Debugging in Visual Studio

Learn how to use the powerful integrated debugging environment provided in Visual Studio 2003 and 2005

By Steve Jones
Game Institute
What we will cover

- Debugging native, 32-bit console and win32 applications
- Learn most common debugging tools

- Visual Studio .NET 2003 and 2005 IDE environments
- The most common techniques for debugging
- Examples
Debugging
What are you trying to find and fix?

- Two main types of code errors
  - Syntax
    - Compiler catches most if not all of these for you.
  - Semantic or logical
    - Syntactically correct yet program may “crash and burn” at run-time!

For example: Compiler will not catch an un-initialized pointer but you WILL get a run-time error if you try to use it!
Why Should I Use Visual Studio to Debug my Program?

• Even most experienced coder creates errors or “bugs”
• Visual Studio debugger will provide two powerful run-time facilities:
  – Trace the program **Execution**
  – **Watch** variables during program execution
• These allow you to stop at procedure locations, inspect memory and register values, change variables, observe message traffic, and get a close look at what your code does.
Project Configuration Settings

• Debug vs. Release Configurations
  – The **Debug** configuration of your program is compiled with full symbolic debug information and no optimization.
  – The **Release** configuration of your program is fully optimized and contains no symbolic debug information.
  – Must be in Debug configuration to debug your program.
Getting Acquainted with Visual Studio Debugger

- Debugger Windows
  - Autos
  - Locals
  - Watch
  - Call Stack
  - Command Window
  - QuickWatch Dialog
  - Breakpoints window
  - Threads
  - Modules
  - Processes
  - Memory
  - Disassembly
  - Registers

- Execution Control
  - Starting or Continuing Execution
  - Stopping
  - Breaking Execution
  - Stepping Into and Out of code
  - Jumping to another location
This simple console program should determine whether two integers are equal.

Code compiled just fine, 0 warnings, 0 errors

... BUT the code obviously has a logical error! 3 does not equal 5!
```cpp
#include <iostream>

int main()
{
    int x, y;
    cout << "Enter first integer: ";
    cin >> x;

    cout << "Enter second integer: ";
    cin >> y;

    if (x == y)
        cout << "They are Equal!" << endl;
    else if (x > y)
        cout << "The first one is bigger!" << endl;
    else
        cout << "The second one is bigger!" << endl;

    cout << endl;
    system("pause");
    return 0;
}
```
What is a Breakpoint?

- Breakpoints are user-defined code locations that pause execution.
- You know them by the little, red “dot” in the left margin of the editor window.
- F9 to add or remove (toggle).
- Or left-mouse click in margin.
- Unlimited number of them to use.
Add a breakpoint here
We arbitrarily picked this line because it seemed like a reasonable place
Starting the Debugging Session

- Make sure you are in a Debug configuration
- Press F5
- Or click on Debug icon
- Or select menu Debug – Start Debugging
Debugging Example #1 - Running in the debugger
Debugging Example #1

Stepping, examine variables
Execution Control
Stepping through your code

- Starting / Stopping
- Breaking
- Stepping through your application
  - (F10, F11 or Toolbar buttons)
- Run to a specific location
  - Run To Cursor (right-click menu)

F11 Step Into
F10 Step Over
Shift + F11 Step Out
Equivalent hot-keys
Autos Window

• **Name**
  - The names of all variables in the current statement and the previous statement. The current statement is the statement at the current execution location, which is the statement that will be executed next if execution continues.

• **Value**
  - The value contained by each variable. By default, integer variables are represented in decimal form.

• **Type**
  - The data type of each variable listed in the **Name** column.
Locals Window

- **Name**
  - This column contains the names of all local variables in the current scope.

- **Value**
  - The value contained by each variable. By default, integer variables are represented in decimal form.

- **Type**
  - The data type of each variable listed in the **Name** column.
Watch window(s)

- Watch window displays Name, Value, and Type of variables
- Type in or click-drag variables into window
- Change values live while at break
- You have 4 independent Watch windows

(VS 2005 & VC++ Express)
Debugging Example #1 - Found error

```cpp
// DebugConsoleApp.cpp
//
#include <iostream>
#include <cchar.h>
using namespace std;

//---------
// Main Entry Point
//---------

int main()
{
    int x, y;
    cout << "Enter first integer: ";
    cin >> x;
    cout << "Enter second integer: ";
    cin >> y;

    if (x - y)
        cout << "They are equal!" << endl;
    else if (x > y)
        cout << "The first one is bigger!" << endl;
    else
        cout << "The second one is bigger!" << endl;

    return 0;
}
```
Debugging Example #1 - Fixed error, recompiled, run, step
Debugging Example #1

Step. Hey the code worked!
The Call Stack

- Call stack window displays each function name in the order they were called.
- **Yellow arrow** identifies the stack frame where the execution pointer is located.
- Double-clicking on a function name takes you to the function in source code.
- Click Debug – Windows – Call Stack to show window (if hidden). It is shown by default.
Example #2
How to Use Conditional Breakpoints

These are breakpoints that only “break” based on a specific condition.
In this example, we will put a conditional breakpoint in the “for” loop and the breakpoint will only stop when our condition is met.
Example #2
Add a breakpoint with a condition

Let’s say you want to break execution only when a condition is met rather than break each time the loop cycles.

1. Add a breakpoint on the line you’re interested in.

Then we’ll configure a condition to it.
Example #2
Breakpoint Properties (VS 2003)

2. Right-mouse click on the breakpoint
3. Select **Breakpoint Properties**…
Example #2
Open Breakpoint Properties Dialog (VS 2003)

4. The breakpoint dialog will open.
Example #2
Set hit count condition (VS 2003)

5. Click on **Hit Count**… button
Select frequency of the break
Default is “break always”
Example #2
Set a condition (VS 2003)

6. Click on **Condition...** to set the criteria for breaking.
Example #2
Set hit count condition (VS 2005 & VC++ Express 2005)

- Right-mouse click on the breakpoint
- Select **Hit Count…**
Example #2
Set a condition (VS 2005 & V C++ Express 2005)

- Right-mouse click on the breakpoint
- Select **Condition**...
Example #2
Result - Code breaks at the desired condition
Memory Leaks!
How do you know you have them?

• Basic project setup to detect them
• We will use the C Run-Time library
• After building and running the program, the output window will display any memory leaks.
• We can call another function to force a breakpoint when the suspect memory is allocated.
Memory Leaks!
Using some C Run-Time Functions

_CrtDumpMemoryLeaks()
Performs leak checking where called. You want to place this call at all possible exits of your app.

_CrtSetDbgFlag ()
Sets debugging flags for the C run-time library.

<table>
<thead>
<tr>
<th>_CrtSetDbgFlag () flag</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CRTDBG_REPORT_FLAG</td>
<td>Gets current flag(s)</td>
</tr>
<tr>
<td>_CRTDBG_LEAK_CHECK_DF</td>
<td>Perform automatic leak checking at program exit through a call to _CrtDumpMemoryLeaks</td>
</tr>
</tbody>
</table>

Microsoft Visual Studio is a registered trademark of Microsoft Corp.
“Hook” into the C Run-time libraries to use the debug heap

1. Include the following lines in your program as the basics.

```cpp
#include <iostream>
#include <tchar.h>
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtDBG.h>

int main(int argc, _TCHAR* argv[])
{
    int *pMyVar;

    int n_dbgFlags = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG)
    n_dbgFlags |= _CRTDBG_LEAK_CHECK_DF;
    _CrtSetDbgFlag(n_dbgFlags);

    // Allocate new memory for an integer
    pMyVar = new int;

    // Notice we did not delete the memory!
    return 0;
}
```
Memory Leaks

_CRTDBG_MAP_ALLOC_

- Including crtdbg.h, you map the malloc and free functions to their Debug versions, _malloc_dbg and _free_dbg, which keep track of memory allocation and deallocation.
- Without _define _CRTDBG_MAP_ALLOC:
  - Memory allocation number (inside curly braces)
  - Block type (normal, client or CRT)
  - Memory location in hex
  - Size of block in bytes
  - Contents of the first 16 bytes in hex
- With it defined you get all the above plus:
  - File name
  - Line number
Memory Leaks
Output window dump

Source file where leak occurred
Line number within source file
Memory allocation number
Block type
Memory location
Block size
Memory Leaks
Locating the memory leak

_detected memory leaks!
Dumping objects ->
c:\program files\microsoft visual studio .net 2003\vc7\include\crtDBG.h(689) : [46] normal block at 0x00323828, 4 bytes long.
   Data: <   > CD CD CD CD
Object dump complete.
The program '[2296] MemoryLeak.exe: Native' has exited with code 0 (0x0).

_CrtSetBreakAlloc(<allocation number>)
  • Sets a breakpoint on a specified object allocation order number (debug version only).
Memory Leaks
Locating the memory leak

Drill down through Call Stack window to find the last called function that belongs to your application. (not a function from a library)
So what have we talked about. . .

• You will spend your time finding semantic errors because the compiler catches syntax errors.
• Visual Studio has a rich suite of debugging tools to help you Trace the execution and Watch variables.
• Control program execution by stopping and stepping through your code.
• Watch variable values to see if they look right.
• Use the C Run-time library for finding memory leaks.
Debugging in Visual Studio

Any questions?

You can contact me in the forums at the Game Institute or you can email me at:

smjones@gameinstitute.com
info@gameengineer.net