P1(a) 15 pts
printf() will output 10 (for x) and the address of x (in hexadecimal notation) which is contained in y.
8 pts
Assignment statement *z = 3 will likely trigger a segmentation fault since a valid address has not been stored in z.
7 pts
P1(b) 15 pts
g is a function that takes a single argument that is a pointer to char (i.e., char *), and g returns a pointer to char (i.e., address that points to char).
4 pts
h is a function pointer that takes a single argument that is a pointer to char, and h returns a value of type char.
4 pts
x is a pointer to char, i.e., char *x.
3 pts
y is a function that takes an argument that is a pointer to char and returns a value of type char, i.e., char y(char *).
4 pts
P2(a) 15 pts
Calling fun() will likely generate a stack smashing error.
5 pts
This is so since x is local to fun() and overflowing the 1-D array (by 3 elements, i.e., 12 bytes) is likely to cause the canary (bit pattern) inserted by gcc (to guard the return address) to be changed.
5 pts
If x is made global, gcc does not insert a canary, hence stack smashing will not occur. However, overflowing x may, or may not, trigger a segmentation fault.
5 pts
P2(b) 15 pts
fopen() may fail.
4 pts
fscanf() may overflow 1-D array r if the character sequence in data.dat exceeds 100 bytes.
5 pts
f = fopen("data.dat", "r");
if(f == NULL) {
    printf("error opening data.dat");
    exit(1);
}
3 pts
fscanf(f, "%99s", r);
// fscanf(f, "%100s", r) is fine as well.
3 pts
P3(a) 20 pts
int main() {
    int **d;
    2 pts
    int N, M;
    int i, j;
```c
scanf("%d %d", &N, &M);
d = (int **) malloc(N * sizeof(int *));
// d = malloc(N * sizeof(int *)) or malloc(N * 8) are fine too.
6 pts
    for(i=0; i<N; i++)
        *(d + i) = (int *) malloc(M * sizeof(int));
// Imitting (int *) and using constant 4 in place of sizeof(int) are fine too.
6 pts
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            scanf("%d", &d[i][j]);
6 pts
}

P3(b) 20 pts

unsigned int x, m;
int i, count = 0;
scanf("%u", &x); // Read unsigned int input.
2 pts
m = ~(~0 << 1); // Set mask to 000...01
// m = 1 is fine too.
6 pts
    for(i=0; i<32; i++) {
        if((x & m) == 0) // If bit value at first position is 0 increment count.
            count++;
6 pts
        x = x >> 1; // Shift bits of x to the right by one position.
6 pts
    }
printf("%d", count);
// Printing count can be omitted.

Bonus 10 pts

printf() only needs a copy of the value of x to do its work of printing the value to stdout.
3 pts
scanf() needs the address of x so that the value entered through stdin (by default, keyboard) can be stored at the address of x.
3 pts

Yes, since following the address of x allows printf() to access its value.
2 pts
It is not necessary to reveal the address of x to printf() since it only requires its value.
2 pts
```