

# Debugging Techniques

Instructor:

© FPT Software

1

Latest updated by: HanhTT1

## Agenda

- 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

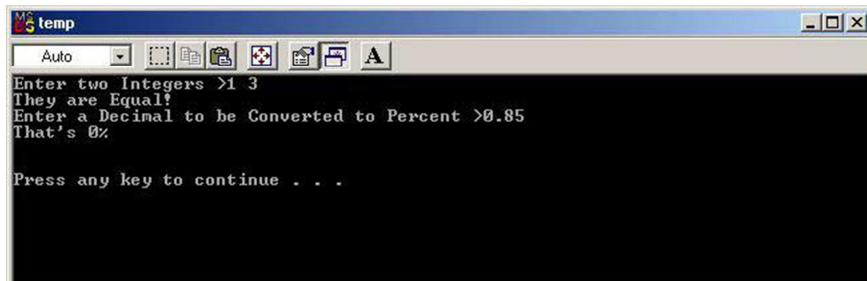
## Basic Debugging Technique

- 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



## 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");
    getchar();
    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

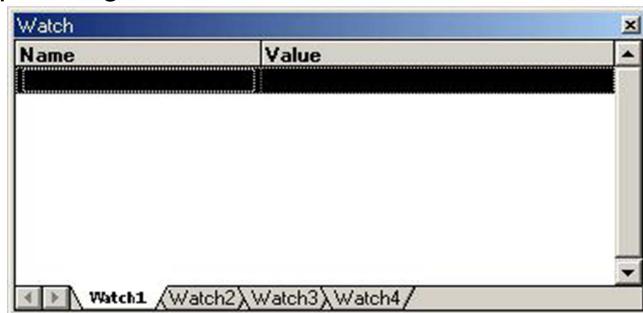
- Breakpoints are the lifeblood of debugging.
- Right-click and select "Insert/Remove Breakpoint" or press the F9 key



```
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

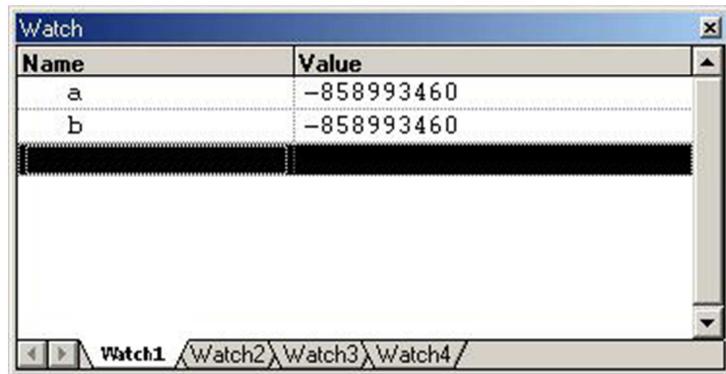


© FPT Software

10

## Watches

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



© FPT Software

11

## Watches

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

The screenshot shows a code editor with the following C++ code:

```
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 }
```

Below the code editor is a "Watch 1" window displaying the memory state:

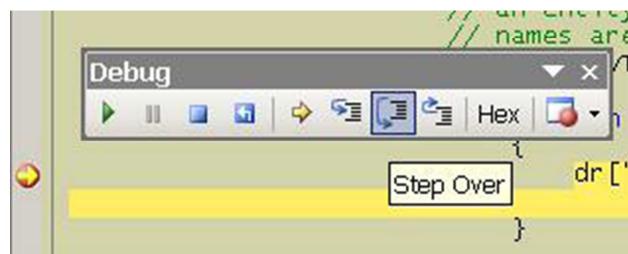
Name	Value
arr, 3	0x00801290
[0]	0
[1]	1
[2]	2
arr + 3, 3	0x0080129c
[0]	3
[1]	4
[2]	5

© FPT Software

12

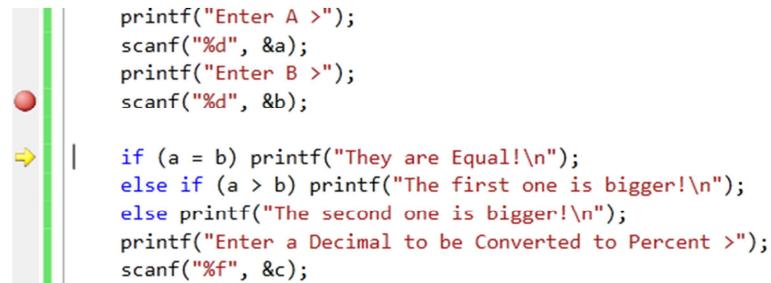
## Stepping

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



## Stepping

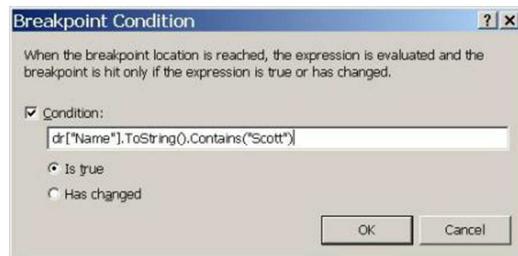
- When you are tired of stepping through the code, F5 resumes execution.



```
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.

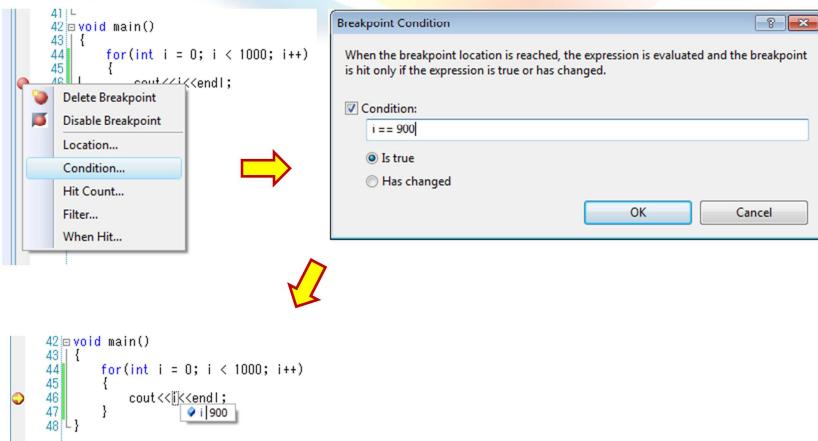
## Conditions and Hit Counts

- 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 are useful when setting breakpoints inside of a loop. For example, if your code iterates through a collection of Customer objects with a for each loop, and you want to break on the 10th iteration of the loop, you can specify a hit count of 10. If something bad only happens when the Customer object's Name property is equal to "Scott", you can right click the breakpoint red glyph, select Condition from the context menu, and enter the expression `customer.Name == "Scott"` into the breakpoint condition textbox. Intellisense is available in this textbox to ensure you are using the correct syntax.

## Conditions and Hit Counts

- Condition: Is true



© FPT Software

17

## Conditions and Hit Counts

- Condition: Has changed

The screenshot shows a debugger interface with a code editor and a 'Breakpoint Condition' dialog box.

**Code Editor (Top):**

```
#include <iostream>
using namespace std;
void main()
{
    bool isDifferent = false;
    int arr1[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    int arr2[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 1};
    for(int i = 0; i < 10; i++)
    {
        if(arr1[i] != arr2[i])
        {
            isDifferent = true;
        }
        cout<<"arr1 = "<<arr1[i]<< " and arr2 = "<<arr2[i]<<endl;
    }
}
```

**Code Editor (Bottom):**

```
#include <iostream>
using namespace std;
void main()
{
    bool isDifferent = false;
    int arr1[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    int arr2[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 1};
    for(int i = 0; i < 10; i++)
    {
        if(arr1[i] != arr2[i])
        {
            isDifferent = true;
        }
        cout<<"arr1 = "<<arr1[i]<< " and arr2 = "<<arr2[i]<<endl;
    }
}
```

**Breakpoint Condition Dialog:**

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:  
isDifferent

Is true  
 Has changed

OK Cancel

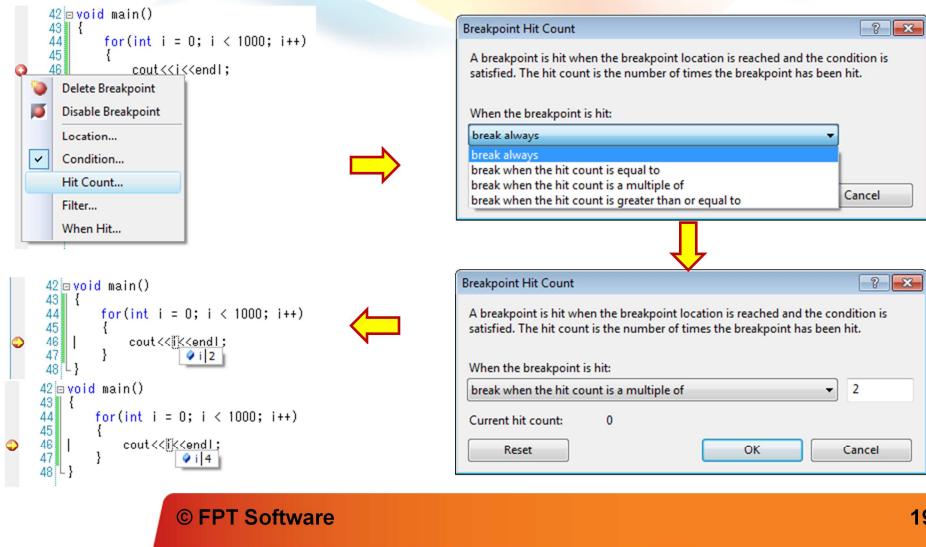
A yellow arrow points from the bottom code editor to the top code editor, indicating a change.

© FPT Software

18

## Conditions and Hit Counts

- Hit Count: is a multiple of



© FPT Software

19

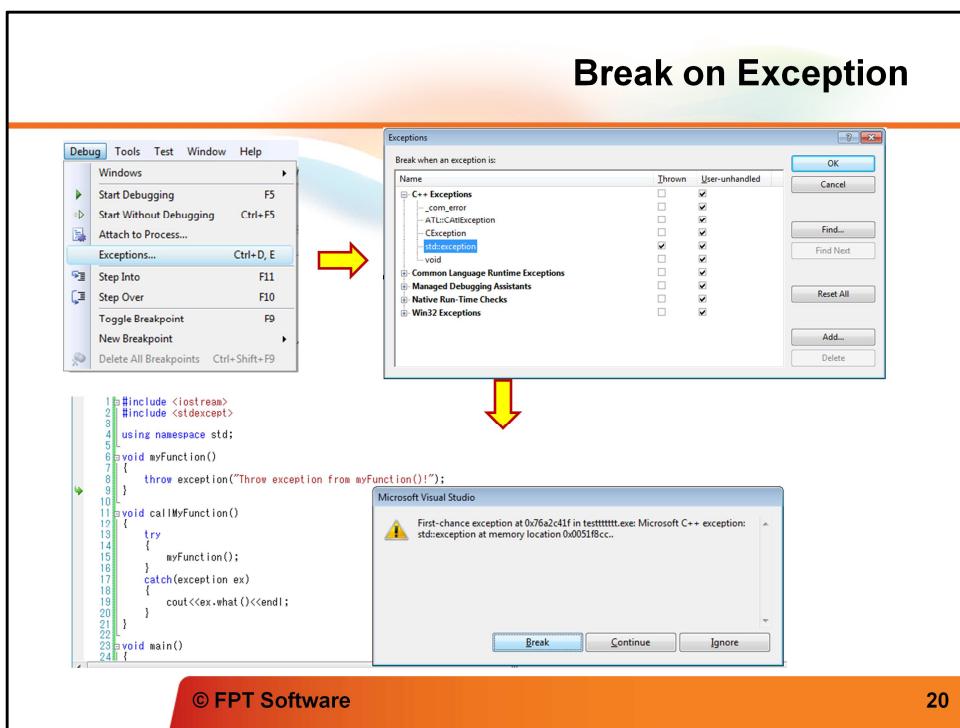
The Hit Count is used along with Condition (use same condition as previous slide: i – Has changed)

Take an example of Hit Count: break when the hit count is a multiple of 2

So, when we press F5, instead of press F5 4 times (if we choose break always) we just press F5 2 times (the number of multiple of 2: 2, 4, 6, 8...)

In this example that is 2, 4

## Break on Exception



© FPT Software

20

## 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_l, 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

The screenshot shows a code editor with C++ code and a debugger window. A yellow callout box points to the F5 button with the text "Press F5". The code uses `Trace::WriteLine` to print values during execution. The output window shows the results of the loop iterations.

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

Output:

```
Show output from: Debug
'AssertTrace.exe': Loaded 'C:\Windows\Assembly\NativeImages_v2_0_0_0\AssertTrace.exe' (Managed). Loaded 'C:\Windows\Assembly\GAC_MSIL\V...
START OPERATION
0
0
0
0
0
-1
-1
-2
-4
END OPERATION
The thread 'Win32 Thread' (0x424) has exited with code 0 (0x0).
```

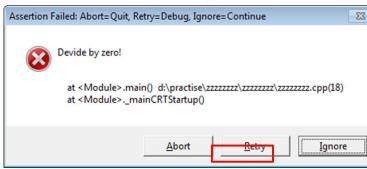
- ❑ Keep tracing code processing by output value during debugging

## Trace and Assert

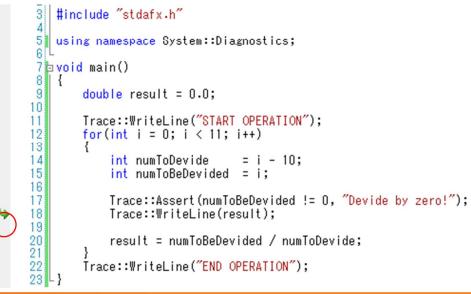
```
3 #include "stdafx.h"
4
5 using namespace System::Diagnostics;
6
7 void main()
8 {
9     double result = 0.0;
10
11    Trace::WriteLine("START OPERATION");
12    for(int i = 0; i < 11; i++)
13    {
14        int numToBeDivide = i - 10;
15        int numToBeDivided = i;
16
17        Trace::Assert(numToBeDivided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



© FPT Software

25

## Trace and Assert

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

## Trace and Assert

The screenshot shows the Visual Studio Properties Manager for a project named "Common Properties". Under the "C/C++" category, the "Preprocessor Definitions" section lists "WIN32; DEBUG". A red arrow points from this entry to the text "DEBUG is automatically defined for Debug mode".

`#include "stdafx.h"  
using namespace System::Diagnostics;  
void main()  
{  
 double result = 0.0;  
#ifdef _DEBUG  
 Trace::WriteLine("START OPERATION");  
#endif  
 for(int i = 0; i < 10; i++)  
 {  
 int numToDevide = i - 10;  
 int numToBeDivided = i;  
#ifdef _DEBUG  
 Trace::WriteLine(result);  
#endif  
 result = numToBeDivided / numToDevide;  
 }  
#ifdef _DEBUG  
 Trace::WriteLine("END OPERATION");  
#endif  
}`

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

© FPT Software

27



© FPT Software

28