Adding 1 to a pointer variable increases its value by the number of bytes of the type to which it points.

Now consider the array expression stacks[1]. The C++ compiler treats this expression exactly as if you wrote it as *(stacks + 1). The second expression means calculate the address of the second element of the array and then find the value stored there. The end result is precisely what stacks[1] means. (Operator precedence requires that you use the parentheses. Without them, 1 would be added to *stacks instead of to stacks.)

The program output demonstrates that *(stacks + 1) and stacks[1] are the same. Similarly, *(stacks + 2) is the same as stacks[2]. In general, wherever you use array notation, C++ makes the following conversion:

arrayname[i] becomes *(arrayname + i)

And if you use a pointer instead of an array name, C++ makes the same conversion:

pointernameli becomes *(pointername + i)

Thus, in many respects you can use pointer names and array names in the same way. You can use the array brackets notation with either. You can apply the dereferencing operator (*) to either. In most expressions, each represents an address. One difference is that you can change the value of a pointer while an array name is a constant:

pointernameli = pointernameli + 1; // valid
arraynameli = arraynameli + 1;     // not allowed

The second difference is that applying the sizeof operator to an array name yields the size of the array, but applying sizeof to a pointer yields the size of the pointer, even if the pointer points to the array. For example, in Listing 4.15, both pw and wages refer to the same array. But applying the sizeof operator to them produces the following results:

24 = size of wages array ® displaying sizeof wages
4 = size of pw pointer   ® displaying sizeof pw

This is one case in which C++ doesn't interpret the array name as an address.