

Why do I get a `_BLOCK_TYPE_IS_VALID` debug assertion failure when I try to delete a WIC pixel buffer?

 devblogs.microsoft.com/oldnewthing/20161202-00

December 2, 2016



Raymond Chen

A customer had a problem deleting a WIC pixel buffer. Fortunately, they were able to reduce it to a few lines of code.

```
// Code in italics is wrong.
UINT sourceBufferSize = 0;
BYTE* sourceBuffer = nullptr;

// The next line succeeds.
bitmapLock->GetDataPointer(&sourceBufferSize, &sourceBuffer));

// Omitted code that performs some tasks on the pixels

if (sourceBuffer != nullptr)
{
    delete sourceBuffer;
}
```

“The problem is that when we try to delete the `sourceBuffer`, we get an error from the Visual C++ Debug Library:

Debug Assertion Failed!

Program: Contoso.exe

File: dbgdel.cpp

Line: 52

Expression: `_BLOCK_TYPE_IS_VALID(pHead->nBlockUser)`

For more information on how your program can cause an assertion failure, see the Visual C++ documentation on asserts.

“We’re not sure what the problem is. Does this mean that we don’t need to delete the pixel buffer, and it will be automatically deleted when we release the `IWICBitmapLock`? I tried leaving the buffer alone, and it doesn’t seem to result in a memory leak, but we want to be

sure that's the right thing."

The answer is obvious if you are an operating system person: The operating system doesn't know anything about your language runtime's memory allocation strategy. It doesn't try to guess what version of the C++ runtime you're using, and even if it tried to guess, what would it do when faced with a C++ runtime from the future? As far as the operating system is concerned, the `delete` operator is just some function that is private to your program. It could be called `flubber` for all the operating system knows. How can the operating system know how to allocate memory so it can be `flubber` ed?

Anyway, the deal is that the pointer you get back from the `GetDataPointer` method is a pointer to memory that is not owned by you. The lock gives you access to the memory, but all you can do is access the memory. You cannot free it because it was never yours.

The same logic applies to GDI bitmaps. When you create a GDI bitmap, you get an `HBITMAP`, which represents the bitmap. You can ask GDI to tell you where it put the bitmap pixels by calling `GetObject`: a pointer to the bits will be returned in the `bmBits` member, and you can use that pointer to read or write the pixels of the bitmap. But you can't free the memory. The memory belongs to the `HBITMAP`.

The `IWICBitmap` works in a similar way. When you create a WIC bitmap, it allocates some memory to hold the pixels, and that memory belongs to the bitmap. You can call `IWICBitmap::Lock`, to get an object represented by the `IWICBitmapLock` interface. From the lock object, you can ask for a pointer to the memory that holds the pixels, at which point you can read or write the pixels. That pointer is valid only for the duration of the lock. After you release the lock, the WIC bitmap is permitted to move the memory to somewhere else. (For example, this might happen as part of atlas compaction.)

This is all spelled out and even demonstrated in the sample code that accompanies [the `IWICBitmapLock::GetDataPointer` method](#).

[Raymond Chen](#)

Follow

