1. For the following C array,
   ```c
   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 an expression for calculating the address of
       the starting byte of `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
       about 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.
   ```c
   switch (a) {
     case 1: z = 3; break;
     case 3: x = 1; break;
     case 4: x = 7; break;
     case 6: x = 5; y = 5; break;
     case 7: y = 4; x = 4; break;
     default: z = 4; break;
   }
   
   switch (b) {
     case 2: a = 42; break;
     case 20: a = 2; break;
     case 110: c = 5; break;
     case 893: b = 2; c = 3; break;
     default: c = 32; 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.
   ```c
   union {
     struct {
       short x; /* 16 bits */
       char y; /* 8 bits */
     } s;
     int z; /* 32 bits */
   } a;
   
   struct {
     char u;
     short v;
     char w;
     char x;
     char y;
     short z;
   } b;
   
   struct {
     int w;
     short x;
     short y;
     char z;
   } c;
   ```

4. Consider the following C-like program.
   ```c
   int w = 5;
   int x = 7;
   
   int incw() { return ++w; }
   int incx() { return ++x; }
   
   void foo(y, z){
     printf("%d\n", y + 1 + y);
     x = 11;
     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?