What is the value of n after the following code is executed?

```
int n = 98;
int *p = &n;

*p++;
```

The code you've presented involves pointer arithmetic, and it's important to understand the order of operations to determine the final value of `n`. Let's break it down:

1. `int n = 98;` - This line initializes an integer `n` with the value `98`.
2. `int *p = &n;` - This line declares a pointer `p` and initializes it with the address of `n`.
3. `*p++;` - This line involves both dereferencing and incrementing the pointer.
    4. `*p++` is equivalent to `*(p++)` due to the operator precedence rules in C/C++. The postfix increment `++` has a higher precedence than the dereference operator `*`.
    5. This means the pointer `p` is incremented first, but the dereference occurs on the original location of `p` (which is the address of `n`).
    6. The increment affects the pointer `p`, not the value pointed to by `p`. Therefore, `n` remains unchanged.

So, after the execution of this code, the value of `n` is still `98`. The pointer `p` will be pointing to some memory location after the location of `n`, but `n` itself is not modified by this operation.

How much space would you need to allocate for a list node with the following structure on a 64-bit machine?

```
/**
* struct list_s - singly linked list
* @str: string - (malloc'ed string)
* @len: length of the string
* @next: points to the next node
*
* Description: singly linked list node structure
* for your project
*/
typedef struct list_s
{
char *str;
unsigned int len;
```

```
    struct list_s *next;
} list_t;
```

To determine the amount of space needed to allocate for a list node with the given structure on a 64-bit machine, we need to consider the sizes of each of the members of the structure:

1. `char *str`: This is a pointer to a character. On a 64-bit system, pointers are typically 64 bits (or 8 bytes) in size.
2. `unsigned int len`: The size of an `unsigned int` can vary depending on the system and compiler, but it is commonly 4 bytes on both 32-bit and 64-bit systems.
3. `struct list_s *next`: This is a pointer to another structure of the same type. Like `str`, it will also be 8 bytes in size on a 64-bit system.

Adding these together, we get:

- Size of `str`: 8 bytes
- Size of `len`: 4 bytes
- Size of `next`: 8 bytes

Total: 20 bytes

However, there's one more factor to consider: padding. Structures in C are often padded to align their members to certain byte boundaries for performance reasons. On a 64-bit system, the structure is likely to be aligned to 8-byte boundaries.

In this case, after the `unsigned int len`, there might be padding added to align the `next` pointer on an 8-byte boundary. This would add an additional 4 bytes of padding after `len`. Thus, the total size would be 24 bytes (8 bytes for `str`, 4 bytes for `len`, 4 bytes for padding, and 8 bytes for `next`).

Keep in mind that this is a typical case for many 64-bit systems, but the exact size can vary depending on the system and compiler specifics, such as the structure packing and alignment settings.