# Hello Breakpoints

Hello World: Series 1 (repository)

Alex Allen (01-10-2025)

Before I get started, I hope everyone had good holidays. This article you are reading is a start to a series called *Hello World*. *Hello World* is a series I am developing to in order to introduce topics and development strategies in an easy to understand and framework to get started learning about the topic. Every programmer is familiar with the “Hello World” program and it is typically the first piece of code that anyone writes and runs. By introducing topics and breaking them down to be on the same level as a Hello World snippet of code, it will become easier to learn, apply, and build off of. Each of these articles should accomplish three things: An introduction and explanation of the topic, the ability to get hands on with the topic and use code found on [***my GitHub***](https://github.com/OathToLegacy/HelloBreakpoints) and allow anyone reading this to walk away and build a piece of code using what they just learned here to build something awesome! Let’s get started.

* *This code was written and debugged within visual studio 2022. The use of breakpoints is based off of visual studio 2022s debugger.*



I will be the first to admit that for most of my coding career, I have spent most of my debugging hours writing print statements all throughout my code to see where exactly it loses all sense of logic. It’s a difficult habit to break, but once one does, they realize how easier it is to just remove a breakpoint or add a breakpoint if necessary instead of hunting down dozens of print statements and removing them one by one.

It’s much easier to think of a debugger and breakpoints as a better version of you! What I mean is, the debugger knows itself very well. It is a very verbose system that just requires you to provide a reference point for the debugger to start examining and translating what it finds to you. The debugger uses breakpoints as its own print statements. But instead of just one print statement, it will have print statements of **every** piece of memory, variable, and instance that changes within that breakpoint.



Instead of having a print statement directly after the address is saved, I have set a breakpoint just before. When I activate the debugger, it will stop the running code where I have set the breakpoint and then informs me on any variable currently in memory and what there current value is. After the address is saved to the variable *arrAddr*, it shows an updated list showing the address that has been saved: A screenshot of a computer

Description automatically generated

This is extremely useful for a lot of reasons: It promotes cleaner code, saves a ton of time, and allows a programmer to see the flow and functionality of the code they are writing more clearly. When I don’t need to know the address I just hide the breakpoints and run as normal until I have another issue.

Here are some good tips to get started debugging in visual studio:

1. While coding there is ever the urge to implement a print statement to figure out what a piece of the code is doing, find this button on the top bar instead. Just see how everything is interacting in memory.



Doing this will help you have a better understanding of how your code is working in general but can give you a greater overview of what exactly is causing issues, leading to a solution sooner.

1. Use the bar to the right to place a breakpoint. This will halt the entire running process so that memory can be examined at that given point in the programs life cycle

A screenshot of a computer

Description automatically generated

The white dot seen on line fifteen indicates the presumed spot you would like to place the breakpoint. It will turn red when one is placed.

1. Once in debugging mode, visual studio offers several buttons in the same place where the **local debugging** button was located.



1. **Continue** will continue on from the breakpoint and run the code as normal until the next breakpoint
2. **Hot reload** will reload the code into memory and run again until that breakpoint
3. **Stop** will exit the running program gracefully and return to a normal IDE setup.
4. **Step Into** will move to the called function (if there is one)
5. **Step Over** is if there is a current function call, it will run the code and then suspend at the first line after the function returns. Note: If there is an I/O call, such as scanf() it will still suspend until user input
6. **Step Out** is very similar to step over, but instead, it suspends at the return of the function. It will also skip through the current function. Once again, on an I/O call, it will still suspend until user input.

The program found on the github can be a good introduction into using breakpoints to debug code instead of print statements. The main exercise of this program is to use the debugger to answer the question that is prompted when the code is ran. There is a lot of other things to explore to. Try to determine how *array A* is stored into array B. You can even answer the following questions.

1. **What is the address of the pointed index of the array?**
2. **What is currently stored in array B with nothing in it?**
3. **Does the address change if you rerun the program?**
4. **How does *array B* know what value to store at that index *i*?**