1. State whether each arithmetic expression will be performed at compile time or run time in the following.

```c
function : void foo() /* Reduced-C. Follow this quarter's Spec */
{
    int a = 5;
    const int b = 17;
    const int c = 7 + b;
    int[c / 6] d;
    int e = d[c + a];
    d[5 - 2 + c] = e;
    e = d[foo1() + b]; /* Assume: function : int foo1() { return 2; } */
    e = d[b + c];
    d[31 - (a * b)] = e;
    e = d[d[0] + b];
}
```

1) Run time
2) Compile time

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 { char a; short b; double c; int d[3]; char e; };
struct foo fubar;
```

What is the `offsetof( struct foo, d[1] )`? ______

What is the `sizeof( struct foo )`? ______

If `struct foo` had been defined as `union foo` instead, what would be the `sizeof( union foo )`? ______
3. What is the difference between a function declaration and a function definition in C?

Give an example of each in C using the function name fubar.

Function declaration _____________________________________________________________

Function definition _____________________________________________________________

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

Using Reduced-C syntax, define an array of 7 pointers to int named foo such that

```c
int x = 42;
foo[6] = &x;
x = *foo[6];
```

are valid expressions. This will take two lines of Reduced-C code.

What question would you like to see on the Midterm?