==========

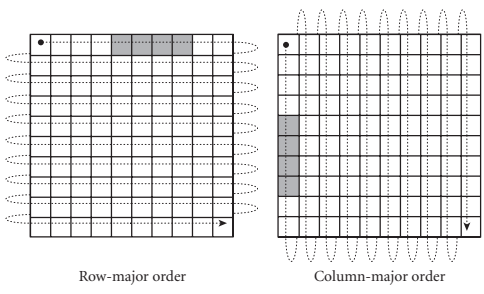
Arrays

Two layout strategies for arrays:

contiguous elements

column major -- basically used only in Fortran
probably an historical accident

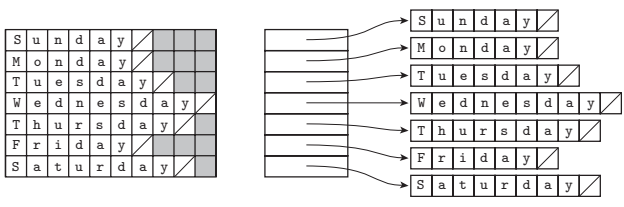
row major -- used by everybody else; makes array [a..b, c..d]
the same as array [a..b] of array [c..d].



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
```

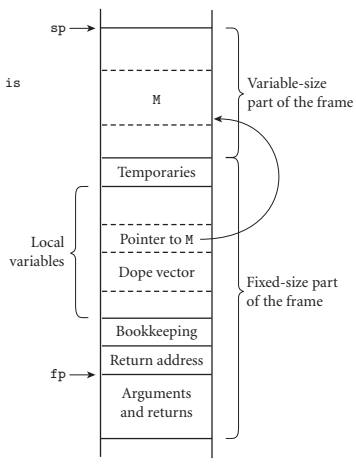
Some languages assume that all array indexing starts at zero.
A few assume it starts at one.
This is ***not*** a performance issue: the lower bound can be factored out at compile time.

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.