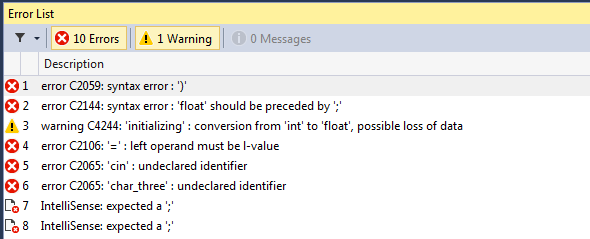
Tutorial – Visual Studio Usage

We’re going to spend some time looking through the main features you can use in Visual Studio.

Finding Syntax Errors

With Visual Studio open, click view->Error List… to open up the Error List window. This window displays all the syntax errors, warnings and messages the compiler has outputted about your code.

Two kinds of errors display in the error list - errors found by the compiler, and IntelliSense errors.

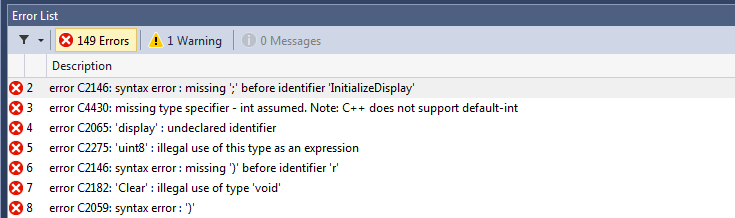


Compiler Error

IntelliSense Error

IntelliSense is a feature of Visual Studio that understands C++ code and tries to figure out if there are errors in your code. The compiler errors are only updated when you compile your code, but IntelliSense is constantly updated as you type.

A rule of thumb, at least while you’re still getting used to the rules of C++ is to only look at the first compiler error, fix it and try recompiling straight away. Often, a single error will confuse the compiler and cascade into displaying many, many errors. Looking at this window, all 149 errors were caused by that first single missing semi-colon.



Finding Warnings

You can toggle between displaying errors, warnings, or both by clicking the warning and error buttons at the top of the error window.

Just like the errors, double clicking the warnings will jump the display to highlight that warning. Here at AIE, all assessments MUST have no warnings to be passed, so whilst you can compile and run your code, even if it has warnings, it is recommended that you treat them as though they were errors.

Exercise 1:

Take this code and fix all the compile errors. Run it and see what it displays. Then fix all warnings.

#include <iostream>

int main()

{

int number\_one;

int number\_two = 10;

int number\_three = number\_one + number\_two;

char char\_one = 'a';

char char\_two = 'b'

int number\_four =

(((int)(char\_one + number\_one) \* number\_three)) \* 6 - char\_two);

float float\_one = number\_two + 16;

char\_two-- = 7;

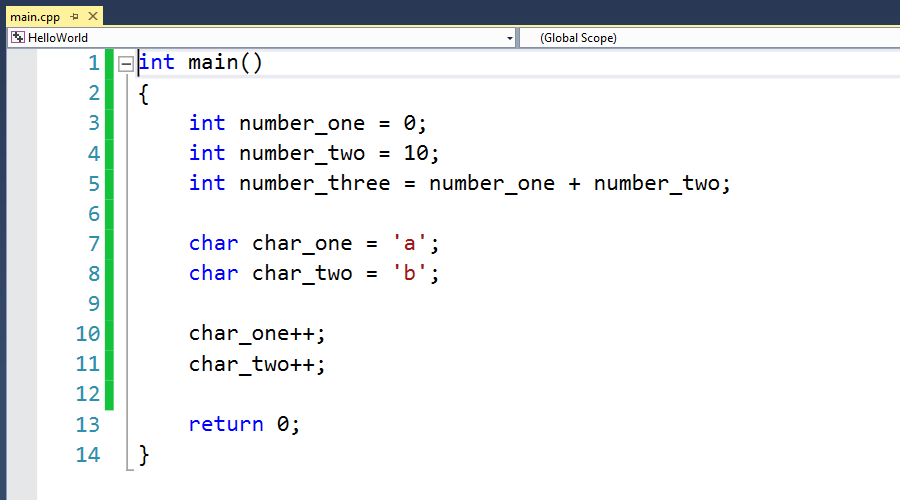
cin >> char\_three;

return 0;

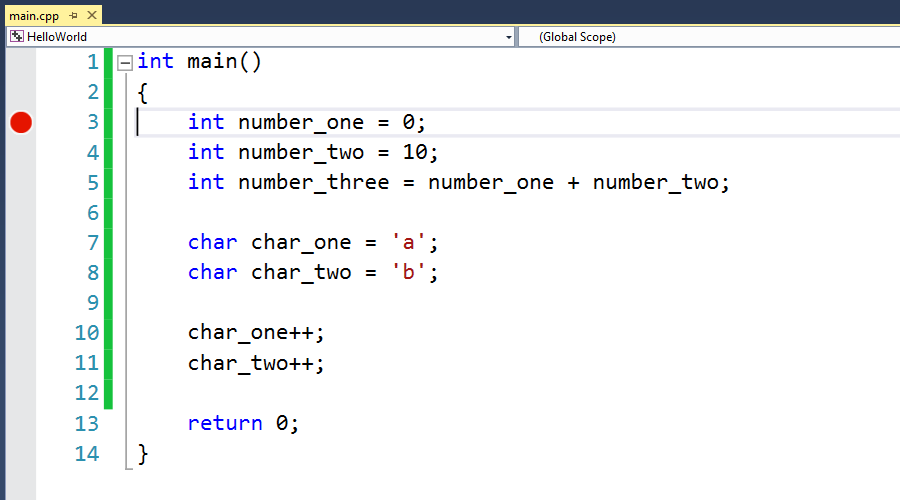
}

Placing Breakpoints

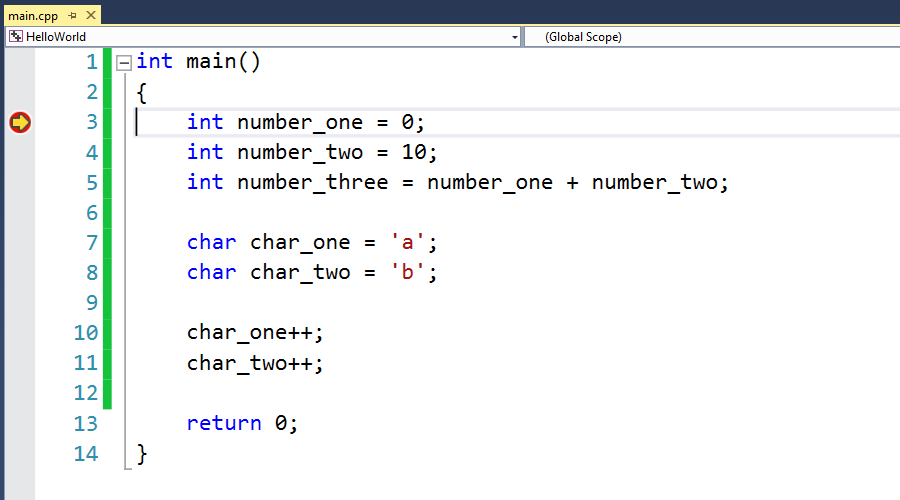
Breakpoints are vital for debugging your code. The dark grey bar to the right of the code window is called the gutter. Clicking here will add a breakpoint on that line of your code.



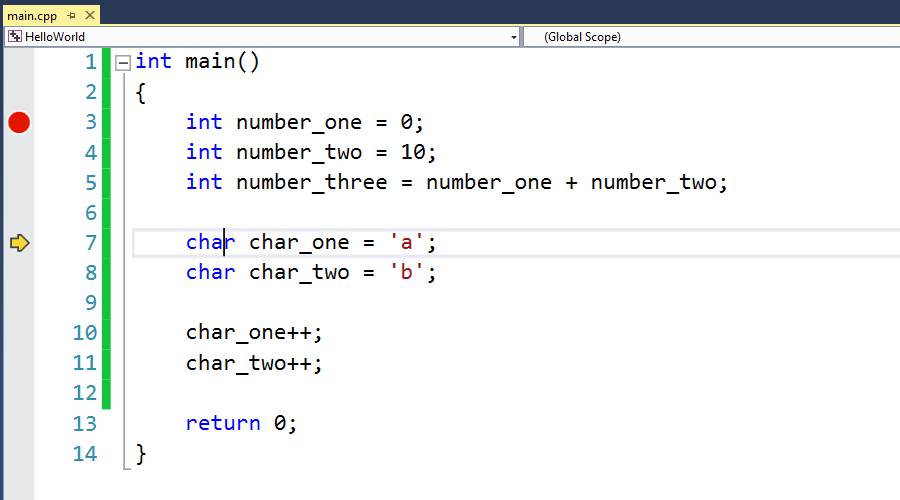
The breakpoint will be displayed as a red circle. You can also set a breakpoint on a line, by placing your text cursor on the line you want, and hitting F9.



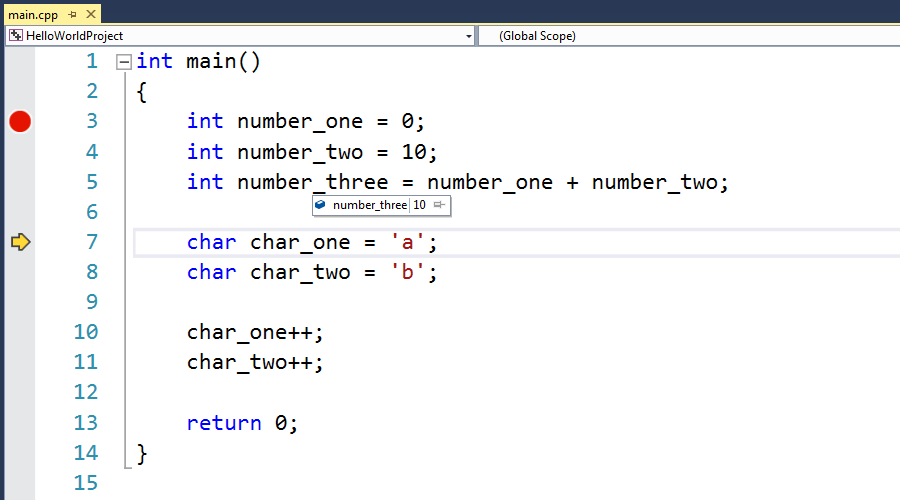
Once a breakpoint is set, if you run your code, your program will pause just before running the line that your breakpoint is on. When your program has been paused by the debugger, a yellow arrow will appear inside the breakpoint.



Once a breakpoint has been hit, you can hit F11 to have the program advance itself by one line. The yellow arrow always points to the next line to be executed.



As your code runs, you can hover your mouse over variables and see how their values change over time.



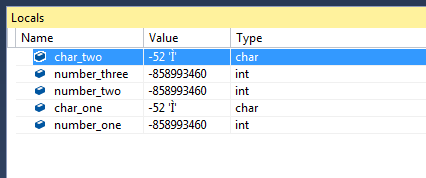
You can add and remove breakpoints freely as your code is running.

If you right click in the gutter, you can choose from a few different kinds of breakpoints. The default breakpoint simply pauses any time the code gets to that line. Conditional breakpoints only pause the program on the line if a condition you give them is true (such as a variable being set to a specific value). Hit Count Breakpoints let you set how many times the code can go past the breakpoint before it actually pauses the program.

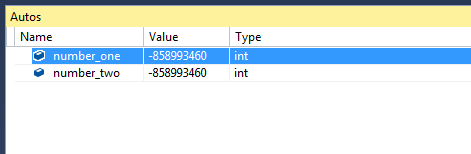
Autos, Locals, Watch Window

Along with hovering your mouse over variables, Visual Studio has a number of dedicated windows for looking at the values of variables.

The Locals window displays all variables active in the current scope, their values and their type.



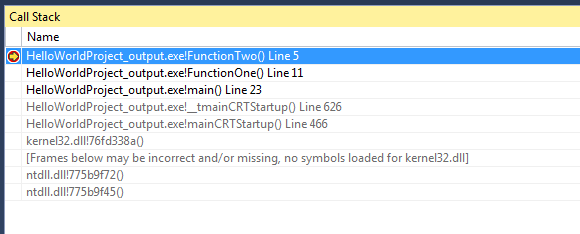
The Autos window shows the same information as the locals window, but only for the variables being used in the current line.



In the Watch window, you can type in any variable name, or even full expressions to keep track of.

Call Stack Window

The call stack window shows you what functions were called to get you to where you are in the code. We haven’t covered functions yet, but once we do, the call stack will be a very useful tool for you. You can double click on each function to jump to where they were called.



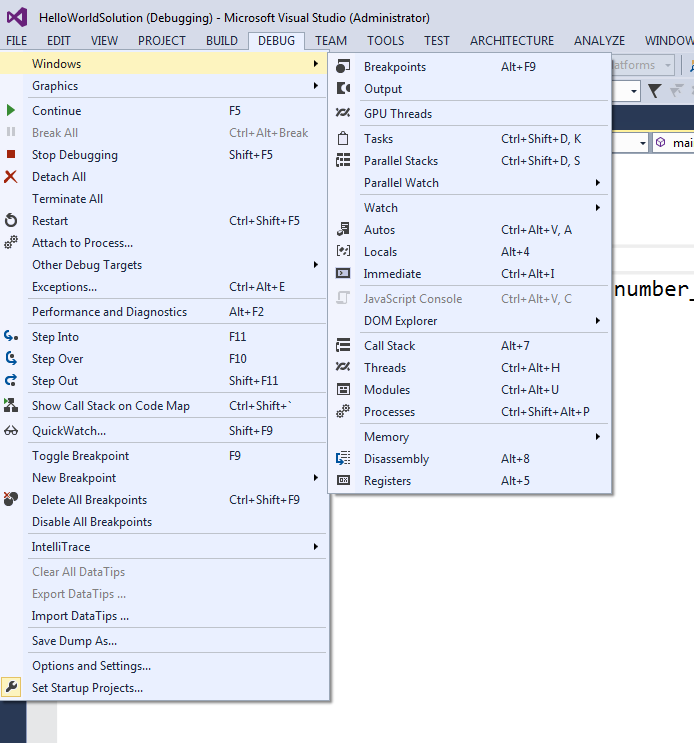
Advanced Windows

These windows let you look at more advanced data. You won’t need to use these windows often for a while, but it’s good to know that they’re there.

* Memory
  + The memory window lets you look through all the memory your program is using in a big block
* Registers
  + The registers window lets you see the current value of the CPU registers as you step through your code.
* Threads
  + Allows you to inspect all the threads your application has spun off.
* Disassembly
  + This window lets you look at the actual machine instructions the compiler has generated from your code. From this window, you can step through your code one instruction at a time instead of by line.

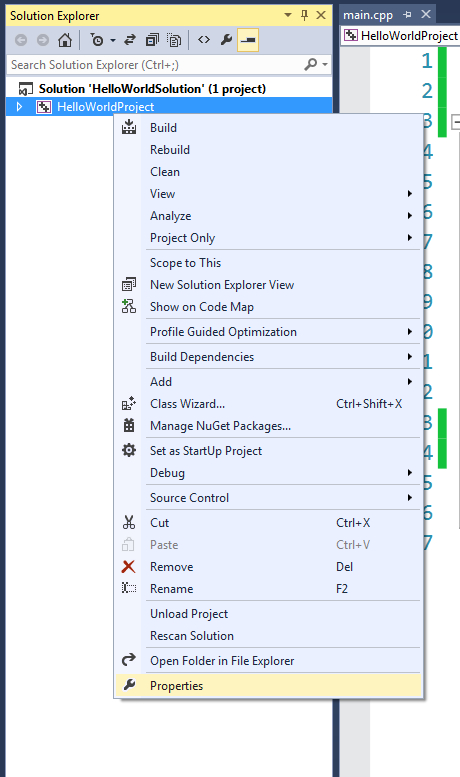
If you can’t find the debug windows we have been talking about, to open any of them:

* First, set a breakpoint and run your program to get VS into debug mode
* Then go to DEBUG->Windows and click the window you want to open.

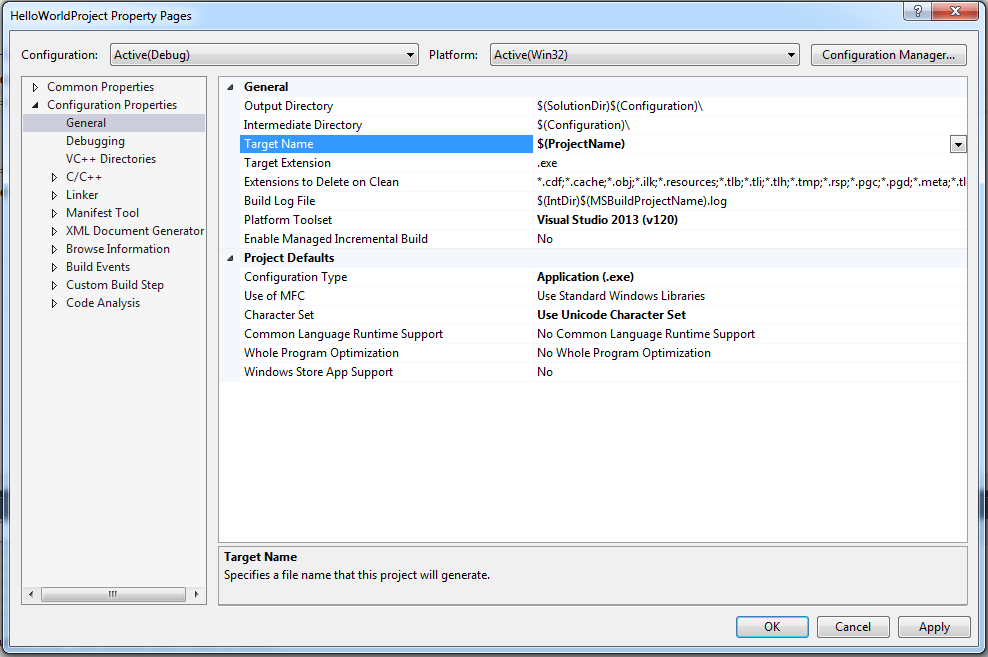


Project Settings

In visual studio, your project controls what and files get built into an executable. There are many useful settings you can change inside your project properties. To see your project properties, right click on the project in the solution explorer and click properties.



This is the window that should appear



Look through the options that are available to you.

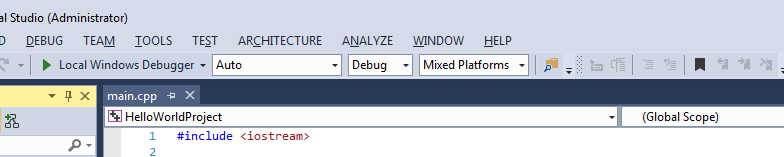
Some simple options to change:

* Changing the output directory [**General->Ouput Directory**]
  + This changes where the compiler builds your final executable to. This is handy to change if you want to customize the folder structure of your project
* Changing the working directory [**Debugging->Working Directory**]
  + This changes what folder the executable thinks it’s running in. This is very useful when trying to load files in your code.
* Changing the warning level [**C/C++->General->Warning Level**]
  + Here you can change what level of warnings the compiler will give you. Higher levels give more warnings. All AIE assessments expect your projects to compile warning free on at least level 3.
* Treat warnings as errors [**C/C++->General->Treat Warnings as Errors**]
  + Here you can tell the compiler not to finish compiling if it detects any warnings. It is recommended you turn this on.
* Turning On Optimizations [**C/C++->Optimization->Optimization**]
  + Here you can change how much the compiler optimizes when it converts it to machine code. Debugging is much easier with optimizations turned off, but your code runs much faster with them turned on.

Build Configurations

Build Configurations allow you to save a group of settings so you can easily switch between them on the fly. There are two configurations created in your project by default – Debug and Release. The main default differences are that debug has optimizations turned off and release has them turned on.

You can swap between the two configurations by using the drop-down at the top of the main window.



When changing settings for your project, you need to make sure you know which configurations you’re modifying. It tells you at the top of the properties page.