P1(a) 14 pts
When the called function (callee) only needs to read but does not need to modify the argument values of the calling function (caller).
4 pts
When the called function (callee) does need to modify the argument values of the calling function (caller).
4 pts
Yes. Rewrite printf() so that arguments are defined as pointers. printf() can use the passed pointers to both read and modify the caller's memory locations.
6 pts
P1(b) 14 pts
Store arguments passed.
2 pts
Store local variables.
2 pts
The stack frame is deleted (or pop'ed).
2 pts
x resides in somefunc's stack frame, y resides elsewhere in a data area that is not deleted when the function returns.
4 pts
1 and 1.
4 pts
P1(c) 14 pts
z contains an address, i.e., is a pointer to int. Call this address A. At A, 5 consecutive memory locations to hold values of type int are allocated.
5 pts
Since z is a pointer to int, the word size is 4 bytes. z+2 adds to the content of z (i.e., the address A) 2 in unit of a word. Hence we get address A+2. As noted above, at A+2, the value of z[2] is placed.
To access the content of address A+2, we prepend with * in C, thus *(z+2).
5 pts
scanf("%d",&z[0]); to read z[0]. Similarly for z[1], ..., z[4].
4 pts
P2(a) 10 pts
x remains 5. Although the pointer y is assigned the address of x, in the following statement we assign it the address 10. In the statement *y = 15 we go to address 10 and store as content the integer 15. Unless (by accident) the address of x happens to be10 (unlikely), the value of x remains 5.
P2(b) 10 pts
In the first case, myfunc() is a function that takes two integer arguments and returns a pointer to char. 3 pts
In the second case, myfunc() is a function pointer to a function that takes two integer arguments and returns a value of type char. 3 pts
Case 1:
char *a;
a = myfunc(2,3); 2 pts
Case 2:
char a, yourfunc(int, int);
myfunc = &yourfunc;
a = (*myfunc)(2,3);  // a = myfunc(2,3) will also work 2 pts
P2(c) 10 pts
main() prints 55. The reason is, in the assignment x = 55 in main(), x is the global variable defined in the file mycode.c. When main() calls abc(), the assignment x = 77 affects the local variable x of abc() which happens to have the same name as the global x but they are distinct. 7 pts
Compilation will fail since #include <stdio.h> is omitted. 3 pts
P3(a) 14 pts
*a = 10.2 likely results in segmentation fault since we have not allocated memory space for the float pointed to by a. 4 pts
b[3] is out of bounds (and may result in segmentation fault) since the declaration float b[3] allocates memory up to b[2]. 3 pts
a = &b is a mismatch of types. b, being a 1-D array, is a pointer to float. Therefore &b is a pointer to a pointer. Since a is a pointer to float, we have a mismatch which usually results in a bug. 4 pts
b[3] = 5.5 and a = &b are easy to detect at compile time since b is declared as float b[3] (hence maximum usage is b[2]) and the types of a and b (and therefore &b) are known at compile time. 2 pts
*a = 10.2 may also be detected during compilation but it may be a bit more tricky since perhaps dynamic memory allocation was performed elsewhere that the compiler missed. Either answer is fine for this statement. 1 pts
P3(b) 14 pts
scanf() reads to value of x
1 pts

~0 creates all 1's
1 pts

~0 >> 1 shifts all 1's to the right by one bit and inserts a 0 in the most significant bit
2 pts

~(~0 >> 1) complements or flips the bits which results in the most significant bit being 1, all other bits being 0; this is stored in the mask m
2 pts

the bits of x and mask m are bit-wise AND'ed and stored in y which implies that y is either all 0's (if the most significant bit of x is 0) or 10000...0000 (if the most significant bit of x is 1).
3 pts

Since the most significant bit of integer x specifies its sign, y !=0 is true if y = 10000...0000 which means that x is negative. If so, the code prints the string "neg".
3 pts

(~0) << 31
2 pts

Bonus
h is a pointer to a pointer of type float, i.e., float **h and at *h we find 3 consecutive pointer locations: *h, *(h+1), *(h+2). If we follow *h we find 5 consecutive location for storing h[0][0], h[0][1], ..., h[0][4] whose float values are accessed by **h, *(h +1), ..., *(*(h+4)). Similarly for the others.
5 pts

Thus *(h+1)+2) is equivalent to h[1][2], and *(*(h+2)+1) is equivalent to h[2][1].
3 pts

When we do not know the size of a 2-D array at compile time, it may be more efficient to declare a double pointer and allocate memory at run time dynamically.
2 pts