

Releasing Better Software Faster with Rational Purify and Rational Quantify

Rational Software White Paper



Rational[®]
the e-development company™

Table of Contents

Abstract.....	1
An overview of Run-time Errors and Performance Issues.....	1
Memory Leaks and Access Errors in C and C++ Code	1
Java™ Garbage Collection Issues	2
Performance Bottlenecks in Java and C/C++ Code	2
Rational Purify and Rational Quantify Solve These Problems Efficiently	2
A Detail Look at Runtime Errors and Performance Issues	3
Finding Memory Access Errors	3
Finding Memory Leaks.....	4
Resolving Java Garbage Collection Issues.....	7
Object References that are No Longer Needed	7
System resources that are not freed or cleaned up	8
Other Memory issues in Java	8
Finding and Eliminating Performance Bottlenecks	9
Summary.....	11

Abstract

This paper describes how Rational Purify and Rational Quantify can help developers on Windows AND UNIX (Java, C/C++ and Visual become more productive immediately. Together, Rational Purify and Rational Quantify help developers make their applications more reliable and perform better.

Rational Purify® is an automated debugging and software testing tool that detects memory leaks, access errors and system API errors. Rational Purify inserts additional checking instructions directly into the code produced by existing compilers. These instructions check every memory read and write performed by the program under test and detect several types of access errors, such as reading uninitialized memory or writing to freed memory.

Rational Quantify® is a performance profiling tool that identifies application performance bottlenecks. Rational Quantify collects repeatable, precise performance data, by inserting similar checking instructions directly into the compiled code, and then provides clear, concise views of the profile data. This enables developers to quickly get to the root of the bottleneck, and to accurately monitor performance improvements.

An overview of Run-time Errors and Performance Issues

Memory Leaks and Access Errors in C and C++ Code

Even a single memory access error, such as reading from uninitialized memory or writing to freed memory, can cause a program to act unpredictably or even crash. Yet it can be nearly impossible to eliminate all such errors from a non-trivial program. These errors may produce observable effects only intermittently. As a result, programmers may spend a lot of time looking for these errors, but without effective tools, end users may in fact be the first ones to find them.

When a memory access error does trigger an observable symptom, the error can take days to track down and eliminate. This is due to the frequently delayed and distant connection between the cause, typically a memory corruption, and the symptom, typically a crash upon the eventual reading of invalid data.

In C and C++ the term ‘memory leak’ refers to a section of allocated memory that is no longer accessible to the program because all reference to it has been discarded. Memory leaks slow program execution by increasing paging, and can ultimately cause programs to run out of memory. Memory leaks are more difficult to detect than illegal memory accesses because they are errors of omission rather than commission (that is, they are caused by something that was not done rather than by an actual erroneous line of code). In addition, memory leaks rarely produce directly observable errors, but instead cumulatively degrade overall performance over time.

When memory is freed prematurely, other fatal memory access errors can result. Complicated memory ownership protocols are often required to administer dynamic memory to avoid memory access violations. Incorrectly coded boundary cases can lurk in otherwise stable code for years.

Both memory leaks and access errors are easy to introduce into a program but hard to eliminate. Without facilities for detecting memory access errors, it is risky for programmers to attempt to reclaim leaked memory aggressively because that may introduce freed-memory access errors with unpredictable results.

Conversely, without feedback on memory leaks, programmers may waste memory by minimizing free calls in order to avoid freed-memory access errors. Rational Purify helps developers improve the robustness and

performance of their C and C++ programs by offering them information on both a program's memory access errors and its memory leaks.

Java™ Garbage Collection Issues

The Java language attempts to address the issue of increased memory consumption over time through its automatic Garbage Collection (GC) feature. But, the deeper one looks into the claim that Java solves this problem, the more obvious it becomes that this is simply not true. While there are some problems that it does help to resolve, there are other memory consumption problems that seem more complicated to deal with than in C++.

A memory leak in C++ code is easy to understand. It is memory that has been allocated, but never freed, and is no longer reachable by any code because there is no longer a reference to it. These kinds of memory leaks don't exist in Java because of the Java Garbage Collector, a background task that the Java runtime periodically runs to recycle any unreferenced memory. But the Java Garbage Collector doesn't address logical memory leaks. These are allocated objects that the program no longer needs (and likely hopes the GC will free) but that cannot be freed because other objects unexpectedly still reference them.

Java developers can use Rational Purify to address these issues. Rational Purify reveals how much memory the program is consuming, which methods in the program are consuming the memory, how much new memory the program consumes for a specific set of actions and even where a forced garbage collection might lead to improved performance.

Performance Bottlenecks in Java and C/C++ Code

Whether programming in Java or C/C++, the last issue to be addressed is usually application performance. All too often developers find themselves approaching the end of the development process, with last minute bugs to find and fix, and very little time to put towards performance analysis and improvement. But, identifying performance bottlenecks can be a difficult and time-consuming task, and removing bottlenecks can have architectural implications on the application that would best be addressed earlier in the development cycle.

Before performance improvements can begin to be made, developers must first identify the biggest problem areas. But which causes the bigger bottleneck, a relatively simple algorithm that is called many times or a very complex algorithm that is called only once or twice? And which offers the greater possibility of improvement? Without thorough performance data it is guesswork, because it is impossible to really know which areas of the code offer the biggest return on performance tuning investment. Accurate performance data on all parts of the application – even components for which source code may not be available – will focus efforts on the areas offering the most potential for improvement.

After the areas to be targeted have been identified, it is critical to be able to monitor the performance improvement (or lack thereof) resulting from the changes made. Did a change make the problem a little better or a lot better? Or did it actually degrade the performance overall for some reason? To be able to answer that question precisely depends upon truly repeatable data collection. That is, detailed data must be collected identically each time, so that performance changes can be measured precisely.

It can be very difficult to get accurate, repeatable performance data without a performance profiling tool such as Rational Quantify. Traditional profiling tools have been sampling based, offering only, statistical data with limited detail, even after a special recompile of your code. Rational Quantify overcomes these limitations to take the guesswork out of application performance profiling.

Rational Purify and Rational Quantify Solve These Problems Efficiently

Rational Purify and Rational Quantify use patented Object Code Insertion (OCI) to augment a C/C++ program with checking logic (Rational Purify) and/or counters (Rational Quantify). OCI can be performed

either during the link phase in the case of SunOS, Solaris, and HP/UX or after the link as on IRIX and Windows NT. Rational Purify reads object files generated by existing compilers and linkers, and adds error checking instructions without disturbing the ability to use conventional debuggers on the executable. In the case of Rational Quantify, counters are inserted at the object code level to time every assembly code sequence, providing performance analysis data down to the source line on every portion of an application's code.

For Java and Visual Basic code Rational Purify and Rational Quantify use industry standard virtual machine profiling APIs and augment them with Rational's Byte Code Insertion (BCI) where necessary to gather the same detailed data as with OCI. Again, this is all done after the program is compiled and collects data from all components in an application.

The source for a large program lives in many directories with many different developers. Since Rational Purify and Rational Quantify are used after the program is compiled they are easy to use and allow developers to concentrate on their code, secure in the knowledge that Rational Purify and Rational Quantify will detect errors and bottlenecks which may appear in interactions with other developers' code.

Using Rational's OCI and BCI technology, all of the code, including third-party and vendor libraries, is checked. Even hand-optimized assembly code is checked. This completeness means bugs (such as calling strcpy with too short a destination array) or bottlenecks in application code that manifest themselves in vendor or third-party libraries are detected. Also, serious bottlenecks or bugs in third-party libraries (like writing into freed memory) can be detected. Moreover, the detection or absence of such potentially fatal errors in a particular third-party library during the library's evaluation phase can increase the developer's knowledge of the quality of the application overall.

A Detail Look at Runtime Errors and Performance Issues

Finding Memory Access Errors

Some memory access errors are detectable statically (e.g. assigning a pointer into a `short` variable); others are detectable only at run-time (e.g. writing past the end of a dynamic array); and others are detectable only by a programmer (e.g. storing a person's age in the memory intended to hold their height). Compilers and tools such as `lint` find statically-detectable errors. Rational Purify finds run-time errors.

Errors detectable only at run-time are challenging to eliminate from a program. Consider the following sample Rational Purify session on a simple hash table program. The program is called `testHash`, and has been instrumented with Rational Purify. As the program runs, Rational Purify generates reports for every run-time error it detects. The following output was generated on Solaris, but similar output is generated on all other platforms Rational Purify supports.

```
ABW: Array bounds write:
```

```
  * This is occurring while in:
```

```
    strcpy[rtlib.o]
```

```
    putHash[hash.c:146]
```

```
    testPutHash[testHash.c:84]
```

```
    main[testHash.c:210]
```



Initially all memory is unallocated. Any reads or writes to unallocated memory will cause Rational Purify to generate a diagnostic message. Once memory is allocated, using `malloc/new` for instance, it is not yet initialized. Any reads from this memory will generate a Rational Purify diagnostic message. Writing uninitialized memory causes the memory's state to become initialized and it can then be read as well. Calling `free/delete` causes the affected memory to enter the unallocated state again.

To catch array bounds violations, Rational Purify allocates a small "red-zone" at the beginning and end of each block returned by `malloc` and `new`. The bytes in the red-zone are recorded as unallocated (unwritable and unreadable). If a program accesses these bytes, Rational Purify generates an array bounds error report.

To catch reads of uninitialized automatic variables, Rational Purify sets the state of the stack frame bytes to the allocated but uninitialized state upon every function entry.

To minimize the chance that accesses to freed memory will go undetected because the affected memory is quickly reallocated, Rational Purify does not reallocate memory until it has "aged", and is thus less likely to remain incorrectly pointed into. The aging is user specifiable and measured in the number of calls to `free`.

In order to identify otherwise anonymous heap blocks, the function call chain is recorded each time `malloc/new` is called. The depth of functions recorded is user specifiable.

Finding Memory Leaks

Memory leaks are even harder than memory access errors to detect. The difficulty in detecting access errors is that the direct symptoms of such a bug may appear only sporadically -- but a memory leak typically doesn't even have a direct symptom. The cumulative effects of memory leaks is that data locality is lost which increases the size of the working set and leads to more memory paging. In the worst case, the program can consume the entire virtual memory of the host system.

The indirect symptom of a memory leak is that an application's address space grows during an activity where one would have expected it to remain constant. Thus the typical test methodology for finding memory leaks is to repeat an action, such as opening and closing a document, many times and to conclude that there are no leaks if the address space growth levels out.

However, there are two problems with this methodology. The first problem is that it does not rule out the possibility that there simply was enough unallocated heap memory in the existing address space to accommodate the leaks. In other words the address space does not grow, but a leak does exist. The implied (but risky) assumption is that if the leak was significant enough to care about, it would have consumed all of the unallocated heap memory within the chosen number of repetitions and forced an expansion of the application's address space.

The second problem with this repetition methodology is that it is quite time consuming to build test suites that repetitively exercise every feature, and automatically watch for improper address space growth. In fact, it is generally so time consuming that it is not done at all. Suppose, however, that a developer is sufficiently motivated to build a leak-detecting test suite, and finds that the address space grows unacceptably due to one or more leaks. The developer still must spend a considerable amount of time to track down the problems. Typically, they would either (1) shrink the test suite bit by bit until the address space growth is no longer observed, or (2) modify malloc and free to record their arguments and perform an analysis of what was malloc'd but not freed. The first technique is fairly brute-force, and can take many iterations to track down a single leak.

The second technique seems powerful but in practice has problems. In any given repetition loop, such as opening and closing a document, there may be malloc chunks that are malloc'd but legitimately not freed until the next iteration. Thus a chunk could be malloc'd but not freed during an iteration without actually causing a leak. It may represent a carry-over from a previous iteration. Another method is to record the malloc and free calls for an entire program run, and look for chunks malloc'd but not freed. The problem with this is the existence of permanently-allocated data, such as a symbol table, that is designed to be reclaimed only when the process terminates. Such permanently-allocated data incorrectly show up as leaks, i.e. malloc'd but not freed, with this technique (2) and its variants.

Memory leaks are so hard to detect and track down that they are often simply tolerated. In short-lived programs such as compilers this is not serious, but in long running programs such as web services it is a major problem. As an example, memory leaks proved to be a problem with the X11R4 server program. Instrumenting it with Rational Purify, and running under it under the dbx debugger shows Rational Purify catching the X server leaking one half of a megabyte from a single place!

...the X server runs, then we interrupt the server with control-C, and call the leak finding routine...

```
(dbx) call Rational Purify_new_leaks ( )
Rational Purify: Searching for new memory leaks...
Memory leaked: 516752 bytes (35.9%); potentially leaked: 0 bytes (0%)
MLK: 516312 bytes leaked in 43026 blocks
* This memory was allocated from:
    Xalloc          [utils.o]
    miRegionCreate  [miregion.o]
    miBSExposeCopy  [mibstore.o]
    miHandleExposUres [miexpose.o]
    mfbCopyArea     [mfbbitblt.o]
    miBSCopyArea    [mibstore.o]
    miSpriteCopyArea [misprite.o]
    ProcCopyArea    [dispatch.o]
    Dispatch        [dispatch.o]
    main            [main.o]
    start           [crt0.o]
* Block of 12 bytes (43026 times); last block at 0x35a058
```

Releasing Better Software Faster with Rational Purify and Rational Quantify

```
MLK: 440 bytes leaked in 11 blocks
```

```
* This memory was allocated from:
```

```
  Xalloc          [utils.o]
  miRectAlloc     [miregion.o]
  miRegionOp      [miregion.o]
  miIntersect     [miregion.o]
  miBSExposeCopy  [mibstore.o]
  miHandleExposures [miexpose.o]
  mfbCopyArea     [mfbbitblt.o]
  ProcCopyArea    [dispatch.o]
  Dispatch        [dispatch.o]
  main            [main.o]
  start           [crt0.o]
```

```
* Block of 40 bytes (11 times); last block at 0x36ee98
```

This example shows two leaks that have appeared so far in the current run of the X server. The first is the dominant leak, so let us walk through how to go from this information to finding the bug. This leak has occurred 43,026 times so far, and each time leaked 12 bytes. The first leak was probably not the responsibility of Xalloc, so we look at miRegionCreate. It creates a region structure and simply returns it. So we turn to the caller of miRegionCreate: miBSExposeCopy and find the following:

```
tempRgn = (*pGC->pScreen->RegionCreate)(NULL, 1);
```

A scan of the function confirms that tempRgn is never freed. A one-line fix suffices to eliminate this bug.

Rational Purify employs advanced detection techniques to identify memory leaks in a program, using a mark and sweep technique. In the mark phase, Rational Purify recursively follows potential pointers from the data and stack segments into the heap and marks all blocks referenced. In the sweep phase, Rational Purify steps through the heap, and reports allocated blocks that are no longer referenced by the program. Identifying leaked blocks only by address would not help programmers track down the source of the leak; it would only confirm that leaks existed. Therefore, Rational Purify modifies malloc to label each allocated block with the return addresses of the functions then on the call stack. These addresses, when translated into function names and line numbers via the symbol table, identify the code path that allocated the leaked memory, and often make it fairly easy for the programmer to eliminate the error.

Resolving Java Garbage Collection Issues

As objects are created, Java stores them in the heap, creating a directed graph of these objects, where a node is an object and an edge is a reference. Java treats the directed graphs that start at the roots as a map of currently used memory. Then, any memory in the heap that isn't in one of these graphs can be collected and thus is available for re-use. If the Garbage Collector supervises and recycles the memory, how do leaks occur in Java? Java memory leaks can be divided into two distinct categories: object references that are no longer needed or unknown, and system resources that are not freed or cleaned up.

Object References that are No Longer Needed

An object reference that is no longer needed basically refers to memory that is being referenced, but is no longer logically needed by the program, so it will not be recycled by the garbage collector, when it really should/could be recycled. Since an object typically holds references to other objects, one object frequently holds an entire tree of objects in memory.

Here's a code snippet that demonstrates how an object is made available for garbage collection:

```
public void ShowBoxster()
{
    CarObject myCar = new CarObject("Porsche Boxster");
    double cost = myCar.GetCost();

    // Make it available to be garbage collected
    myCar = null;
    .
    .    // Some code to display the cost to the user
    .
}

// In the CarObject class
public double GetCost()
{
    CostObject myCost = new CostObject();
    double carCost = myCost.lookupCost();
    return carCost;
}
```

In this example, myCost is created in GetCost, and no longer referenced when the routine leaves, thus becoming available to be garbage collected. myCar is also available to be garbage collected after it's set to null;

If we were to now to add the following code:

```
public void ShowBoxster()
{
    CarObject myCar = new CarObject("Porsche Boxster");
    double cost = myCar.GetCost();

    InventoryList.GetInventoryList.AddToList(myCar);

    // Make it available to be garbage collected
    myCar = null;
    .
    .    // Some code to display the cost to the user
    .
}
```

```
}  
  
// In the InventoryList class  
public static InventoryList GetInventoryList()  
{  
    if( inventoryList == NULL )  
        inventoryList = new InventoryList;  
    return inventoryList;  
}  
  
// Adds a car to the  
public void AddTolist( CarObject car )  
{  
    GetInventoryList().carList.addElement( car );  
}
```

myCar is no longer available to be garbage collected in ShowBoxster, because we have created a reference to it in the InventoryList object. In this case we may think that we've taken care of myCar, but we haven't, because InventoryList references it, and InventoryList itself is still in memory (being a static object). There are some distinct code patterns that can cause these unknown or unwanted object references. They are:

- Adding objects to collections or arrays (as illustrated in our example), and forgetting about them.
- A reference that is always reset on next use, for example menu associations. The object isn't being used, but it can't be garbage collected until the next time this routine is called. If the routine is never called again, the object's memory is effectively kept out of circulation.
- An object that changes state, but some references still refer to the old state. An example would be a program that keeps a list of properties of a file, say size, type, and number of characters. It is presented a .txt file and sets all 3 properties. It then is presented a binary file and sets size and type but unintentionally keeps a reference to the .txt file for the number of characters.
- A reference that is pinned by a long running thread. Even if you set the object reference to NULL, it won't be garbage collected until the thread terminates, or goes idle.

System resources that are not freed or cleaned up

These memory leaks are caused because methods have the means to allocate heap memory from the JVM that exist outside of Java instances (resources for windows, bitmaps, etc). They can typically be allocated by calling C or C++ routines via JNI (Java Native Interface) calls. As an example, the Abstract Windowing Toolkit (AWT) for Sun Java requires a dispose() call to free the system resources used by the Frame, Dialog and Graphics classes, much like the malloc/free in C++.

Other Memory issues in Java

- Class Loader - Java loads classes and instances. The class is loaded by the JVM via a tool called the class loader. There can be one class loaded, and several instances of the class can be allocated. In our example, CarObject is the class, and myCar is an instance. There is only 1 CarObject in memory, but there can be mycar, yourCar, etc, instances. Unless you provide your own class loader, a class remains in memory for the duration of the program. This is not necessarily a leak, but is a source of memory usage that doesn't go away.
- Other JVM data areas - These include things like the [Method Area](#), the [Runtime Constant Pool](#), and [Java Virtual Runtime Stacks](#), which are possibly (but not required to be) in the heap.

- It's important to keep in mind that the leaks may come from the JDK, or some 3rd party library. These leaks are not common but have been found in conjunction with calls to various JAVA and other third party libraries.
- In a multi-language environment, leaks could be introduced from C/C++ code called from JNI routines, and thus wouldn't show up in the JVM.

In all of these cases except C/C++, where Rational Purify finds traditional C/C++ errors) Rational Purify tracks where every object is allocated and deallocated, then compares this data for two points in time, after some program activity has occurred. Methods that have more allocated objects attributed to them in the later snapshot are then identified as creating these logical leaks.

Finding and Eliminating Performance Bottlenecks

What Causes Slow Software?

It would seem that run-time bottlenecks--the cause of slow software--are as numerous as specific implementations. Indeed, the manner in which a particular program computes and requests services determines its performance characteristics. Digging deeper into what actually brings about slow executions, however, we find four major causes that cover the vast majority of performance problems:

- Useless computation,
- Needless re-computation,
- Excessive requests for services
- Waiting for service requests to complete.
- Useless computation

Useless Computation: As programs evolve and algorithms are refined, portions of code that were needed in earlier versions can end up falling by the wayside, without ever being removed. The end result is that many large programs might be performing computations whose results are never used. Bottlenecks are caused by time wasted on this "dead" code.

Another common source of useless computations are those made automatically or "by default", even if they are not required. Applications that needlessly free data structures during the shutdown of a program or open connections to workstations even though there isn't a user for them are examples of this type of bottleneck.

Needless re-computation: Programs sometimes re-compute values rather than storing (caching) them for later use. Bottlenecks are caused by wasting time re-computing the values. An example of a potentially needless re-computation is the determination of the length of a string. If this computation is embedded in a loop, the length of the string can be re-computed hundreds or thousands of times, each time resulting in the exact same value!

There are times, of course, when you must continually re-compute values because the cost (in terms of memory) of caching a value exceeds the cost of re-computation.

Excessive requests for services: Programs can make unnecessary requests for services, slowing down execution time and leading to bottlenecks. Excessive requests for operating system services are particularly expensive because, in addition to the time required to perform the service itself, the operating system must "context switch" simply to make the request and retrieve the result. This context switch operation incurs a larger overhead than a normal function call.

Waiting for service requests to complete: When programs request services, they typically "block." Blocking prohibits your program from continuing until the request is completed. Waiting for service requests to complete is a common bottleneck source. One of the most widely recognized bottlenecks of this type is synchronous communication (round-trip requests) between window-based applications and the display server.

Waiting for service requests to complete can occur in a number of other important areas. One such area is retrieving data from a file on disk. Your program can incur a relatively lengthy stall even when accessing a local disk. If the file is mounted on a remote file system, however, the retrieval time via the file server can dramatically increase and be magnified by network delays.

Another area where delay can occur is when the program requests a page of main memory and that page is not present in the memory cache. The operating system must be called to retrieve the page from the swap device and load it into main memory. This, in turn might cause other pages to be written to the swap device to make room for the new page. Note that unlike the explicit file data retrieval requests mentioned above, these time consuming paging operations are triggered by an implicit request for data from memory. They are often caused by seemingly simple actions, such as de-referencing a pointer.

Bottlenecks occur whenever a computation or request is performed needlessly or inefficiently. These four categories represent most performance problems found in software programs.

Reducing and Eliminating Bottlenecks: Eliminating, or at least reducing, bottlenecks requires that you apply one or more of the following five approaches to the appropriate computations and service requests:

- Don't Do It
- Do Fewer
- Do It Later
- Do It, But Don't Do It Again
- Do It Cheaper

To avoid useless or unnecessary computation, **Don't Do It**. Most unnecessary computation can be avoided by simply restructuring the program. If it isn't necessary--for example, freeing data structures during the shutdown of a program--don't waste the cycles doing it.

Do Fewer calls for services. By making fewer calls you reduce the number of context switches that can dramatically slow down your program. Instead of requesting files that don't exist, double check the status of the requested file first, and then request only those.

By delaying computations or requests until they are needed or solicited, you can **Do It Later**. If your program can wait and make the calls for service during what would otherwise be user idle time, then the performance penalty of the context switch and service delay can be effectively hidden from the user.

Do It, But Don't Do It Again and cache a result, if possible, for later reuse. For example, X-Windows client's widgets often cache data available via a round-trip request to the X server. Changing your requests to use the cached data can save this expensive round trip request time.

You can solve bottlenecks caused by needlessly re-computing a string's length this way as well. Computing the value once, outside of the loop, storing it as a variable, and then using the stored length when needed will dramatically improve the performance of your program.

Finally, **Do It Cheaper**. Use an alternative algorithm that is less costly. For example, using a bubble sort when sorting a set of elements requires time proportional to the square of the number of elements in the set.

On the other hand, using a quicksort requires time proportional to the number of elements multiplied by the log of the number of elements. Consequently, it is cheaper to use a quicksort routine for a large number of elements.

These five approaches can serve as a guide as you attack the bottlenecks Rational Quantify finds in your code.

Summary

Rational Purify provides memory access checking and memory leak detection. It fits cleanly into the development environment by instrumenting the program under test without a recompilation. Most important, Rational Purify yields executables that are fast enough to use during the entire development and test process. Rational Purify's relatively low overhead, ease of setup, and thorough error detection everywhere in an application, permits more robust software to be developed faster, yet it entails no overhead in code delivered to customers. Rational Purify can help bridge the gap between a program plagued by intermittent errors and that same program working robustly and continuously over long periods of time.

Rational Quantify collects repeatable performance data on an application, enabling software developers to quickly identify the source of bottlenecks in whichever component they occur, and to measure the improvement resulting from changes. It does this without forcing developers to change the way they do their work, create special profiling builds, or cause them to take a tremendous 'hit' in run-time performance. Rational Quantify provides clear, concise views and analysis of the performance data and supports solid performance engineering practices by actually verifying performance improvements made in applications. Rational Purify and Rational Quantify help developers to develop faster, more reliable code. For additional information on Rational Purify and Rational Quantify, including a fully functional 15 day evaluation download, visit <http://www.rational.com/products/pqc/index.jsp> Try them both today!

Rational®

the e-development company™

Dual Headquarters:
Rational Software
18880 Homestead Road
Cupertino, CA 95014
Tel: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
Tel: (781) 676-2400

Toll-free: (800) 728-1212
E-mail: info@rational.com
Web: www.rational.com
International Locations: www.rational.com/worldwide

Rational, the Rational logo, Rational the e-development company, Rational Purify and Rational Quantify are trademarks of Rational Software Corporation. JAVA is a registered trademark of Sun Microsystems. References to other companies and their products use trademarks owned by the respective companies and are for reference purposes only.

© Copyright 2000 by Rational Software Corporation.
TP- 605 1/01 Subject to change without notice. All rights reserved