1. State whether constant folding can be performed by the compiler in the following statements (Y or N)

```c
function : void foo()
{
    const int a = 5;
    int b = 17;
    const int c = a + 10;  // ______
    int[3 + a] d;          // ______
    int e = d[a + c];      // ______
    d[5 - 2 + c] = e;      // ______
    e = d[13 + b];        // ______
    e = d[b + c];         // ______
    d[13 + (a * b)] = e;  // ______
    e = d[d[2] + c];      // ______
}
```

2. Show the memory layout of the following C struct/record definition taking into consideration the SPARC data type memory alignment restrictions discussed in class. Fill bytes in memory with the appropriate struct/record member/field name. For example, if member/field name `p` takes 4 bytes, you will have 4 `p`'s in the appropriate memory locations. If the member/field is an array, use the name followed by the index number. For example, some number of `p[0]`s, `p[1]`s, `p[2]`s, etc. Place an `X` in any bytes of padding. Structs and unions are padded so the total size is evenly divisible by the most strict alignment requirement of its members.

```c
struct foo {
    int a;
    double b;
    short c[5];
    int d;
    char e;
};

struct foo fubar;
```

- What is the `offsetof( struct foo, c[4] )`? ______
- What is the `sizeof( struct foo )`? ______
- If `struct foo` had been defined as `union foo` instead, what would be the `sizeof( union foo )`? ______
3. Give an example of a converting type cast (underlying bit pattern does change).

Give an example of a non-converting type cast (underlying bit pattern does not change).

4. Use of typedefs in Reduced-C to define composite types

Using Reduced-C syntax, define an array of array of int named $foo$ with dimensions 5 x 10 (5 rows, 10 cols) such that $foo[4][9]$ is a valid index expression. This will take two lines of code.

What question would you like to see on the Midterm?