DEBUG WITH VISUAL STUDIO

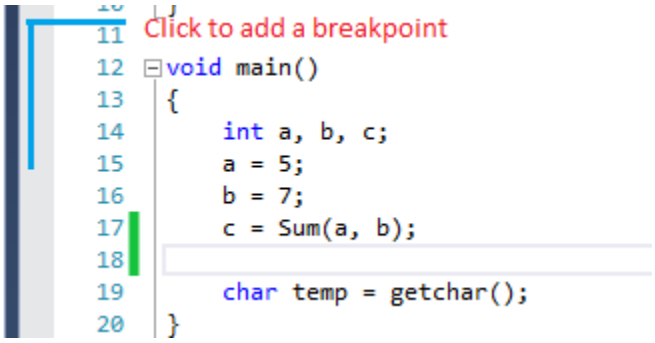
Version of Visual Studio: VS2013 for Desktop

1. Using Breakpoint

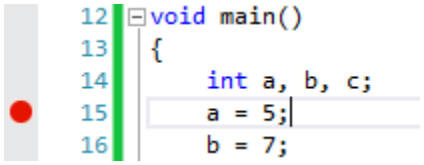
A Breakpoint is a location in executable code at which the operating system stops execution and breaks into the debugger. This allows you to analyze the target and issue debugger commands.

Add

- Move the cursor at the line you want to add a Breakpoint.
- Press F9 or left-mouse click the gutter next to the line where you want to add a breakpoint.



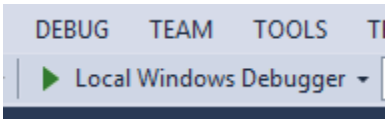
- When you see a red-button, a Breakpoint is created.



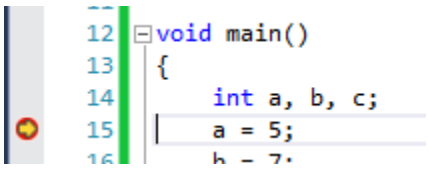
How to work with breakpoint?

When you run the above example by:

- **Pressing F5** or Press into the blue arrow

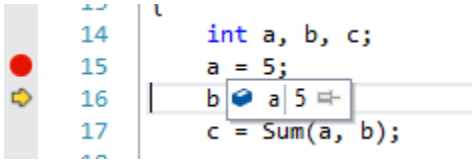


The system will stop the execution and breaks into VS debugger.

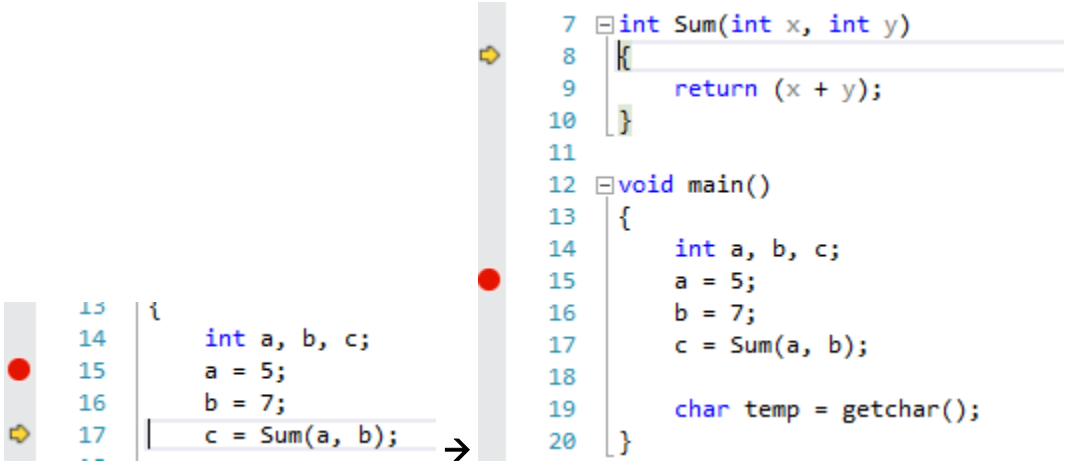


- **Step over (F10)**: Execute the next line of code but not follow execution through any function calls.

By mouse hover to the variables, you can see their values if the code line is already executed. On below example, when the execute cursor moves to line 16, you can see value of variable a on line 15.



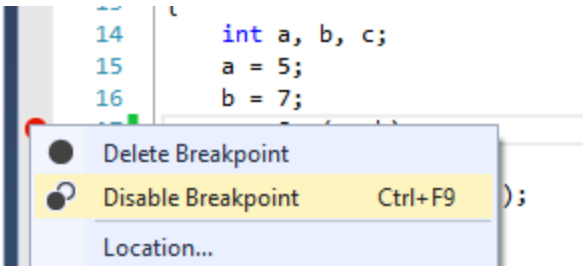
- **Step into (F11)**: Execute code one statement at a time, following execution into function calls. On below example, you will go in Sum method if you press F11 at line 17.



- **Step out: (Shift-F11):** Execute the remaining lines of a function in which the current execution point lies. On above example, the cursor will back position at line 17.
- **Ctrl+F10 (Run To Cursor)**: Instead of using F10, F11 continuously, you choose exactly the code line which you want to be in. Then, you move the cursor to that line and press Ctrl-F10 or right click then choose Run To Cursor.

Disable breakpoint

You can disable breakpoints without deleting them by Ctrl-F9 or choose Disable-breakpoint function

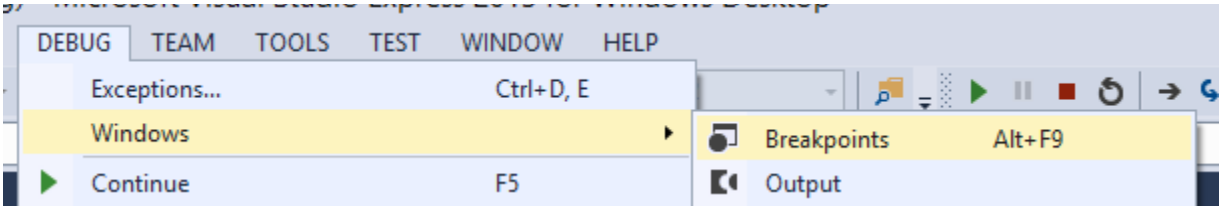


Remove

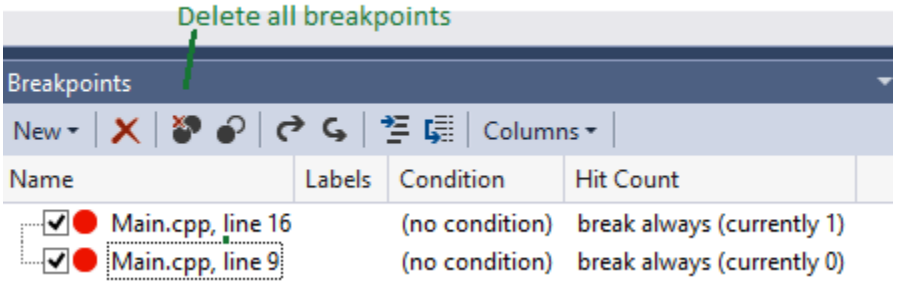
If you want to remove a Breakpoint, press F9 again or left-mouse click on the existing breakpoint.

Remove all breakpoints

Display breakpoint window by Debug ➔Windows➔Breakpoints



Then chose “delete all breakpoints” or Ctr-Shift-F9. Besides, you also see disable or delete any breakpoint, ect. in this window



Backward

To go back to a break point that you have passed, you can drag the yellow arrow showing your current line you're debugging to a previous line. With this way, you can come back to a previous state of the program without restarting the debug.

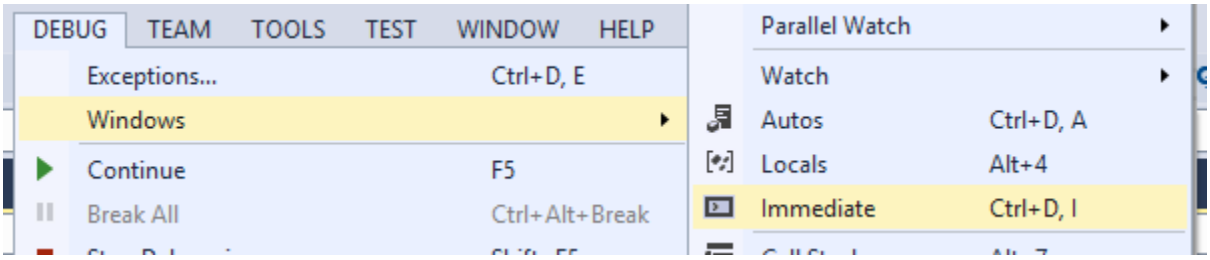
Summary key short-cut with breakpoint:

- F9: create a breakpoint
- Ctrl-F9: disable a breakpoint.
- Ctr-Shift-F9: delete all breakpoints.
- F5 (Continue): run the program or go to next breakpoint.
- Shift-F5 (Stop debugging): Stop debugging, stop running the application.
- Ctrl-F5: Run the application without debugging.
- F10 (Step Over): Execute the next line of code without following execution through any function call.
- Ctrl-F10 (Run to Cursor): go to the chosen cursor.
- F11 (Step Into): Execute code one statement at a time, following execution into function calls.
- Shift-F11 (Step out): Execute the remaining lines of a function in which the current execution point lies.

2. Using Immediate Window and command window

The Immediate window is used to debug and evaluate expressions, execute statements, print variable values, and so forth. It allows you to enter expressions to be evaluated or executed by the development language during debugging.

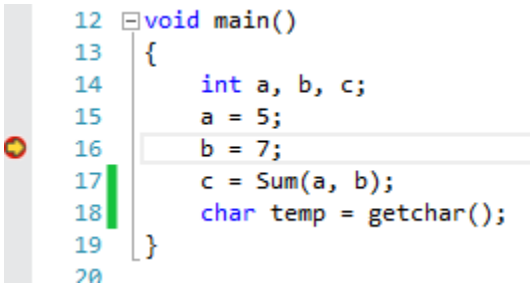
To display Immediate Window, choose Debug →Windows →Immediate or press Ctrl-Alt-I



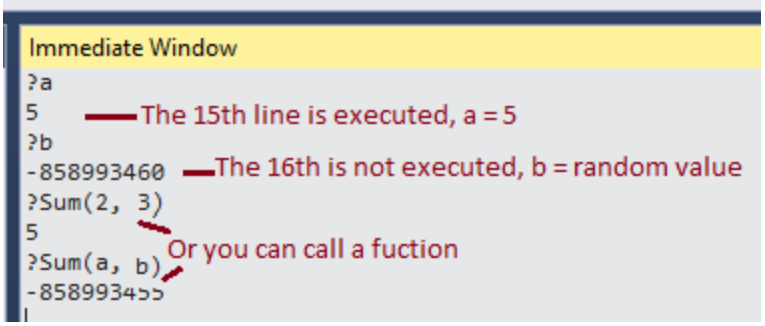
Here is an example with Immediate Window:

```
int Sum(int x, int y)
{
	return (x + y);
}

void main()
{
	int a, b, c;
	a = 5;
	b = 7;
	c = Sum(a, b);
	char temp = getchar();
}
```

Set a breakpoint any where to start debugging mode.



Then call commands to retrieve values.



Common commands:

Task	Solution	Example
Evaluate an expression.	Preface the expression with a question mark (`?`).	`? myvar` `? Function(x)`
Evaluate an expression.	With a greater than symbol (>)	`>Debug.Print varA`
Switch to an Immediate window.	Enter `immed` into the window without the greater than sign (>)	`immed`
Switch back to the Command window from an Immediate window.	Enter `cmd` into the window.	`>cmd`

The table below contains a list of the pre-defined aliases

Command Name	Alias	Complete Name
Print	?	Debug.Print
Quick Watch	??	Debug.Quickwatch
Add New Project	AddProj	File.AddNewProject
Alias	Alias	Tools.Alias
Autos window	Autos	Debug.Autos
Breakpoints window	bl	Debug.Breakpoints
Toggle Breakpoint	bp	Debug.ToggleBreakPoint
Call Stack window	CallStack	Debug.CallStack

Clear Bookmarks	ClearBook	Edit.ClearBookmarks
Close	Close	File.Close

Read more at http://dotnetdud.blogspot.com/2007/12/visual-studio-immediate-window.html#cQOfqRQHUO4jCSm5.99

3. Adding a data breakpoint

Data breakpoints break execution when a value that is stored at a specified memory location is written. If the value is read but not written, execution does not break.

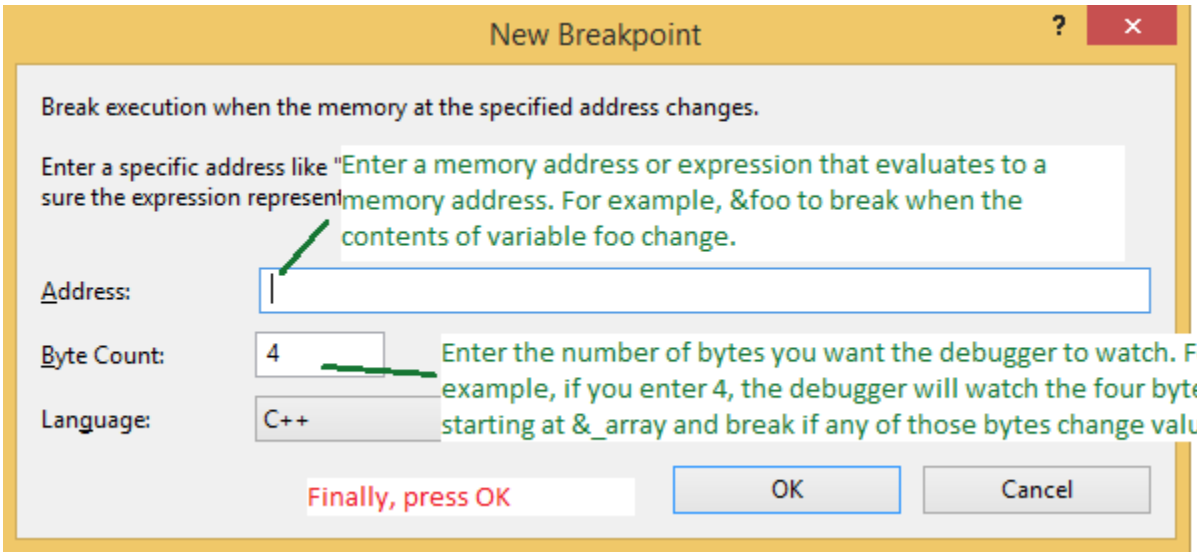
You can set data breakpoints in break mode only.

To set a Memory Change Breakpoint:

- Set any breakpoint and run program to launch debug mode
- From the **Debug Menu** ➔ **New Breakpoint** ➔ **New Data Breakpoint**

  or

  In the **Breakpoints window** Menu, click the **New** dropdown and choose **New Data Breakpoint**.
- The **New Breakpoint** dialog box appears.



4. Debug with watch: to view variables, modify values of variables on run-time

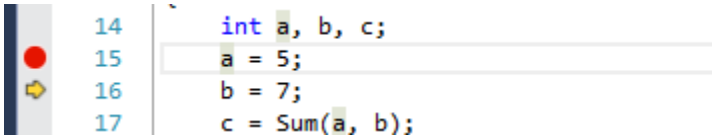
To open the Watch window

From Debug menu ➔ Windows ➔ Watch ➔ Watch1, or Watch2, ect.

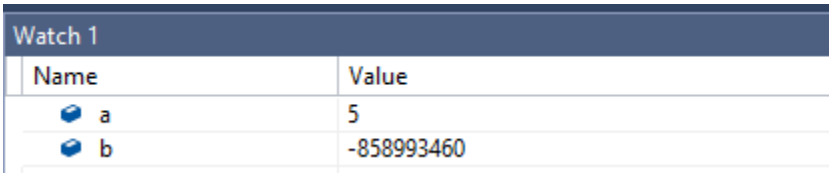
To display variables in watch

- Choose variable ➔ Right mouse click ➔ Add watch or
- In Watch Window, type the variable name.

Suppose you are debugging to line 16:



Watch window will be:



Some special features of Watch:

Special feature of Watch - Pseudo-variables

Besides normal variables, you can also see Pseudo-variables:

- $tid – the thread ID of the current thread
- $pid – the process ID
- $cmdline – the command line string that launched the program
- $user – information for the account running the program
- $registername – displays the content of the register register name

And pseudo-variables for last errors:

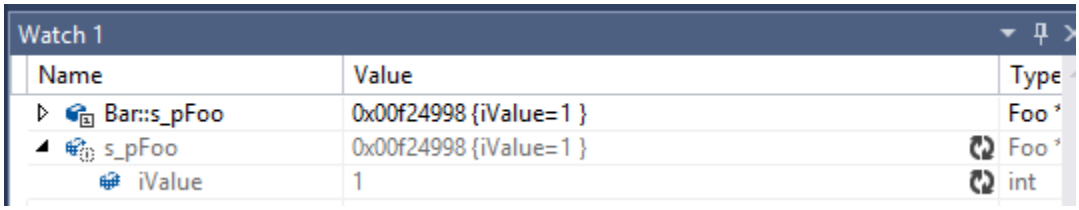
- $err – displays the numeric code of the last error
- $err, hr – displays the message of the last error

Special feature of Watch - Watch Heap Objects after Symbol goes out of scope

Sometimes you'd like to watch the value of an object (on the heap) even after the symbol goes of scope. When that happens, the variable in the Watch window is disabled and cannot be inspected any more (nor updated) even if the object is still alive and well. It is possible to continue to watch it in full capability if you know the address of the object. You can then cast the address to a pointer of the object type and put that in the Watch window.

```
7   class Foo
8   {
9   public:
10      int iValue;
11      Foo(int value) : iValue(value){};
12  };
13
14  class Bar
15  {
16      static Foo* s_pFoo;
17  public:
18      static void DoFoo()
19      {
20          s_pFoo->iValue++;
21      }
22  };
23
24  Foo* Bar::s_pFoo = new Foo(0);
25
26  void main()
27  {
28      Bar::DoFoo();
29      return;
30  }
```

In the example above, s_pFoo is no longer accessible in the Watch window after stepping out of DoFoo(). However, taking its address and casting it to Foo* we can still watch the object.



Special feature of Watch - Watch a range of value inside an array

You can use the syntax (array + <offset>), <count> to watch a particular range of <count> elements starting at the <offset> position.

If you want to watch the entire array, you can simply say array, <count>.

```
26  void TestArray()
27  {
28      int* _array = new int[1000];
29      for (int i = 0; i < 1000; i++)
30      {
31          _array[i] = i;
32      }
33      int a = 0;
34  }
35
36  void main()
37  {
38      TestArray();
39      return;
40  }
```

And values are:

Watch 1

Name	Value
_array	0x00f8a2c0 {0}
0	
(_array + 10), 5	0x00f8a2e8 {10, 11, 12, 13, 14}
[0]	10
[1]	11
[2]	12
[3]	13
[4]	14

Special feature of Watch - Calling Methods From Watch Window

We used watch window to explore the objects and their properties but we also can call a method from watch window as well.

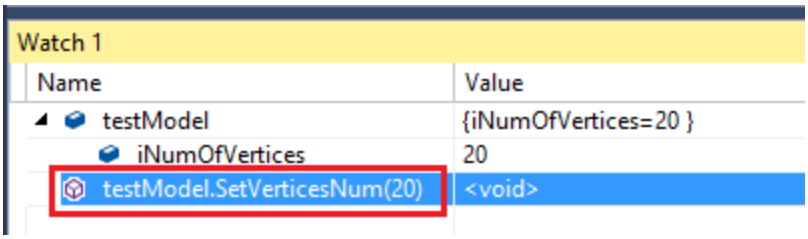
In below example, we set NumOfVertices = 100.

```
59  class Model
60  {
61  public:
62      int iNumOfVertices;
63
64      void SetVerticesNum(int num)
65      {
66          iNumOfVertices = num;
67      }
68  };
69
70  void main()
71  {
72      Model testModel;
73      testModel.SetVerticesNum(100);
74      printf("Is testing watch\n");
75
76      return;
77  }
```

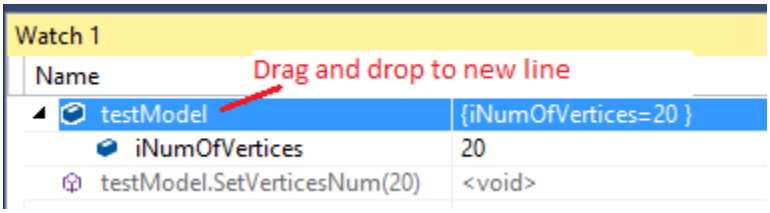
Watch 1

Name	Value
testModel	{iNumOfVertices=100 }
iNumOfVertices	100

However, you call method inside watch. The value will change accordingly:



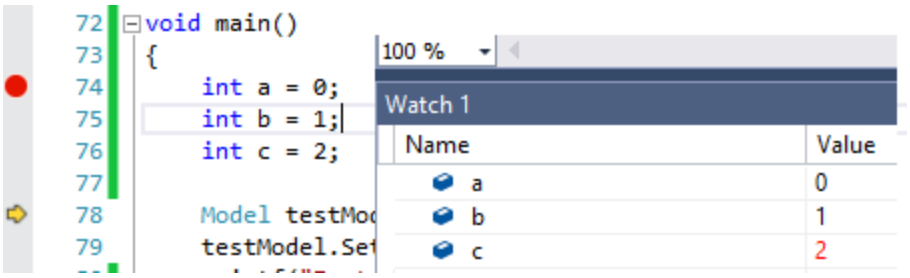
Special feature of Watch – Drag/Drop And Copy/Paste Code inside Watch Windows



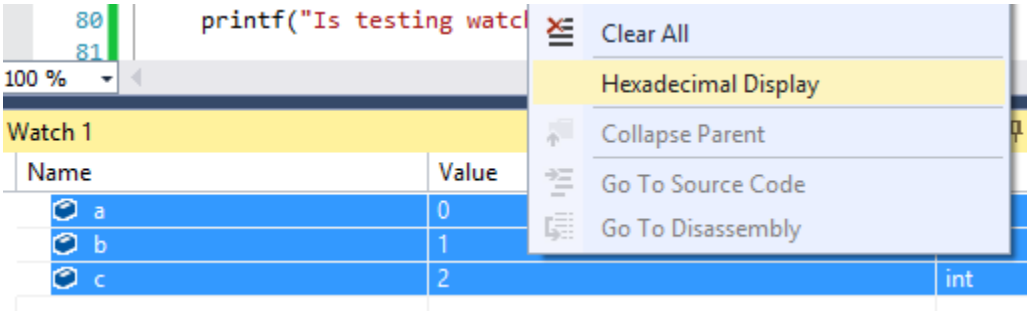
You can also pass values among different Watch Windows (Watch1, Watch2, Watch3, Watch4) by Copy Paste or Drag and Drop

Special feature of Watch – Different Type of Display Format

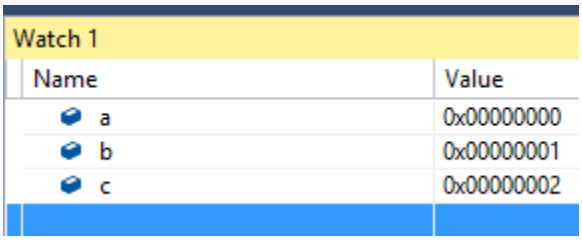
Example with variable a, b, c as below:



Choose Hexadecimal Display:



The result is:

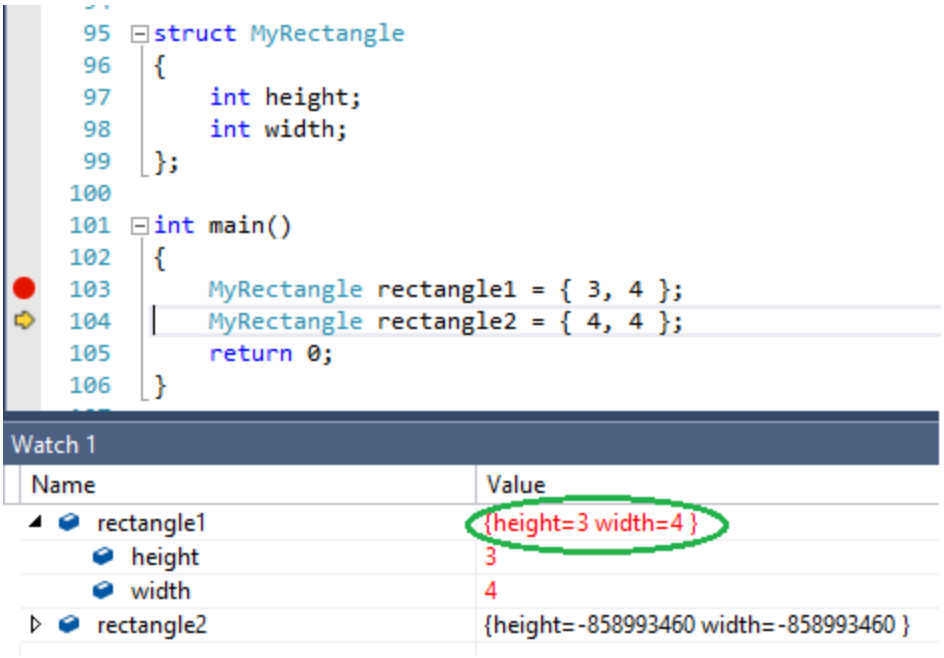


Special feature of Watch – Debugger Type Visualizers for C++

What is Data Type Visualizers?

One of the new features for C++ developers is the new native type visualization framework (natvis) added to the debugger which allows customizing the way data types are displayed in debugger variable windows.

For below example, the data type display is “heigh=3 width=4”



Create a type visualizer to customize data type:

Type visualizers for C++ types are specified in .natvis files. A natvis file is simply an xml file (with .natvis extension) that contains visualization rules for one or more types. At the start of each debugging session, Visual Studio processes any natvis files it can find in the following locations:

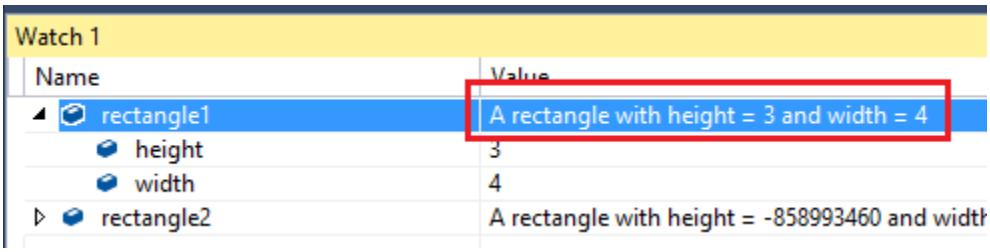
- %VSINSTALLDIR%\Common7\Packages\Debugger\Visualizers (requires admin access)
- %USERPROFILE%\Documents\Visual Studio 2013\Visualizers\
- VS extension folders

For above example, you will create a new file named “rectangle.natvis” containing the following xml (explained further below) and save it in “%USERPROFILE%\Documents\Visual Studio 2013\Visualizers” folder.

```
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">
    <Type Name="MyRectangle">
        <DisplayString>A rectangle with height = {height} and width = {width}</DisplayString>
    </Type>
</AutoVisualizer>
```

Type: element represents a visualizer entry for a type whose fully qualified name is specified in the **Name** attribute.
DisplayString: keywork to customize the string shown in value columm.

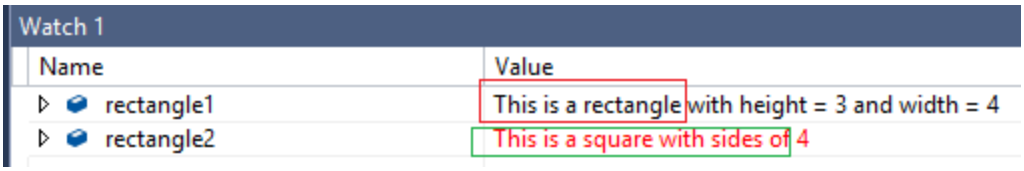
You don’t need to restart VS, just restart debugging. The debugger now displays the object as:



Visualizers with Condition attribute:

```
<?xml version="1.0" encoding="utf-8"?>

<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">

    <Type Name="MyRectangle">

        <DisplayString Condition="height == width">This is a square with sides of
{height}</DisplayString>

        <DisplayString>This is a rectangle with height = {height} and width =
{width}</DisplayString>

    </Type>

</AutoVisualizer>
```

The result will be:



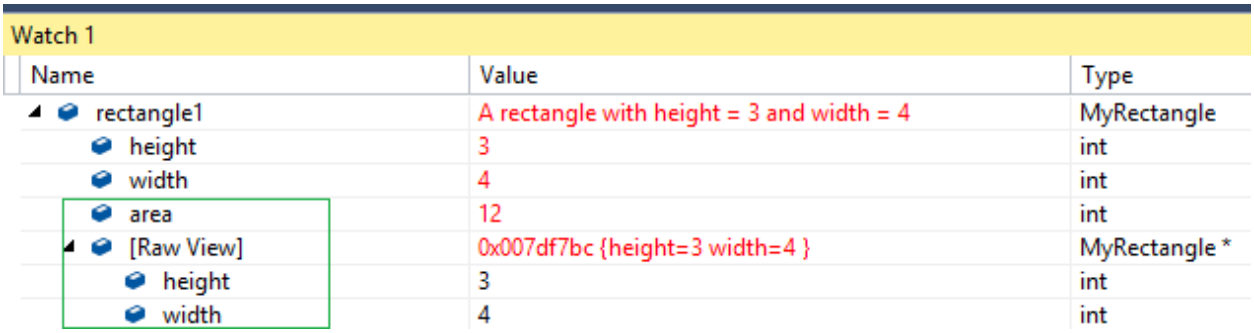
The visualizer with Expand node:

The **Expand** node, which allows you to define child elements for a type is to serve your own purpose. Add the highlighted section below to the visualizer entry and save the file:

```
<?xml version="1.0" encoding="utf-8"?>

<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">

    <Type Name="MyRectangle">

        <DisplayString Condition="height == width">A square with sides of {height}</DisplayString>

        <DisplayString>A rectangle with height = {height} and width = {width}</DisplayString>

        <Expand>

            <Item Name="height">height</Item>

            <Item Name="width">width</Item>
```

```
            <Item Name="area">height * width</Item>
        </Expand>
    </Type>
</AutoVisualizer>
```

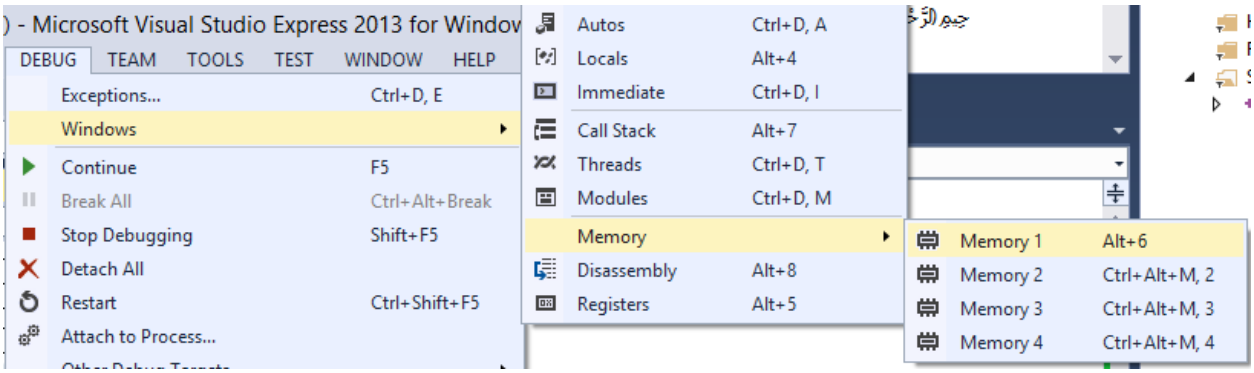
Each **Item** node defines a single child element whose name is given by the **Name** attribute and whose value is given by the expression in the node text.
The result will be:



5. Using Memory Window

The Memory window provides a view into the memory space that is used by your application. It shows you the large-scale picture of memory and does not limited to displaying data. It displays everything in the memory space, whether the content is data, code, or random bits of garbage in unassigned memory.

To display Memory Window:



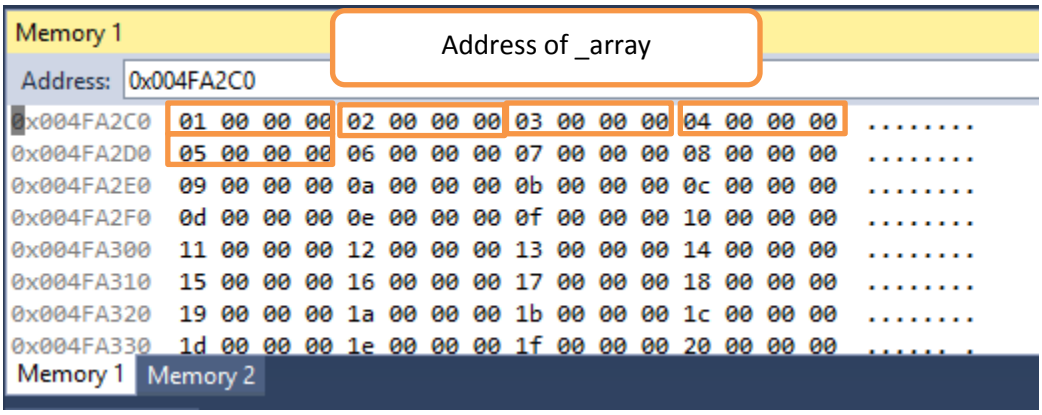
You have Memory 1, 2, 3, 4 to choose.

To select a memory location:
Suppose that you have below code:

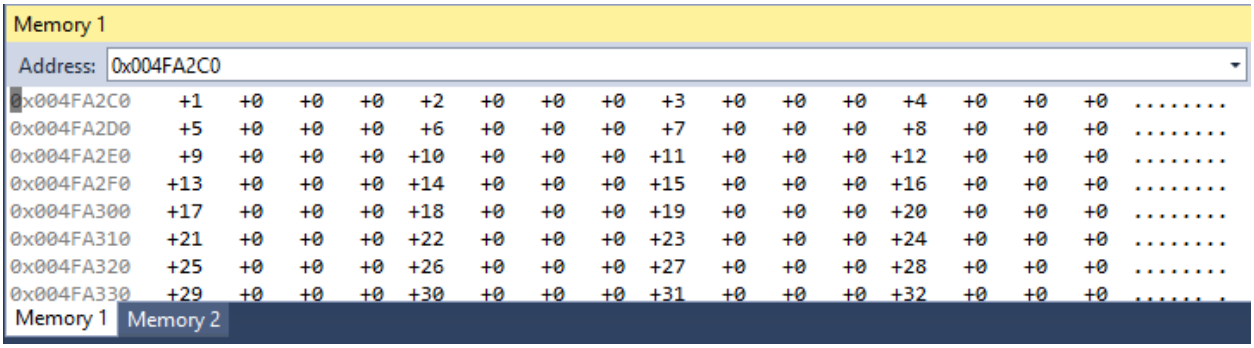
```
26  void TestArray()
27  {
28      int* _array = new int[1000];
29      for (int i = 0; i < 1000; i++)
30      {
31          _array[i] = i + 1;
32      }
33      int a = 0;
34  }
```

Value of _array = {1, 2, 3, 4, 5, ….}

To select _array location, just select _array (a memory address or pointer variable that contains a memory address) and drag the address or pointer to the Memory window.
You can see the memory address here:



Right-mouse click to change view-format. Ex by “Singed display”

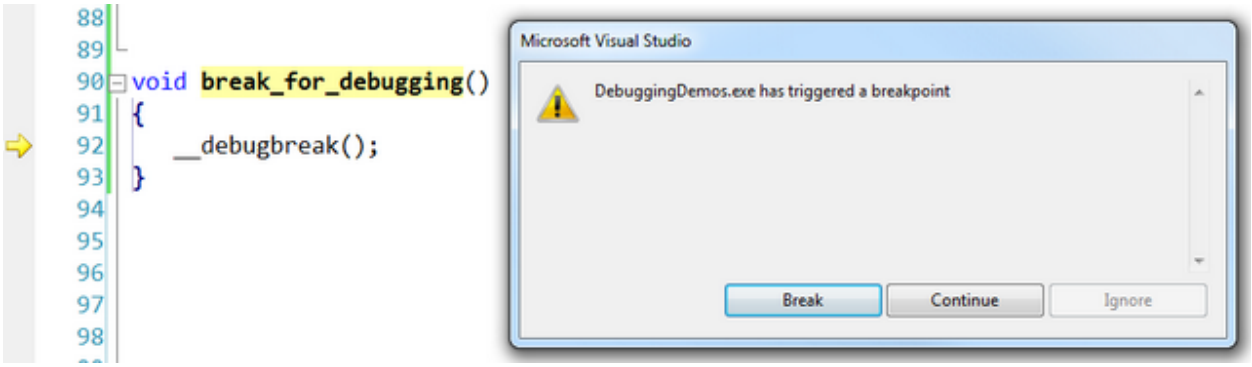


6. Launch the debugger from code

Seldom you might need to attach with the debugger to a program, but you cannot do it with the Attach window (maybe because the break would occur too fast to catch by attaching), nor you can start the program in debugger in the first place. You can cause a break of the program and give the debugger a chance to attach by calling the `__debugbreak()` intrinsic.

```
void break_for_debugging()
{
   __debugbreak();
}
```

When break_for_debugging() is called:



7. Detect Memory leaks

The primary tools for detecting memory leaks are the debugger and the C Run-Time Libraries (CRT) debug heap functions.

To enable the debug heap functions, include the following statements in your program in exact order:

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
```

Place a call to _CrtDumpMemoryLeaks() before an application exit point to display a memory-leak report when your application exits.

```
_CrtDumpMemoryLeaks();
```

If your application has multiple exits, a call to _CrtSetDbgFlag at the beginning of your application will cause an automatic call to _CrtDumpMemoryLeaks at each exit point.

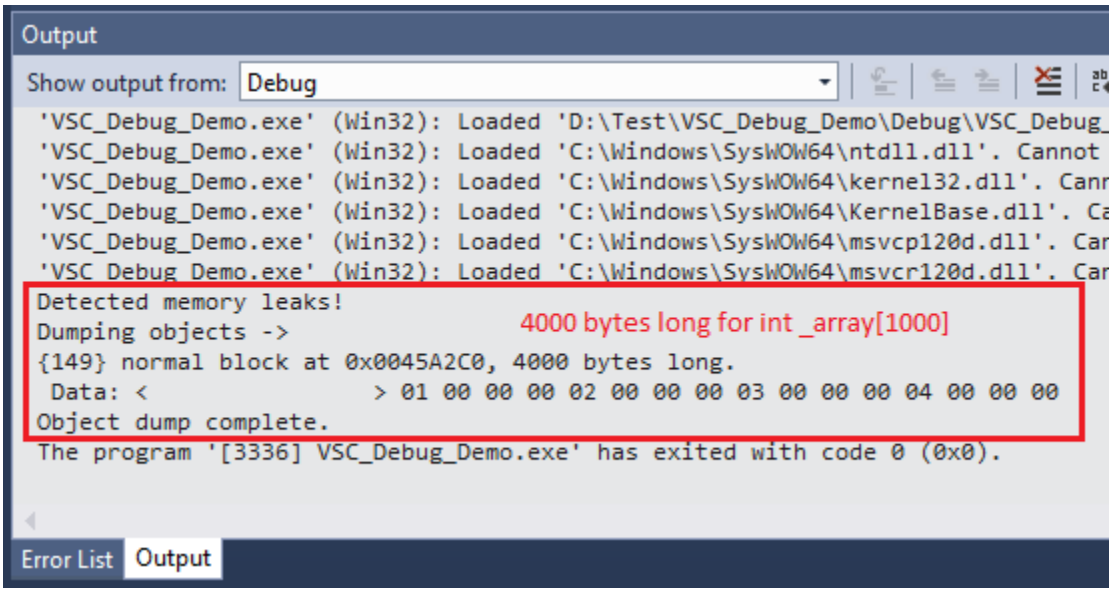
```
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
```

Example with below code:

```
void TestArray()
{
	int* _array = new int[1000];
	for (int i = 0; i < 1000; i++)
	{
		_array[i] = i + 1;
	}
	int a = 0;
}
```

```
void main()
{
	_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
	TestArray();

	return;
}
```

You can see memory leak at **Output Window** as following:



If your application does not define _CRTDBG_MAP_ALLOC, _CrtDumpMemoryLeaks displays a memory-leak report that looks like this:

```
Detected memory leaks!
Dumping objects ->
{150} normal block at 0x0064A2C0, 4000 bytes long.
 Data: <                > 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00
Object dump complete.
```

With define

```
#ifdef _DEBUG
#ifndef DBG_NEW
#define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
#define new DBG_NEW
#endif
#endif  // _DEBUG
```

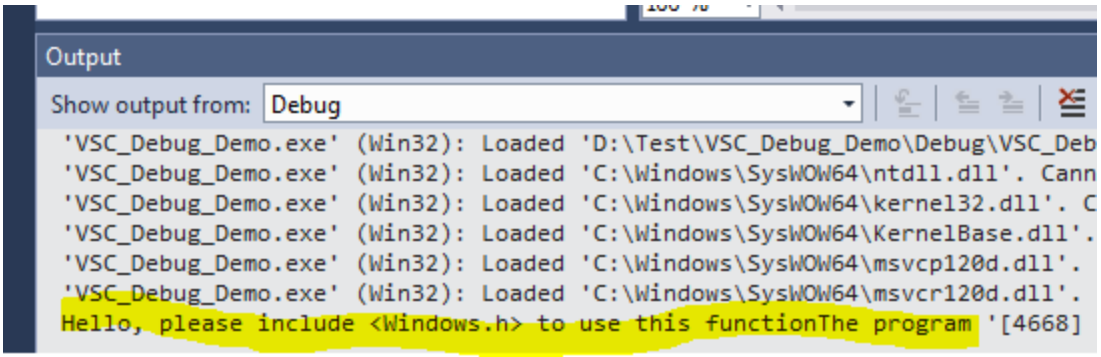
Your application will look like:

```
Detected memory leaks!
Dumping objects ->
d:\test\vsc_debug_demo\vsc_debug_demo\main.cpp(40) : {150} normal block at 0x00B6A2C0,
4000 bytes long.
 Data: <                > 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00
Object dump complete.
```

8. Print to Output Window

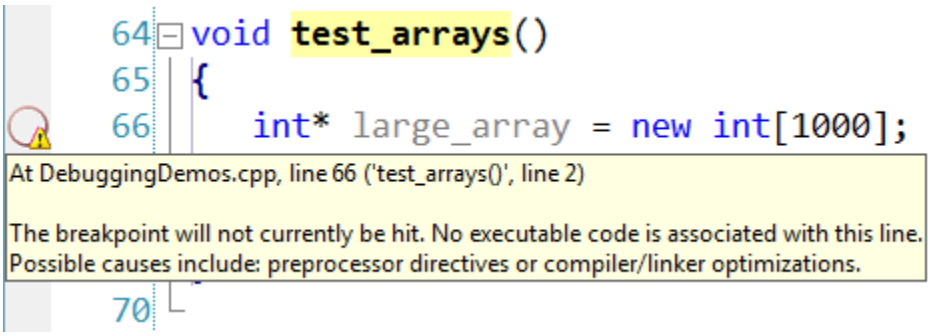
```
void main()
{
    ::OutputDebugString(LPCSTR("Hello, please include <Windows.h> to use this function"));
    return;
}
```

The result is:



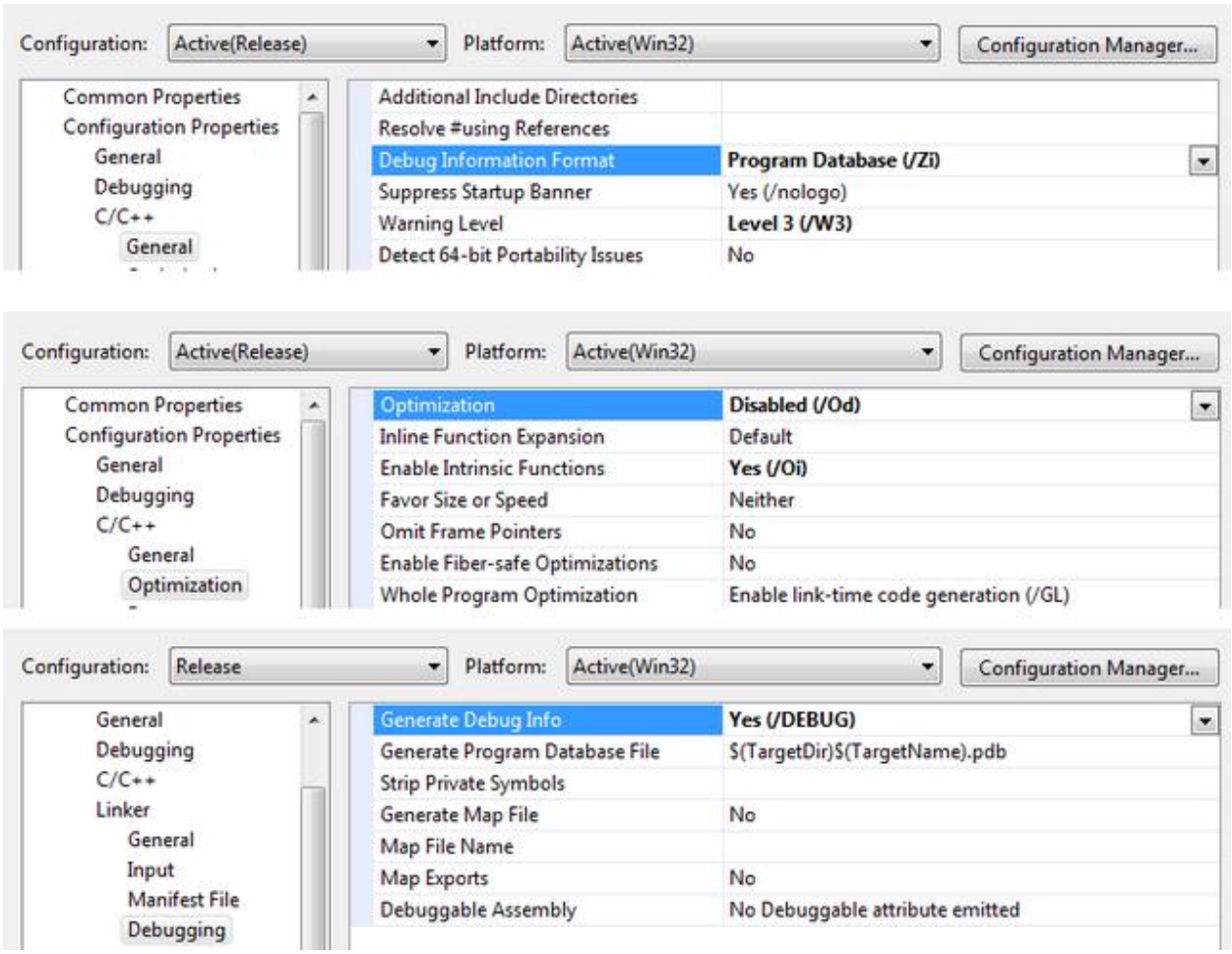
9. Debug the Release Build

Debug and Release builds are meant for different purposes. While a Debug configuration is used for development, a Release configuration, as the name implies should be used for the final version of a program. Since it's supposed that the application meets the required quality to be published, such a configuration contains optimizations and settings that break the debugging experience of a Debug build. Still, sometimes you'd like to be able to debug the Release build the same way you debug the Debug build. To do that, you need to perform some changes in the configuration. However, in this case one could argue you no longer debug the Release build, but rather a mixture of the Debug and the Release builds.



There are several things you should do; the mandatory ones are:

- C/C++ > General > Debug Information Format should be "Program Database (/Zi)"
- C/C++ > Optimization > Optimization should be "Disabled (/Od)"
- Linker > Debugging > Generate Debug Info should be "Yes (/DEBUG)"


Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Additional Include Directories
Resolve #using References
Debug Information Format
Program Database (/Zi)
Suppress Startup Banner
Yes (/nologo)
Warning Level
Level 3 (/W3)
Detect 64-bit Portability Issues
No
Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Optimization
Optimization
Disabled (/Od)
Inline Function Expansion
Default
Enable Intrinsic Functions
Yes (/Oi)
Favor Size or Speed
Neither
Omit Frame Pointers
No
Enable Fiber-safe Optimizations
No
Whole Program Optimization
Enable link-time code generation (/GL)
Configuration: Release
Platform: Active(Win32)
Configuration Manager...
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
Generate Debug Info
Yes (/DEBUG)
Generate Program Database File
$(TargetDir)$(TargetName).pdb
Strip Private Symbols
Generate Map File
No
Map File Name
Map Exports
No
Debuggable Assembly
No Debuggable attribute emitted