==========
Pointers and recursive types

pointers serve two purposes:
- efficient (and sometimes intuitive) access to elaborated objects (as in C)
- dynamic creation of linked data structures, in conjunction with a heap storage manager

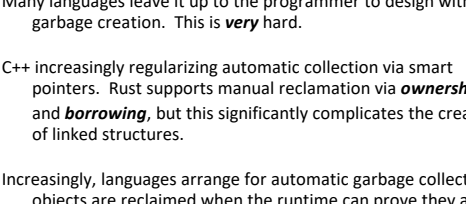
Note that pointers are ***not*** the same thing as addresses. Pointers are an abstraction. Addresses are an implementation. Pointers are ***not*** always implemented as addresses:
- machines with segments
- error checks (e.g. locks and keys -- see below)
- swizzling
- cursors
- C++ overloading of *, –> (e.g., for smart pointers)

Many languages restrict pointers to accessing things in the heap: the only way to get a pointer is by calling new. Others (e.g., C) allow you to create a pointer to any existing object.

Pointers are used with a value model of variables.
They aren't needed with a reference model.

Good implementations of languages with a reference model of variables represent primitive (immutable) types the same way you would for a language with a value model of variables -- you think of your variable x as a reference to "the" 3 (the Platonic ideal), but the compiler implements it as a box with a copy of "the" 3 in it.

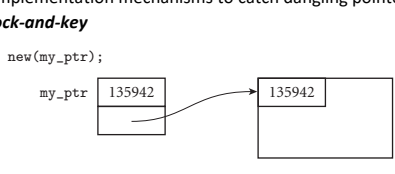
```
y := x
```

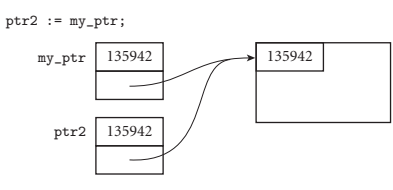


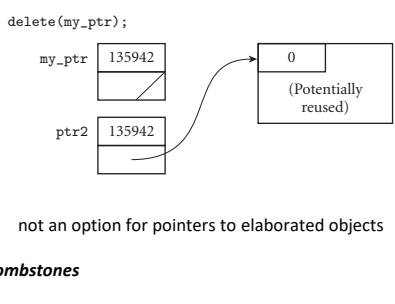
Problems:
- syntax of pointer dereferencing
  - typically explicit, as in C
  - a few languages dereference automatically, depending on context
    - Ada, for example, does implicit dereferencing for record field references, and has special syntax to name the entire referenced object

```
type foo is record ...
type fp is access foo
f : xp := new foo;
...
y := f.field1;      -- implicit dereference
g : foo := f.all;   -- whole object
```

- dangling pointers
  - due to
    - explicit deallocation of heap objects
      - only in languages that ***have*** explicit deallocation
    - implicit deallocation of elaborated objects
      - only in languages that let you create pointers to these
  - two implementation mechanisms to catch dangling pointers:
    - ***lock-and-key***

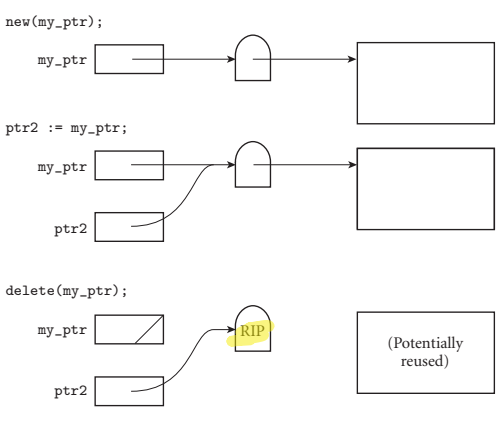






not an option for pointers to elaborated objects

tombstones



tombstones themselves live a long time, but can be garbage collected using reference counts; more later

---
Garbage collection

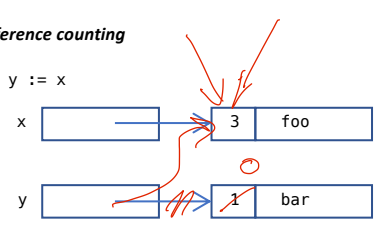
Many languages leave it up to the programmer to design without garbage creation. This is ***very*** hard.

C++ increasingly regularizing automatic collection via smart pointers. Rust supports manual reclamation via ***ownership*** and ***borrowing***, but this significantly complicates the creation of linked structures.

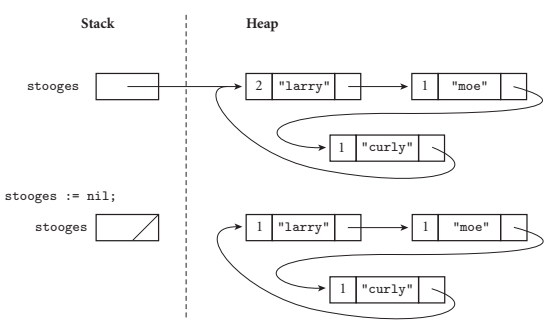
Increasingly, languages arrange for automatic garbage collection objects are reclaimed when the runtime can prove they are no longer accessible. (Note: this is ***not*** the same as no longer needed -- may be overly conservative.)

Two common implementations: reference counting and tracing

reference counting



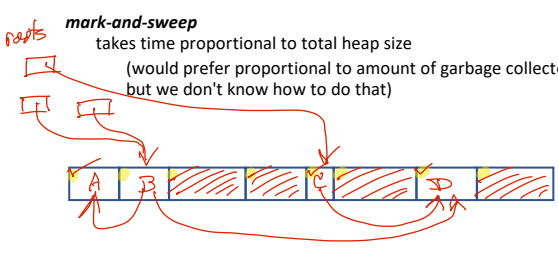
works great for strings
does not work for circular structures



does work for tombstones, though you have to make sure that when you delete a struct containing pointers (or allow it to go out of scope) the compiler decrements the reference counts of the tombstones for those pointers. Key observation is that tombstones are used with explicit object deletion: ref. counts fail to reclaim tombstone only when user fails to reclaim object.

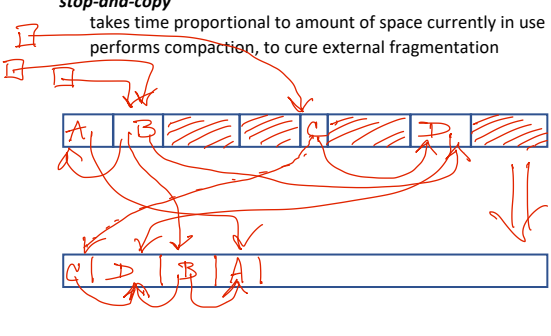
tracing
- generally requires strong typing
  - (but see conservative collection below)
- used routinely in Java, C#, Scala, Swift, Kotlin, Go,
  - Lisp, ML/OCaml/Haskell, scripting languages, ...
- variants

mark-and-sweep
- takes time proportional to total heap size
  - (would prefer proportional to amount of garbage collected, but we don't know how to do that)



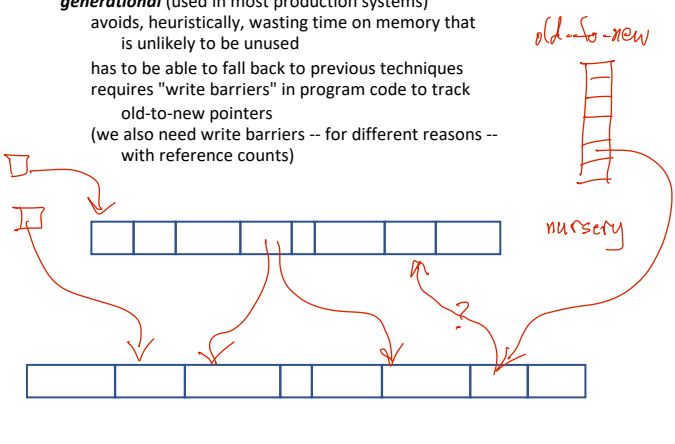
- can use ***pointer reversal*** for space-efficient tracing

stop-and-copy
- takes time proportional to amount of space currently in use
- performs compaction, to cure external fragmentation



- might be expected to double space requirements, but doesn't really, given virtual memory

generational (used in most production systems)
- avoids, heuristically, wasting time on memory that is unlikely to be unused
- has to be able to fall back to previous techniques
- requires "write barriers" in program code to track old-to-new pointers
- (we also need write barriers -- for different reasons -- with reference counts)



Conservative approximation possible in almost any language:

Assume any pointer-sized aligned value is a pointer if its bit pattern is the address of (the beginning of) a block in the heap.

Limitations:
- pointers to ***interior*** of objects not generally supported
- pointers must not be hidden (stored in any way other than a full-word aligned address)
- can leak storage when the address of an unneeded block happens to match the bit pattern of some non-pointer object.

hybrids also possible: e.g., reference count most of the time, do a mark-and-sweep once in a while to catch circular structures.

C pointers and arrays

The basic idea: an array variable is (in most respects) treated like a pointer to the array's first element; subscripting is defined in terms of pointer arithmetic:

```
E1[E2] == (*((E1)+(E2))) = (*((E2)+(E1))) !
```

So given
```
int n, *p;
```
You can say not only
```
n = p[3];
```
but also
```
n = 3[p];        // surprise!
```

Subscripting scales to the size of array elements in C precisely because pointer arithmetic does.

When is an array not a pointer?

(a) in a variable definition, where the array allocates space

(b) in a `sizeof`, where the array represents the whole thing

```
double A[10];
double *p = A;
sizeof(A) == 80        // the whole array
sizeof(A[0]) == 8      // one element
sizeof(p) == 4         // a pointer (on a 32-bit machine)
```


Variable definitions:
```
int *a[n]              // n-element array of row pointers
int a[n][m]            // 2-D array
```

Beware the difference between ***definitions***, which allocate space, and ***declarations***, which merely introduce names.

Since function prototypes (headers) are just declarations, and don't allocate space, and since arrays are passed as pointers, the following parameter declarations are equivalent:

```
int *a == int a[]      // pointer to int
int **a == int *a[]    // pointer to pointer to int
```

Note that these equivalences do ***not*** hold for definitions.

Compiler has to be able to tell the size of the things to which you point. So the following aren't valid, even as parameter declarations:

```
int a[][]        // bad
int (*a)[]       // bad
```

But `a[][10]` is ok, even as a parameter, and the compiler will do the right thing. `(*a)[10]` is equivalent as a parameter.

You ***can*** pass contiguous arrays to subroutines, but you have to specify the size of all inner dimensions:

```
int a[][10]      // ok (as declaration, not definition)
int (*a)[10]     // "; does the same thing
int a[10][10]    // also ok, but first 10 is unnecessary
```

C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat.

```
int *a[n]          // n-element array of pointers to integers
int (*a)[n]        // pointer to n-element array of integers

int (*f)(int *)  // pointer to function taking pointer to
                 // integer as argument, and returning integer
```

Choice between pointer arithmetic & subscripts is largely a matter of taste. Pointer arithmetic used to be faster with stupid compilers. With modern compilers it's often the other way around, particularly given the tendency of aliases to inhibit optimization.

Cf. choice between row-pointer and contiguous layout: tradeoff has reversed with time.