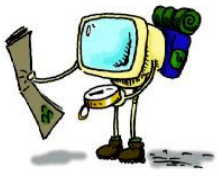
Programming Languages

Tevfik Koşar

Lecture - XVIII
March 23rd , 2006

Roadmap

- Arrays
- Pointers
- Lists
- Files and I/O



Arrays

- Two layout strategies for arrays
  - Contiguous elements
  - Row pointers
- Row pointers
  - an option in C
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication
  - nice for matrices whose rows are of different lengths
    - e.g. an array of strings
  - requires extra space for the pointers

3

Arrays

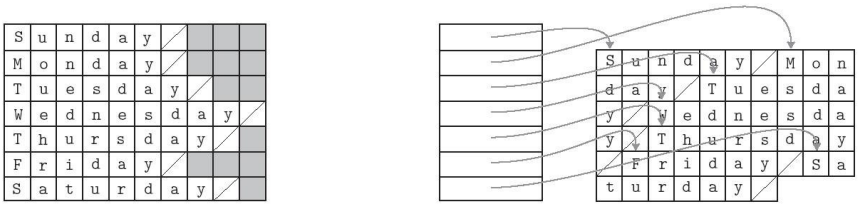


Figure 7.10: **Contiguous array allocation v. row pointers in C.** The declaration on the left is a true two-dimensional array. The slashed boxes are `NUL` bytes; the shaded areas are holes. The declaration on the right is an array of pointers to arrays of characters. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer (aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout (and corresponding address arithmetic) is quite different.

4

Arrays

- **Example:** Suppose

```
A : array [L1..U1] of array [L2..U2] of
  array [L3..U3] of elem;
  D1 = U1-L1+1
  D2 = U2-L2+1
  D3 = U3-L3+1
```

Let

```
  S3 = size of elem
  S2 = D3 * S3
  S1 = D2 * S2
```

5

Arrays

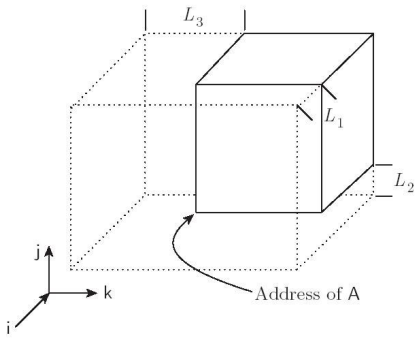


Figure 7.11: **Virtual location of an array with nonzero lower bounds.** By computing the constant portions of an array index at compile time, we effectively index into an array whose starting address is offset in memory, but whose lower bounds are all zero.

6

Arrays

- **Example** (continued)

  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 compile-time constant that depends only on the type of A

7

Strings

- Strings are really just arrays of characters
- They are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is 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 (more important) non-circular

8

Sets

- We learned about a lot of possible implementations
  - Bitsets are what usually get built into programming languages
  - Things like intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions
  - Some languages place limits on the sizes of sets to make it easier for the implementor
    - There is really no excuse for this

9

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
- Several languages (e.g. Pascal) restrict pointers to accessing things in the heap
- Pointers are used with a value model of variables
  - They aren't needed with a reference model

10

Pointers And Recursive Types

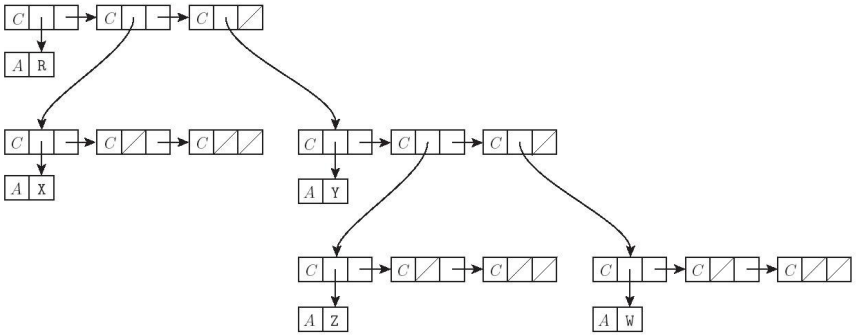


Figure 7.13: **Implementation of a tree in Lisp.** A diagonal slash through a box indicates a `nil` pointer. The *C* and *A* tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

11

Pointers And Recursive Types

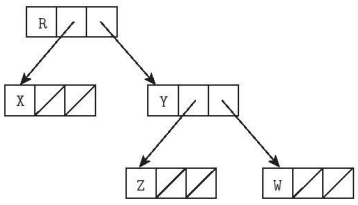


Figure 7.14: **Typical implementation of a tree in a language with explicit pointers.** As in Figure 7.13, a diagonal slash through a box indicates a `nil` pointer.

12

Pointers And Recursive Types

- C pointers and arrays

```
int *a == int a[]
int **a == int *a[]
```

- BUT equivalences don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension
  - otherwise it allocates a pointer

```
int **a, int *a[]    pointer to pointer to int
int *a[n], n-element array of row pointers
int a[n][m], 2-d array
```

13

Pointers And Recursive Types

- Compiler has to be able to tell the size of the things to which you point
  - So the following aren't valid:

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

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

14

Pointers And Recursive Types

- Problems with dangling pointers are due to
  - explicit deallocation of heap objects
    - only in languages that *have* explicit deallocation
  - implicit deallocation of elaborated objects
- Two implementation mechanisms to catch dangling pointers
  - Tombstones
  - Locks and Keys

15

Pointers And Recursive Types

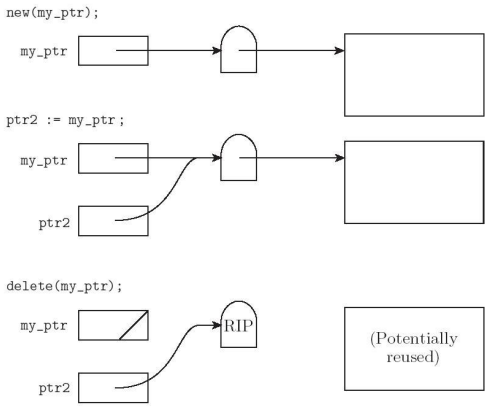


Figure 7.15: **Tombstones.** A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

16

Pointers And Recursive Types

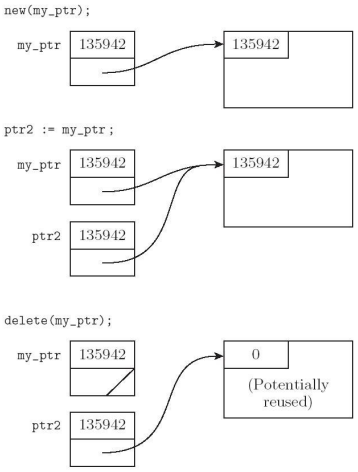


Figure 7.16: **Locks and Keys.** A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

17

Pointers And Recursive Types

- Problems with garbage collection
  - many languages leave it up to the programmer to design without garbage creation - this is VERY hard
  - others arrange for automatic garbage collection
  - reference counting
    - does not work for circular structures
    - works great for strings
    - should also work to collect unneeded tombstones

18

Pointers And Recursive Types

- Garbage collection with reference counts

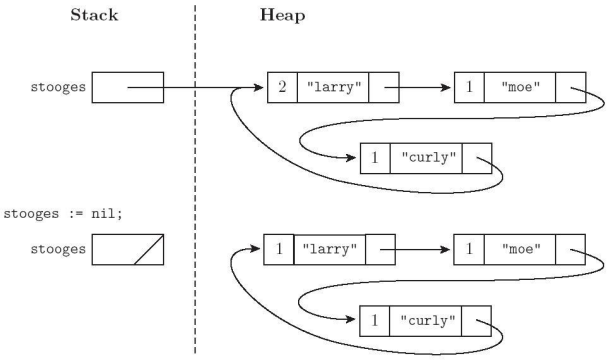


Figure 7.17: **Reference counts and circular lists.** The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

19

Pointers And Recursive Types

- Mark-and-sweep
  - commonplace in Lisp dialects
  - complicated in languages with rich type structure, but possible if language is strongly typed
  - achieved successfully in Cedar, Ada, Java, Modula-3, ML
  - complete solution impossible in languages that are not strongly typed
  - conservative approximation possible in almost any language (Xerox Portable Common Runtime approach)

20

Pointers And Recursive Types

- Garbage collection with pointer reversal

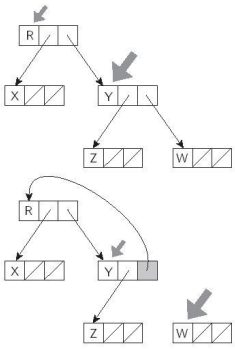


Figure 7.18: **Heap exploration via pointer reversal.** The block currently under examination is indicated by the large grey arrow. The previous block is indicated by the small grey arrow. As the garbage collector moves from one block to the next, it changes the pointer it follows to refer back to the previous block. When it returns to a block it restores the pointer. Each reversed pointer must be marked, to distinguish it from other, forward pointers in the same block. We assume in this figure that the root node `R` is outside the heap, so none of its pointers are reversed.

21

Lists

- A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list
  - Lists are ideally suited to programming in functional and logic languages
    - In Lisp, in fact, a program *is* a list, and can extend itself at run time by constructing a list and executing it
  - Lists can also be used in imperative programs

22

Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - *interactive* I/O and I/O with files
- Interactive I/O generally implies communication with human users or physical devices
- Files generally refer to off-line storage implemented by the operating system
- Files may be further categorized into
  - *temporary*
  - *persistent*

23