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 the expression for calculating 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 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 4:  z = 2; break;
   case 5:  x = 1; break;
   case 6:  x = 8; break;
   case 8:  x = 17; y = 10; break;
   case 9:  y = 3; x = 5; break;
   default: z = 5; break;
   }
   ```

   ```c
   switch (b) {
   case 2:  a = 18; break;
   case 20: a = 2; break;
   case 108: b = 7; c = 10;
   case 254: c = 8; break;
   default: c = 17; 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 a; /* 16 bits */
         char b; /* 8 bits */
      } s;
      int c; /* 32 bits */
   } u1;
   ```

   ```c
   struct {
      char a;
      short b;
      short c;
      short d;
      char e;
   } s1;
   ```

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

4. Consider the following C-like program.

   ```c
   int w = 9;
   int x = 3;
   ```

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

   ```c
   void foo(y, z){
      printf("%d\n", y + 1 + y);
      x = 12;
      printf("%d\n", z);
   }
   ```

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

   What does it print if the language uses

   (a) Applicative-order evaluation?

   (b) Normal-order evaluation?