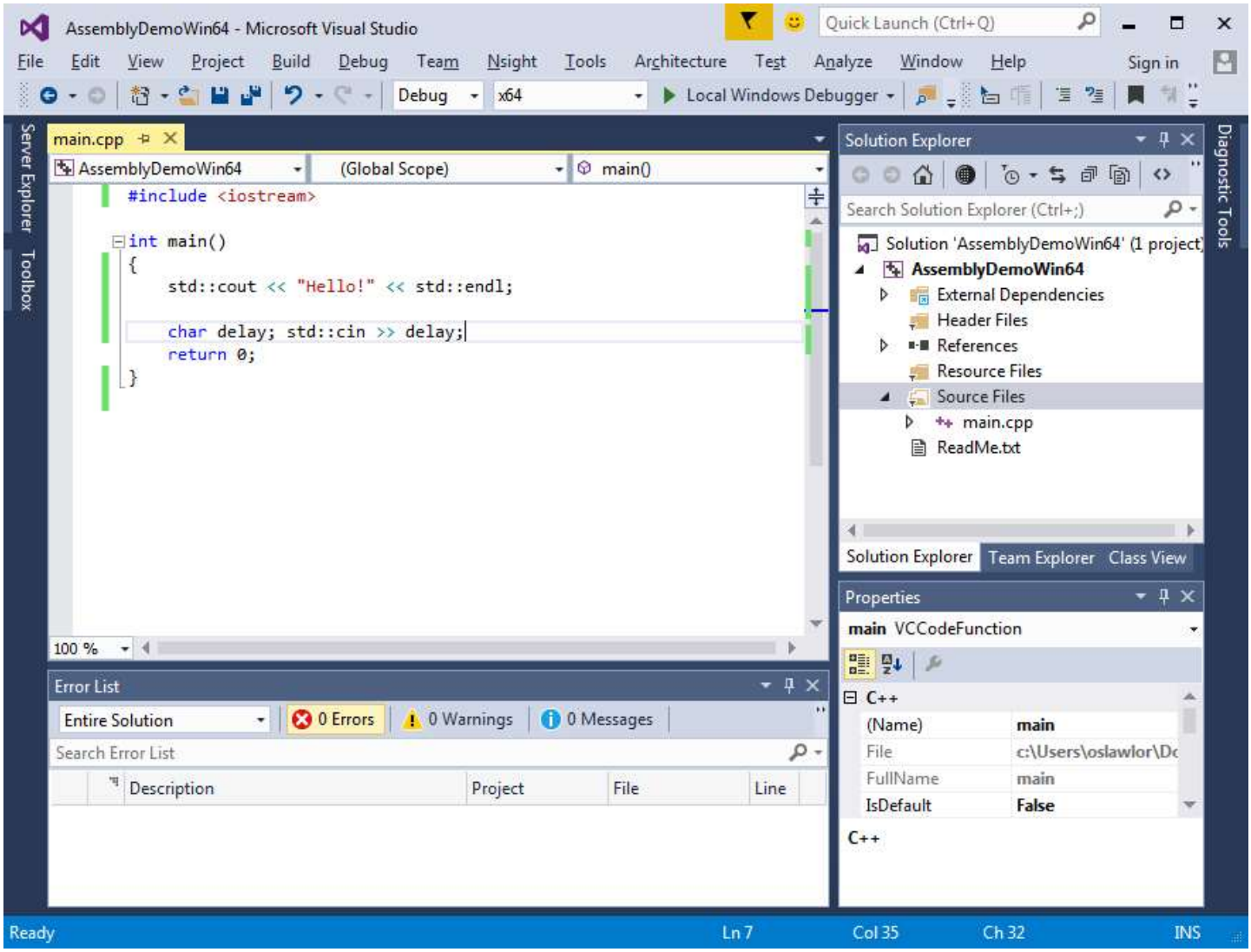
Using 64-bit Windows Assembly Language from Microsoft Visual Studio

Dr. Orion Lawlor. Last updated 2020-10-02. I works basically this way in Visual Studio Community 2019, and back to at least VS 2015.

Step 1: Write C++ Code

First, make an empty C++ project.
Switch to 64 bit mode (x64 dropdown, just to the right of the Debug drop down).
Add a C++ main file, and test out the program to make sure plain C++ works.



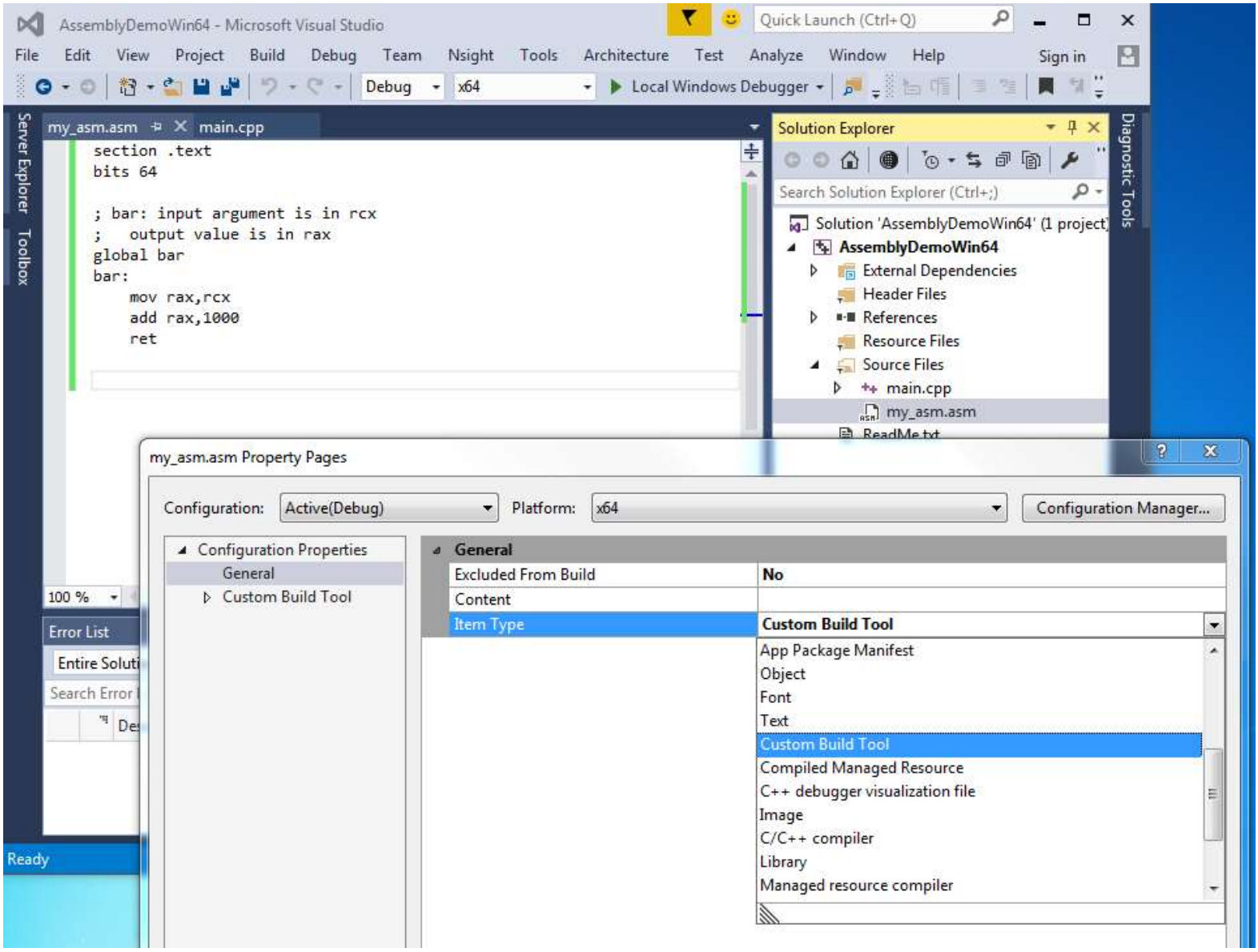
Step 2: Write assembly code

Now write an assembly language source file with these critical pieces:

- Give the file the .asm extension, the standard extension for assembly code. (At least on Windows. Unix uses .S or .s too.)
- Start the file with "section .text", so the code is executable.
- Add the directive "bits 64" so NASM knows you want 64-bit registers.
- Make the function "global", so the linker can see it. (No leading underscores are needed in 64-bit mode.)

Finally, right click on your new file, and hit properties. Set:

- "Excluded From Build" to "No"
- "Item Type" to "Custom Build Tool"
- Hit Apply



Step 3: Make Visual Studio call NASM

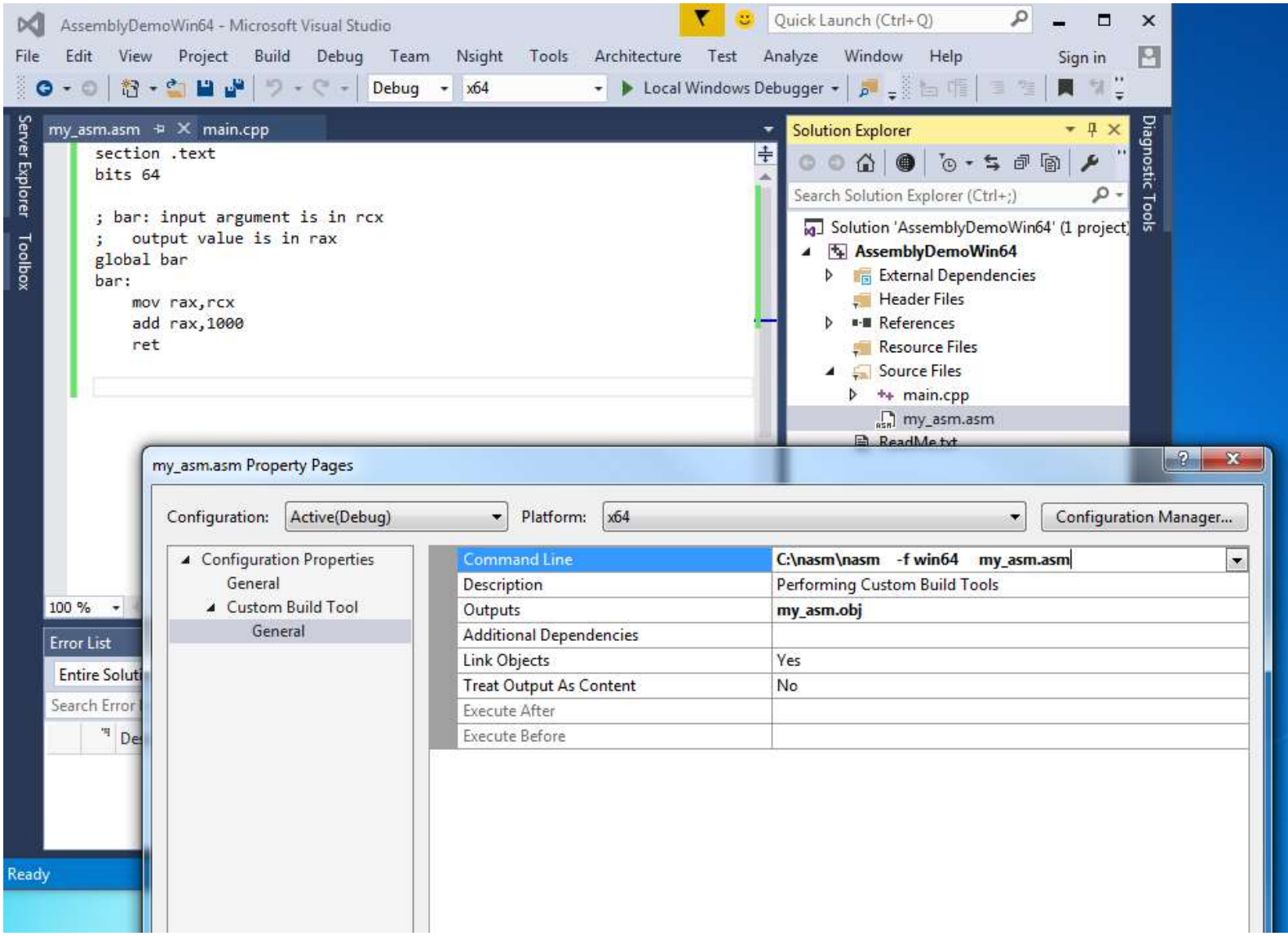
Now you need to download and install NASM (or YASM, etc.)
Unzip nasm to a handy location, like C:\nasm\nasm.exe.

You then give Visual Studio the full Command Line needed to run your assembler, just like you'd type at a command prompt:

```
C:\nasm\nasm.exe  -f win64  my_asm.asm
```

The installer puts it in C:\Program Files, so you'll need to quote it like "C:\Program Files\NASM\nasm.exe", or you'll get "Unrecognized command C:\Program" because of the space in "Program Files".

The assembler will output a .obj file, which you list under "Outputs" so the linker can see it.



Build the project, and you should be able to call assembly functions from your C++, and vice versa! Don't forget extern "C" from C++!

Simple C++ code:

```
#include <iostream>
extern "C" int foo(void); // written in assembly!

int main() {
        std::cout<<"Foo returns "<<foo()<<"\n";
        system("pause");
        return 0;
}
```

64-bit Windows assembly code:

```
section .text ; makes this executable
bits 64 ; allow 64-bit register names
global foo ; makes this visible to linker
foo:
        mov rax,7 ; just return a constant
        ret
```

Common Errors

'ADDR32' relocation to '.text' invalid without /LARGEADDRESSAWARE:NO

In more recent versions of Visual Studio such as 2019, you may get this link error when you reference memory from assembly using normal absolute addresses. You can fix this by adding the /LARGEADDRESSAWARE:NO flag:

- Right click your project

- Properties
- Linker
- System
- Set the "Enable Large Addresses" dropdown to "No"
- Apply

This is the equivalent of the Linux "-no-pie" flag: it turns off the fancy address relocation support, to allow you to use simple absolute addresses.

If you want to be very modern you could instead switch to fancy new rip-relative addresses, but if you're at all hazy about pointers, just do the fix above.

Link error: missing function "foo()" (note the parenthesis)

C++ silently adds the parameter types in parenthesis to the linker names of all C++ functions, to support function overloading.

Mark your C++-side prototype 'extern "C"' like the examples above, and C++ won't include the parenthesis, which will let you write the function in assembly (or plain C).

My assembly works fine in NetRun, but in Visual Studio rdi arguments don't work

NetRun runs your code in Linux, where a function's first argument is passed in rdi.

Windows 64 uses a different parameter passing convention, where a function's first argument is passed in rcx. There are also a different set of preserved registers, and different stack alignment requirements.

cmd.exe returned error 1

One annoyance in older (pre-2019) versions: any NASM build errors show up in "Errors" as "cmd.exe returned error 1", which tells you nothing useful. Check the "Output" tab to see nasm's line numbers and other helpful error info. This is fixed in VS 2019.
