#  Functions Revisited Homework

1. Write a function, timeUpdate(), that is passed three int parameters, hour, min, sec, representing the current time and returns the time updated by one second. The function should work with a 24-hour clock where 4:30 am is represented by 4:30 and 4:30 pm is represented by 16:30.

void timeUpdate(int \*hours, int \*minutes, int \*seconds) {

++\*seconds;

if (\*seconds == 60) {

\*seconds = 0;

++\*minutes;

if (\*minutes == 60) {

\*minutes = 0;

++\*hours;

if (\*hours == 24)

\*hours = 0;

}

}

}

2. Explain the output of the following program. Does interchanging the arguments in the last printf() statement make any difference?

#include<stdio.h>

int main(void) {

char a = 127, b = 128;

printf("%5d %5d %5d\n", a, a + 1, b);

printf("%5d %5d %5d %5d\n", a, a += 1, b, a);

return 0;

}

**Output:**

127 128 -128

127 -128 -128 -128

C does not specify the order in which function parameters are processed. Therefore, some C compilers might look at the a +=1 parameter first and therefore change the value of other parameters that might reference a. A good rule to follow is to never have an expression in a parameter that might actually change the value of the parameter before it is passed to a function.

**Note: Here is a more thorough explanation:**

First remember that a char is an 8-bit signed number that is represented as an ASCII numeric value. That means that 7 bits are used to represent the magnitude of the number (0 to 127) and the 8th leftmost bit is used to represent the sign of the number.

**First line of output:**

a is assigned 127 (0111 1111) so the value for a is output as 127.

Next, 1 is added to a and is assigned to a temp variable (a itself is not modified). So 127 + 1 = 128 and therefore 128 is output.

Next, the value of b is output. Now b has been assigned 128 which, in binary, is 1000 0000. This binary value is placed into the char variable b. The problem is that 1000 0000 is now interpreted as a negative number being represented in its 2's complement notation, which equals -128. This number is output.

So the first line of output is 127, 128, -128.

**Second line of output:**

What this is showing is that you can never be certain the order in which the arguments to a function are looked at. Here, it appears that the arguments are evaluated and looked at from right to left.

So the 4th argument, a, is output as 127.

The 3rd argument, b, is output as -128 (discussed above).

The 2nd argument, a += 1, increments the value of a by 1. Now the value of a is 0111 1111 so adding 1 gives you 1000 0000. Note the overflow into the sign bit once again. This number is now interpreted as a negative number being represented in its 2's complement notation. Taking the 2's complement of the number gives us 128 so therefore the new value is -128, which is what gets output.

The 1st argument is output as -128 because the value of a has already been changed from within the 2nd argument which was evaluated before the 1st argument.

So the second line of output is -128 -128 -128 127

Lesson: Never assume the arguments of a function will be evaluated in any specific order. The next release of a compiler might change how things are done. The process of the order of evaluating function arguments is undefined by C and should therefore be avoided. Operations such as a += 1 should never appear within a function argument.