Memory Usage Errors: Purify Gathers the Evidence, You Solve the Crime
Allan Pratt, IBM Rational

The PurifyPlus™ product from IBM® Rational® includes Purify®, a memory usage test tool that monitors your C/C++ program while it runs and reports memory usage errors. As a developer, your job begins where the tool leaves off. Purify can report a violation, but only you can find the cause and fix it. In a way, Purify has written the first page of a mystery novel: as the detective, you need to identify what crime has been committed and by whom, decide what to do about it, and try to see that it doesn’t happen again.

There are four major categories of memory usage errors that Purify can detect and report, along with their three-letter acronyms:

- reading from memory you haven’t initialized (UMR)
- using more than you allocated (ABR/ABW)
- using memory after freeing it (FMR/FMW)
- failing to free memory (leaks - MLK)

A fifth type of error is using “wild pointers,” where a pointer value is completely random. Errors like this usually cause your program to crash outright, and tools have a hard time improving on a simple debugger when it comes to finding them.

Let’s explore what you should look for when you get a report about one of these violations.

Reading from uninitialized memory

When you read from memory that you never initialized, you get garbage: unpredictable, inconsistent, and meaningless values. At least that’s what you learn in programming classes.

The truth is, you only sometimes get garbage. Early in your program’s execution, when each allocation is returning “fresh” memory from the operating system, it’s likely that every new memory block you get will just happen to be full of zeroes. This can lead to a false sense of security: your program can seem to work for short runs, and it’s even possible that your whole test suite passes.

When you ship your program to a customer, it crashes and burns. What happened? The customer probably used it for a longer time and for a more complex task than your little test cases. Longer and more varied runs result in your program seeing true garbage – leftover values in blocks that were allocated, used, freed, and then recycled by the memory manager.

Your test suite might pass because it suffers from “small program syndrome”: a bunch of little tests, each one consuming or producing just a little data, each exercising one part of your application. Little tests might not expose the problems that real-world, long-running execution will see, with more recycling of memory and tricky interactions among different parts of your program.

Purify can catch this kind of error, even in small programs. Purify reports an error when you use uninitialized memory, even if the actual values (like the initial zeroes) don’t cause your program to crash or misbehave. Based on these reports, you can find and fix these bugs before they cause a more complex run to crash. Purify can do this because it monitors all your memory access operations and tracks the initialization status of every byte of memory. That way it can issue a report when you read from memory you’ve never written to.

When Purify reports that you’ve used uninitialized memory, the two prime suspects are the point where the memory was allocated and the point where the memory is being used. Until you do some investigation, you don’t know which location hides the bug. Sometimes the allocation site...
should also have initialized the memory, but didn’t. Other times, the problem lies at the access point, where the code should have known better than to read the memory at all. Perhaps there is a missing test for some kind of isValid flag before using the memory. A third category is a missing step in the middle, between allocation and use, that should have put meaningful values into the memory in question. It’s usually harder to see when and why something didn’t happen than why something did. No tool can tell you which of these things is happening: it’s up to you to do the detective work.

An even more interesting case is when the memory appears to be initialized, but it’s really not. For example, you might see “initialization” code in your program that copies data into the allocated block, making you think all is well. But if the source region for the copy is itself uninitialized, then the destination block will contain garbage – actually, a copy of garbage. When you see a block getting initialized by copying, you must look to the source of the copy to see if it contains valid data to begin with. This can happen with memory blocks or single variables: if you see “x=y;” then ask yourself whether “y” was initialized in the first place, or if this statement is just copying garbage.

**Buffer overruns: using more than you allocated**

Buffer overruns are another type of memory error that can hide in a program that appears to work. Let’s say your program allocates 115 bytes for a string buffer but actually uses 120. It’s possible that the memory manager gives you a 128-byte block in this case, rounding up to a 16-byte boundary for efficiency. In that case, the extra 5 bytes you use won’t hurt anybody. Your tests pass and your program seems to work, but only because you’re lucky. Some time after you ship your product, this same section of code is going to ask for 254 bytes and get 256. Then when you overrun by five bytes, you will corrupt a heap data structure or other meaningful data.

When you overrun an allocated block, you risk corrupting memory. Without tools that examine every memory access, this can be a very hard problem to find. The symptoms can be widely separated from the cause. The memory you corrupt could be the heap data structures, which often use pointers and data between allocated blocks. In this case you are likely see a crash in a subsequent allocation or free operation, anywhere in your program, nowhere near the code where the buffer overrun occurred. Alternatively, your buffer overrun could corrupt program data in the next memory block instead of heap linkage data. In that case, the symptoms of the crash will seem to point to the poor innocent part of your program that owns the block that got corrupted. This is a classic case where the symptom is far from the cause.

Any time you overrun a buffer or otherwise use more memory than you allocated, there are again two suspects. Either the allocation site didn’t allocate enough for the task, or the usage site has run off the rails, crashing through the end of the block or reaching beyond the allocated area. You can’t tell which one happened based on Purify’s error report. You just have to investigate the code and see.

One common source of overrun errors is the classic off-by-one error. Did you use strlen() and fail to add one for the null terminator? Did you allocate 10 entries in an array, then forget to start counting from zero? These kinds of things lead to off-by-one errors.

Back in the Bad Old Days of limited memory and slow processors, programmers used to guess at how much space they were going to need. “A path name should fit in 256 bytes,” they’d say, or “A buffer size of 512 should be enough to hold any URL.” But then along comes a deep directory structure or a URL that includes a hashed customer ID string, and all bets are off. It’s bad practice to try to guess in advance what the maximum lengths for strings are. If you do guess, you should always verify that the strings you get will actually fit in your buffers, and truncate them or error out if not.
If you're using C++, the `std::string` class in STL (the Standard Template Library) should be your friend. Today, memory and processor cycles are cheap enough that you don't need to resort to `char*` strings for efficiency very often. The easier memory management for strings that the STL class provides will more than make up for any jokes you suffer about how "real coders program down to the bare metal." Here's a perspective check: real coders want to deliver working, robust programs on schedule.

**Using memory after it's been freed**
One of the most insidious memory errors is using data in a block of memory after it's been freed. This is another one of those bugs that can lurk even when your test suite passes, because your memory manager hasn't gotten around to recycling the freed blocks yet. As long as the freed block hasn't been handed out to satisfy another allocation, the values you last wrote will still be there. You'll never know you're committing a crime by reading them.

Some memory usage test tools help identify this error by "spraying" a pattern into each block as it is freed. That way the data you wrote isn't there any more, and if you access the block after freeing it, the hope is that you will get unexpected data and clearly incorrect results. This technique can report an error sometimes, but often the reported location is far from the scene of the crime. Purify is more advanced than this: by monitoring the status of each byte of memory and checking all your accesses, Purify reports the violation as soon as you touch that memory, whether for reading or writing.

This is another case where you have to do some legwork to identify the proper suspect. The fault could lie at the point where your program accesses the freed memory, in the failure to check an `isValid` flag on some data structure or pointer. Alternatively, the error could be back at the point where the block was freed. Either it was freed in error or prematurely, or your program left out some necessary bookkeeping, like setting that `isValid` flag to false or doing some other cleanup. For this type of error, Purify reports the location of the access violation, the allocation site, and the freeing site to give you as many clues as possible.

**Memory leaks: the crime of omission**
A memory leak occurs when your program should have freed some memory, but didn't. Memory tools for C/C++ generally define a leak as a block of memory that you allocated, but for which you no longer have any valid pointers. Your program can never use or free the block, because you've lost its address.

There is another kind of leak which I call a "moral leak." This is where you still have a pointer to a given block, but nowhere in the whole future logic of the program will you use or free that memory. Tools will show this memory as "in use," and some call it "drag," but morally you've leaked it. (By the way, this is the only kind of leak that's possible in Java or C#. In those languages, the way to free memory is to deliberately lose your last pointer to it. If you accidentally keep even one outstanding reference to a block, the garbage collector won't ever reclaim it.)

Programs that leak memory (either literally or morally) will eventually run out, if they run long enough. Even if your program doesn't run that long, leaks cause your program to use up more memory space than it needs. That can hurt performance: it can cause more paging and swapping in your program and in the rest of the system the program is running on, as other programs compete for physical memory.

When it comes to memory leaks, sometimes the suspect is easy to identify. If each node in a data structure contains a pointer to a string, and the strings leak, you can find the code that frees the nodes and make sure it frees the strings, too. Other times the suspect is lurking deeper underground: often there is no line of code you can point to and say, "Here's where that block was not freed."
For a real leak (not a “moral leak”), the bug officially occurs at the moment when you lose the last pointer to a block. You lose a pointer when:

- you assign some other value to the pointer variable
- the pointer variable goes out of scope
- you free the struct or block that the pointer is stored in

You can also leak memory indirectly. Say you have a struct in memory block X which contains the only pointer to block Y; if you leak block X then you’ve also leaked block Y, because you no longer know its address.

Regardless of the precise moment when the leak occurs, you must examine the whole program to decide on the proper time when you should have freed that block. It might be that you should have released the memory much earlier.

It takes careful analysis to identify the moment when a “moral” leak happens – that is, the last moment that any part of your program could ever touch the memory. Tools can tell you what blocks are in use, but only you can decide which ones are just dead weight and should have been freed, and when. Static analysis tools can sometimes help with this, because they can examine all future possible paths of the program. Purify only sees the parts of the program that actually execute during Purify runs.

To fix both real leaks and moral leaks you have to take a holistic view of the code, understand what the memory is used for, and choose the right strategy for managing its lifetime.

Proving a negative
Part of the value of Purify is that you can use it to prove that no crime is being committed. It may sound strange, but for a software project, the most valuable Purify report is one that contains no errors. Assuming you are running Purify on your program against a reasonable test suite, the “no errors” report means your code is clean and free of the memory access bugs that Purify can find. You can release that program with confidence that it doesn’t abuse memory in these ways. If it did, Purify would have told you so.

Memory usage test tools make you better
Debugging is a journey which starts when you see that a problem exists, and ends when you’ve solved it. Like all good journeys, this is one you can learn from, and at the end you can be a better developer. By seeing the kinds of mistakes you’ve made – or that others have made in code you now own or maintain – you learn to recognize them and avoid them in the future.

When a problem crops up, it’s also good practice to examine other parts of the program to see if it’s happening in other places and fix it there too. Ultimately, thinking about the usage and lifetime of memory in your programs should become second nature: when you write code that allocates a block, your next thoughts should be about who’s going to initialize it, where it will be used, and who’s going to free it and when. That’s the way to write better code and avoid these errors.

Purify may write the first page of the mystery, but only you can write the ending. You have to round up the suspects, analyze the evidence, develop and test your theories, identify the offenders, and put them away for good.

About the Author
Allan Pratt is a Principal Engineer on the PurifyPlus™ product team in IBM’s Rational software division. He has been a professional developer for over twenty years. You can contact him at apratt@us.ibm.com.