Debugging

Features that IDEs offer to assist with development.

Debugging is a core component of writing code, as issues arise very often, and having a toolkit to identify and resolve these issues is essential. In this document, I outline some of the processes that I apply and applied in the development of my windows from application for sorting algorithm comparison.

IDE Debugging

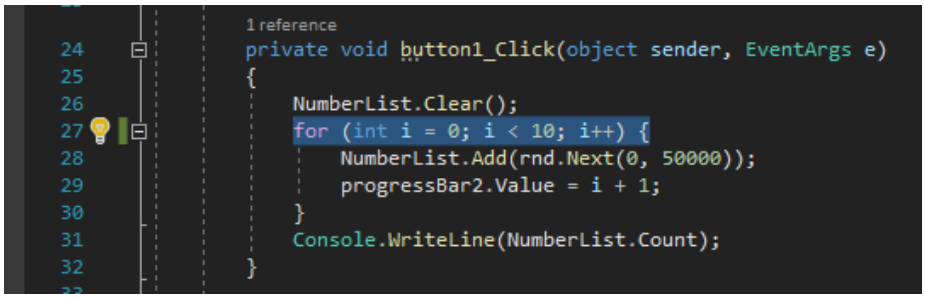
IDE Debugging is especially useful while writing code, as it helps to debug the code as it's being written. This doesn't help with any logical errors, such as a greater than or less than symbol being swapped, or an if statement being constructed wrong. Having the global and local definitions available through intellisense and linting is also incredibly useful in development because of it helps to identify available variables which are key in functions like sorting (Microsoft, 2019).

This process was applied as I was writing my code, as it's such a key feature for development. Some core examples of where this was particularly useful was writing the methods for button events, with the global/local definitions being incredibly useful to prevent the usage of the wrong variables; each method has it's own set of private variables (bTempList and iTempList, as examples), and mixing these up would've been very confusing without this form of debugging.

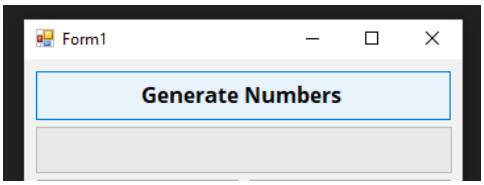
Runtime Debugging

Runtime debugging, using simulated instruction sets and special features like halting and breakpoints, allows for code to be analysed while it's being executed. This has a huge amount of versatility with code debugging, demonstrating the order of code that's executed, showing the conditions for branches and loops, showing variables that are referenced and relevant in each line.

I used a set of breakpoints in the production of this application, along with a modified quantity of array elements, to help reduce the time taken to run through with breakpoints.

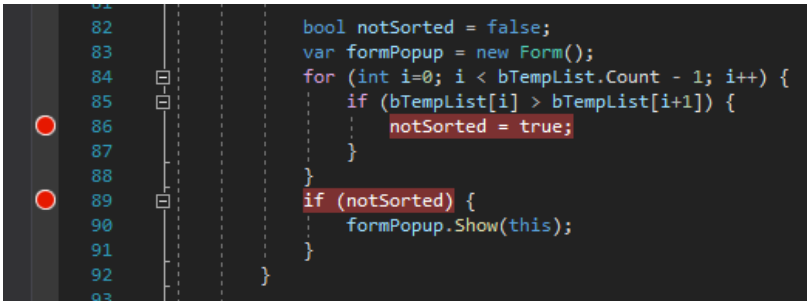


Interestingly, changing this number from 50,000 to 10 causes the progress bar for array generation to not show anything: this is due to the progress bar's value being 10/50,000, rather than 50,000/50,000.



These breakpoints show the variables being compared, checking whether the list is iterating properly, and if the elements are swapping properly. With bubble sort, this is demonstrated in the debugging execution itself, with long chains of swaps++ breaks due to the bubbles rising upwards, showing the correct behaviour. These chains usually occur with the larger values, closer to 50,000 (the upper limit). This behaviour is expected for bubble sort.

The insertion sort runs similarly, with short chains of swaps at first, symptomatic of the insertion into a smaller external list at first. This further illustrates the similarity between these two algorithms and demonstrates that they're implemented correctly.



These two breakpoints demonstrate whether the list is sorted, using the check system in either sorting algorithms. The two breakpoints are redundant here as they would both only show the value of notSorted. The first breakpoint only occurs if notSorted is set to true, whereas the second always happens, even if the list is sorted correctly. This helps to demonstrate, further than a popup window (which may be faulty), whether the lists are sorted.

Test Plan

Test plans are a core component of debugging, analysing a set of inputs and comparing the expected outcome to the real outcome, assisting the uncovering and diagnosis of issues. They allow for the conceptual planning of inputs that are expected, as well as those that are not, additionally inputs that make no sense at all in the given context, but which are still technically valid.

A test plan for a converter from strings to integers would look something like this:

#	Input	Expected Result	Actual Result	Description
1	“5”	Input casts the string to 5	Correctly cast to 5	
2	“5.2”	Input casts the string to 5 (rounded, not floor/ceil)	Error, string (float) cannot be converted to int.	Converter correctly identifies it’s a float, however it cannot be cast without a rounding system.
3	“5,000”	Casts the string to 5000	Correctly ignores the comma and casts to 5000	
4	“2147483648”	Casts the string to 2147483648	Incorrectly casts the string to -2147483648	2^31 + 1, which is over the limit for a 32-bit signed integer. Caused by an integer overflow.

The test plan that I used for this software has a similar structure, however the input method is different. As there are no input boxes for text, dropdowns for options or anything similar, only buttons, the input section describes the order that buttons are pressed in.

#	Input	Expected Result	Actual Result	Description
1	Generate Numbers Button	Progress bar fills to full, with a full array of numbers stored in memory.	Progress bar fills to full, then the program crashes because the progress bar is set to a value higher than the maximum.	Generating too many numbers for the array (compared with the progress bar maximum) causes the progress bar value to go past the limit. Limits should be the same.
2	Generate numbers, then bubble sort.	Bubble sort progress bar fills to full, after number progress bar filling to full.	Both bars fill to full.	Unclear whether the bubble sort process correctly sorted the array.
3	Bubble sort button before generating numbers.	Nothing – no errors or action or sorting (as array is empty)	Error trying to get the length of the array that doesn’t exist.	Since the array is null, not even empty, getting the length is impossible. Needs a check to see if it exists.
4	Generate Numbers, then bubble sort, then insertion sort.	All three progress bars fill up, with values for swaps/comparisons showing for both sort types.	The numbers and bubble sort bars work fine, with bubble sort showing sorts and comparison count, however insertion	The bubble sort algorithm is sorting the integer array itself, rather than affecting a local array separate from the

			sort does not show a value for swaps. Comparisons is the length of the array.	global one. This means the insertion sort method is trying to sort a sorted list.
5	Press bubble sort button twice.	Should fill the progress bar twice and have the same comparison/swap values.	Both bars fill and the same values are outputted correctly.	After fixing the global/local array problem, this result works as expected.

Debugging and Security/Robustness

Debugging is a process designed to identify and resolve issues in code, most usually edge cases that would, by definition, not arise during intended use. With the nature of security issues being about abusing these edge cases and issues in order to bypass security or perform an action that has negative impacts. Debugging tools add to the extensive toolkit that developers have for identifying security issues, giving access to information that may be vital for this process. Additionally, the robustness of any software relies on the minimising of the potential for errors or bugs occurring, dependent on the ability to handle border and edge cases without crashing (Pan, 1999).

It’s important to remember that debugging is not the perfect solution for robustness and security, with test plans and debugging systems only being so useful. Additional code analysis, like the utilisation of static code analysis and security tools, may be required to further insulate the software from potential security or usage issues.

References

Microsoft, 2019. *Debugging.* [Online]
Available at: https://code.visualstudio.com/docs/editor/Debugging
[Accessed 2nd January 2020].

Microsoft, 2019. *IntelliSense.* [Online]
Available at: https://code.visualstudio.com/docs/editor/intellisense#_intellisense-for-your-programming-language
[Accessed 2nd January 2020].

Pan, J., 1999. *Software Reliability.* [Online]
Available at:
https://users.ece.cmu.edu/~koopman/des_s99/sw_reliability/#introduction
[Accessed 6th January 2020].