

# Debugging Heap Corruption in Visual C++ Using Microsoft Debugging Tools for Windows

Version 10.0

David Dahlbacka

## Contents

Heap Corruption.....	2
Causes of Heap Corruption.....	2
Debugging Heap Corruption.....	2
Debugging Tools.....	3
Specific WinDbg Commands .....	3
Specific GFlags Commands.....	4
Preparation for Debugging.....	5
Program Installation and Compilation .....	5
First-Time WinDbg Options .....	5
Live Debugging .....	6
Standard Heap Options .....	6
Full or DLLs Heap Options .....	7
Postmortem Debugging .....	8
Analyzing a Memory Dump .....	9
References.....	9
Example Program.....	10
Example Program Code .....	10

As an experienced programmer, you may have already faced one of the hardest parts of your job: fixing a bug, such as an access violation, caused by corruption in program-allocated heap memory. Such bugs can be very difficult and frustrating to diagnose, because every change you make to the program also changes heap memory – including adding debug print code, commenting out code, running a debugger, and changing the input data. Thus, anything you do to investigate the problem may cause the symptom to disappear or move to another point in the program execution.

## Heap Corruption

Heap corruption is an undesired change in the data allocated by your program. Its symptoms include:

- System errors, such as access violations.
- Unexpected data in program output.
- Unexpected paths of program execution.

Your program may show a symptom of heap corruption immediately or may delay it indefinitely, depending on the execution path through the program.

### ***Causes of Heap Corruption***

Your program can cause heap corruption in several ways, including:

- Using too large a value to index an array. If the adjacent memory is allocated to another object, the program will overwrite that object's data.
- Casting a pointer to a data type larger than the original allocation's data type. If the adjacent memory is allocated to another object, the program will overwrite that object's data when it accesses a data field outside the original allocation.
- Deleting an object prematurely while retaining a pointer to it. When the operating system allocates the memory for new data, the program will overwrite the old data with the new data.
- (COM only) Releasing all pointers to a particular COM interface in a subprogram while retaining a COM pointer to that interface in the calling subprogram. When your program releases the last COM pointer in the subprogram, the operating system will mark that interface and all its methods and data items as not valid. If the calling subprogram then uses its own COM pointer, the system will generate an access violation.

### ***Debugging Heap Corruption***

To debug heap corruption, you must identify both the code that allocated the memory involved and the code that deleted, released, or overwrote it. If the symptom appears immediately, you can often diagnose the problem by examining code near where the error occurred. Often, however, the symptom is delayed, sometimes for hours. In such cases, you must force a symptom to appear at a time and place where you can derive useful information from it.

A common way to do this is for you to command the operating system to insert a special suffix pattern into a small segment of extra memory and check that pattern when the memory is deleted. Another way is for the operating system to allocate extra memory after each allocation and mark it as Protected, which would cause the system to generate an access violation when it was accessed.

## Debugging Tools

To perform this procedure, you will use the Debugging Tools for Windows, available for free download from the Microsoft WKD and Developer Tools web pages. To find the download site, search the Web for a string similar to "Install Debugging Tools for Windows 32-bit Version" and download and install the following version or later, taking all defaults: "Current Release version 6.6.7.5 - July 18, 2006".

From the Debugging Tools for Windows, you will use the following programs:

- *WinDbg.exe*: A heap-level debugging program similar to the Visual C++ debugger. Using WinDbg, you can set breakpoints in source and assembly code, display the values of variables, dump heap memory into a file, and analyze such a dump offline.
- *GFlags.exe*: A heap debug program. Using GFlags, you can establish standard, /full, or /dlls heap options that will force the operating system to generate access violations and corruption errors when your program overwrites heap memory.

### Specific WinDbg Commands

You will use the following specific WinDbg commands and menu items, in addition to the familiar debugger commands of Go, Step, Watch, and so on:

Command	Meaning
File, then Open Executable	Brings up your program inside the WinDbg debugger. This allows you to debug the initialization phases of your program's execution. However, the program's memory usage is somewhat different from that of ordinary execution.
File, then Attach to a Process	Links the WinDbg debugger to your program after it has started running. You cannot debug the initialization phases of your program's execution. However, the program's memory usage is more similar to that of ordinary execution than if you had brought up the program inside the debugger.
File, then Symbol File Path	Directs the WinDbg debugger to the directories in which the VC++ compiler has placed the debug (.pdb) files that your program needs to display symbols, such as subprogram names.
File, then Source File Path	Directs the WinDbg debugger to the directories in which the source files for your program reside.
.symfix+	Turns on access to the Microsoft online symbol server. The resulting path appears in the Symbol File

Command	Meaning
	Path. You must be connected to the Web for this command to work.
<code>.dump /ma <i>Filename.dmp</i></code>	Dumps heap memory in mini-dump format. The <code>/ma</code> option means it is a mini-dump ( <code>/m...</code> ) containing all (...a) information needed to reconstruct the state of the program when the dump occurred.
File, then Open Crash Dump	Directs the WinDbg debugger to a dump file produced by the <code>.dump</code> command.
<code>!analyze -v</code>	Analyzes the contents of a dump file produced by the <code>.dump</code> command. The <code>-v</code> stands for "verbose" information.

### Specific GFlags Commands

You will use the following specific GFlags commands:

Command	Meaning
<code>GFlags /p</code>	Displays the state of the page heap options.
<code>GFlags /p /enable <i>Program.exe</i></code>	Enables standard heap options for <i>Program.exe</i> . GFlags inserts a suffix pattern into the heap after each allocation by your program. Your program will use somewhat more memory and run somewhat slower than normal.
<code>GFlags /p /enable <i>Program.exe</i> /full</code>	Enables full heap options for <i>Program.exe</i> . GFlags inserts an entire page of protected memory into the heap after each allocation by your program. Your program will use much more memory and run much slower than normal.
<code>GFlags /p /enable <i>Program.exe</i> /dlls <i>Library1.dll,Library2.dll,...</i></code>	Enables standard heap options for <i>Program.exe</i> and all dynamic link libraries (DLLs) not listed after the <code>/dlls</code> option. In addition, it enables full heap options for the listed DLL or DLLs. Separate items in the DLL list by commas. Your program will run faster and use less memory than it would if you had used the <code>/full</code> option.
<code>GFlags /p /enable <i>Program.exe</i></code>	Enables standard heap options for <i>Program.exe</i> . In

Command	Meaning
<code>/debug <i>Debugger.exe</i></code>	In addition, it designates <i>Debugger.exe</i> as the debugger to automatically run if the program crashes. The <code>/debug</code> option comes after <code>/full</code> or <code>/dlls</code> in the command line.
<code>GFlags /p /disable <i>Program.exe</i></code>	Disables all heap options for <i>Program.exe</i> . You should do this at the end of each debug session. Otherwise, every user of the program after you will experience slow performance.
<code>GFlags</code>	Invokes a dialog box that allows you to review and change the current GFlags settings.

## Preparation for Debugging

You must prepare the debugging environment before you run the program. In particular, you must install the debugging tools, compile your program with the correct options, and configure the debugging tools to run with your program.

### ***Program Installation and Compilation***

1. Install Microsoft Debugging Tools for Windows as indicated in Debugging Tools.
2. Add the path to the Debugging Tools for Windows directory to your system path via ControlPanel, then System, then Advanced, then Environment Variables. This will allow you to run Debugging Tools programs from a command line.
3. Compile your program and any DLLs it uses with the following compiler options:

```
/Od /Oi
```

These options turn off compiler optimization and store debug symbols in .pdb files in the same directory as the .exe file, usually the Debug directory.

### ***First-Time WinDbg Options***

You should execute the following one-time tasks the first time you run your program with WinDbg:

1. Start *WinDbg.exe*.
2. If you can attach WinDbg to your program after it has started, start your program and attach WinDbg to the process by clicking on File, then Attach to a Process and following the prompts. Otherwise, run your program from WinDbg using File, then Open Executable likewise.

3. In the command line at the bottom of the Command window, type the following command.

```
.symfix+
```

This will connect this and future WinDbg sessions associated with this program to the Microsoft symbol server

4. Click on File, then Symbol File Path and browse to the directory where your program's .exe file resides. Click again and browse to the directory where any DLLs you are using reside.
5. Click on File, then Source File Path and browse to the directory where your program's source files reside. Click again and browse to the directory where the source files for any DLLs you are using reside.
6. Exit WinDbg, clicking Yes to store the WinDbg options associated with this program.

You will set other options from the command line before you run the program and from the WinDbg interface after you have attached to the program.

## Live Debugging

A live debugging session has three main parts, executed iteratively as you debug:

1. Setting GFlags options.
2. Running the program to a breakpoint.
3. Analyzing the program state at the breakpoint.

As a rule, you should plan at least two iterations, one with standard heap options and one with either the /full or the /dlls options. The example program in the appendix is small enough to allow you to use the /full option. Normally you will use the /dlls option.

Note that you must set up all GFlags options before you run your program. When the system starts your program, it uses whatever GFlags options were set at that time and does not change them until the run is finished.

### **Standard Heap Options**

A session with standard heap options would go as follows:

1. Set standard heap options.

```
GFlags /p /enable Program.exe
```

2. Invoke *WinDbg.exe*.
3. Run *Program.exe* from WinDbg using File, then Open Executable. If this were a program that brought up a command window or a dialog box that waited for user

input, you could also run it from the command line or via Windows, and then attach WinDbg to it by using File, then Attach to a Process.

4. Click View, then Call Stack to display the call stack.
5. Click WinDbg, then Go from the toolbar.
6. Wait for an error to occur. The standard heap options cause the system to place a suffix pattern after each allocation.

When your program attempts to delete the allocation, the system will check the suffix pattern, and if it has been overwritten, the system will raise an error. The symptom might be an error popup, an Application Verify Stop, an Access Violation, a Memory Check Error, or any other error type severe enough to force the operation system to halt execution.

You may be able to find out why the memory is being corrupted by examining the error message, the contents of the registers and memory at various levels in the call stack, and the corresponding source and assembly-language code.

7. Click Stop Debugging in the WinDbg toolbar. This will halt the program and empty the debug window.
8. Clear the heap options. If you do not do this, every user of the program after you will experience slow performance.

```
GFlags /p /disable Program.exe
```

### **Full or DLLs Heap Options**

From the stack trace, you've discovered where the object was deleted, though not necessarily what allocated the memory or what corrupted it. You have developed some notion of where it might have been created and how it was processed. You can use this information to guide your search for a solution.

1. Set /full or /dlls heap options. For a very small program, you can use /full heap options, as follows:

```
GFlags /p /enable Program.exe /full
```

For more complex code, you would use /dlls options, as follows:

```
GFlags /p /enable Program.exe /dlls Library1.dll,Library2.dll,...
```

2. Invoke *WinDbg.exe*.
3. Run *Program.exe* from WinDbg using File, then Load Executable.
4. Click View, then Call Stack to display the call stack.
5. Click WinDbg, then Go from the toolbar.

6. Wait for an error to occur. The `/full` and `/dlls` heap options cause the system to place a full page of protected memory after each allocation from the designated scope or scopes. For the `/dlls` options, the system places a suffix pattern after each allocation from the remaining scopes.

When your program attempts to write into the protected memory, the system will raise an error. The symptom might be an error popup, an Application Verify Stop, an Access Violation, a Memory Check Error, or any other error type severe enough to force the operation system to halt execution. Note that the system will raise errors with `/full` or `/dlls` heap options that it will not raise with standard heap options.

You may be able to find out why the memory is being corrupted by examining the error message, the contents of the registers and memory at various levels in the call stack, and the corresponding source and assembly-language code.

7. Click Stop Debugging in the WinDbg toolbar. This will halt the program and empty the debug window.
8. Clear the heap options:

```
GFlags /p /disable Program.exe
```

## Postmortem Debugging

You prepare and execute postmortem debugging in much the same way you prepare for live debugging, including settings for GFlags. The main difference is that you do not run the program in the debugger or attach the debugger to the program after it starts. Instead, you run the program as intended, but use GFlags to designate a debugger that the system will attach to the program if there is an error. The advantage of this procedure is that you can minimize the effect upon program execution of your preparations for debugging.

1. Set standard heap options with a designated debugger:

```
GFlags /p /enable Program.exe /debug WinDbg.exe
```

2. Run your program.
3. Wait for an error to occur. The error might be an Application Verify Stop, an Access Violation, a Memory Check Error, or any other error type severe enough to force the operation system to attach the debugger to the process and bring it up at a breakpoint. Then you can either analyze the error online as in any other debug session, or evaluate the error offline.

To evaluate the error offline, you can pull a memory dump by typing the following into the WinDbg command line:



```
.dump /ma Filename.dmp
```

4. Click Stop Debugging in the WinDbg toolbar. This will halt the program and empty the debug window.
5. Clear the heap options:

```
GFlags /p /disable Program.exe
```

## Analyzing a Memory Dump

If you have pulled a memory dump using the .dump command, you can load and analyze it as follows:

1. Invoke *WinDbg.exe* and set up symbol paths and source file paths as above.
2. Load the memory dump file by File, then Open Crash Dump.
3. Analyze the memory dump by typing the following command into the command window:

```
!analyze -v
```

The !analyze command will produce a verbose stack trace and an error exception, much like those that WinDbg produces when you are debugging online. The main difference is that you cannot run the program and sometimes lack complete access to runtime variables.

## References

The key reference material for the Microsoft Debugging Tools for Windows is the online help, accessible from the WinDbg Help menu. See in particular the following:

- Example 12: Using Page Heap Verification to Find a Bug
- GFlags Commands
- Specific Exceptions
- Create Dump File

These tools and the Debugging Tools for Windows themselves implement many other useful features. To find out more about them, consult Debugging Help from Start, then Debugging Tools for Windows.

The Microsoft web site contains a great deal of useful information about debugging Windows programs. Search on "Debugging Tools for Windows."

You may also wish to investigate commercially-available tools, such as Purify and BoundsChecker. For more information on these tools, Google on "Rational Purify" and "

DevPartner Studio". For more on heap debugging in general, Google on "heap debugging".

## Example Program

As an example, you may set up the following program, which contains an out-of-range array error, an oversized structure cast error, and a premature delete error.

1. Create a Win32 Console Application project named ConsoleExe. (Do not create a Win32 Application. ConsoleExe uses standard I/O, which requires a DOS window.)
2. Copy the code at the end of this appendix into ConsoleExe.cpp in place of the existing main program.
3. Compile and link the ConsoleExe application with the compiler options /Od /Oi.
4. At this point, you have three alternatives: (a) Run it from the command line and attach WinDbg to it as it waits for input; (b) start it from WinDbg; or (c) use the /debug *WinDbg.exe* option to bring up the debugger on an error.

You can then observe how the various heap debug options affect the behavior of the program and that of the debugger.

### Example Program Code

```
// ConsoleEXE.cpp : Defines the entry point for the console
// application.

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>

const int ciSize = 5;

typedef struct SmallStruct
{
    short shShort1;
    short shShort2;
} SmallStruct;

typedef struct BigStruct
{
    long lLong1;
    long lLong2;
} BigStruct;

void ArrayOverrunCrash(void);
void OversizedCastCrash(void);
void PrematureDeleteCrash(void);
```

```
int main(int argc, char* argv[])
{
    bool bExit = false;
    char sInput[ciSize];

    //=====

    printf("Welcome to Crash Test Dummy.\n");

    do
    {
        printf("\nEnter 'a' for array overrun.\n");
        printf("Enter 'c' for oversized cast.\n");
        printf("Enter 'd' for premature delete.\n");
        printf("Enter 'x' to exit.>>");
        scanf("%s", sInput);

        switch (sInput[0])
        {
            case 'a':
            case 'A':
                ArrayOverrunCrash();
                break;

            case 'c':
            case 'C':
                OversizedCastCrash();
                break;

            case 'd':
            case 'D':
                PrematureDeleteCrash();
                break;

            case 'x':
            case 'X':
                bExit = true;
                break;

            default:;
                // Do nothing
        }
    }
    while (! bExit);

    return 0;
}

void ArrayOverrunCrash(void)
{
    char *sASCII = new char[ciSize];
    for (int i = 0; i <= ciSize; i++)
    {
        // sASCII[ciSize] is outside array.
        sASCII[i] = (char)rand();
    }
    delete [] sASCII;
}
```

```
void OversizedCastCrash(void)
{
    SmallStruct *pSmallStruct = new SmallStruct;

    pSmallStruct->shShort1 = 1;
    pSmallStruct->shShort2 = 2;

    BigStruct *pBigStruct =(BigStruct *)pSmallStruct;
    pBigStruct->lLong1 = 1;

    // lLong2 is outside pSmallStruct memory.
    pBigStruct->lLong2 = 2;

    delete pSmallStruct;
}

void PrematureDeleteCrash(void)
{
    SmallStruct *pSmallStruct = new SmallStruct;
    pSmallStruct->shShort1 = 1;
    delete pSmallStruct;

    // pSmallStruct points to deleted memory.
    pSmallStruct->shShort2 = 2;
}
```