Prerequisites

- An Open Watcom v2 installation (1.9 may work, but not tested).
- A DOS installation ready. Here we shall use DOSBox-X.
- This tutorial is written for people with at least minimal modern C experience but no 8086 or MS-DOS programming experience.
- This tutorial also assumes you'll be using the project template given in the main page.

Introduction to C programming for MS-DOS

Welcome! I'm assuming you have your OpenWatcom installation prepared and in the PATH and all that stuff, and the project template ready.

Look inside `src/main.c`. You'll see the following:

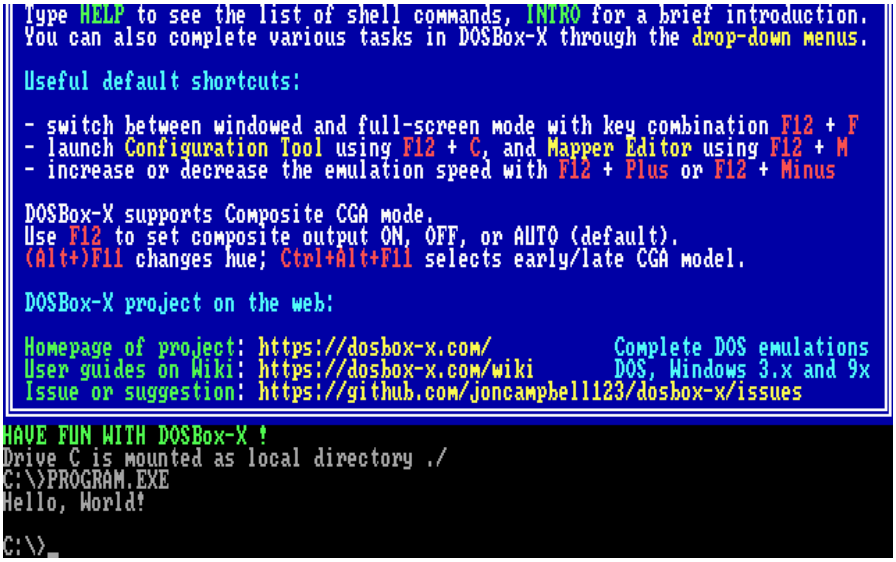
```
#include<stdio.h>

int main(int argc, char **argv) {
    puts("Hello, World!");

    return 0;
}
```

This should be self-explanatory. It is, in fact, standard C. But we'll be deviating from it for performance's sake.

For now, let's try running it. Open your terminal in the project root directory, and type `wmake` (it should've come with your OW package). You should now have `program.exe` in there as well. We can run DOSBox-X and mount the project root directory using `dosbox-x ./` (run this from the project root, so the configuration file is noticed). Once you're in DOSBox-X, type `PROGRAM` and press enter.



While it might not look like much, you've just compiled a program that will run on any DOS install.

OpenWatcom features a complete C standard library for you to play around with. The project template is configured to compile C99 code with Watcom extensions, but Watcom

does *not* support all C99 features such as designated initializers.

8086 memory addressing

If you've done any modern C programming whatsoever, you've probably become accustomed to the idea that a pointer is really just one integer, an index into a large array which is the address space. What you've been working with is called a linear address space, and this is not the case on the 8086.

Because of the limits of 16-bit addresses, the 8086 employs a technique called "segment addressing". Addresses using this system are comprised of a segment and an offset. The segment is shifted left by four bits (the equivalent of multiplying by 16), and added to the "offset". This in effect allows the programmer to use a full 20 bits of address (enough to access 1 MiB of memory!).

For the mathemaphiles, a formula:

```
linear = segment * 16 + offset
```

The multiplication by 16 can be easily visualized in hexadecimal: it's just adding a zero to the end. For example:

```
linear = 0xB800 * 0x10 + 0x0123 =
       = 0xB8000 + 0x0123 =
       = 0xB8123
```

The processor would actually pass `linear` to whatever handled memory, since only the 8086 has to deal with segmentation.

The notation `segment:offset == linear` is commonly used.

This form of "virtual addressing" is being performed constantly and implicitly at no real speed cost.

Changing segments implies a state change, which would require coordination throughout the entire program. C doesn't have any method to perform such low-level coordination (and the next best thing, N19, isn't ready as of now), so either C implementations had to handle the segmentation cruft themselves (slow) or they would implement non-standard solutions that programmers could use to optimize their programs for the 8086. They had done both, by splitting pointers into multiple types: near, far and huge.

Near pointers always deal with the data segment. This is helpful when you're working with variables you've allocated statically, for instance. Near pointers should be used as much as possible.

Far pointers hold both the segment and the offset in one 32-bit value. Upon each access, the compiler has to make sure that the segment registers are set correctly, and access it. For this reason, far pointers are slow. They are necessary if you are working with data that have an unknown linear address, though.

Huge pointers solve a few problems one'd have with far pointers. Segmentation may allow us to access 1MB, but the memory addressing function above is many-to-one: there are multiple combinations of segment bases and offsets that give us the same linear address. In fact, 4096. If you were comparing two far pointers, say, `0xB800:16` and `0xB801:0`, they'll be considered unequal even though they represent the same linear address. The second problem

is that the segment field of a far pointer stays fixed: if you have a far pointer to some buffer that you continually increment, and it crosses the segment boundary, the offset field will overflow, yet the segment field stays the same. You would have accidentally jumped back, the opposite of what you wanted. Huge pointers solve both these issues by "normalizing" the values when necessary (the actual details are unimportant). Of course, these are the slowest.

As lovely as it would be, OpenWatcom can't handle linear addresses for us. Since we're dealing with linear addresses directly, we're forced to use segmented pointers instead. But since we're learners and we're not really trying to be portable, this is not a problem.

```
uint8_t *ptr = 0xB8000; // This is incorrect.
uint8_t far *ptr = MK_FP(0xB800, 0); // This is correct. As said before,
other combinations will work, such as MK_FP(0xB000, 0x8000)
```

IO ports

Apart from the normal address space we just talked about, the 8086 featured a second address space: the IO space. This one has been phased out in modern computers due to more focus being put on optimizing memory access. It also works fundamentally differently to regular memory: a 16-bit access to memory at, say, 0x300 will access addresses 0x300 and 0x301. With IO ports, one half is sent to 0x300, then another half to 0x300 again.

A few devices we'll be working with in the future shall require us to work with this space. I promised you wouldn't touch Assembly directly, so the project template comes with the `inb`, `outb`, `inw`, `outw` functions, available through the `dio.h` header.

Next