

Best Practices: Code Reviews

Last Updated October 22nd, 2007
Copyright © Microsoft Corporation. All Rights Reserved.
Created by Josh Poley

Forward

The first version of this document was sent out in September 2006 to all the testers in the Xbox organization. The intent was to help increase the productivity during code reviews (of test code and product code), as well as highlight some common mistakes that should be watched for. After it was sent out, it quickly spread through the development teams as a good reference.

This document has helped our testers learn some nuances of the C/C++ languages and the Windows APIs that can lead to hard to diagnose bugs. It is much cheaper to find bugs during a code review, so it makes it worthwhile to arm yourself with as many tools and as much knowledge as possible.

Introduction

This document is designed to point out some of the common areas in C and C++ code that frequently have defects which are easy to spot or need special attention during reviews. When you see these patterns in the source, you should give them special consideration and verify that the intention is correctly coded.

This is by no means intended to be an exhaustive list of issues. Use this document as a basic primer or guide to aid you in finding common issues; and to help you think about additional defects. The goal is to improve your odds of finding defects in a formal or informal review.

Setting It Up

When setting up a formal code review, be sure to invite both new team members and more senior developers/testers. This way you can help maximize the learning and bug finding process with a better distribution of experiences.

Hand out hard copies several days before the walkthrough meeting. This gives everybody a chance to go through the code and look for issues before the meeting. The walkthrough meeting should be spent going over problems and answering specific questions about the code, it shouldn't be spent reading. The printouts must contain page numbers, line numbers, and the filename. Otherwise, everyone will get lost as you jump around. Printing out two pages per side works very well in landscape mode. This allows for longer code lines without wasting paper.

When you hand out the printouts, be sure to also send a link to the actual source code. This will allow reviews the ability to search online, run additional tools: The compiler's /W4 switch is the easiest way to catch potential problems, and should be the first thing done. The next step would be to run through Microsoft's PreFast¹, or your favorite static code analyzer (such as Gimpel Software's PC-Lint). These analysis packages can find an amazing number of problems.

Once you've given out the source code, you need to prevent edits to the files. Merging changes in becomes more difficult if there have been large scale changes. And it wastes the reviewers' time as they look for bugs that are either obsolete because of code changes or have already been fixed.

Patterns to Look For

First of all, don't review for code style; there are much better things to spend your time on than arguing over the placement of whitespaces. Looking for bugs and performance issues is the primary goal and is where the majority of your time should be spent (during both the review and the walkthrough). Suggestions on maintenance best practices (covered in the "Maintainability and Robustness" section) can be brought up after the bugs have been discussed.

Memory

Memory Leaks

Memory leaks are the obvious scenario to pay close attention too. Be sure to match up every allocation with a de-allocation. Also be sure to match calls to bracketed new[] with bracketed delete[]. It is also very important to do this at every point in the function where it returns or can possibly generate an exception - memory leaks are very common during error handling.

¹ PreFast is currently available in Visual Studio 2005 Team Suite and Team Edition for Developers, drivers developers also have access to it through the Windows Driver Development Kit:
<http://www.microsoft.com/whdc/devtools/tools/PREfast.msp>.

Additional information on PreFix and PreFast can be found here:
<http://search.live.com/results.aspx?q=prefix+AND+prefast+site%3Amicrosoft.com>.

VirtualFree Leaks

Watch out for VirtualFree calls that do not specify 0 for the size when also passing in MEM_RELEASE. The size must be 0 when using this form, otherwise the memory is not freed and you will end up with a leak, even though it *looks* like the memory is being deallocated.

NULL vs throw

```
char *foo = new char[3];
if(!foo) return;
```

This code looks good, except for when new doesn't return NULL on error, but instead throws a C++ exception. It is recommended to always use the placement new syntax to enforce a behavior:

```
char *foo = new(std::nothrow) char[3];
```

This tells the compiler to ensure the new call will return a NULL and not throw an exception. The other solution is to control which version of libc is used at link time, which will globally define the behavior. But this can be very bad if you are using libraries (such as the STL) which require the throwing version of new.

If this code is in a library, then it must explicitly use the placement syntax to enforce a behavior as there is no guarantee which version of libc will be linked in with the final application.

Is alloca NULL?

```
char *foo = (char*)_alloca(3);
if(!foo) return;
```

As with the "NULL vs throw" pattern, this code is wrong because _alloca does not return NULL on failure. Instead it throws an OS exception (not a C++ exception). If you must handle out of stack conditions, allocations should be wrapped in an appropriate exception handler. Also note that (depending on the platform) _resetstkoflw() needs to be called if an exception is thrown, otherwise your stack's guard page will be left in a bad state. Depending on your application, it may be better to architect it such that you can guarantee there is enough space available.

Integer Overflow to Buffer Overflow

```
if(a+b < 10) return;
char *buffer = new char[a+b];
if(buffer)
{
    strncpy(buffer, input, a);
    buffer[a-1] = '\0';
}
```

Here it looks like the code is doing the right thing. We check to make sure our sizes are within expected limits. Then after the allocation, we copy some data into the buffer. Unfortunately, if 'a' is significantly large, we can overflow the addition so we allocate a buffer less than 10 characters, but then we overflow that buffer during the string copy.

When checking sizes for validity, they should be done separately. Also, using the "safe" version of strncpy

is a better way to do the copy (strncpy_s or StringCbCopyN).

During reviews, every time you see an allocation where the size goes through a series of mathematical operations, be sure to check for possible integer overflows.

Integer Overflow Inside new

Be aware that there is a multiply operation that happens inside the call to the new operator. Make sure to look for code checks which ensures that sizeof(type) * number-of-objects does not overflow. The following code produces an overflow to zero (on a 32bit platform) which is a valid allocation amount, but one that will always lead to a buffer overflow:

```
int size = 1073741824;
int *buffer = new int[size];
```

At this point "buffer" is a valid pointer, but one that points to a zero sized chunk of memory. Any write attempts to buffer[n] will result in memory corruption.

Note that some compilers (such as Visual Studio 2005) will cap the buffer size, preventing an internal overflow of the value.

Array Indexing & Pointer Arithmetic

When working with pointers and arrays, it is almost too easy to walk off the end, or even the beginning, of a buffer. When a pointer is being indexed, look at the ranges that are being used and verify (if possible) against the size of the buffer. You should also be wary of open ended loops:

```
for(;;)
for(i=0; ; i++)
while(1)
while(*ptr)
```

With the last sample, we terminate based on the data itself, and not a buffer size. If the input is untrusted then this can cause a problem if the program is being handed corrupt or malicious data.

For and while loops that *do* have bounds are often sources of "off by one" errors. Check to make sure that the correct usage of "less than/greater than" vs "less than or equal/greater than or equal" is used. Be sure to also watch for integer overflows, as these will lead to reading or writing to the wrong memory address.

IsBadPtr

General consensus is that the IsBad family of functions (IsBadReadPtr, IsBadWritePtr, etc) is broken and shouldn't be used to validate pointers. For one, they are not thread safe; meaning the APIs themselves can cause bugs if the address is in use by another thread at the same time. They only try to find one class of "bad" pointers; these APIs can tell you everything is fine, when in fact it isn't.

It is *not* recommended to use this type of parameter validation as it will often hide bugs. Any code that relies on these APIs for validation probably needs to be

rewritten to be safer in the first place. But if you must probe for access using a `__try / __except` pattern, it is better to do it yourself. This way you can catch guard page exceptions and handle them correctly. The `IsBad` APIs do not do this and can leave your stack's guard page in a bad state.

Initializing sizeof(pointer)

When zeroing out a structure via a pointer, it is all too easy to specify the wrong size.

```
PBITMAPFILEHEADER bmp;
...
ZeroMemory(bmp, sizeof(bmp));
```

By passing `sizeof(bmp)` instead of `sizeof(*bmp)` we only zero out the size of a pointer instead of the struct. This simple bug can be hard to track down as stale data on the stack can look valid. See also: [Initialized by Disjunct Size](#).

Handles

Close

As with memory allocations, handles need to be freed when done. Every time a handle is created, be sure to verify that there is a matching close. Also be sure to check every place that the code returns or can possibly throw an exception.

Handles created by `CreateThread` need to be closed. Though note that they can be closed while the thread is still running.

Related to this, you should also watch for double closes. Closing a handle twice can cause bad side effects if the second close actually freed a valid handle that was created by somebody else.

INVALID_HANDLE_VALUE vs NULL

Unfortunately, some Win32 APIs return `INVALID_HANDLE_VALUE` on error and some return `NULL`. Be sure to read the documentation for each API that returns a handle to see which value is used to represent an error. A common mistake is to check for the wrong identifier when looking for failures.

Here is a list of typical Win32 functions and what they return on error.

<code>CreateEvent</code>	<code>NULL</code>
<code>CreateFile</code>	<code>INVALID_HANDLE_VALUE</code>
<code>CreateMutex</code>	<code>NULL</code>
<code>CreateThread</code>	<code>NULL</code>
<code>FindFirstFile</code>	<code>INVALID_HANDLE_VALUE</code>

Macros

Parenthesizing

When looking at macros, the first thing to check is proper parenthesizing.

```
#define MUL(a, b) a*b
#define ADD(a, b) a+b
```

These are broken because the input parameters must be wrapped in parentheses to preserve proper evaluation. The entire expression should also be wrapped in parentheses to ensure that more complex expressions are evaluated correctly.

Multiple Expansion

A macro that expands and evaluates an input multiple times is a common source of problems. These are good candidates for promotion to functions (inline or otherwise).

Multithreading

Critical Sections

With critical sections, you want to pay attention to the scope and the order in which resources are protected. Scope is important because the resource must be protected for a sufficient amount of time to prevent concurrent accesses, but you also need to keep the time spent in a lock to a minimum. The order in which locks are obtained and released is very important in preventing dead locks. Be sure to review critical sections carefully and look for ways that multiple threads can end up blocking each other.

It is also a good idea to verify that a critical section is needed. Other mechanisms, such as `Mutexes` or the `Interlocked` APIs may be better suited.

Unprotected Resources

Be sure to verify that all accesses to shared resources are synchronized. Memory is the most common, but read and writes to files, the network, etc. may need to be controlled as well.

Freed Resources

A common problem with multi-threading comes from using a resource on one thread that has been released by another. During reviews, look for places where a thread can keep using memory, handles, etc. after they have been released.

Orphaned Threads

Another issue to watch out for is when a program exits (or a DLL gets unloaded) without waiting for all its worker threads to finish first. Be sure to review the thread management scheme and look for places that the main application does not wait on worker threads to finish.

Single-Threaded CRT

If the application is using threads, be sure to check the project settings to ensure it is *not* using the single-threaded CRT library. Using the non-thread-safe CRT

functions in a multi-threaded environment can lead to some very difficult to diagnose bugs. It is much easier to find and fix this through a review.

Note that this is not an issue on Visual C++ 2005 and Windows projects as the single-threaded CRT no longer exists.

Logic

AND OR

When you see an expression using logical or bitwise operators, be sure to investigate the intent. It is common to accidentally use & when && was meant, or to interchange && and ||.

Overloaded AND OR

If a class overloads the && or || operators, be aware that expression short-circuiting rules will be broken. In the below example if 'string' is a character pointer, then strlen will not be called if the pointer is equivalent to NULL.

```
if(string && strlen(string))
```

Now, if 'string' is actually a class that happens to overload the && operator, then the resulting code will actually look like this to the compiler:

```
if(string.operator&&( strlen(string) ))
```

In this case, strlen(string) is *always* evaluated, so the semantics of this code have completely changed.

Precedence

People often mistakenly interpret a given operator as having a lesser or greater importance depending on the context. Unfortunately this can lead to precedence errors when expressions are evaluated. Most of the time it is better to have too many parentheses than too few. For example, these two lines are equivalent.

```
(x * 2) + 1;  
x << 1) + 1;
```

However, these are not:

```
x * 2 + 1;  
x << 1 + 1;
```

This problem is often more frequent in C++ when a class overloads the operators as expectations of operator importance may change. For example, the "streaming" operators used with the IO classes often give users a different impression of the precedence:

```
cout << x&0xFFFF;
```

The above line is incorrect because the shift operator will be evaluated before the bitwise AND operator.

Sequence Points

In the following code, there is no guarantee which function will be called first:

```
DoSomething() + DoSomethingElse()
```

Whereas this statement guarantees that DoSomething will be called first:

```
DoSomething() , DoSomethingElse()
```

This is because the standard does not define, outside of "sequence points", which sub statements/expressions are evaluated first. The following operators and conditions create sequence points²:

- && – left hand side is fully evaluated before the right hand side.
- || – left hand side is fully evaluated before the right hand side.
- , – the comma operator when used in an expression. It is *not* a sequence point when used to separate function parameters, meaning that function inputs are not evaluated in any specific order.
- Function call – all inputs are evaluated before entering the function.
- ? : – the expression before the question mark is fully evaluated before any of the conditionals.
- ; – the statement end is a sequence point.
- if, switch, for, while, and do while – the "conditional" portion is fully evaluated before the body.

What this boils down to: watch out for expressions with side effects (function calls and assignments) that happen between sequence points. If you see these, then the code is buggy and the order of evaluation may change with compiler updates. Below are some common examples of code to watch for:

Function inputs are not evaluated in any specific order, so the first parameter in this sample may be zero, or it may be something else.

```
x = 0;  
f(x++, x++, x++);
```

In the below expression, there is no guarantee that the first value read from the stream will be multiplied by 256.

```
x = ReadFromStream() * 256 + ReadFromStream();
```

Multiple assignments using the same variable in between sequence points are bad, these should always be avoided.

```
x = x++;  
array[x] = ++x;
```

Miscellaneous

Resource Use after Release

Watch out for cases where resources are used after they have been released. This can happen with pretty much any type of resource: memory, handles, critical sections, etc. It is good defensive practice to invalidate the association to the resource after it has been

² Note that operator overloading will convert these operators into the "Function call" semantics. Also note that parentheses do not introduce sequence points.

released. This will help catch these instances without trying to debug something that only fails a percentage of the time (because the data being re-used "looks" correct).

Calling Convention

As new public APIs are added to a library, it is not uncommon to forget to explicitly state which calling convention is used. When reviewing the header file, ensure that all the public functions have an explicit declaration of the calling convention.

Formatting

When an application uses `sprintf` style formatting, be sure to validate the number of arguments and the types. Here are some of the common issues:

64 bit Values

When writing out a 64 bit value, you need to use the proper type modifier:

```
printf("%u", val64bits);
```

Should be:

```
printf("%I64u", val64bits);
```

Untrusted Inputs

If the input buffer comes from an untrusted source, this can result in a security attack as the input can contain formatting specifiers that will pull data off the stack. Additionally it can cause buffer overflows by expanding the resulting string beyond expected limits.

```
printf(buffer);
```

These should always be converted to:

```
printf("%s", buffer);
```

Null Termination

Also be sure to verify the size passed to `snprintf`. This function does *not* terminate the output buffer with a NULL character if the resulting formatted text maxes out the buffer. It should be a habit to always add in a NULL terminator after calling this family of functions.

```
char output[SPECIAL_NUMBER];
snprintf(output, ARRAYSIZE(output), "%s",
    input);
output[ARRAYSIZE(output)-1] = '\0';
```

strncmp

When comparing the beginning of a buffer with a static string, be sure to validate that the passed in size is correct. Also watch for hard coded values or improper uses of `sizeof`:

```
char prefix[] = "HELLO";
strncmp(string, prefix, 5);
strncmp(string, prefix, sizeof(prefix));
```

Using the hard-coded value of 5 is dangerous as the prefix may change later. The `sizeof` statement is incorrect as it will return 6.

Using `strlen` is often the safest approach, although may incur a performance loss.

```
strncmp(string, prefix, strlen(prefix));
```

Safe String Functions

Moving away from the standard CRT string handling functions to the safer `StringCb` functions is a good practice. Though be sure to review the sizes passed to these functions. They are no safer than the original versions if they are lied to.

Switch Fall Through

Review all switch cases for accidental fall through conditions. Every place that is explicitly designed to fall through should be commented as such.

Equals TRUE

When dealing with pseudo Booleans (ones represented as integers under the covers), be aware of code that checks equivalence to `TRUE`.

```
result = Funct();
if(result == TRUE)
```

The problem with this is that any non-zero value is "true" so comparing to a single value is probably not correct. Instead the code should be written as:

```
if(result != FALSE)
if(result)
```

Code that uses the C++ `bool` built in type does not suffer from this as the value can *only* be true or false. When reviewing code, be sure that the source does not try to mix and match between representations (`BOOL` vs `bool`) as they are different types and shouldn't be cross pollinating.

Also be aware that `sizeof(BOOL)` is 4 (on 32bit platforms), whereas `sizeof(bool)` is always 1.

Equals S_OK

```
HRESULT result = Funct();
if(result == S_OK)
```

This code suffers the same problems as outlined in the previous pattern. Successful `HRESULT`s can be represented with more than one numerical value. Use the `SUCCEEDED` or `FAILED` macros to check for result status on returned `HRESULT` types.

Return S_FALSE

If a function returns `S_FALSE`, be sure to trace through the call paths and review how the return values are interpreted. It may not have the same meaning as "the function completed through to success" and so just checking for `SECCEEDED` or `FAILED` may not be the correct thing to do. It may be better to explicitly check for `S_FALSE`.

Also of note is that `S_FALSE` does not equal zero and thus does not equal the definition for `FALSE` (or lower case "false" for that matter).

__assume and __restrict

These keywords allow the compiler to make some very powerful optimizations. Unfortunately, if you lie to the compiler, then it can go and generate some very wrong code. These bugs can be hard to find as a cursory look through the C source code will not indicate a bug: the algorithm can be coded correctly, but the output is incorrect.

When you see the `__restrict` keyword being used on input parameters, make sure to review everywhere the function is called to ensure that any buffers can never overlap. If they *do* overlap, then any operations performed on the memory may be done with stale and incorrect data.

The `__assume` keyword can be just as dangerous. When you see the code author making an assumption about the value of a variable, be sure that you can validate and confirm the theory.

```
__assume(counter <= 10);
```

If 'counter' can ever be greater than ten, the generated code may or may not do the expected thing. Trying to debug when the "wrong" thing happens can be very difficult. It is also good practice to use compile time asserts to help validate the assumptions: everywhere you use an `__assume`, you should use an `assert`.

Signed vs Unsigned

When creating local variables, especially ones used to index through a loop, look for instances where the developer used a signed integer instead of an unsigned type. Most often than not, the value has no reason to go negative.

Re-throwing Objects by Name

If you catch and then re-throw the caught object, it is improper form to use the name of the object in the throw statement:

```
catch(MyClass &b)
{
    throw b;
}
```

Doing this will create a temporary of the caught type. If the thrown object is actually that of a derived class, but caught by the base class' type, then you will lose any information associated with the derived object.

By using the "throw" keyword by itself, you will no longer re-throw a temporary and thus retain the full identity and data of the original object.

```
catch(MyClass &b)
{
    throw;
}
```

Maintainability and Robustness

Here we border on some style issues, but ones that can affect the maintainability of the code as it is passed down the line to different developers. Patterns here don't

necessarily represent immediate issues, but rather *potential* bugs.

/GS

Make sure the project is being compiled with the `/GS` flag. This option will add some basic stack overflow checks to the code. There is no reason *not* to always run with this extra bit of application protection.

Multi-lined Macros

Macros that get so complex that they contain multiple statements should probably be turned into functions. Many developers use complex macros to "simplify" or make their code "easier to read". Unfortunately this will increase the learning curve for others that now have to dig through a bunch of header files in order to understand the program flow. It is often better to have lengthier code where it is easier to follow the actual execution flow than to have macros that make the source look "cleaner". Another drawback with macros is that they make debugging harder. It is much better to have a function that can be traced through versus a bunch of instructions that get inserted and don't allow source code stepping.

Array Size Macros

Many projects have an `ARRAYSIZE` or `NUMELEMENTS` macro defined which is used to generate (at compile time) the number of elements a static array has. The common implementation looks something like this:

```
#define ARRAYSIZE(x) (sizeof(x)/sizeof(x[0]))
```

This macro works great assuming you don't accidentally pass in a pointer. The better solution is to use this code which will generate a compiler error if you don't actually pass in an array.

```
template <typename T, size_t N>
char (& SAFE_ARRAYCNT(T (&)[N]))[N];
#define ARRAYSIZE(x) sizeof(SAFE_ARRAYCNT(x))
```

Delayed GetLastError

If a call to `GetLastError` is not invoked immediately following the API call that failed, the error returned may be overwritten by somebody else.

```
hFile = CreateFile(...);
if(hFile == INVALID_HANDLE_VALUE)
{
    ErrorEvent("Could not open file, err: %u",
        GetLastError());
    return GetLastError();
}
```

This pattern is very common in test code. If any additional calls are added to the 'if' statement or `ErrorEvent()` grows in complexity, there is a greater chance that the second `GetLastError()` will not return the failure from `CreateFile`. It is much better to call `GetLastError` once, immediately following the main API, and cache the value if necessary.

Hiding Errors

Watch out for code that obfuscates errors.

```
hr = OtherFunction();  
if(FAILED(hr))  
    return E_FAIL;
```

On error conditions, functions should provide as much detail about the failure as possible. This pattern is also very common when developers call Win32 APIs, look for code that does not call GetLastError and percolate this value back to the caller.

Also make sure that the error codes returned from a function are correct with respect to what happened. Especially watch out for functions that return a "success" code when they actually failed.

Initialized by Disjunct Size

When initializing a buffer or struct, it is always best to use the name of the variable instead of a size that can later become unrelated to the buffer.

```
#define BUFF_SIZE 1024  
char buffer[BUFF_SIZE];  
...  
memset(buffer, 0, BUFF_SIZE);
```

For example, in this sample, if someone goes and changes the buffer length to `BUFF_SIZE+1` to allow for a terminating NULL character, then the `memset` won't set that last byte. Instead, the code should be:

```
memset(buffer, 0, ARRAYSIZE(buffer));
```

Changing the underlying type during a re-write is less common but does happen. For consistency and stability it is probably better to just use the variable's name.

```
MyStruct item;  
memset(&item, 0, sizeof(MyStruct));
```

Should be:

```
memset(&item, 0, sizeof(item));
```

Just be sure to watch out for instances of the "Initializing `sizeof(pointer)`" bugs.

Where appropriate, it is best to use the compiler to initialize objects and buffers:

```
MyStruct item = {0};
```

Additional References

Maguire, Steve. *Writing Solid Code*. Redmond, Washington: Microsoft Press, 1993.

McConnell, Steve. *Code Complete 2nd Edition*. Redmond, Washington: Microsoft Press, 2004.

Meyers, Scott. *Effective C++ Second Edition*. Boston, Massachusetts: Addison-Wesley, 1998.

Meyers, Scott. *More Effective C++*. Boston, Massachusetts: Addison-Wesley, 1996.

About the Author

Josh Poley has been a tester at Microsoft since 1998. He initially worked on the very first version of the Passport authentication service (currently called Windows Live ID). Then, in the spring of 2000, Josh moved over and joined a small handful of people who were starting to work on a project code-named Xbox. His initial responsibilities covered various pieces of the low level operating system (file systems, peripheral communication, etc.). Shortly after the Xbox game console launched in 2001, Josh took over as lead of the Kernel Test Team and remained in charge of validating the core operating system throughout the development and launch of the Xbox 360. Then, in the spring of 2007, Josh joined the Zune team where he is currently working on new media devices.