What is Debugging?

Debugging is the process by which developers locate and resolve defects in their programs. Needless to say, none of us write perfect code and as such being comfortable with the debugging process is a critical job skill for developers.

We cannot understate the importance of learning the debugger. You are unemployable as a developer if you do not learn this skill!

The Visual Studio IDE has a suite of very powerful debugging and performance monitoring tools embedded in it. Entire books have been written on the topic but in this lesson, we will cover the basics that every beginner should have in their tool belt.

Error, Exception, What's the Difference?

Nothing conceptually. In C#, all error types derive from the Exception type. So, C# developers will use the words exception and error interchangeably. However, exception is the proper term.

Compile Time, Run Time, and Logic Errors

Before we get started with debugging, we must recognize that there are three basic types of errors our programs can encounter: compile time, run time, and logic errors.

Compile time errors are the most obvious because Visual Studio will bring them to your attention and not build or run your solution while any exist in your code. Compile errors are due to inaccuracies in your code such as:

Trying to assign an invalid type (ex: string to an int)

Forgetting a required symbol (ex: ;, "", '', {})

Trying to use a variable that is not declared

Trying to access a type member that is inaccessible or does not exist. (Usually due to a misspelling, namespace issue, or a member that is not marked as public)

A run time error only occurs while your program is running. An example of this is a DivideByZeroException. Consider this method:

public decimal Divide(decimal a, decimal b)

{

return a / b;

}

This method will compile just fine, dividing two decimals is completely valid. However, if you were to prompt a user for the parameters and they entered a 0 as the divisor, it would cause the code to fail at run time. When this happens in Visual Studio, the program will pause on the line that caused the error and information about the type of error will be given (and often suggestions on how to fix it).

The last type of error is the reason we need good debugging tools. Logic errors are when the program does not have any compile time or run time errors, however the program logic is incorrect. An example of this would be if your Add() method returned a value of 5 for 2+2 or if your payroll program deducted the wrong state tax amounts for the state of New York. The code itself is syntactically correct, but the behavior of the program is not correct.

The Error List Window

The Error List window is the place we want to check for compile time errors. This window defaults to the bottom dock of Visual Studio and typically will appear when you attempt to build a solution that has compile time errors in it. If for some reason you should close or lose this window, you can access it via the View menu.

To see this window in action, let's write an invalid statement:

Console.WriteLine("Forgot a semi colon!")

Oops! We forgot a semi colon at the end of our statement. Visual Studio is pretty good about checking for syntax errors like this before you even try to build. If you open the Error List you will see information about the compile time error:

Contents of the error list window, described below

The Description column of the error list will provide the message about the error that occurred. In this case, the description reads: "; expected". That is to say the compiler expected a semi colon and did not find one. It also tells us that the error is in the ConsoleApplication1 project in the Program.cs file on or around line 13.

Note: The line number is not always accurate, but it is usually in the vicinity of the error.

The Code column generally contains a clickable link which can take you to more information about this type of error, what it means, and how to fix it. Sometimes clicking this link will simply "Google It" for you. In fact, when you receive a compile time error you do not understand the first thing you should do is perform an internet search based on part or all of the error description.

You may also notice that the error list window has tabs for warnings as well. Warnings are merely suggestions for your code and do not necessarily have to be followed. A common warning is "unreachable code detected" which occurs when the compiler detects that due to a return statement in a method that some code will always be skipped. We can cause this warning fairly easily:

int a = 5;

int b = 10;

return;

int sum = a + b;

The line of code involving the sum will never be executed under any circumstance. The code will compile, because this code is syntactically correct, but a warning will appear because the compiler thinks (and is probably correct) that the above code is not what you truly intended to happen.

Break Points

Not all errors are as polite as compile-time errors. Run-time and logic errors don't show up in the Error List Window. To troubleshoot them, it's convenient to inspect our programs while they're running. Let's start with a loop that contains a common run-time error:

int[] numbers = new int[] { 3, 1, 4 };

for(int i = 0; i <= numbers.Length; i++)

{

Console.WriteLine($"The number at index {i} is {numbers[i]}.");

}

When we compile this code, there are no errors. When we run it, we see three messages in the Console. Then our program crashes with an IndexOutOfRangeException.

Since the code kind of works, let's walk through it step-by-step as it's running to find the problem. We need three things to make this happen. First, we need a breakpoint. A breakpoint tells the debugger to pause on a specific line of code as it's running. (More on the debugger later.) In Visual Studio, we set a breakpoint on any executable line of code by highlighting that line and pressing F9 or clicking the "gutter" (the left-most part of the code window) next to the line. A breakpoint appears as a specially highlighted line of code with an indicator in the gutter.

A breakpoint highlighted in red

Second, we Debug our application. This is simply running our application with the C# debugger attached. The debugger is a special program that knows how to pause and restart an application. It also knows how to map specially compiled IL code to its source. Taken together, this means it can pause on a specific line in the source code. That's our breakpoint. You've probably already started your application using the debugger without knowing it. To debug, we press F5 or click the green "Start" button.

Visual Studio's Start or debug button

Finally, we need a way to step through our running code. With the debugger running, we should see execution paused at our breakpoint. (The current line is usually highlighted in yellow with a yellow arrow in the gutter.)

The next line to execute highlighted in Visual Studio debug mode

To navigate further, we use one of the following:

F10 or select Step Over in the debugging toolbar. This executes the next line of code without stepping inside a method call.

F11 or Step Into in the debugging toolbar. This executes the next line of code and steps inside any method that is called.

Shift + F11 or Step Out in the debugging toolbar. This runs to the end of the current method and stops.

F5 or click the green Continue button. This runs to the next breakpoint or until the program ends, whichever is first.

Shift + F5 or the red Stop Debugging square. This immediately terminates the program and stops the debugger.

Visual Studio's debug menu bar including buttons for Step Over, Step Into, Step Out, Continue and Stop Debugging

Now that we have the three things we need, let's fix our loop's run-time error.

First, set a breakpoint on the numbers array's declaration.

Debug the application. (You may also hear people say, "Start it in debug mode.") Press F5 or click the green "Start" button. If everything's configured correctly, the debugger is now paused at our breakpoint.

Press F10 or click the Step Over button on the debug toolbar. That's interesting. The debugger is smart enough to highlight i's declaration and initialization in the for loop.

Step Over again. Now we're stopped on the loop's condition: i <= numbers.Length.

Step Over. Hmmm, stopped on the loop's opening curly brace. Not sure how that's useful, but no matter.

Step Over. Ah, we finally stopped "inside" our loop on the Console.WriteLine statement. Hover over the variables i and numbers. Confirm that i equals 0 and numbers is an array of three elements: 3, 1, and 4.

Step Over twice. The first time we stop on the loop's closing curly brace. Again, not sure how that's helpful... The second time we stop on the loop's iterator statement: i++.

Now repeat. One interesting thing to note is that we don't stop on i's initialization the second time through the loop. That makes sense since that's how for loops work, but it's interesting to watch it unfold in the debugger.

Keep stepping with an eye on i. First it equals 0, then 1, then 2, then 3... and BOOM! -- IndexOutOfRangeException

So, what went wrong? Let's make a mental model. In this case, it's pretty straight forward. Start with the givens. numbers is an array of integers with three elements. The only other variable in play is i. We use i to both count our loop iterations and access an element at a given index in numbers. How many values does i have during its lifetime? 0, 1, 2, and 3. That's four. That's one more than the number of elements in our array! If we use it to access array elements counting from zero, its final value, 3, will try to access an element that doesn't exist. It runs off the end of the array or it's "out of range". Hence, the IndexOutOfRangeException.

That's an off-by-one error (Links to an external site.)Links to an external site.. They're all-too common. Easy enough to fix, though. Simply stop the for loop one iteration earlier. Something like this:

int[] numbers = new int[] { 3, 1, 4 };

for(int i = 0; i < numbers.Length; i++) // The condition <= is now just <.

{

Console.WriteLine($"The number at index {i} is {numbers[i]}.");

}

The Locals and Autos Windows

Visual Studio is a sophisticated piece of software. As we debug our code step-by-step, we can hover over any variable in scope to see its current value. That seems pretty cool, and it is... but it's a little fussy if we're stepping through lots of code and we're only interested in how one or two variables change over time. All of that clicking and hovering takes more time than you'd think! Lucky for us, Visual Studio summarizes variable values in debug windows.

The Locals window displays current values for all local variables. It appears when Visual Studio is in debug mode. If you don't see it, try showing it by selecting the menu Debug -> Windows -> Locals. There's a hotkey sequence as well.

The Locals window, part of Visual Studio's debug mode

The Autos window displays current values for "interesting" variables. That means variables referenced in the current and previous statement, including property access.

The Autos window, part of Visual Studio's debug mode

Both windows are extremely useful for tracking value changes through many lines of code. For example, observing changes during a loop saves a lot of time. In addition, you can reach down and change any variable's value while your application is running. Value types and strings are easy to change, but other types can be changed as well, within reason.

Note: Be careful! Changing an application's state at runtime can create unintended behavior and mask the underlying causes of an error.

The Watch Window

For even more control during debugging, use the Watch window. Again, this window is only active while debugging. If it's missing, show it via the Debug menu -> Windows -> Watch. It accepts any valid C# expression and displays its value. If the expression is a variable or property, its displayed value updates automatically. If the expression contains a method call, you may need to refresh the value when underlying data changes. If the expression is invalid or causes a side effect, the watch window displays an error message.

The Watch window, part of Visual Studio's debug mode

The Immediate Window

The Immediate window allows you to enter C# expressions and statements that act on the current scope. If you preface an expression with a question mark (?), the expression is evaluated and its value is printed back to the immediate window. The immediate window is especially convenient because it supports intellisense and auto-complete. It makes interactively testing your code a breeze. Examples:

Print the value of a variable.

?numbers

{int[3]}

[0]: 3

[1]: 1

[2]: 4

Clear the immediate window. (The > temporarily enters command mode and executes a command.)

>cls

Print the value of an expression.

?numbers.Sum();

8

Print the value of another expression. (You're not limited to variables. You can use literal values as well.)

?25 / 5.9;

4.23728813559322

Test for equality.

?a == 4.2

false

Update a variable.

?numbers = new int[] { 4, 5, 6, 7};

{int[4]}

[0]: 4

[1]: 5

[2]: 6

[3]: 7

?numbers.Sum();

22

Execute any in-scope method.

?string.Join(",", numbers);

"4,5,6,7"

Summary

Strong debugging skills are not optional if you want to be a professional developer. They are essential.

Start by learning to recognize common compile-time errors in the Error List window. Jump to errors in one click. Retrieve information about errors you don't recognize. Learn to recognize errors that cause other down-stream errors.

From there, master your debugging tools. Walk through your running code line-by-line to flush out even the toughest run-time or logic error. Form a hypothesis about each error and then confirm or refute it using the debugger. Use Visual Studio's debugging windows to interrogate variable values, change variable values, evaluate complex expressions, and generally tinker with your code while it's running.

Don't stop there. Visual Studio offers so much more (Links to an external site.)Links to an external site.. Leverage its additional debug windows like the Call Stack and Memory Diagnostics. Tweak your Exception Settings to fine-tune a debugging session. Attach your debugger to a remote process. Finally, don't forget that your greatest debugging tool is your brain\*. Think deeply about how something might go wrong before you fire up some sophisticated software. Software won't help if your mental model is weak.

\* "The Best Programming Advice I Ever Got" with Rob Pike (Links to an external site.)Links to an external site.