

# Debugging Techniques



# Objectives

- Basic Debugging Technique
- Breakpoints
- Watches
- Stepping
- Stopping the Debugger
- Conditions and Hit Counts
- Break on Exception
- Step Into
- Trace and Assert

# Basic Debugging Technique



- The debugger is a tool to help correct runtime and semantic errors
- note that no debugging tools are useful in solving *compiler errors*.
- Compiler errors are those that show at the bottom of the screen when compiling

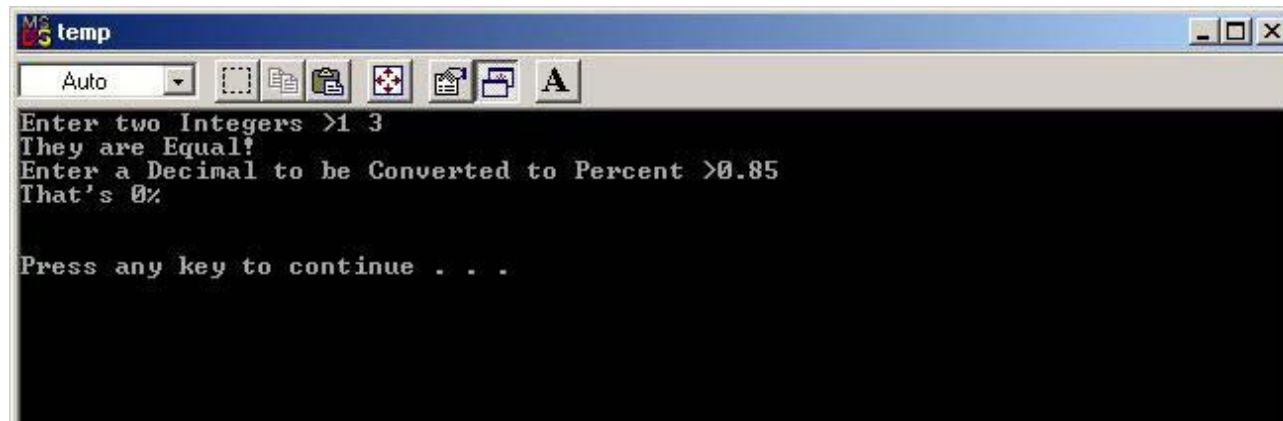
# Basic Debugging Technique

- If the program isn't working correctly, one of two things could be going wrong:
  - ✓ *Data is corrupt somewhere*
  - ✓ The code isn't correct
- Example

```
int a = 0;  
int b = 1;  
printf("%d", (b/a));
```

# A Buggy Program

- Trying to debug a program that's working perfectly is rather pointless



```
temp
Auto
Enter two Integers >1 3
They are Equal!
Enter a Decimal to be Converted to Percent >0.85
That's 0%

Press any key to continue . . .
```

# The Buggy Code

```
#include <stdio.h>
int toPercent (float decimal);
int main() {
    int a, b;
    float c;
    int cAsPercent;
    printf("Enter A >");
    scanf("%d", &a);
    printf("Enter B >");
    scanf("%d", &b);

    if (a = b) printf("They are Equal!\n");
    else if (a > b) printf("The first one is bigger!\n");
    else printf("The second one is bigger!\n");
    printf("Enter a Decimal to be Converted to Percent >");
    scanf("%f", &c);
    cAsPercent = toPercent(c);
    printf("That's %d %%\n", cAsPercent);
    printf("\n\n");
    return 0;
}
```

# The Buggy Code

```
/* ToPercent():  
Converts a given float (eg 0.9) to a percentage (90).  
*/  
int toPercent (float decimal) {  
    int result;  
    result = int(decimal) * 100;  
    return result;  
}
```

# Debug Mode or Not?

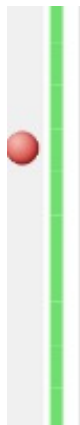


- Ctrl+F5 to run your program
- The F5 key alone will also run in debug mode.
- Build for Debug
- Build for Release



# Breakpoints

- Breakpoints are the lifeblood of debugging.
- Right-click and select "Insert/Remove Breakpoint" or press the F9 key

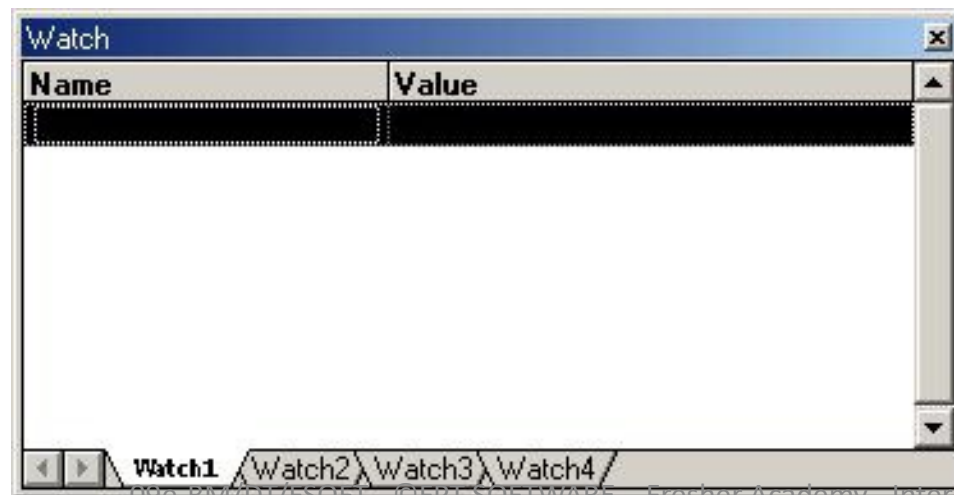


```
printf("Enter A >");
scanf("%d", &a);
printf("Enter B >");
scanf("%d", &b);

if (a = b) printf("They are Equal!\n");
else if (a > b) printf("The first one is bigger!\n");
else printf("The second one is bigger!\n");
printf("Enter a Decimal to be Converted to Percent >");
scanf("%f", &c);
```

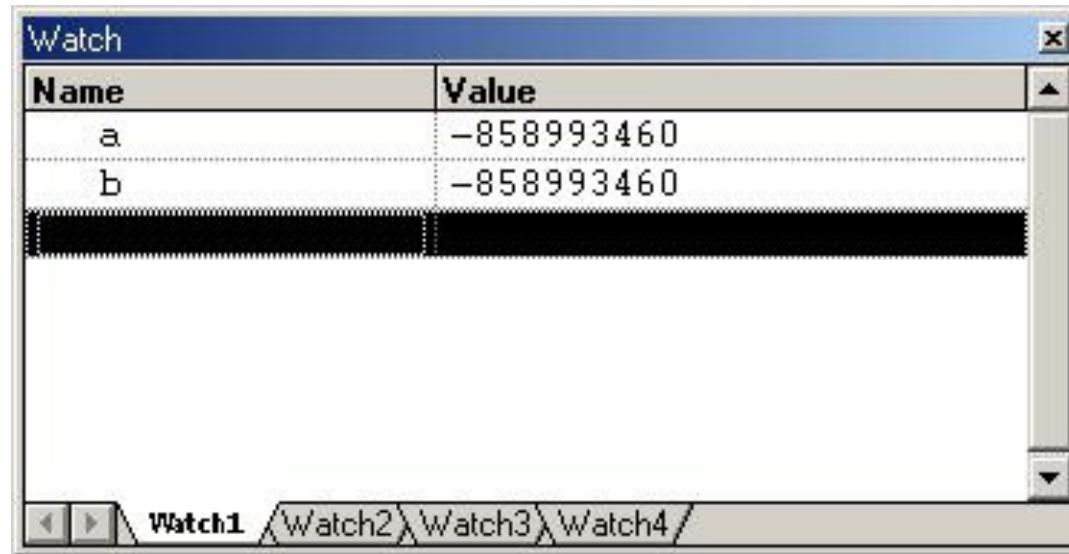
# Watches

- The "Watch" window lets you *watch the contents of any variables you select as your program* executes.
- Open it from the View menu (Debug Windows > Watch), or by clicking the "Watch" icon in the toolbar, or by pressing Alt+3



# Watches

- Enter to add *variables to your Watch* list:



# Watches

- Watch a range of values inside array:  
Syntax: array + <offset>, <range>

```
1 #include <iostream>
2 #include <stdexcept>
3
4 using namespace std;
5
6 void main()
7 {
8     int* arr = new int[1000];
9
10    for(int i = 0; i < 1000; i++)
11    {
12        arr[i] = i;
13    }
14 }
```

Watch 1

Name	Value
arr, 3	0x00801290
[0]	0
[1]	1
[2]	2
arr + 3, 3	0x0080129c
[0]	3
[1]	4
[2]	5


# Stepping

- Step Over F10
- Step Into F11 (Some code inside a function *may or may not need to be examined*)
- Step Out Shift + F11



# Stepping

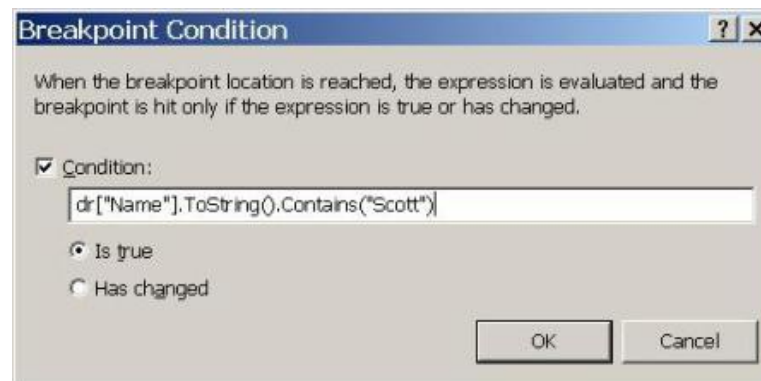
- When you are tired of stepping through the code, F5 resumes execution.



```
printf("Enter A >");  
scanf("%d", &a);  
printf("Enter B >");  
scanf("%d", &b);  
  
if (a = b) printf("They are Equal!\n");  
else if (a > b) printf("The first one is bigger!\n");  
else printf("The second one is bigger!\n");  
printf("Enter a Decimal to be Converted to Percent >");  
scanf("%f", &c);
```

# Stopping the Debugger

When you've found a problem to correct, it may be tempting to press Ctrl+C in your program window to end the program



Select "Stop Debugging" from the Debug menu or on the toolbar or press Shift+F5.

# Conditions and Hit Counts

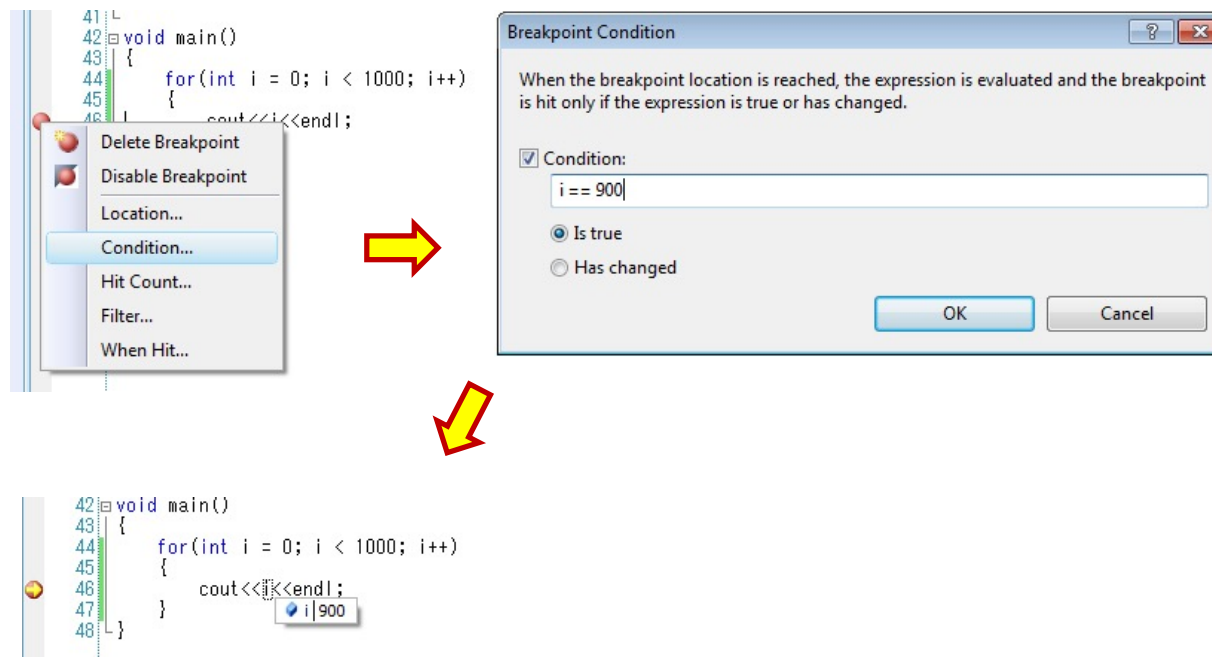


- Breakpoint can use conditions and hit counts
- Conditions and hit counts are useful if you don't want the debugger to halt execution *every time the program reaches the breakpoint*
- Only when a condition is true, or a condition has changed, or execution has reached the breakpoint a specified number of times.



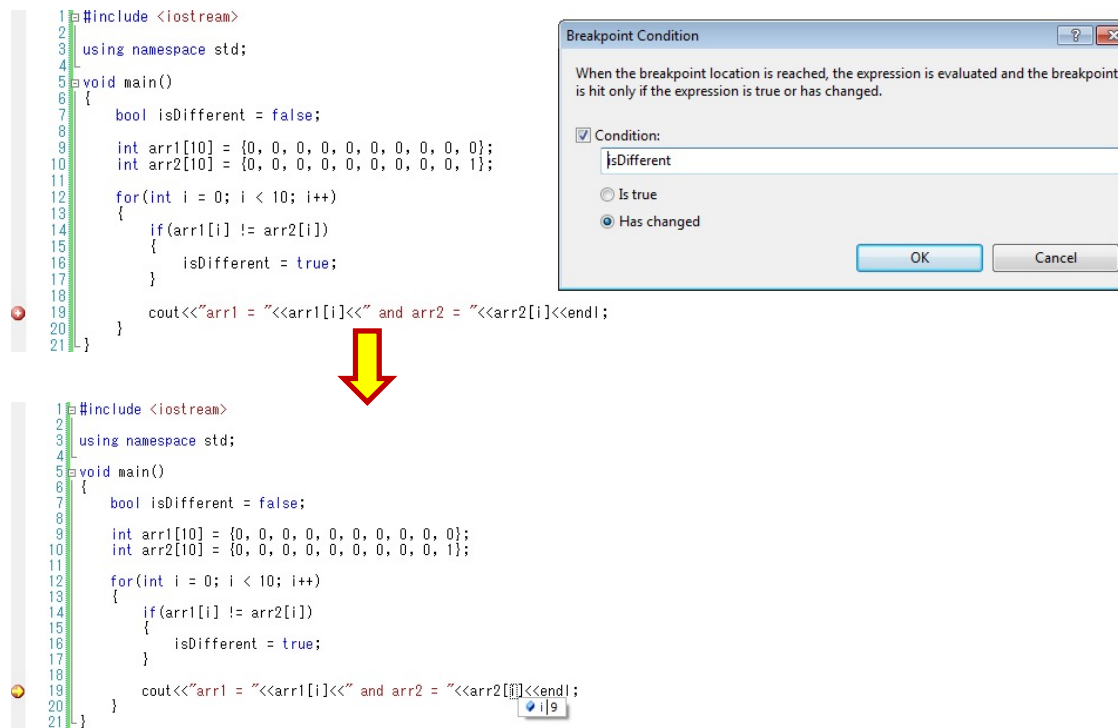
# Conditions and Hit Counts

- Condition: Is true



# Conditions and Hit Counts

- Condition: Has changed



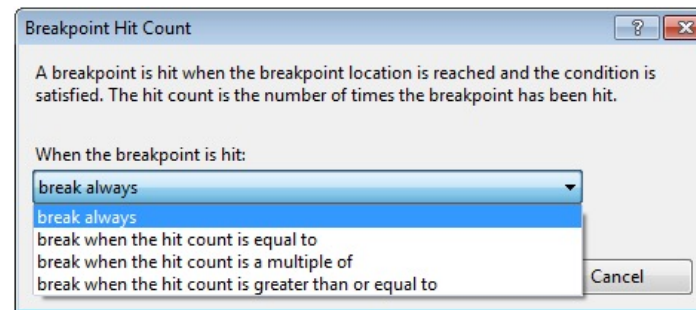
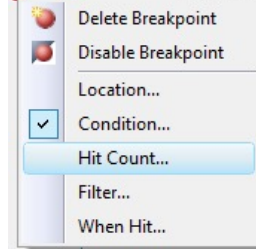
The screenshot shows a C++ code editor with a breakpoint set on line 19, which is a `cout` statement. A red arrow points to this breakpoint. A 'Breakpoint Condition' dialog box is open, showing the condition `isDifferent` and the 'Has changed' option selected. The code in the editor is as follows:

```
1 #include <iostream>
2
3 using namespace std;
4
5 void main()
6 {
7     bool isDifferent = false;
8
9     int arr1[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
10    int arr2[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 1};
11
12    for(int i = 0; i < 10; i++)
13    {
14        if(arr1[i] != arr2[i])
15        {
16            isDifferent = true;
17        }
18
19        cout<<"arr1 = "<<arr1[i]<<" and arr2 = "<<arr2[i]<<endl;
20    }
21 }
```

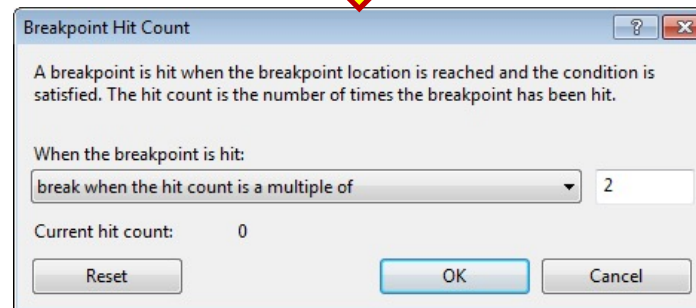
# Conditions and Hit Counts

- Hit Count: is a multiple of

```
42 void main()  
43 {  
44     for(int i = 0; i < 1000; i++)  
45     {  
46         cout<<i<<endl;
```



```
42 void main()  
43 {  
44     for(int i = 0; i < 1000; i++)  
45     {  
46         cout<<i<<endl;  
47         i|2  
48     }  
42 void main()  
43 {  
44     for(int i = 0; i < 1000; i++)  
45     {  
46         cout<<i<<endl;  
47         i|4  
48     }
```



# Break on Exception

The screenshot illustrates the process of setting a breakpoint on an exception in Visual Studio. The 'Exceptions' dialog box is open, showing the 'Break when an exception is:' section. The 'std::exception' is selected in the list, and the 'Thrown' checkbox is checked. The 'Debug' menu is visible, with a yellow arrow pointing to the 'Exceptions...' option. Another yellow arrow points from the 'Exceptions...' option to the 'Exceptions' dialog box. The code editor shows a C++ program with a try-catch block. The exception is thrown from myFunction() and caught in callMyFunction().

Exceptions

Break when an exception is:

Name	Thrown	User-unhandled
C++ Exceptions	<input type="checkbox"/>	<input checked="" type="checkbox"/>
_com_error	<input type="checkbox"/>	<input checked="" type="checkbox"/>
ATL::CAtlException	<input type="checkbox"/>	<input checked="" type="checkbox"/>
CException	<input type="checkbox"/>	<input checked="" type="checkbox"/>
std::exception	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
void	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Common Language Runtime Exceptions	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Managed Debugging Assistants	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Native Run-Time Checks	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Win32 Exceptions	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Debug

Exceptions... Ctrl+D, E

```
1 #include <iostream>
2 #include <stdexcept>
3
4 using namespace std;
5
6 void myFunction()
7 {
8     throw exception("Throw exception from myFunction()!");
9 }
10
11 void callMyFunction()
12 {
13     try
14     {
15         myFunction();
16     }
17     catch(exception ex)
18     {
19         cout<<ex.what()<<endl;
20     }
21 }
22
23 void main()
24 {
```

Microsoft Visual Studio

First-chance exception at 0x76a2c41f in testttttt.exe: Microsoft C++ exception: std::exception at memory location 0x0051f8cc..

Break Continue Ignore

# Stepping Into Assembly

- Be careful when you "Step Into" lines involving printf, scanf, or other system functions!

```
int __cdecl scanf (  
    const char *format,  
    ...  
)  
{  
    va_list arglist;  
    va_start(arglist, format);  
    return vscanf(_input_1, format, NULL, arglist);  
}
```

# Debug commands

Command	Meaning
<b>Ctrl+F5</b>	Run program
<b>F5</b>	Run in debug mode
<b>F9</b>	Create breakpoint
<b>F10</b>	Step over
<b>F11</b>	Step into
<b>Shift + F11</b>	Step out
<b>Shift + F5</b>	Stop debugging
<b>Ctrl + Tab</b>	Change window

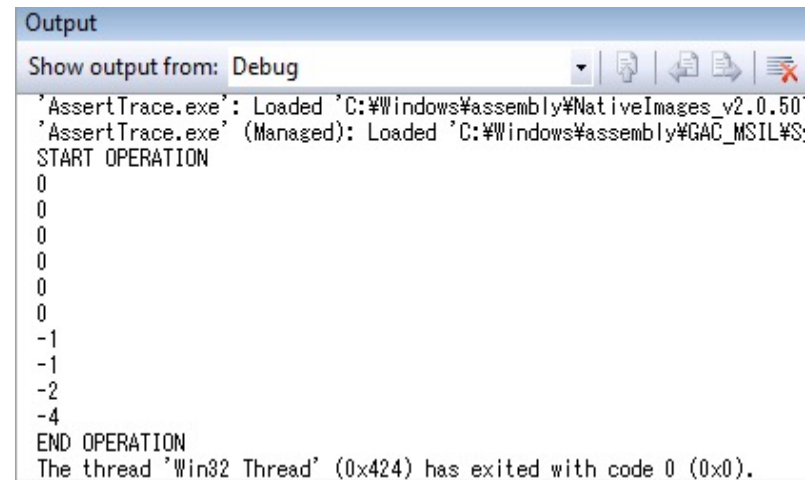
# Trace and Assert

- Trace: Allows the programmer to put a log message onto the main output window
- Assert: To check program assumptions

# Trace and Assert

```
3  #include "stdafx.h"
4
5  using namespace System::Diagnostics;
6
7  void main()
8  {
9      double result = 0.0;
10
11     Trace::WriteLine("START OPERATION");
12     for(int i = 0; i < 10; i++)
13     {
14         int numToDevide   = i - 10;
15         int numToBeDevided = i;
16
17         Trace::WriteLine(result);
18
19         result = numToBeDevided / numToDevide;
20     }
21     Trace::WriteLine("END OPERATION");
22 }
```

Press  
F5



The screenshot shows the 'Output' window in Visual Studio with the filter set to 'Debug'. The output text is as follows:

```
Output
Show output from: Debug
'AssertTrace.exe': Loaded 'C:\Windows\assembly\NativeImages_v2.0.50
'AssertTrace.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL#S
START OPERATION
0
0
0
0
0
0
-1
-1
-2
-4
END OPERATION
The thread 'Win32 Thread' (0x424) has exited with code 0 (0x0).
```

- ❑ Keep tracing code processing by output value during debugging

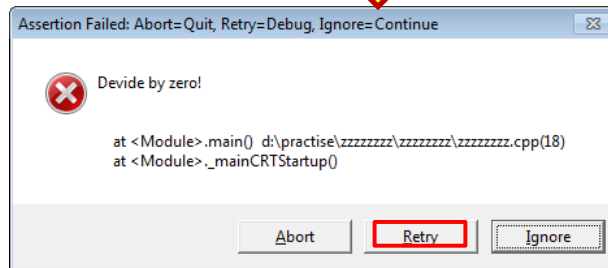


# Trace and Assert

```
3 | #include "stdafx.h"
4 |
5 | using namespace System::Diagnostics;
6 |
7 | void main()
8 | {
9 |     double result = 0.0;
10 |
11 |     Trace::WriteLine("START OPERATION");
12 |     for(int i = 0; i < 10; i++)
13 |     {
14 |         int numToDevide    = i - 10;
15 |         int numToBeDevided = i;
16 |
17 |         Trace::Assert(numToBeDevided != 0, "Devide by zero!");
18 |         Trace::WriteLine(result);
19 |
20 |         result = numToBeDevided / numToDevide;
21 |     }
22 |     Trace::WriteLine("END OPERATION");
23 | }
```

This code contain potential bug, if another developer change 10 to other values (such as 11)

We use Assert to validate that the value is valid or not



Press Retry allow us to debug after Assert

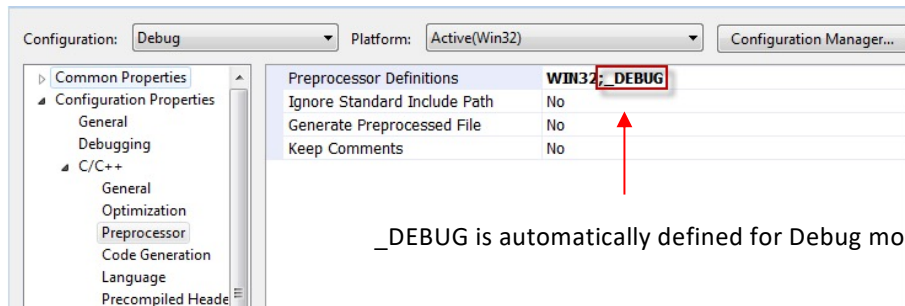
```
3 | #include "stdafx.h"
4 |
5 | using namespace System::Diagnostics;
6 |
7 | void main()
8 | {
9 |     double result = 0.0;
10 |
11 |     Trace::WriteLine("START OPERATION");
12 |     for(int i = 0; i < 11; i++)
13 |     {
14 |         int numToDevide    = i - 10;
15 |         int numToBeDevided = i;
16 |
17 |         Trace::Assert(numToBeDevided != 0, "Devide by zero!");
18 |         Trace::WriteLine(result);
19 |
20 |         result = numToBeDevided / numToDevide;
21 |     }
22 |     Trace::WriteLine("END OPERATION");
23 | }
```

# Trace and Assert



- The behavior for Trace will not change between a debug and a release build
- This mean that we must `#ifdef` any Trace-related code to prevent debug behavior in a release build

# Trace and Assert



```
3 #include "stdafx.h"
4 using namespace System::Diagnostics;
5
6 void main()
7 {
8     double result = 0.0;
9
10    #ifdef _DEBUG
11    Trace::WriteLine("START OPERATION");
12    #endif
13
14    for(int i = 0; i < 10; i++)
15    {
16        int numToDevide = i - 10;
17        int numToBeDevided = i;
18
19    #ifdef _DEBUG
20    Trace::WriteLine(result);
21    #endif
22
23        result = numToBeDevided / numToDevide;
24    }
25    #ifdef _DEBUG
26    Trace::WriteLine("END OPERATION");
27    #endif
28 }
29
30 }
```

We can use #ifdef \_DEBUG to prevent debug behavior in release mode

# Thank you

Q&A

