Microsoft Visual C++ Tips and Tricks

by Alan De Smet

If you're stuck using Microsoft Visual C++, these tips may help make the experience more pleasant. I've been collecting these little tidbits for a while, and decided to collect them in one spot. I've chosen to collect them here because I can't think of a better spot.

First thing, get yourself a copy of Workspace Whiz. Workspace Whiz gives you a nice GUI interface to Exhuberant CTAGS. It also gives you a handy button to switch between the header and source files. It's cheap ($10), and makes life better.

Next, get yourself a copy of STLFilt. It's completely free. It significantly cleans up the garbage error messages Visual C++ generates for STL code. (It does require Perl, but every developer should have Perl installed anyway. The easiest way to get Perl for Windows is ActiveState's free ActivePerl.

Whenever I refer to the "sharedide" directory, you'll need to change it to the appropriate directory. In Visual C++ 5, it's actually called "sharedide" under the directory where you installed VC. In Visual C++ 6, it's in the "Common\MSDev98" directory under the directory where you installed VC.

Show compile duration

Simply add '/Y3' to the command line of VC (In the short cut). You'll now get reports on how long a compile took in your Build window.

Update 2003-01-10: I haven't had a chance to use Visual Studio .NET yet, but I'm told that you can set this option with a switch under Options > Projects > Build.

Teach VC to intelligently expand classes/structs in debugger

Isn't it neat how VC's debugger knows how to intelligently expand CStrings, CPoints, POINTS, and alot of other stuff? Well, you can teach it to handle your own structs and classes. Just edit `autoexp.dat` in `sharedide/bin` (in the directory where Visual Studio is installed.) The format of that file is fairly complicated, so I suggest just copying the examples already in the file.

Add user defined keywords for syntax highlighting

Why can you set a color for user defined keywords in *Tools > Options > Format*? Where do you set the keywords? Easy, just create `usertype.dat` in the `sharedide/bin` directory that Visual Studio is installed in. Put your keywords in that file, one per line.

Custom Tip of the Day

Any files with the extension "`.tip`" in the `sharedide/bin/ide` directory where Visual Studio is installed will be read. You can delete the .tip files Microsoft provides and add your own. You might want to take a look at Microsoft's files to see the format. I personally suggest filling the .tip file with quotes, news, or something more useful and entertaining that the TotDs.

How to use .cc file extensions for C++

Make the following modifications to the registry:

```
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Text Editor\Tabs/Language Settings\C/C++
FileExtensions=cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2;cc;cp

HKEY_USERS\S-1-5-21-1219703950-274334628-1532313055-1335\Software\Microsoft\DevStudio\6.0\Build System\Components\Platforms\Win32 (x86)\Tools\32-bit
Input_Spec=*.c;*.cpp;*.cxx,*.cc,*.cp

HKEY_USERS\S-1-5-21-1219703950-274334628-1532313055-1335\Software\Microsoft\DevStudio\6.0\Build System\Components\Tools\<Component 0x3>
Input_Spec=*.c;*.cpp;*.cxx;*.cc;*.cp
```

Add the flag "/Tp" to the compiler settings for the project.

Removing the "docking" capability from the menus

In *Tools* > *Options...*, in the Workspace tab, turn on "Use screen reader compatible menus". Your menus will lose the gripper (the double line on the left edge indicating dockability), and will stay nailed down like they should. Unfortunately this also removes the icons from the menus.

Useful build messages

The following macros make it easy to add reminders which are displayed when code is compiled. You can double click on a reminder in the Output Window and jump to the line. Useful for marking TODOs. (Originally from Windows Developer Journal, 1997?)

```
// Statements like:
//              #pragma message(Reminder "Fix this problem!")
// Which will cause messages like:
//              C:\Source\Project\main.cpp(47): Reminder: Fix this problem!
// to show up during compiles.  Note that you can NOT use the
// words "error" or "warning" in your reminders, since it will
// make the IDE think it should abort execution.  You can double
// click on these messages and jump to the line in question.
#define Stringize( L )                  #L
#define MakeString( M, L )              M(L)
#define $Line                                   \
        MakeString( Stringize, __LINE__ )
#define Reminder                                \
        __FILE__ "(" $Line ") : Reminder: "
```

Once defined, use like so:

```
     #pragma message(Reminder "Fix this problem!")
```

This will create output like:

```
     C:\Source\Project\main.cpp(47): Reminder: Fix this problem!
```

Hard code a debugger breakpoint

If you need to insert a hard breakpoint in your code (perhaps because you need to attach to a process), simply add the following line to your code.

```
__asm int 3;
```

Tracking GDI resource leaks

Plenty of tools exist to help track down memory leaks. You've got the debug heap, Rational Purify for Windows, HeapAgent, and other tools. But there aren't any good tools to help track GDI resource leaks. A resource leak can crash the system under Windows 95 or Windows 98, and can ruin performance on any Windows operating system.

The article "Resource Leaks: Detecting, Locating, and Repairing Your Leaky GDI Code" is the only real help I've found. In particular, the article includes a program that tracks current globally allocated resources under Windows 95 or Windows 98. You can track what resources were created between a start and stop points, and display what those resources are. Check the article for the download (Leaks.exe).

Update 2003-01-10: The very friendly people at Compuware have pointed out that BoundsChecker will detect GDI resource leaks. At the moment I'm doing Unix development work, so I haven't had a chance to test it myself, but if you're fighting GDI leaks, it might be worth checking out. At the moment it appears to carry a $695 price tag. They offer a trial period. (Regrettably of the "give us your contact info and we'll contact you about a trial" type.)

Memory Values

Check this page form information on "Funny" Memory Values. In particular:

If you're using the debug heap, memory is initialized and cleared with special values. Typically MFC automatically adds something like the following to your .cpp files to enable it:

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

You can find information on using the debug heap here. Microsoft defines some of the magic values here.

While using the debug heap, you'll see the values:

Value	Usage
0xCDCDCDCD	Allocated in heap, but not initialized
0xDDDDDDDD	Released heap memory.
0xFDFDFDFD	"NoMansLand" fences automatically placed at boundary of heap memory. Should never be overwritten. If you do overwrite one, you're probably walking off the end of an array.
0xCCCCCCCC	Allocated on stack, but not initialized

Watch Values

The watch window has a number of undocumented or minimally documented features that make it even more useful.

Display GetLastError's value and message

You can display the value GetLastError() will return by putting "`@err`" in your watch window. You can see the error message associated with that value by putting "`@err,hr`" in your watch window. If you've placed an HRESULT in a variable, adding "`,hr`" to the variable name in the watch window will display the associated text.

Display pointer as an array

If you expand a pointer and you only get a single item, just add ",n" to the entry in the watch window where n is the number of elements to expand. For example, if you have a `foo * pFoo` pointing to an array of ten elements, put `pFoo,10` in your watch window to see all of the element. This can be useful to view parts of a large array. If pFoo points to an array of 5,000 elements, you might use `(pFoo + 2000),10` to see elements 2000 through 2009.

Debug checked casts

If you want maximum safety, you should always use dynamic_cast. However, if you feel you *must* optimize away those costs, use this version of checked_cast. It will ASSERT on a bad cast in Debug builds, but not do the slightly more expensive dynamic_cast in Release builds.

```
// checked_cast - Uses fast static_cast in Release build,
// but checks cast with an ASSERT in Debug.
//
// Typical usage:
//   class Foo { /* ... */  };
//   class Bar : public Foo { /* ... */  };
//   Foo * pFoo = new Bar;
//   Bar * pBar = checked_cast<Bar *>(pFoo);

template <class TypeTo, class TypeFrom>
TypeTo checked_cast(TypeFrom p)
{
        ASSERT(dynamic_cast<TypeTo>(p));
        return static_cast<TypeTo>(p);
}
```

Don't use SourceSafe

SourceSafe is problematic revision control system. The integration with Visual Studio is *not* worth the trouble. There are better solutions available. I've written a page with specific details on why Visual SourceSafe is a bad choice.

Unverified

These are notes taken by a David Carley from a video of a talk given by a Microsoft Visual Studio developer. Unfortunately the original video is no longer available (or at least hidden). I haven't verified most of these.

Avoiding Stepping Into Things

It's often useful to avoid stepping into some common code like constructors or overloaded operators. `autoexp.dat` provides this capability. Add a section called "`[ExecutionControl]`". Add keys where the key is the function name and the value is "`NoStepInto`". You can specify an asterisk (`*`) as a wildcard as the first set of colons for a namespace or class.

`autoexp.dat` is only read on Visual Studio's start up.

To ignore the function myfunctionname, and all calls to the class CFoo:

```
[ExecutionControl]
myfunctionname=NoStepInto
CFoo::*=NoStepInto
```

To ignore construction and assignment of MFC CStrings: (Notice the extra = in CString::operator=.)

```
[ExecutionControl]
CString::CString=NoStepInto
CString::operator==NoStepInto
```

To ignore all ATL calls:

```
[ExecutionControl]
ATL::*=NoStepInto
```

Function Evaluation

```
    Call program code from debugger
        Use quickwatch window, not watch window.
        e.g. DumpInfo (pFoo)
    Great for debug data
        Use OutputDebugString or printf
    Limitations
        20 seconds max.
        Terminate on exception.
        Only one thread
```

Multi-Threaded Debugging

```
    Current Thread ID:
        NT4: dw @tib+24   (thread information block)
        Win9x: FS register is unique per thread
        You can use this to set a break point to only fire on a
        given thread
```

Naming Threads

Use "`SetThreadName`". The name is limited to 9 characters. SetThreadName fires an exception, which the debugger will catch and use to name the thread. The name will appears in Debug > Threads dialog.

```
#define MS_VC_EXCEPTION 0x406d1388

typedef struct tagTHREADNAME_INFO
{
    DWORD dwType;        // must be 0x1000
    LPCSTR szName;       // pointer to name (in same addr space)
    DWORD dwThreadID;    // thread ID (-1 caller thread)
    DWORD dwFlags;       // reserved for future use, most be zero
} THREADNAME_INFO;

void SetThreadName(DWORD dwThreadID, LPCTSTR szThreadName)
{
    THREADNAME_INFO info;
    info.dwType = 0x1000;
    info.szName = szThreadName;
    info.dwThreadID = dwThreadID;
    info.dwFlags = 0;

    __try
    {
        RaiseException(MS_VC_EXCEPTION, 0, sizeof(info) / sizeof(DWORD),
(DWORD *)&info);
    }
    except (EXCEPTION_CONTINUE_EXECUTION)
    {
    }
}
```

Win32 Exceptions

```
        If unhandled, kills app
        Debug > Exceptions dialog
        Stop Always or Stop if not Handled.
            Debugger sees exceptions before app, has control
            Stop Always lets you see where thrown
            Then, step, see where caught.
        Can add new exceptions
            watch output window
        Decoding C++ Exceptions
            Install ntdll.dbg to winnt\symbols\dll
            Set "Microsoft C++ Exception" to Stop Always
```

Breakpoints in System DLLs

```
    System DLLs
        Windows NT only
            Can't just use the function name.
            Need the dll name too
        Determine DLL
            From Help
            or, grep in lib\win32api.csv
        Determinte the exact, unmangled name
        Set BP on {,,dllname}Function   <=- special syntax
        Find true function name.
            Use dumpbin
            May need to enable "Load COFF & Exports" in 6.0
```

Breakpoint on certain argument values

1. Set breakpoint
2. Determine stack offset to argument (see disassembly window)
3. Set condition e.g. dw esp+0x8 == 0xFFFFFFFF

Data Breakpoints

Data breakpoints are very powerful, but suffers from a confusing interface. Avoid entering "emulation" mode if possible, it's very slow.

It's often useful to break on an address: "`*(long*)0x1234ABCD, length = 1`".

The debugger can only support 4 hardware data breakpoints. If you use expressions (a + b), debugger uses emulation mode.

Data breakpoints only work on x86 processors, and can't catch kernel mode writes.

Data breakpoints can trigger operating system bugs. A data breakpoint set in one process might not be unset when the OS switches processes. This can cause other applications to crash. Windows 9x breaks often. Windows NT breaks less often. Reboot when it happens.

```
Count BPs
    Stop on the Nth iteration
    To Help find N:
        Set BP w/ very large count (C==10000)
        Run until your app crashes
        Look at count value (X)
        Set count BP on C-X-1
By Name
    Stays put through edits (as opposed to F9, for
    file/line number BPs)
```

Memory Leaks

```
Docs are confusing
    Indexed entries talk about MFC
    No clear description
    Go to "Visual C++ Documentation/Using Visual C++/Visual
        C++ Programmers Guide/Debugging/Debugging
        Techniques.Problems and Solutions/Solving Buffer
        Overwrites and Memory Leaks" (6.0)
Include order is important
    Some things redefine malloc and free, etc.
Step 1, include in global header file
    #define _CRTDBG_MAP_ALLOC
    #include <stdlib.h>
    #include <crtdbg.h>
Step 2, enable checks in WinMain:
    // Enables tracking and reporting on shutdown.
    _CrtSetDbgFlag (
        _CRTDBG_ALLOC_MEM_DF |
        _CRTDBG_LEAK_CHECK_DF);
    _CrtSetReportMode ( _CRT_ERROR,
        _CRTDBG_MODE_DEBUG);

    It may list file/line number
    Will list leak number
        set {,,msvcrtd.dll}_crtBreakAlloc = n (leak number)
    Or, look for ASCII clues.
```

Memory Corruption

```
Heap Corruption
    Enable heap checking (slow)
        {,,msvcrtd.dll}_crtDbgFlag = 5
Data BPs may be useful
```

Optimized Code

```
It's hard
    Compile with /Zi
Variables vanish or are wrong
    Use disassembly window for truth
Arguments usually OK except
    this
    fastcall
Just find your bugs in Debug!  8)
```

Casting in the debugger

```
Need correct type name
    Debugger doesn't know typedefs
```

```
        To find the true typename,
            Add variable to watch, RClick, choose properties
    Scoping:
        variable defined in current DLL, type defined in another
        DLL:
        {,,foo.dll}(CMyClass *){*}pObject
        pObject is local, CMyClass defined in foo.dll
```

Modules Window

```
    Debug.Modules
    Sort by:
        Module name
        Address range
            Useful for finding out what module you're in
                Find DLL of EIP
        Full path
            Ensure you're using the correct DLL
        Load order (default)
```

Debug Works, Release Doesn't

- Uninitialized variables
  - Often 0 used in debug builds
  - Unless /GZ switch is enabled
- Under/Overruns of memory
  - use debug heap (running in debugger on NT)
- Wrong calling convention (esp. GetProcAddress)
  - Use /GZ in compiler
- Optimizer unforgiving
- Overwriting locals more likely
  - Locals packed on stack
  - Locals reused

It works When I Don't use the Debugger

```
    Windows NT uses debug heap under debugger
        To avoid, attach to process, rather than launching under
        debugger.
    Thread timing different
    PATH may be different (when lauched under MSDEV, vs shell)
        Call GetEnvironmentString in app to check.
```

Timing Code

Add two watches

```
@clk
@clk = 0
```

Watches are eval'd top to bottom. First one tells you the value of clock, second resets it. Each step will then update these values, and you can do simple profiling. Not good for assembly timing, because of debugger overhead.

Edit And Continue

```
    Works on any builds
        Just use /ZI
    use SetNextStatement before you edit.
        either set it to a block before the section you want to
        edit, or, step out of the function, then step back in.
    Beware if you copy binaries
        Auto-relink only updates the target of the build, not the
        currently running copy.
        Your only warning is in the Output window

    Poor Man's Edit & Continue
        0x90 = NOP
            Use in the disassmebly window to "comment-out" chunks
            of code
        0x74, 0x75 == JE, JNE
            Use to switch the directional sense of an if
            statement.
```

Disassembly Tricks

```
    Disambiguation dialog is annoying
        Choose any
        Ctrl+G (goto address) EIP
        Goes to current location.
```

Remote Debugging Made Easy

```
    1. Local Machine
        Build
        Share directory to application
        Map back to local machine (O:)
        Build.Remote
            TCP/IP
            Settings.Machine
        Project.Settings.Debug
            Remote path = O:\foo\bar.exe
    2. Remote machine:
        Install MSVCMON
            (you can Just install VC)
            (or not, if you want a very clean machine)
        Map share to same letter as local machine
            Makes finding DLLs easier
        Launch MSVCMON on remote
            Click Connect
            ignore all options
    3. Start debugging
        Step & Go OK, no attach
        Default DLL path should be correct (VC6)
```

COM Debugging

```
    Demo
        eax, hr
            Return values are in eax.  If you know it's an
            HRESULT, you can do "eax,hr" and see the error
            message associated with that HRESULT.
        ,wm for window messages
        @err
            Use this in the watch window to display the value
            that GetLastError would return.
        @err, hr will show the msg associated with the
            GetLastError value.
        GUIDS and VARIANTs decoded
            New in VC 6.0
        Make sure "Display Unicode Strings" is enabled
            in the options to show BSTRs etc.
```

Verify validity of COM object

```
// Verify that a pointer to a COM object is still valid
//
// Usage:
//   VERIFY_COM_INTERFACE(pFoo);
//
```

```
template <class Q>
void VERIFY_COM_INTERFACE (Q *p)
{
#   ifdef DEBUG
        p->AddRef();
        p->Release();
#   endif
}
```

Verify validity of a BSTR

```
// Verify that a BSTER really is a BSTR and is handled correctly
// Warning! Fails on BSTRs which contain embedded nulls
// Usage:
//  VERIFY_BSTR(bstrName)
//
#ifdef DEBUG
#    define VERIFY_BSTR(bstr) do {    \
        ASSERT(SysStringLen(bstr) == wcslen(bstr)); \
     } while (0)
#else
#    define VERIFY_BSTR(bstr) do {} while (0);
#endif
```