Notes for CSC 254, 16 and 21 Nov. 2022

==========

A smorgasbord of types

scalar types -- one or two-dimensional
- discrete -- one-dimensional and countable
  - integer, boolean, char, enumeration, subrange
- rational
- real
- complex

composite types
- records/structs/tuples
- variants/unions
- arrays
  - strings
- sets
- pointers
- lists
- files

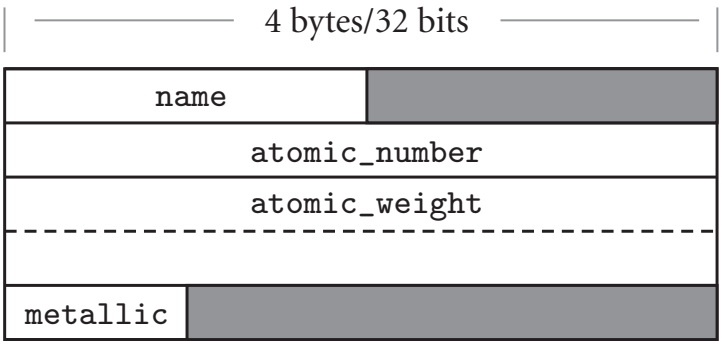
- mappings // common in scripting languages

Records
- usually laid out contiguously
- possible holes for alignment reasons
  - permits copying but ***not*** comparison with simple block operations

- example:

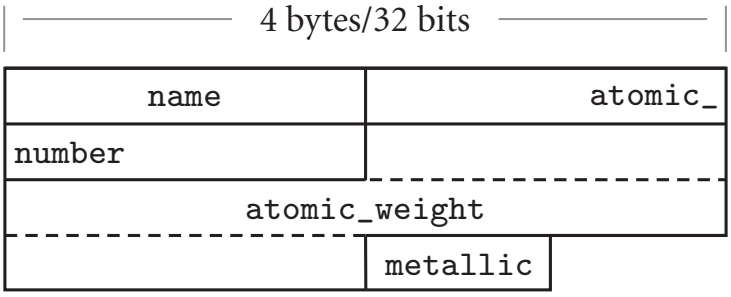
```
struct element {
   char name[2];
   int atomic_number;
   double atomic_weight;
   bool metallic;
}
```

layout on a 32-bit machine:



A few languages allow the programmer to specify that a record is ***packed***, meaning there are no (internal) holds, but fields may be unaligned.

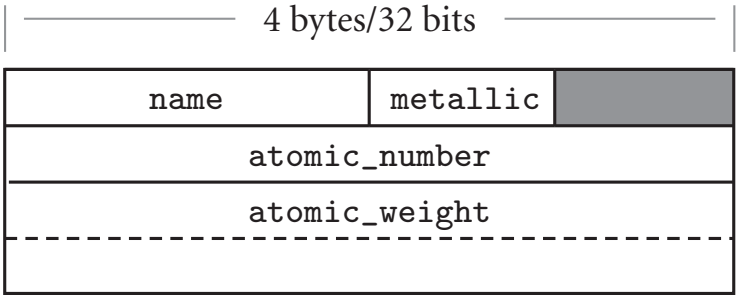
less space, but significant run-time access penalty



Smart compilers may re-arrange fields to minimize holes

largest first or smallest first

latter maximizes # of fields with a small offset from the beginning



C compilers promise not to rearrange

Unions (variant records)

overlay space

w/ tag: ***discriminated*** union

w/out tag: nondiscriminated union

cause problems for type checking -- you don't know what is there

ability to change tag and then access fields hardly any better

- can make fields "uninitialized" when tag is changed (this generally requires extensive run-time support)
- can require assignment of entire variant (w/ tag), as in Ada or OCaml

Several languages (including Algol68, Ada, and ML) require access to variant portions of a record to be confined to a "conformity clause" (e.g., OCaml's match) that ensures type safety.

If structs and unions are independent, declarations can be quite ugly, as in this legacy C:

```
struct employee {
   ...
   union {
      struct {  // hourly employee
         double hourly_pay;
         ...
      } S1;
      struct {  // salaried employee
         double annual_salary;
         ...
      } S2;
   } U1;
};
...
this_employee.U1.S1.hourly_pay     // yuk!
```

Pascal unified records and variants:

```
type employee = record
   ...
   case boolean of    (* hourly? *)
     true:
      hourly_pay : real;
      ...
     false:
```

```
        annual_salary: real;
        ...
    end;
    ...
    this_employee.hourly_pay            // better
```

Recent versions of C and C++ achieve a similar effect with ***anonymous*** structs and unions. Strike out the S1, S2, U1 names above.

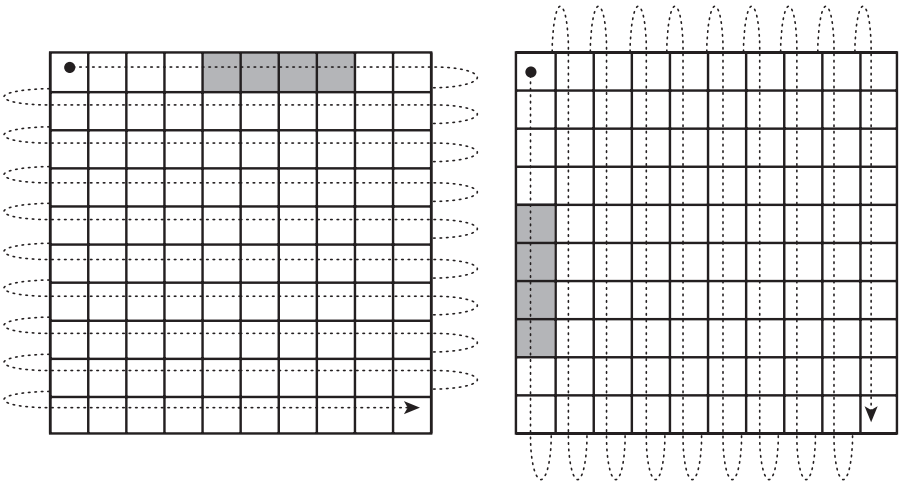
Note that the problem of uninitialized variables is more general than variant records. Some languages say variables start out with certain values (e.g. 0 for globals [but not locals!] in C). Many just say it's erroneous to use an uninitialized variable. A few actually try to prevent you from accessing one. In general, the only ways to do this are (1) restrict the language, e.g., as Java and C# do to ensure ***definite assignment***; (2) initialize variables automatically with a special "uninitialized" value and check most references at run time.

==========

Arrays

Two layout strategies for arrays:

- ***contiguous elements***
  - ***column major*** -- basically used only in Fortran
    probably an historical accident
  - ***row major*** -- used by everybody else; makes array [a..b, c..d]
    the same as array [a..b] of array [c..d].



Row-major order Column-major order

row pointers

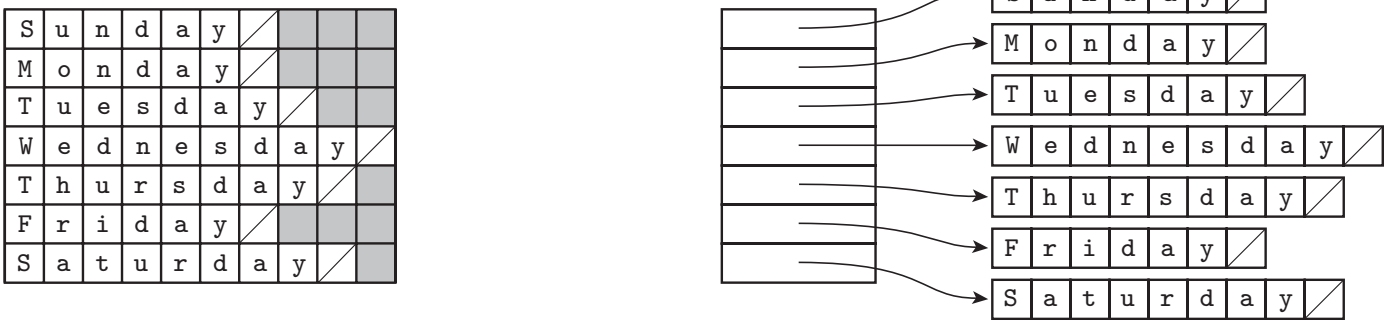
an option in C; only option in Java and some scripting languages

allows rows to be put anywhere -- nice for big arrays on
legacy machines with segmentation problems

avoids multiplication -- nice for legacy machines with slow multiply

nice for matrices whose rows are of different lengths
e.g. an array of strings

requires extra space for the pointers



Descriptors (***dope vectors***) required when bounds not known at compile time.

When bounds ***are*** known, much of the arithmetic can be done at compile time.

Given

```
A : array [L1..U1] of array [L2..U2]
      of array [L3..U3] of glarch;
```

Let

```
D1 = U1−L1+1
D2 = U2−L2+1
D3 = U3−L3+1
```

Let

```
S3 = sizeof glarch
S2 = D3 * S3
S1 = D2 * S2
```

The address of `A[i][j][k]` is

```
  (i − L1) * S1
+ (j − L2) * S2
+ (k − L3) * S3 + address of A
```

We could compute all that at run time, but we can make do with fewer subtractions:

```
== (i * S1) + (j * S2) + (k * S3)
 + address of A
 - [(L1 * S1) + (L2 * S2) + (L3 * S3)]
```

The stuff in square brackets is a compile-time constant that depends only on the type of A. We can combine easily with records:

Another example: Suppose A is a messy local variable.

The address of `A[i].B[3][j]` is

```
i * S1
          - L1 * S1
          + B's field offset
          + (3-L2) * S2
+ j * S3
          - L3 * S3
+ fp
          + A's offset in frame
```

Some languages assume that all array indexing starts at zero.
A few assume it starts at one.
This is ***not*** a performance issue: the lower bound can be factored out at compile time.

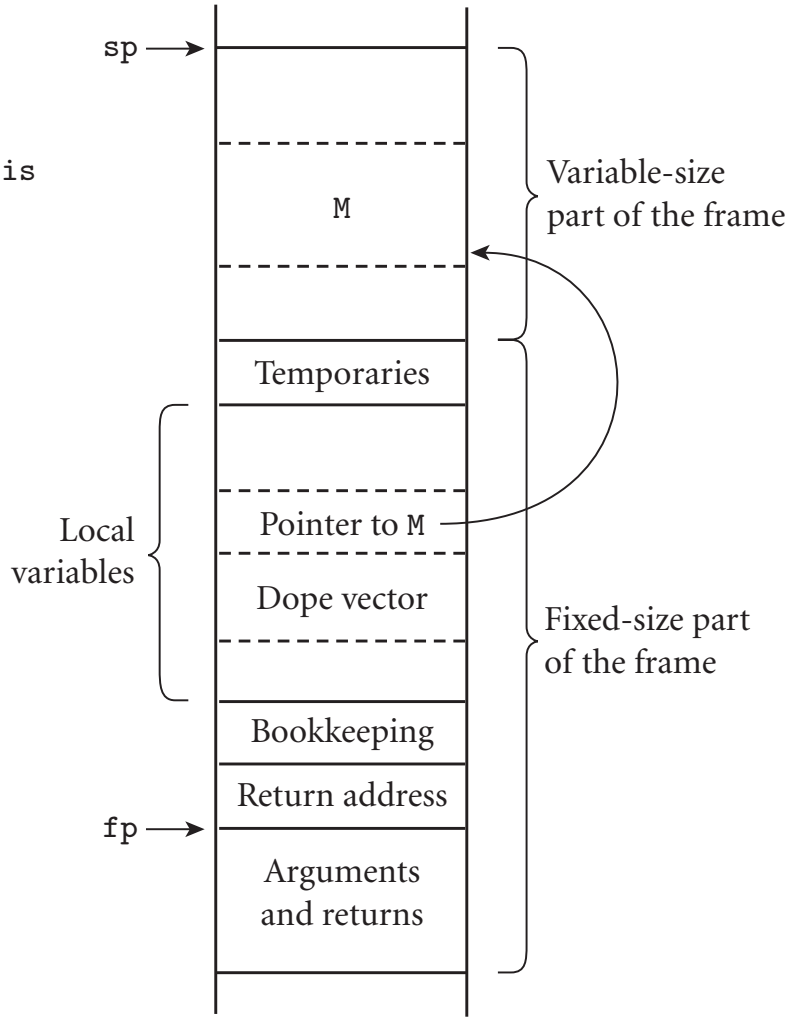
Lifetime (how long object exists)
and ***shape*** (bounds and possibly dimensions)
common options:

- global lifetime, static shape
  - globals in C
- local lifetime, static shape
  - subroutine locals in many classic imperative languages, including historical C
- local lifetime, shape bound at elaboration
  - subroutine locals in Ada or modern C
- arbitrary lifetime, shape bound at elaboration
  - Java arrays
- arbitrary lifetime, dynamic shape
  - most scripting languages, APL, Icon

The first two categories are just familiar global and local variables.
With dynamic shape you need dope vectors
The fourth and fifth categories have to be allocated off a heap.
The third category can still be put in a procedure's activation record;
Dope vector and a pointer go at a fixed offset from the FP;
the data itself is higher up in the frame
This divides the frame into fixed-size and variable-sized parts;
also requires a frame pointer.

```
-- Ada:
procedure foo(size : integer) is
M : array (1..size, 1..size)
        of long_float;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```



Note that deallocating a fully dynamic array on procedure exit requires
some extra code -- doesn't happen automatically via pop of stack frame.
Cf: C++ destructors

Slices (Fortran 90, APL, MATLAB, others)

```
matrix(3:6, 4:7)        columns 3-6, rows 4-7
matrix(6:, 5)           columns 6-end, row 5
matrix(:4, 2:8:2)       columns 1-4, every other row from 2-8
matrix(:, /2, 5, 9/)    all columns, rows 2, 5, and 9
```

can assign into each other as if they were smaller arrays.

Vectors

Supported by container libraries in many languages.
Built into a few -- esp. scripting languages.

Basically just arrays that automatically resize when you run off the end.

May also support operations like `push_back` (which extends the underlying array) or `delete` (which removes an element and moves all remaining elements down to fill the gap).

Strings

Basically arrays of characters.
But often special-cased, to give them flexibility (e.g., dynamic sizing) and operators not available for arrays in general.

It's easier to provide these things for strings than for arrays in general because strings are one-dimensional and non-circular (meaning you can garbage-collect them with reference counts; more later). Some languages make them all constant: you can create new strings, but not modify old ones.

Sets & ***mappings***

You learned about a lot of possible implementations in 172.
Bit vectors are what usually get built into compiled programming languages.
Things like intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions.

Some languages place draconian limits on the sizes of sets to make
    it easier for the implementor. There is really no excuse for this.
Scripting languages typically use hash tables. May use trees, or
    thread the hash table, for fast enumeration.

==========

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

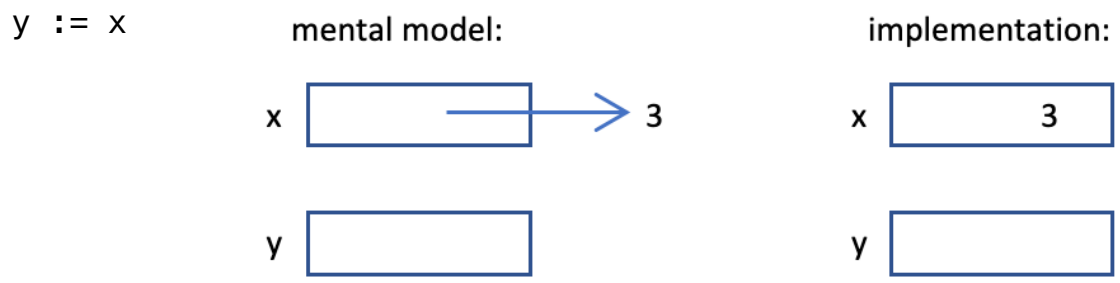
Note that pointers are ***not*** the same thing as addresses. Pointers are an abstraction. Addresses are an implementation. Pointers are ***not*** always implemented as addresses:

- 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.



Problems:

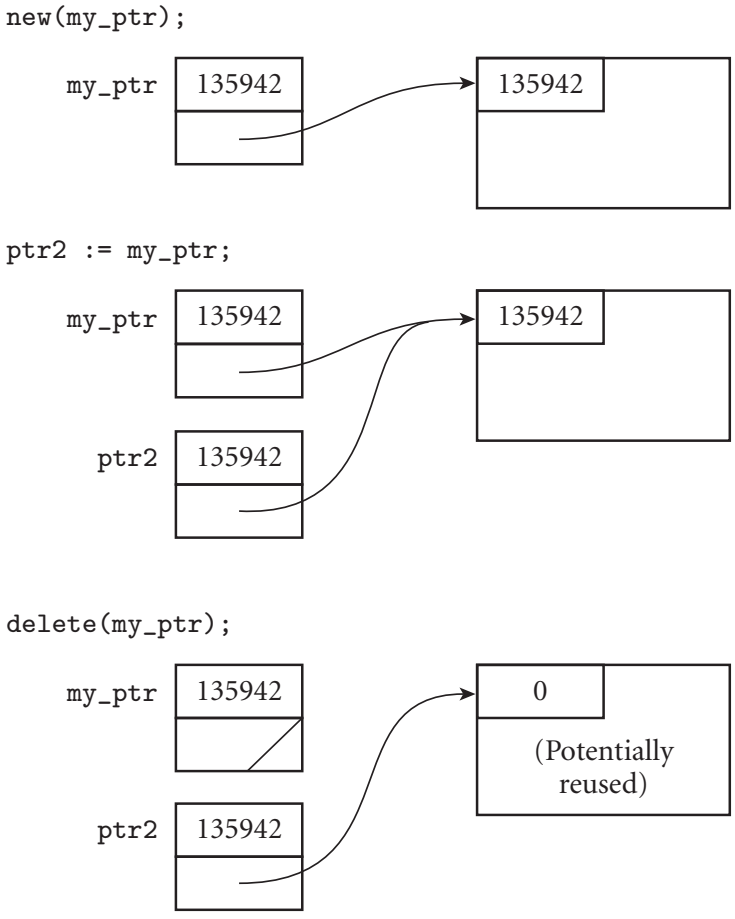
- syntax of pointer dereferencing
  - typically explicit, as in C
  - a few languages dereference automatically, depending on context
    - Ada, for example, does implicit dereferencing for record field references, and has special syntax to name the entire referenced object

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

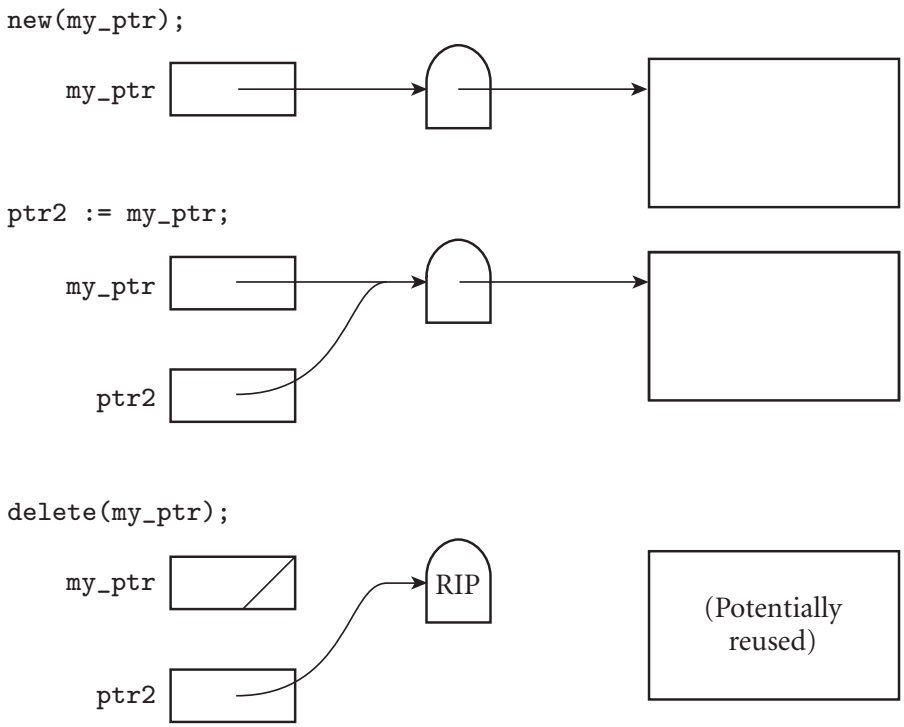
- dangling pointers due to
  - explicit deallocation of heap objects
    - only in languages that ***have*** explicit deallocation
  - implicit deallocation of elaborated objects
    - only in languages that let you create pointers to these
- two implementation mechanisms to catch:

locks and keys

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

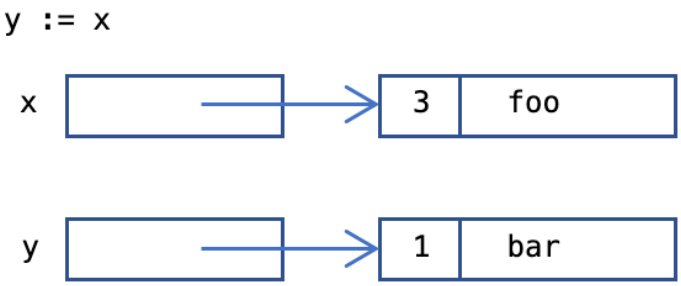
Many languages leave it up to the programmer to design without garbage creation. This is ***very*** hard.

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.

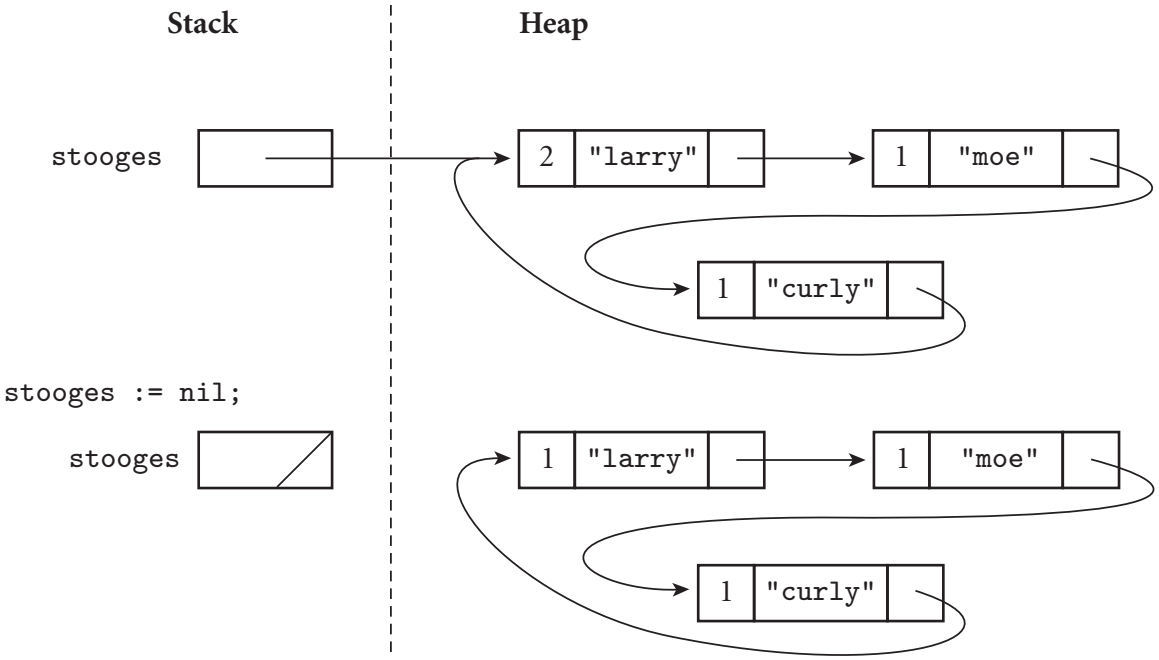
Increasingly, languages arrange for automatic garbage collection objects are reclaimed when the runtime can prove they are no longer accessible. (Note: this is ***not*** the same as no longer needed -- may be overly conservative.)

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.