Declaration of pointers and the new operator

Here is a question on the declaration of pointers and the use of the new operator in Display 9.2 and Display 9.3:

> I'm confused about the need for the `new` operator:
>
> In Display 9.2, there is a definition for `*p1` and `*p2` as `int`. So pointers `p1` and `p2` have thus been defined / created - Line 7
>
> Then in line 9, `p1 = new int`. How is this different to line 7? Why the need to define this if `p1` has already been created in line 7?
>
> The only place where it seems reasonable is in line 17 where a new pointer address `p1` is created. Following Display 9.3 also leads me to the above reasoning:
>
> (d) `p2` was created like `p1` in line 7, and assigned to the address of `p1`
>
> only (g) makes sense to create a new place to sort another number.
>
> The new operator definition at the bottom of page 548 is also just as confusing.
>
> Please clarify?

Answer:

A pointer variable is a variable that can hold the address of another variable. When we declare a pointer variable

```
int* p1;
```

we specify that `p1` can hold the address of an integer variable. It does not yet point to any integer, in the same way that when you declare an `int` variable `I` without assigning a value to it, we do not know what `I`'s value is. If we declare

```
int I;
```

we do not know what the value of `I` is, since we have not assigned a value to it. It can have any value. This is why we will always initialise `I` to 0 if want to use it to count.

Similarly, when we declare a pointer variable `p1`, it does not yet point to another variable. Only when we use the `new` operator, we 'assign' a value to the pointer variable `p1` in our example, i.e. let it point to a space in memory reserved for an integer, as (b) in Display 9.3 indicates. We also do not know what value is inside the memory location to which `p1` points, unless we assign a value to that location as with the statement

`*p1 =42;` //(see (c) in Display 9.3).

In (d) the statement `p1 = p2;` means that `p2` will point to the same location as `p1`, i.e. to the same `int` variable that `p1` is pointing to. See the two arrows both pointing to the same memory location where 42 is stored.

The *address* of `p1` is not the same as the memory location to which `p1` is pointing. The memory location to which `p1` is pointing can be seen as the *value* of `p1`. This is represented by the arrow that points to a memory location from the box that represents `p1` in (b) in Display 9.3.

The address of p1 is represented by the box with the question mark in in (a) in Display 9.3.

So, to get back to the original question about the **difference between line 7 and line 9 in Display 9.2**:

In line 7 we declare two pointer variables, without initialising them, i.e. they do not point to any memory location yet. We just tell the compiler what their names are (`p1` and `p2`), what type of values they can hold (pointers to `int` variables), and reserve space in memory for `p1` and `p2` to store the addresses of the `int` variables to which `p1` and `p2` can point.

In line 9 we assign a value to `p1` with the `new` operator, i.e. we reserve memory for an `int` variable to which `p1`points. That `int` variable also do not have a value yet, although we have now reserved space in memory for it. The only 'name' we have to refer to that `int` variable is `*p1`.

In line 10, we use the name for the variable to which `p1` points, `*p1`, to assign a value to the `int` variable to which `p1` points, in the statement

```
*p1 =42;
```

In Display 9.2 we see two ways in which to assign a value to a pointer, the `new` operator (line 9) is one of them and having a pointer point to the same location as another pointer (line 11) is the other way. Assigning a null value to a pointer is a third way of assigning a value to a pointer (see the additional notes on 'Dangling Pointers and the Null Value' under Additional Resources).

A pointer variable is a **dynamic variable** because we can both *reserve* memory with a pointer variable using the `new` operator *and release or deallocate* the memory with the `delete` operator while a program is executing. The deallocated memory can then be used by another dynamic variable during the same program execution. This is in contrast to ordinary variables for whom memory is reserved when they are declared in the program. This memory will remain reserved during program execution and only be released once the program ends.