



IBM Software Group

# Develop Faster, More Reliable C/C++ Code with Rational PurifyPlus



**PURIFY**



**QUANTIFY**



**PURECOVERAGE**



@business on demand software

Jon Sanders, [jmsanders@us.ibm.com](mailto:jmsanders@us.ibm.com)  
PurifyPlus Engineering Manager

# Typical agenda

- What is it?
- How will it improve my life?
- How do I get it?
- How does it work?
- Advanced moves



# Summary

```
C:> myapp  
...crash...
```

```
C:> purify myapp  
...bugs and bottlenecks are pinpointed automatically...
```

You fix the bugs and bottlenecks.

```
C:> myapp  
...works perfectly...
```

You're a rock star. Go to the beach.



# What is PurifyPlus?

- Help, my app...
  - ▶ Crashes intermittently
  - ▶ Uses too much memory
  - ▶ Runs too slowly
  - ▶ Isn't well tested
  - ▶ Is about to ship
  
- You've felt the pain - PurifyPlus is your painkiller
  - ▶ See what your code is really doing
  - ▶ Spend less time finding bugs and bottlenecks

# What is PurifyPlus?

- A set of runtime analysis tools



- ▶ Runtime error detection
- ▶ Automatically pinpoints hard-to-find bugs



- ▶ Application profiler
- ▶ Highlights performance bottlenecks



- ▶ Source code coverage analysis
- ▶ Helps avoid shipping untested code

# What is PurifyPlus?

- For everybody
  - ▶ Unix, Windows, C, C++, Java, .NET
  - ▶ Developers and testers
- Thorough
  - ▶ It even monitors components you don't have source for
- Easy to use
  - ▶ No recompile needed
  - ▶ VS, Robot, ClearQuest & ClearCase integrations
  - ▶ Rich CLI and batch mode for automation
- The standard by which others are measured

# Examples of runtime errors

- Read uninitialized memory – yields unpredictable results

```
void foo() {  
    int *ptr = new int;  
    cerr << "*ptr is " << *ptr << '\n'; //UMR: no value was set in ptr  
    delete[] ptr;  
}
```

- Access off the end of an allocated block – unpredictable/corrupting

```
void foo() {  
    int *ptr = new int[2];  
    ptr[0] = 0;  
    ptr[1] = 1;  
    for (int i=0; i <= 2; i++) {  
        cerr << "ptr[" << i << "] == " << ptr[i] << '\n'; //ABR when i is 2  
    }  
    delete[] ptr;  
}
```

# Examples of runtime errors

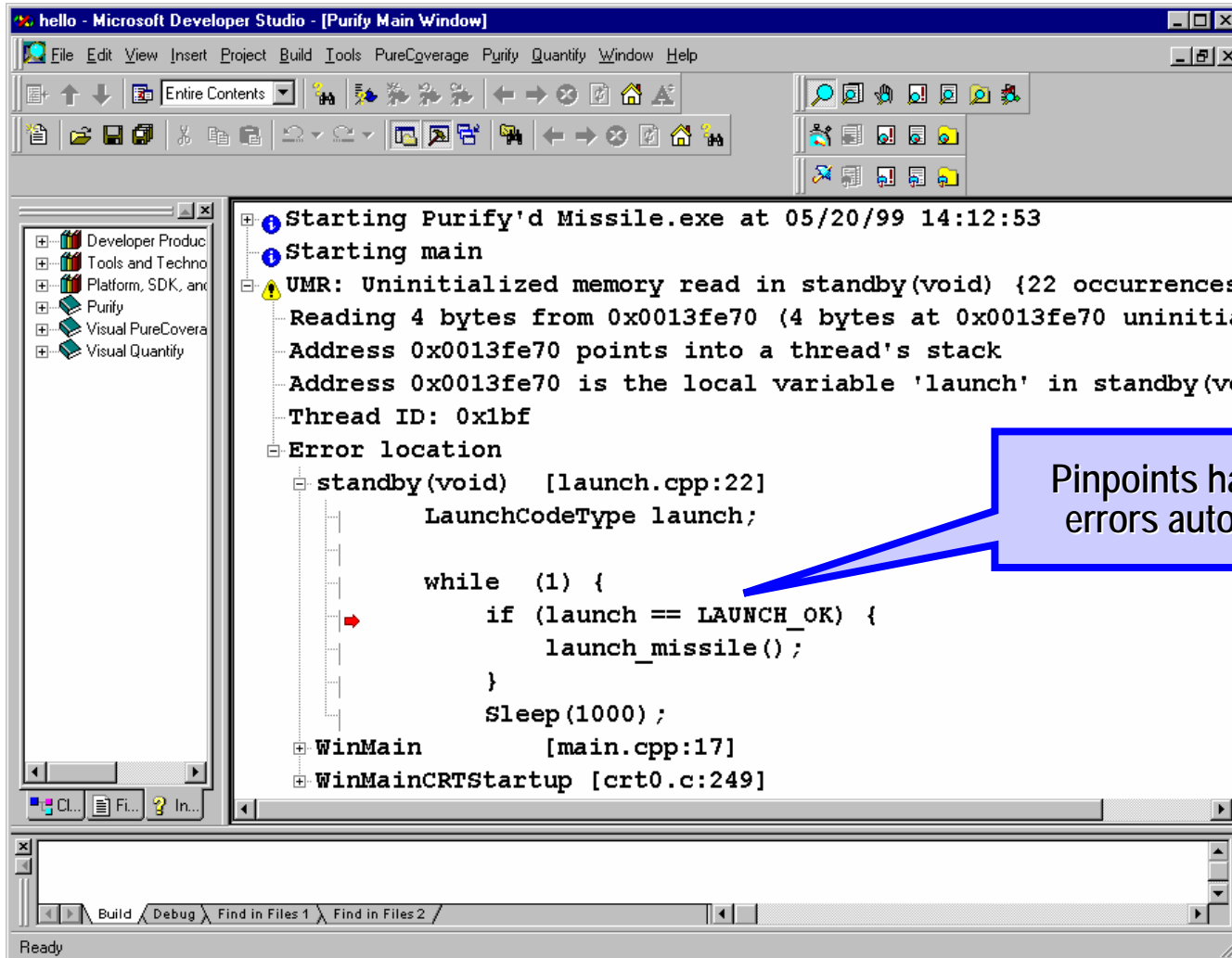
- And of course memory leaks

```
void foo() {  
    int *ptr = new int;    // MLK: ptr lost when foo returns  
    *ptr = 42;  
    cerr << "**ptr is " << *ptr << '\n';  
}
```

- Anyone can spot these by inspection
  - ▶ Now imagine these lines of code spread out over your multi-threaded componentized monster app



# What if a tool would just show you the bugs?



# And show you the bottlenecks

**Function Detail: hello.exe**

Function: WinMain  
 Calls: 1  
 Function time: 0.01 (0.00% of Focus)  
 F+D time: 18,867.88 (52.23% of Focus)  
 Avg F time: 0.01  
 Min F time: 0.01  
 Max F time: 0.01

**Function List: hello.exe**

Function	Calls	Function time	F+D time	F time (% of Focus)	F+D time (% of Focus)	Avg F time	Min F time
SetTimer	5	3,181.21	3,181.21	8.81	8.81	636.24	6.7
GetCharABCWidthsl	1	3,158.43	3,158.43	8.74	8.74	3,158.43	3,158.43
ExitProcess	1	2,954.93	15,654.09	8.18	43.34	2,954.93	2,954.93
.moduleEntry.ole32	2	2,661.23	3,304.50	7.37	9.15	1,330.62	36.7
SetWindowRgn	1	1,180.44	1,994.67	3.27	5.52	1,180.44	1,180.44
GdiDrawStream	30	743.54	743.54	2.06	2.06	24.78	1.1
ScriptStringAnalyse	135	684.31	4,803.87	1.89	13.30	5.07	0.8
CreateCompatibleBit...	6	503.97	503.97	1.40	1.40	83.99	19.3
GetTextExtentExPol...	108	434.37	434.37	1.20	1.20	4.02	0.8
StretchDIBits	2	406.41	406.41	1.13	1.13	203.20	14.5
GetModuleHandleW	6	362.68	362.68	1.00	1.00	60.45	35.6
DeleteObject	27	321.02	323.09	0.89	0.89	11.89	0.7
GetWindowDC	30	314.69	314.69	0.87	0.87	10.49	0.9
.moduleEntry.msctfime	2	260.70	1,317.05	0.72	3.65	130.35	61.3
GdiDllInitialize	1	227.73	1,135.37	0.63	3.14	227.73	227.73
NtConnectPort	1	182.20	182.20	0.50	0.50	182.20	182.20
GetDC	3	156.84	156.84	0.43	0.43	52.28	2.9
BaseCheckAppcom...	2	122.11	122.11	0.34	0.34	61.06	42.3
IntersectClipRect	23	121.39	121.39	0.34	0.34	5.28	0.1
ExitThread	27	110.26	110.26	0.31	0.31	4.08	0.1

**Callers:**

Caller	Percent	Calls	Propagated time
WinMainCRTStartup	100.00	1	18,867.88

**Descendants:**

Descendant	Percent	Calls	Propagated time
LpkDrawTextEx	23.24	101	4,384.57
SetTimer	16.86	5	3,181.21
DrawTextExW	15.00	17	2,830.43
SetWindowRgn	10.57	1	1,994.67
.moduleEntry.msctfime	5.11	1	963.33
GdiDrawStream	3.94	30	743.54

**Call Graph: hello.exe**

Zoom: [Slider] Highlight: Node: Maximum Path to Root

Visible: 21/324 Highlighted: 4/4 WinMain [C:\Program Files\Rational\PurifyPlus\Quantify\Samples\hello.exe]

Highlights bottlenecks automatically



# And show you what you forgot to test

Rational PureCoverage - [Coverage Browser:java.exe]

File Edit View Settings Window Help

Module View File View

Coverage Item	Calls	Methods Missed	Methods Hit	% Methods Hit	Lines Missed	Lines Hit	% Lines Hit
Run @ 11/28/2000 14:41:16 LeakSample	32	0	8	100.00	5	81	94.19
(Unknown Directory)	32	0	8	100.00	5	81	94.19
(Unknown File)	22	0	1	100.00			
JVM Garbage Collector	22		hit				
LeakSample.java	10	0	7	100.00	5	81	94.19
LeakSample\$Action.<init>(LeakS...	1		hit		1	2	66.67
LeakSample\$Action.actionPerfor..	3		hit		0	6	100.00
LeakSample\$Process.<init>(Leal..	1		hit		0	5	100.00
LeakSample\$Process.run()	1		hit		1	13	92.86
LeakSample.<init>(java.lang.Strir..	1		hit		0	41	100.00
LeakSample.btnStart_Clicked(jav..	2		hit		3	12	80.00
LeakSample.main(java.lang.Strin..	1		hit		0	2	100.00

Summary of untested lines and functions

# Even a single untested line

The screenshot shows the Rational PureCoverage interface for the file `LeakSample.java`. The left pane shows the project structure. The main pane displays the source code with a coverage table on the left. A context menu is open over line 166, showing options like 'Hit Lines...', 'Missed Lines...', 'Dead Lines...', 'Summaries...', and 'Partially Hit Multi-block Lines...'. A blue callout box labeled 'Untested lines' points to line 166.

Lin	Cover	Source
2,4		<code>bytes.addElement (new byte[8196]);</code>
2,4		<code>cnt = 0; i &lt; cnt; i++</code>
2,210		<code>bytes.removeElementAt (0)</code>
221		<code>if (bLeakOnce == true)</code>
0	166	<code>    bLeakMemory = false;</code>
221		<code>txtFreeMemory.setText (String.valueOf (rt.</code>
221		<code>txtTotalMemory.setText (String.valueOf (rt</code>
648		<code>try {</code>
649		<code>    java.lang.Thread.sleep (100);</code>
0	177	<code>    } catch (InterruptedException e)</code>
		<code>    {</code>

Line: 185 of 201      Method: LeakSample\$Action.<init>(LeakSample)

# Many alternatives

- Perhaps you already use some of these runtime analysis tools:
  - ▶ NuMega BoundsChecker (Windows)
  - ▶ Valgrind (Linux)
  - ▶ ZeroFault (AIX)
  - ▶ Parasoft Insure++ (SCI)
  - ▶ GlowCode (Lite)
  - ▶ Rational Test RealTime (Embedded)
  - ▶ Intel vTune (Windows/Linux)
  - ▶ IBM tprof
  - ▶ Quest Jprobe (Java)
  - ▶ Borland Optimizelt (Java)
  - ▶ RAD Profiling (Java)
  - ▶ Etc.
  
- ▶ ... Mail me with use cases, design/capability wins, etc.
  - [jmsanders@us.ibm.com](mailto:jmsanders@us.ibm.com)
  
- Each has its pros and cons
  - ▶ Some people keep several in their arsenal
  - ▶ In general PurifyPlus
    - Goes deeper
    - Runs faster
    - Provides more analysis
    - Works for larger apps
    - Is easier to deploy and to automate
  - ▶ Perhaps I'm biased, but 100,000 users can't be too wrong



## Some internal users of PurifyPlus

- AIX JVM & AIX C++ compiler - ISL
  - ▶ Purify C++ on AIX
- Rational ClearQuest Web server - Adam Skwersky
  - ▶ Purify C++ on Windows
- Rational XDE – Matt Halls
  - ▶ Quantify and Purify C++ and Java on Windows
- Rational PurifyPlus team – Jon Sanders
  - ▶ Purify, Quantify and PureCoverage C++ on all platforms
- WAS Performance – Andrew Spyker
  - ▶ Quantify Java
- DB2 JDBC Universal Driver – Suja Viswesan
- WebSphere Commerce Server – Priti Shah
- WPLC – David Ogle
  - ▶ PureCoverage Java
  
- Ask your friends. Tell your friends!



# What about static analysis tools

- Static analysis tools are a great complement to Purify
- They can...
  - ▶ Find errors that you don't exercise in test cases
  - ▶ Find richer semantic errors, e.g. type safety
  - ▶ Find potential errors if calling patterns change
  - ▶ Analyze code sections before you have a working executable
- Static analysis tools have limitations
- They...
  - ▶ Only find errors in the code you have source for
    - Not in libraries that you pass bad data too or that are buggy
  - ▶ Can miss errors that are distant in time/space cause/effect
    - Or bury you in "possible" errors
  - ▶ Can take a long time to run
  - ▶ Are not for profiling or coverage
- Use both as appropriate

# How do I get PurifyPlus?

- Getting PurifyPlus takes about 15 minutes
- Download & install it from XL
  - ▶ [http://w3.ibm.com/software/sales/saletool.nsf/salestools/bt-rational\\$Rational\\_download](http://w3.ibm.com/software/sales/saletool.nsf/salestools/bt-rational$Rational_download)
  - ▶ Search for text “PurifyPlus Enterprise”, pick the eAssembly with the latest version (currently v7.0), and download the binary for your platform (Linux/UNIX or Windows)
- Get a license
  - ▶ [http://w3.ibm.com/software/sales/saletool.nsf/salestools/bt-rational\\$Rational\\_licensechoose](http://w3.ibm.com/software/sales/saletool.nsf/salestools/bt-rational$Rational_licensechoose)
  - ▶ “Option #1” - pointing at a floating license server - is the simplest move
  - ▶ For laptop users, this path still permits disconnected use for several days at a time
- Download & install the latest iFix/Fixpack (very important)
  - ▶ <http://www.ibm.com/products/finder/us/finders?pg=ddfindex&C1=5000583&C2=5000623&rcss=rtlprfypls>
  - ▶ Pick the most recently published download for your version of PurifyPlus



# Supported environments

- See datasheets

- ▶ <http://www-306.ibm.com/software/awdtools/purifyplus/unix/sysreq/>
- ▶ <http://www-306.ibm.com/software/awdtools/purifyplus/win/sysreq/>

OS	Processor	OS version	Compilers	Bits
Linux	x86/x64	RHEL2.1-4/SLES8,9	gcc	32 & 64
AIX	PowerPC	5.1-5.3	XLC 6, 7, 8	32 & 64
Solaris	SPARC	6-10	Studio 7-10, gcc	32 & 64
HPUX	PA-RISC	10.20-11iv2	aCC, gcc	32 & 64
IRIX	MIPS	6.5	SGI	32 & 64
Windows	x86	NT/2000/XP/2003	Microsoft	32

- ▶ Additionally supports Java on Windows and Solaris
- ▶ Additionally supports .NET and VB applications on Windows
- ▶ AIX 64 bit Quantify & PureCoverage is in early access - contact [krangara@in.ibm.com](mailto:krangara@in.ibm.com)
- ▶ Some of the older platforms above are no longer formally supported in current version, but either continue to work or the previous version of PurifyPlus can be downloaded (v2003.06.15) to run with them
- ▶ Select additional variants have best-effort support
  - E.g. other compiler/OS rev levels, Intel compiler
- ▶ More platforms in development
  - Contact [jmsanders@us.ibm.com](mailto:jmsanders@us.ibm.com)

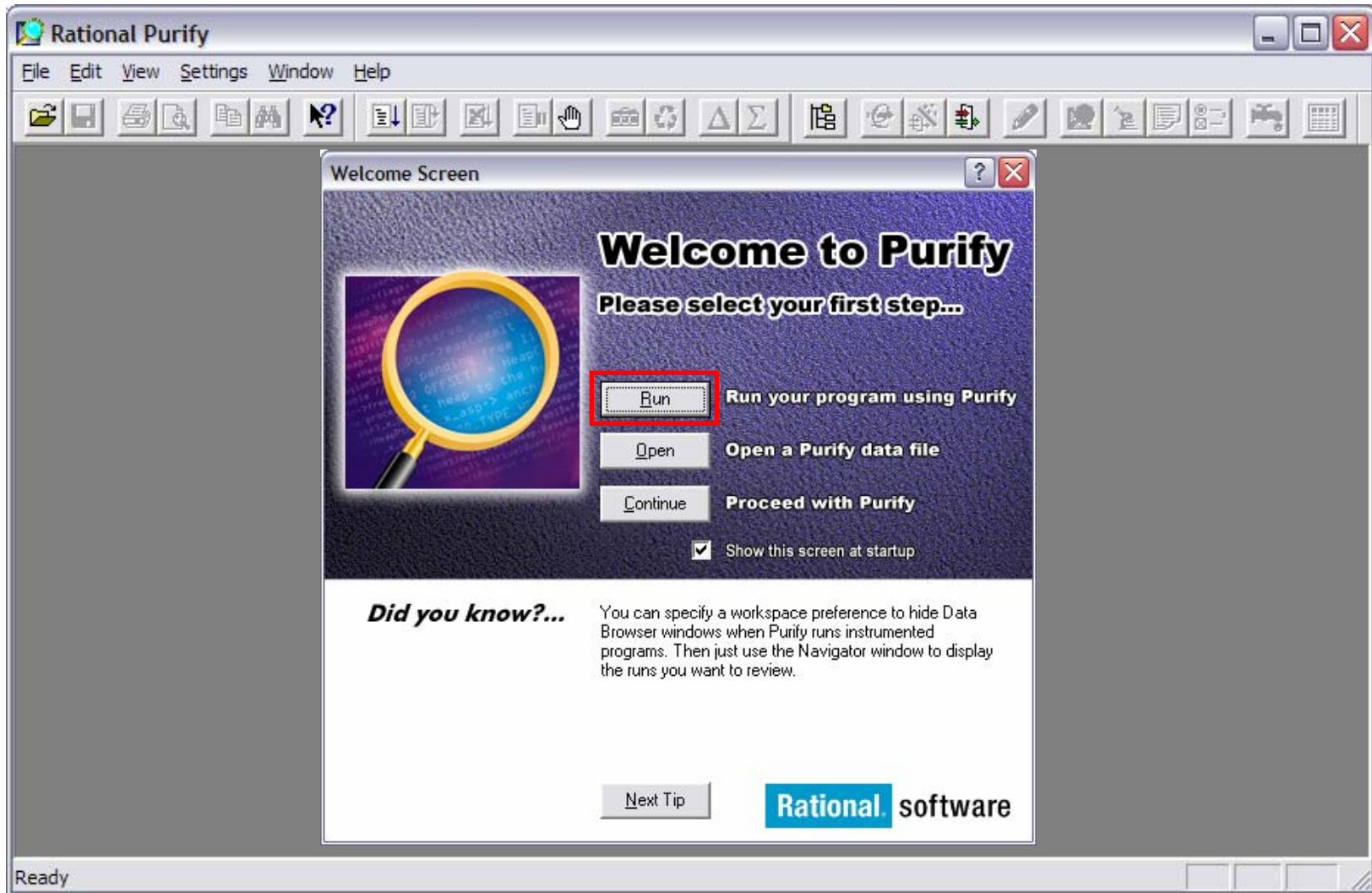


# How do I use PurifyPlus?

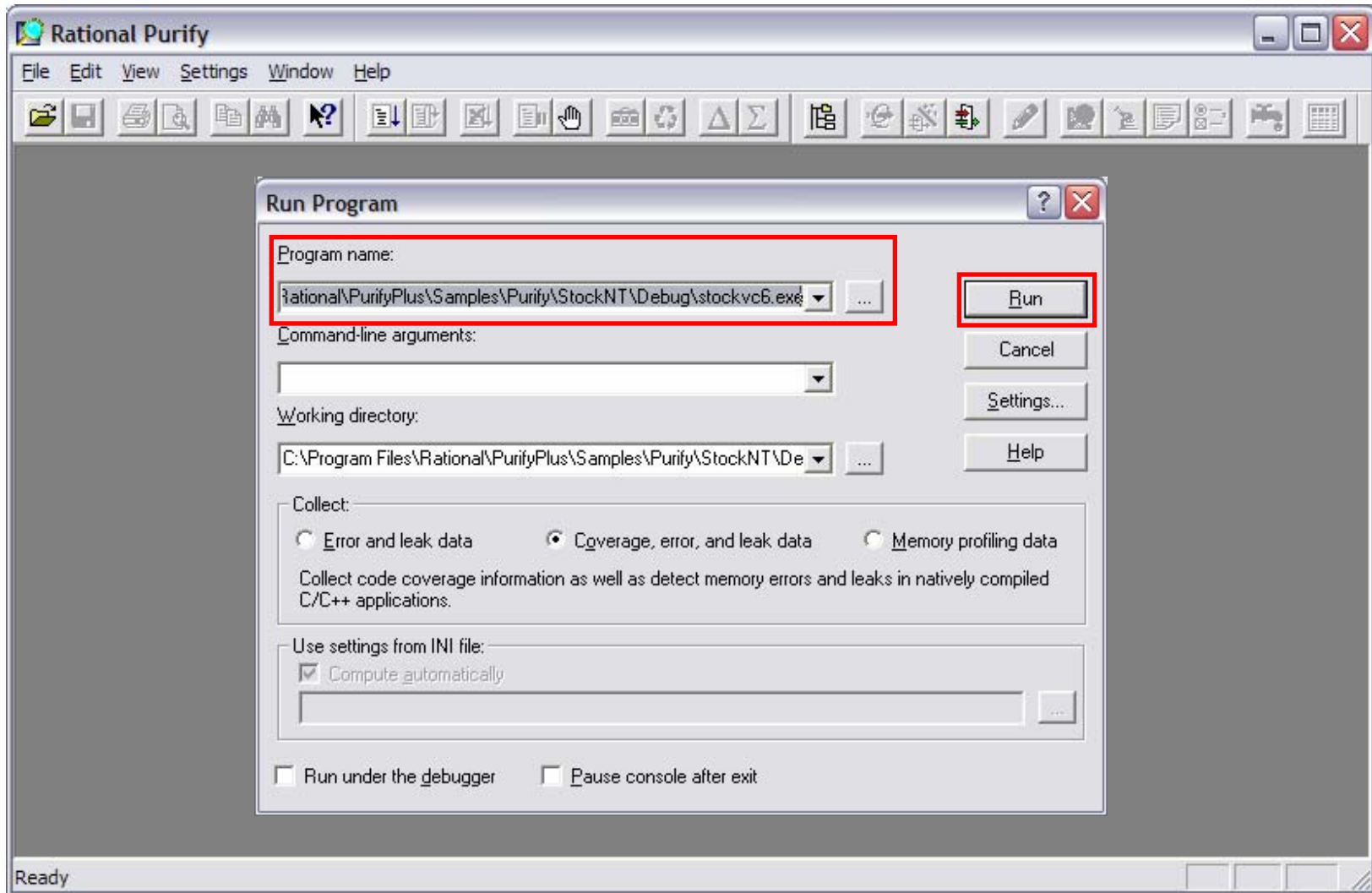
- The rest is really easy!
  - ▶ Just instrument and run
- On AIX, SGI and Windows executables are instrumented directly
  - ▶ C:> **purify** foo.exe
  - ▶ ksh% **purify** a.out
  - ▶ Or use the GUI
- On Linux, Solaris and HP-UX instrumentation is at link time
  - ▶ Prefix link line with “purify” in makefile

```
a.out: foo.c bar.c
    $(CC) $(FLAGS) -o $@ $?
a.out.pure: foo.c bar.c
    purify $(CC) $(FLAGS) -o $@ $?
```
- Quantify and Purecov are invoked similarly
  - ▶ Purify and PureCov can be used simultaneously

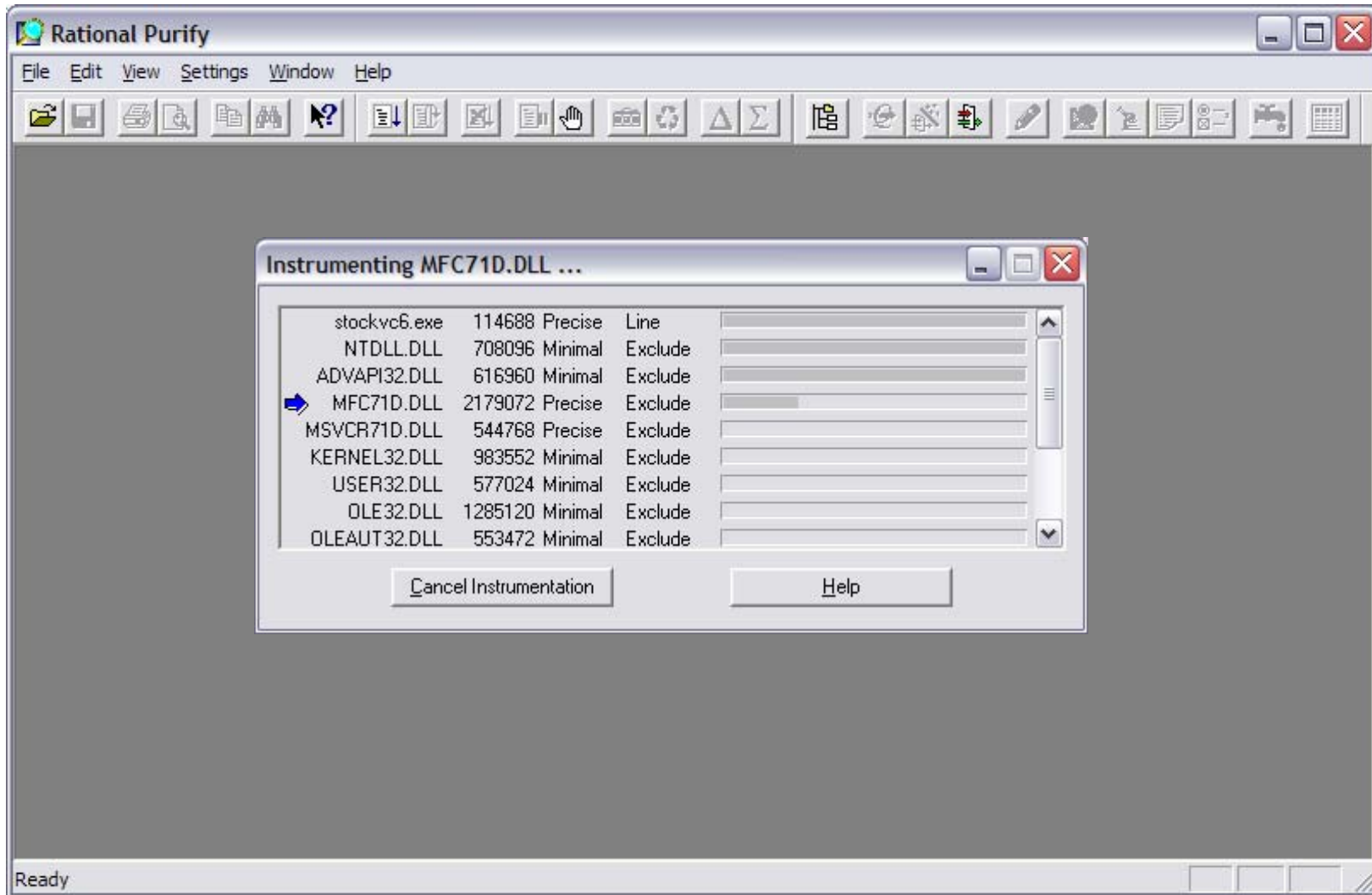
# Example: Launch Purify, press Run



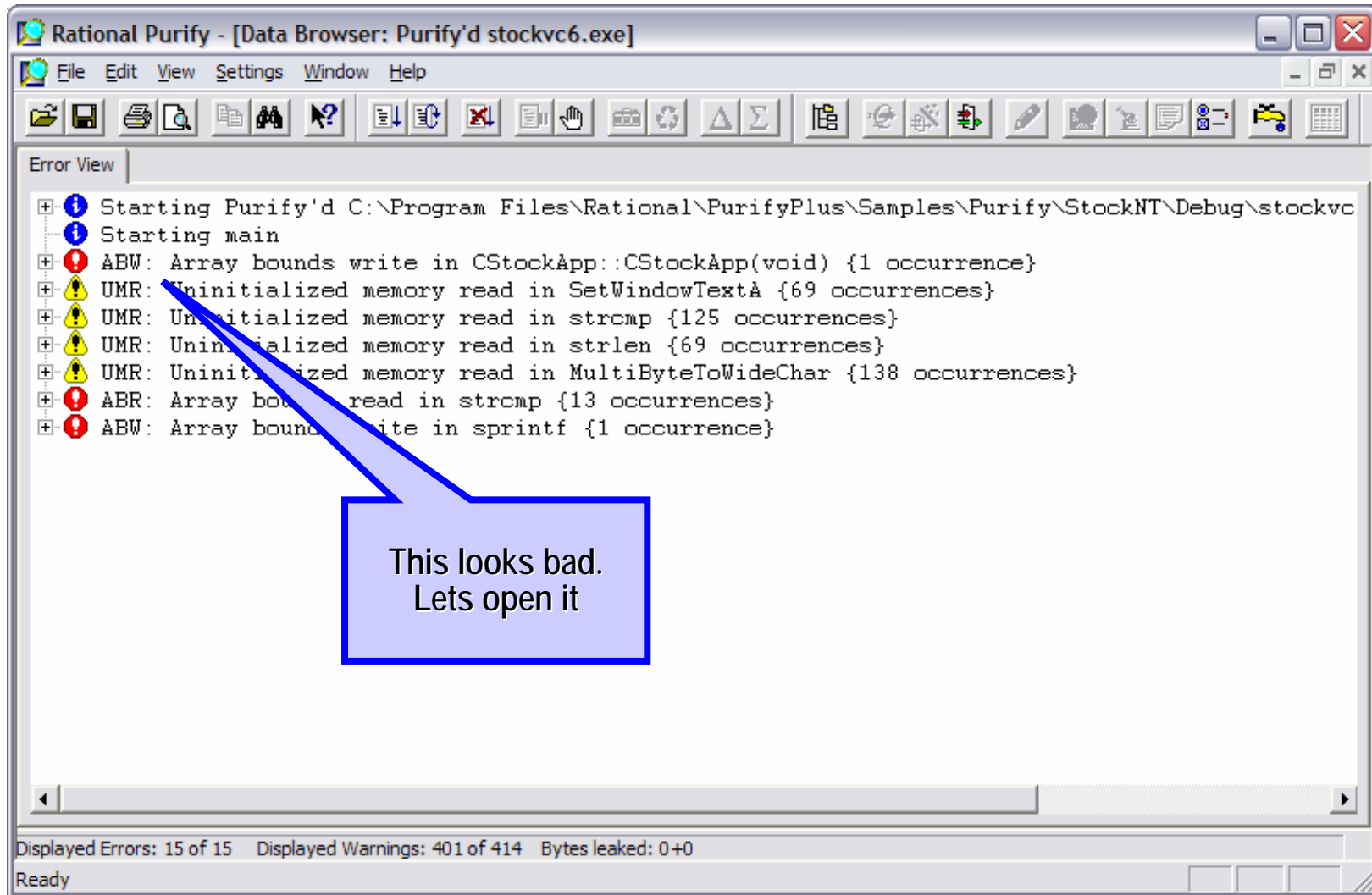
# Type in your program name, press Run



# Wait while Purify instruments all your code



# Your app runs, Purify logs errors alongside





# Open a message to see details

Rational Purify - [Data Browser: Purify'd stockvc6.exe]

File Edit View Settings Window Help

Error View

- Starting Purify'd C:\Program Files\Rational\PurifyPlus\Samples\Purify\StockNT\Debug\stock
- Starting main
- ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
  - Writing 1 byte to 0x00158ae8 (1 byte at 0x00158ae8 illegal)
  - Address 0x00158ae8 is 1 byte past the end of a 80 byte block at 0x00158a9
  - Address 0x00158ae8 points to a HeapAlloc'd block in the default heap
  - Thread ID: 0x1264
  - Error location
    - CStockApp::CStockApp(void) [c:\program files\rational\purifyplus\samples\purify\stock
    - \$E4 (C++ ctor/dtor) [c:\program files\rational\purifyplus\samples\purify\stocknt\stc
    - initterm [f:\vs70builds\3077\vc\crtbld\crt\src\crt0dat.c:599]
    - WinMainCRTStartup [f:\vs70builds\3077\vc\crtbld\crt\src\crtexe.c:336]
  - Allocation location
    - HeapAlloc [C:\WINDOWS\system32\KERNEL32.dll]
    - CStockApp::CStockApp(void) [c:\program files\rational\purifyplus\samples\purify\stock
    - \$E4 (C++ ctor/dtor) [c:\program files\rational\purifyplus\samples\purify\stocknt\stc
    - initterm [f:\vs70builds\3077\vc\crtbld\crt\src\crt0dat.c:599]
    - WinMainCRTStartup [f:\vs70builds\3077\vc\crtbld\crt\src\crtexe.c:336]
- UMR: Uninitialized memory read in SetWindowTextA {69 occurrences}
- UMR: Uninitialized memory read in strcmp {125 occurrences}
- UMR: Uninitialized memory read in strlen {69 occurrences}
- UMR: Uninitialized memory read in MultiByteToWideChar {120 occurrences}

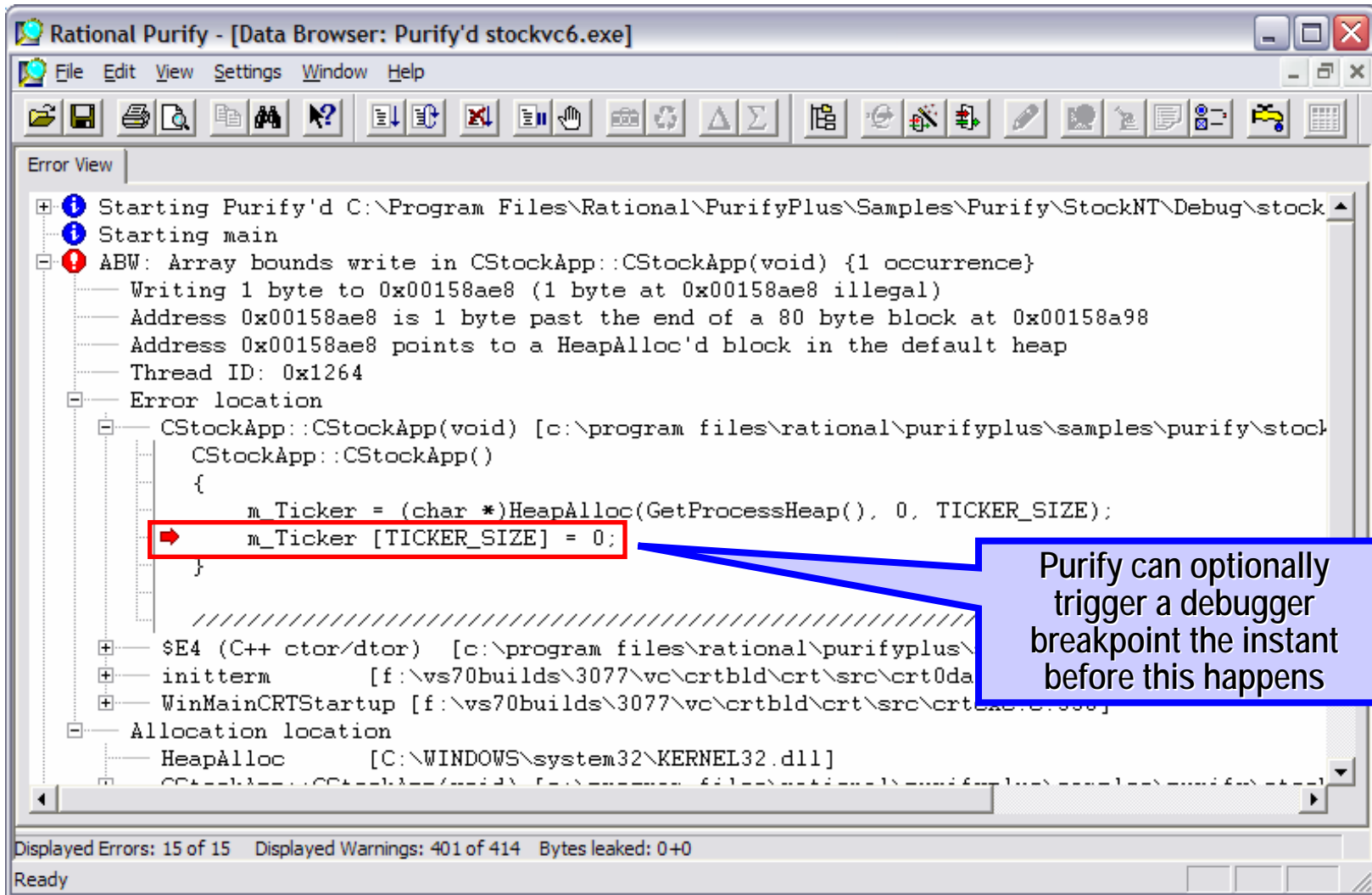
Displayed Errors: 15 of 15    Displayed Warnings: 401 of 414    Bytes leaked: 0+0

Ready

**Examine the buggy code**

**All the salient details:**  
 - what was accessed  
 - from where  
 - allocated when

# Pinpoints error to line of source



The screenshot shows the Rational Purify interface. The 'Error View' pane displays a list of errors. The first error is expanded, showing the following details:

- Starting Purify'd C:\Program Files\Rational\PurifyPlus\Samples\Purify\StockNT\Debug\stock
- Starting main
- ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
  - Writing 1 byte to 0x00158ae8 (1 byte at 0x00158ae8 illegal)
  - Address 0x00158ae8 is 1 byte past the end of a 80 byte block at 0x00158a98
  - Address 0x00158ae8 points to a HeapAlloc'd block in the default heap
  - Thread ID: 0x1264
- Error location
  - CStockApp::CStockApp(void) [c:\program files\rational\purifyplus\samples\purify\stock
  - CStockApp::CStockApp()
    - {
    - m\_Ticker = (char \*)HeapAlloc(GetProcessHeap(), 0, TICKER\_SIZE);
    - m\_Ticker [TICKER\_SIZE] = 0;**
    - }
- Allocation location
  - HeapAlloc [C:\WINDOWS\system32\KERNEL32.dll]

The line `m_Ticker [TICKER_SIZE] = 0;` is highlighted with a red box. A blue callout box points to this line with the text: "Purify can optionally trigger a debugger breakpoint the instant before this happens".

At the bottom of the window, the status bar shows: "Displayed Errors: 15 of 15 Displayed Warnings: 401 of 414 Bytes leaked: 0+0 Ready".



# How does it work?

- The “tech” part of the talk
- All it does is:
  - ▶ Take your compiled application apart
  - ▶ Find interesting places to insert probes
  - ▶ Put your application back together again
  - ▶ Run it and make lights blink



# Instrumentation using Object Code Insertion - OCI

- Stretching your code
  - ▶ Code- and partial data-flow analysis on exe/dll
    - Separate code from data
    - Find all cross-references in code and data
    - Heuristics tuned to compiler behaviors
  - ▶ Insert inline assembly code or jumps at locations like
    - Every memory read/write
    - Function calls, function entries
    - Source line, basic blocks
  - ▶ Modify imports to reference instrumented dlls
  - ▶ “Wrap” interesting APIs
    - malloc/free, New/Delete, LoadLibrary/dlopen
  - ▶ Bind a runtime support library

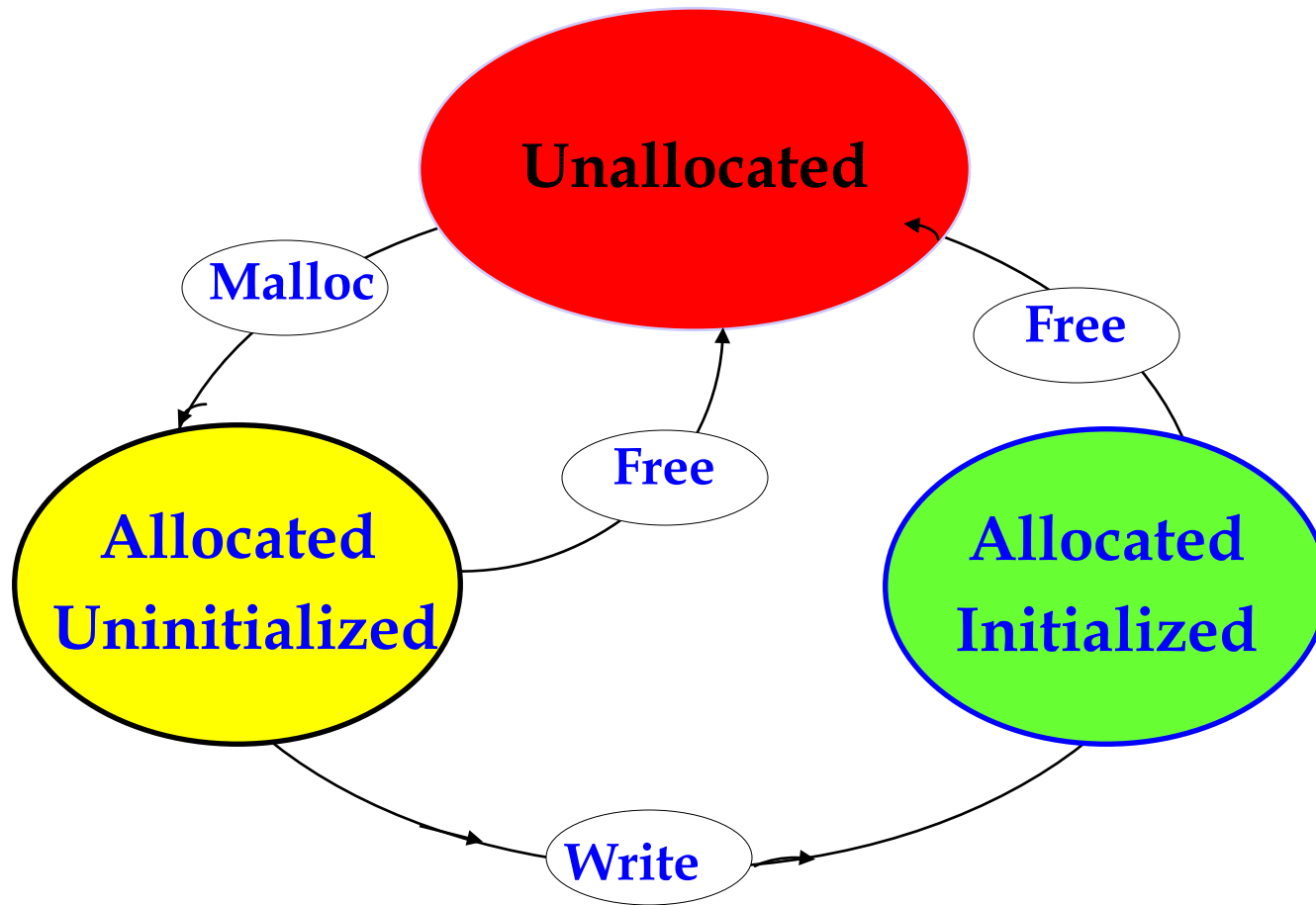
# Detecting memory leaks

- Purify's best known feature
- Simple GC-like algorithm
  - ▶ Maintain list of all allocated blocks in all heaps
    - And callchain of allocator
  - ▶ On demand, scan memory for all pointers
    - Starting from anchors – stack, statics, registers
  - ▶ Any block not pointed to is “leaked” (MLK/PLK)
    - All other blocks are “memory in use” (MIU)

# Memory access errors

- Hard to find bugs
  - ▶ ABR/ABW – array bounds read/write
  - ▶ FMR/FMW – free'd memory read/write
  - ▶ UMR – uninitialized memory read
- Technique
  - ▶ Track state of each byte in process address space
    - Red = logically unallocated to app
    - Yellow = allocated (malloc/new) but not written to yet (uninitialized)
    - Green = allocated and initialized
  - ▶ Add red-zone to ends of allocated blocks
    - To catch array out of bounds errors
  - ▶ Monitor every read and write instruction

# Purify's memory state tracking

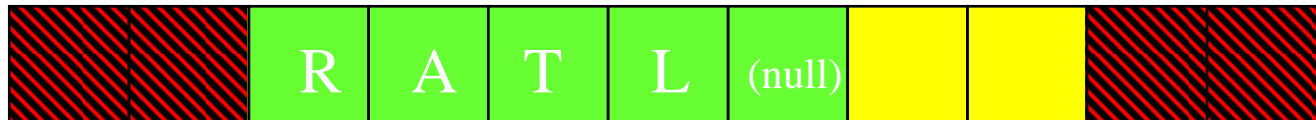


## Purify's array bounds detection

- Purify inserts guard zones around each block allocated using malloc(). Guard zones are colored red. A read or write to red memory triggers an array bounds violation.



memory returned by malloc()



After strcpy(buf, "RATL")

- A read of yellow memory triggers an uninitialized read violation

# Quantify

- Records dynamic callgraph
  - ▶ Per thread data
  - ▶ Shows top 20 slowest functions by default
- Counts instructions and computes runtime
  - ▶ Repeatable, high detail results
  - ▶ Blind to memory delays
  - ▶ Simulates processor resource use per basic block
- Optional timing instrumentation mode on windows
  - ▶ Limited resolution and repeatability
  - ▶ Function-level detail only, not per source line
  - ▶ Records memory delays
- Coverage is like Quantify but without the timing or callgraph

# How to get the most out of PurifyPlus



- Use it
  - Use it regularly
  - Automate its use
- 
- The online help is really good & concise!



# Use it everywhere

- Think about all your executable components
  - ▶ Multiple executables
  - ▶ Services, daemons



# Use it regularly

- Spectrum
  1. Interactive Purify spot-check by QE before release
    - Requires representative selection of use cases
  2. Occasional Purify bash by a developer
  3. Occasional Quantify bash by a developer
    - Requires known slow use cases
  4. Automated regression tests with Coverage, Purify
    - Requires process to decide what to do with results
  5. Purify & Coverage changed code before check-in
    - You write unit tests, right 😊

Painkiller



Vitamin



# Getting ready

- Use a current version
  - ▶ Check for updates often (see earlier fixpack url)
  - ▶ Read the release notes and online help “common questions”
  - ▶ If you need for support of a new or pre-release OS/compiler, call me
  
- Get enough RAM and swap
  - ▶ Instrumentation 20x exe size
  - ▶ Runtime varies, 2x VM size typical
  - ▶ If it's running very slowly check for paging
    - Process/task manager VM size, paging

# Setting expectations

- It will run slower
  - ▶ OCI instrumentation can take minutes first time
    - It's cached for next time
  - ▶ Starting and stopping an instrumented app has overhead
  - ▶ Runtime slowdown is from 5% (cov) to 5x (bad Purify case)
    - Many tuning options
  - ▶ It's worth the wait
  
- Limited to data from one language at a time
  - ▶ C/C++, Java, .NET
  - ▶ Multiple runs required for mixed language apps

# Setting expectations

- Special attention needed for
  - ▶ Apps using compilers not supported
    - E.g. Smalltalk, BorlandC
  
  - ▶ Analysis of/with other runtime observation tools
    - Use of Windows hooks
    - E.g. Robot
  
- We're here to help
  - ▶ And we're not from the government
  - ▶ [jmsanders@us.ibm.com](mailto:jmsanders@us.ibm.com)

## Tips: Select your instrumentation mode per dll/so

- Trade off performance/footprint for detail
- Purify
  - ▶ Precise - Every memory access
  - ▶ Minimal (Windows) - Only intercept APIs
- Quantify
  - ▶ Line - Every source line, counting cycles
  - ▶ Function - Every function, counting cycles
  - ▶ Timed (Windows) - Functions exported from a library, per-thread timer
- PureCoverage
  - ▶ Line – Every source line and basic block
  - ▶ Function – Every function
  - ▶ Exclude (Windows) – Nothing
- Windows and AIX also support “selective” instrumentation
  - ▶ Only instrument selected exe/dll's, leave others untouched

## Tips: Check instrumentation mode

- On Windows, visit the Run:Settings:PowerCheck:Modules dialog
  - ▶ See what modules are getting instrumented how
  - ▶ On Unix all modules get instrumented the same way
- Build optimized code with debug data (-g / pdb)
  - ▶ Allows PurifyPlus to show you source files and line numbers
  - ▶ If the debugger can't show it, neither can PurifyPlus
- Build Windows exes with relocs - link option “/fixed:no”
  - ▶ So “precise” instrumentation is possible for exes
- If you're trying to analyze Java, don't OCI it by mistake
  - ▶ Clues:
    - You shouldn't see an “instrumenting” dialog
    - You shouldn't see Windows API-looking function names in results
    - You shouldn't see Purify error reports

## Tips: Purify C++

- Leak hunting
  - ▶ If you *just* want leaks, use “minimal” instrumentation (Windows) or -memory-leaks-only option (AIX)
  - ▶ Look at large and repeated leaks first
  - ▶ Use the NewLeaks button/API to narrow time window of lost pointer
  - ▶ Set a watchpoint on a pointer that’s being corrupted
  - ▶ If growth isn’t shown as a leak, use New In Use reports
- Use filters/suppressions
  - ▶ To look at errors before warnings
  - ▶ To hide things you don’t want to solve today
  - ▶ Store them per dll/so & check them in
  - ▶ Don’t forget to turn them off periodically
- Look at Coverage data
  - ▶ Unexecuted code is not Purify’d



## Tips: Purify C++

- Run your Purify'd app under the debugger
  - ▶ Use “Break on error” or breakpoint at `purify_stop_here()`
  - ▶ See values that are about to be corrupted
  - ▶ Get more clues with Purify APIs from the debugger
    - `Purify_what_colors(addr, len)`
    - `Purify_describe(addr)`
    - `Purify_new_leaks()`
  - ▶ Filter all but the one error you're trying to track down
  - ▶ Set Purify watchpoints (Unix)
    - High speed address breakpoints
    - See every time an address is read, written, alloc'd &/or free'd

# Tips: Purify C++

- Trace UMRs back via UMCs
  - ▶ Uninitialized Memory Copy is a legal operation, filtered by default

```
int a[10];  
a[1] = a[0];           // UMC, no foul  
a[2] = a[1] + 1;      // UMR since UMC'd data operated on
```

- ▶ Unix: Use Purify watchpoints to see who wrote uninitialized data somewhere unexpectedly
  - `purify_watch(&a[1], 4, "w")`
  - Generates WPW message on every write to `a[1]`

## Tips: Quantify

- Focus on subtree of interest
- Filter/delete uninteresting system blocking time
- Clear/Snapshot around slow use-case
  - ▶ Eliminates data for use-case setup
- Look for “unexpected” behavior, even if not slow today
  - ▶ Unexpected recursion in callgraph
  - ▶ Sort by number of function calls – unscaleable algorithms
  - ▶ Sort by “F time” – compute intensive functions

## Tips: Coverage

- First analysis to consider putting in automated regression tests
  - ▶ Lowest overhead
  - ▶ Use batch mode
    - E.g. `C:> coverage -savedata=foo.cfy myapp.exe`
  - ▶ Merge data over multiple runs/tests
    - Automerge, active merge, manual/command-line merge
  - ▶ Difference coverage between two builds
  
- Natural QE use case
  - ▶ Extend black box testing to white box
  - ▶ Plan how to act on the results collected

# Help us improve PurifyPlus

- If your instrumented app works incorrectly
- If PurifyPlus fails to perform as advertised
  - ▶ Rational folks: file a bug in ClearQuest RATLC: Product=PurifyPlus
    - Detailed description & steps to reproduce
  - ▶ Contact:
    - Windows – [gridings@us.ibm.com](mailto:gridings@us.ibm.com)
    - Solaris/Linux – [jim.veroulis@us.ibm.com](mailto:jim.veroulis@us.ibm.com)
    - AIX/HPUX/SGI - [krangara@in.ibm.com](mailto:krangara@in.ibm.com)
    - Anything – [jmsanders@us.ibm.com](mailto:jmsanders@us.ibm.com)
  - ▶ We'll reproduce locally or remote debug on your machine

## More information

- The online help is really good & concise
- The Getting Started guides really are informative, easy reads
  - ▶ Installed in the Start menu with Windows
  - ▶ Unix: <http://publibfp.boulder.ibm.com/epubs/pdf/12653120.pdf>
- developerWorks has several how-to articles and a Q&A Forum
  - ▶ <http://www-128.ibm.com/developerworks/rational/products/purifyplus>
- Web-based training course DEV205
  - ▶ <http://www-128.ibm.com/developerworks/rational/library/4181.html>
- 30 minute webcast, much like this QSE session
  - ▶ "Develop Faster, More Reliable C/C++ Code with IBM Rational PurifyPlus"
  - ▶ [http://www-128.ibm.com/developerworks/views/rational/events.jsp?search\\_by=purifyplus](http://www-128.ibm.com/developerworks/views/rational/events.jsp?search_by=purifyplus)
- Formal channel for internal support requests
  - ▶ [http://w3.ibm.com/software/sales/saletool.nsf/salestools/bt-rational\\$rational\\_support](http://w3.ibm.com/software/sales/saletool.nsf/salestools/bt-rational$rational_support)

# Questions?

PurifyPlus – the X-ray for your code



Use it today!

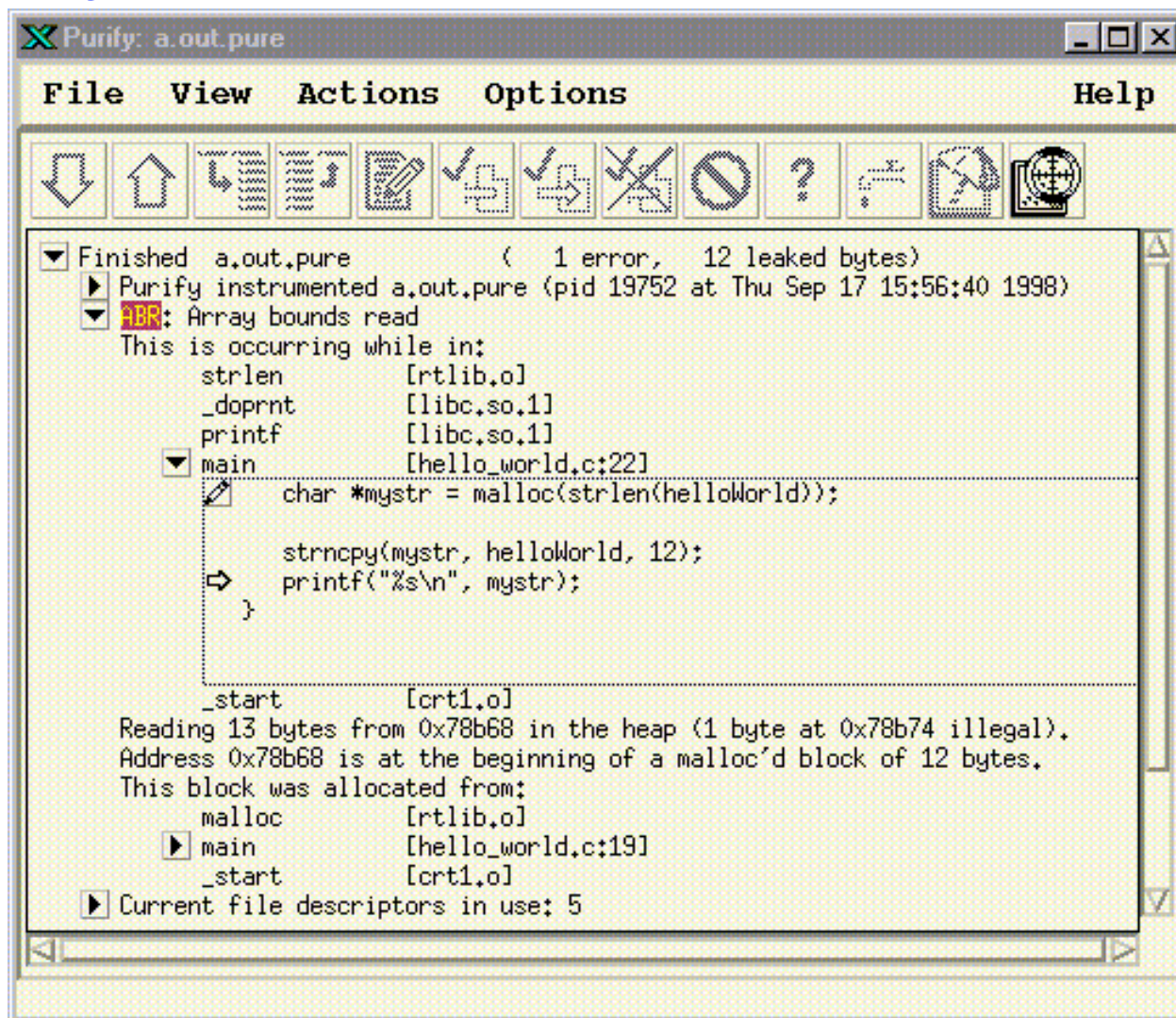
# Backup slides

- Unix interface examples
- More Unix usage tips
- Java & .NET leaks

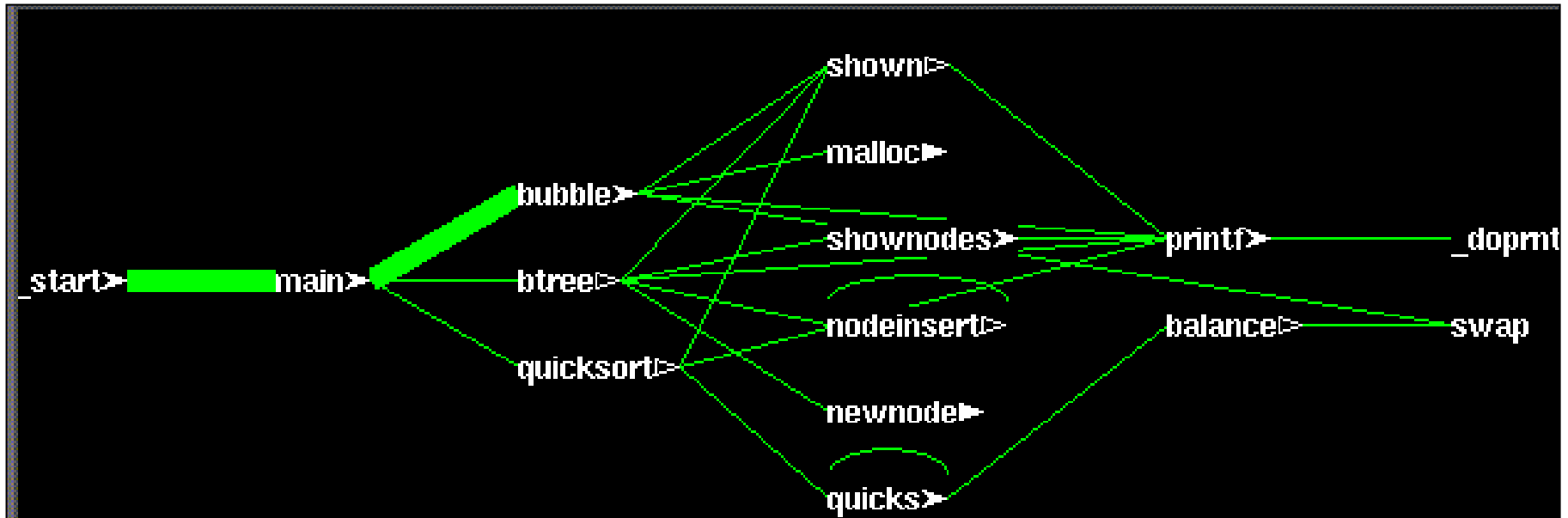




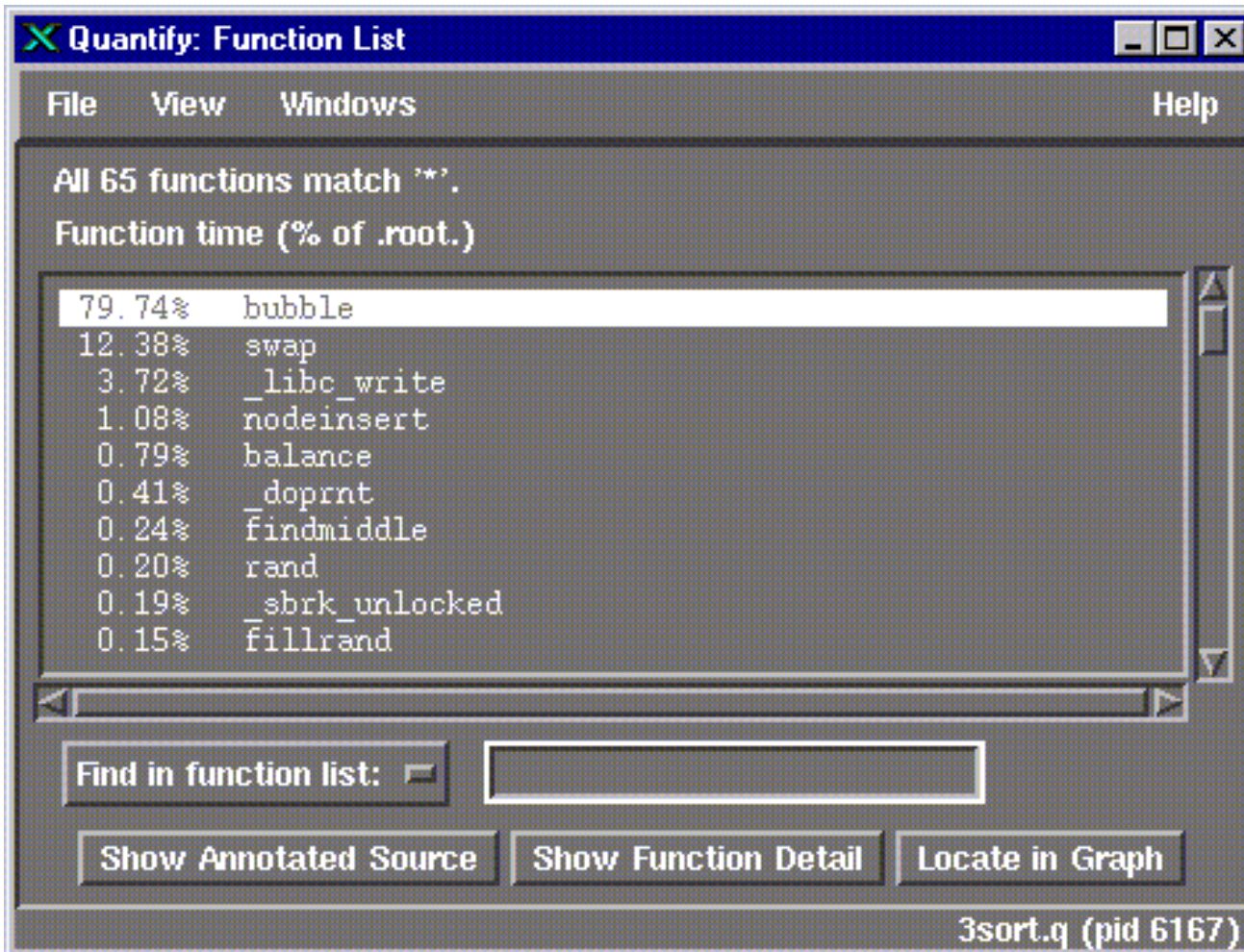
# Unix Purify interface



# Unix Quantify – callgraph, thick lines = more time



# Unix Quantify – top 10 functions by various metrics



The screenshot shows a window titled "Quantify: Function List" with a menu bar (File, View, Windows, Help). The main area displays the text "All 65 functions match '\*'. Function time (% of .root.)" followed by a list of functions and their corresponding percentages. The function "bubble" is highlighted. Below the list is a search field labeled "Find in function list:" and three buttons: "Show Annotated Source", "Show Function Detail", and "Locate in Graph". The status bar at the bottom right indicates "3sort.q (pid 6167)".

Function Time (% of .root.)	Function Name
79.74%	bubble
12.38%	swap
3.72%	_libc_write
1.08%	nodeinsert
0.79%	balance
0.41%	_doprnt
0.24%	findmiddle
0.20%	rand
0.19%	_sbrk_unlocked
0.15%	fillrand



# Unix Quantify – callers and descendants

**Quantify: Function Detail**

File View Windows Help

Function name: bubble  
 Filename: /nfs/u191/home/rajk/Training/3Sort/3sort.c  
 Called: 3 times  
 Function time: 18641355 cycles (79.74% of .root.)  
 Function+descendants time: 21979375 cycles (94.02% of .root.)  
 Average function time: 6213785 cycles  
 Minimum function time: 1929 cycles  
 Maximum function time: 18633439 cycles

**Distribution to callers:**

3 times	main
---------	------

**Contributions from descendants:**

238974 times (13.05%)	swap
6 times (1.47%)	shown
9 times (0.62%)	printf
6 times (0.05%)	malloc

Find:

Show Annotated Source Show Function Detail Locate in Graph

3sort.q (pid 6167)

# Unix PureCoverage – metrics by dir/file/function

**PureCoverage**

File View Actions Adjustments Help

Sorting order:  
Adjusted unused lines

	Runs	Calls	FUNCTIONS			ADJUSTED LINES			ADJS
			unused	used	used%	unused	used	used%	total
▼ Total Coverage			1	5	83%	3	6	66%	0
▼ .../rajc/Training/example/			1	2	66%	3	6	66%	0
▼ hello_world.c	1		1	2	66%	3	6	66%	0
display_message		0	unused			2	0	0%	0
main		1		used		1	4	80%	0
display_hello_world		1		used		0	2	100%	0
▶ .../SUNWspro/SC4.0/lib/			0	3	100%	0	0	--	0



# Unix PureCoverage – coverage per line

PureCoverage: Annotated Source -- hello\_world.c (Adjusted coverage) [Read only]

File View Help

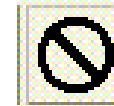
Line	D	I	T	Hits	Annotated Source
14					
15					void display_hello_world();
16					void display_message();
17					
18					main(argc, argv)
19					int argc;
20					char** argv;
21					{
22				1	if (argc == 1)
23				1	display_hello_world();
24					else
25				0	display_message(argv[1]);
26				1	exit(0);
27				1	}
28					
29					void
30					display_hello_world()
31					{
32				1	printf("Hello, World\n");
33				1	}
34					
35					void
36					display_message(s)
37					char *s;
38					{
39				0	printf("%s, World\n", s);
40				0	}

Next unused Prev unused Go to line #: Find:

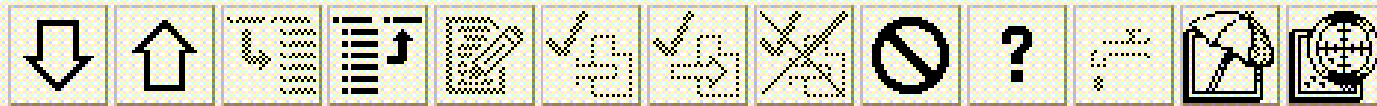
/nfs/ul91/home/rajk/Training/example/hello\_world.c



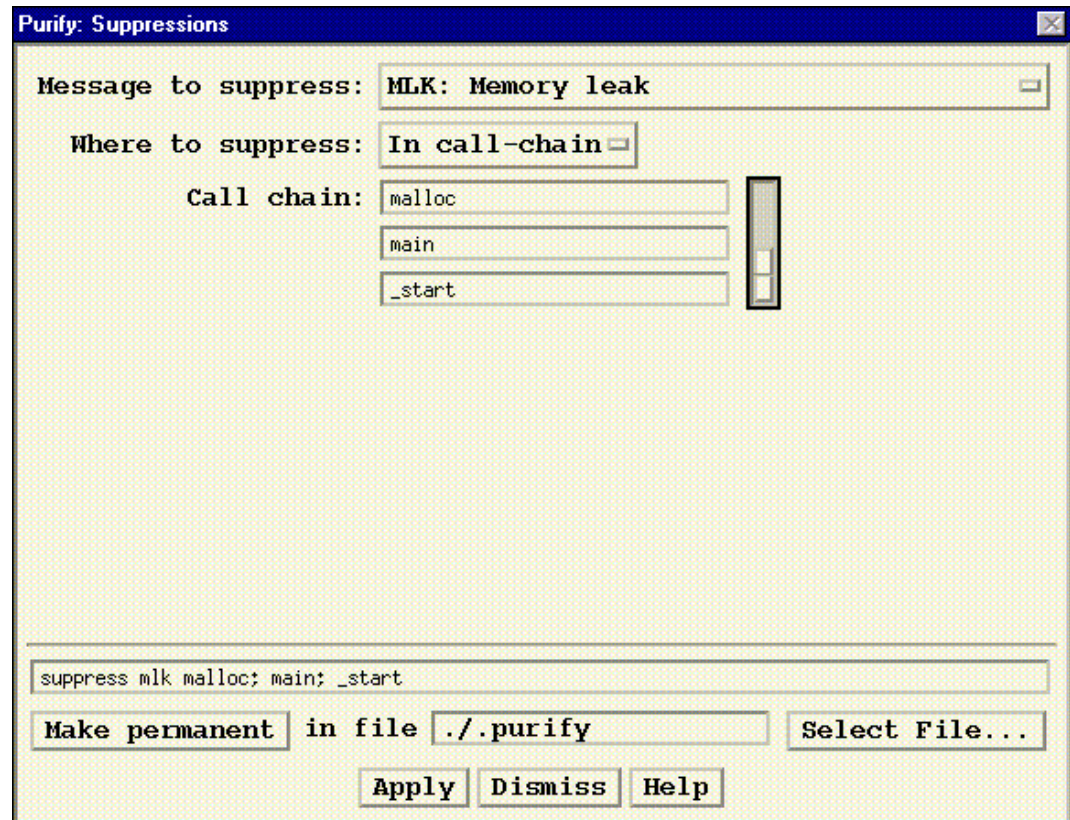
# Unix Purify Suppressions



- To suppress an error, select the error, and click



To make your suppression permanent, click on the "Make permanent" button.



## Unix Tips: Kill Directive

- Sample suppression directives in **.purify** file.
  - ▶ `suppress abr newnode`
  - ▶ `suppress abw newnode; btree`
  - ▶ `suppress fnh newnode; b*`
  - ▶ `suppress umr newnode; ...; btree`
  - ▶ `suppress * *`
  - ▶ `unsuppress * new_*`
  - ▶ `suppress * ...; "libc*"`
  - ▶ `kill umr ...; "libnsl*"` (undocumented)



## Unix Tips: Compiler Options (Solaris)

- g Generate Debug info
- xs Pull Debug info into the library

## Unix Tips: Purify Build-time Options

### Caching options

-cache-dir	Cache directory
-always-use-cache-dir	Place all instrumented files in cache

Example: `purify -cache-dir=/pure/cache -always-use-cache-dir ....`

## Unix Tips: Purify Run-time Options

Show instruction offsets. (Useful if you don't have debug info).

-show-pc                    Absolute.

-show-pc-offset          Relative to the start of the function.

Example:            `setenv PURIFYOPTIONS "-show-pc"`  
                  `<Run executable>`

# Unix Tips: Reporting Options

Specify `-log-file=<log file>` to save output to a file.

Use conversion characters to name the output files. For example, to use the program name and PID in the log file name, use:

```
-log-file=%v_%p.log.
```

(%v is the program name, %p is the process id)



# Unix Tips: Reporting Options

- Specify `-mail-to-user=<user>` to mail purify output on termination (only if there are errors).
- Specify `-run-at-exit="<cmd>"` to run a shell command on termination.



# Unix Tips: Output Options

- To report repeated occurrences of errors only once, use **-messages=first** (default).
- To elect to defer message generation until end of run, use **-messages=batch**.
- To elect to see all messages, use **-messages=all**.



## Unix Tips: Purify API

- The Purify API consists of functions that you can use to help debug and diagnose memory errors.
- Some functions are meant for use from within a debugger:
  - ▶ `purify_stop_here`  
Set a breakpoint here to stop when Purify reports an error.
  - ▶ `purify_describe(addr)`  
Tells how Purify sees the memory: “global data” or “on the stack” or “X bytes from the start of the malloc’ed block at Y.”
  - ▶ `purify_what_colors(addr, count)`  
Dumps the “colors” (red/yellow/green) of a range of memory to the log window.



## Unix Tips: Purify API (contd.)

- Some API functions are meant for use from within your program:
  - ▶ `purify_is_running`  
Return TRUE when the program is Purify'd.
  - ▶ `purify_printf (_with_call_chain)`  
Print a message to the log (with call-stack information).
  - ▶ `purify_new_leaks / purify_new_inuse`  
Report how much more memory is leaked/in use since the last call.
- Linking your program with [\*purify\\_stubs.a\*](#) eliminates the need for conditional compilation.





# Unix Tips: PureCoverage Adjustments

Use Adjustments to mark code as Deadcode, or Inspected, or Tested

Line	D	I	T	Hits	Annotated Source
46					void shown(arr, size)
47					long int *arr;
48					int size;
49					{
50					/* dont want to show ALL the elements of such potentially large arrays,
51					so we set a show-limit, and only show 'lim' many. */
52					int i;
53					int lim;
54					
55				15	switch(size) {
56				5	case SMALL: lim = SMSHOW; break;
57				5	case MEDIUM: lim = MEDSHOW; break;
58				5	case LARGE: lim = LARSHOW; break;
59			T	TEST	case XLARGE: lim = XLARGE; break; /* purecov: tested */
60			T	TEST	default: printf("\nshown: Error: No case for switch size."); /* purecov: tested */
61			T	TEST	lim = SMSHOW; /* purecov: tested */
62				15	break;
63					}
64					
65					/* example of over-coverage (missing break statements) */
66				15	switch(size) {
67				5	case SMALL: printf("\nHere are the first %d numbers of %d.\n", lim, SMALL);
68				10	case MEDIUM: printf("\nHere are the first %d numbers of %d.\n", lim, MEDIUM);
69				15	case LARGE: printf("\nHere are the first %d numbers of %d.\n", lim, LARGE);
70				15	case XLARGE: printf("\nHere are the first %d numbers of %d.\n", lim, XLARGE);

Next unused

Next unused Prev unused Go to line #: Find:

/nfs/u191/home/rajk/Training/3Sort/3sort.c



# Unix Tips: PureCoverage Adjustments

Before Adjustments

After Adjustments

Sorting order: Unsorted	Runs	Calls	FUNCTIONS			ADJUSTED LINES			ADJS total
			unused	used	used%	unused	used	used%	
▼ Total Coverage			2	18	90%	16	137	89%	3
▼ .../raj/k/Training/3Sort/			2	15	88%	16	137	89%	3
▼ 3sort.c	1		2	15	88%	16	137	89%	3
delnodes		0	unused			8	0	0%	0
dellinkList		0	unused			4	0	0%	0
<b>shown</b>		15	used			0	17	100%	3
btree		3	used			1	10	90%	0
findmiddle		1349	used			1	8	88%	0
main		1	used			1	18	94%	0
quicks		1349	used			1	7	87%	0
balance		673	used			0	11	100%	0
bubble		3	used			0	24	100%	0
fillrand		9	used			0	3	100%	0
nalloc		1030	used			0	2	100%	0
newnode		1030	used			0	7	100%	0
nodeinsert		10K+	used			0	8	100%	0
quicksort		3	used			0	6	100%	0
shownodes		3	used			0	4	100%	0
swap		10K+	used			0	4	100%	0
walknodes		31	used			0	8	100%	0



# Unix Tips: Merge Data Over Multiple Runs

- Automatically merges data from one execution to the next.
- Ability to merge different execution data into one file:
  - ▶ **%setenv PURECOVOPTIONS -counts-file=*filename.pcv***
  - ▶ **%purecov -merge=results.pcv a.pcv b.pcv**
- Merge data from different versions of same object file (use with caution):
  - ▶ **%purecov -view -force-merge version1.pcv version2.pcv**



## Unix Tips: Differences Among Multiple Runs

- Ability to report the list of files for which coverage has changed
  - ▶ **%pc\_diff old.pcv new.pcv**
- Ability to compare coverage data from two builds of an application
  - ▶ **%pc\_build\_diff old.pcv new.pcv**



# Unix Tips: Measure Coverage difference

```

xterm
> cat test.c@@/main/LATEST
main()
{
  printf("Coverage diff example\n")
  printf("This file is version 1\n")
  return 0;
}

xterm
> cat test.c

xterm
> ct diff -diff test.c@@/main/LATEST test.c | pc_covdiff -file=test.c -lines=no -width=37
test.pcv
Change                Old Version                New Version                Usage
=====
main()
{
  printf("Coverage diff example\n")
5c5,8  printf("This file is version 1\n" |  printf("This file is version 2\n"  1
>                                     >                                     1
>                                     >                                     1
>                                     >                                     0
  return 0;
}
>                                     }

```

# Unix Tips

- Write scripts to analyze ASCII data.
- Some pre-written scripts available:
  - ▶ **pc\_summary**
  - ▶ **pc\_below -percent=*pct***
  - ▶ **pc\_email -percent=*pct***
  - ▶ **pc\_ssheet**
  - ▶ **pc\_annotate**



## Java and .NET specific capabilities

- Quantify and PureCoverage work the same as for C++
- Purify detects Java/.NET memory leaks



# Leaks in Java and .NET

- Purify can find “leaks” in Java and .NET apps
  - ▶ Garbage collection eliminates classic leaks
  - ▶ “Memory in use” can still grow without bounds
- Identify even small leaks to avoid field failure over time
- Powerful “net allocation callgraph” pinpoints the source of leaks



## Java/.NET are immune to this

```
void runForever () {  
    while (true) {  
        String s = new String("Java is Great");  
        process(s);  
        s = null;           // GC frees the unused memory  
    }  
}
```

--- A.N.Other component -----

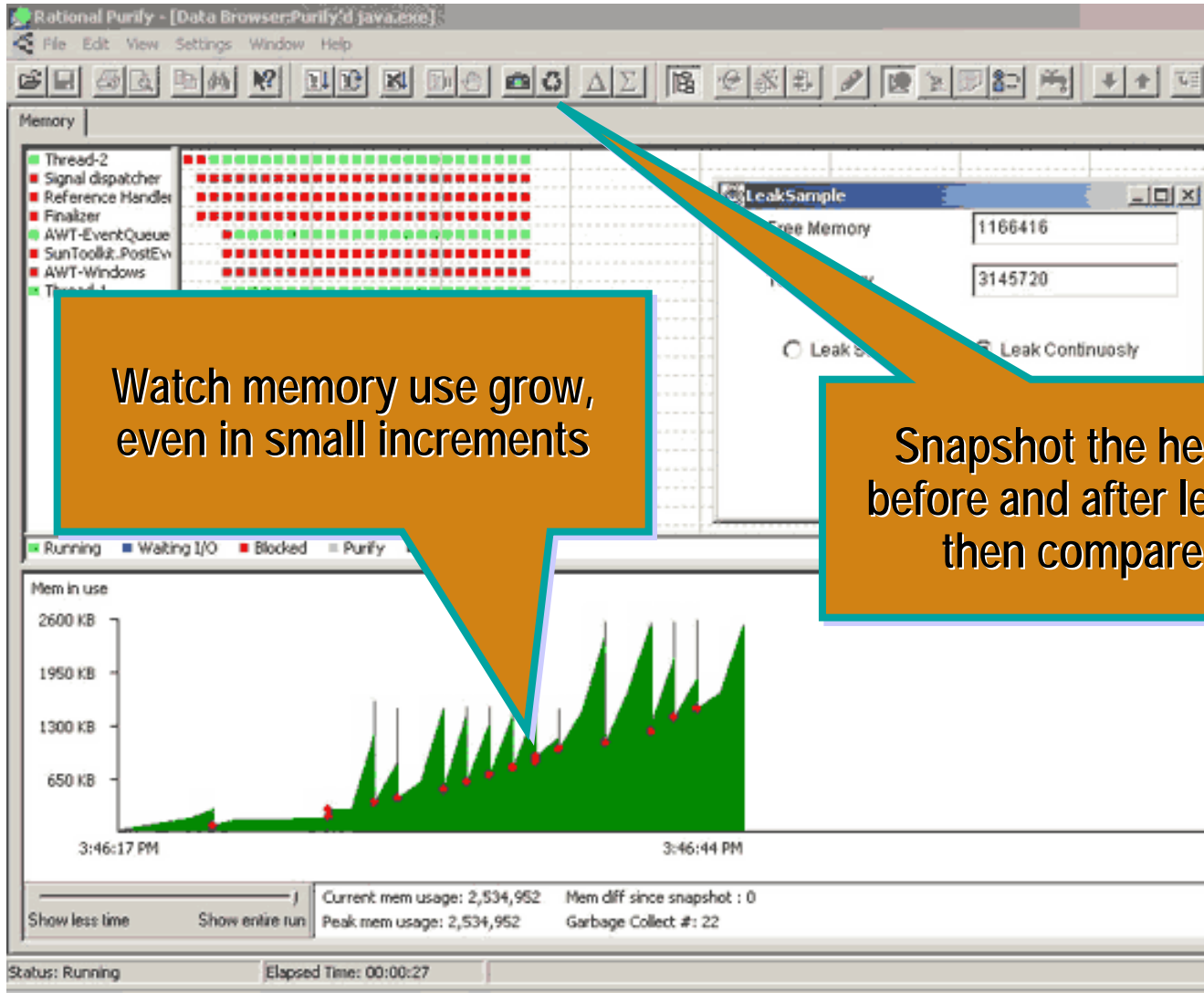
```
void process (String s) {           // do something with the string  
  
}
```



## ..but not this

```
void runForever () {  
    while (true) {  
        String s = new String("Java is Great");  
        process(s);  
        s = null;           // The memory is no longer freed  
    }  
}
```

```
--- A.N.Other component -----  
Vector bottomLess;  
void process (String s) {           // do something with the string  
    bottomLess.add (s);           // and cache it for later reference  
}                                   // expect caller to DoneProcess()  
                                   // to release cache
```



Watch memory use grow, even in small increments

Snapshot the heap before and after leaks then compare



Rational Purify - [Data Browser:java.exe (Diff)]

File Edit View Settings Window Help

java.exe

- Run @ 04/20/2001 10:59:56 -class
- Snapshot @ 04/20/2001 11:00:03
- Snapshot @ 04/20/2001 11:00:09
- Diff @ 04/20/2001 11:00:46 -class

Call Graph | Function List View

Identifies methods leaking the most memory  
Drill down...

Zoom: [Slider] Highlight: Allocation Changes

Visible: 12/12 Highlighted: 5/5 forEver.runForEver() [forEver]

Ready



Rational Purify - [Function Detail: java.exe (Snapshot)]

File Edit View Settings Window Help

java.exe

- Run @ 04/20/2001 11:10:14 -clas
- Snapshot @ 04/20/2001 11:10:33
- Snapshot @ 04/20/2001 11:10:40

**% of Focus**

**Method:** forEver.runForEver

**Calls:** 1

**Current method bytes allocated:** 120,260 (38.35% of Focus)

**Total method bytes allocated:** 718,700

**Number of Objects:** 4,008

**M+D bytes:** 130,504 (41.61% of Focus)

**Avg M bytes:** 718,700

**Min M bytes:** 718,688

**Max M bytes:** 718,688

Object Name	Class Name	Size	GCs Survived	Creation Time	Line Number	References
String 2117FA90	String	20	3	11:10:37 AM	12	1
String 2117FA48	String	20	3	11:10:37 AM	12	1
String 2117FA00	String	20	3	11:10:37 AM	12	1
String 2117F9B8	String	20	3	11:10:37 AM	12	1
String 2117F970	String	20	3	11:10:37 AM	12	1
String 2117F928	String	20	3	11:10:37 AM	12	1

**Callers**

Caller	Calls	Current method bytes allocated
forEver.run	1	130,504

**Descendants**

Descendant	Calls	Current method bytes allocated
forEver.proces..	2,000	10,244

Callers: 1 Descendants: 1 forEver.runForEver()

Detailed memory consumption per method

Sort object list by age and size  
Drill down...



Rational Purify - [Object Detail: java.exe (Snapshot)]

File Edit View Settings Window Help

Zoom: [Slider]

String objects

...anchored by Vector object

...allocated in forEver class constructor

Object Name: Vector 2115BC80  
 Class Name: Vector  
 Method Name: forEver.<init>  
 Size: 20  
 O+R Size: 0  
 GCs Survived: 6

Name	Value
java/lang/Object [] ...	211718D0
int elementCount	2000
int capacityIncrement	0

References: 1    Referees: 1

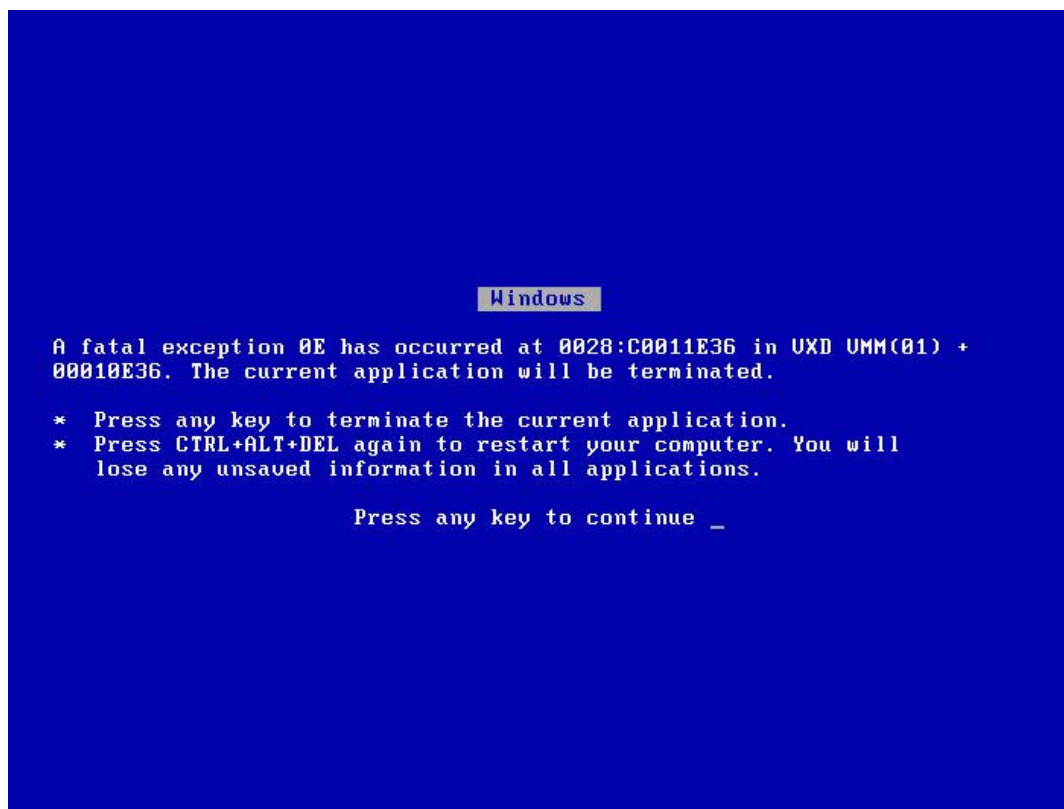


# What is Memory Corruption?

- Points to Remember:
  - ▶ What happens when software misbehaves?
    - Software problems in the Therac-25 radiation therapy machine caused it to deliver fatal doses of radiation to 6 patients.
    - Software (and mechanical) problems with the automated baggage handling system at the Denver International Airport resulted in the airport opening 16 months later than originally planned. (The estimated cost of the delay was \$330,000 per month).
    - According to unnamed White House sources, it was actually a software glitch that caused Dick Cheney's 28-gauge shotgun to shoot millionaire attorney Harry Whittington in the face while quail hunting in Texas.

# What is Memory Corruption?

- Memory corruption can cause an application to crash or misbehave.





# What is Memory Corruption?

- Memory corruption can cause an application to crash or misbehave.



# What is Memory Corruption?

- Memory corruption can cause an application to crash or misbehave.



# What is Memory Corruption?

- Memory corruption can cause an application to crash or misbehave.



# What is Memory Corruption?

- Memory corruption can cause an application to crash or misbehave.

