

=====

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

A bit about chars. Multiple early character sets, 6 or 7 bit.

Convergence on 7-bit ASCII in the 1970s. Several (mutually incompatible) 8-bit extensions to accommodate diacritical marks in Western European languages.

Unicode introduced in the 1990s. Natural languages use 16-bit “Basic Multilingual Plane.” (Things like Klingon, Tengwar, and Cirth are in the other part.) Some languages still use 1B characters; others have gone to 2B (16b).

Ken Thompson invented UTF-8 as an encoding backward compatible with ASCII:

0 single-byte character

110 first byte of two-byte character (up to 11 sig. bits)

1110 . . . first byte of three-byte character (up to 16 sig. bits)

10 continuation byte

Records

usually laid out contiguously

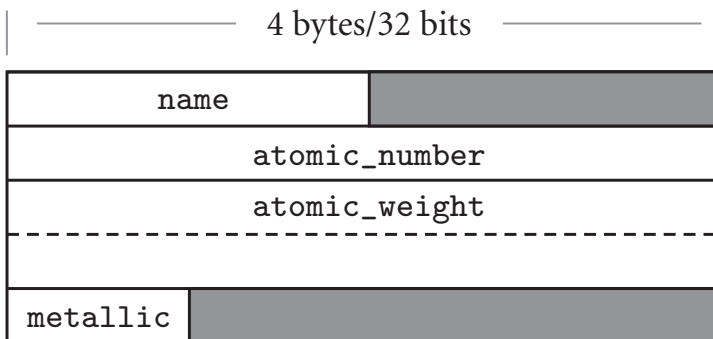
possible holes for alignment reasons

permits copying but **not** comparison with simple block operations

example:

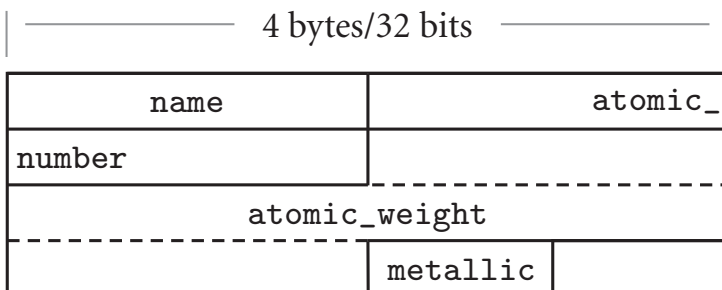
```
struct element {  
    char name[2];  
    int atomic_number;  
    double atomic_weight;  
    bool metallic;  
}
```

layout on a 32-bit machine:



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

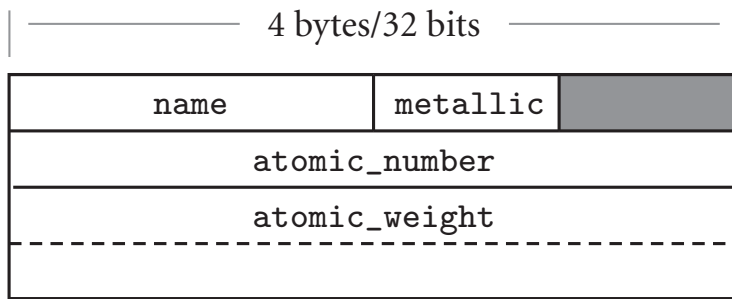
less space, but significant run-time access penalty



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 (as in OCaml or Rust)

w/out tag: nondiscriminated union (as in C)

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

ability to change tag and then access fields hardly any better (as in Pascal)

modern languages typically require whole-struct assignment and/or access within a “conformity clause” (e.g., Rust or OCaml match) that checks the tag.

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

=====

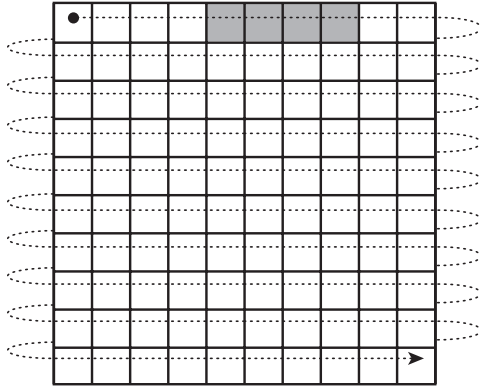
Arrays

Two layout strategies for arrays:

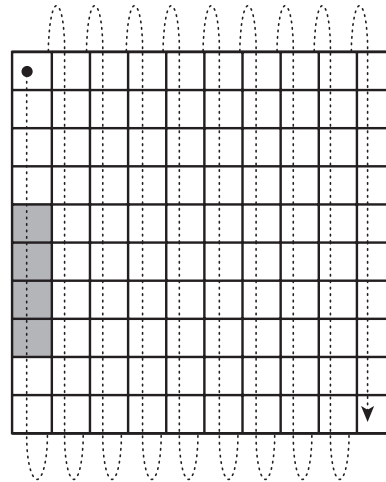
contiguous elements

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 and 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

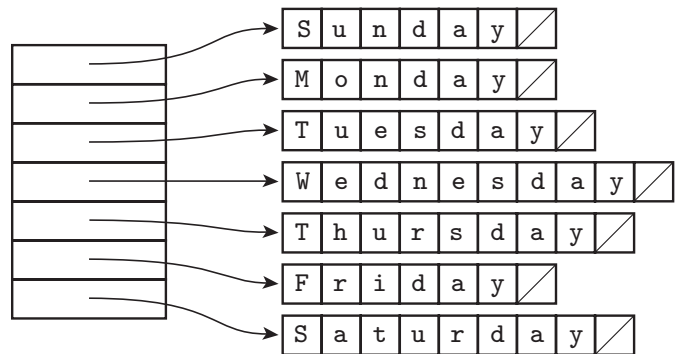
avoids multiplication -- nice for legacy machines with slow multiply

nice for matrices whose rows are of different lengths

e.g. an array of strings

requires extra space for the pointers

S	u	n	d	a	y	/			
M	o	n	d	a	y	/			
T	u	e	s	d	a	y	/		
W	e	d	n	e	s	d	a	y	/
T	h	u	r	s	d	a	y	/	
F	r	i	d	a	y	/			
S	a	t	u	r	d	a	y	/	



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: 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

NB: add to fp and load of value
can be combined in a single
instruction on most machines.

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)

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

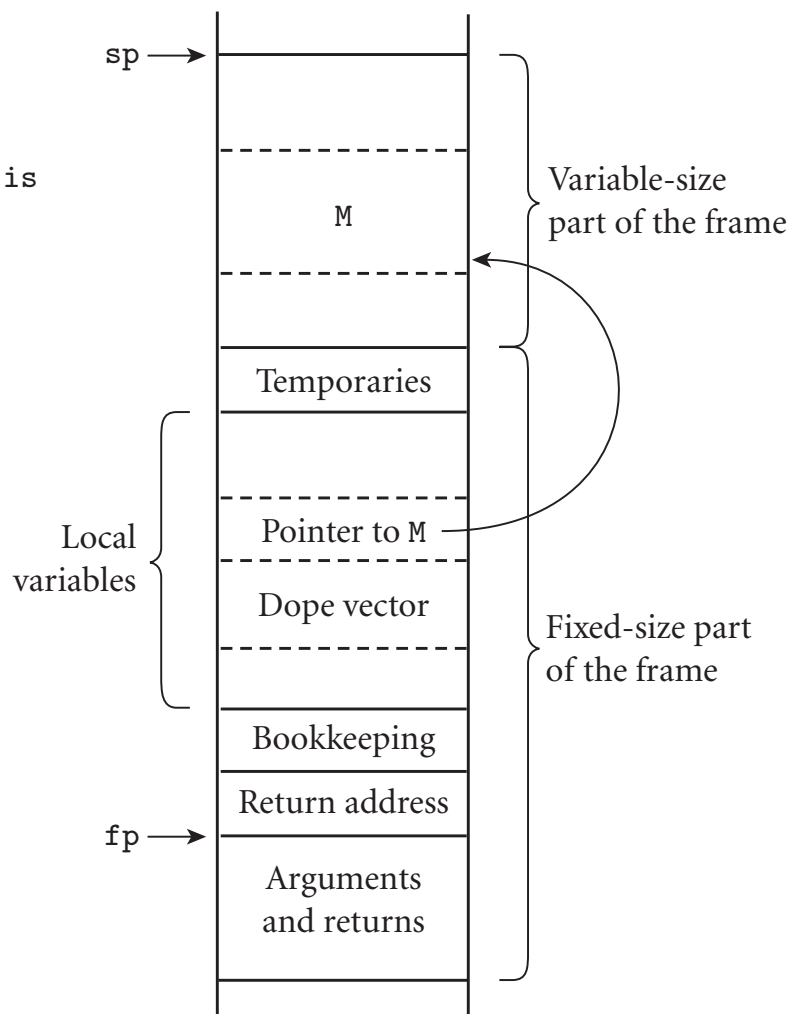
The first two categories are just familiar global and local variables. With dynamic shape you need dope vectors.

The fourth and fifth categories have to be allocated off a heap.

The third category can still be put in a subroutine's stack frame: 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 subroutine exit requires some extra code -- doesn't happen automatically via pop of stack frame.
Cf: C++ destructors, Rust drop

Slices (Fortran 90, APL, MATLAB, others)

<code>matrix(3:6, 4:7)</code>	columns 3-6, rows 4-7
<code>matrix(6:, 5)</code>	columns 6-end, row 5
<code>matrix(:, 2:8:2)</code>	columns 1-4, every other row from 2-8
<code>matrix(:, /2, 5, 9/)</code>	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 (for discrete types w/ small universe)

Scripting languages typically use hash tables for *unordered* sets and trees or skip lists for *ordered* sets (with fast enumeration and range queries).

Pointers and recursive types

For efficient (and sometimes intuitive) access to elaborated objects (as in C) or for dynamic creation of linked structures in the heap.

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)

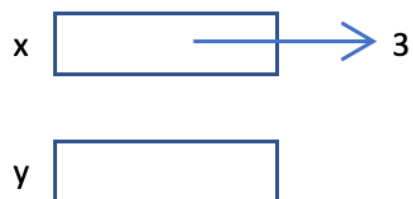
Many languages restrict pointers to accessing things in the heap: the only way to get a pointer is by calling `new`. Some (e.g., C, Rust) allow you to create a pointer to any existing object.

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

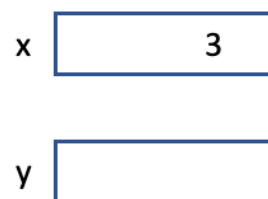
Good implementations of languages with a reference model of variables represent primitive (immutable) types the same way you would for a language with a value model of variables -- you think of your variable `x` as a reference to "the" 3 (the Platonic ideal), but the compiler implements it as a box with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

`y := x`

mental model:



implementation:



Problems:

syntax of dereferencing

typically implicit with a reference model of variables, or with reference parameters or variables (e.g., in Rust or C++)

typically explicit for pointers, as in C

a few languages dereference pointers 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
```

memory safety of storage management

garbage is memory you'll never use again

if you neglect to reclaim garbage, you have a *memory leak*

if you reclaim non-garbage, you may follow a *dangling reference*

garbage can be reclaimed manually or automatically

automatic techniques have to be conservative: reclaim things that *can't* be used again—not necessarily everything that *won't* be used again

2 key challenges for memory safety: dangling refs (w/ manual delete) and garbage collection (w/out manual delete)

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 common implementation mechanisms to catch:

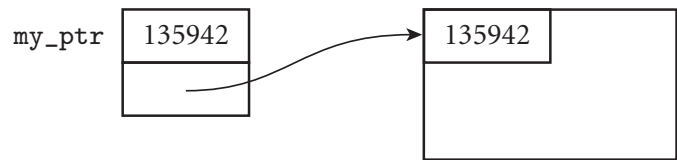
locks & keys simple; probabilistic protection

tombstones guaranteed, but more complex, slower

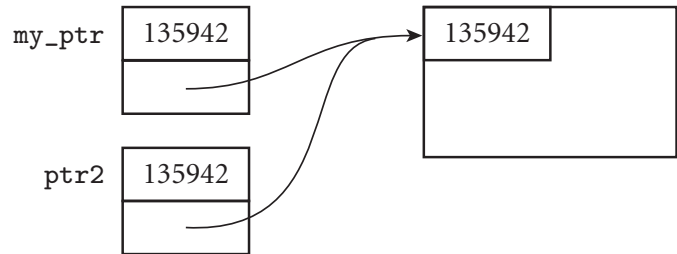
locks and keys

pointer is now at least 2 words—
3 if you can refer to an array
element or a field of a struct

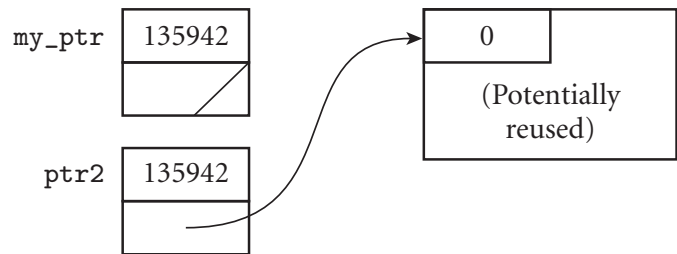
```
new(my_ptr);
```



```
ptr2 := my_ptr;
```



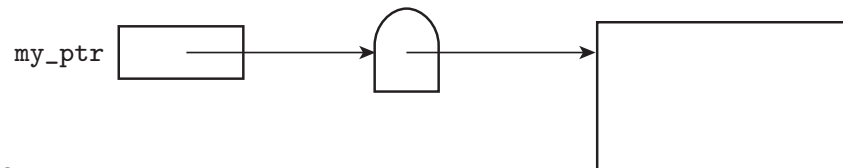
```
delete(my_ptr);
```



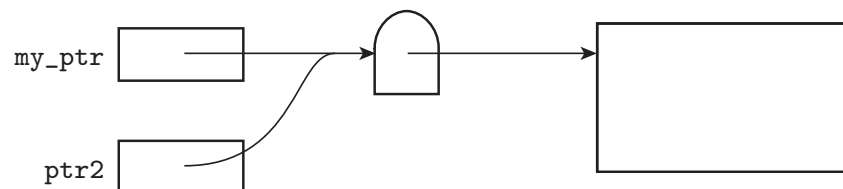
tombstones

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

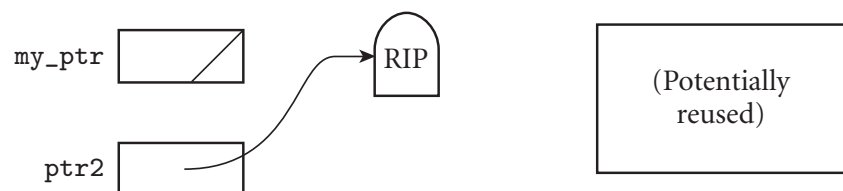
```
new(my_ptr);
```



```
ptr2 := my_ptr;
```



```
delete(my_ptr);
```



Garbage collection

Many languages historically left the programmer responsible for garbage collection. This is **very** hard.

C++ has moved over time toward automatic collection via smart pointers.

Rust does this systematically via **ownership** and **borrowing**, but that significantly complicates the creation of linked structures.

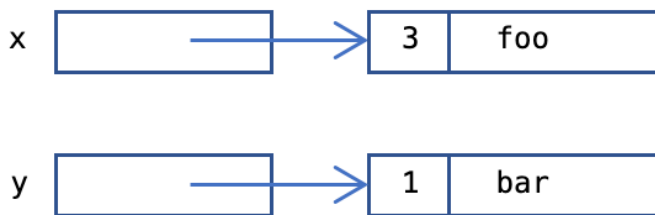
Most modern languages arrange for automatic garbage collection: objects are reclaimed when the runtime can prove they are no longer accessible. (As noted above, this is **not** the same as no longer needed -- may be overly conservative.)

Two common implementations: reference counting and tracing

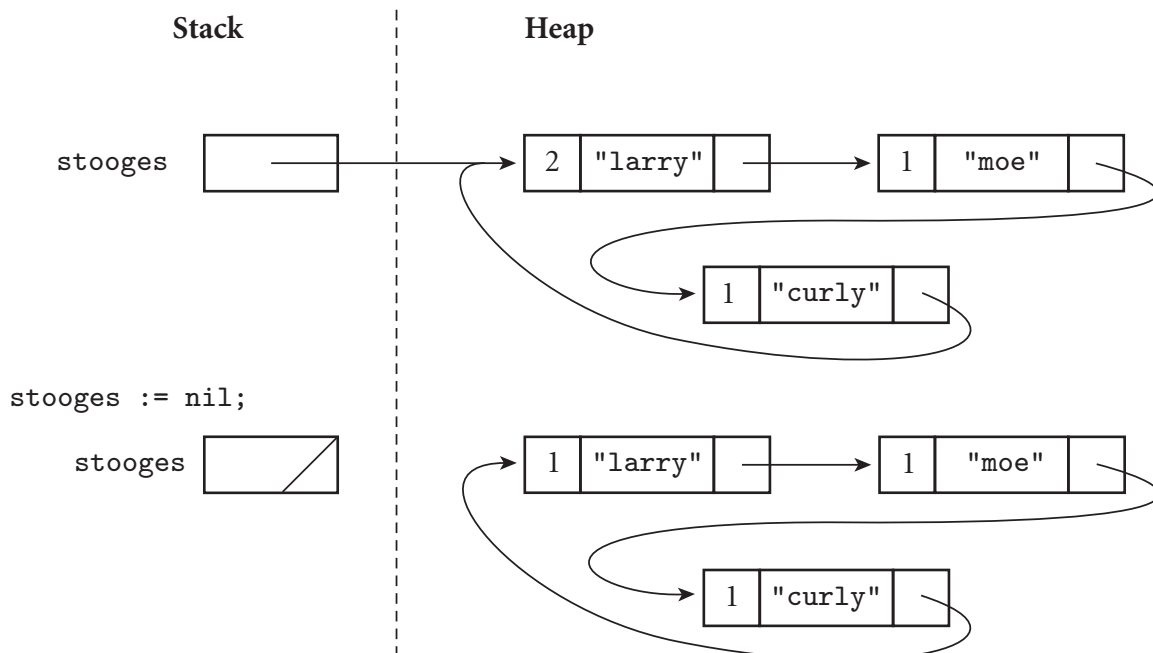
reference counting

`y := x`

works great for strings



does not work for circular structures



Does work for tombstones, assuming the compiler correctly decrements the reference counts of the tombstones for pointers that are deleted or go out of scope. 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.