Debugging in Visual Studio 2010

Lee, Sungwon

Department of Computer Engineering,
Kyung Hee University.

Introduction

❑ Debugging :

- a process of finding out defects in the program and fixing them.
- When you have some defects in your code, first of all you need to identify the **root cause** of the defect.

❑ How to debug the code?

- **Visual Studio IDE**
- VS IDE provides a lot of handy tools which help to debug code

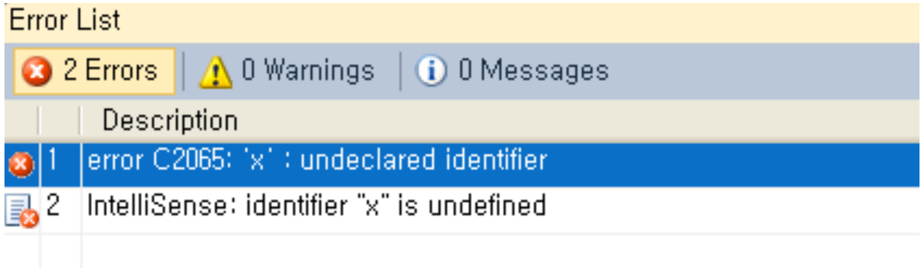
Debugger Features

- ❑ error listening
- ❑ adding breakpoints
- ❑ visualize the program flow
- ❑ control the flow of execution
- ❑ data tips
- ❑ watch variables

Debugging

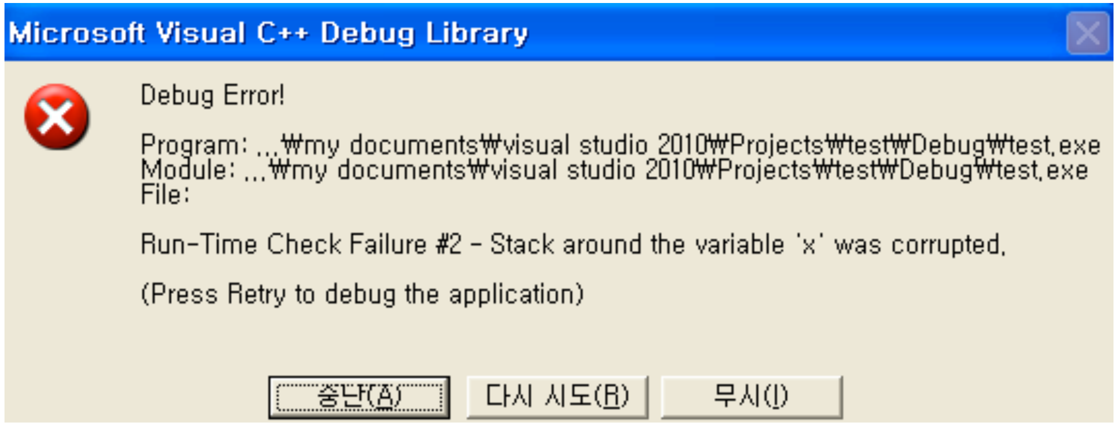
❑ Two main types of code errors

- 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



```
Error List
2 Errors | 0 Warnings | 0 Messages
    Description
1   error C2065: 'x' : undeclared identifier
2   IntelliSense: identifier "x" is undefined
```

<Fig. 1> Syntax error



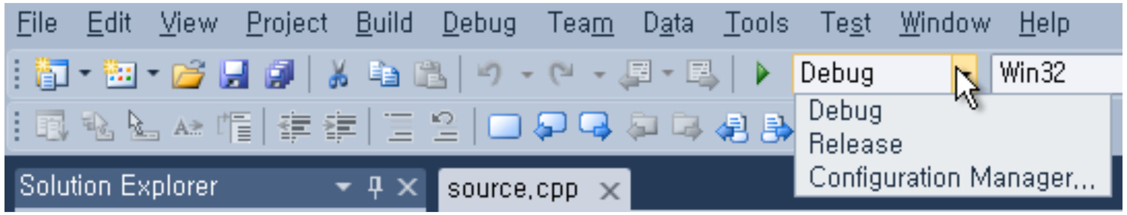
```
Microsoft Visual C++ Debug Library

Debug Error!

Program: ...₩my documents₩visual studio 2010₩Projects₩test₩Debug₩test.exe
Module: ...₩my documents₩visual studio 2010₩Projects₩test₩Debug₩test.exe
File:

Run-Time Check Failure #2 - Stack around the variable 'x' was corrupted.

(Press Retry to debug the application)
```

<Fig. 2> Runtime error

Project Configuration Setting

❑ Debug vs. Release Configurations

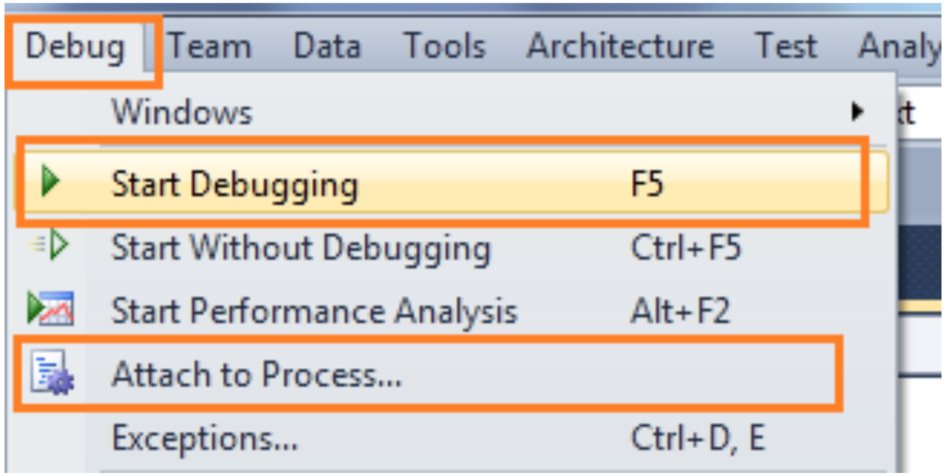
- The **Debug** configuration of your program is compiled with ful symbolic debug information and no optimization
- The **Release** configuration of your program is fully optimized and contains no symbolic debug information
- Must be in Debug configuration to debug your program



〈Fig. 3〉 Project configuration settings

How to Start?

- ❑ You can start debugging form the Debug menu
- ❑ Select “Start Debugging” or just press “F5”

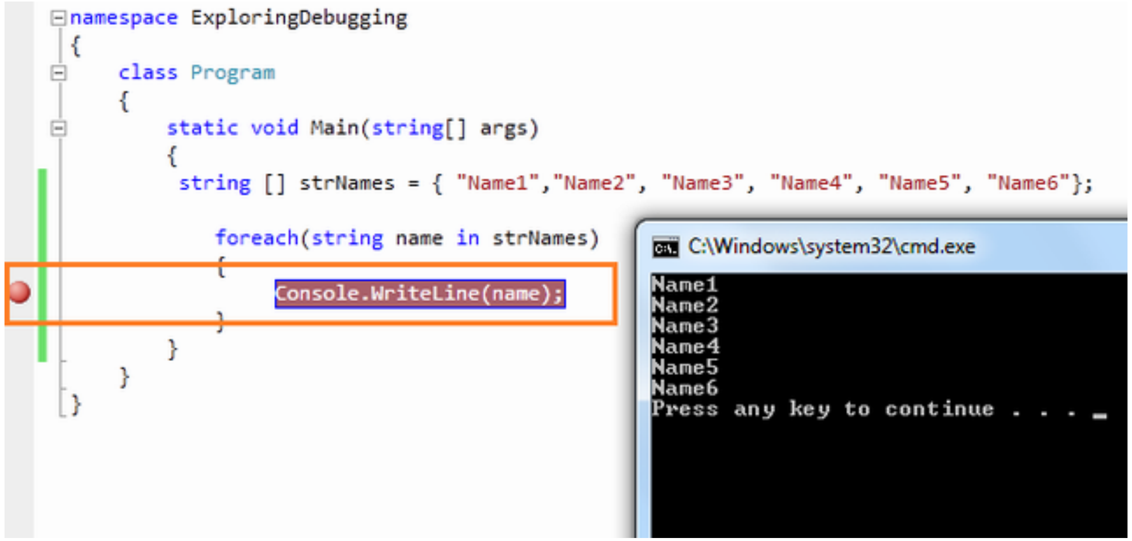


<Fig. 4> Start Debugging

- “Attach to Process” sill start a debug session for the application
- Mainly attaching process for debugging ASP.NET web application

Breakpoints

- ❑ Breakpoint is used to notify debugger where and when to pause the execution of program
- ❑ add or remove(toggle) breakpoint
  - clicking on the side bar of code
  - pressing F9 at the front of the line
- ❑ When the debugger reaches the breakpoint, you can check out what’s going wrong within the code



```
namespace ExploringDebugging
{
    class Program
    {
        static void Main(string[] args)
        {
         string [] strNames = { "Name1","Name2", "Name3", "Name4", "Name5", "Name6"};

            foreach(string name in strNames)
            {
                Console.WriteLine(name);
            }
        }
    }
}
```

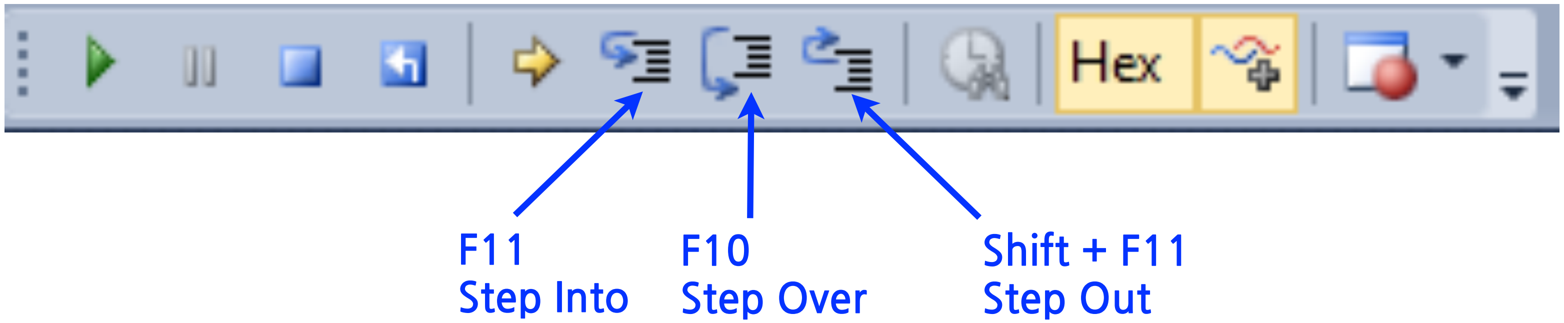
C:\Windows\system32\cmd.exe

```
Name1
Name2
Name3
Name4
Name5
Name6
Press any key to continue . . . _
```

<Fig. 5> Set Breakpoint

Debugging with Breakpoints

- Start program by pressing “**F5**”
- When the program reaches the breakpoint, execution will automatically pause
- You have several commands available in break mode, using which you can proceed for further debugging



F11
Step Into

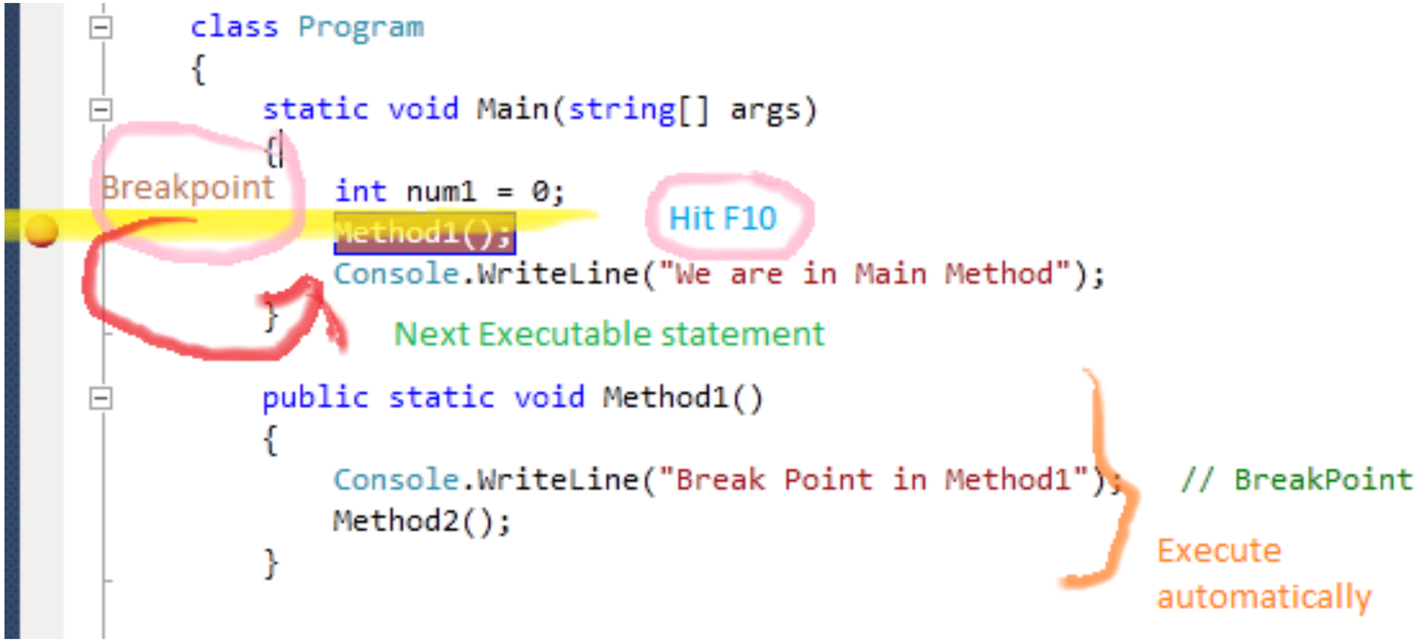
F10
Step Over

Shift + F11
Step Out

<Fig. 6> Breakpoint Toolbar

Step Over

- ❑ You may need to execute the code line by line
- ❑ **“Step Over”[F10]** command is used to execute the code line by line
- ❑ Step Over will **execute the entire method at a time**

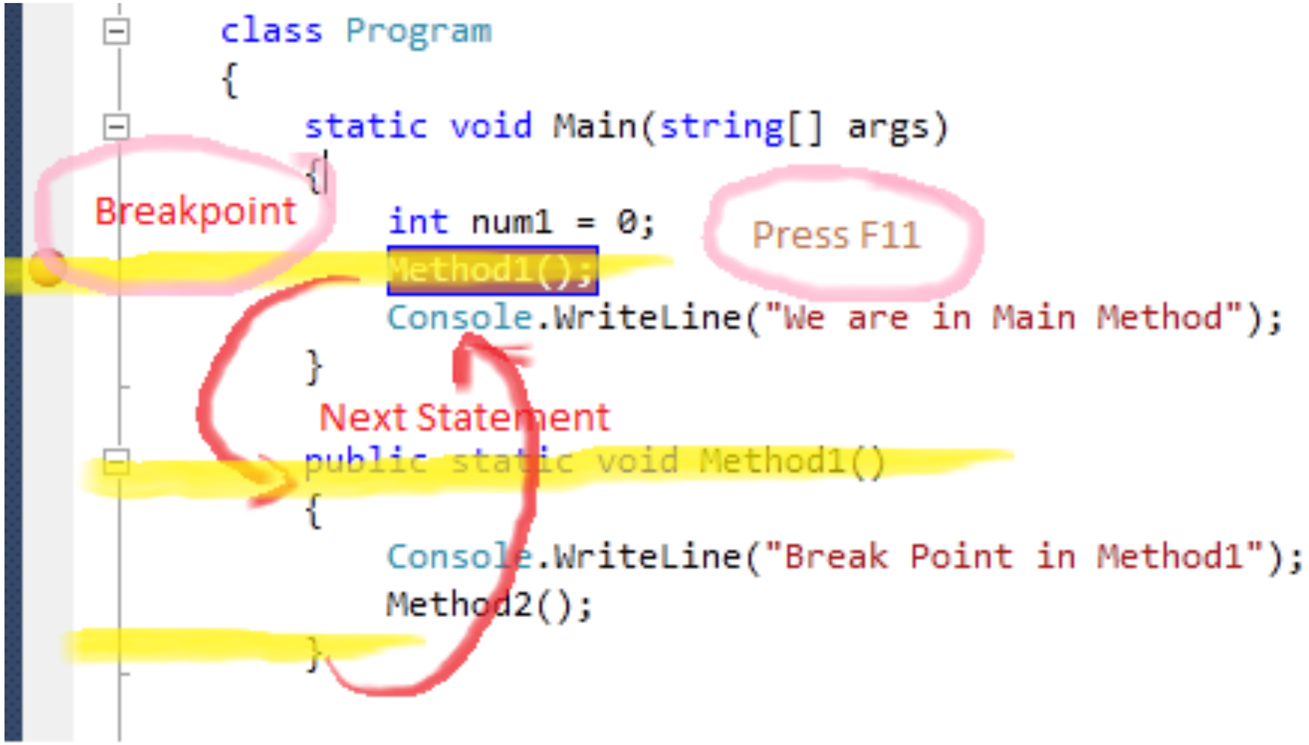


```
class Program
{
    static void Main(string[] args)
    {
        int num1 = 0;
        Method1();
        Console.WriteLine("We are in Main Method");
    }

    public static void Method1()
    {
        Console.WriteLine("Break Point in Method1");   // BreakPoint
        Method2();
    }
```

<Fig. 7> Step Over - F10

Step Into

- ❑ Similar to Step Over
- ❑ Difference
  - if the current highlighted section is any method call, the debugger will **go inside the method**



```
class Program
{
    static void Main(string[] args)
    {
        int num1 = 0;
        Method1();
        Console.WriteLine("We are in Main Method");
    }

    public static void Method1()
    {
        Console.WriteLine("Break Point in Method1");
        Method2();
    }
```

<Fig. 8> Step Into - F11

Step Out / Continue

- ❑ Step Out [Shift + F11]
  - When you are debugging inside a method
  - Complete the execution of the method
  - Pause at the next statement from where it called.
- ❑ Continue [F5]
  - Run your application again
  - Continue the program flow unless it reaches the next breakpoint

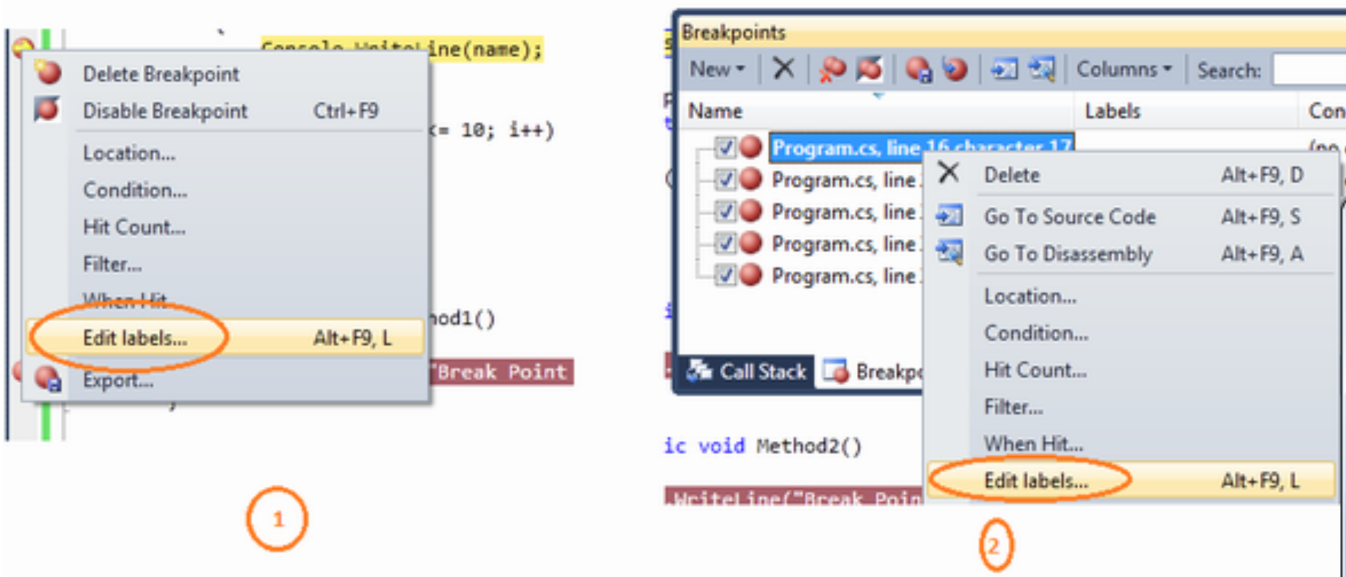
Set Next Statement

- ❑ **Change the path of execution** of program while debugging
- ❑ If you want to change the execution path when your program paused
  - **Go to the particular line**
  - **Right click** on the line and select "**Set Next Statement**" or press [**Ctrl + Shift + F10**]
- ❑ Show next Statement [Ctrl + *]
  - Link marked as a yellow arrow
  - These lines indicate that it will be executed next when we continue the program

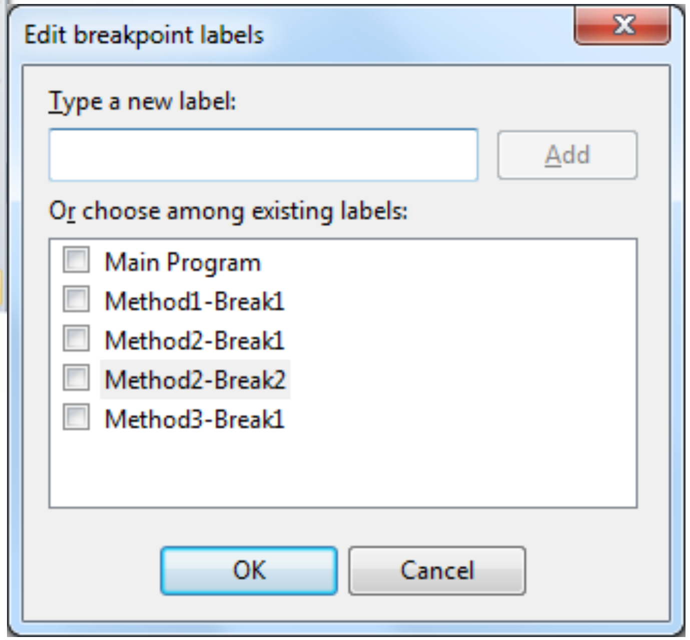
Labeling in Breakpoint (1/2)

❑ You can add the label for each and every breakpoints

- **Right Click** on Breakpoint, Click on the **Edit Labels** link
- Side bar of code / Breakpoint List



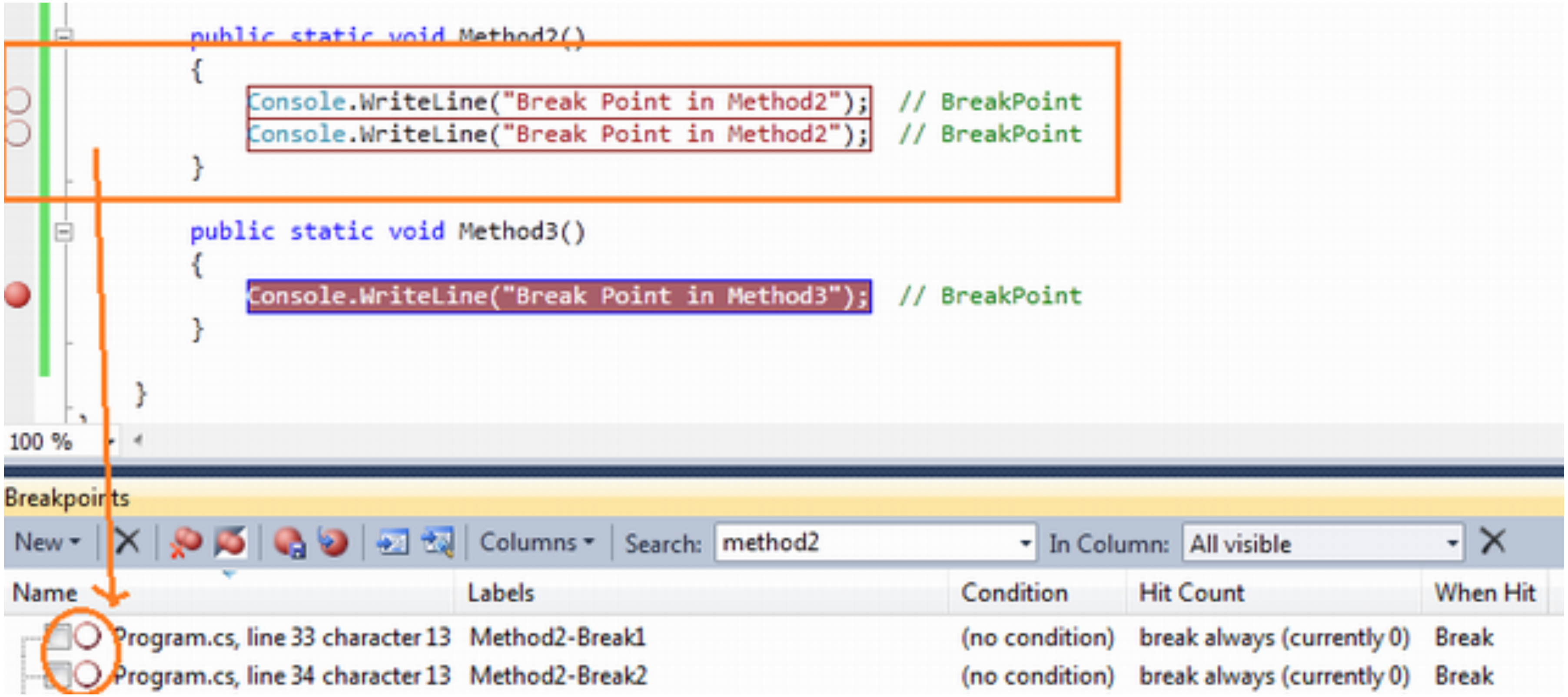
<Fig. 9> Setting Breakpoint Label



<Fig. 10> Adding Breakpoint Label

Labeling in Breakpoint (2/2)

- If you don't want to debug some method, you can filter/search breakpoints in there by label name and disable easily by selecting them together.



```
public static void Method2()
{
    Console.WriteLine("Break Point in Method2");  // BreakPoint
    Console.WriteLine("Break Point in Method2");  // BreakPoint
}

public static void Method3()
{
    Console.WriteLine("Break Point in Method3");  // BreakPoint
}
}
```

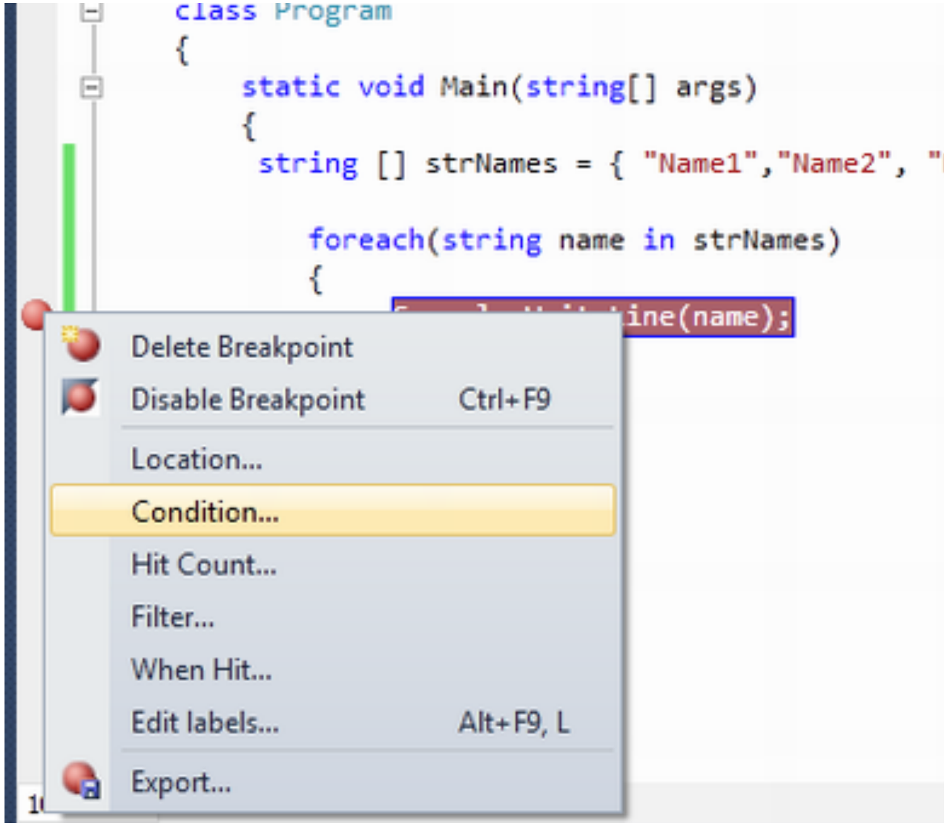
Name	Labels	Condition	Hit Count	When Hit
☐ ○ Program.cs, line 33 character 13	Method2-Break1	(no condition)	break always (currently 0)	Break
☐ ○ Program.cs, line 34 character 13	Method2-Break2	(no condition)	break always (currently 0)	Break

<Fig. 11> Filter Breakpoint Using Labels

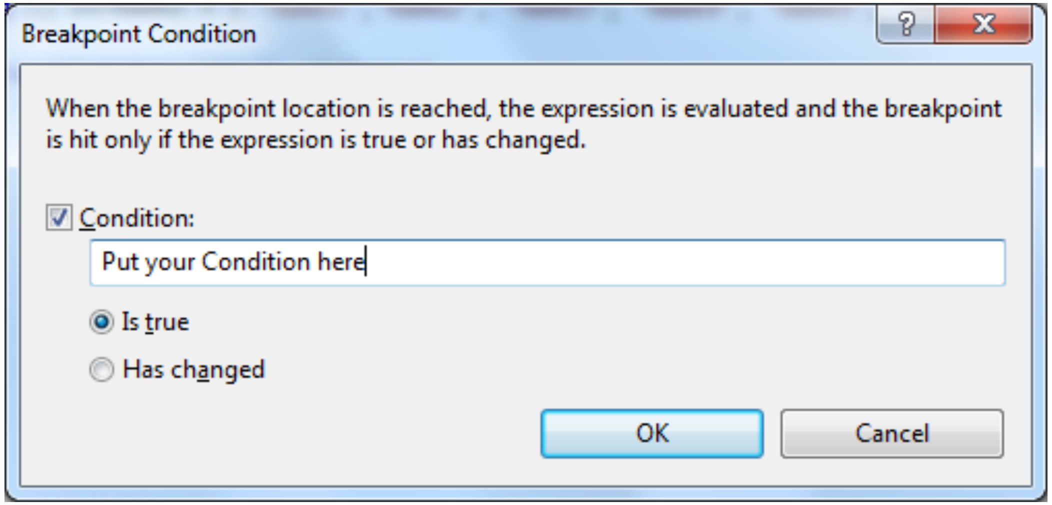
Conditional Breakpoint (1/2)

❑ If you want to pause your program on some specific condition

- Visual Studio Breakpoints allow you to put conditional breakpoint



```
class Program
{
    static void Main(string[] args)
    {
     string [] strNames = { "Name1","Name2", "Name3", "N

        foreach(string name in strNames)
        {
            ine(name);
```



Breakpoint Condition

When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.

☑ Condition:

Put your Condition here

● Is true

○ Has changed

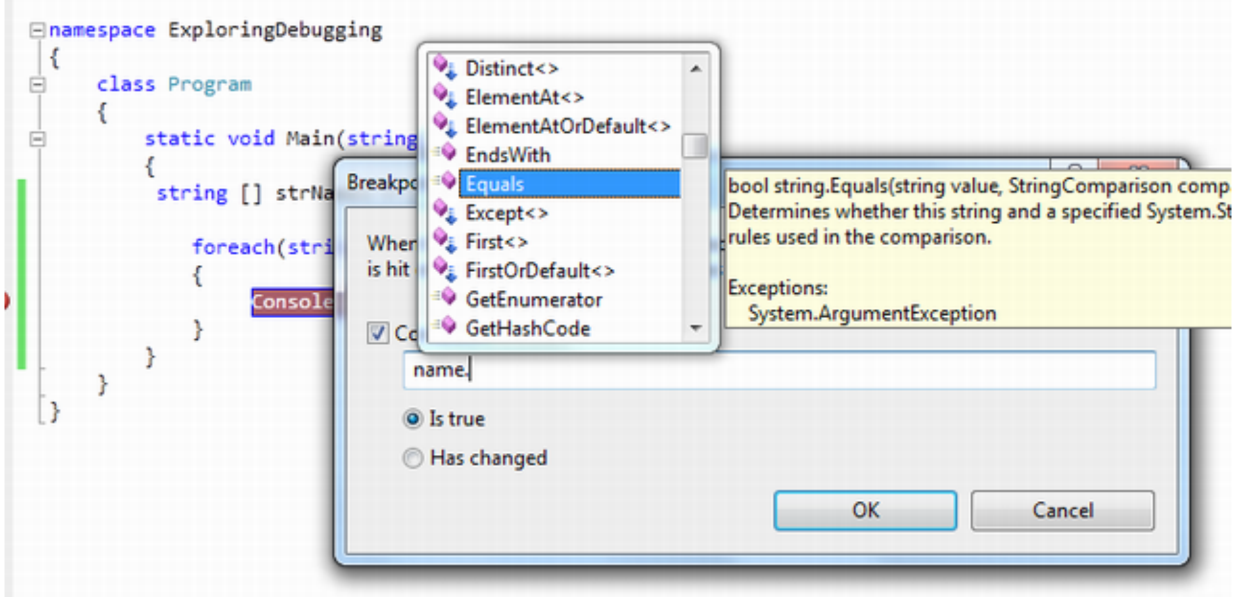
〈Fig. 13〉 Breakpoint Condition Settings

〈Fig. 12〉 Set Breakpoint Condition

Conditional Breakpoint (2/2)

❑ Intellisense In Condition Text Box

- VS IDE provide the intellisense within the condition textbox



```
namespace ExploringDebugging
{
    class Program
    {
        static void Main(string
        {
         string [] strNa

            foreach(stri
            {
                Console
            }
        }
    }
}
```

bool string.Equals(string value, StringComparison comp
Determines whether this string and a specified System.St
rules used in the comparison.

Exceptions:
  System.ArgumentException

name.

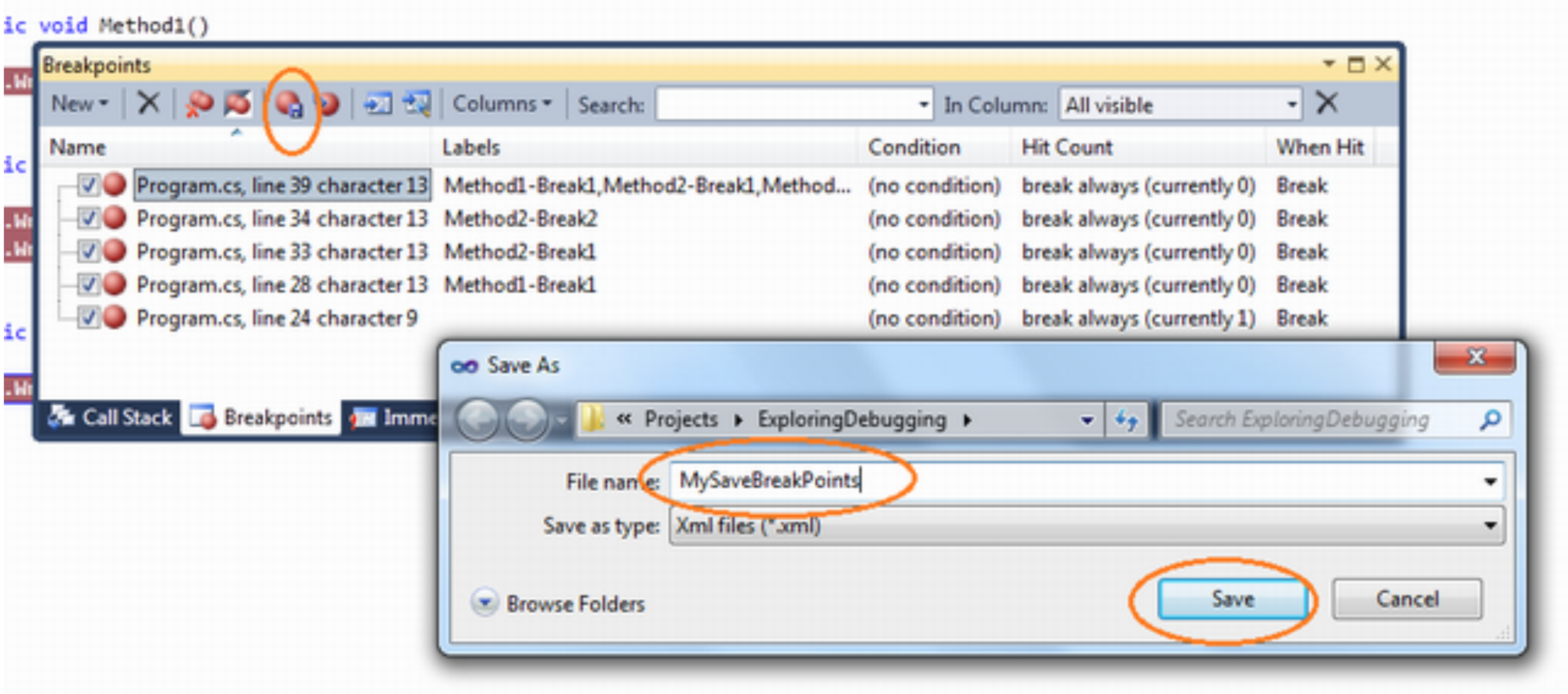
<Fig. 14> Intellisense in condition textbox

❑ Options

- Is True
- Has Changed

Import / Export Breakpoint

- Visual Studio saves breakpoints in as XML Format
- Breakpoint Import depends on link number



```
ic void Method1()
```

Name	Labels	Condition	Hit Count	When Hit
☑ ● Program.cs, line 39 character 13	Method1-Break1,Method2-Break1,Method...	(no condition)	break always (currently 0)	Break
☑ ● Program.cs, line 34 character 13	Method2-Break2	(no condition)	break always (currently 0)	Break
☑ ● Program.cs, line 33 character 13	Method2-Break1	(no condition)	break always (currently 0)	Break
☑ ● Program.cs, line 28 character 13	Method1-Break1	(no condition)	break always (currently 0)	Break
☑ ● Program.cs, line 24 character 9		(no condition)	break always (currently 1)	Break

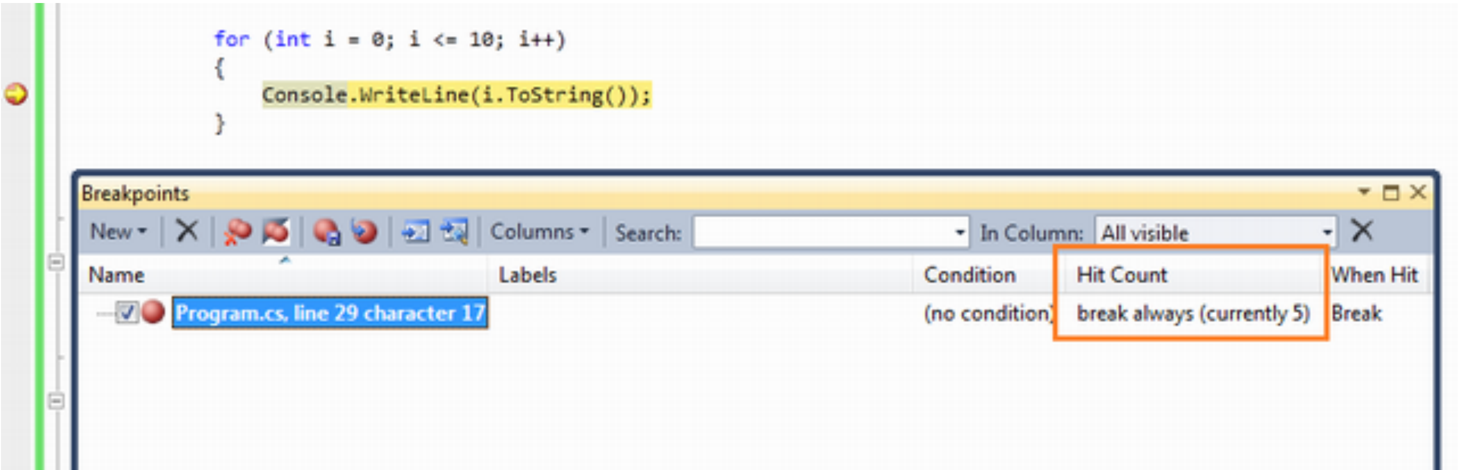
File name: MySaveBreakPoints

Save as type: Xml files (*.xml)

<Fig. 15> Save Breakpoints

Breakpoint Hit Count

- ❑ To keep track of how many times the debugger has paused at some particular breakpoint

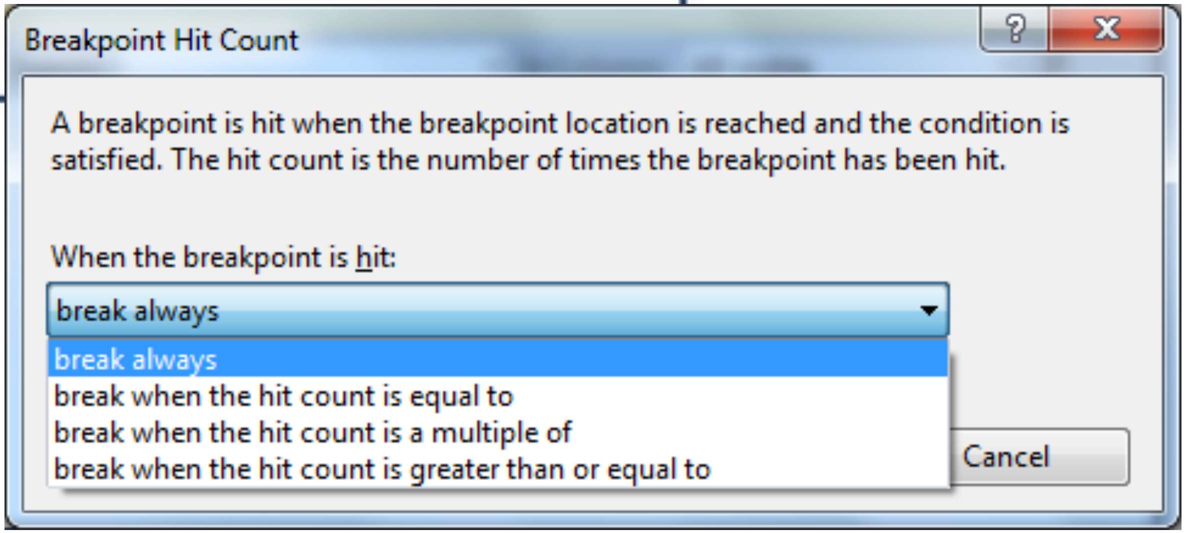


```
for (int i = 0; i <= 10; i++)
{
    Console.WriteLine(i.ToString());
}
```

Breakpoints

Name	Labels	Condition	Hit Count	When Hit
☑ Program.cs, line 29 character 17		(no condition)	break always (currently 5)	Break

<Fig. 16> Breakpoint Hit Count



Breakpoint Hit Count

A breakpoint is hit when the breakpoint location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

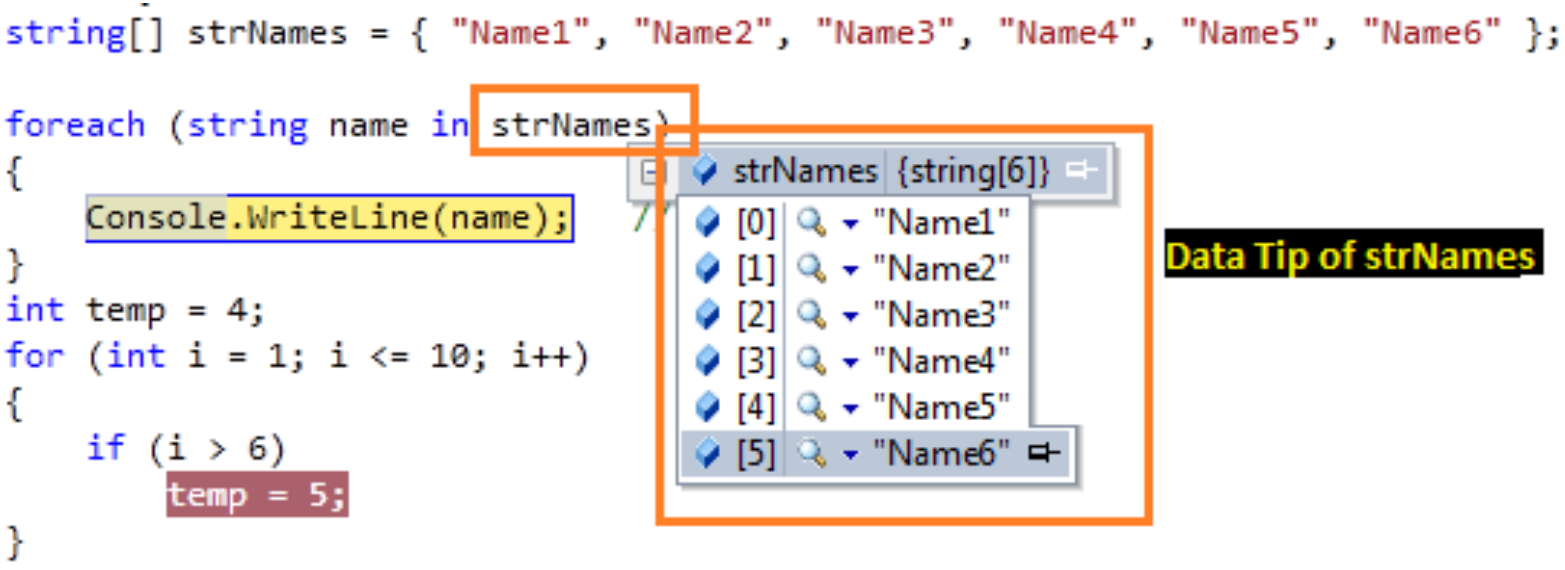
When the breakpoint is hit: break always

break always
break when the hit count is equal to
break when the hit count is a multiple of
break when the hit count is greater than or equal to
Cancel

<Fig. 17> Breakpoint Hit Count Options

Data Tip

- ❑ Kind of an advanced tool tip message which is used to inspect the objects or variable during the debugging of the application
- ❑ When debugger hits the breakpoint
  - if you mouse over to any of the objects of variables, you can see their current values.



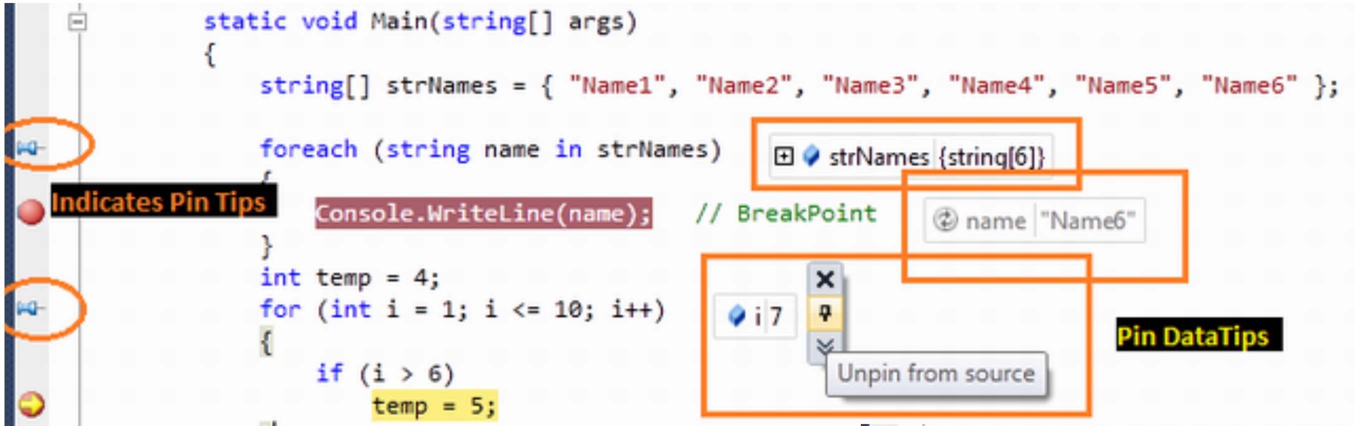
```
string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

foreach (string name in strNames)
{
    Console.WriteLine(name);
}
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}
```

```
strNames {string[6]}
[0] "Name1"
[1] "Name2"
[2] "Name3"
[3] "Name4"
[4] "Name5"
[5] "Name6"
```

Data Tip of strNames

<Fig. 18> DataTips During Debugging

Pin Inspect Value / Drag-Drop Pin Data Tip



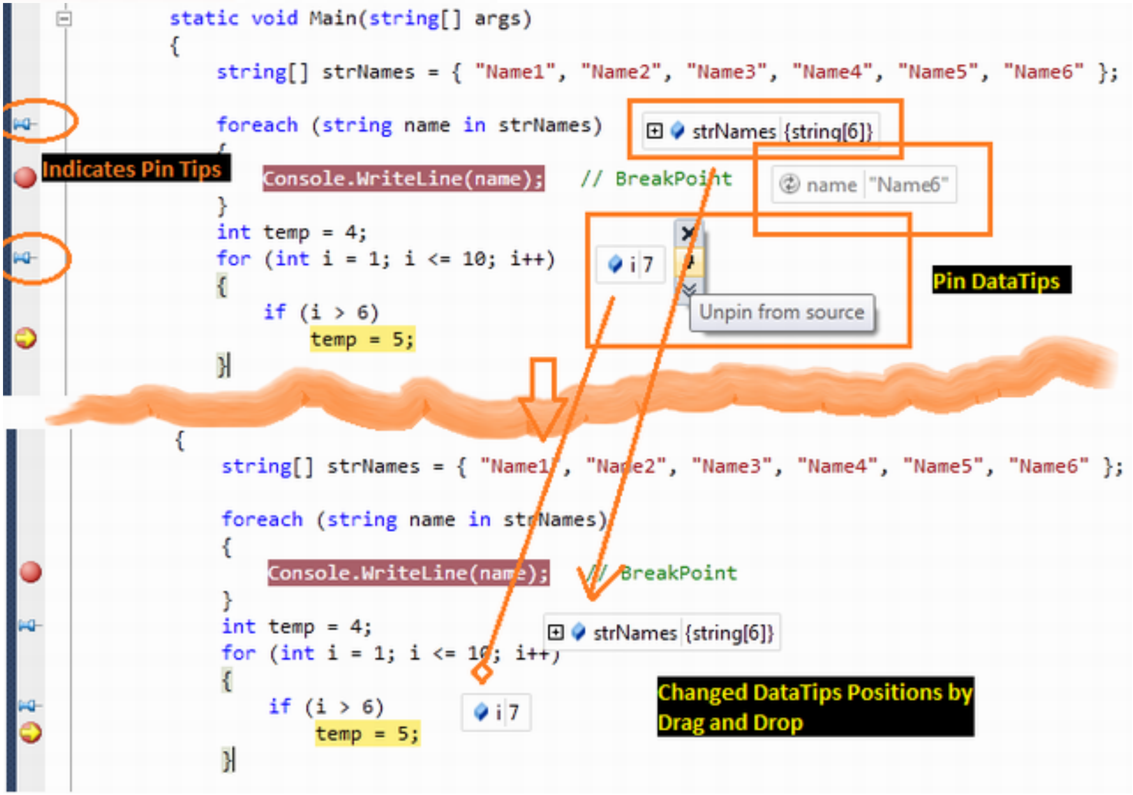
```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // BreakPoint
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

strNames {string[6]}

name "Name6"

i 7

<Fig. 19> Pin Inspect Value During Debugging



```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // BreakPoint
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

strNames {string[6]}

name "Name6"

i 7

```
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // BreakPoint
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

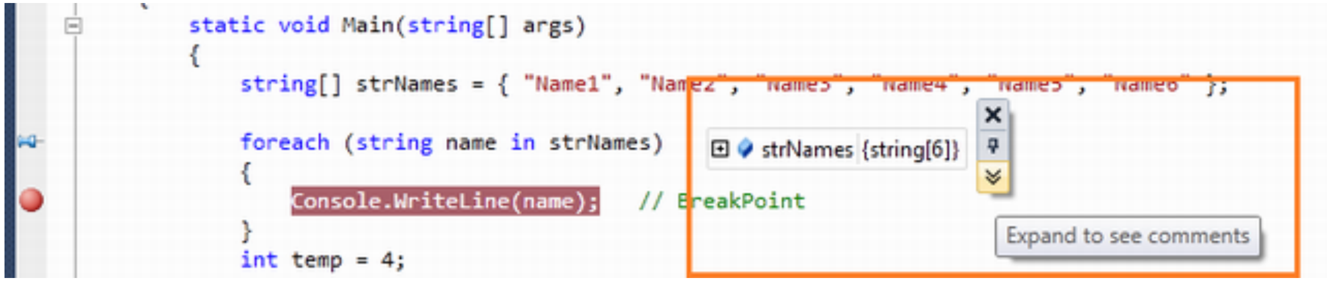
strNames {string[6]}

i 7

<Fig. 20> Drag and Drop Data Tips

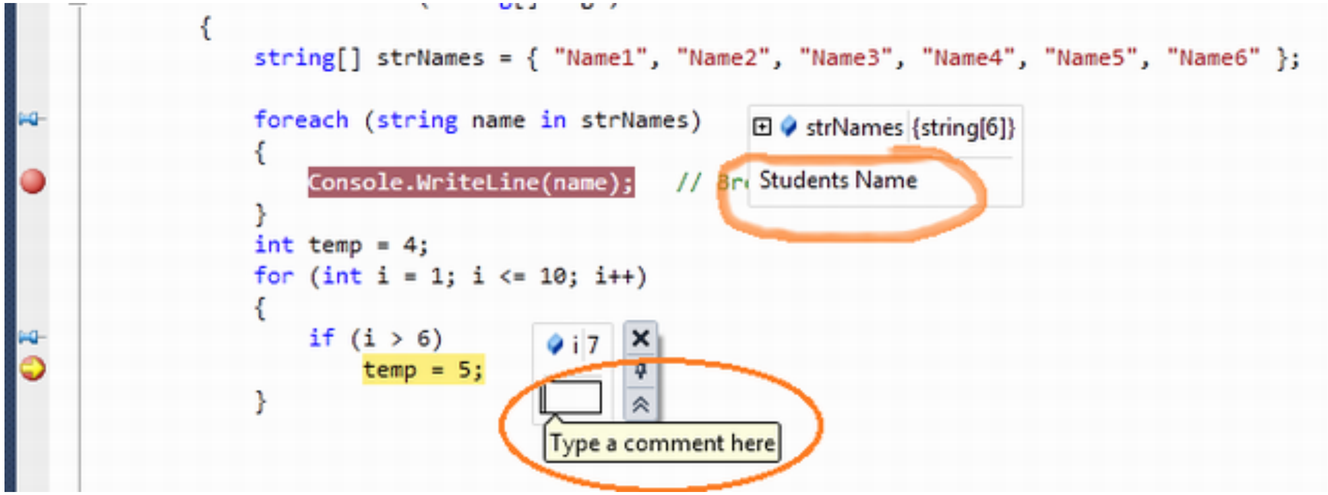
Adding Comments

- ❑ Expand to see the comments
- ❑ Adding comments on pinned inspect value



```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // BreakPoint
    }
    int temp = 4;
```

〈Fig. 21〉 Comments in DataTip



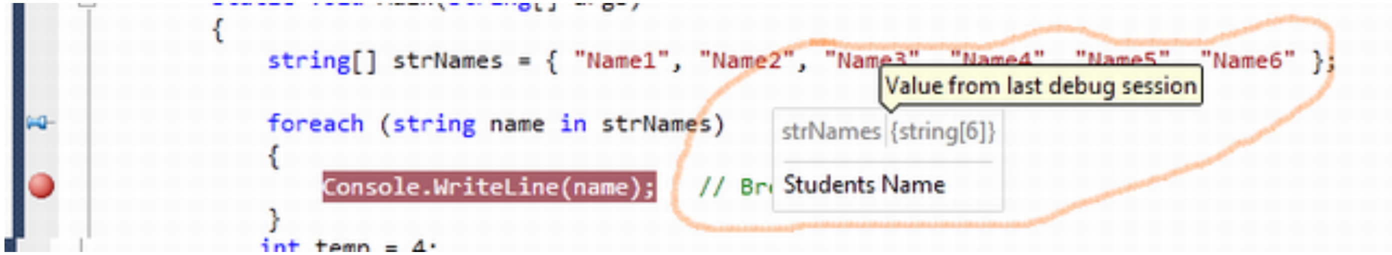
```
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);    // Br
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }
```

〈Fig. 22〉 Adding Comments For Datatips

Last Session / Change Value

❑ Last session debug value

- The value of pinned item will remain stored in a session.
- If you mouse over the pin icon, it will show the details of the last debugging session value



```
        static void Main(string[] args)
        {
            string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

            foreach (string name in strNames)
            {
                Console.WriteLine(name);    // Br
            }
            int temp = 4;
```

Value from last debug session

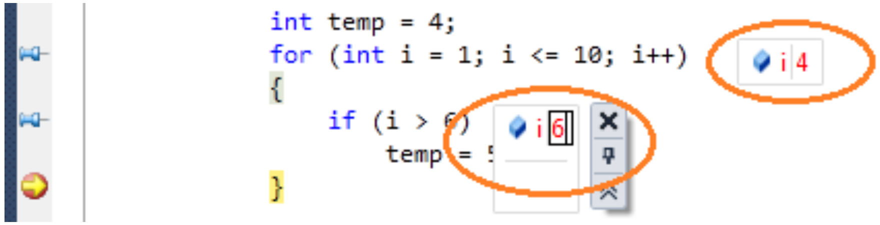
strNames {string[6]}

Students Name

<Fig. 23> Last Session Debug Value

❑ Change value using data tips

- From the list of Pinned objects, you can change their value to see the impact on the program



```
            int temp = 4;
            for (int i = 1; i <= 10; i++)
            {
                if (i > 6)
                    temp = 5
            }
```

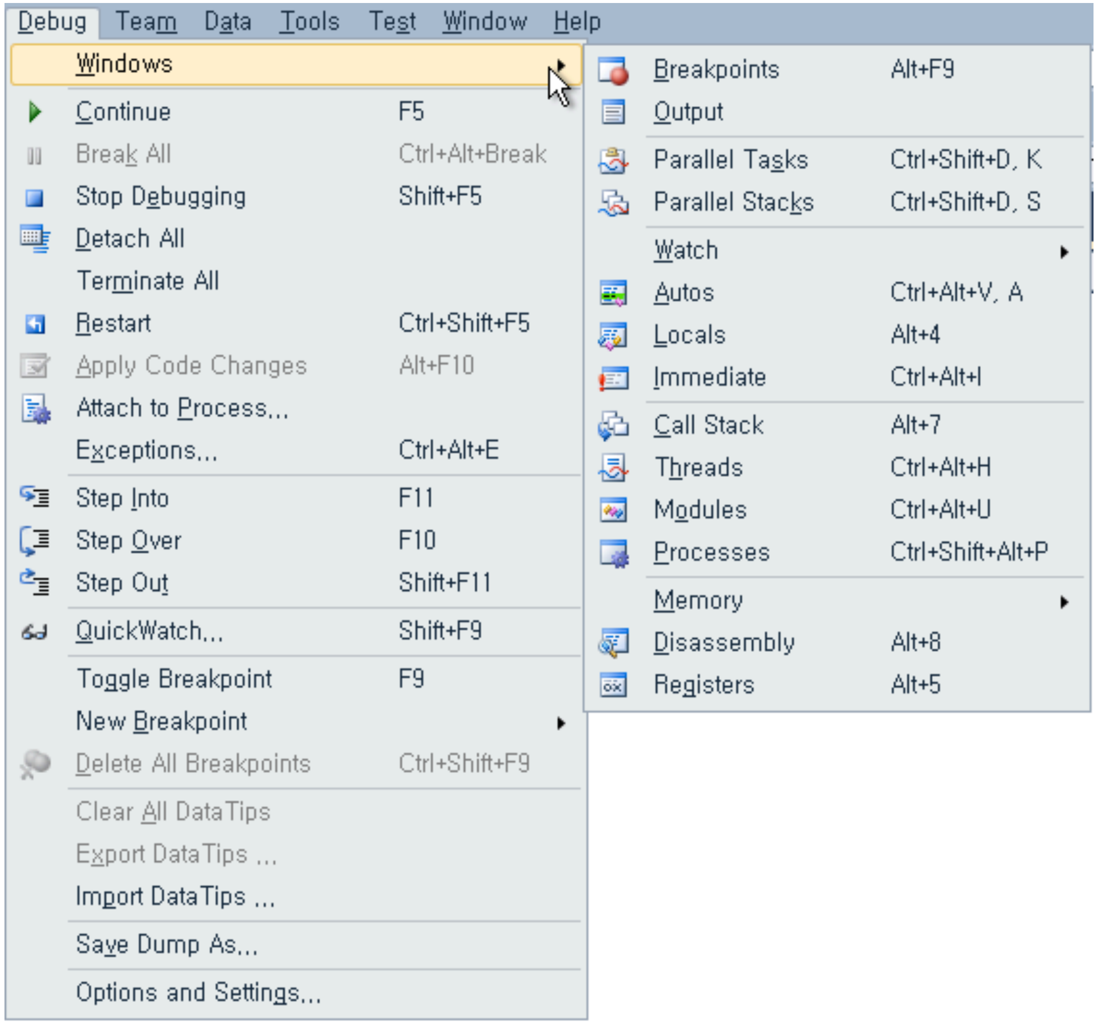
i 4

i 6

<Fig. 24> Change Value Within Data Tip

Debug Windows

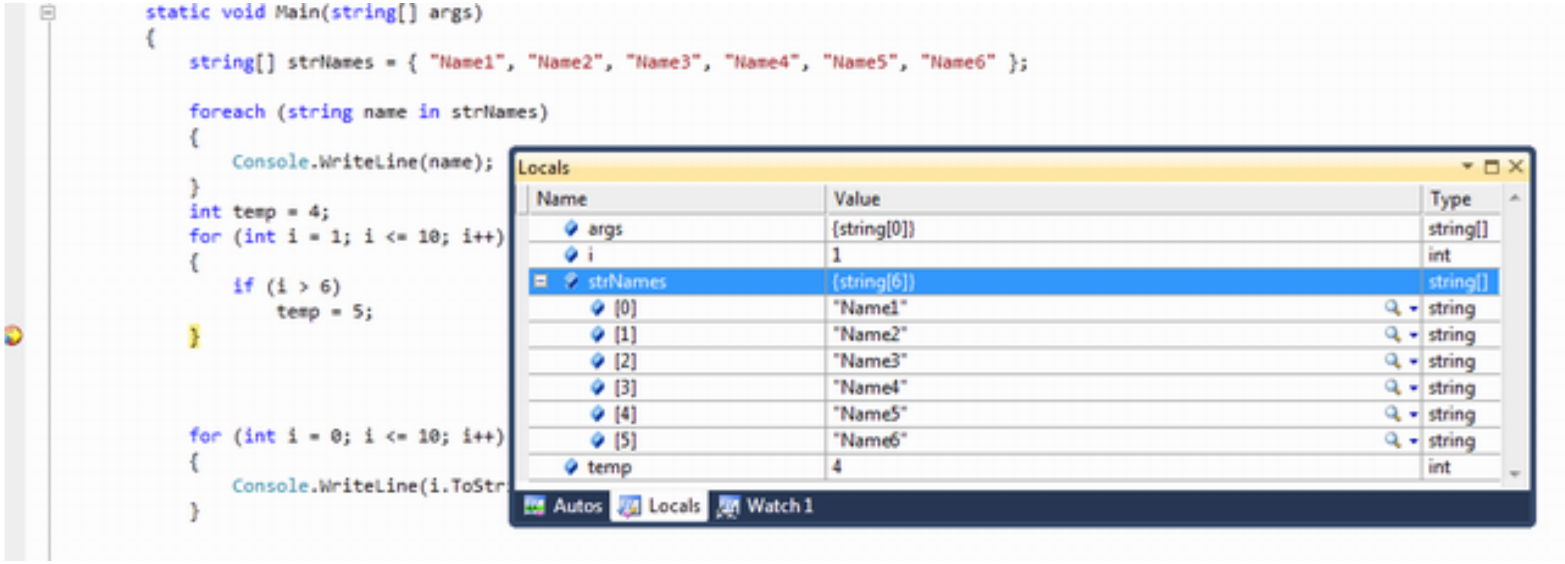
- ❑ Investigation window
  - Local
  - Autos
  - Watch
- ❑ Immediate window
- ❑ Call stack



<Fig. 25> Debug windows

Locals

- ❑ Automatically displays the list of variables within the scope of current methods
- ❑ Current scope object variable along with the value



```
static void Main(string[] args)
{
    string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

    foreach (string name in strNames)
    {
        Console.WriteLine(name);
    }
    int temp = 4;
    for (int i = 1; i <= 10; i++)
    {
        if (i > 6)
            temp = 5;
    }



    for (int i = 0; i <= 10; i++)
    {
        Console.WriteLine(i.ToStr
    }
```

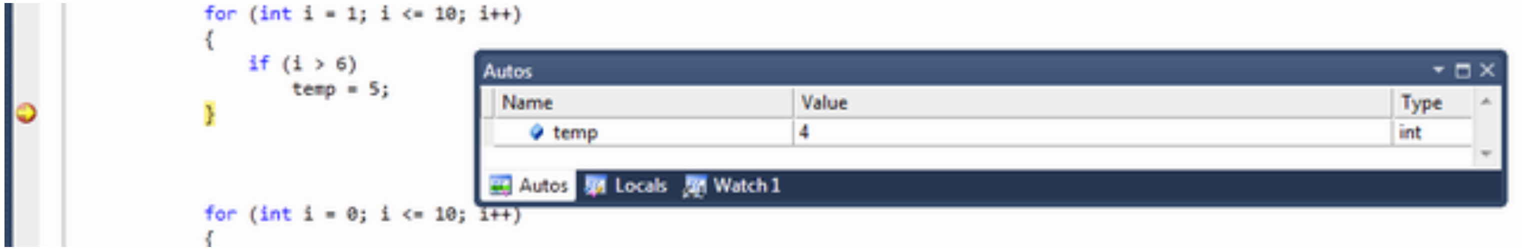
Locals

Name	Value	Type
args	{string[0]}	string[]
i	1	int
⊟ strNames	{string[6]}	string[]
[0]	"Name1"	string
[1]	"Name2"	string
[2]	"Name3"	string
[3]	"Name4"	string
[4]	"Name5"	string
[5]	"Name6"	string
temp	4	int

〈Fig. 26〉 Local Variables

Autos

- ❑ Automatically detect by the VS debugger during the debugging
- ❑ Visual Studio determines which objects or variables are important for the current code statement



```
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}


for (int i = 0; i <= 10; i++)
{
```

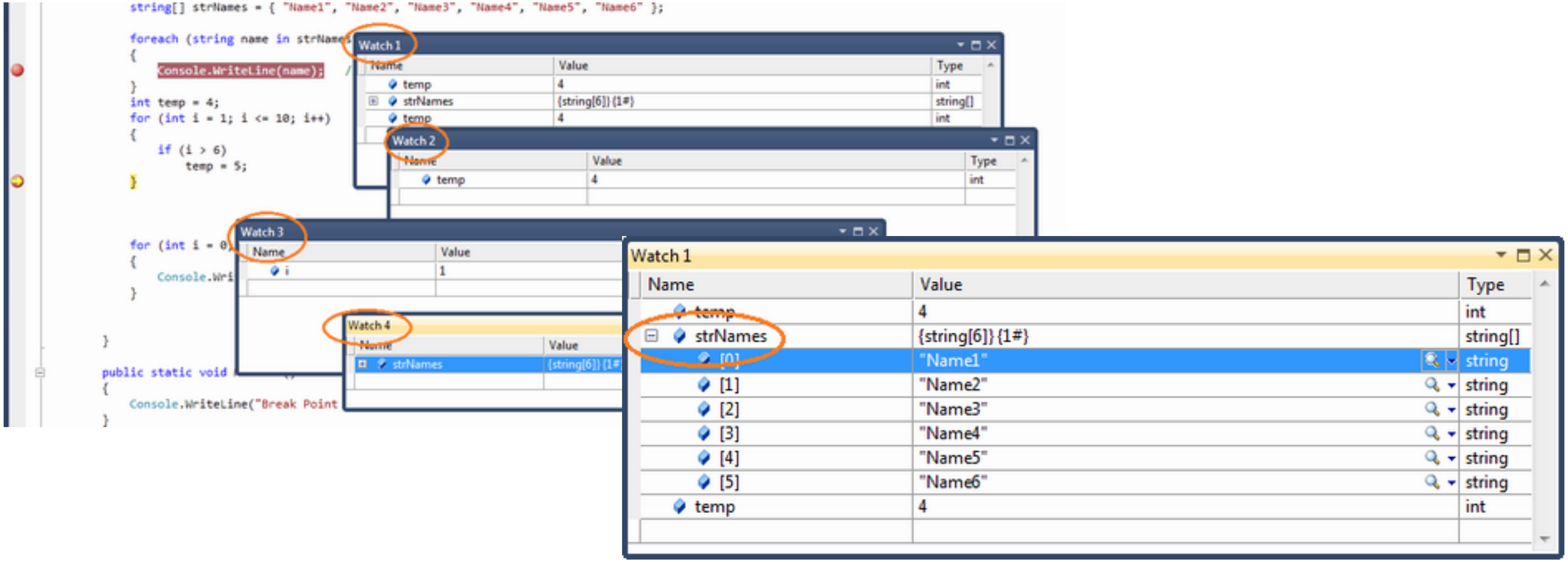
Autos

Name	Value	Type
temp	4	int

〈Fig. 27〉 Autos - [Ctrl + D + A]

Watch

- ❑ It displays variables that you have added
- ❑ There are 4 different watch windows available
- ❑ You can have “+” symbol with the variable to explore the properties and member of that object variable



```
        string[] strNames = { "Name1", "Name2", "Name3", "Name4", "Name5", "Name6" };

        foreach (string name in strNames
        {
            Console.WriteLine(name);
        }
        int temp = 4;
        for (int i = 1; i <= 10; i++)
        {
            if (i > 6)
                temp = 5;
        }


        for (int i = 0
        {
            Console.Wri
        }

    }

    public static void
    {
        Console.WriteLine("Break Point
    }
```

Watch 1

Name	Value	Type
temp	4	int
⊞ strNames	{string[6]} {1#}	string[]
temp	4	int

Watch 2

Name	Value	Type
temp	4	int

Watch 3

Name	Value
i	1

Watch 4

Name	Value
⊞ strNames	{string[6]} {1#

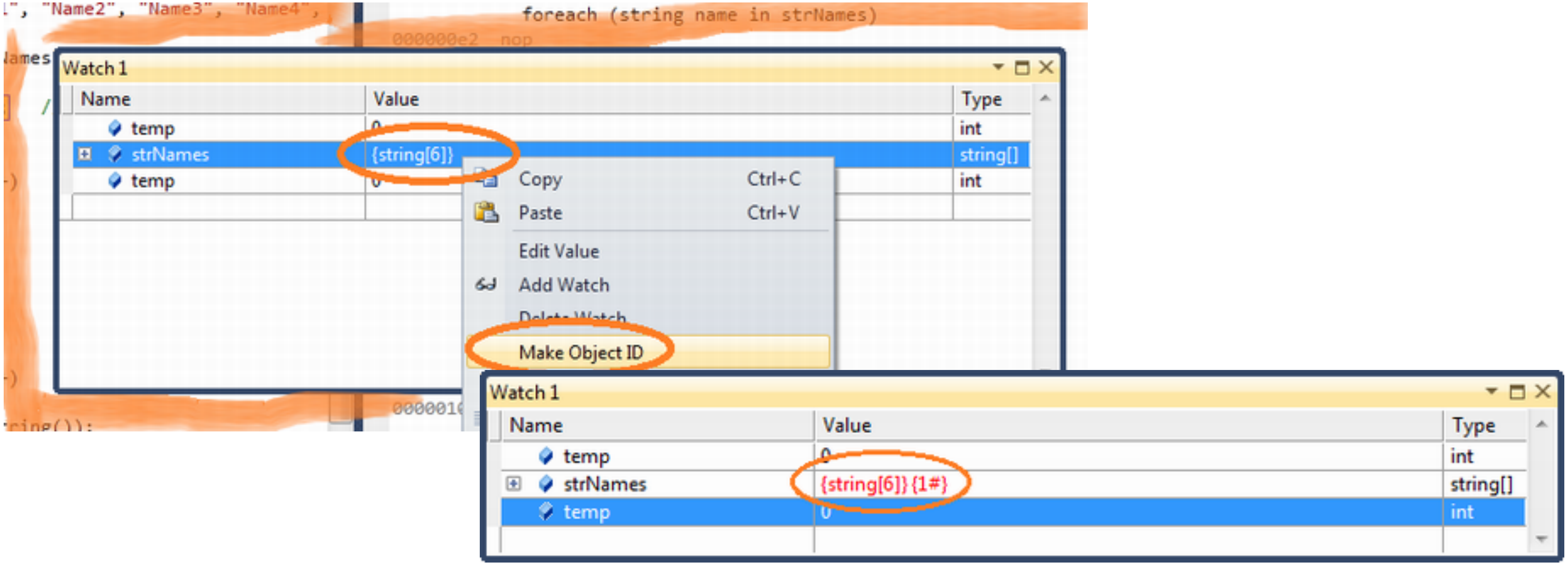
Watch 1

Name	Value	Type
temp	4	int
⊟ strNames	{string[6]} {1#}	string[]
[0]	"Name1"	string
[1]	"Name2"	string
[2]	"Name3"	string
[3]	"Name4"	string
[4]	"Name5"	string
[5]	"Name6"	string
temp	4	int

〈Fig. 28〉 Watch window

Creating object ID

- ❑ You can create an object ID for any particular instance of object
- ❑ When you want to monitor any object at any point of time even if it goes out of scope



```
1", "Name2", "Name3", "Name4",          foreach (string name in strNames)
                                         000000e2  nop
```

Watch 1

Name	Value	Type
temp	0	int
strNames	{string[6]}	string[]
temp	0	int

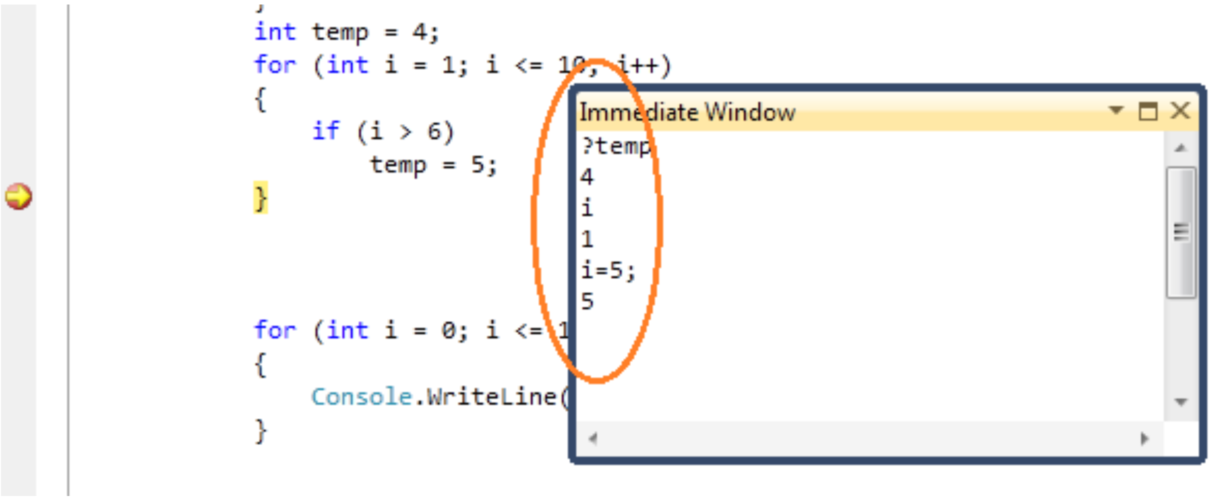
Watch 1

Name	Value	Type
temp	0	int
strNames	{string[6]} {1#}	string[]
temp	0	int

〈Fig. 28〉 Object ID

Immediate Window

- ❑ If you want to change the variable values or execute some statement without impacting your current debugging steps
- ❑ Immediate window gas a set of commands which can be executed any times during debugging
- ❑ You can execute any command or execute any code statement from here



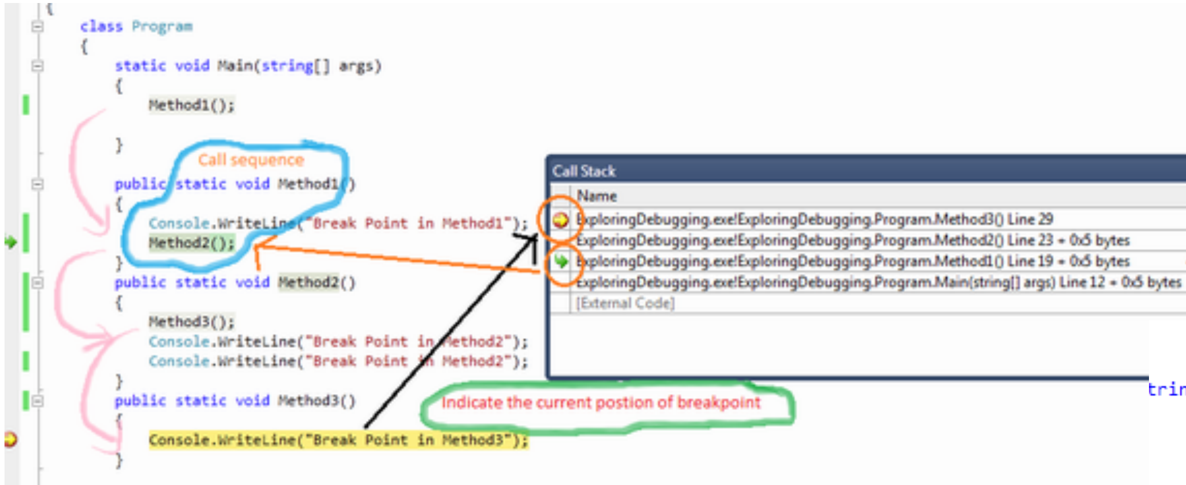
```
int temp = 4;
for (int i = 1; i <= 10; i++)
{
    if (i > 6)
        temp = 5;
}


for (int i = 0; i <= 1
{
    Console.WriteLine(
}
```

```
?temp
4
i
1
i=5;
5
```

〈Fig. 29〉 Basic Immediate Window

Call Stack

- ❑ Call stack window show that current method call nesting
- ❑ Yellow arrow identifies the stack frame where the execution pointer is located

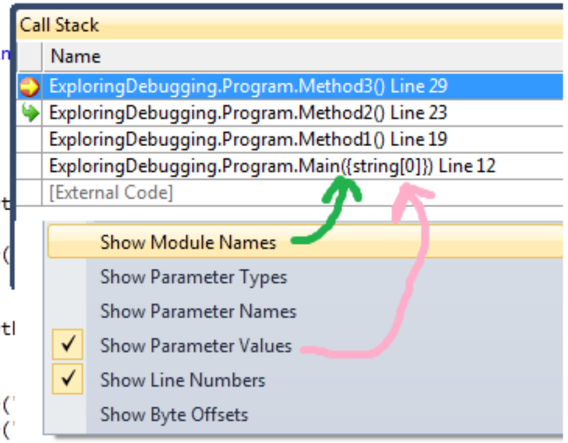


```
{
    class Program
    {
        static void Main(string[] args)
        {
            Method1();

        }
        public static void Method1()
        {
            Console.WriteLine("Break Point in Method1");
            Method2();
        }
        public static void Method2()
        {
            Method3();
            Console.WriteLine("Break Point in Method2");
            Console.WriteLine("Break Point in Method2");
        }
        public static void Method3()
        {
            Console.WriteLine("Break Point in Method3");
        }
```

Call Stack

	Name	Lang
➲	ExploringDebugging.exe!ExploringDebugging.Program.Method3() Line 29	C#
	ExploringDebugging.exe!ExploringDebugging.Program.Method2() Line 23 + 0x5 bytes	C#
➜	ExploringDebugging.exe!ExploringDebugging.Program.Method1() Line 19 + 0x5 bytes	C#
	ExploringDebugging.exe!ExploringDebugging.Program.Main(string[] args) Line 12 + 0x5 bytes	C#
	[External Code]	

<Fig. 30> Call Stack



Call Stack

	Name
➲	ExploringDebugging.Program.Method3() Line 29
➜	ExploringDebugging.Program.Method2() Line 23
	ExploringDebugging.Program.Method1() Line 19
	ExploringDebugging.Program.Main({string[0]}) Line 12
	[External Code]

<Fig. 31> Call Stack Customization

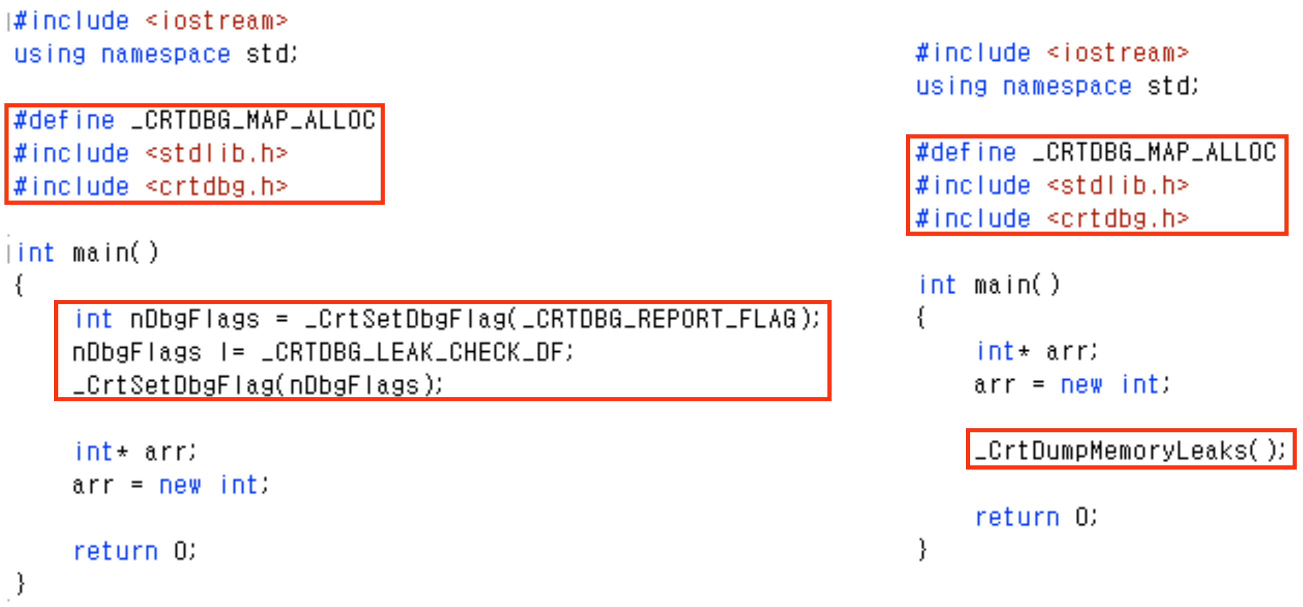
Memory Leaks (1/3)

- ❑ 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
- ❑ 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

Memory Leaks (2/3)

❑ “Hook” into the C Run-time libraries to use the debug heap

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



```
#include <iostream>
using namespace std;

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

int main()
{
    int nDbgFlags = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG);
    nDbgFlags |= _CRTDBG_LEAK_CHECK_DF;
    _CrtSetDbgFlag(nDbgFlags);

    int* arr;
    arr = new int;

    return 0;
}
```

```
#include <iostream>
using namespace std;

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

int main()
{
    int* arr;
    arr = new int;

    _CrtDumpMemoryLeaks();

    return 0;
}
```

〈Fig. 32〉 Setting up for detection for Console or Win32

Memory Leaks (3/3)

❑ _CRTDBG_MAP_ALLOC

- Including crtdbg.h, you map the malloc and free functions to their Debug versions, _malloc_dbg and _free_dbg, whichkeep track of memory allocation and deallocation
- With out **#define _CRTDBG_MAP_AlIOC** :
  - 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

Q & A