Pointers and References:
In C++ (and C) we distinguish:

rvalues = the sorts of things that can go on the RHS (right-hand side) of assignment operator.

lvalues = the sorts of things that can go on the LHS of an assignment operator.

Since the purpose of an assignment operator is to change a location in memory, the only things that can go on the LHS are things that in some way translate into the address of a location in memory.

This is why "X = 2" is OK (because X is an lvalue),
but "2 = X" is not OK, because 2 is an rvalue, not an lvalue.

Pointers are addresses treated as rvalues, so you put them into other variables.

```
int x = 17;
int * xp; // declares a pointer variable, i.e., it can hold an address.

xp = &x;  // puts the address of x into xp
```

Why use pointers?

(1) Pointers often represent relationships between objects. For example, in an organization chart, the links (or edges) show who each person's supervisor is. If we were to do this in OOP, the natural way to do it would be to have a pointer from the object representing an employee to the object representing their supervisor.
Then we can go directly from the employee to their supervisor.

(2) Pointers are efficient because they allow direct access from one object to another. For example, if we didn't have a pointer from the employee's object to their supervisor's object, but only had something like a name or employee ID, we would have to search the database to find the supervisor's object.

Dynamic Memory Allocation:
C++ allocates memory in three different ways.

(1) Statically (doesn't change while program is running): it sets aside memory for the variable when the main program starts running. Examples: variables declared outside or within main().

(2) Stack storage: dynamically on the system stack. Every time you call a function, it allocates memory space for the variables declared in that function (local variables). So if the function is recursive, there may be multiple copies of the same variable. As functions return, their space is released so that it can be reused later for other function calls. This is LIFO storage. Because function calls are LIFO too, this is an efficient and simple way to allocate storage. This memory management is automatic (you don't have to worry about it).

(3) Heap storage. Heap = an area of memory, from which you can request chunks for holding different kinds of objects. When you are done with them, you can return them to the heap, so they can be reused. But there doesn't have to be any systematic order (such as LIFO) in the way you allocate and deallocate the storage. More flexible, but less efficient, because the system has to manage this heap storage. When you ask for a chunk of heap memory, you get a pointer to it. When you deallocate it, you use the pointer to say what you want to deallocate. This memory management is manual. You have to handle it, and you may make

mistakes (e.g., releasing storage before you are done using it).

To get a hunk of storage to hold an object of type T, use "new T" and it returns a T* (pointer to T). You can also use "new T (....)" if you want to pass parameters to the T constructor.

Vectors, arrays, stacks, lists are linear (they are in a line) data structures.
Pointers allow you to construct nonlinear (they are not in a line) data structures, such as trees (e.g., org. charts, genealogical trees), networks, graphs, etc.

Reference "variables" (they are not variable!).
Both "pointer" and "reference" are essentially synonymous with "address," but they are used differently in C++.
Reference variables basically declare aliases for other variables.

```
SOMETYPE x;
....
SOMETYPE & rx = x; // declares rx to be a reference to x
```

Anywhere I use rx (either on LHS or RHS of an assignment), it's the same as using x. The above makes rx an alias for x.

(1) reference variables (such rx) must be initialized
(2) they can't be reassigned to refer to alias another variable.

As a general rule it's not a good idea to have aliases for variables, because it makes programs hard to understand.
If you are looking for everywhere x get clobbered, you might notice assignments to rx.