Systems Programming under Windows

Dmitri Loguinov (ver 1.7, Spring 2018)

The purpose of this document is to refresh common C/C++ programming techniques, introduce MSDN notation, and set forth basic ideas for effective debugging in Windows.

1. MSDN and Visual Studio

MSDN notation may first appear unusual for those mostly familiar with Unix; however, it is quite simple to learn. This section outlines the basic steps in tackling MSDN and overviews the main elements you should know.

1.1. General Ideas

The first step is to understand how to search MSDN (http://msdn.microsoft.com). To avoid frustration, you should be prepared that certain functions (e.g., `select`) produce many irrelevant results from .NET, C#, SQL, and other areas. One approach is to allow Google to pick the specific page based on the frequency of request from other users and its internal ranking algorithm (e.g., by searching for "MSDN select c++"). Another approach is to type the desired function in Visual Studio (any .cpp or .h file), place the cursor within its name, and press F1 to bring up the relevant documentation. Finally, you can manually scroll though MSDN search results until you see the version with "(Windows)" in the title (e.g., "select function (Windows)").

Reading MSDN and understanding the various caveats will save you plenty of time when an APIs crashes because you are passing NULL pointers to it or incorrectly handling its response. Part of this understanding involves knowing the various datatypes that Microsoft uses. It is recommended that you perform a lookup on every unfamiliar type you encounter. Below are some of the most common datatypes:

```
CHAR = char
DWORD = unsigned int          // "double word"
BYTE = unsigned char
BOOL = int                    // boolean
HANDLE = void*
LONG = int
UINT = unsigned int
UINT64 = unsigned __int64
```

Many pointer types are created from existing scalar types by prepending them with `LP` (long pointer):

```
LPDWORD = DWORD*
LPVOID = void*
LPCSTR = char* (expects NULL termination, C-style string)
```

Note that there is another (rarely used) type called `LPSTR`. This is also a `char*`, but there is no expectation that the byte array be NULL-terminated. For more type definitions, see:

http://msdn.microsoft.com/en-us/library/aa383751(VS.85).aspx

1.2. Example

To make these concepts clearer, consider the following example:

```
BOOL WINAPI ReadFile(
  __in         HANDLE hFile,
  __out        LPVOID lpBuffer,
  __in         DWORD nNumberOfBytesToRead,
  __out_opt    LPDWORD lpNumberOfBytesRead,
  __inout_opt  LPOVERLAPPED lpOverlapped
);
```

The first thing to notice is that the return value of the function is `BOOL` and the calling convention is `WINAPI` (look this up). The function takes a file handle, a void pointer to your buffer, and how many bytes to read. It then fills the buffer and returns the number of bytes written using the `DWORD*` pointer you supplied during the call. This argument is optional only when using overlapped I/O (see function description) and should be present otherwise. The last argument is always optional and specifies a pointer to an `OVERLAPPED` structure that can be used to make the function asynchronous.

In general, parameters marked with `__in` must be initialized prior to calling the function. Those marked with `__out` receive output results, usually through the use of pointers, but do not need to contain any specific initial values. Those with `__inout` must contain initial values that are overridden after successful completion.

For optional parameters that you do not wish to provide, pointers are usually set to NULL, while scalar datatypes to 0, although you should read the entire function description to see if MSDN says otherwise.

1.3. Return Values

Always read MSDN about what each function returns and how to check for errors. *You should be prepared that any API can fail, even at the most unexpected time.* Thus, you must write code that remains robust to API failure (i.e., recovers from errors in ways that do not lead to crashes, undetected error conditions, or deadlocks). The correct way of processing fatal errors is to print the error code (optionally the function where this occurred and API name) and then quit the program:

```
if (someAPI (...) != SUCCESS)
{
       printf ("%s: someAPI failed with code %d\n", __FUNCTION__, GetLastError());
       exit (-1);
}
```

To put this into practice, consider the following piece that tries to print the first 128 bytes of a file:

```
HANDLE h = CreateFile (name, ...);          // open file for reading
```

```
char buf [128];
ReadFile (h, buf, 128, ...);
printf ("%s\n", buf);
```

First, suppose the file does not exist. In that case, `CreateFile` returns an invalid handle, which the program then attempts to use in other API calls. Second, suppose the file exists, but has length 0. Then this fragment will print garbage from an uninitialized array. A similar result occurs if there is an error *while* reading the file. The final issue is that `printf` is likely to run outside buffer boundaries because the received data is not NULL-terminated, creating another crash possibility.

The proper approach is to first check if `CreateFile` succeeds; if not, the program should print the result of `GetLastError()` and quit. Next, if the program passes this stage, it has to check the return value from `ReadFile` and how many bytes were read. If more than 0 bytes arrived, the code may proceed to NULL-terminating the buffer and then printing its contents. The final caveat is that in this example you can read only up to 127 bytes, always leaving at least one byte for the NULL.

Exercise: write a bullet-proof version of the above code segment.

1.4. Unicode

It is recommended that you disable Unicode in your project properties. This is accomplishing by changing General → Character Set to "Not Set." Otherwise, you will have to deal with wide-char strings in many system APIs.

The general Microsoft term for wide (i.e., 2-byte) chars is `wchar_t`, which is `typedef`'ed to the more commonly seen WCHAR. To allow the same code to compile with Unicode enabled or disabled without hacking the program, Microsoft introduced another type called `TCHAR` that many Unicode-friendly APIs now use on MSDN:

```
#ifdef UNICODE
 typedef WCHAR TCHAR;
#else
 typedef char TCHAR;
#endif
```

In simple terms, `TCHAR` maps to regular chars if Unicode is disabled and wide chars otherwise. If you have to proceed with the Unicode route, you can use the following helper macros:

```
WCHAR strW[] = L"hello";            // wide-char version of string
TCHAR strT1[] = _T("hello");        // either WCHAR or CHAR
TCHAR strT2[] = TEXT("hello");      // same
```

and typedef shortcuts:

```
LPTCSTR = TCHAR*                    // NULL-terminated
LPWCSTR = WCHAR*                    // NULL-terminated
```

For Unicode text manipulation, you can use `wprintf, _tprintf, swprintf, _stprintf,` and their numerous variations.

1.5. Variable Names

Microsoft's naming convention has two basic rules. First, variables start with a lower-case letter (sometimes indicating what type of data it is), with each subsequent word capitalized, e.g., bRet, nBytesToRead. Second, functions are similar, except they don't have the initial datatype-indicator, e.g., ReadFile, GetOverlappedResult. On Unix, a more common approach is to use underscore to separate words. You can use either convention as long as you are consistent.

1.6. Linker Input

MSDN provides the necessary header files and libraries for most APIs at the bottom of the help page. For example, `timeGetTime()` requires `windows.h` and `winmm.lib`. Libraries can be inserted through Project Properties → Linker → Input → Additional Dependencies. You can also use

```
#pragma comment(lib, "winmm.lib");
```

from within the source code.

1.7. Command-Line Arguments

To input command-line arguments into Visual Studio, right click the project in Solution Explorer, then select Properties → Debugging → Command Arguments.

1.8. Precompiled Headers

Default projects in Visual Studio utilize precompiled headers, which allow much quicker compilation of the program. There are two basic rules to follow. First, every .cpp file must start with `#include "stdafx.h"`. Second, you should include all standard header files (e.g., `windows.h, stdlib.h`, and STL) in `stdafx.h`, which gets compiled once and is then used to support all .cpp files in your project. Unlike traditional compilation, modification to .cpp code under precompiled headers does not require any of the .h files to be revisited.

You should also include your own frequently-used .h files in `stdafx.h`. To avoid looping and double-inclusion, it is a good idea to start all header files with `#pragma once`, which prevents the compiler from including the file more than once per .cpp.

1.9. When MSDN is Insufficient

If MSDN fails to provide enough information, use other sources to obtain answers. You should persist until you completely understand the underlying functionality of the APIs in question and the cause of the various undesirable phenomena your code may exhibit. You

should strive to fix every bug in your program and make it produce the results *you* want, not something random it does on its own.

Start with Googling the specific error conditions and/or function usage. Read tutorials and other people's recommendations (e.g., stackoverflow.com) for similar problems. If this fails, post general-scope (i.e., not involving your code) questions in Piazza. In more specific cases, you can email the instructors for help, providing them with as much data and your analysis as possible.

2. General C/C++

This section covers standard C/C++ problems that students encounter when dealing with system APIs and memory.

1.1 Pointer Background

Many programming tasks require manipulation of buffers using C/C++ pointers. While textbook pointers cover the basic concepts and syntax, this often turns out insufficient to prevent major headaches during systems programming.

The first prerequisite is to thoroughly understand how C/C++ deal with pointers, which can be accomplished by reading online tutorials:

http://boredzo.org/pointers/
http://pw1.netcom.com/~tjensen/ptr/pointers.htm
http://www.augustcouncil.com/~tgibson/tutorial/ptr.html
http://www.cplusplus.com/doc/tutorial/pointers/
http://www.cprogramming.com/tutorial/lesson6.html

and/or select chapters in C books:

B.W. Kernigan and D.M. Ritchie, "The C Programming Language," Prentice Hall, 1988.
I. Horton, "Beginning C: From Novice to Professional," Apress, 2006.

The rest of this document assumes sound knowledge of these concepts.

1.2 C Strings

A huge number of problems arise from not understanding C strings. Recall that C terminates each string with a NULL character, which is done so that string-manipulation functions (e.g., `strlen, strcpy, printf`) know when to stop scanning the string. NULL is not required if you do not rely on these functions, but you should be aware of its presence when dealing with normal string objects and take it into account during memory allocation. Consider this fragment:

```
char str1 [] = "hello";
```

```
char buf [128];
char *str2 = new char [strlen(str1)];        // allocates 5 bytes
strcpy (str2, str1);                         // copies 6 bytes, overflows
memcpy (buf, str1, strlen(str1));            // copies "hello" without the NULL
printf ("%s", buf);                          // prints garbage, then possibly crashes
```

Observe that `strcpy` NULL-terminates the destination string and thus writes outside the block allocated to `str2`, which causes heap corruption (in some cases, this may also lead to a crash with an access violation). Furthermore, since `buf` in this example is not NULL-terminated after `memcpy`, `printf` scans the entire 128 bytes searching for the trailing zero and prints a bunch of random values found there. If none of them happen to be NULL, it runs outside the allocated space and starts printing everything else found in the stack after the buffer. This continues until a zero byte is found or the program hits the end of the allocated stack region and crashes.

Buffer overflows like these are probably the most common problem in using pointers. While they are relatively simple to debug in single-threaded programs, multi-threading makes these issues a nightmare, which means that pointers *should be thoroughly understood and debugged before parallelization takes place*.

It is also a good idea to make sure you grasp how number encoding works, including binary numbers, ASCII digits, and signed representation:

```
char a = 0, b = '0', c = 250, d = NULL;
printf ("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);
```

Exercise: what are the printed values?

See http://www.asciitable.com/ for more information on ASCII and http://en.wikipedia.org/wiki/Signed_number_representations for common ways to represent signed integers.

It is recommended that you become familiar with traditional C functions `strlen`, `strcpy`, `strcmp`, `stricmp`, `strncmp`, `strcat`, `memchr`, `strtok`, `memset`, `memcmp`, `strstr`. See the following for more:

http://msdn.microsoft.com/en-us/library/f0151s4x(v=vs.80).aspx

1.3 Memory and Pointers

From the perspective of the CPU, memory is a *sequence of bytes* rather than some magical data structure, which is a common misconception for students with an extensive STL and/or Java background. Compilers for low-level languages (C/C++, Assembler) subscribe to the same philosophy as the CPU and require the programmer to explicitly tell them how to interpret each RAM location through the use of pointers. For example, nothing stops you from reading a `char` array using an `int` pointer and vice versa, or exceeding boundaries of the allocated space. With greater flexibility and higher performance of C/C++ comes the potential for crashing the code and corrupting memory in rather subtle ways, which require significant debugging effort. However, this problem

can be overcome if sufficient energy is put into studying and practicing pointer usage. This also may help in the future as many software companies (e.g., Google, Microsoft) ask various pointer questions during their interviews.

To refresh, you can treat pointers as arrays:

```
int arr [100];
// initialize arr by writing a sequence of bytes with value 1
memset (arr, 1, sizeof (int) * 100);
int *ptr = arr;               // points to the array
// read the first element
int a = *ptr;                 // or alternatively a = ptr[0]
int a1 = *arr;                // or arr[0]
// reading the fourth element of arr
int b = ptr [3];              // preferred way of reading values
int b1 = arr [3];
// more tedious version of the same
int c = *(ptr + 3);           // or even worse: *(&ptr[3])
int c1 = *(arr + 3);
```

The lesson here is to use notation that is the simplest and most revealing about your intentions. If you are using a pointer to read an array, then square brackets [] is the best route. *Exercise: figure out the values of a, b, c in the fragment above.*

Along similar lines, many API functions return values through the use of pointers. Suppose we have this declaration:

```
void DoSomething (__in int op, __out LPDWORD result);
```

This function takes as input integer `op` and returns the result into another integer. A common approach for students unfamiliar with pointers is to declare a pointer variable and pass it to the function:

```
LPDWORD result;                     // same as DWORD *result;
DoSomething (OP_READ, result);      // crash
```

This snippet crashes because the function writes into an uninitialized pointer. After some debugging, students usually change this to:

```
DWORD result;
LPDWORD result_ptr = &result;
DoSomething (OP_READ, result_ptr);
```

While this works, it is unnecessarily cluttered. A better approach is to reduce the amount of notation and make the intended meaning of the second argument more clear:

```
DWORD result;
DoSomething (OP_READ, &result);
```

1.4 Pointer Arithmetic

Recall that pointer arithmetic moves the pointer by the *size of the datatype it was declared as pointing to*. For example, notation `p + 3` adds 12 bytes to the pointer if `p` is an `int*` or 24 bytes if it is a `double*`.

Exercise: write an expression that moves ptr (defined as an int) by 3 bytes forward.*

1.5 Casting Pointers

Pointers hold addresses of RAM locations, expressed as *byte offsets* from some initial address allocated to your program. There is never any attached meaning with the pointer as to how large the allocated data segment is or how it was used in the past. This means you are free to change the meaning of any address depending on what is convenient.

`Void*` is the only pointer type that does not tell the compiler the size of the data it points to. Thus, void pointers cannot be dereferenced or included in arithmetic operations; however, they can be converted to other pointer types when access is required. A common use for void pointers is passing or returning generic memory addresses to/from functions:

```
// argument op determines the return type
void *someFunc (int op);             // declaration

char *ptrA = (char*) someFunc (1);   // returns a C string
int *ptrB = (int*) someFunc (2);     // returns an int array
```

For non-void pointers, the declared type tells the compiler how to read/write RAM at that particular address *if and when this pointer is dereferenced*, but nothing more. This interpretation may change during run-time as desired:

```
char str [] = "hello";        // str is a pointer to some location on the stack
int *p = (int*) str;          // cast a char* to an int*
printf ("str = %p, p = %p, initial 4 bytes are %X\n", str, p, *p);
```

Exercise: what are the printed values?

Starting with C++, conversion between non-identical pointer types requires an explicit cast. If the compiler complains about incompatible pointers, you should make sure that casting is done properly. Given successful compilation, the code above prints the two pointer values and the first four bytes of the string as a hex[1] integer.

Another example is to read from the front of a character buffer a 4-byte integer and then an 8-byte `uint64`:

```
char buf [128] = "hello world!!";
int a = *(int*) buf;                          // read the first 4 bytes as int
uint64 b = *(uint64*) (buf + sizeof(int));    // read the next 8 bytes as uint64
```

An equivalent version:

```
char buf [128] = "hello world!!";
```

[1] Using hex notation is helpful in general. The reason is that it explicitly maps each byte to two consecutive hex digits and greatly helps in debugging. It also keeps numbers shorter, which becomes quite noticeable with 32-bit and especially 64-bit values.

```
int *ptr1 = (int*) buf;
uint64 *ptr2 = (uint64*) (ptr1 + 1);         // notice pointer arithmetic!
int a = *ptr1;
uint64 b = *ptr2;
```

Neither one is elegant. A better approach is to declare a class/struct that contains both elements we are trying to read and give them some meaningful names:

```
class SomeData {
public:
       int    len;
       UINT64 hash;                 // UINT64 is declared in Windows.h
};
char buf [128] = "hello world";
SomeData *sd = (SomeData*) buf;
printf ("len = %d, hash = %I64X\n", sd->len, sd->hash);
```

Now suppose that there is an array of floats immediately following the `SomeData` class in the buffer. The length of this array is given by `sd->len`. Continuing the previous example, printing the array is quite simple:

```
float *arr = (float*) (sd + 1);              // notice how pointer arithmetic helps
for (int i = 0; i < sd->len; i++)
       printf ("arr [%d] = %f\n", i, arr [i]);
```

Exercise: modify the loop to prevent overflowing the 128 bytes allocated to buf in case `sd->len` happens to be corrupted/invalid.

1.6 Struct Packing

When using structs to read raw buffers, it is important to be aware of compiler-related data alignment. Consider the following:

```
class SomeData2 {
public:
       int a;
       int b;
       char c;
};

char str [] = "hi there";                    // 9 bytes allocated
SomeData2 x;
memcpy (str, &x, sizeof (SomeData2));        // overflow
```

This example aims to copy object `x` (containing 9 bytes worth of elements) in place of the string. Instead, the result is stack corruption that happens due to *struct alignment*. In this case, the compiler pads the end of the class to a multiple of `sizeof(SomeData2::a)`. This is needed to prevent alignment issues when working with arrays of structs. As a result, `sizeof(SomeData2)`comes back with 12 bytes rather than 9.

A related problem occurs in our next example that intends to read the first byte of `str` using `p->a`, the next 2 bytes using `p->b`, and finally the next 8 using `p->c`:

```
class SomeData3 {
public:
       char a;               // offset 0 from beginning of class
```

```
        short b;              // offset 2, not 1
        __int64 c;            // offset 8, not 3
};

char str [] = "hello there";          // 12 bytes allocated
SomeData3 *p = (SomeData3 *) str;
printf ("buffer contents: %d, %d, %I64d\n", p->a, p->b, p->c);      // overflow
```

The overflow here occurs because individual fields are *themselves* aligned to a multiple of their own data-type size, meaning that a `short` may consume 2, 4, or even 8 bytes depending on what follows it. It is easy to see then why `SomeData3` contains 16 bytes.

To solve this problem, you must pack all sensitive structs (e.g., network headers, any data sent to another program or kernel) to 1 byte:

```
#pragma pack(push,1)          // save current packing, then change to 1 byte
class SomeData4 {
public:
        char a;               // offset 0 from beginning of class
        short b;              // offset 1
        __int64 c;            // offset 3
};
#pragma pack(pop)             // restore old packing
```

Because packing persists until changed, it applies to all .h files included afterwards. *It is thus critical that you restore default packing after you are done declaring your classes.* In fact, certain system .h files are known to break with incorrect packing, which leads to strange compiler complaints and unpleasantly long debug sessions.

Working with pointers to unnaturally aligned structs, MSDN recommends using a special keyword that tells the compiler to optimize its element-fetching routine (this only affects speed, not correctness of execution):

```
SomeData4 __unaligned *p = new SomeData4 [100];
```

1.7 Sizing Data

Instead of hardwiring 4 for the size of an integer or 11 for `SomeData4` above, the recommended approach is to always use `sizeof()`. This avoids mistakes in manual computation of object length and allows easy modification to the class without having to change all hardwired constants.

The main pitfall with `sizeof()` is that it is often incorrectly applied to *pointers*:

```
class Circle {
public:
        double x;
        double y;
        double radius;
};
Circle c, *ptr = &c;
// the next function takes a void pointer and the size of the data
ProcessData ((void*) ptr, sizeof (ptr));
```

In this case, `sizeof` returns 4 in Win32 and 8 in x64, while the correct size of the structure is 24 bytes, regardless of the architecture. The proper version sizes the class directly and avoids declaring a redundant variable `ptr` altogether:

```
Circle c;
ProcessData ((void*) &c, sizeof (Circle));          // sizeof(c) is possible too
```

1.8 Byte Order

You should be aware of how the CPU stores data in RAM, specifically various integer types (i.e., `short, int, __int64`). Two approaches are the *least-significant* byte (LSB) first (e.g., Intel, AMD) and the *most-significant* byte (MSB) first (e.g., Motorola). Consider string "hello" again. In this case, the memory holds 6 bytes:

0x68, 0x65, 0x6C, 0x6C, 0x6F, 0

Performing a fetch operation on this buffer using an Intel CPU:

```
int val = *(int*) str;
```

makes the CPU assume that `str` points to an integer stored in LSB order and results in `val = 0x6C6C6568`. See the following for a more detailed discussion:

http://en.wikipedia.org/wiki/Endianness

Exercise: write an expression that loads the same 4 bytes from the front of str into an integer, but in MSB order and without knowing the underlying LSB/MSB architecture.

When you need to convert your local representation to MSB (also called *network byte order*), you can use networking functions `htons` (host-to-network short) and `htonl` (host-to-network long) that simply flip the byte order on LSB machines and keep the value unchanged on MSB. Thus, one solution to the last exercise would be:

```
int val = htonl (*(int*) str);
```

However, you should think of other ways that do not require any special APIs.

1.9 Operator Precedence

Another source of bugs is incorrect assumptions about operator order. For example:

```
char str[] = "hello";
int len;

if (len = strlen (str) == 5)
       printf ("good\n");
else
       printf ("something's shady\n");
// what is the value of len here?
```

Since assignment has lower precedence than comparison, the above assigns 1 to `len` if the string length is 5 and 0 otherwise. Probably the intended meaning was:

```
if ((len = strlen (str)) == 5)
```

Another example:

```
int a = 5 << 3 + 1;
```

Here, addition has a higher precedence than bit shifting, which results in compiler's seeing:

```
int a = 5 << (3 + 1);         // a is now 80 instead of 41
```

For a complete list, see:

http://en.cppreference.com/w/cpp/language/operator_precedence

1.10 Bit Shifts

Another often overlooked, but powerful, technique involves bit manipulation. In some cases, bit operations significantly speed up computation (e.g., bit shifting vs multiplication). In other cases, they are required to read/write individual bits that encode certain information that the program needs. See the following for the basic syntax and usage:

http://www.cprogramming.com/tutorial/bitwise_operators.html
http://en.wikipedia.org/wiki/Bitwise_operation

Bit shifts are also often used in assembling numbers from individual bytes. For example, given two single-byte values, we can create a two-byte number as:

```
char x = 0x44, y = 0x20;              // x is MSB, y is LSB
short z = (x << 8) + y;               // z is now 0x4420
```

While this may work in some cases, there are additional caveats. First, when `x` is shifted left, it may exceed the size of the default allocation unit (typically an int, but this might be compiler-specific) and result in an overflow. It is thus advisable to perform an explicit cast to the larger datatype *before the shift*.

```
int x = 0x44, y = 0x20;                     // x is MSB, y is LSB
__int64 zA = (x << 32) + y;                 // overflow!
__int64 zB = ((__int64)x << 32) + y;        // correct
```

Second, when combining bytes declared using a *signed* datatype, the sign bit gets propagated with pretty undesirable side effects:

```
char x = 0xF4, y = 0x20;              // x is MSB, y is LSB
int z = (x << 8) + y;                 // z is now 0xFFFFF420 instead of 0xF420
```

Thus, it is important to convert each byte to an unsigned datatype *before shifting it*. With this in mind, the correct version of the above is:

```
char x = 0xF4, y = 0x20;                    // x is MSB, y is LSB
int z = ((unsigned char)x << 8) + y;        // z is now 0xF420
```

It might be temping to directly upconvert `x` to `unsigned int`, but that again leads to problems since 0xF4 is treated as a signed 1-byte number whose sign bit propagates to 4 bytes. Perhaps the safest route is to operate with all-unsigned types:

```
typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned __int64 uint64;

uchar x = 0xF4, y = 0x20;            // x is MSB, y is LSB
uint z = ((uint)x << 8) + y;         // OK
uint64 w = ((uint64) z << 32) + z;   // OK
```

Similarly, when using chars as indexes into arrays, keep an eye on values above 0x7F (127 decimal). When the compiler implicitly upconverts them to an `int` (Win32) or `__int64` (x64), they become negative. For example, the following code incorrectly computes the histogram of byte frequency:

```
char str [] = "hello...";            // suppose the string may contain Unicode chars
int hist [256];                      // histogram values
memset (hist, 0, sizeof (int) * 256); // init to zero
// the loop runs while str [i] != 0
for (int i = 0; str [i]; i++)
       hist [str [i]] ++;            // memory corruption when str[i] > 127
```

The correct version casts each byte of the string to `unsigned char`:

```
       hist[(uchar) str [i]] ++;
```

or alternatively:

```
uchar *str2 = (uchar*) str;
for (int i = 0; str2 [i]; i++)
       hist [str2 [i]] ++;
```

1.11 Access to Bits

Another common task is to access bit groups within a single byte or a larger combination of bytes. For individual bits, the standard approach is to use bitwise operations:

```
DWORD value = 0x893784;       // some integer
int bit = 3;                  // which bit to examine, right-most bit is the 0th bit
DWORD mask = 1 << bit;
if (value & mask == 0)
{
       printf ("bit %d is not set! I will set it\n", bit);
       value |= mask;
}
else
     printf ("bit %d is already set!\n", bit);
```

Exercise: fix this fragment to leverage C++ operators in correct order.

Exercise: change the above segment to clear the same bit if it is already set and leave it untouched otherwise.

For multi-bit groups, bitmasking becomes tedious. An alternative approach is to break the bytes into bits using structs. Suppose we know the layout of the target memory chunk and assume it represents the flags that an API returns to us:

```
int value = GetStatusCode();  // some integer to examine
class Flags {
public:
       int disk:3;            // disk code: lowest bits 0-2
       int network:2;         // network code: bits 3-4
       int other:27;          // remaining 27 bits
};

// interpret the return value as of type Flags
Flags *ff = (Flags *) &value;
if (ff->network == STATUS_OK)
       printf ("operation successful\n");
else
     printf ("network error %d\n", ff->network);
```

For more information, see:

http://www.cs.cf.ac.uk/Dave/C/node13.html

1.12 Large Buffers

A common technique for speeding up and simplifying buffer operations is to use `mem*` functions (e.g., `memset, memcpy, memcmp`). For example:

```
double data [2000], copy [100];
// make a copy of the first 100 values
memcpy (copy, data, sizeof(double) * 100);
```

Note that these functions require conversion of array size to *bytes* using `sizeof`.

1.13 Static and Dynamic Variables

Recall that *static* variables are allocated on the stack, *dynamic* from the process heap:

```
void f (void) {
       int a [100];                        // static
       int *b = new int [100];             // dynamic
}
```

The reason why this distinction is important is that your code may crash with stack overflows if you attempt to place huge static arrays into functions. This is especially true if you are running many Win32 threads, which require small stacks to fit in the 2-GB process limit of 32-bit Windows.

C/C++ also have *global* variables (declared outside of functions), which are considered bad programming style and should be avoided whenever feasible.

1.14 Purpose of Header Files

Header files in C/C++ are sometimes misunderstood. Their purpose is *not* to contain your entire program; instead, they should only define classes and function prototypes used in other cpp files/modules. If you split your code into say three separate parts – memory manager, network client, and user interface – each would be written in a separate `.cpp` file[2]. If one of the modules needs services of other modules, it must be able to instantiate their classes. Then, it would include the corresponding `.h` file that defines sufficient information for compilation to succeed. The actual function bodies should be located in separate .obj files and combined into the executable by the linker.

1.15 String Literals

In C/C++, setting up pointers to `char*` literals is allowed by the compiler, but this is not usually recommended. Example:

```
char *ptr = "hello";  // hello exists in compiler's immutable space
ptr[0] = 'a';         // crash or unpredictable behavior
memset (ptr, 0, 5);   // crash or unpredictable behavior
OtherFunction (ptr);  // may crash internally on write
```

It is possible to prevent the code from compiling in the first two examples by explicitly defining the pointer as `const`:

```
const char *ptr = "hello";    // declared as read-only
ptr[0] = 'a';                 // compiler fails
memset (ptr, 0, 5);           // compiler fails
```

However, if multiple people work on the project, someone else may decide they don't like these compilation errors and attempt to override them using explicit casting to non-const pointers, leading to an eventual crash:

```
memset ((void*)ptr, 0, 5);    // compiles, then crashes
OtherFunction ((char*)ptr);   // compiles, then may crash internally on write
```

The preferred approach is to allocate regular static memory for short strings and dynamic memory for larger arrays, e.g.,

```
char ptr[] = "hello";
char *ptr2 = new char [1<<24];              // 16 MB
strcpy (ptr2, "hello");
```

1.16 Double Pointers

A common mistake is to inadvertently create a double pointer when a single pointer is needed:

```
char *buf = new char [128];
ReadFile (handle, &buf, 128, ...);
```

[2] Templated classes and inline functions are an exception to this rule.

This dumps the data on top of variable `buf` (which is just a pointer) instead of the buffer it's pointing to, leading to stack corruption and a crash.

One typical task that actually requires double pointers is creation of arrays of arrays:

```
int **dbl = new int* [100];          // 100 ptrs to int
for (int i = 0; i < 100; i++)
       dbl [i] = new int [i+1];      // allocate i+1 elements for the i-th pointer
```

This sets up 100 arrays whose size increases from 1 to 100 elements. It is a good idea to familiarize yourself with this and even more complex pointer usage.

1.17 Function Arguments

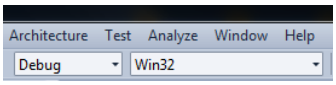
It will be common that your functions require access to some non-trivial-size objects (e.g., queues, sets). A typical mistake in these cases is to pass the object by value:

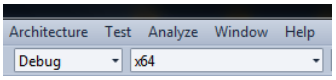
```
set<UINT64> s;
for (int i = 0; i < 1e6; i++)
       s.insert (i*i);                     // add some initial values to s

int n = 1 << 24;                           // size of input array = 2^24
int *inputArr = new int [n];
// count the number of items in array that are squares of other integers
for (int i = 0; i < n; i++)
{
       if (CheckSet (s, inputArr [i]) == true) // true if the set has this item
              count ++;
}
...
bool CheckSet (set<int> s, int val)
{
       ...
}
```

Now, notice how *the entire set gets copied* into the function 16M times within this loop. Changing `CheckSet()` to accept a pointer to the set makes the loop run 186K times faster in debug mode and 925K times faster in release mode.

3. Debugging

This section deals with basic single-threaded debugging techniques and their usage in Visual Studio. Multi-threading is covered in the next section.



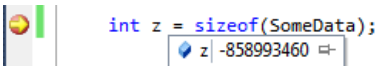


3.1. Debug Mode

When a program is compiled in debug mode, you can run it with or without the debugger attached to the process. The former case, started with F5, allows you to stop on breakpoints and trace the program. To exit the debugger, press Shift-F5. The latter case, started with Ctrl-F5, generally runs a lot faster, but skips over breakpoints; however, when your code crashes, it still allows you to see the various variables and debug the program.

To set a breakpoint, press F9 on the line where you want the debugger to stop. Once you have control of the program, you can step through it with either F10 (skipping over functions) or F11 (going into functions). You also execute Run To Cursor with Ctrl-F10 or Step Out (i.e., exit the current function) with Shift-F11.

To see variable values, you can point to them with the cursor, keeping in mind that the debugger always stops *before* executing the breakpoint line (e.g., the fragment on the right correctly shows the yet-uninitialized value of z).



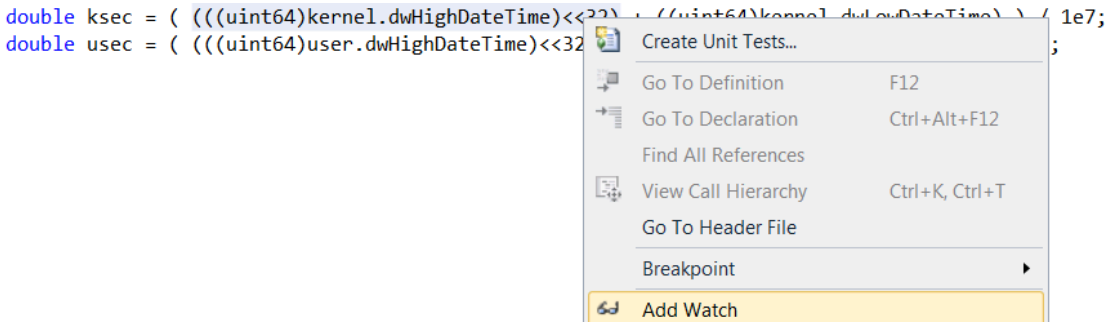
3.2. Debug Windows

There is a number of debug windows in Visual Studio. To see them all, run the program to a breakpoint, then check out Debug → Windows. The most basic of them is the Watch Window (shown as "Watch 1") that allows you to see variable values. You can insert simple expressions into its cells and inspect memory locations. The values can be shown in both decimal (default) or hex (by appending ",x" after the value):

Watch 1

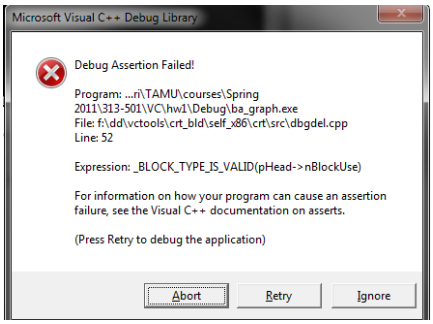
Name	Value
a	0
ptr	0x00000000000df284
ptr +3	0x00000000000df290
(a+3)*200	600
(a+3)*200,x	0x00000258

You can also change the default to hex by right-clicking the white space and selecting "hexadecimal display." If you want to add some expression in the program to the watch window, you can select it and right-click to Add Watch:

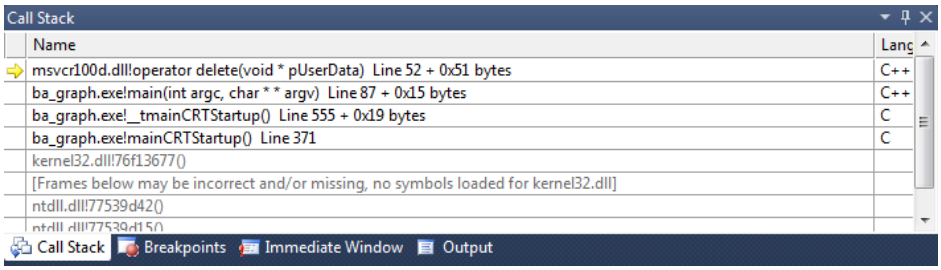


The Auto-Watch version (called "Autos") automatically populates the variables based on what your code is doing. The Local-Watch (called "Locals") shows the variables defined with the scope of the current function. The Immediate Window lets you compute various expressions using command-line format and even run functions that are members of existing classes (e.g., `queue.size()`).

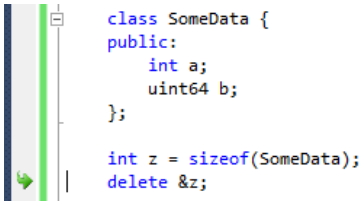
When debugging a crash, perhaps the most useful window is the Call Stack. It allows one to see the nested order of function calls that have led to the crash and unwind the stack to check what parameters/conditions have produced this sequence of calls.



Consider the crash example on the right, where the heap throws a debug assertion. Clicking on Retry, then Break, shows line 52 of `dbgdel.cpp`. This by itself isn't very useful in fixing the problem; however, examining the call stack



shows that the crash was caused by the delete operator called around line 87 of `main()`. Double-clicking `main()`in the call stack brings up the portion of code shown on the right[3]. The bug is obvious here as the pointer being deleted was not obtained from the heap, but in more complex cases you can examine variables in this function using the Watch Window and traverse even further up the stack to see what originated the faulty condition.



```
class SomeData {
public:
    int a;
    uint64 b;
};

int z = sizeof(SomeData);
delete &z;
```

Note that variable values are not available in release mode; however, the call stack works fine in many cases and allows identification of who called the function that crashed. This is very helpful when the crash condition occurs inside someone else's code (e.g., STL) and you have to roll up the call stack to see what you did to cause this.

3.3. Advanced Breakpoints

In some cases, your code will execute the same fragment for many iterations (e.g., millions or billions) without a problem, but will eventually run into something unexpected and crash. The goal in such cases is to reproduce the exact conditions that led to the crash and stop the program just *before* it crashes, not after. This allows you to step through the crashing scenario and observe how it develops this particular problem.

Consider this fragment driven by an LCG (linear congruential generator), which is a high-performance random-number generator with a given period:

[3] The debugger will usually point the green arrow to the *next* line after the call that led to the crash since this is the return value found in the stack. However, in some cases, like with operator `delete` in this figure, it may directly identify the location of the call.

```
DWORD val = 0;                                      // random number, initially 0
int period = 1 << 24;                               // period of LCG, 16M
char *buf = (char*) VirtualAlloc (NULL, 1<<31, ...); // allocate 2 GB
...                                                 // initialize buffer
for (int i = 0; i < (1 << 30); i++)
{
       val = (val * 5 + 1) & (period - 1);          // next random value
       TweakBuffer (buf, val);                      // applies some alg to buf
}
```

Suppose you discover that this program crashes in `TweakBuffer()` after some number of iterations. After the crash, you find out (using the call stack and the watch window) that `i` equals 80,998,213. Obviously, it is unrealistic to manually step through this loop almost 81M times to catch `TweakBuffer` on crashing.

The first technique is to use Visual Studio breakpoint facilities. You can right-click the red ball indicating a breakpoint and select "Conditions..." This opens a box into which one can type a boolean expression, such as `i == 89998213`. The code is supposed to break when this condition becomes true. However, there are two issues with this approach. First, the debugger sometimes misses the correct condition and does not break at all. Second, since the debugger basically wakes up for each iteration to check the condition, your code executes excruciatingly slow. On a 2.8 GHz computer, conditional breakpoints run at 1160 iterations/sec, meaning that grinding through a loop with 80.9M steps takes just over 20 hours. While this approach is viable for a handful of iterations, it becomes inapplicable in general cases.

The second (correct) technique is to embed an if statement that checks for the desired condition and then set a regular (unconditional) breakpoint inside of it:

```
for (int i = 0; i < 1<<30; i++)
{
       val = (val * 5 + 1) & (period - 1);          // next random value
       if (i == 80998213)
              int z = 0;            // F9 <-------------- breakpoint here!
       TweakBuffer (buf, val);                      // applies some alg to buf
}
```

Ignoring the overhead of `TweakBuffer`, this version gets through 80.9M iterations and breaks on the desired condition in 160 ms.

3.4. Debug Too Slow

In certain cases, the debug-mode version of your program takes too much time to crash. This often occurs when you have a large number of iterations with calls to libraries that implement extravagantly meticulous debug checks, with two typical cases being STL and `new/delete`. For debugging in release mode, which normally does not show correct variable values, see the following:

http://msdn.microsoft.com/en-us/library/fsk896zz.aspx

3.5. Typical Errors

Many standard libraries throw debug assertions as a way to prevent a crash at some later point. Most of these are memory-related, such as heap/stack corruption, deletion of invalid pointers, and bad_alloc (e.g., due to insufficient RAM). In some cases, they indicate conditions that potentially lead to a bug (e.g., usage of uninitialized variables).

In release mode, debug assertions are ignored and your program is left to crash in some other (possibly unrelated) location, which makes debugging harder. In some scenarios, however, the release mode may not crash at all, *which does not by itself mean the code is safe*. Instead, successful execution simply shows that the bug did not surface in ways that the CPU or the kernel were able to notice.

In addition to assertion failures, there are also *hard-crash* conditions (exceptions) that the CPU throws and the debugger catches, the most common of which is an access violation (i.e., reading from/writing to an invalid pointer). Next, if you run into attempts to execute an invalid instruction or a data segment, the most likely cause is your stack getting wiped and the return address of some function pointing to garbage. In such cases, you should check where you might be overflowing a static array declared on the stack.

Stack overflows is another common crash condition that arises when you place too much data into local variables inside functions and/or recurse too much. Any array larger than a couple of KB should be allocated dynamically from the heap. While the default thread stack in Windows is 1 MB, homework sometimes requires lowering this to 64 KB, which leaves little room for large local arrays.

It is important to remember that release and debug builds result in different versions of the program (e.g., due to optimizations and reordering of instructions) and apply different initialization to buffers, which explains why crashes seen in one type of build may go unnoticed in another. Thus, you should always verify your results in both.

3.6. Magic Numbers

When you print your variables in hex and see these special cases, you can easily detect where they came from (most likely out-of-boundary array violations):

```
cdcdcdcd, baadf00d                  uninitialized heap memory
cccccccc                            uninitialized stack memory
abababab, fdfdfdfd                  heap guard blocks
feeefeee                            freed heap memory
```

You should back-trace to the first occurrence of these values in your printouts and find out how they come about. For a longer list, see:

http://en.wikipedia.org/wiki/Magic_number_(programming)

3.7. When the Debugger is Insufficient

While the debugger is very useful in general, there are conditions it cannot help with. The best type of bug is the one that results in a crash and leads you to the exact line that you need to fix. However, in certain cases, the program may deadlock or produce an incorrect result without explicitly crashing. In these situations, you have the oldest and still most powerful debugging tool at your disposal – `printf`. To find out where the code deadlocks, you can scatter `printfs` across the program and see the last printout before the code hangs. Some of this functionality can also be accomplished by breaking into a hanging program from the debugger (start with F5, then Debug→Break All…) and examining the call stack, which should tell you what the program is currently doing.

When Break All fails to produce anything useful and for debugging program logistics, the best route is to create a detailed trace of what your code has been doing and how it got to the final (incorrect) output. Start with printing a message for every major step, action performed, and variables involved. Then sit down with a piece of paper and trace by hand what the correct sequence should have been. Once you identify the step where things go wrong, add more printouts targeting specifically that step and its substeps (if any). For this to be feasible, the input parameters to the program must result in a small-enough computation that you can manually trace.

3.8. When Printf is Insufficient

For every debugging technique, including `printf`, there is some code that defeats it. In addition, even if `printf` eventually leads to finding the bug, in some cases it might require such an enormous amount of time that the whole approach becomes infeasible. In such cases, you can unleash the *divide-and-conquer* technique on the program, which entails commenting out large chunks of the code, patching the remainder to run, and observing what combinations of code exhibits the flaw. In some cases, this also requires replacing certain components with equivalent constructs written by others. For example, consider:

```
for (int i=0; i < 1e9; i++)
{
        DWORD x = GetSomeVal (i);
        if (hashTable.Add (x) == NOT_FOUND)
        {
                hashTable.Add (x);
                queue.Push (AnotherFunction (hashTable.Find (x+2), i, buf));
        }
        ComputeStats (buf);
}
```

When this loop runs and eventually produces something incorrect, the starting point is to understand what is causing the bug – `GetSomeVal, hashTable.Add, hashTable.Find, queue.Push, AnotherFunction,` or `ComputeStats`. To rule out cross-function contamination, one could start by commenting out `ComputeStats` and seeing if the other functions work as expected. If the bug persists, you can then replace your version of the hash table and queue with STL and so on. By narrowing down the cause of the problem, you can then zoom into the specific (buggy) function and repeat the same process therein.

Of course when there are multiple bugs in different functions that create errors in the final result only through *interaction* with each other, this process may take much longer.

3.9. Debugging Small Chunks

One code-development technique, often seen in CS homework, is to write (sometimes on paper) the entire program before even trying to compile it. While this might be fine in theory for tiny programs, writing lots of code at once is poorly scalable (i.e., does not work for large programs) and is a debugging nightmare as the programmer faces a huge codebase full of bugs on the very first run. If we assume that a function contains k bugs on average and the debugging complexity scales quadratically with the number of bugs, the overall workload involved in debugging a complete N-function program is $(Nk)^2$.

A much better approach is to come up with some initial high-level design, but then perform code development in stages that are closely coupled with debugging. Placing the various functional pieces into separate functions/classes and *debugging them as soon as practically possible* allows the programmer to apply divide-and-conquer principles to the development complexity and keep the total debugging workload at Nk^2 instead of $(Nk)^2$. For example, given $N = 1000$ functions with $k = 2$ bugs each, this reduces the amount of work from 4M units of time (e.g., hours) to 4K. The improvement is by a factor of N, which becomes more noticeable as program size grows.

Furthermore, early debugging allows the programmer to better reason about the problems they are facing and dynamically redesign the remaining pieces to better suit the environment in which the program is expected to run. Unless this is a repeat of an earlier task, coding often requires a gradual build-up before the programmer can achieve an in-depth understanding of the conditions returned by the various APIs, deadlock possibilities, synchronization failures, and various avenues for optimization. Thus, early debugging may allow a significant reduction in development time and may prevent the need to rewrite large chunks of code that were designed on paper without fully understanding the underlying problem.

3.10. Testing

While both writing code and finding crash-related bugs are important, one must also come up with sufficient test cases to verify that the program is robust to various types of input. It is recommended that all new code be stepped through with F10 and the operational logic be checked along each branching path (i.e., for different inputs that trigger those paths). Until the code has been extensively verified, it should not be considered correct.

Poor testing and/or lacking effort to anticipate API/user behavior makes your code easy to break and will result in deduction of points.

3.11. Memory Leaks

Visual Studio will normally show leaked memory in the Output window after the program terminates (run with F5 under debugger), including block size and memory contents. However, it does not tell you the name of the variables or where the allocation took place. To do more advanced debugging, use Visual Leak Detector:

http://vld.codeplex.com/

3.12. CRT Debug Heap

It is sometimes convenient to let the program periodically check for heap corruption and throw an exception as soon as something abnormal is detected. This can be done using a special version of the heap that checks for consistency on every call to memory-related functions (e.g., `new/delete`, `malloc/free`). Example:

```
#include <crtdbg.h>

main ()
{
        _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_CHECK_ALWAYS_DF);
        char *p = new char [100];
        memset (p, 0, 120);           // corruption here
        printf ("Finished\n");
 }
```

The debugger will throw an assert failure on the last line of `main`, because `printf` uses internal calls to `malloc`. However, if you had a ton of code between `memset` and `printf`, this result would be largely useless.

It should also be noted that calls to `new/delete` in the debug heap are extremely slow. This may completely prevent usage of this strategy on certain projects (i.e., those that manipulate huge amounts of data).

3.13. Page Heap

While the debug heap is useful in many cases, it does not always help locate the bug. As explained above, this happens if the delay between memory corruption and the next `new/delete` request is large. A more powerful tool is the *page heap*, which places an inaccessible page immediately after each buffer that is returned by `new/malloc`. In such cases, exceeding array boundaries immediately crashes the program (at least in theory, some exceptions apply). To enable the page heap on a particular process:

1) Find gflags.exe in your Windows SDK (e.g., C:\Program Files (x86)\Windows Kits\10\Debuggers\x64). Visual Studio sometimes installs an incomplete version of the kit, which doesn't have gflags. This can be fixed by visiting

https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk

2) Run **gflags /p /enable yourprogram.exe /full**

3) Run the program in Visual Studio using F5 (under debugger) and wait for it to crash with an access violation. The location of the crash should indicate where the bug is.

After you are done, run **gflags /p /disable yourprogram.exe**

Additional links:

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags

https://stackoverflow.com/questions/2470131/visual-studio-how-to-find-source-of-heap-corruption-errors

4. Multi-Threading

This section deals with multi-threaded debugging. In general, there are no good ways of debugging parallel programs as the most basic tools (such as stepping-thru or breakpoints) become useless. In fact, concurrent programs are usually highly sensitive to timing effects and may not run into any problems if the debugger alters their execution.

Your first goal should be to separate general algorithmic flaws from concurrency bugs, eliminating the former before you attempt to catch the latter. In practice, this translates into running code with *one* thread and verifying that it produces the desired outcome. If bugs persist in single-threaded mode, you know they are not related to synchronization and can be tackled using techniques in the previous section.

Once you have a perfectly working single-threaded program, the next step is to start adding threads and keeping track of the conditions that lead to the problem (e.g., more than 1000 threads crash). The goal here is to find the minimum number (ideally just 2) that allows you to see the problem. Once you settle on this number, make sure the conditions are easily reproducible (e.g., by tweaking the input or running on small datasets) such that you do not have to wait a long time to experience the problem.

The final step is to sprinkle `printf` statements all over the program and start tracking what it is doing. The goal is to identify the first deviation from the correct operation and then stop the program just before it reaches a faulty state. For example, suppose your program searches a graph with 1 billion nodes using BFS and discovers the exit at incorrect distance 11 after crunching through 700M rooms. Since it is impossible to directly detect which of the 700M steps was incorrect, you should start by verifying that the number of nodes at each depth makes sense (e.g., by comparing to some reference output or computing it manually). If you detect that your depth-2 result contains 302 nodes instead of the correct 299, you can significantly reduce the scope of the problem and refocus your energy on just the first 300 nodes. In CSCE 313, you can request an entire trace of rooms in the order they should be discovered and catch your program when it deviates from this list.

If at any point you need to find out what your threads are currently doing, stop the program on a breakpoint (or break into it) and use the Threads Window (Debug → Windows → Threads). This is very helpful in debugging deadlocks.