```c
#include <stdio.h>

void main() {
        int i = 2;
        int a = (++i) * (++i);
        printf("i = %d, a = %d\n", i, a);
        int b = (++i) * (++i) * (++i);
        printf("i = %d, b = %d\n", i, b);
        int c = a *b;
        printf("c = %d\n", c);
}
```

This was quite interesting. Compiling to assembly code actually shows how this expression is parsed by the compiler. You can try doing this by: gcc -S program.c

This produces program.s which is the assembly language code. We haven't discussed this in class, so it would sound very new to many students. Rest assured this won't be part of the syllabus!

So what happens is that we like to write programs in high level languages like C, but like we saw in the class, the computer only knows how to execute some very basic instructions like ADD X,Y to add X and Y, or MUL X,Y to multiply X and Y. The job of compilers is to convert the C program to these basic instructions. Normally when you compile as gcc program.c, you get an a.out or a.exe as an executable file but this is in binary and not human readable. Compiling using gcc -S program.c produces a human readable program using these basic instructions. This language is called assembly language. Very early on, programmers would actually write in assembly language! In fact, basic parts of many compilers are written in assembly language, otherwise how would you write a compiler in the first place?

Here is a snippet of the assembly language code for the part: int a = (++i) * (++i)

```
    addl    $1, -4(%rbp)
    addl    $1, -4(%rbp)
    movl    -4(%rbp), %eax
    imull   %eax, %eax
```

As you can roughly make out, -4(%rbp) seems to refer to the address of i. The first addl instruction is adding 1 to i. Then another 1 is added to i. Then the value of i is moved to a register in the CPU, called the EAX register. Recall the Von Neumann architecture - the instructions are fetched to the CPU, decoded, and executed. These instructions need the data to be in the CPU, not in the memory. The place in the CPU where data is stored for operations is called a register. A CPU has several registers. So finally, the imull instruction then multiplies the value in the EAX register by itself.

What does this tell us about how the compiler parsed the expression? We can see clearly now that the compiler first converts the expression in the first bracket to assembly, i.e. it converts (++i) to the first addl instruction, and when executed this changes the value of i to 3. Then it converts the expression in the second bracket to assembly in the same way, and when executed the value of i becomes 4. Next it does something quite clever. It notices that first operand (++i) is the same as the second operand (++i), and therefore it just moves the value once to the register and multiplies the value by itself, i.e. 4 x 4, and we get the result 16. Check what happens if you have an instruction like int a = i * j.

The next instruction is:

```
        movl    %eax, -8(%rbp)
```

The result of the imull instruction remains in EAX. The movl instruction moves this value in the register to the address of b, which is at -8(%rbp). Notice the multiples of 4. You now know that this is basically sizeof(int), i.e. the 4 bytes that an integer variable occupies in the memory. The "-4" and "-8" are actually offsets which tell the CPU to pick up data from those memory locations.

Now you can figure out what happens with the other expression: int b = (++i) * (++i) * (++i)

The assembly code looks like this:

```
        addl    $1, -4(%rbp)
        addl    $1, -4(%rbp)
        movl    -4(%rbp), %eax
        imull   %eax, %eax
        addl    $1, -4(%rbp)
        movl    -4(%rbp), %edx
        imull   %edx, %eax
        movl    %eax, -12(%rbp)
```

So like before, the compiler first converts the first (++i) into an addl instruction, then second (++i), then it moves the value to i (which would be 6 by now) to the EAX register and multiplies it by itself. The value of the result is in EAX, and which is 6 x 6 = 36. Next it again converts the third (++i) into an addl instruction and i is now 7. It then moves this value to the EDX register in the CPU. EAX and EDX are similar to each other, these are called general purpose registers. The imull instruction multiplies the values in these registers and leaves the result in EAX. This multiplication as you can now see was between 36 and 7, giving the result 252. The last movl instruction then moves the value in EAX to the address of c.

If you are able to understand this then you've completely got how a computer actually works. In software, you can't get any closer to the metal than this!

If you are more curious, try figuring out what happens with the printf calls.