1. Consider the following datatype definition on a 32-bit x86 machine:

```c
typedef struct {
    char c;
    double *p;
    int i;
    double d;
    short s;
} struct1;
```

a. Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a structure of type `struct1`. Mark off and label the areas for each individual element (there are 5 of them). Cross hatch the parts that are allocated, but not used (to satisfy alignment). Assume the following alignment rule: data types of size $x$ must be aligned on $x$-byte boundaries. **Clearly indicate the right hand boundary of the data structure with a vertical line.**

```
0   1   2   3   4   5   6   7  8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
|                                                                                                                               |
```

b. How many bytes are allocated for an object of type `struct1`?

c. What alignment is required for an object of type `struct1`? Consider the implications of a declaration of the form `struct1 foo[4];`

d. If we define the fields of `struct1` in a different order, we can reduce the number of unused, allocated bytes in the best case?

2. The following declaration defines a class of structures for use in constructing binary trees:

```c
typedef struct ELE *tree_ptr;
struct ELE {
    long val;
    tree_ptr left;
    tree_ptr right;
}
```

For a function with the following prototype:

```c
long trace(tree_ptr tp);
```

gcc generates the following x86_64 code:
1 trace:
    /* tp in %rdi */
2    movl $0, %eax
3    testq %rdi, %rdi
4    je .L3
5 .L5:
6    movq (%rdi), %rax
7    movq 16(%rdi), %rdi
8    testq %rdi, %rdi
9    jne .L5
10 .L3:
11    rep
12    ret

a. Generate a C version of the function, using a while loop.
b. Explain in English what this function computes.

3. In this problem, let REF(x.i) --> DEF(x.k) denote that the linker will associate an arbitrary reference to symbol x in module i to the definition of x in module k. For each example below, use this notation to indicate how the linker would resolve references to the multiply defined symbol in each module. If there is a link-time error (rule 1), write “ERROR.” If the linker arbitrarily chooses one of the definitions (rule 3), write “UNKNOWN.”

a. /* Module 1 */ /* Module 2 */
   int main()
   { int main=1;
     int p2();
   }

   (a.1) REF(main.1) --> DEF(______._____
   (a.2) REF(main.2) --> DEF(______._____

b. /* Module 1 */ /* Module 2 */
   int x;
   double x;
   void main()
   { int p2();
     
   }

   (b.1) REF(x.1) --> DEF(______._____
   (b.2) REF(x.2) --> DEF(______._____

c. /* Module 1 */ /* Module 2 */
   int x=1;
   double x=1.0;
   void main()
   { int p2();
     
   }

   (c.1) REF(x.1) --> DEF(______._____
   (c.2) REF(x.2) --> DEF(______._____)
4. Consider the following program, which consists of two object modules:

```c
/* foo6.c */
void p2(void);

int main()
{
    p2();
    return 0;
}

/* bar6.c */
#include <stdio.h>

char main;

void p2()
{
    printf("0x%x\n", main);
}
```

When this program is compiled and executed on a Linux system, it prints the string “0x55\n” and terminates normally, even though `p2` never initializes variable `main`. Can you explain this?

5. Let `a` and `b` denote object modules or static libraries in the current directory, and let `a \rightarrow b` denote that `a` depends upon `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references:
   a. `p.o \rightarrow libx.a \rightarrow p.o`
   b. `p.o \rightarrow libx.a \rightarrow liby.a` and `liby.a \rightarrow libx.a`
   c. `p.o \rightarrow libx.a \rightarrow liby.a` and `liby.a \rightarrow libz.a`