symbol table
(source: Louden, Ch. 5, pp. 128-129)

- "bindings must be MAINTAINED by a translator so that appropriate meanings are given to names during translation and execution."

- "A translator does this by creating a data structure to maintain this [binding] information."

- This data structure is usually called the symbol table

- So, a symbol table can be thought of as mapping names to attributes...
object lifetimes

(source: Scott, Ch. 3, p. 115)

• "The period of time between the creation and the destruction of a name-to-object binding is called the binding's lifetime."

• "the time between the creation & destruction of an object is the object's lifetime."

• "These need not necessarily coincide" --
  – "an object may retain its value and the potential to be accessed even when a given name can no longer be used to access it"
  – e.g., a pass-by-reference parameter -- "the binding between the parameter name and the variable that was passed has a lifetime shorter than the variable itself"
  – usually a bug if "a name-to-object binding has a lifetime longer than that of the object" (dangling references, anyone?)
storage allocation mechanisms
(source: Scott, Ch. 3, p. 115)

- "Object lifetimes generally correspond to one of three principal storage allocation mechanisms, used to manage the object's space:

1. **Static** objects are given an absolute address that is retained throughout the program's execution.

2. **Stack** objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.

3. **Heap** objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm."
storage allocation: heap

• (Louden, p. 164) "the environment must have an area in memory from which memory can be allocated ... and ... returned in response..." to run-time allocation requests;

• "Such an area is traditionally called a heap (although it has nothing to do with the heap data structure)"

• Allocation using this heap is usually called dynamic allocation
  – (...even though allocation of local variables is also dynamic, in the sense that it actually occurs during execution;
  – ...but those local variables are allocated using a stack, and memory allocated in this way is usually called stack-based or automatic allocation)
where are these in memory?

- (Louden, p. 164) NOTE that the "automatic allocation" stack and the "dynamic allocation" heap are usually DIFFERENT sections of memory;

- ...and that first storage mechanism we mentioned, for static objects, are usually in a separate, static area;

- These three areas could be anywhere!
  - BUT one common strategy is to place them "adjacent to one another,
  - with the global area first,
  - the stack next,
  - and the heap last,
  - with the heap and stack growing in opposite directions"
how can heap storage be reclaimed? p. 1


- "Many computer languages require garbage collection, either as part of the language specification (e.g. Java, C#, and most scripting languages) or effectively for practical implementation (e.g. formal languages like lambda calculus); these are said to be garbage-collected languages."

- "Other languages were designed for use with manual memory management (e.g., C, C++)
  - but this Wikipedia article mentioned that there are garbage collected implementations of even C and C++...!"
how can heap storage be reclaimed? p. 2

• "Some languages, like Modula-3, allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects"

• And there are even languages, "like D, which is garbage-collected but allows the user to manually delete objects and also entirely disable garbage collection when speed is required."

– Perhaps similarly, Louden notes that Ada will let you call the garbage collector, or turn it off for certain variables?

– (because of its goal to be usable in real-time situations where control of the speed of execution of a program is critical --- Louden, p. 179)
reference counts (p. 1)

[source: MacLennan, Ch. 11, pp. 388-394]

• when something points to a cell -- increment its reference count;

• when a reference to a cell is destroyed -- decrement its reference count;

• "When a cell's reference count becomes ZERO, it means that the cell is inaccessible and can be returned to the free list."
reference counts (p. 2)

- MacLennan, p. 391: (pseudocode!!!)

decrement (C):

\[
\text{reference\_count}(C) := \text{reference\_count}(C) - 1
\]

if reference\_count(C) = 0 then

\[
\text{decrement } (C^\cdot\text{left});
\]

\[
\text{decrement } (C^\cdot\text{right});
\]

return C to free-list;

end if.
mark-sweep (p. 1)

[source: MacLennan, Ch. 11, pp. 388-394]

• in its simplest/most-naive form:
  
  – in the mark phase, the gc identifies all cells that ARE accessible, that are NOT garbage;

  – in the sweep phase, all of the cells that are left (and inaccessible) are made available, "often by placing them on the free list"
mark-sweep (p. 2)

mark phase:

for each root R, mark (R).

mark (R):

if R is not marked, then:

set mark bit of R;

mark (R^.left);

mark (R^.right);

endif.