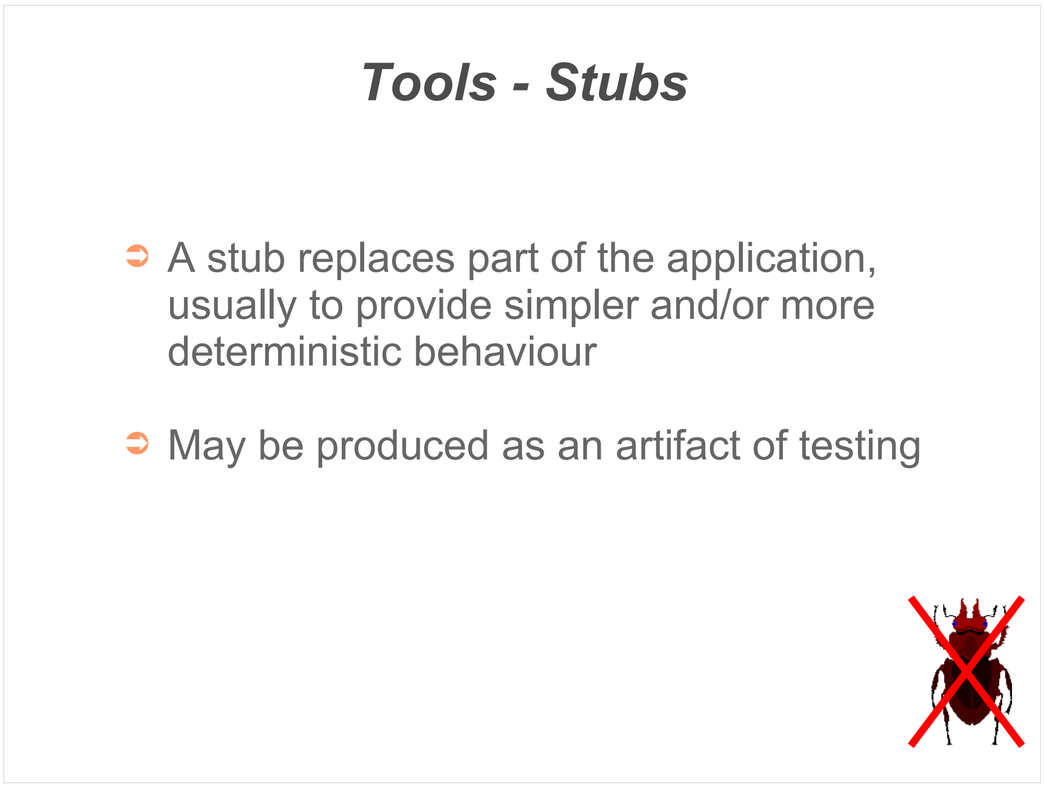
Memory problems are very hard to track down, mostly because of the lack of localisation.



Tools - Proxies

- Proxies wrap up part of the system and often provide an instrumented interface
- Some proxies are just used for tracing
- A proxy can perform logic analysis and maintain internal state

Distributed applications lend themselves very nicely to debugging with proxies.
A proxy can often be inserted into the application, even in production, and provide a way to identify which component was at fault.
In some cases the proxy can even detect and resolve the fault without needing to change the code. This can be useful for short-term fixes or when the faulty component cannot easily be changed.

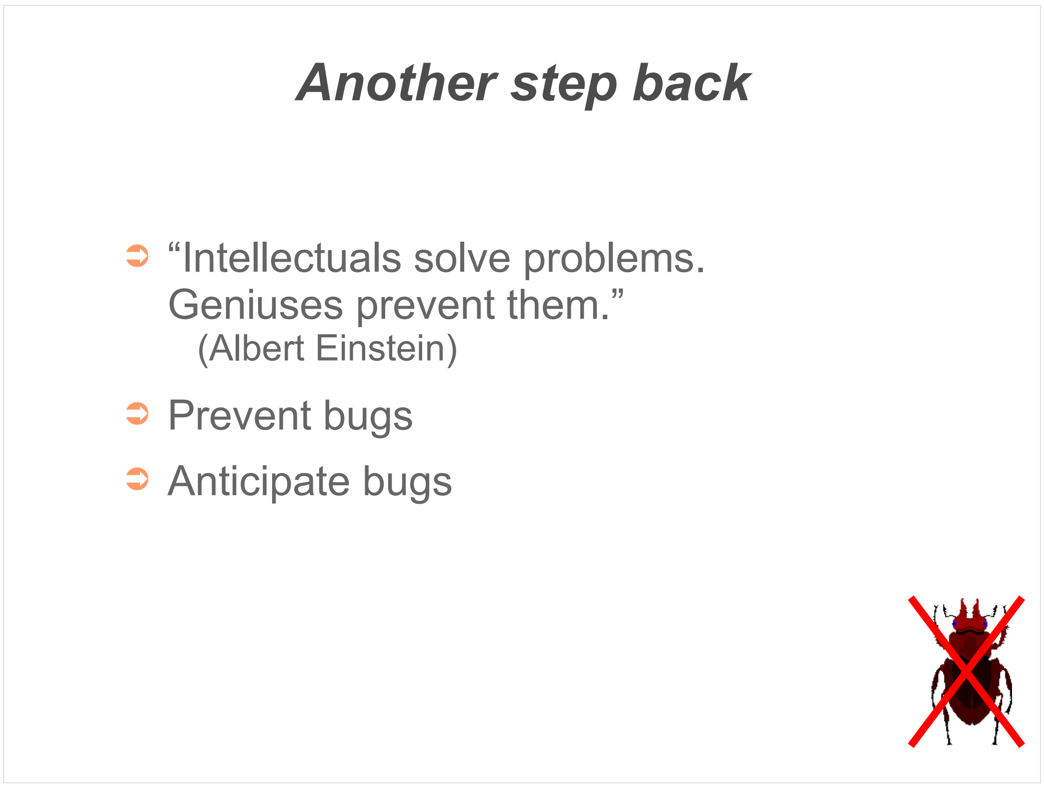


Tools - Stubs

- A stub replaces part of the application, usually to provide simpler and/or more deterministic behaviour
- May be produced as an artifact of testing

Stubs are useful for testing as they enable part of the application to be tested in isolation. This can help identify the failing component and provide a simple way to reproduce and examine the fault.

They are useful for other reasons too – such as producing demonstration versions or supporting independent development streams.

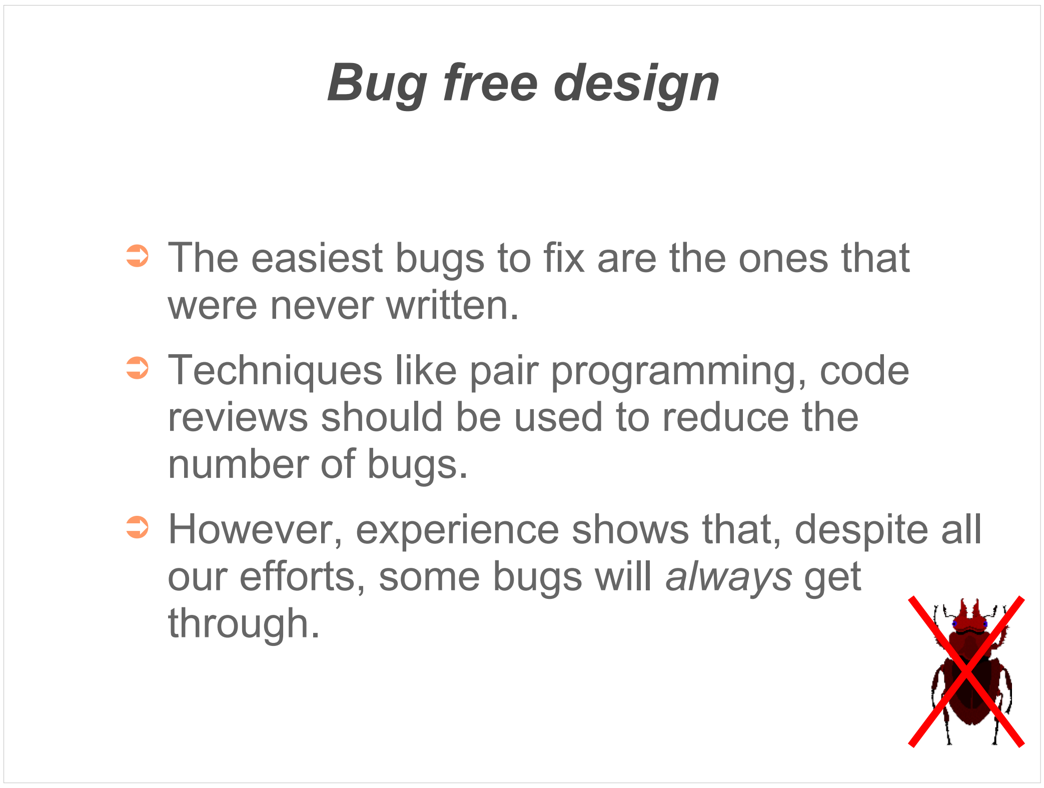


Another step back

- "Intellectuals solve problems. Geniuses prevent them."
  (Albert Einstein)
- Prevent bugs
- Anticipate bugs

Can we, rather than heroically solving difficult debugging tasks, prevent the need to find bugs?

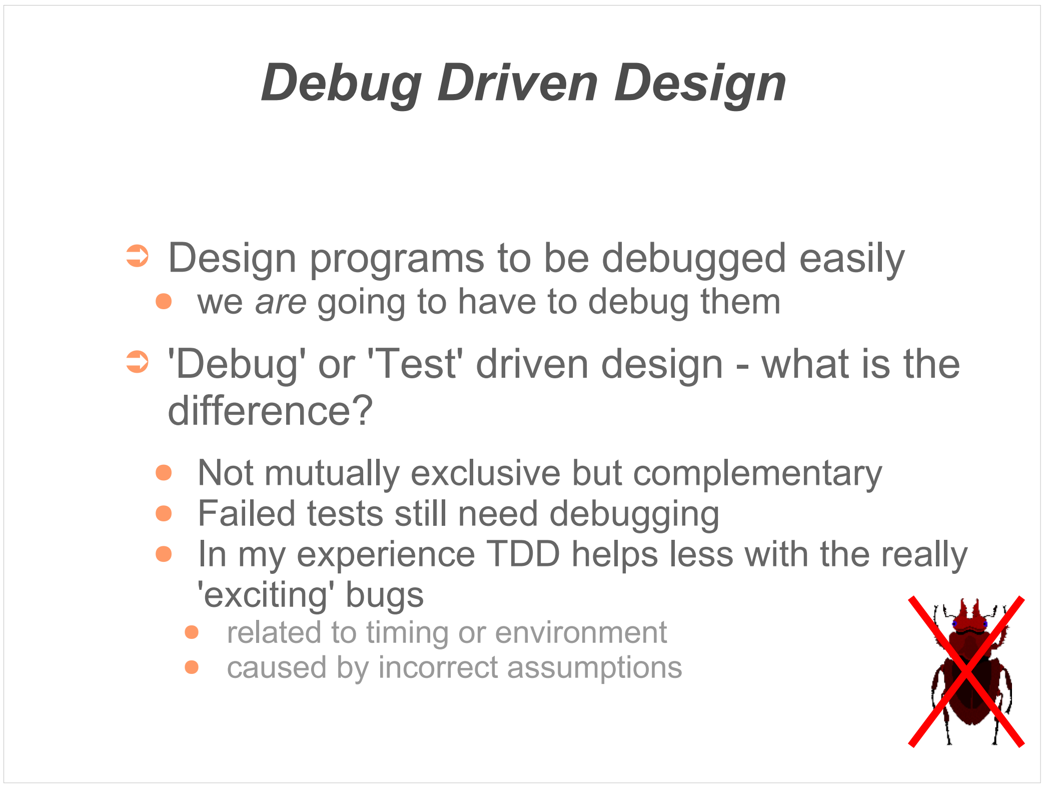
The optimal technique is to prevent bugs being written in the first place; the second line of defence is to plan for debugging ahead of the event.



Bug free design

- The easiest bugs to fix are the ones that were never written.
- Techniques like pair programming, code reviews should be used to reduce the number of bugs.
- However, experience shows that, despite all our efforts, some bugs will *always* get through.

There is a lot written about reducing the number of bugs in code. This is a laudable aim, but I'm not planning on adding to the body of knowledge in this presentation.

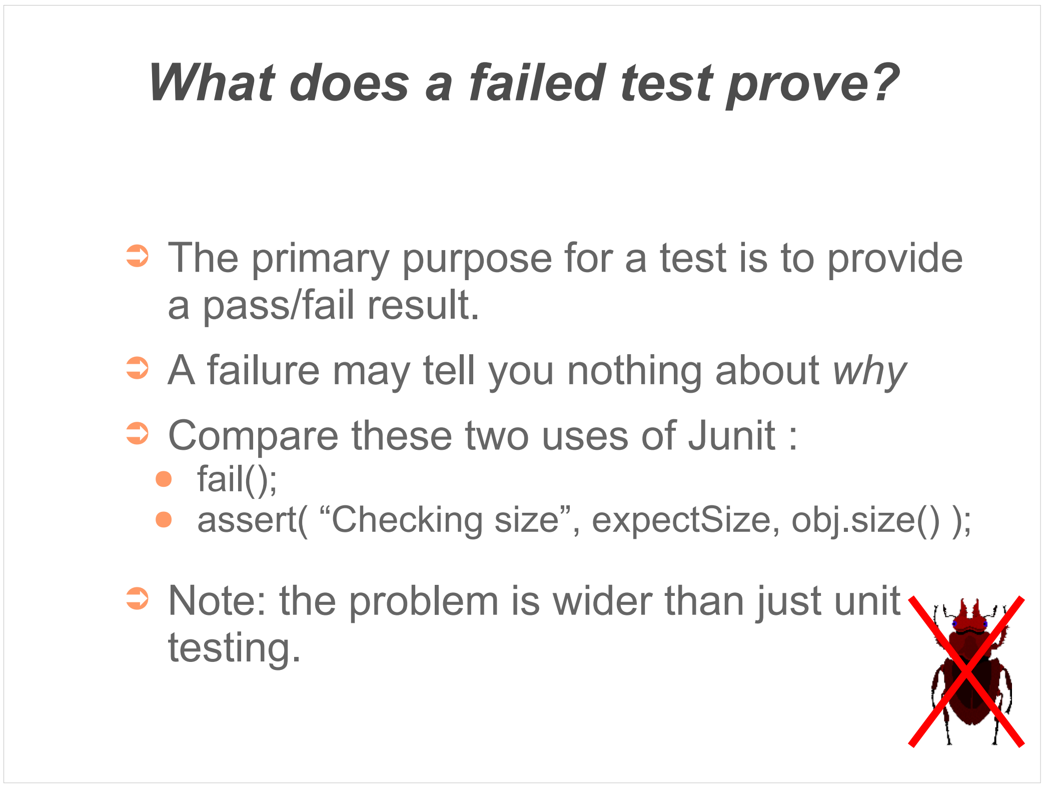


Debug Driven Design

- Design programs to be debugged easily
  - we *are* going to have to debug them
- 'Debug' or 'Test' driven design - what is the difference?
  - Not mutually exclusive but complementary
  - Failed tests still need debugging
  - In my experience TDD helps less with the really 'exciting' bugs
    - related to timing or environment
    - caused by incorrect assumptions

Classically test driven development starts by writing a test that fails, then fixing the bug at which point the test succeeds. It does not necessarily help much with finding the bug.

Debug driven development focuses on the fixing process – simplifying the tasks performed when code under test fails.



What does a failed test prove?

- The primary purpose for a test is to provide a pass/fail result.
- A failure may tell you nothing about *why*
- Compare these two uses of Junit :
  - fail();
  - assert(“Checking size”, expectSize, obj.size());
- Note: the problem is wider than just unit testing.

My frustration with testing (whether unit testing or other system testing) is with what happens when a test fails.

Finding the problem is useful, but fixing the problem is even better. Focusing on testing should include looking at what has to be done when programs fail tests.

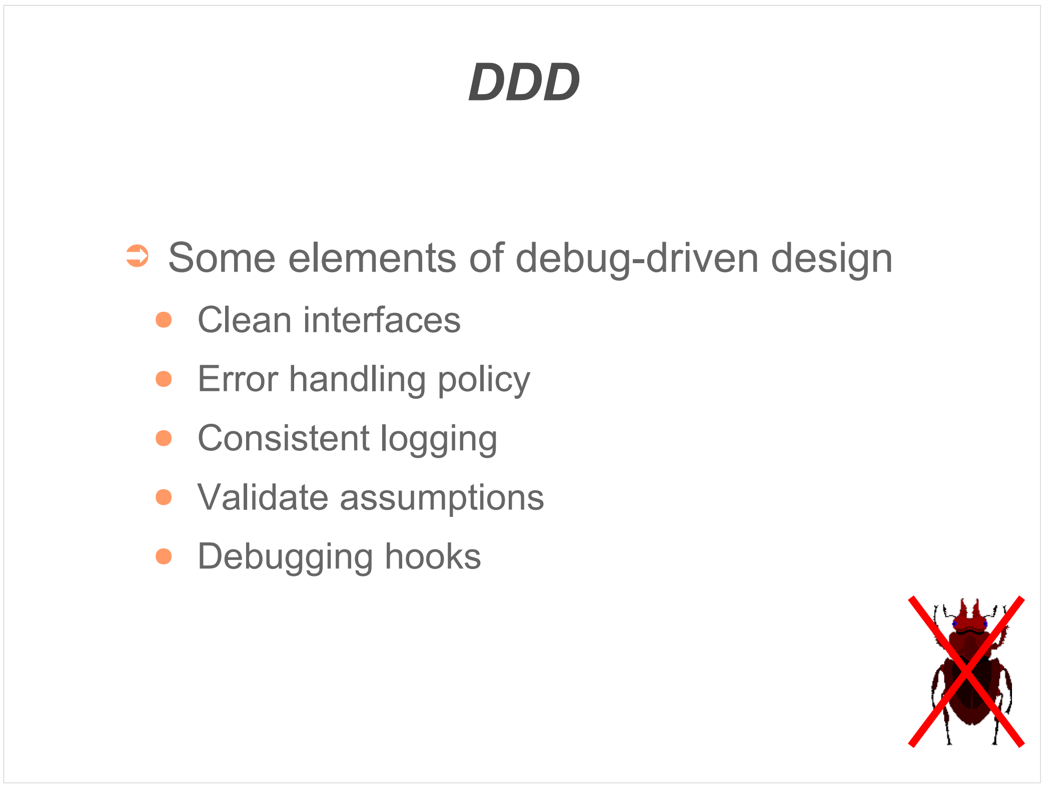
DDD

- How can we write programs that help us debug them?
- Reflect on what questions you will need answering to diagnose likely problems
- Can we detect the infection earlier?
- Can we catch the fault?

There are things we can do in the code to assist with analysis when we have a symptom of a bug; there may also be things we can do within the program itself to help move back up the cause-effect tree to detect infected code or even catch the underlying fault.

A key component is reflecting on how you are going to debug the code – how easy will it be; what sort of things are most likely to go wrong and what information will you need?

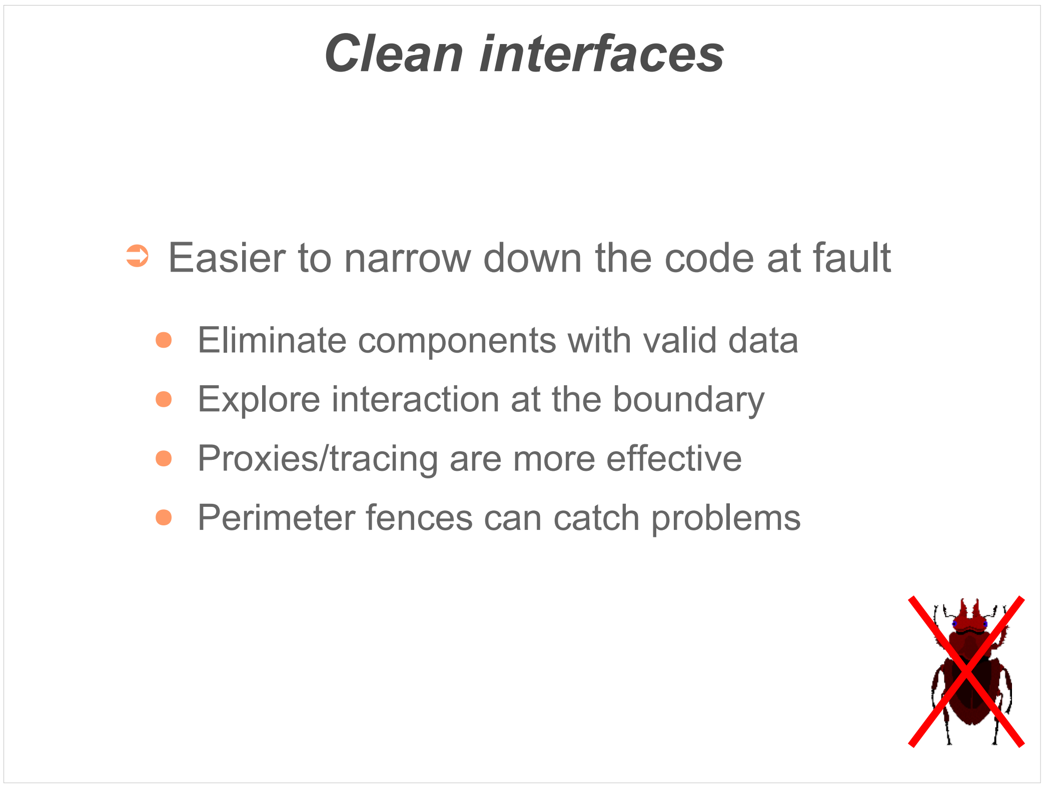
Often developers can have a good idea, based on previous experience, where the most likely places are for bugs.



DDD

- Some elements of debug-driven design
  - Clean interfaces
  - Error handling policy
  - Consistent logging
  - Validate assumptions
  - Debugging hooks

Somewhat arbitrarily I've identified five main areas in which thoughts of debugging influence the design.
The five elements are interdependent – for example debugging hooks often require consistent logging to be really useful.

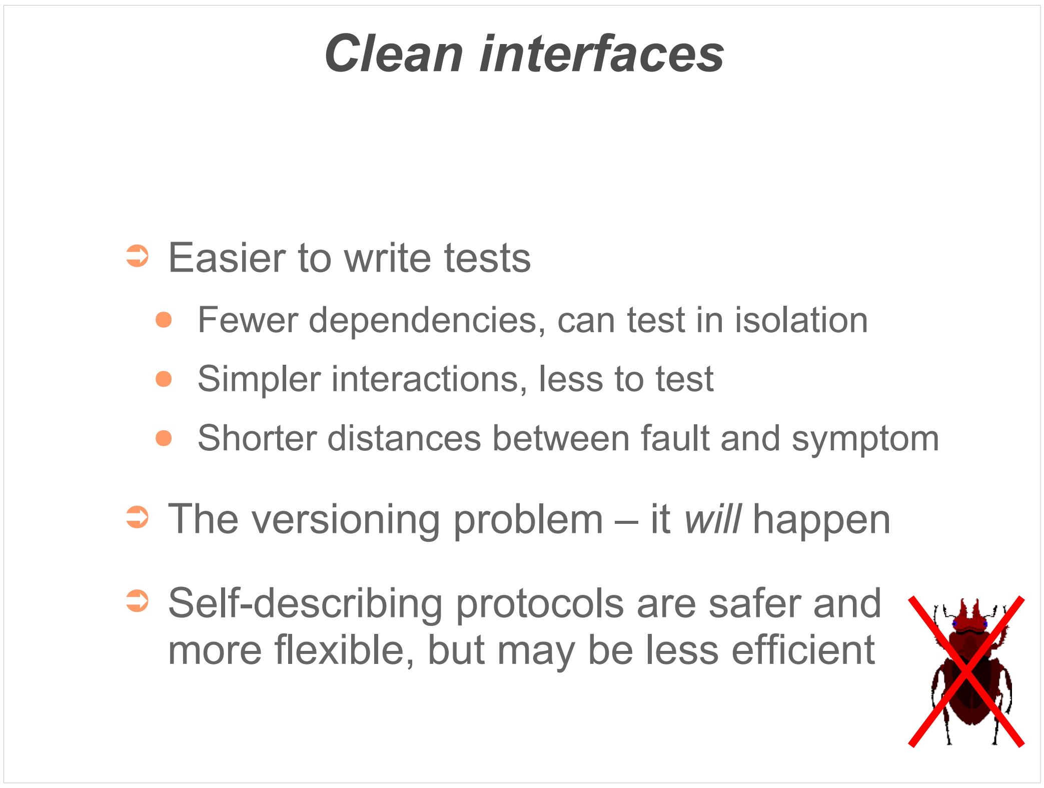


Clean interfaces

- Easier to narrow down the code at fault
  - Eliminate components with valid data
  - Explore interaction at the boundary
  - Proxies/tracing are more effective
  - Perimeter fences can catch problems

Keeping interfaces clean is a good design goal, but it also has clear advantages when it comes to debugging.

An example: separate processes may be cleaner than separate threads, as the interface is much more explicit.



Clean interfaces

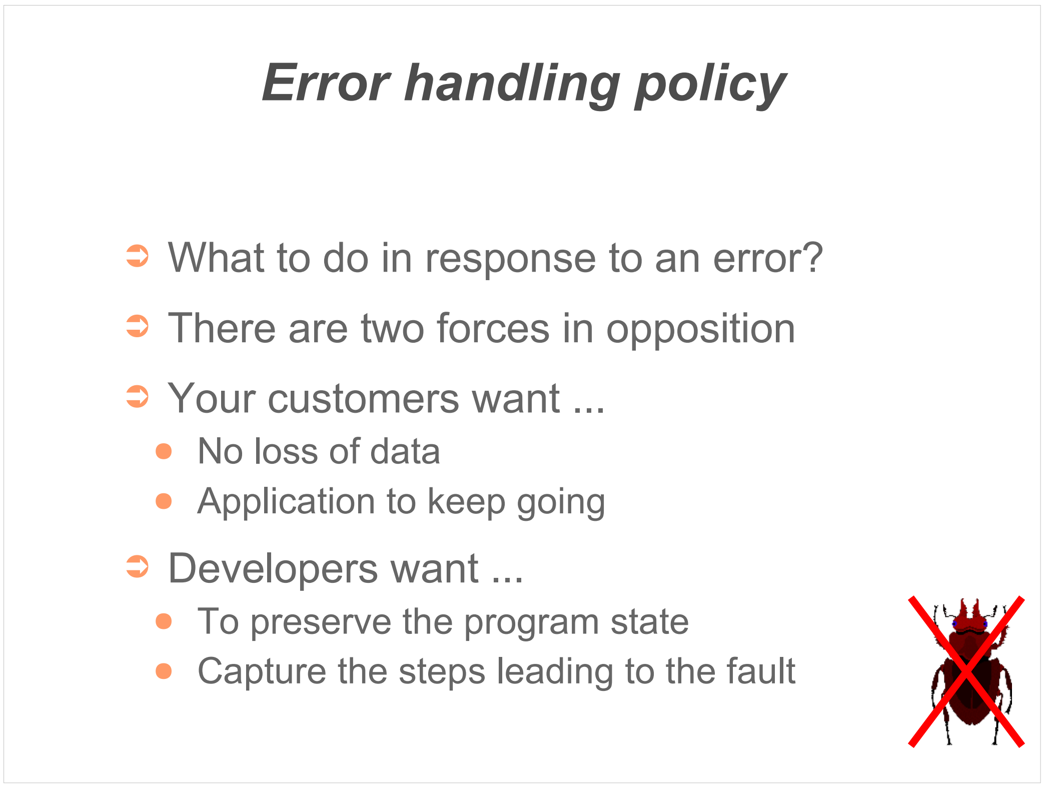
- Easier to write tests
  - Fewer dependencies, can test in isolation
  - Simpler interactions, less to test
  - Shorter distances between fault and symptom
- The versioning problem – it *will* happen
- Self-describing protocols are safer and more flexible, but may be less efficient

The are two main approaches to the versioning problem.

1) Verify the versions *match* – for example a version tag in the message, or adding a version to an interface name.

2) Detect the actual version at runtime and support multiple iterations of the protocol. This is more flexible, but can get quite complex.

For closely coupled systems people often hope they can avoid the problem, leading to subtle problems when a mix of components is accidentally installed.



Error handling policy

- What to do in response to an error?
- There are two forces in opposition
- Your customers want ...
  - No loss of data
  - Application to keep going
- Developers want ...
  - To preserve the program state
  - Capture the steps leading to the fault

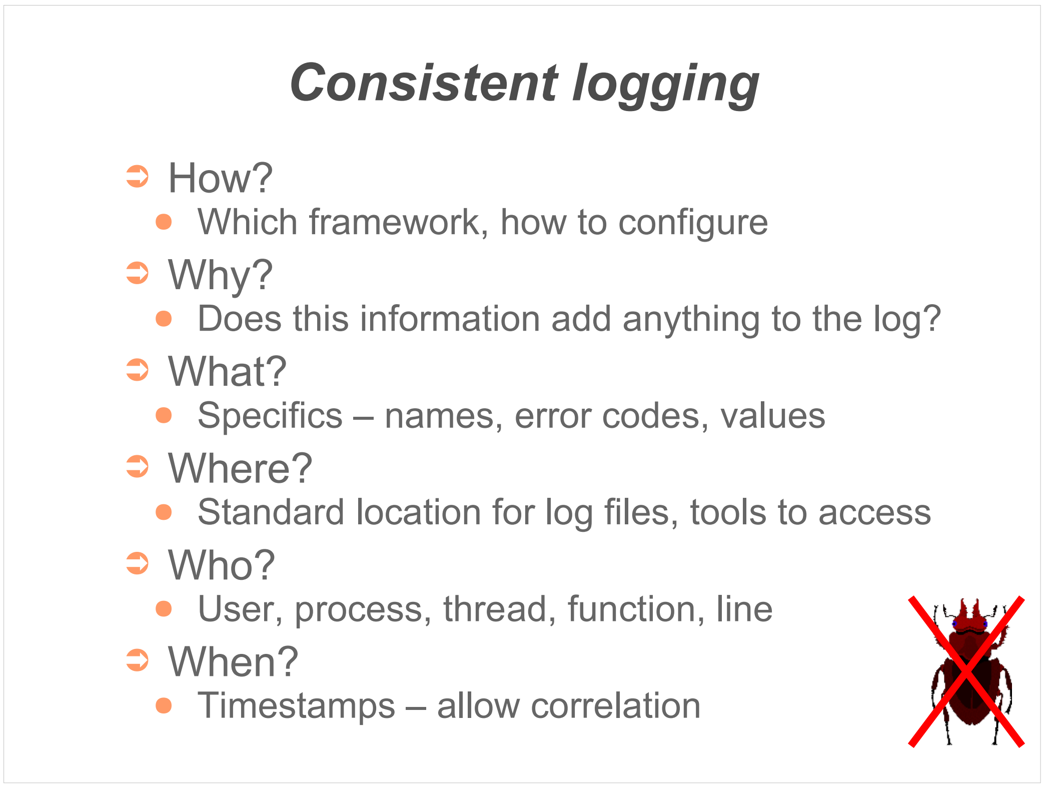
Error handling policies are hard to get right, partly because of the changing balance between the developers and customers as the code goes through the release cycle.

Specifics of policy depend on the application domain – is *a* (possibly wrong) result better or worse than *no* result ?

The `assert` problem – developers like being able to dive into the debugger early and there is low cost to the program stopping.

However this means the code *after* the assert has never been tested.

Some environments allow assertions to be ignored, this can be useful (opinions vary!)



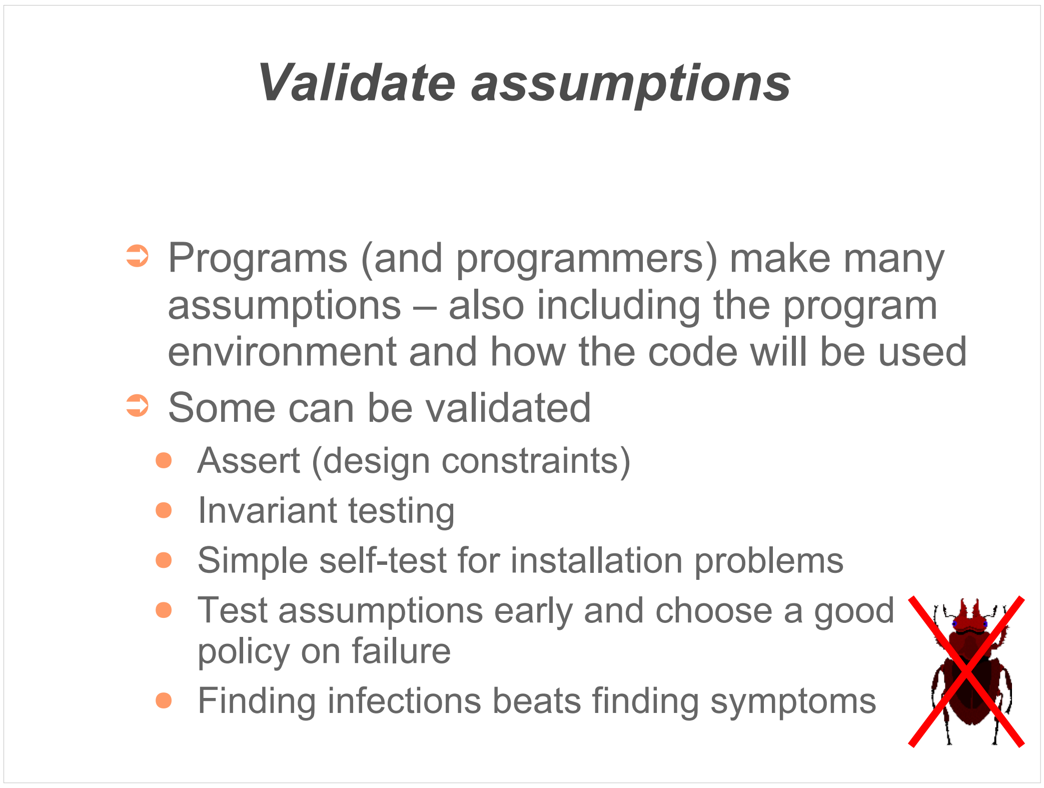
Consistent logging

- How?
  - Which framework, how to configure
- Why?
  - Does this information add anything to the log?
- What?
  - Specifics – names, error codes, values
- Where?
  - Standard location for log files, tools to access
- Who?
  - User, process, thread, function, line
- When?
  - Timestamps – allow correlation

It can be instructive to ask, after a fault has been reported and fixed, what could have made the log file(s) more useful in solving the problem.

There is a tension with (a) performance and (b) security.

If development and release log settings differ, check the release logs are still meaningful.



Validate assumptions

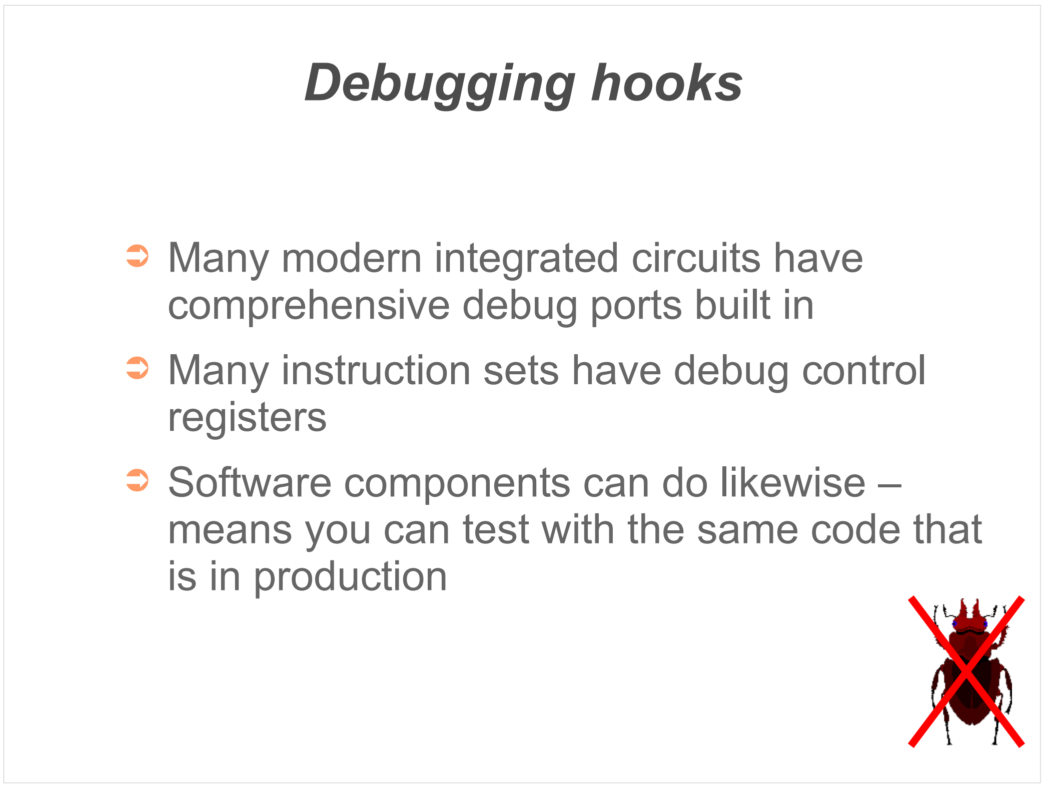
- Programs (and programmers) make many assumptions – also including the program environment and how the code will be used
- Some can be validated
  - Assert (design constraints)
  - Invariant testing
  - Simple self-test for installation problems
  - Test assumptions early and choose a good policy on failure
  - Finding infections beats finding symptoms

It is important to be aware of what is being assumed. This information needs to be documented – the best place is often in the code itself.

If design constraints are violated continuing program execution at all can be problematic – but see 'error handling' above...

If environmental assumptions prove false the program is still internally viable, so efforts should be made to report the problem and abort the function(s) affected.

For example, if there is no printer then trying to print should not abort the process nor should it prevent using the spell checker.



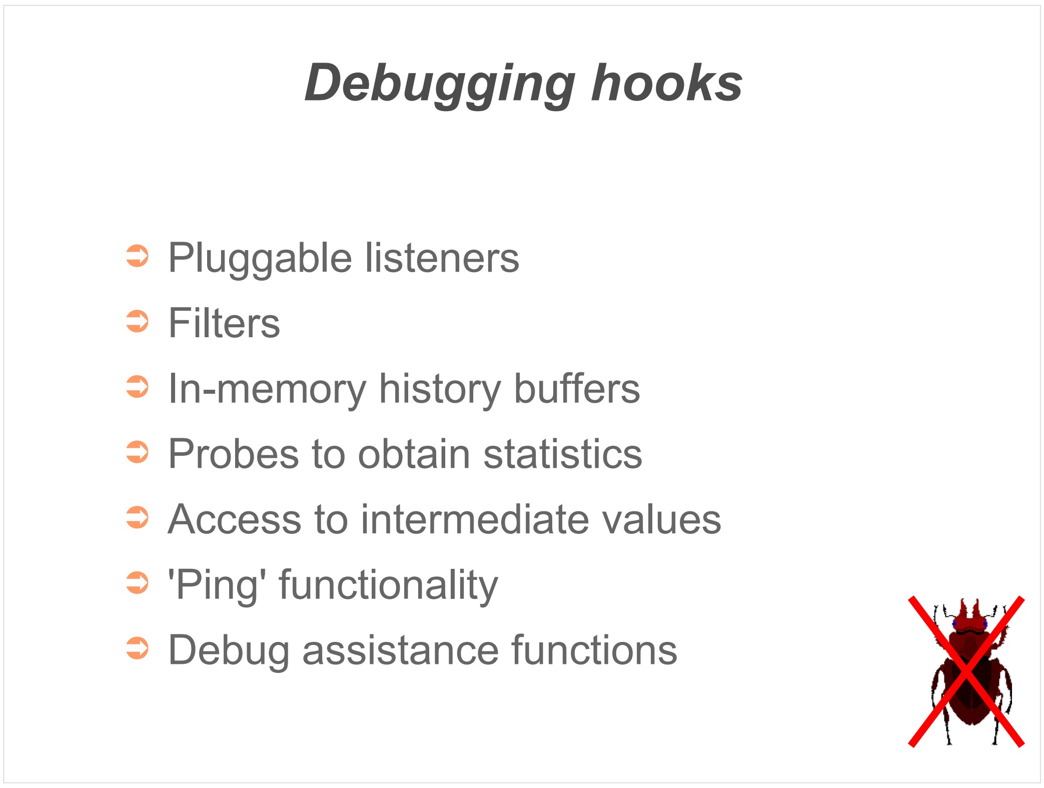
Debugging hooks

- Many modern integrated circuits have comprehensive debug ports built in
- Many instruction sets have debug control registers
- Software components can do likewise – means you can test with the same code that is in production

Why do chip manufacturers go to the (not insignificant) expense of adding debug ports to ICs?

Many of these reasons are equally valid for software.

You can get additional information about the problem in the environment in which it is detected.

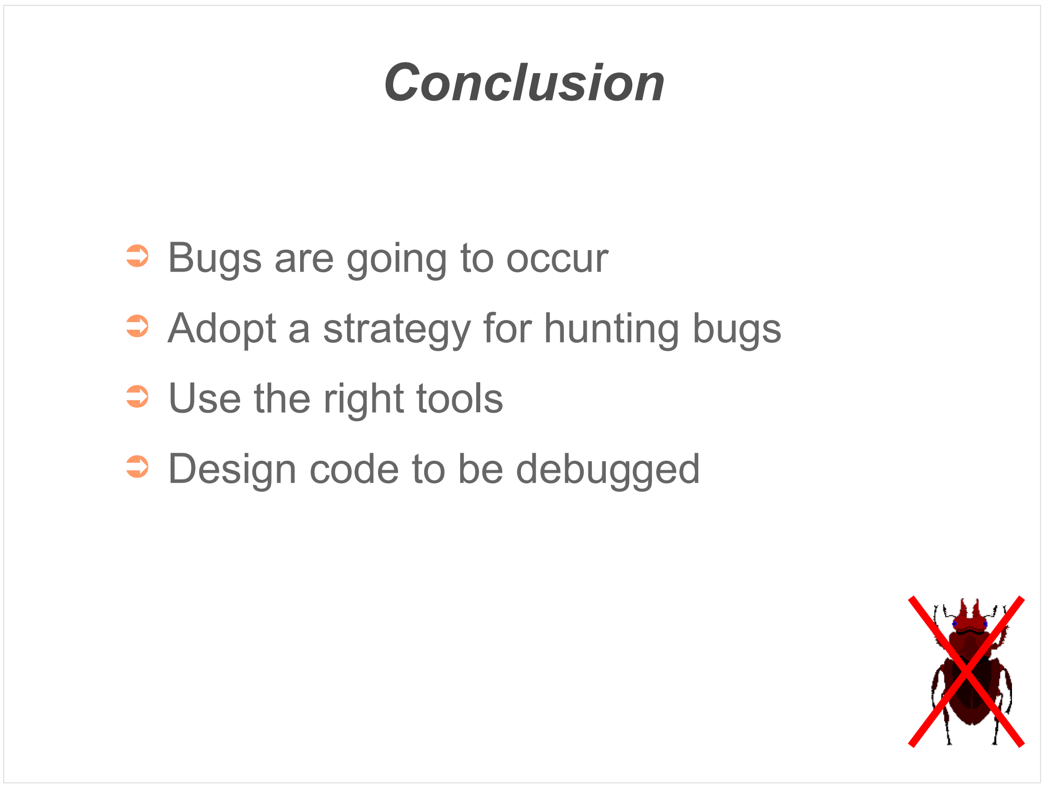


Debugging hooks

- Pluggable listeners
- Filters
- In-memory history buffers
- Probes to obtain statistics
- Access to intermediate values
- 'Ping' functionality
- Debug assistance functions

There are many places where functionality designed for debugging may be valuable.

Don't forget there is an associated cost, in terms of developer time, resource use and testing. However this can be offset by the reduced time spent debugging.



Conclusion

- Bugs are going to occur
- Adopt a strategy for hunting bugs
- Use the right tools
- Design code to be debugged

Debugging looks set to remain part of the IT landscape for some time to come.
If we apply sensible strategies, use the power of the computer itself and design our code to be debugged then we can significantly reduce the amount of time we have to spend fixing bugs.