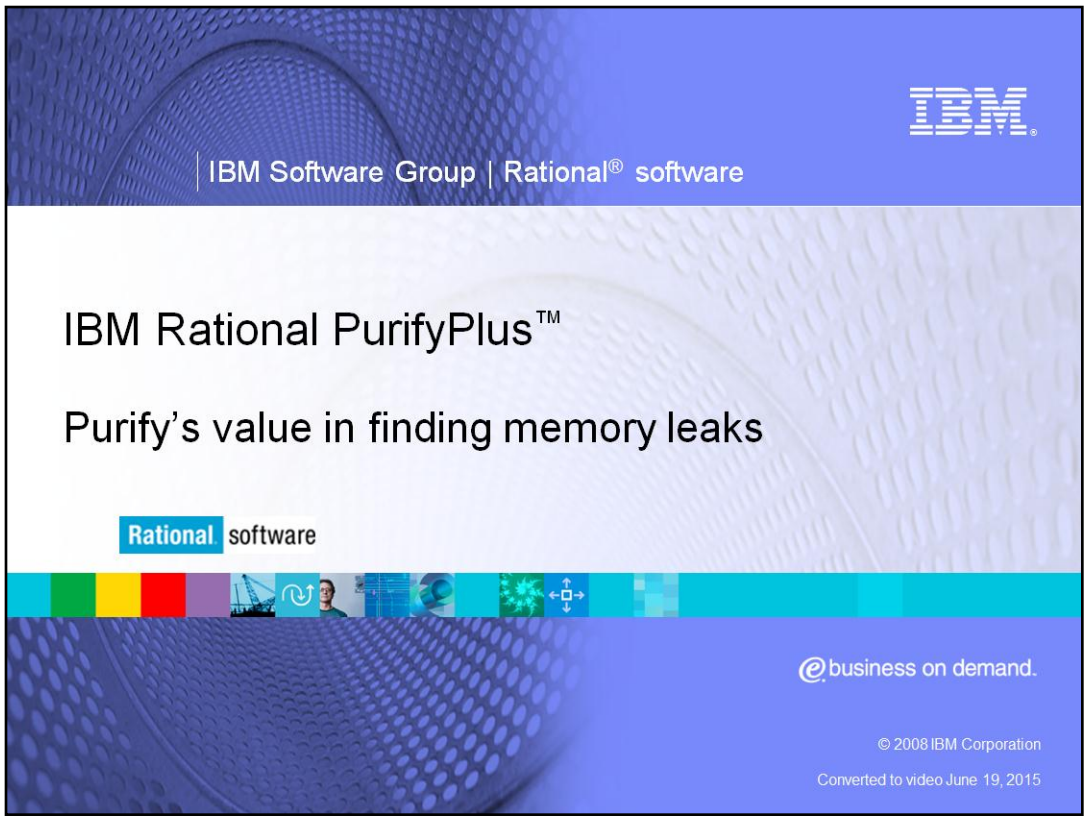
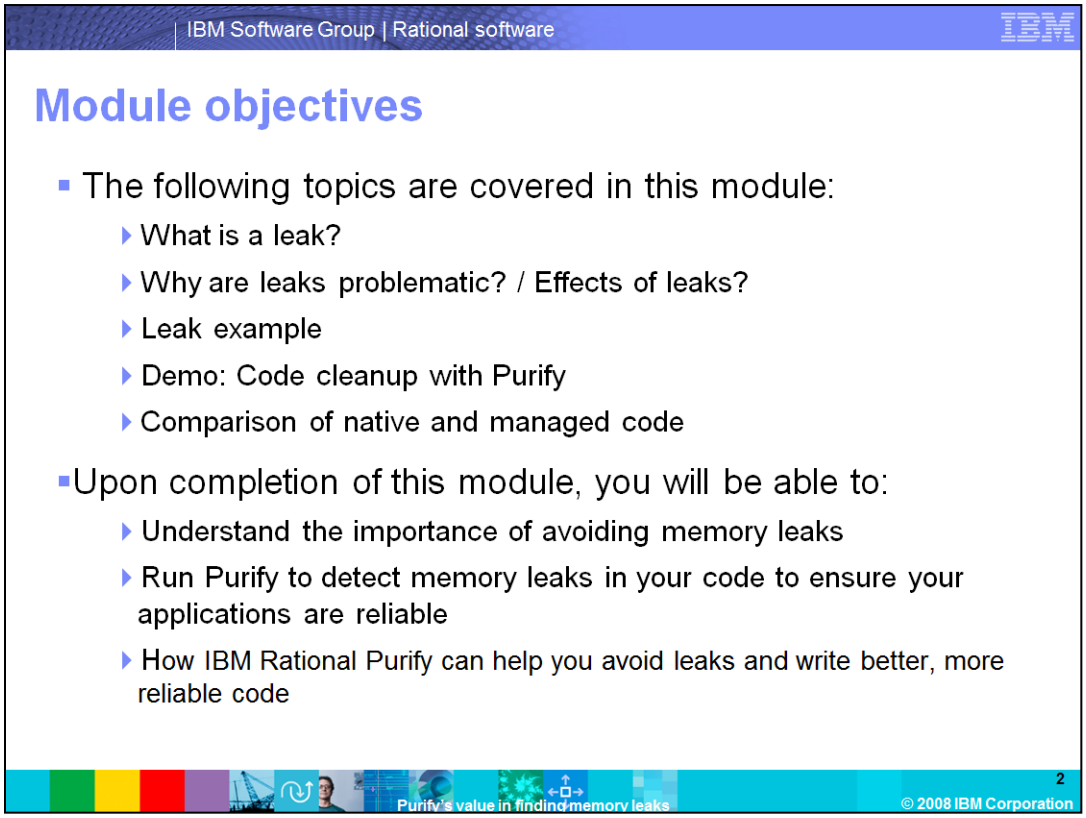


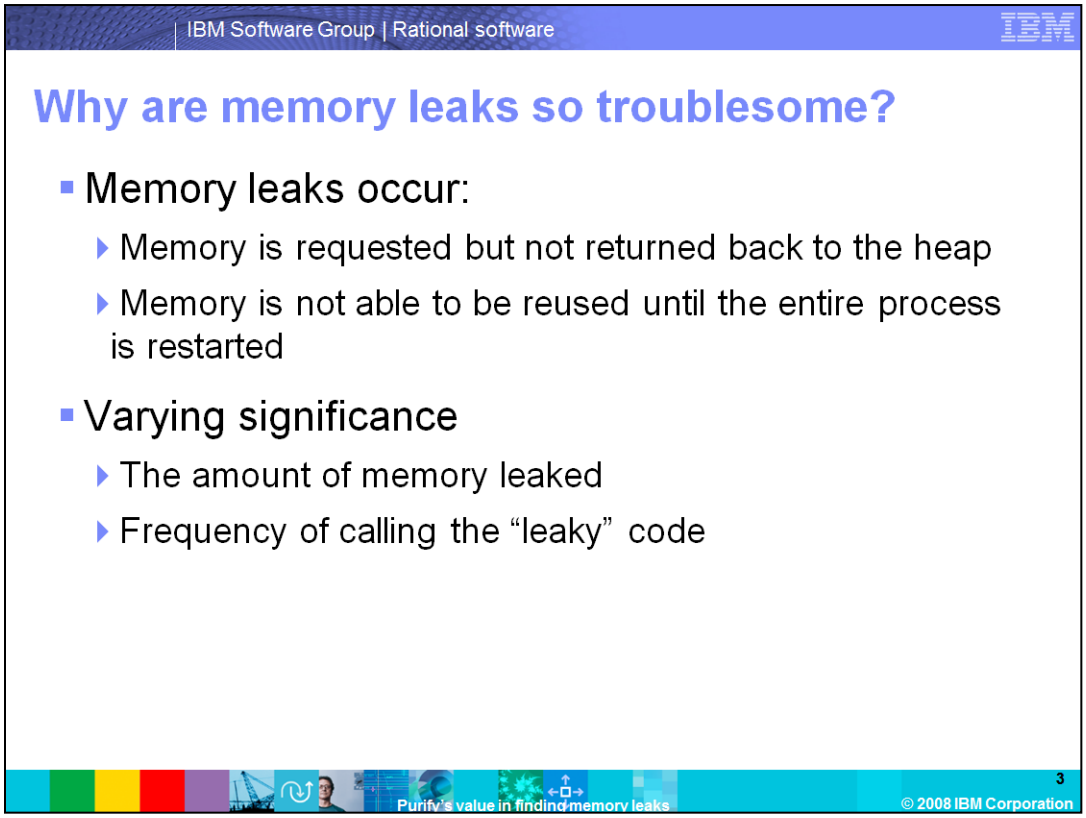
This module will cover the basics of memory leaks in C / C++ code, and will show how IBM Rational Purify® can provide value in eliminating leaks.



This module will cover what memory leaks are, and why they can be so troublesome. This module will also provide a leak example and a demonstration of code cleanup with Purify. Finally, this module will cover a comparison of native and managed code.

Upon completion of this module, you will be able to: Understand the importance of avoiding memory leaks, and run Purify to detect leaks. You should know how you can use IBM Rational Purify to detect memory leaks in your code to ensure your applications are reliable and how IBM Rational Purify can help you avoid leaks and write better, more reliable code.

You should be familiar with general programming concepts and C or C++ before continuing, because this module is geared toward individuals with some background in software development.



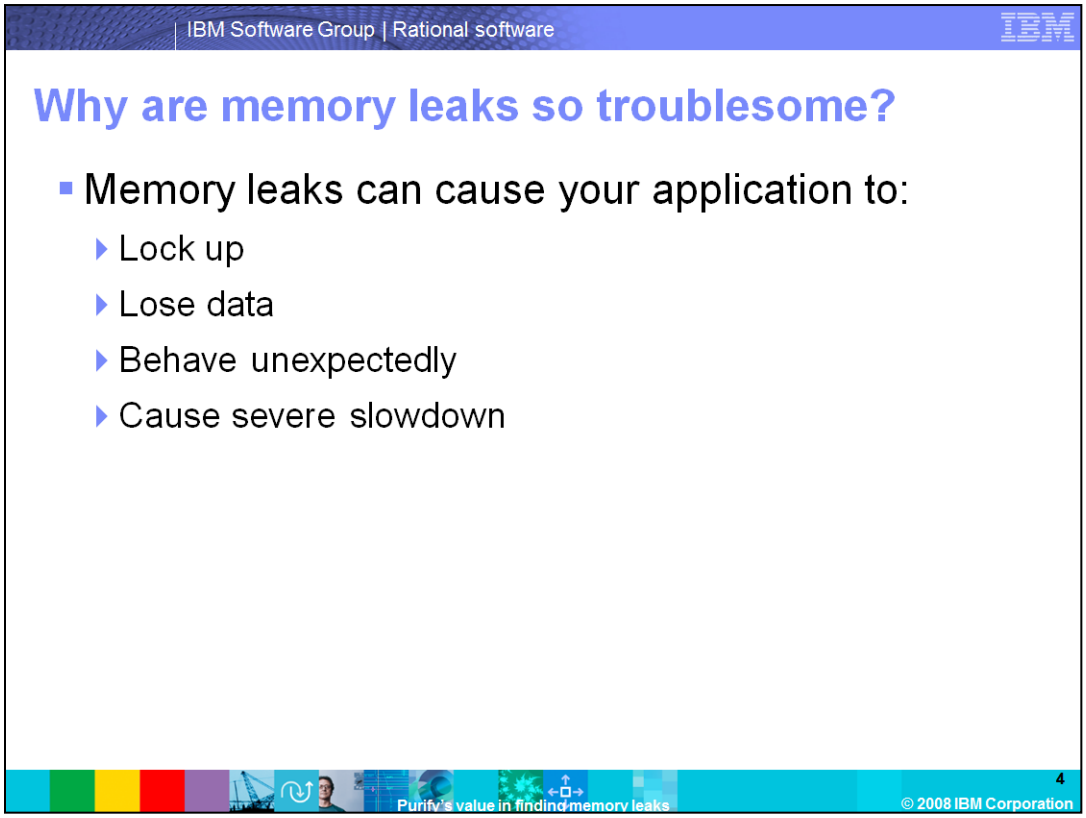
In this section, we'll take a deeper look at memory leaks, starting with why they are so troublesome.

A memory leak occurs when memory is requested at runtime from the heap (typically through malloc or new) but is not returned back to the heap (typically through free or delete) while the associated pointer is still in scope. Once the pointer goes out of scope, this memory is not accessible by your code. Also, the memory is not able to be reused until the entire process is restarted since it is "leaked."

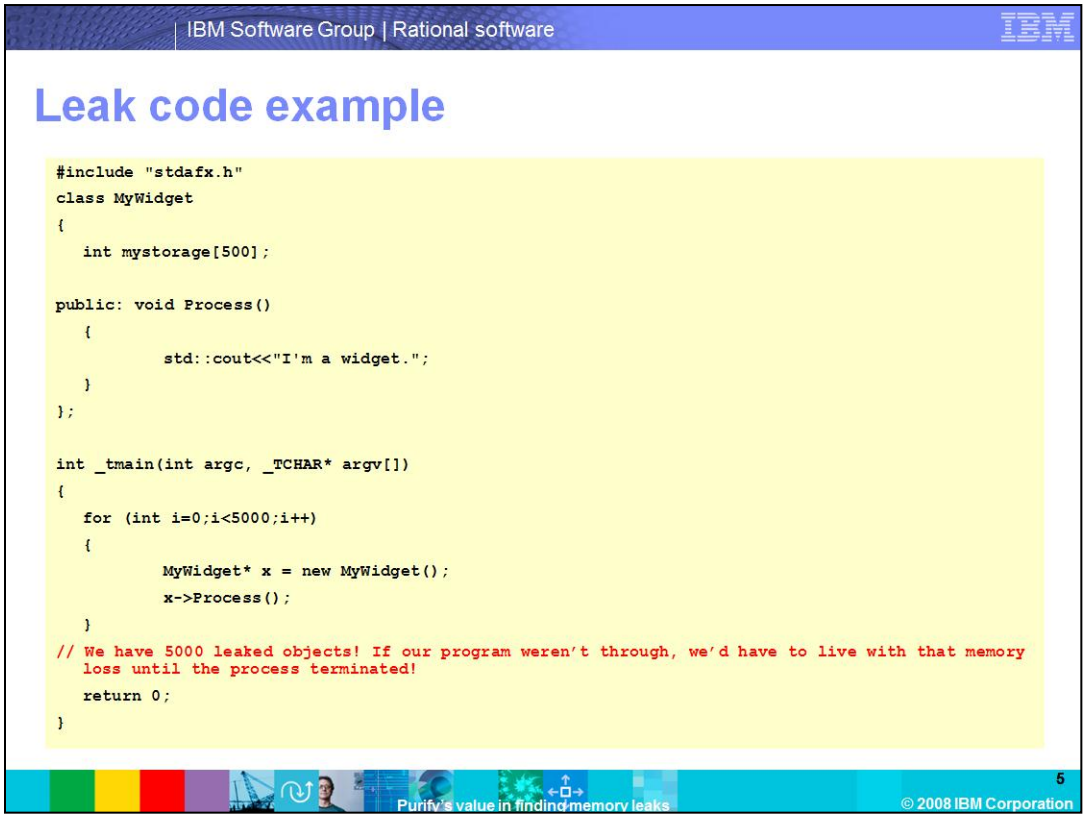
A memory leak can be of varying significance. There are two factors to consider:

1. The amount of memory leaked, and
2. The frequency of calling the "leaky" code

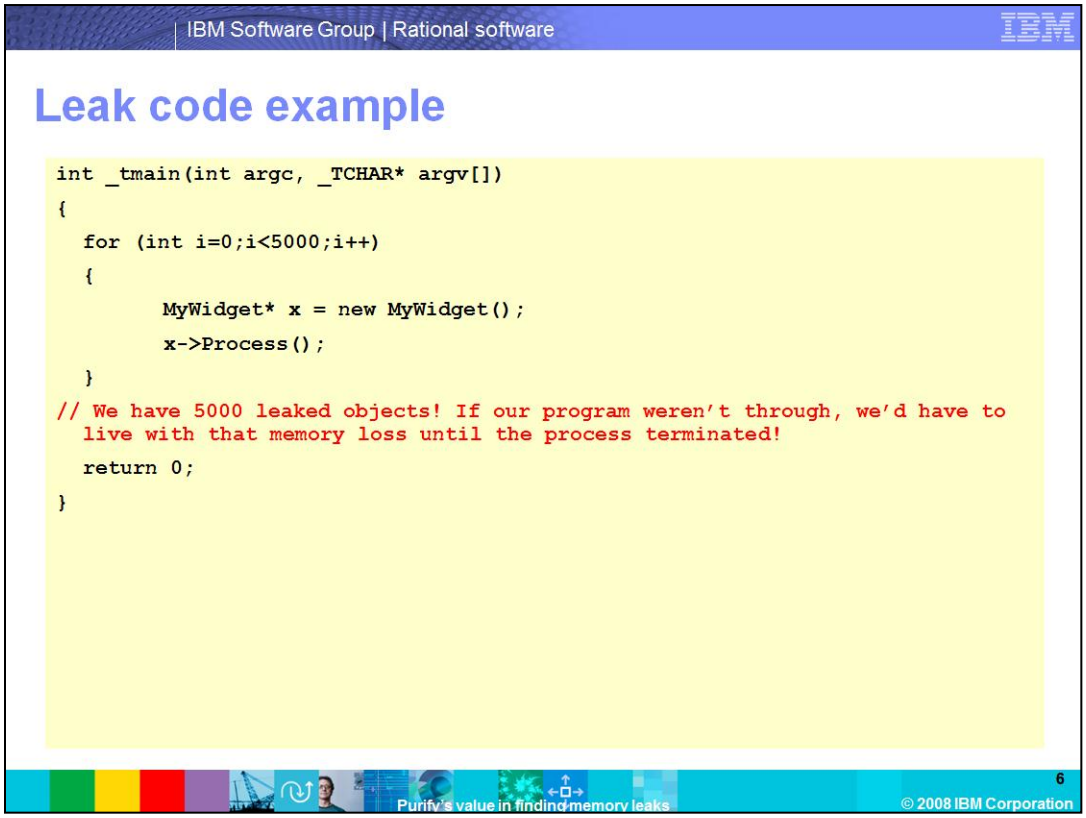
If either of these factors is high, the result can be a significant amount of memory becoming unusable at runtime, resulting in sluggish application performance, or worse, unexpected results. All of these things can tarnish your corporate image in the eyes of the consumers of your software.



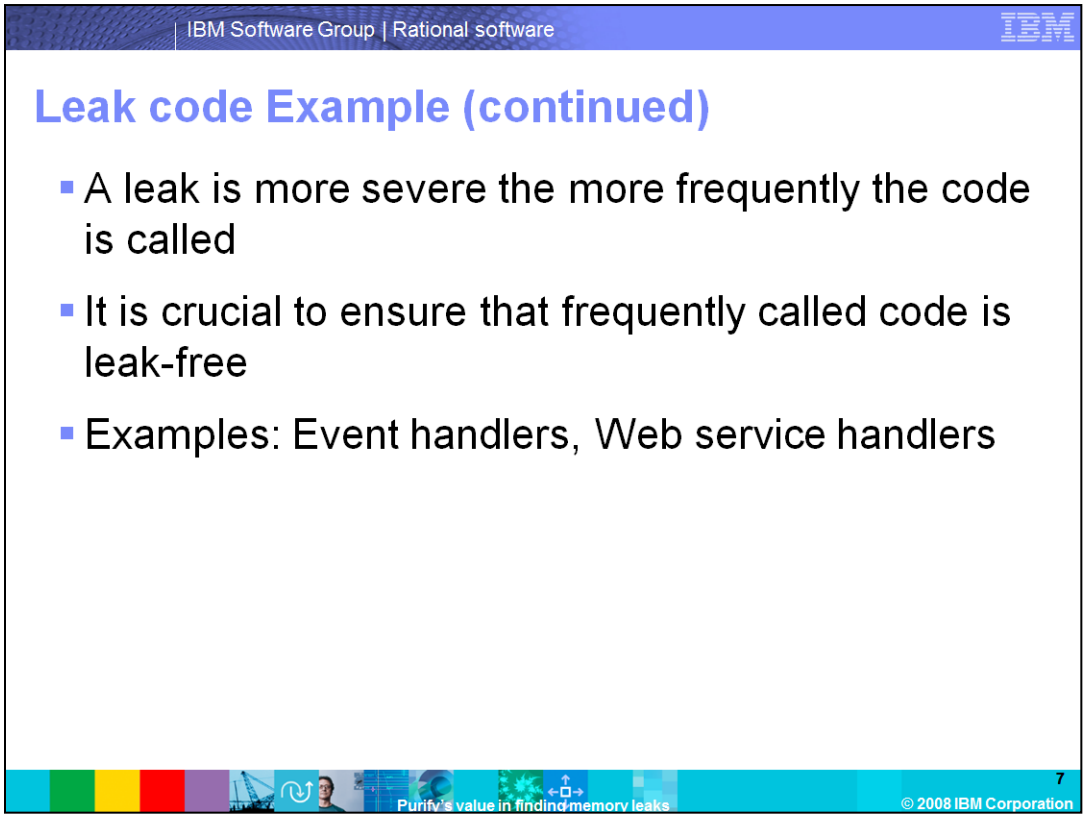
If either of these varying factors is high, the result can be a significant amount of memory becoming unusable at runtime, resulting in sluggish application performance, lock ups, lost data, or worse, unexpected results.



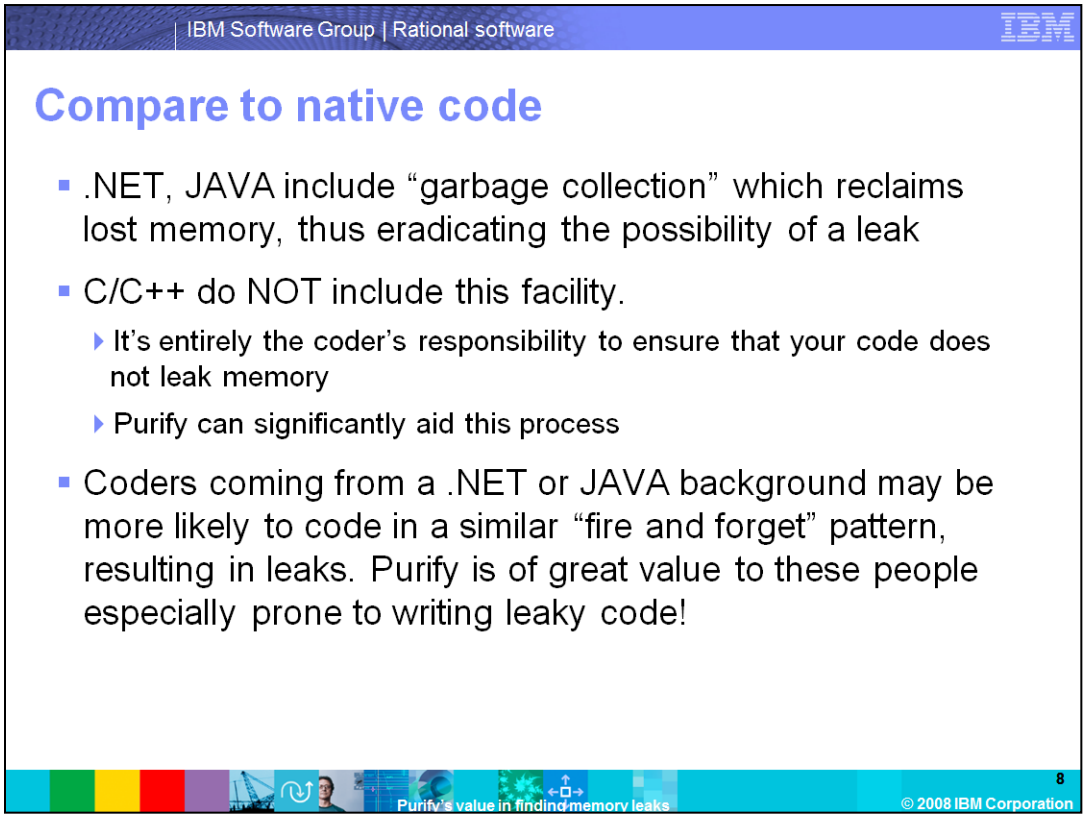
In this leak code example, you have a very simple C++ program. This program will run a loop 5,000 times, each time allocating an instance of the class MyWidget which includes 500 bytes of storage.



The important part of the code is highlighted here. You will notice that the code instantiates the MyWidget instance each time, but never deletes the object, so the memory is lost immediately after the x variable goes out of scope. This is a memory leak!

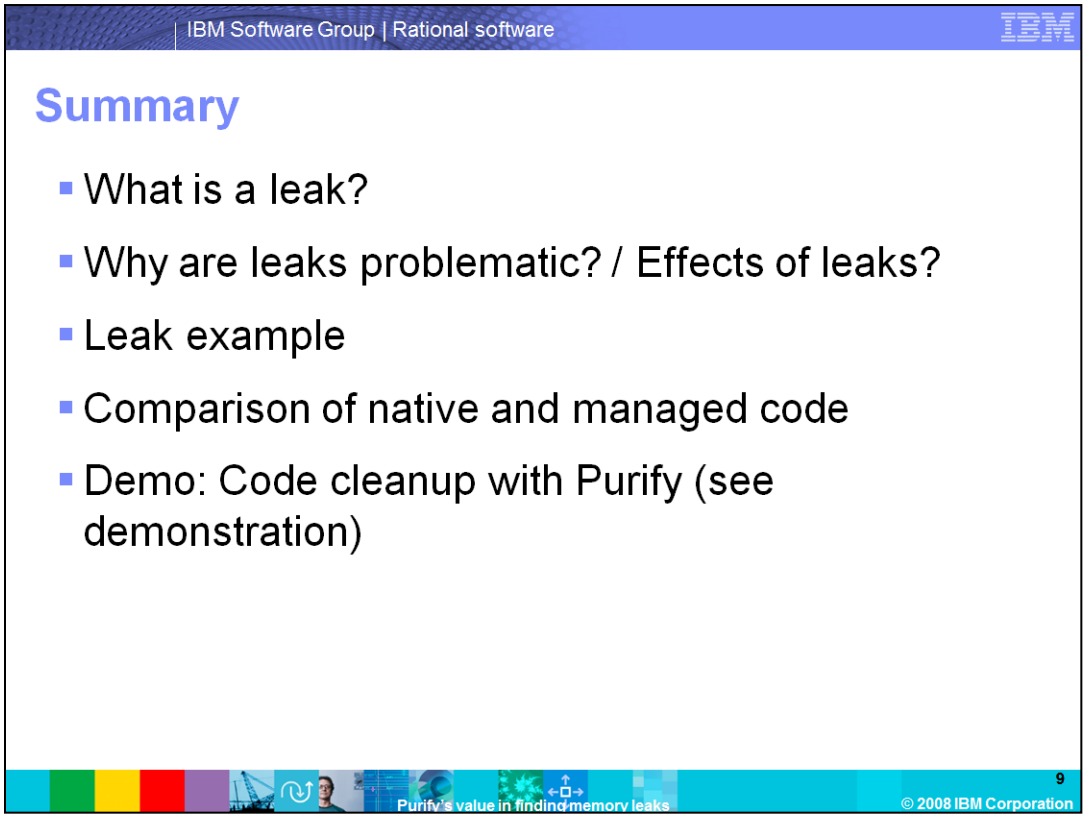


In some circumstances, code is called very frequently. Imagine that you have a routine that is called every time that a key is pressed on the keyboard to perform some validation. This code has the potential to be called many times. If there is a 1k memory leak that occurs in this code, the total bytes leaked can grow very quickly so it's crucial to ensure that frequently called code is leak-free. For example, code that handles Web service or web page requests are generally called frequently, making the need to eradicate leaks very important in these areas.



Programming languages such as Java or C# / VB.NET which are built on the .NET framework incorporate a mechanism called garbage collection. This mechanism essentially eradicates the possibility of a memory leak.

As a result, people familiar with these languages who then write C/C++ code may be more prone to writing code with memory leaks as C/C++ does not include this facility. It is entirely the coder’s responsibility to ensure that your code does not leak memory. As a result, Purify’s ability to find memory leaks becomes even more valuable.



In summary, this module covered what memory leaks are, and why they can be so troublesome. Finally, this module covered a comparison of native and managed code. By now, you should understand the importance of avoiding memory leaks, and how to run Purify to detect leaks. You should know how you can use IBM Rational Purify to detect memory leaks in your code to ensure your applications are reliable and how IBM Rational Purify can help you avoid leaks and write better, more reliable code. You can learn more about code cleanup in Purify with the Code Cleanup demonstration associated with this module.

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM Purify PurifyPlus Rational

A current list of other IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

Rational is a trademark of International Business Machines Corporation and Rational Software Corporation in the United States, Other Countries, or both.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.