1. For the following C array,

    int a[3][2];

assume you are working with a 32-bit little-endian processor with the usual alignment rules (e.g., a Pentium) and

(a) Show how its elements are laid out in memory.
(b) Write the address expression for accessing a[i][j].
(c) Verify parts a) and b) by writing a small C program that contains and accesses such an array and looking at the assembly language output with the C compiler’s -S flag (e.g., gcc -O -S array.c. Turn in a copy of your C program and an annotated version of the assembly listing. Make sure the assembly listing is no more than 40 lines.

2. In an assembly-language-like notation (e.g., use MIPS or a pseudocode of your own choosing), write what an optimizing compiler would produce for the following two switch statements.

switch (a) {
    case 1: x = 7; break;
    case 2: x = 1; break;
    case 3: x = 8; y = 42; break;
    case 4: y = 17; break;
    case 5: z = 18; break;
    default: z = 3; break;
}

switch (b) {
    case 1: a = 3; break;
    case 10: a = 9; break;
    case 100: b = 14; c = 12; break;
    case 1000: c = 23; break;
    default: c = 20; break;
}

3. For a 32-bit little-endian processor with the usual alignment rules, show the memory layout and size in bytes of the following three C variables.

    union {
        int a;
        struct {
            char b; /* 8-bit */
            int c; /* 32-bit */
        } s;
    } u1;

    struct {
        char a;
        short c;
        char b;
        int d;
    } s1;

    struct {
        char a;
        short b;
        int c;
        char d;
    } s2;

4. Consider the following C-like program.

    int w = 5;
    int x = 6;

    int incw() { return ++w; }
    int incx() { return ++x; }

    void foo(y, z){
        printf("%d\n", y + y);
        x = 1;
        printf("%d\n", z);
    }

    int main() {
        foo(incw(), incx()); return 0;
    }

What does it print if the language uses

(a) Applicative-order evaluation?
(b) Normal-order evaluation?