# What is Debugging?

It is a systematic process of spotting and fixing the number of bugs, or defects, in a piece of software so that the software is behaving as expected. Debugging is harder for complex systems in particular when various subsystems are tightly coupled as changes in one system or interface may cause bugs to emerge in another.

Debugging will not give confidence that the system meets its requirements completely but testing gives confidence.

Under a debugger, you can step through code as it is running, examining the values of variables, structures, program flow, screen output, and other elements of the application for errors. It is slower than running the program outside of the debugger, but vital to understanding the inner workings of an application.

You may need to debug when:

* Code crashes randomly, stopping the application and losing data
* Results are wrong or different from previous versions
* Divide by zero
* You need to optimize code based on profiling results
* Feature does not work as expected (selecting “print” produces a garbled page)
* User interface elements are in the wrong place, wrong size, have wrong labels…
* Network communication is not working
* Game enemy keeps running into the corner and dying (AI issue)
* Multi-threaded code is deadlocking
* Component fails to load
* Customer files a bug in your feature area
* You need a better understanding of how code works under the hood

# Integrated and Stand-Alone Debuggers

Some debuggers are stand-alone and some are integrated into a development environment.

## Stand-alone debugger

A stand-alone debugger exists independent of other development tools. Because it is independent, it might have a different user interface, might support a broader set of languages, and might need to be told which program to launch for debugging or which running program to attach to for debugging. [WinDBG](https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windbg) and [GDB](https://www.gnu.org/software/gdb/) are popular stand-alone debuggers.

## Integrated debugger

An integrated debugger is part of a development tool, usually alongside other useful development tools in the IDE. With an integrated debugger, you do not need to switch from editor to debugger to diagnose issues. The user interface is consistent, languages supported by the IDE are usually supported by the integrated debugger, and it is already configured for common scenarios. Visual Studio has an integrated debugger that works with all Visual Studio programming languages and their associated libraries.

# Common Features

Most debuggers share a common set of features

## Start debugging

To debug, you need to start your app with the debugger attached to the process.

In Visual Studio, press F5 or select Debug | Start Debugging. Your code may need to be rebuilt.

Sometimes, in the heat of debugging, you may forget whether you are in edit mode or actively debugging an application. Visual Studio provides clues. In edit mode, the bar at the bottom of the editor is blue.

In debugging mode, “(Debugging)” appears in the title bar and the bar on the bottom of the screen is orange. And unless you’ve made configuration changes, there are debugging and diagnostic panels in the editor.

## Setting a breakpoint

**What is a breakpoint?**

**What is a conditional breakpoint?**

**What is a watchpoint?**

Breakpoints are useful when you know the line of code or the section of code that you want to examine in detail. By setting a breakpoint, you tell the debugger to pause when execution hits a selected line of code so you can inspect values, examine output, change program flow, override values, and perform other diagnostic tasks.

In Visual Studio, you set a breakpoint by clicking in the margin to the left of a line of code, by pressing F9, or by selecting Debug | Toggle Breakpoint on the menu. When in debug mode, you can Run to Click, to advance the debugger to the line of code where you clicked by hovering over over a line of code until the Run to Click (Run execution to here) button appears.

## Step into code

Stepping into code advances the app execution into the details of the function. You will see the underlying function code in detail and is helpful when you are trying to identify where a bug might be hiding.

In Visual Studio, press F11 to step into code. If you press this while not debugging, the debugger will start and execution will begin at the first line of your code.

## Step over code

Stepping over code advances the debugger without exposing function details. The code still executes, but execution advances in a single “leap”, avoiding the implementation. It is a good way to skip over code that you’re not interested in, so you can quickly get to code that you are more interested in.

Press F10 in Visual Studio to step over code.

## Inspect variables

Inspect the **current type and value** of variables in scope to find incorrect data and better understand program behavior. You might be able to see variables used in the previous few lines of code (called automatic or *auto* variables) and *local* variables (variables currently in scope, often including the implicit “this” pointer when inside a C++ object).

The Visual Studio debugger displays both Autos and Locals windows while debugging. You can see the value of an individual variable by hovering the mouse cursor over it.

## Modify a variable

Sometimes it is useful to see what happens when variables have different values while debugging.

Values can be C and C++ expressions. If you use operators that modify values (like post-increment) or call a function, it can change the value of other variables or otherwise affect the state of your application.

In Visual Studio, the variable windows, Autos, Locals, and Watch, display the values of certain variables during a debugging session. The QuickWatch dialog box can also display variables. When the debugger is in break mode, you can use the variable windows to edit the values of most variables that appear in these locations.

## Examine the call stack

The call stack shows the order in which methods and functions are called (like function x calling function y which in turn calls function z). This is a good way to understand the execution flow of an app and helps answer the question “where am I in code and how did I get here?”

The Visual Studio debugger automatically shows the Call Stack.

## Change execution flow

By changing execution flow, you change which statement will be executed next. This is helpful to force execution flow down a certain code path or to re-execute a block of code after setting variables to different values.

With the Visual Studio debugger paused on a line of code, use the mouse to grab the yellow arrow pointer on the left and move the yellow arrow pointer to a different point in the code execution path. Then you use F5 or a step command to continue running the app.

# Notes

Debugging doesn't refer to removing compiler errors, does it? Maybe you could split it into "Compile-time (compiler) Errors", "Runtime Errors" and "Bugs".  
  
Also you should mention assertions. Assertions are incredibly useful because you can sprinkle them throughout your code, and, rather than having to remove them, you can just stick **"#define NDEBUG"** in a global header file (make sure it's the first thing included in any file) and the assertions get turned into null statements by the preprocessor.