P1(a) 12 pts

gcc only runs the C preprocessor which handles all statements starting with #. (The resultant "pure" C code is sent to stdout.)
3 pts

gcc generates object code (ending with postfix .o) which is machine code that has not yet been linked.
3 pts

In static linking, all functions that are part of the same C program are linked together by gcc and put into an executable (by default, a.out).
3 pts

The shell app (e.g., bash) will invoke the help of the operating system (e.g., Linux) who loads the executable into main memory (i.e., RAM) so that a CPU can start executing the instructions of the now running program (i.e., process).
3 pts

P1(b) 12 pts

The first part, %s paired with s, prints 'A' to stdout (by default, display/terminal/screen) since '\0' signifies end of string.
6 pts

The second part prints "abcd" to stdout, possibly followed by additional characters. Since end of string '\0' is not provided, what happens depends on what the memory locations following s[3] contain and how printf() responds to what it finds.
6 pts

P1(c) 12 pts

It is likely that the running program will terminate with a segmentation fault error without printing anything since *z = 30 is likely to result in accessing memory that does not belong to the process.
This is so since z is a pointer and what address is contained at z is undetermined.
6 pts

printf() may succeed in printing 20 and 20, the values of x and *y
(not that *y = 20 changed the value of x), but attempting to print *z is likely to result in segmentation fault since address 30 may not belong to the running process. If, by chance, it does, then what gets printed is uncertain since we do not know what is contained at address 30.
6 pts

P2(a) 17 pts

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>503</td>
<td>3</td>
</tr>
<tr>
<td>502</td>
<td>2</td>
</tr>
<tr>
<td>501</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In the above example, d is a symbol that represents address 200. At address 200, we find its content to be another address 500. At the four addresses starting at 500, we find the integers 0, 1, 2, 3 stored consecutively.
10 pts

*(d+2) = 2 is the equivalent statement. At address 200 (which d represents), we find 500 (another address since d is pointer) to which we add 2. The resulting address, 502, contains the integer value 2 which is what d[2] represents.
7 pts

P2(b) 17 pts

The stack memory is used to store arguments passed by the caller (i.e., calling function), the return address, and local variables of the callee (i.e., called function).
6 pts

The process, when trying to return from myfunc() back to main(), is likely to terminate with an error message indicating stack smashing. This is so since m[4], m[5], ..., m[9] in the for-loop are likely to overwrite the canary inserted by gcc thereby changing its value. Before returning from myfunc(), the code inserted by gcc detects that the canary has been modified (i.e., it is "dead") which results in termination.
Without the canary and additional security checking code inserted by gcc, the process may crash with a segmentation fault if \( m[4], \ldots, m[9] \) overwrote the return address with a value which, interpreted as an address, does not belong to the process. If the return address has not been overwritten, then `myfunc()` will return to `main()` without crashing.

5 pts

P3(a) 15 pts

\[ *(u+1)+2 = '\0' \]

is equivalent to \( u[1][2] = '\0' \). \( u \) contains an address to a block of two addresses. The first address in the block points to block of three addresses containing 'g', 'o', '\0' which is a 1-D character array.

The second points to a block of three addresses containing 'P', 'U', '\0'. Thus the second address is \( u+1 \). The third address in the block pointed to by \( u+1 \) is \( *(u+1)+2 \). The content at that address, \( *(u+1)+2 \), is set to '\0'.

8 pts

The argument passed to `printf()` to print "PU" is \( *(u+1) \). This is so since we need to pass to `printf()` the beginning address of a block of three address where 'P', 'U', and '\0' are stored. As discussed above, this beginning address is \( *(u+1) \).

7 pts

P3(b) 15 pts

***\( d \) means that its content, **\( d \), is another address. **\( d \) means that its content, *\( d \), is yet another address. And the address contained in \( d \), finally contains an integer.

7 pts

```c
int a, *b, **c, ***d;
b = &a;
c = &b;
d = &c;
***d = 7;
printf("%d", ***d);
```

8 pts

Bonus 10 pts

Global variables are stored in an area of memory just above the code (i.e., text). Local variables of a function are stored in the stack memory allocated to the function when it is
When somefunc() returns, z is deleted since the stack memory allocated to somefunc() is deleted.

What value z will contain depends on where in memory the stack memory of second call to somefunc() is allocated. In general, this is not possible to predict.