Pointers and Recursive Types

pointer: a variable (or field) whose value is a reference to some object.

A recursive type is one whose objects may contain one or more references to other objects of the type. Most recursive types are records, since they need to contain something in addition to the reference, implying the existence of heterogeneous fields. Recursive types are used to build a wide variety of "linked" data structures, includinglists and trees.

In languages that use a reference model of variables, it is easy for a record of type `foo` to include a reference to another record of type `foo`: every variable is a reference anyway. In languages that use a value model of variables, recursive types require the notion of a **pointer**. Pointers we first introduced in PL/I.

In some languages, pointers were restricted to point only to objects in the heap. The only way to create a new pointer value (without using variant records or casts to bypass the type system) was to call a built-in fuction that allocated a new object in the heap and returned a pointer to it. In other languages, one can create a pointer to a nonheap object by using an "address of" operator.

In any language that permits new objects to be allocated from the heap, the question arises: how and when is storage reclaimed for objects that are no longer needed? In short-lived programs it may be accetable simply yo leave the storage unused, but in most cases unused storage must be reclaimed, to make room for other things. A program that fails to reclaim the space for objects that are no longer needed is said to "leak memory". If such a program runs for an extended period of time, it may run out of space and crash.

Some languages, including C, C++, and Rust, require the programmer to reclaim space explicitly. Other languages, including Java, C#, Scala, Go, and all the functional and scripting languages, require the language implementation to reclaim unused objects automatically. Explicit storage reclamation simplifies the language implementation, but raises the possibility that the programmer will forget to reclaim objects that are no longer live (therby leaking memory), or will accidentally reclaim objects that are still in use (therby creating dangling references). Automatic storage reclaimation (otherwise known as **garbage collection**) dramatically simplifies the programmer's task, but imposes certain run-time cost, and raises the question of how the language implementation is to distinguish garbage from active objects.