MFC Debugging Techniques

Visual Studio 2015

The new home for Visual Studio documentation is Visual Studio 2017 Documentation on docs.microsoft.com.

The latest version of this topic can be found at MFC Debugging Techniques.

If you are debugging an MFC program, these debugging techniques may be useful.

In this topic

AfxDebugBreak

MFC provides a special AfxDebugBreak function for hard-coding breakpoints in source code:

```
AfxDebugBreak( );
```

On Intel platforms, `AfxDebugBreak` produces the following code, which breaks in source code rather than kernel code:

```
_asm int 3
```

On other platforms, `AfxDebugBreak` merely calls `DebugBreak`.

Be sure to remove `AfxDebugBreak` statements when you create a release build or use `#ifdef _DEBUG` to surround them.

In this topic

The TRACE macro

To display messages from your program in the debugger Output window, you can use the ATLTRACE macro or the MFC TRACE macro. Like assertions, the trace macros are active only in the Debug version of your program and disappear when compiled in the Release version.

The following examples show some of the ways you can use the **TRACE** macro. Like `printf`, the **TRACE** macro can handle a number of arguments.

```
int x = 1;
int y = 16;
float z = 32.0;
TRACE( "This is a TRACE statement\n" );

TRACE( "The value of x is %d\n", x );

TRACE( "x = %d and y = %d\n", x, y );

TRACE( "x = %d and y = %x and z = %f\n", x, y, z );
```

The TRACE macro appropriately handles both char* and wchar_t* parameters. The following examples demonstrate the use of the TRACE macro together with different types of string parameters.

```
TRACE( "This is a test of the TRACE macro that uses an ANSI string: %s %d\n", "The
number is:", 2);

TRACE( L"This is a test of the TRACE macro that uses a UNICODE string: %s %d\n", L"The
number is:", 2);

TRACE( _T("This is a test of the TRACE macro that uses a TCHAR string: %s %d\n"),
_T("The number is:"), 2);
```

For more information on the **TRACE** macro, see Diagnostic Services.

In this topic

Detecting memory leaks in MFC

MFC provides classes and functions for detecting memory that is allocated but never deallocated.

Tracking memory allocations

In MFC, you can use the macro DEBUG_NEW in place of the **new** operator to help locate memory leaks. In the Debug version of your program, `DEBUG_NEW` keeps track of the file name and line number for each object that it allocates. When you compile a Release version of your program, `DEBUG_NEW` resolves to a simple **new** operation without the file name and line number information. Thus, you pay no speed penalty in the Release version of your program.

If you do not want to rewrite your entire program to use `DEBUG_NEW` in place of **new**, you can define this macro in your source files:

```
#define new DEBUG_NEW
```

When you do an object dump, each object allocated with `DEBUG_NEW` will show the file and line number where it was allocated, allowing you to pinpoint the sources of memory leaks.

The Debug version of the MFC framework uses `DEBUG_NEW` automatically, but your code does not. If you want the benefits of `DEBUG_NEW`, you must use `DEBUG_NEW` explicitly or **#define new** as shown above.

In this topic

Enabling memory diagnostics

Before you can use the memory diagnostics facilities, you must enable diagnostic tracing.

To enable or disable memory diagnostics

- Call the global function AfxEnableMemoryTracking to enable or disable the diagnostic memory allocator. Because memory diagnostics are on by default in the debug library, you will typically use this function to temporarily turn them off, which increases program execution speed and reduces diagnostic output.

To select specific memory diagnostic features with afxMemDF

- If you want more precise control over the memory diagnostic features, you can selectively turn individual memory diagnostic features on and off by setting the value of the MFC global variable afxMemDF. This variable can have the following values as specified by the enumerated type **afxMemDF**.

Value	Description
allocMemDF	Turn on diagnostic memory allocator (default).
delayFreeMemDF	Delay freeing memory when calling `delete` or `free` until program exits. This will cause your program to allocate the maximum possible amount of memory.
checkAlwaysMemDF	Call AfxCheckMemory every time memory is allocated or freed.

These values can be used in combination by performing a logical-OR operation, as shown here:

C++

```
afxMemDF = allocMemDF | delayFreeMemDF | checkAlwaysMemDF;
```

In this topic

Taking memory snapshots

1. Create a CMemoryState object and call the CMemoryState::Checkpoint member function. This creates the first memory snapshot.

2. After your program performs its memory allocation and deallocation operations, create another `CMemoryState` object and call `Checkpoint` for that object. This gets a second snapshot of memory usage.

3. Create a third `CMemoryState` object and call its CMemoryState::Difference member function, supplying as arguments the two previous `CMemoryState` objects. If there is a difference between the two memory states, the `Difference` function returns a nonzero value. This indicates that some memory blocks have not been deallocated.

   This example shows what the code looks like:

```
// Declare the variables needed
#ifdef _DEBUG
    CMemoryState oldMemState, newMemState, diffMemState;
    oldMemState.Checkpoint();
#endif

    // Do your memory allocations and deallocations.
    CString s("This is a frame variable");
    // The next object is a heap object.
   CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );

#ifdef _DEBUG
    newMemState.Checkpoint();
    if( diffMemState.Difference( oldMemState, newMemState ) )
    {
        TRACE( "Memory leaked!\n" );
    }
#endif
```

   Notice that the memory-checking statements are bracketed by `#ifdef`_DEBUG/ **#endif** blocks so that they are compiled only in Debug versions of your program.

   Now that you know a memory leak exists, you can use another member function, CMemoryState::DumpStatistics that will help you locate it.

In this topic

Viewing memory statistics

The CMemoryState::Difference function looks at two memory-state objects and detects any objects not deallocated from the heap between the beginning and end states. After you have taken memory snapshots and compared them using `CMemoryState::Difference`, you can call CMemoryState::DumpStatistics to get information about the objects that have not been deallocated.

Consider the following example:

```
if( diffMemState.Difference( oldMemState, newMemState ) )
{
   TRACE( "Memory leaked!\n" );
   diffMemState.DumpStatistics();
}
```

A sample dump from the example looks like this:

```
0 bytes in 0 Free Blocks
22 bytes in 1 Object Blocks
45 bytes in 4 Non-Object Blocks
Largest number used: 67 bytes
Total allocations: 67 bytes
```

Free blocks are blocks whose deallocation is delayed if `afxMemDF` was set to `delayFreeMemDF`.

Ordinary object blocks, shown on the second line, remain allocated on the heap.

Non-object blocks include arrays and structures allocated with `new`. In this case, four non-object blocks were allocated on the heap but not deallocated.

`Largest number used` gives the maximum memory used by the program at any time.

`Total allocations` gives the total amount of memory used by the program.

In this topic

Taking object dumps

In an MFC program, you can use CMemoryState::DumpAllObjectsSince to dump a description of all objects on the heap that have not been deallocated. `DumpAllObjectsSince` dumps all objects allocated since the last CMemoryState::Checkpoint. If no `Checkpoint` call has taken place, `DumpAllObjectsSince` dumps all objects and nonobjects currently in memory.

Note
Before you can use MFC object dumping, you must enable diagnostic tracing.

Note
MFC automatically dumps all leaked objects when your program exits, so you do not need to create code to dump objects at that point.

The following code tests for a memory leak by comparing two memory states and dumps all objects if a leak is detected.

```
if( diffMemState.Difference( oldMemState, newMemState ) )
{
   TRACE( "Memory leaked!\n" );
   diffMemState.DumpAllObjectsSince();
}
```

The contents of the dump look like this:

```
Dumping objects ->

{5} strcore.cpp(80) : non-object block at $00A7521A, 9 bytes long
{4} strcore.cpp(80) : non-object block at $00A751F8, 5 bytes long
{3} strcore.cpp(80) : non-object block at $00A751D6, 6 bytes long
{2} a CPerson at $51A4

Last Name: Smith
First Name: Alan
Phone #: 581-0215

{1} strcore.cpp(80) : non-object block at $00A7516E, 25 bytes long
```

The numbers in braces at the beginning of most lines specify the order in which the objects were allocated. The most recently allocated object has the highest number and appears at the top of the dump.

To get the maximum amount of information out of an object dump, you can override the `Dump` member function of any `CObject`-derived object to customize the object dump.

You can set a breakpoint on a particular memory allocation by setting the global variable `_afxBreakAlloc` to the number shown in the braces. If you rerun the program the debugger will break execution when that allocation takes place. You can then look at the call stack to see how your program got to that point.

The C run-time library has a similar function, _CrtSetBreakAlloc, that you can use for C run-time allocations.

In this topic

Interpreting memory dumps

Look at this object dump in more detail:

```
{5} strcore.cpp(80) : non-object block at $00A7521A, 9 bytes long
{4} strcore.cpp(80) : non-object block at $00A751F8, 5 bytes long
{3} strcore.cpp(80) : non-object block at $00A751D6, 6 bytes long
{2} a CPerson at $51A4

Last Name: Smith
First Name: Alan
Phone #: 581-0215
```

```
{1} strcore.cpp(80) : non-object block at $00A7516E, 25 bytes long
```

The program that generated this dump had only two explicit allocations—one on the stack and one on the heap:

```
// Do your memory allocations and deallocations.
CString s("This is a frame variable");
// The next object is a heap object.
CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
```

The `CPerson` constructor takes three arguments that are pointers to `char`, which are used to initialize `CString` member variables. In the memory dump, you can see the `CPerson` object along with three nonobject blocks (3, 4, and 5). These hold the characters for the `CString` member variables and will not be deleted when the `CPerson` object destructor is invoked.

Block number 2 is the `CPerson` object itself. `$51A4` represents the address of the block and is followed by the contents of the object, which were output by `CPerson::Dump` when called by DumpAllObjectsSince.

You can guess that block number 1 is associated with the `CString` frame variable because of its sequence number and size, which matches the number of characters in the frame `CString` variable. Variables allocated on the frame are automatically deallocated when the frame goes out of scope.

Frame Variables

In general, you should not worry about heap objects associated with frame variables because they are automatically deallocated when the frame variables go out of scope. To avoid clutter in your memory diagnostic dumps, you should position your calls to `Checkpoint` so that they are outside the scope of frame variables. For example, place scope brackets around the previous allocation code, as shown here:

```
oldMemState.Checkpoint();
{
    // Do your memory allocations and deallocations ...
    CString s("This is a frame variable");
    // The next object is a heap object.
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
}
newMemState.Checkpoint();
```

With the scope brackets in place, the memory dump for this example is as follows:

```
Dumping objects ->

{5} strcore.cpp(80) : non-object block at $00A7521A, 9 bytes long
{4} strcore.cpp(80) : non-object block at $00A751F8, 5 bytes long
{3} strcore.cpp(80) : non-object block at $00A751D6, 6 bytes long
{2} a CPerson at $51A4

Last Name: Smith
First Name: Alan
```

```
Phone #: 581-0215
```

Nonobject Allocations

Notice that some allocations are objects (such as `CPerson`) and some are nonobject allocations. "Nonobject allocations" are allocations for objects not derived from `CObject` or allocations of primitive C types such as `char`, `int`, or **long**. If the **CObject-**derived class allocates additional space, such as for internal buffers, those objects will show both object and nonobject allocations.

Preventing Memory Leaks

Notice in the code above that the memory block associated with the `CString` frame variable has been deallocated automatically and does not show up as a memory leak. The automatic deallocation associated with scoping rules takes care of most memory leaks associated with frame variables.

For objects allocated on the heap, however, you must explicitly delete the object to prevent a memory leak. To clean up the last memory leak in the previous example, delete the `CPerson` object allocated on the heap, as follows:

```
{
    // Do your memory allocations and deallocations.
    CString s("This is a frame variable");
    // The next object is a heap object.
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
    delete p;
}
```

In this topic

Customizing object dumps

When you derive a class from CObject, you can override the `Dump` member function to provide additional information when you use DumpAllObjectsSince to dump objects to the Output window.

The `Dump` function writes a textual representation of the object's member variables to a dump context (CDumpContext). The dump context is similar to an I/O stream. You can use the append operator (**<<**) to send data to a `CDumpContext`.

When you override the `Dump` function, you should first call the base class version of `Dump` to dump the contents of the base class object. Then output a textual description and value for each member variable of your derived class.

The declaration of the `Dump` function looks like this:

```
class CPerson : public CObject
{
public:
#ifdef _DEBUG
    virtual void Dump( CDumpContext& dc ) const;
#endif

    CString m_firstName;
    CString m_lastName;
    // And so on...
};
```

Because object dumping only makes sense when you are debugging your program, the declaration of the `Dump` function is bracketed with an **#ifdef _DEBUG / #endif** block.

In the following example, the `Dump` function first calls the `Dump` function for its base class. It then writes a short description of each member variable along with the member's value to the diagnostic stream.

```
#ifdef _DEBUG
void CPerson::Dump( CDumpContext& dc ) const
{
    // Call the base class function first.
    CObject::Dump( dc );

    // Now do the stuff for our specific class.
    dc << "last name: " << m_lastName << "\n"
        << "first name: " << m_firstName << "\n";
}
#endif
```

You must supply a `CDumpContext` argument to specify where the dump output will go. The Debug version of MFC supplies a predefined `CDumpContext` object named `afxDump` that sends output to the debugger.

```
CPerson* pMyPerson = new CPerson;
// Set some fields of the CPerson object.
//...
// Now dump the contents.
#ifdef _DEBUG
pMyPerson->Dump( afxDump );
#endif
```

In this topic

Reducing the size of an MFC Debug build

The debug information for a large MFC application can take up a lot of disk space. You can use one of these procedures to reduce the size:

1. Rebuild the MFC libraries using the /Z7, /Zi, /ZI (Debug Information Format) option, instead of **/Z7**. These options build a single program database (PDB) file that contains debug information for the entire library, reducing redundancy and saving space.

2. Rebuild the MFC libraries without debug information (no /Z7, /Zi, /ZI (Debug Information Format) option). In this case, the lack of debug information will prevent you from using most debugger facilities within the MFC library code, but because the MFC libraries are already thoroughly debugged, this may not be a problem.

3. Build your own application with debug information for selected modules only as described below.

In this topic

Building an MFC app with debug information for selected modules

Building selected modules with the MFC debug libraries enables you to use stepping and the other debug facilities in those modules. This procedure makes use of both the Debug and Release modes of the Visual C++ makefile, thus necessitating the changes described in the following steps (and also making a "rebuild all" necessary when a full Release build is required).

1. In Solution Explorer, select the project.
2. From the **View** menu, select **Property Pages**.
3. First, you will create a new project configuration.
    a. In the **<Project> Property Pages** dialog box, click the **Configuration Manager** button.
    b. In the Configuration Manager dialog box, locate your project in the grid. In the **Configuration** column, select **<New...>**.
    c. In the New Project Configuration dialog box, type a name for your new configuration, such as "Partial Debug", in the **Project Configuration Name** box.
    d. In the **Copy Settings from** list, choose **Release**.
    e. Click **OK** to close the **New Project Configuration**dialog box.
    f. Close the **Configuration Manager** dialog box.
4. Now, you will set options for the entire project.
    a. In the **Property Pages** dialog box, under the **Configuration Properties** folder, select the **General** category.
    b. In the project settings grid, expand **Project Defaults** (if necessary).
    c. Under **Project Defaults**, find **Use of MFC**. The current setting appears in the right column of the grid. Click on the current setting and change it to **Use MFC in a Static Library**.
    d. In the left pane of the **Properties Pages** dialog box, open the **C/C++** folder and select **Preprocessor**. In the properties grid, find **Preprocessor Definitions** and replace "NDEBUG" with "_DEBUG".
    e. In the left pane of the **Properties Pages** dialog box, open the **Linker** folder and select the **Input** Category. In the properties grid, find **Additional Dependencies**. In the **Additional Dependencies** setting, type "NAFXCWD.LIB" and "LIBCMT."
    f. Click **OK** to save the new build options and close the **Property Pages** dialog box.
5. From the **Build** menu, select **Rebuild**. This removes all debug information from your modules but does not affect the MFC library.
6. Now you must add debug information back to selected modules in your application. Remember that you can set breakpoints and perform other debugger functions only in modules you have compiled with debug information. For each project file in which you want to include debug information, carry out the following steps:
    a. In Solution Explorer, open the **Source Files** folder located under your project.
    b. Select the file you want to set debug information for.
    c. From the **View** menu, select **Property Pages**.
    d. In the **Property Pages** dialog box, under the **Configuration Settings** folder, open the **C/C++** folder then select the **General** category.
    e. In the properties grid, find **Debug Information Format.**

f. Click the **Debug Information Format** settings and select the desired option (usually **/ZI**) for debug information.

g. If you are using an application wizard-generated application or have precompiled headers, you have to turn off the precompiled headers or recompile them before compiling the other modules. Otherwise, you will receive warning C4650 and error message C2855. You can turn off precompiled headers by changing the **Create/Use Precompiled Headers** setting in the **<Project> Properties** dialog box (**Configuration Properties** folder, **C/C++** subfolder, **Precompiled Headers** category).

7. From the **Build** menu, select **Build** to rebuild project files that are out of date.

As an alternative to the technique described in this topic, you can use an external makefile to define individual options for each file. In that case, to link with the MFC debug libraries, you must define the _DEBUG flag for each module. If you want to use MFC release libraries, you must define NDEBUG. For more information on writing external makefiles, see the NMAKE Reference.

In this topic

See Also

Debugging Visual C++