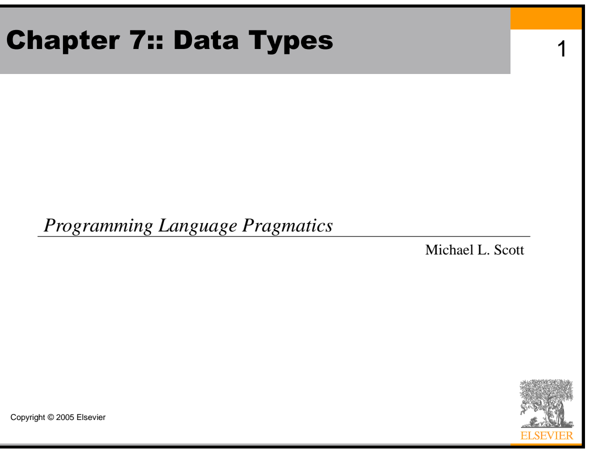


Data Types

2

- most programming languages include a notion of types for expressions, variables, and values
- two main purposes of types
  - imply a context for operations
    - programmer does not have to state explicitly
    - example: a + b means mathematical addition if a and b are integers
  - limit the set of operations that can be performed
    - prevents the programmer from making mistakes
    - type checking cannot prevent all meaningless operations
    - it catches enough of them to be useful

Copyright © 2005 Elsevier

Type Systems

3

- bits in memory can be interpreted in different ways
  - instructions
  - addresses
  - characters
  - integers, floats, etc.
- bits themselves are untyped, but high-level languages associate types with values
  - useful for operator context
  - allows error checking

Copyright © 2005 Elsevier

Type Systems

4

- a type system consists of
  - a way to define types and associate them with language constructs
    - constructs that must have values include constants, variables, record fields, parameters, literal constants, subroutines, and complex expressions containing these
  - rules for type equivalence, type compatibility, and type inference
    - type equivalence: when the types of two values are the same
    - type compatibility: when a value of a given type can be used in a particular context
    - type inference: type of an expression based on the types of its parts

Copyright © 2005 Elsevier

Type Checking

5

- type checking ensures a program obeys the language's type compatibility rules
  - type clash: violation of type rules
- a language is strongly typed if it prohibits an operation from performing on an object it does not support
- a language is statically typed if it is strongly typed and type checking can be performed at compile time
  - few languages fall completely in this category
  - questions about its value

Copyright © 2005 Elsevier

Type Checking

6

- examples
  - Common Lisp is strongly typed, but not statically typed
  - Ada is statically typed
  - Pascal is almost statically typed
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically

Copyright © 2005 Elsevier

Type Checking

7

- a language is dynamically typed if type checking is performed at run time
  - late binding
  - List and Smalltalk
  - scripting languages, such as Python and Ruby

Copyright © 2005 Elsevier

Polymorphism

8

- polymorphism results when the compiler finds that it doesn't need to know certain things
  - allows a single body of code to work with objects of multiple types
  - with dynamic typing, arbitrary operations can be applied to arbitrary objects
    - implicit parametric polymorphism: types can be thought of to be implied unspecified parameters
    - incurs significant run-time costs
    - for example, ML's type inference/unification scheme

Copyright © 2005 Elsevier

Polymorphism

9

- subtype polymorphism in object-oriented languages
  - a variable of the base type can refer to an object of the derived type
  - explicit parametric polymorphism, or generics
    - C++, Eiffel, Java
    - useful for container classes

      List<T> where T is left unspecified

Copyright © 2005 Elsevier

Data Types

10

- we all have developed an intuitive notion of what types are; what's behind the intuition?
  - collection of values from a domain (the denotational approach)
  - internal structure of data, described down to the level of a small set of fundamental types (the structural approach)
  - collection of well-defined operations that can be applied to objects of that type (the abstraction approach)

Copyright © 2005 Elsevier

Data Types

11

- specified in different ways
  - Fortran: spelling of variable names
  - Lisp: infer types at compile time
  - List, Smalltalk: track at run time
  - C, C++: declared at compile time

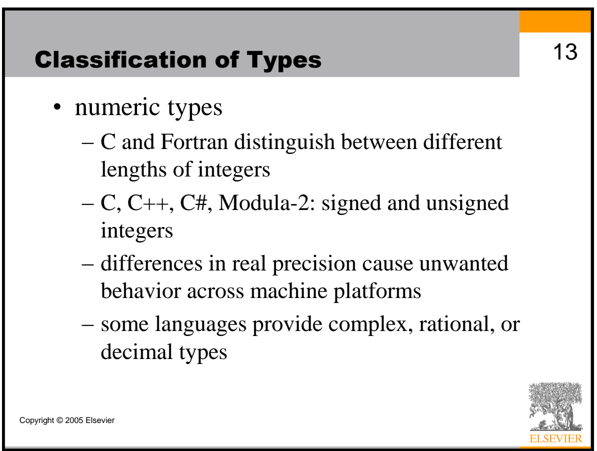
Copyright © 2005 Elsevier

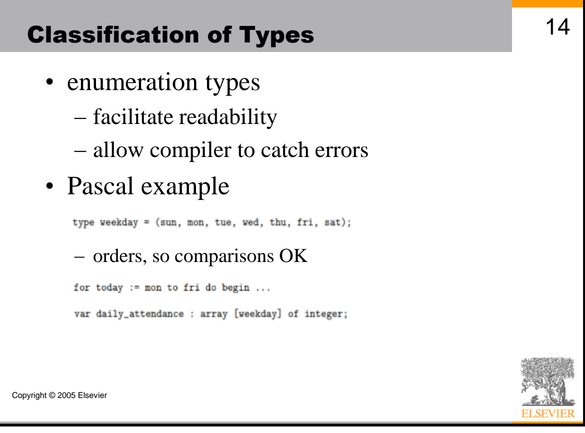
Classification of Types

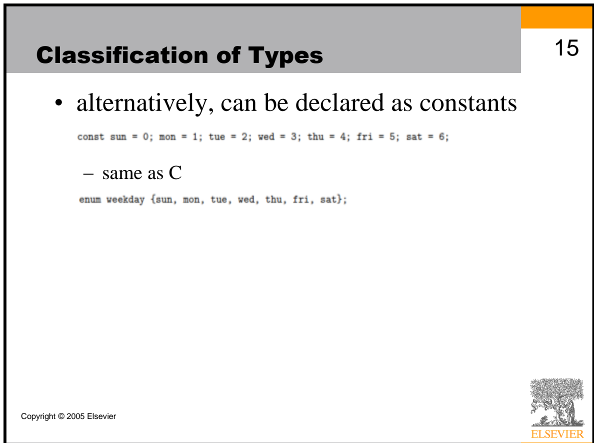
12

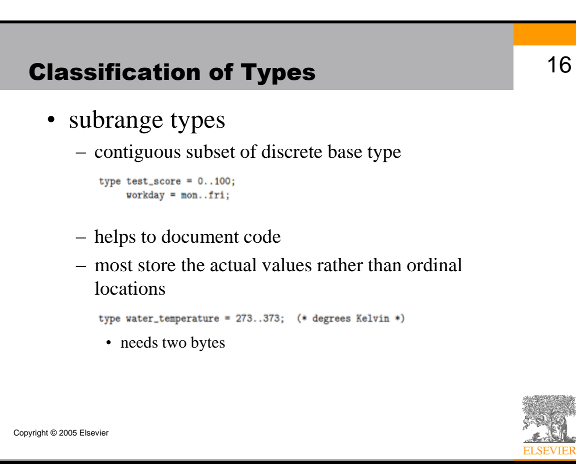
- most languages provide a set of built-in types
  - integer
  - boolean
    - often single character with 1 as true and 0 as false
    - C: no explicit boolean; 0 is false, anything else is true
  - char
    - traditionally one byte
    - ASCII, Unicode
  - real

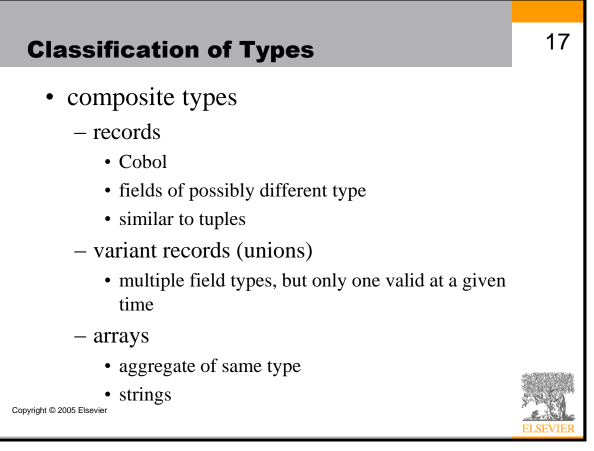
Copyright © 2005 Elsevier

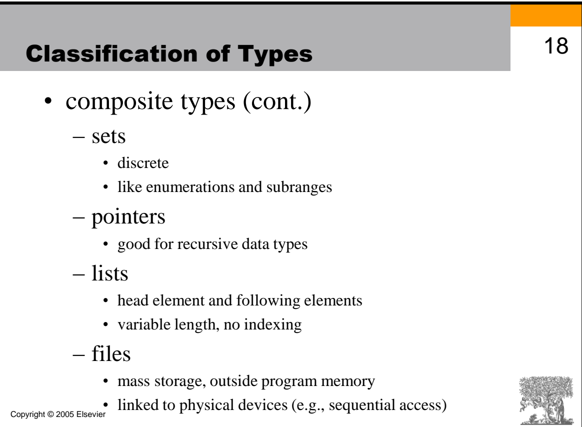

Classification of Types
13
• numeric types
– C and Fortran distinguish between different lengths of integers
– C, C++, C#, Modula-2: signed and unsigned integers
– differences in real precision cause unwanted behavior across machine platforms
– some languages provide complex, rational, or decimal types
Copyright © 2005 Elsevier


Classification of Types
14
• enumeration types
– facilitate readability
– allow compiler to catch errors
• Pascal example
type weekday = (sun, mon, tue, wed, thu, fri, sat);
– orders, so comparisons OK
for today := mon to fri do begin ...
var daily_attendance : array [weekday] of integer;
Copyright © 2005 Elsevier


Classification of Types
15
• alternatively, can be declared as constants
const sun = 0; mon = 1; tue = 2; wed = 3; thu = 4; fri = 5; sat = 6;
– same as C
enum weekday {sun, mon, tue, wed, thu, fri, sat};
Copyright © 2005 Elsevier


Classification of Types
16
• subrange types
– contiguous subset of discrete base type
type test_score = 0..100;
workday = mon..fri;
– helps to document code
– most store the actual values rather than ordinal locations
type water_temperature = 273..373; (* degrees Kelvin *)
• needs two bytes
Copyright © 2005 Elsevier


Classification of Types
17
• composite types
– records
• Cobol
• fields of possibly different type
• similar to tuples
– variant records (unions)
• multiple field types, but only one valid at a given time
– arrays
• aggregate of same type
• strings
Copyright © 2005 Elsevier


Classification of Types
18
• composite types (cont.)
– sets
• discrete
• like enumerations and subranges
– pointers
• good for recursive data types
– lists
• head element and following elements
• variable length, no indexing
– files
• mass storage, outside program memory
• linked to physical devices (e.g., sequential access)
Copyright © 2005 Elsevier

Orthogonality

- orthogonality is a useful goal in the design of a language, particularly its type system
  - a collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined (analogy to vectors)

Copyright © 2005 Elsevier

Orthogonality

- for example
  - Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance)
- orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

Copyright © 2005 Elsevier

Type Checking

- a type system has rules for
  - type equivalence (when are the types of two values the same?)
  - type compatibility (when can a value of type A be used in a context that expects type B?)
  - type inference (what is the type of an expression, given the types of the operands?)

Copyright © 2005 Elsevier

Type Checking

- type compatibility / type equivalence
  - compatibility is the more useful concept, because it tells you what you can do
  - terms are often used interchangeably (incorrectly)

Copyright © 2005 Elsevier

Type Checking

- format does not matter

```
struct { int a, b; }
```

is the same as

```
struct {
    int a, b;
}
```

- want them to be the same as

```
struct {
    int a;
    int b;
}
```

Copyright © 2005 Elsevier

Type Checking

- two major approaches: structural equivalence and name equivalence
  - name equivalence is based on declarations
  - structural equivalence is based on some notion of meaning behind those declarations
  - name equivalence is more fashionable these days

Copyright © 2005 Elsevier

Type Checking

- at least two common variants on name equivalence
  - differences between all these approaches boils down to where the line is drawn between important and unimportant differences between type descriptions
  - in all three schemes, we begin by putting every type description in a standard form that takes care of "obviously unimportant" distinctions like those above

Copyright © 2005 Elsevier

Type Checking

- structural equivalence depends on simple comparison of type descriptions
  - substitute out all names
  - expand all the way to built-in types
- original types are equivalent if the expanded type descriptions are the same

Copyright © 2005 Elsevier

Type Checking

- example in Ada illustrating type conversion

```
n : integer;          -- assume 32 bits
r : real;             -- assume IEEE double-precision
t : test_score;       -- as in Example 7.9  (1..100)
c : celsius_temp;     -- as in Example 7.19 (integer)
...
t := test_score(n);    -- run-time semantic check required
n := integer(t);       -- no check req.; every test_score is an int
r := real(n);          -- requires run-time conversion
n := integer(r);       -- requires run-time conversion and check
n := integer(c);       -- no run-time code required
c := celsius_temp(n);  -- no run-time code required
```

Copyright © 2005 Elsevier

Type Checking

- coercion
  - when an expression of one type is used in a context where a different type is expected, one normally gets a type error
  - but what about

```
var a : integer; b, c : real;
      ...
c := a + b;
```

Copyright © 2005 Elsevier

Type Checking

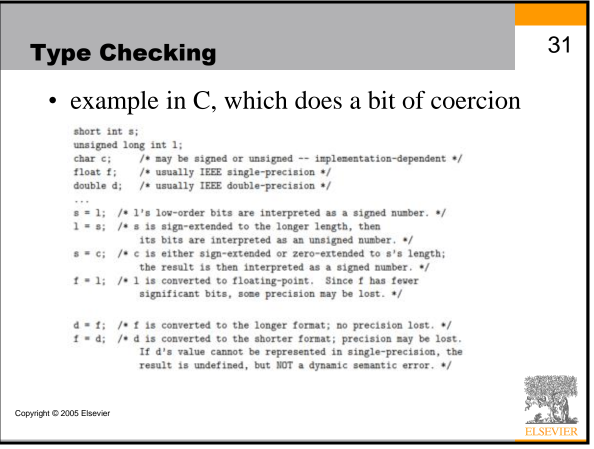
- coercion
  - many languages allow such statements, and coerce an expression to be of the proper type
  - coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
  - Fortran has lots of coercion, all based on operand type

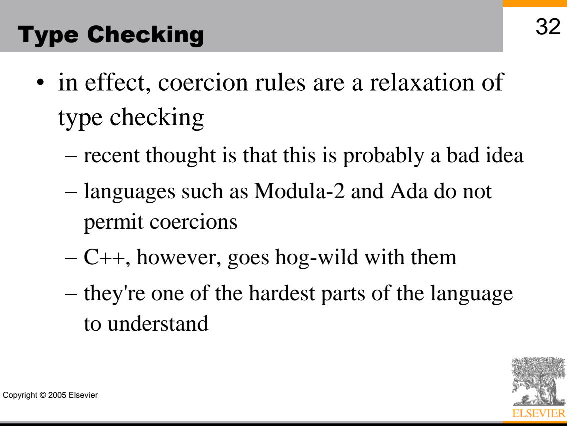
Copyright © 2005 Elsevier

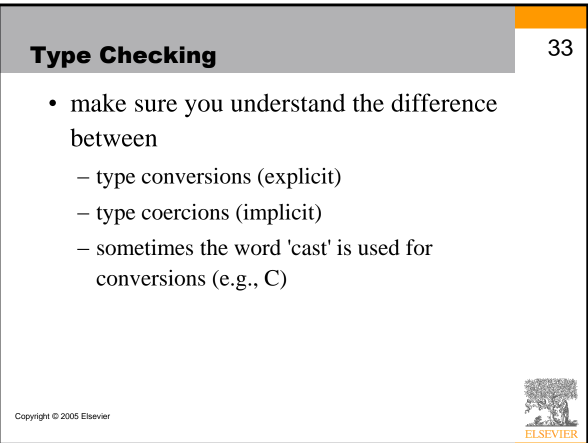
Type Checking

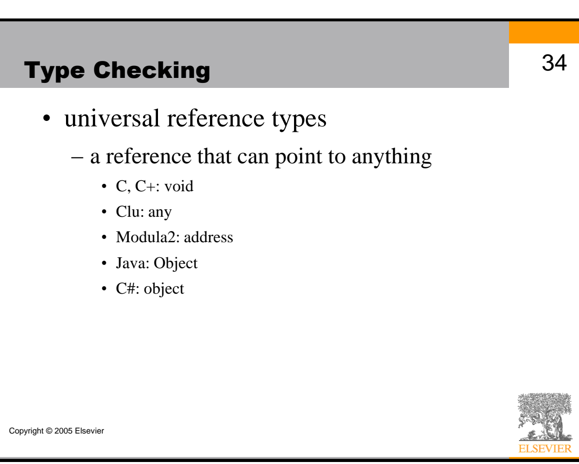
- C has lots of coercion, too, but with simpler rules:
  - all `float`s in expressions become `double`s
  - short `int` and `char` become `int` in expressions
  - if necessary, precision is removed when assigning into LHS

Copyright © 2005 Elsevier









Records (Structures) and Variants (Unions)

35

- records
  - allow related heterogeneous types to be stored together
  - C, C++, and Common Lisp: structure
    - C++: class with globally visible members
  - Fortran: types

Copyright © 2005 Elsevier

ELSEVIER

Records (Structures) and Variants (Unions)

36

- C

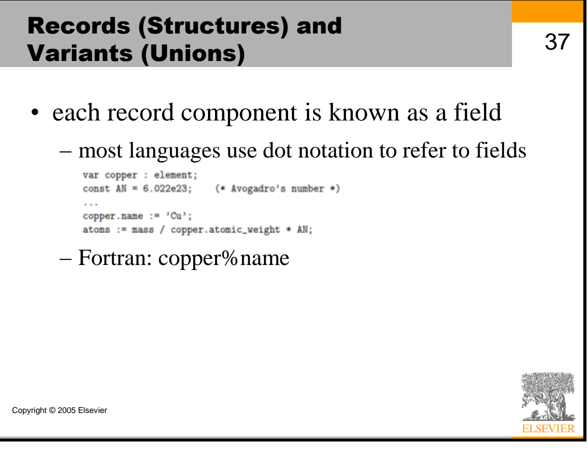
```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```

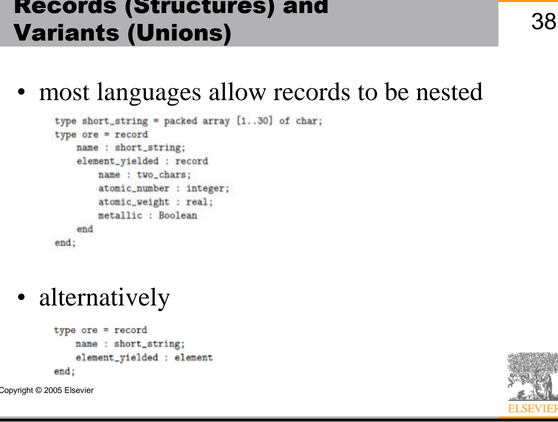
- Pascal

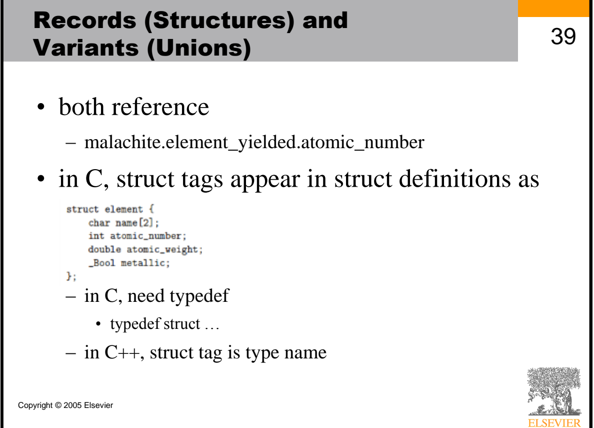
```
type two_chars = packed array [1..2] of char;
    (* Packed arrays will be explained in Example 7.39.
       Packed arrays of char are compatible with quoted strings. *)
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;
```

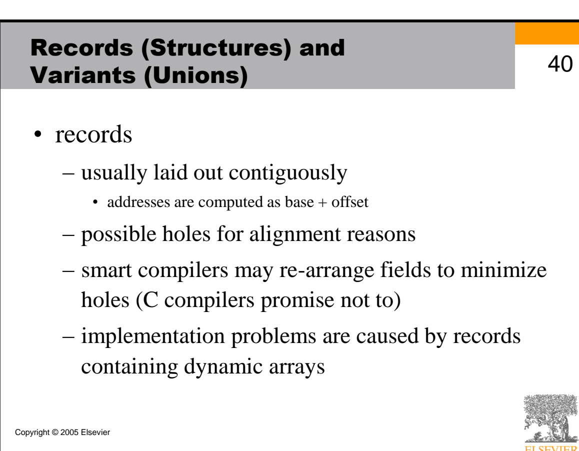
Copyright © 2005 Elsevier

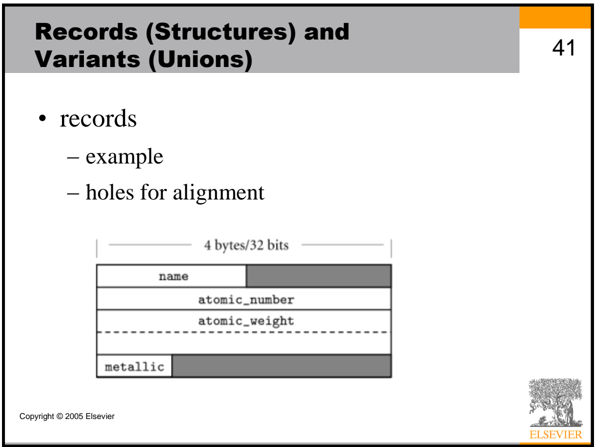
ELSEVIER

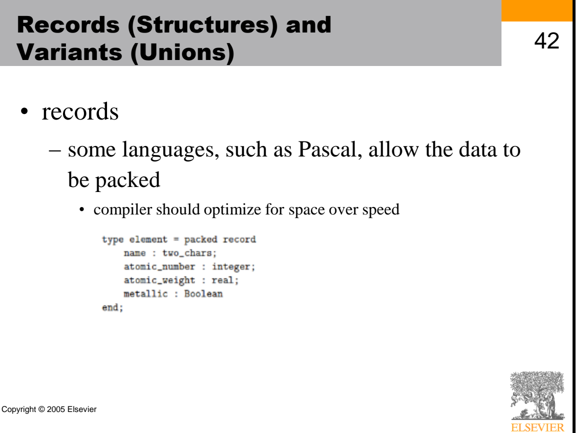

Records (Structures) and Variants (Unions)
37
• each record component is known as a field
– most languages use dot notation to refer to fields
var copper : element;
const AN = 6.022e23;   (* Avogadro's number *)
...
copper.name := 'Cu';
atoms := mass / copper.atomic_weight * AN;
– Fortran: copper%name
Copyright © 2005 Elsevier
ELSEVIER

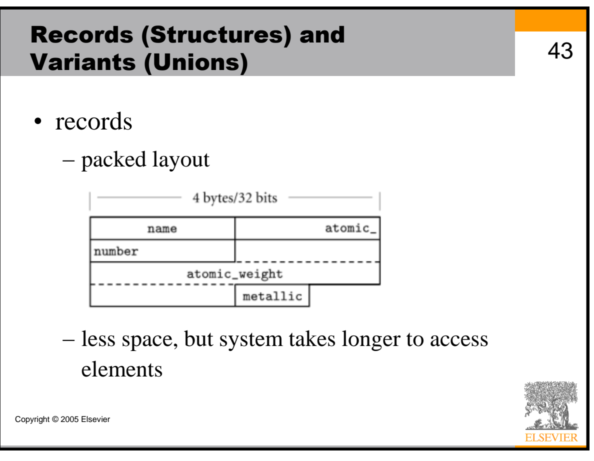

Records (Structures) and Variants (Unions)
38
• most languages allow records to be nested
type short_string = packed array [1..30] of char;
type ore = record
name : short_string;
element_yielded : record
name : two_chars;
atomic_number : integer;
atomic_weight : real;
metallic : Boolean
end
end;
• alternatively
type ore = record
name : short_string;
element_yielded : element
end;
Copyright © 2005 Elsevier
ELSEVIER

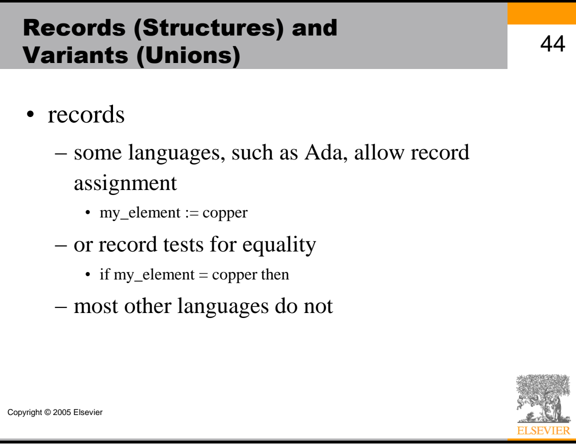

Records (Structures) and Variants (Unions)
39
• both reference
– malachite.element_yielded.atomic_number
• in C, struct tags appear in struct definitions as
struct element {
char name[2];
int atomic_number;
double atomic_weight;
_Bool metallic;
};
– in C, need typedef
• typedef struct …
– in C++, struct tag is type name
Copyright © 2005 Elsevier
ELSEVIER

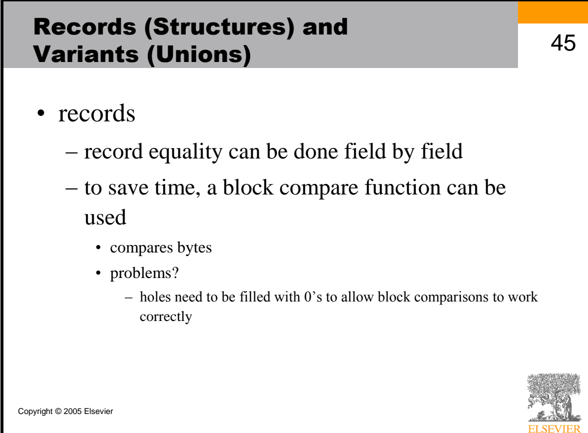

Records (Structures) and Variants (Unions)
40
• records
– usually laid out contiguously
• addresses are computed as base + offset
– possible holes for alignment reasons
– smart compilers may re-arrange fields to minimize holes (C compilers promise not to)
– implementation problems are caused by records containing dynamic arrays
Copyright © 2005 Elsevier
ELSEVIER

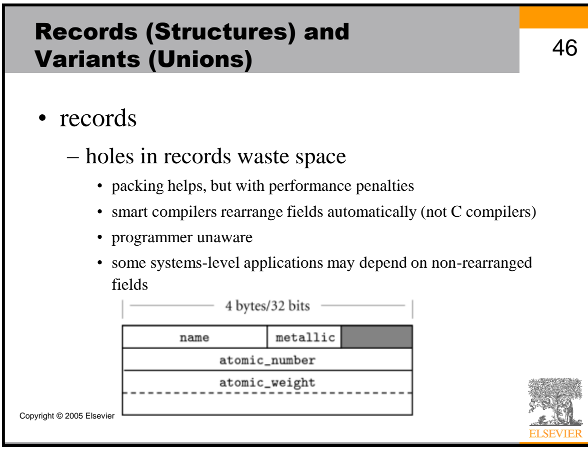

Records (Structures) and Variants (Unions)
41
• records
– example
– holes for alignment
4 bytes/32 bits
name
atomic_number
atomic_weight
metallic
Copyright © 2005 Elsevier
ELSEVIER


Records (Structures) and Variants (Unions)
42
• records
– some languages, such as Pascal, allow the data to be packed
• compiler should optimize for space over speed
type element = packed record
name : two_chars;
atomic_number : integer;
atomic_weight : real;
metallic : Boolean
end;
Copyright © 2005 Elsevier
ELSEVIER









Records (Structures) and Variants (Unions)

47

- unions (variant records)
  - space was extremely limited in early days of programming
  - overlay space
  - cause problems for type checking
- can be used for automatically re-interpreting bits without casts

Copyright © 2005 Elsevier

ELSEVIER

Records (Structures) and Variants (Unions)

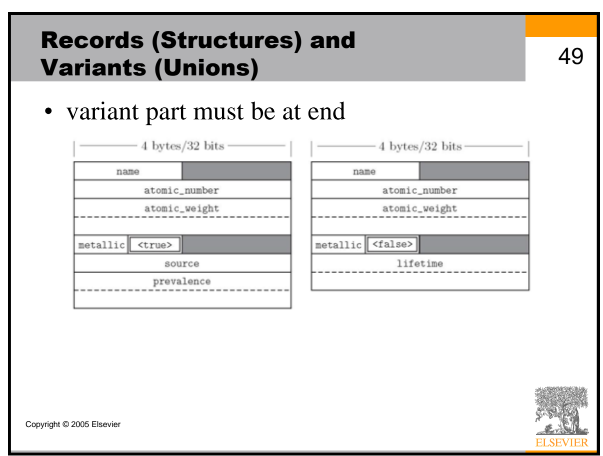
48

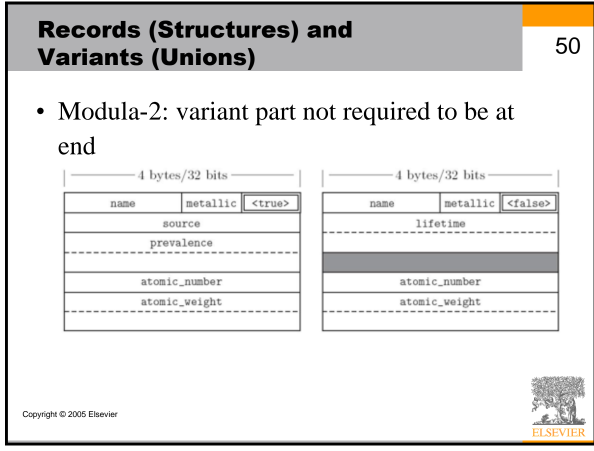
- Pascal

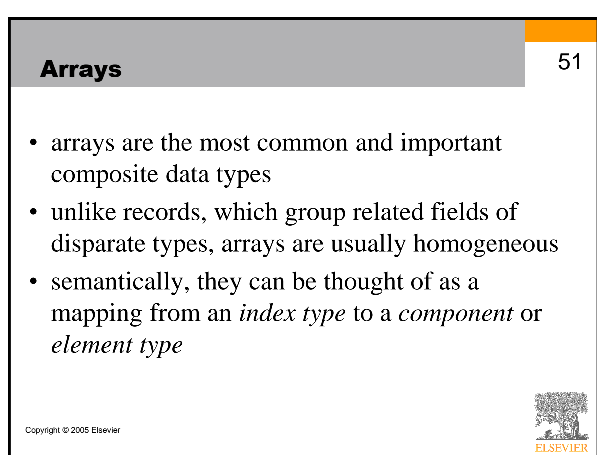
```
type long_string = packed array [1..200] of char;
type string_ptr = ^long_string;
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
    case naturally_occurring : Boolean of
      true : (
        source : string_ptr;
            (* textual description of principal commercial source *)
        prevalence : real;
            (* fraction, by weight, of Earth's crust *)
      );
      false : (
        lifetime : real;
            (* half-life in seconds of the most stable known isotope *)
      )
end;
```

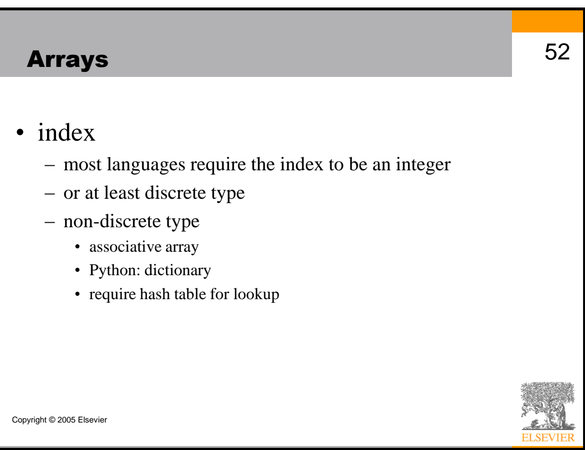
Copyright © 2005 Elsevier

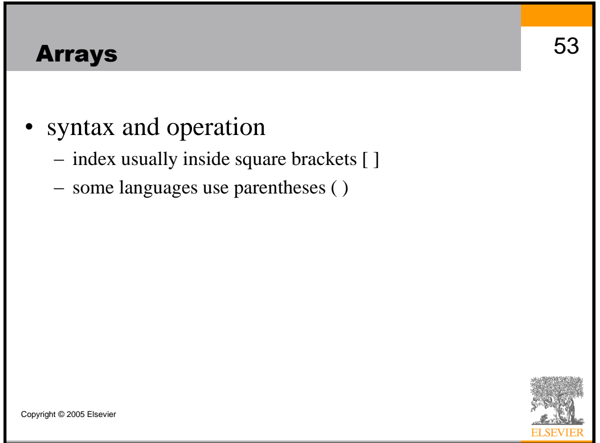
ELSEVIER

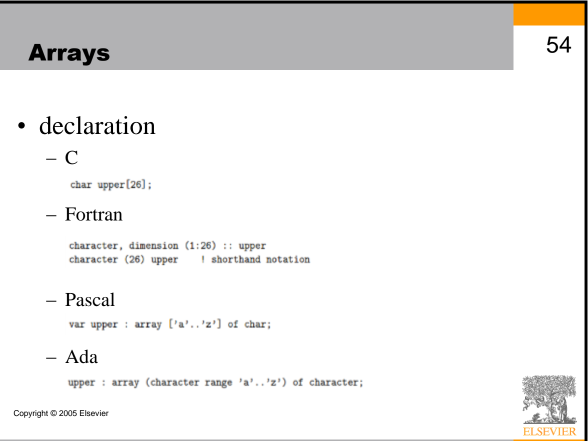

Records (Structures) and
Variants (Unions)
49
variant part must be at end
4 bytes/32 bits
name
atomic_number
atomic_weight
metallic
<true>
source
prevalence
4 bytes/32 bits
name
atomic_number
atomic_weight
metallic
<false>
lifetime
Copyright © 2005 Elsevier
ELSEVIER

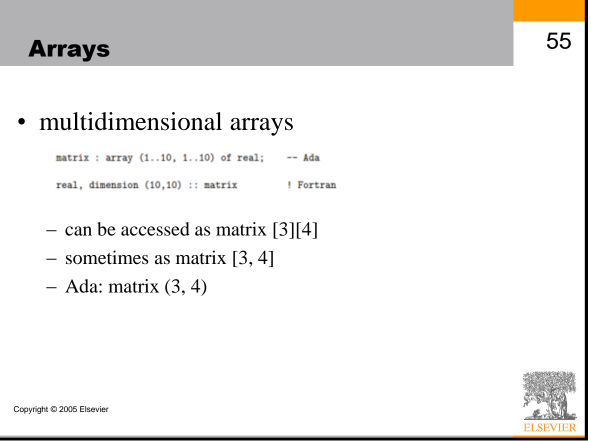

Records (Structures) and
Variants (Unions)
50
Modula-2: variant part not required to be at end
4 bytes/32 bits
name
metallic
<true>
source
prevalence
atomic_number
atomic_weight
4 bytes/32 bits
name
metallic
<false>
lifetime
atomic_number
atomic_weight
Copyright © 2005 Elsevier
ELSEVIER

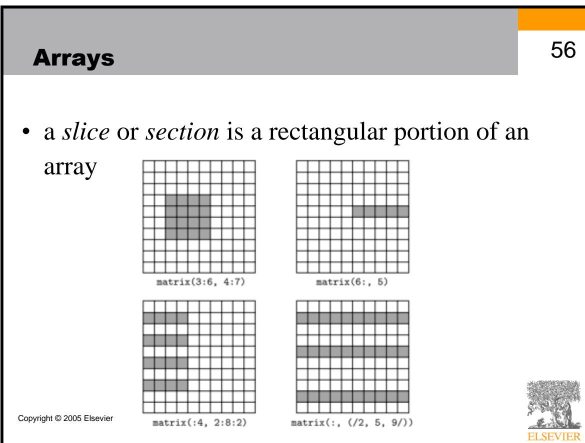

Arrays
51
arrays are the most common and important composite data types
unlike records, which group related fields of disparate types, arrays are usually homogeneous
semantically, they can be thought of as a mapping from an index type to a component or element type
Copyright © 2005 Elsevier
ELSEVIER

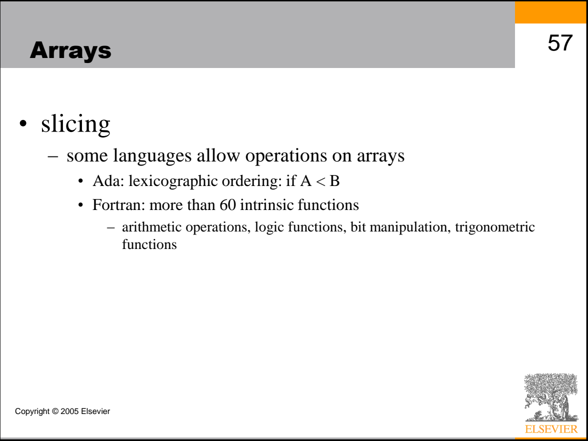

Arrays
52
index
most languages require the index to be an integer
or at least discrete type
non-discrete type
associative array
Python: dictionary
require hash table for lookup
Copyright © 2005 Elsevier
ELSEVIER

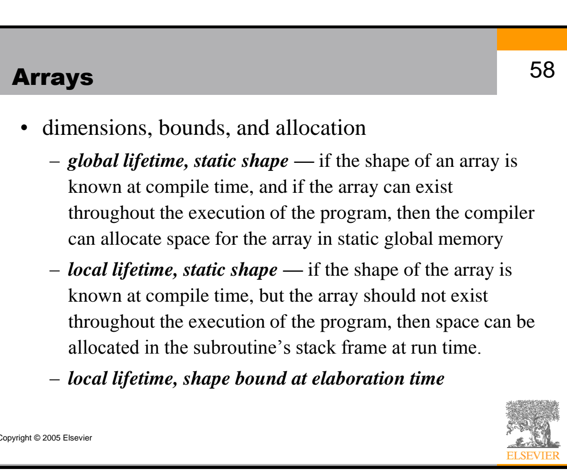

Arrays
53
syntax and operation
index usually inside square brackets [ ]
some languages use parentheses ( )
Copyright © 2005 Elsevier
ELSEVIER

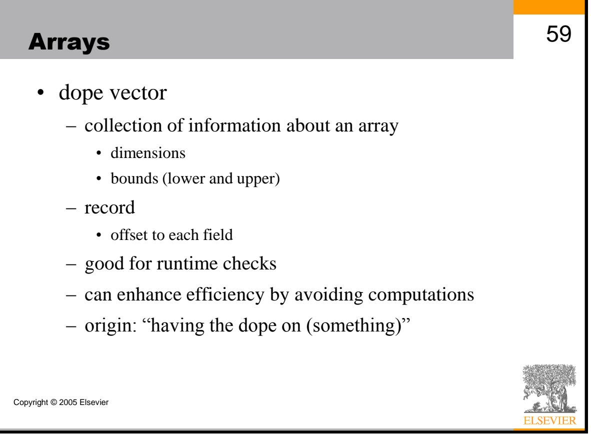

Arrays
54
declaration
C
char upper[26];
Fortran
character, dimension (1:26) :: upper
character (26) upper   ! shorthand notation
Pascal
var upper : array ['a'..'z'] of char;
Ada
upper : array (character range 'a'..'z') of character;
Copyright © 2005 Elsevier
ELSEVIER











Arrays

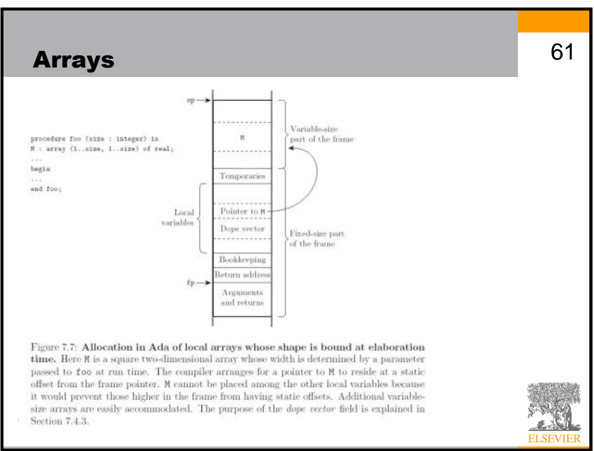
60

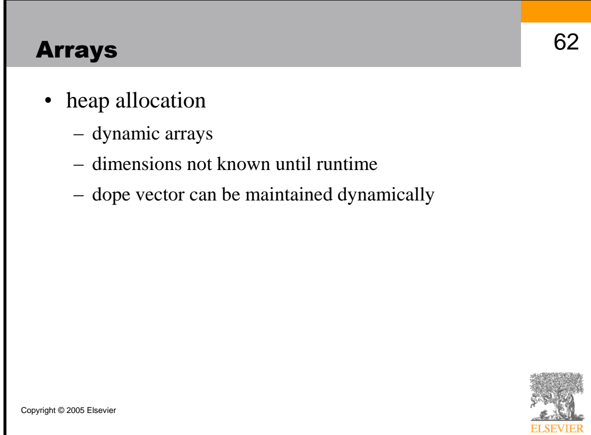
- stack allocation
  - conformant arrays
    - parameters to subroutines
    - dimensions not known
    - dope vector passed to help

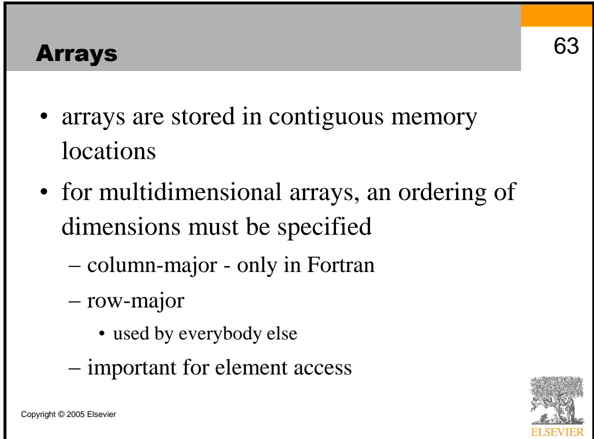
```
function DotProduct(A, B : array [lower..upper : integer] of real)
    : real;
var i : integer;
    rtn : real;
begin
    rtn := 0;
    for i := lower to upper do rtn := rtn + A[i] * B[i];
    DotProduct := rtn
end;
```

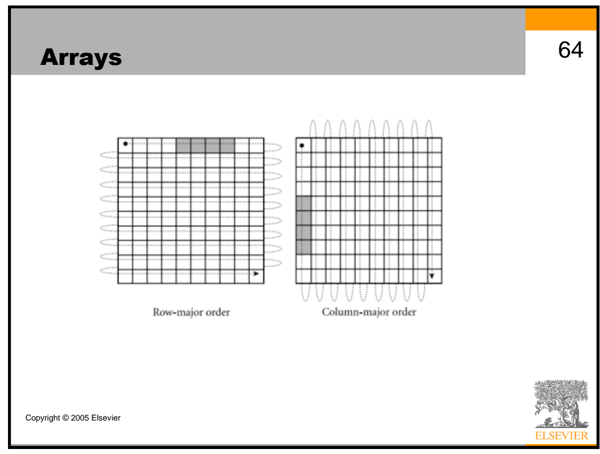
Copyright © 2005 Elsevier

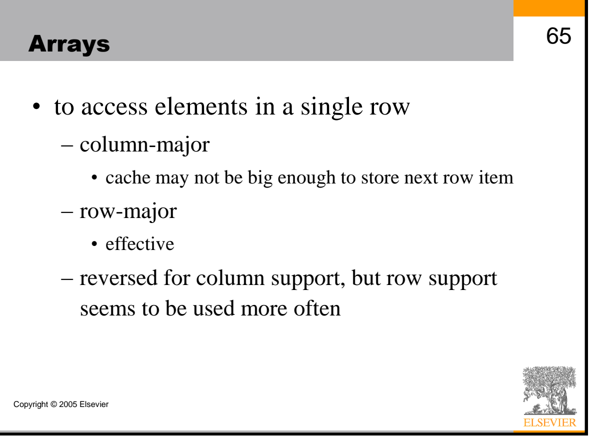
ELSEVIER

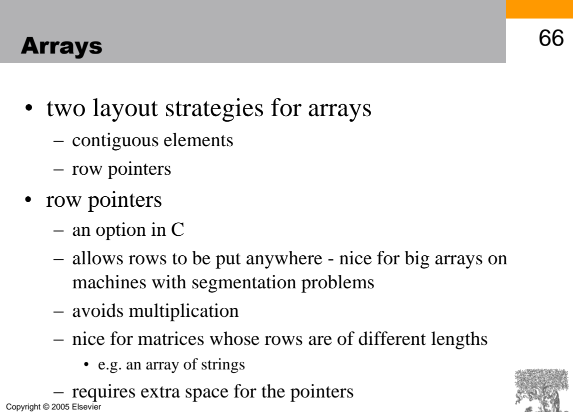

Arrays
61
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
...
end foo;
sp
Variable-size part of the frame
Temporaries
Local variables
Pointer to M
Dope vector
Fixed-size part of the frame
Bookkeeping
Return address
fp
Arguments and returns
Figure 7.7: Allocation in Ada of local arrays whose shape is bound at elaboration time. Here M is a square two-dimensional array whose width is determined by a parameter passed to foo at run time. The compiler arranges for a pointer to M to reside at a static offset from the frame pointer. M cannot be placed among the other local variables because it would prevent those higher in the frame from having static offsets. Additional variable-size arrays are easily accommodated. The purpose of the dope vector field is explained in Section 7.4.3.
ELSEVIER

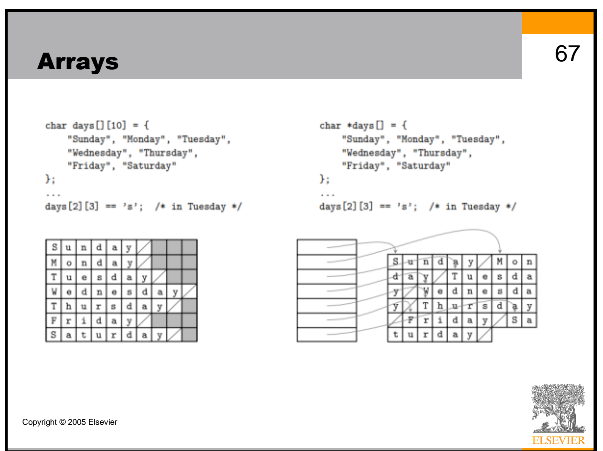

Arrays
62
• heap allocation
– dynamic arrays
– dimensions not known until runtime
– dope vector can be maintained dynamically
Copyright © 2005 Elsevier
ELSEVIER

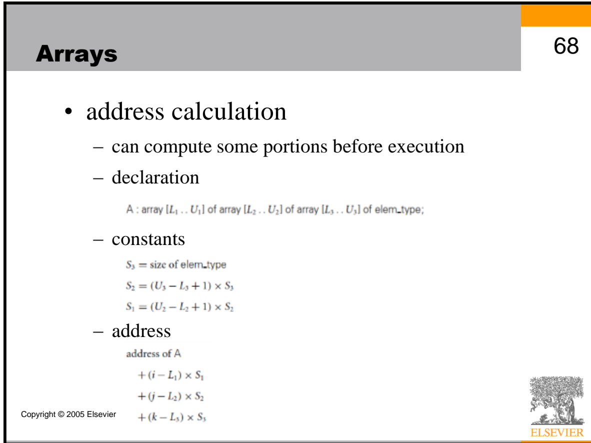

Arrays
63
• arrays are stored in contiguous memory locations
• for multidimensional arrays, an ordering of dimensions must be specified
– column-major - only in Fortran
– row-major
• used by everybody else
– important for element access
Copyright © 2005 Elsevier
ELSEVIER

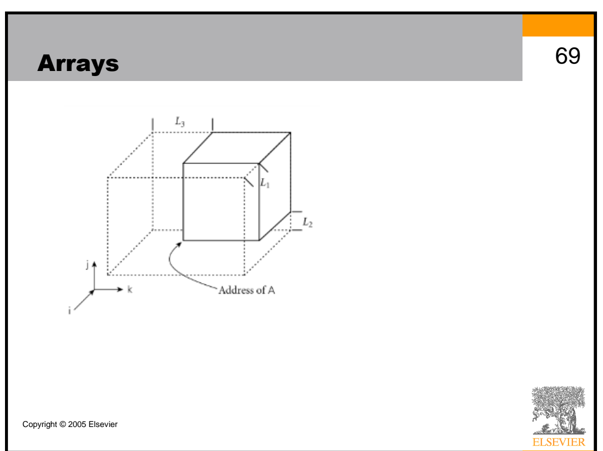

Arrays
64
Row-major order
Column-major order
Copyright © 2005 Elsevier
ELSEVIER

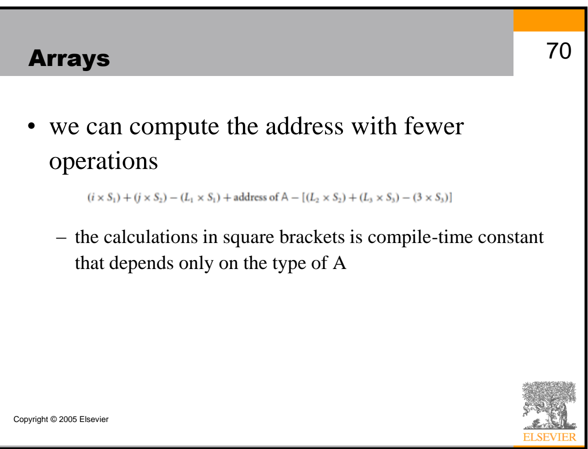

Arrays
65
• to access elements in a single row
– column-major
• cache may not be big enough to store next row item
– row-major
• effective
– reversed for column support, but row support seems to be used more often
Copyright © 2005 Elsevier
ELSEVIER


Arrays
66
• 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
Copyright © 2005 Elsevier
ELSEVIER









Strings

71

- 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
  - easier to provide these operations for strings than for arrays in general because strings are one-dimensional and (more importantly) non-circular

Copyright © 2005 Elsevier

ELSEVIER

Sets

72

- possible implementations
  - bitsets are what usually get built into programming languages
  - 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 implementer
    - for 32-bit int’s, bitset size is 500MB
    - for 64-bit int’s, bitset size enormous

Copyright © 2005 Elsevier

ELSEVIER

Pointers and Recursive Types

73

- pointers serve two purposes
  - efficient (and sometimes intuitive) access to elaborated objects (as in C)
  - dynamic creation of linked data structures (recursive types), in conjunction with a heap storage manager
- several languages (e.g., Pascal) restrict pointers to accessing objects in the heap

Copyright © 2005 Elsevier

Pointers and Recursive Types

74

- operations on pointers
  - allocation and deallocation of objects in the heap
  - dereferencing of pointers to access the objects to which they point
  - assignment from one pointer to another
- operations behave differently depending on whether the language employs a reference model or value model for names
  - functional languages use reference model

Copyright © 2005 Elsevier

Pointers and Recursive Types

75

- reference model
  - ML example

```
datatype chr_tree = empty | node of char * chr_tree * chr_tree;
```

  - example tree

```
(#"R", node (#"X", empty, empty), node
(#"Y", node (#"Z", empty, empty), node (#"W", empty, empty)))
```

  - block storage from heap

Copyright © 2005 Elsevier

Pointers and Recursive Types

76

Copyright © 2005 Elsevier

Pointers and Recursive Types

77

- reference model
  - Lisp example

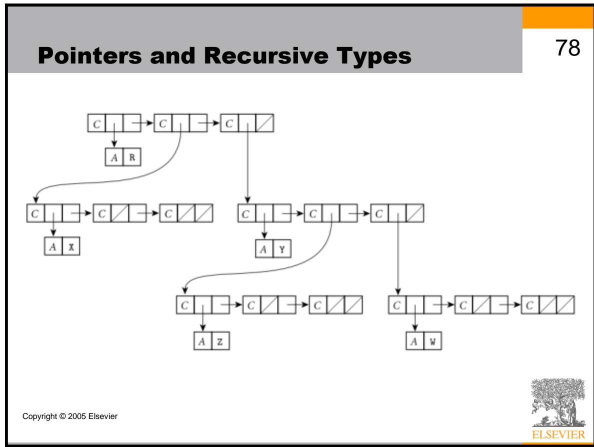
```
'(#\R (#\X ()()) (#\Y (#\Z ()())(#\W ()())))
```

  - each level of parentheses brackets the elements of a list
    - head and remainder of list
    - each node is denoted as a construct cell or atom

Copyright © 2005 Elsevier

Pointers and Recursive Types

78



Copyright © 2005 Elsevier

Pointers and Recursive Types

79

- references in functional languages are acyclic
  - circular references typically found only in imperative languages
- often need mutually recursive types
  - example: symbol table and syntax tree nodes need to refer to each other
  - if declared one at a time, not able to refer to each other

Copyright © 2005 Elsevier

Pointers and Recursive Types

80

- mutually recursive types
  - ML example: symbol table and syntax tree nodes declared together

```
datatype sym_tab_rec = variable of ...
    | type of ...
    | ...
    | subroutine of {code : syn_tree_node, ...}
and syn_tree_node = expression of ...
    | loop of ...
    | ...
    | subr_call of {subr : sym_tab_rec, ...};
```

Copyright © 2005 Elsevier

Pointers and Recursive Types

81

- tree types in value model languages
  - Pascal

```
type chr_tree_ptr = ^chr_tree;
     chr_tree = record
        left, right : chr_tree_ptr;
        val : char
     end;
```

  - Ada

```
type chr_tree;
type chr_tree_ptr is access chr_tree;
type chr_tree is record
    left, right : chr_tree_ptr;
    val : character;
end record;
```

Copyright © 2005 Elsevier

Pointers and Recursive Types

82

- tree types in value model languages (cont.)
  - C

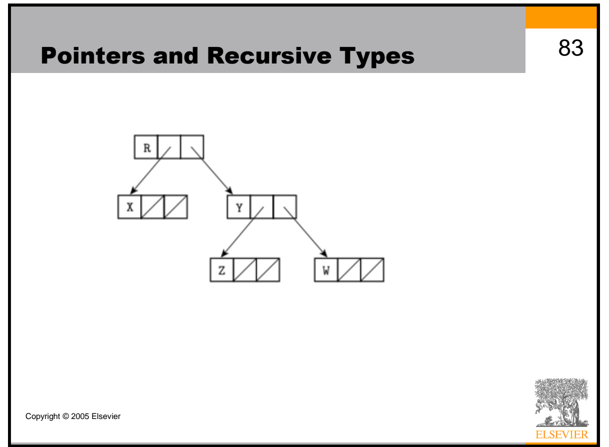
```
struct chr_tree {
    struct chr_tree *left, *right;
    char val;
};
```

- for mutually recursive types
  - Pascal: forward references
  - Ada and C: incomplete data types

Copyright © 2005 Elsevier

Pointers and Recursive Types

83



Copyright © 2005 Elsevier

Pointers and Recursive Types

84

- allocation of space from the heap
  - Pascal

```
new(my_ptr);
```

  - Ada

```
my_ptr := new chr_tree;
```

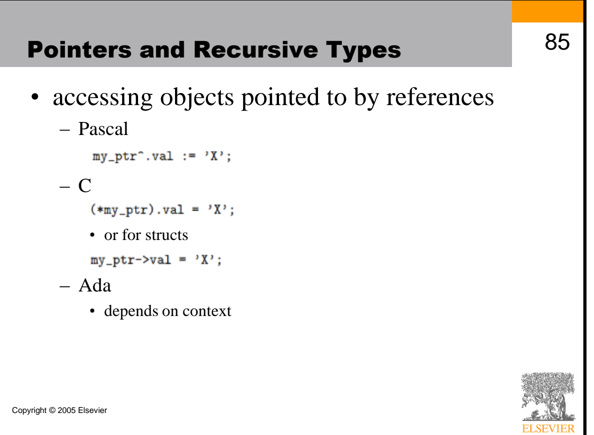
  - C

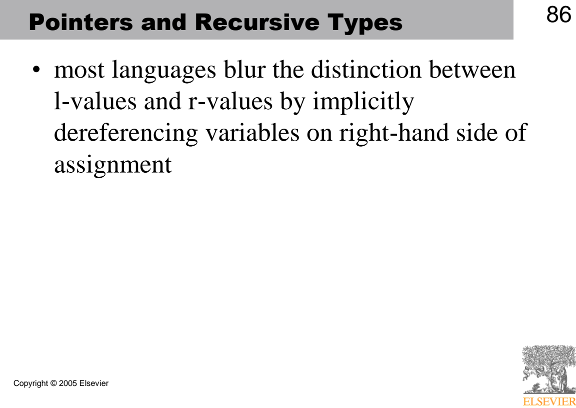
```
my_ptr = (struct chr_tree *) malloc(sizeof(struct chr_tree));
```

    - returns void*
  - C++, Java, C#

```
my_ptr = new chr_tree( arg_list );
```

Copyright © 2005 Elsevier





Pointers and Recursive Types

87

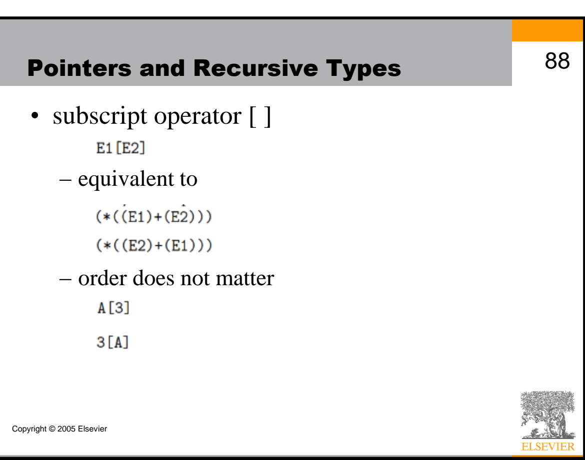
- C pointers and arrays

```
int n;
int *a;        /* pointer to integer */
int b[10];     /* array of 10 integers */
```

  - all of the following are valid

```
1. a = b;          /* make a point to the initial element of b */
2. n = a[3];
3. n = *(a+3);     /* equivalent to previous line */
4. n = b[3];
5. n = *(b+3);     /* equivalent to previous line */
```

Copyright © 2005 Elsevier



Pointers and Recursive Types

89

- C pointers and arrays closely linked

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

- some subtle differences
  - a declaration allocates an array if it specifies a size in 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
```

Copyright © 2005 Elsevier

Pointers and Recursive Types

90

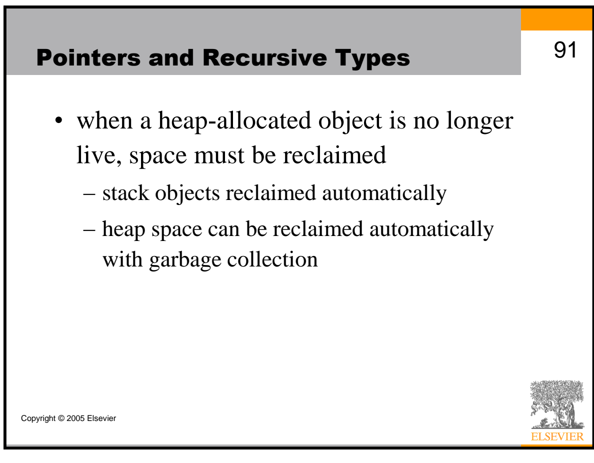
- declaring an array requires all dimensions except the last
  - invalid

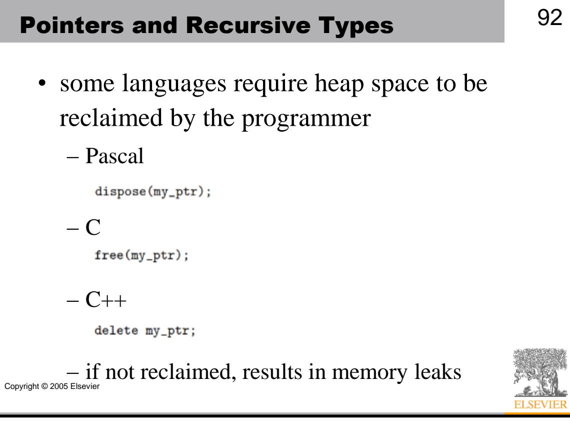
```
int a[][];
int (*a)[];
```

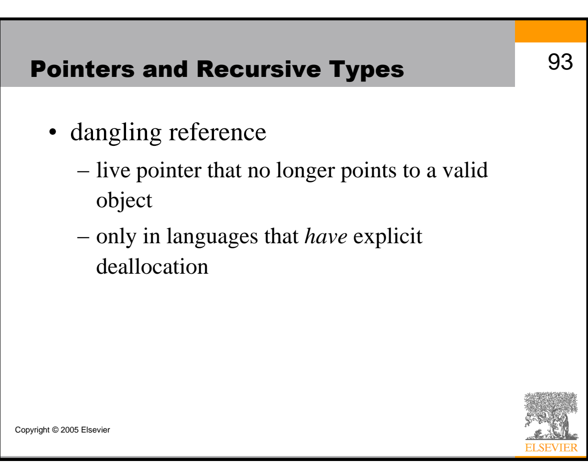
  - C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat

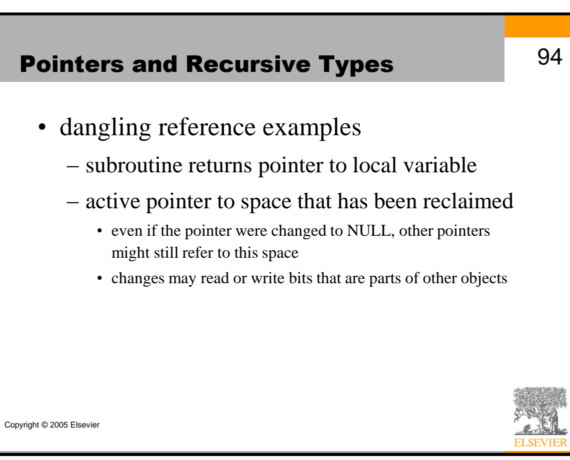
```
int *a[n];   // n-element array of pointers to
             //  integer
int (*a)[n]; // pointer to n-element array
             //  of integers
```

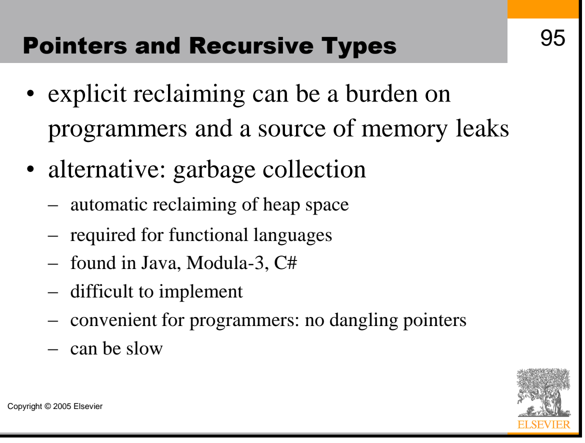
Copyright © 2005 Elsevier

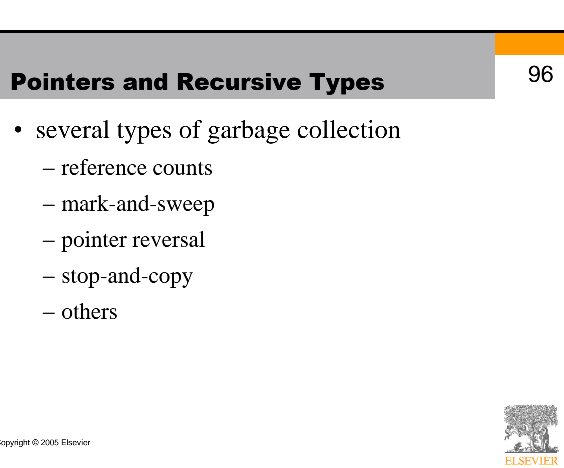

Pointers and Recursive Types
91
• when a heap-allocated object is no longer live, space must be reclaimed
– stack objects reclaimed automatically
– heap space can be reclaimed automatically with garbage collection
Copyright © 2005 Elsevier
ELSEVIER


Pointers and Recursive Types
92
• some languages require heap space to be reclaimed by the programmer
– Pascal
dispose(my_ptr);
– C
free(my_ptr);
– C++
delete my_ptr;
– if not reclaimed, results in memory leaks
Copyright © 2005 Elsevier
ELSEVIER


Pointers and Recursive Types
93
• dangling reference
– live pointer that no longer points to a valid object
– only in languages that have explicit deallocation
Copyright © 2005 Elsevier
ELSEVIER


Pointers and Recursive Types
94
• dangling reference examples
– subroutine returns pointer to local variable
– active pointer to space that has been reclaimed
• even if the pointer were changed to NULL, other pointers might still refer to this space
• changes may read or write bits that are parts of other objects
Copyright © 2005 Elsevier
ELSEVIER


Pointers and Recursive Types
95
• explicit reclaiming can be a burden on programmers and a source of memory leaks
• alternative: garbage collection
– automatic reclaiming of heap space
– required for functional languages
– found in Java, Modula-3, C#
– difficult to implement
– convenient for programmers: no dangling pointers
– can be slow
Copyright © 2005 Elsevier
ELSEVIER


Pointers and Recursive Types
96
• several types of garbage collection
– reference counts
– mark-and-sweep
– pointer reversal
– stop-and-copy
– others
Copyright © 2005 Elsevier
ELSEVIER

Pointers and Recursive Types

97

- reference counts
  - one way to designate an object as garbage is when no pointers to it exist
  - place a reference count in each object that keeps track of the number of pointers to it
  - when object is created, reference count is set to 1
  - when pointers are assigned or re-assigned, count is incremented/decremented
  - when count reaches 0, space can be reclaimed

Copyright © 2005 Elsevier

Pointers and Recursive Types

98

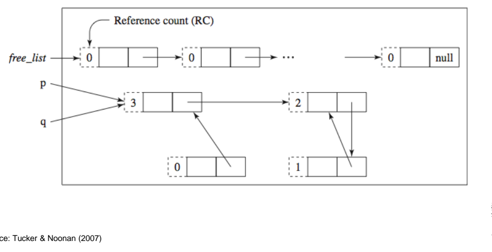
- reference counts (cont.)
  - reference count also decremented upon subroutine return if local pointers involved
  - system must run recursively, so that any pointers in reclaimed space are accounted for
  - pointers must otherwise be initialized to null
  - works well with strings, which have no circular dependencies

Copyright © 2005 Elsevier

Pointers and Recursive Types

99

- the heap is a chain of nodes (the *free_list)*
- each node has a reference count (RC)
- for an assignment, like q = p, garbage can occur



Source: Tucker & Noonan (2007)

ELSEVIER

Pointers and Recursive Types

100

- reference counts may miss circular structures

Stack
Heap
stooges
"larry"
"moe"
"curly"
stooges := nil;
stooges

Copyright © 2005 Elsevier

Pointers and Recursive Types

101

- disadvantages
  - failure to detect inaccessible circular chains
  - storage overhead created by appending an integer reference count to every node in the heap
  - performance overhead whenever a pointer is assigned or a heap block is allocated or deallocated

Source: Tucker & Noonan (2007)

Pointers and Recursive Types

102

- some garbage collection schemes use tracing
  - instead of counting the number or references to an object
  - find all good space by following valid pointers that are active
  - work recursively through the heap, starting from external pointers

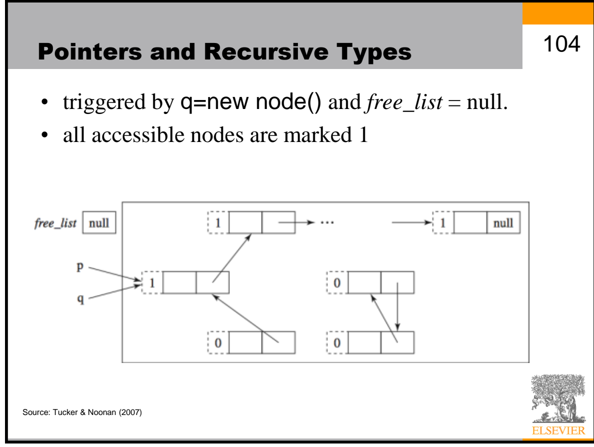
Copyright © 2005 Elsevier

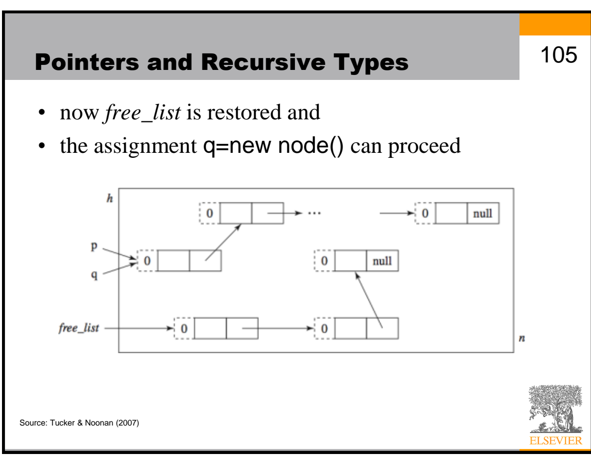
Pointers and Recursive Types

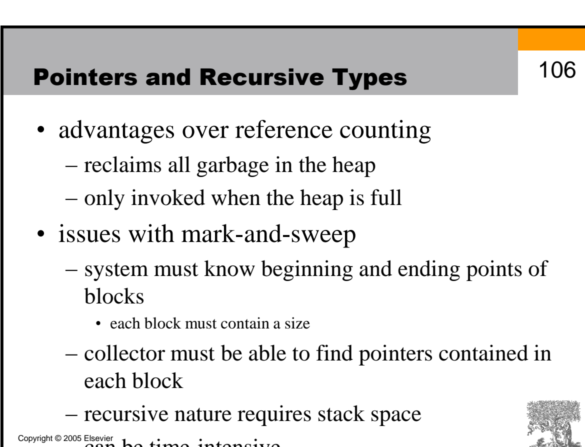
103

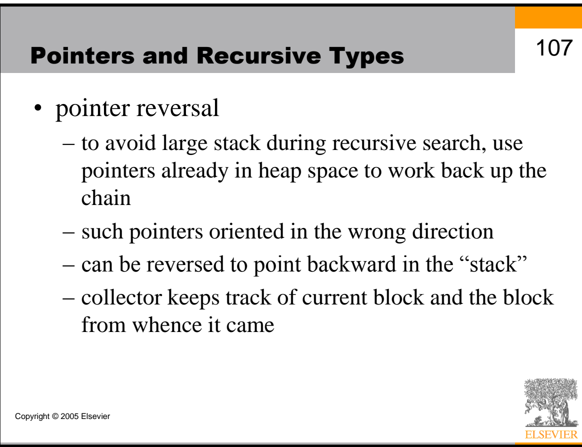
- mark-and-sweep
  - 3 main steps executed when space is low
    1. all blocks in the heap marked with 0
    2. follows pointers outside the heap and recursively explores all heap space; each newly discovered block marked with 1
       - if already marked with 1, can stop following that chain
    3. final walk through heap to remove every block still marked with 0

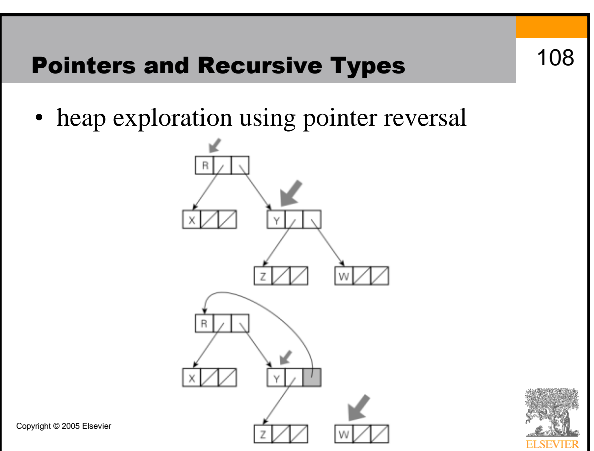
Copyright © 2005 Elsevier











Pointers and Recursive Types

109

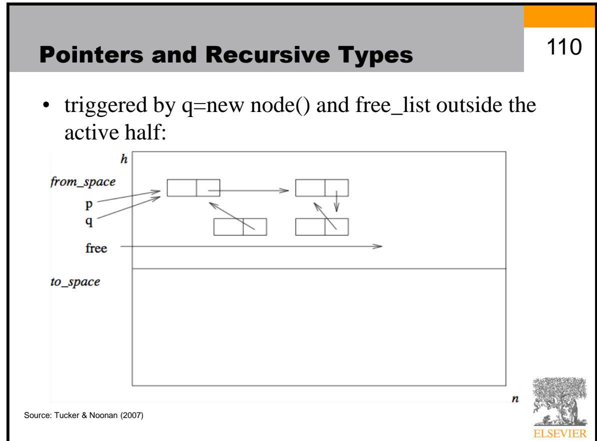
- stop-and-copy (aka copy collection)
  - heap space divided into two halves
    - only one is active at any given time
  - when one half nearly full, collector recursively explores heap, copying all valid blocks to other half
  - defragments while copying
  - pointers to blocks changed during copy
  - when finished, only useful blocks in heap; other half is no longer used
  - on next collection, halves are reversed

Copyright © 2005 Elsevier

Pointers and Recursive Types

110

- triggered by q=new node() and free_list outside the active half:

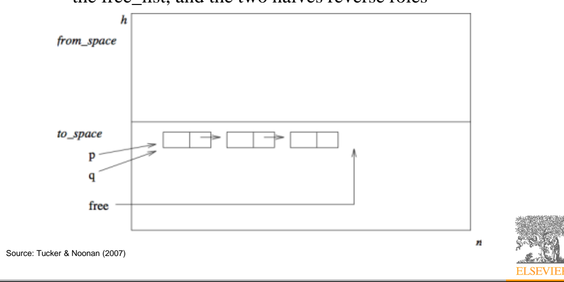


Source: Tucker & Noonan (2007)

Pointers and Recursive Types

111

- accessible nodes copied to other half
  - the accessible nodes are packed, orphans are returned to the free_list, and the two halves reverse roles



Source: Tucker & Noonan (2007)

Pointers and Recursive Types

112

- issues with stop-and-copy
  - only half of the heap can be used at any one time
  - incurs additional overhead
  - can be time-intensive

Copyright © 2005 Elsevier

Pointers and Recursive Types

113

- other techniques
  - generational collection
    - since most dynamically allocated blocks are short-lived, newer ones stored in half the heap
    - this half is checked first if garbage collection necessary
    - during garbage collection, valid blocks transferred to long-term heap
    - system must examine long-term heap if more space needed, and to reduce memory leaks
  - conservative collection
    - if blocks in the heap do not state explicitly the location of pointers, any space that "looks like" a pointer can be considered such and used with mark-and-sweep
    - may leave some useless blocks, but guaranteed not to delete currently used blocks

Copyright © 2005 Elsevier

Pointers and Recursive Types

114

- summary
  - modern algorithms are more elaborate
    - most are hybrids/refinements of the above three
  - in Java, garbage collection is built-in
    - runs as a low-priority thread
    - also, `System.gc` may be called by the program
  - functional languages have garbage collection built-in
  - C/C++ default garbage collection to the programmer

Source: Tucker & Noonan (2007)

Lists

115

- 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
  - ideally suited to programming in functional and logic languages, which rely on recursion
    - Lisp: 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

Copyright © 2005 Elsevier

Lists

116

- list operations tend to generate garbage
  - useful to have automatic garbage collection
- some languages (e.g., ML) require lists to contain items of the same type
- other languages (e.g., Python, Lisp) allow lists to contain items of differing types
  - implemented as chain of two pointers: one to element and one to next block

Copyright © 2005 Elsevier

Lists

117

- since everything in Lisp is a list, differentiation required between lists to be evaluated and constant lists
- Lisp list constructors

```
(cons 'a '(b))          ⟹ (a b)
(car '(a b))            ⟹ a
(car nil)               ⟹ ??
(cdr '(a b c))          ⟹ (b c)
(cdr '(a))              ⟹ nil
(cdr nil)               ⟹ ??
(append '(a b) '(c d))  ⟹ (a b c d)
```

Copyright © 2005 Elsevier

Lists

118

- list comprehension (e.g., Haskell, Python)
  - Python example: all odd numbers less than 100

```
[i*i for i in range (1, 100) if i % 2 == 1]
```

Copyright © 2005 Elsevier

Files and Input/Output

119

- input/output (I/O) facilities allow a program to communicate with the outside world
  - 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*

Copyright © 2005 Elsevier

Equality Testing and Assignment

120

- when an expression is encountered testing for equality, how should it be interpreted?
  - for primitive types, easily defined
  - example: s = t for character strings
    - are s and t aliases?
    - occupy the same storage that is bit-wise identical over its full length?
    - contain the same sequence of characters?
    - would appear the same if printed?
  - shallow vs. deep comparison

Copyright © 2005 Elsevier

Equality Testing and Assignment

121

- Scheme has distinct equality-testing functions for each

```
(eq? a b)        ; do a and b refer to the same object?
(eqv? a b)       ; are a and b provably semantically equivalent?
(equal? a b)     ; do a and b have the same recursive structure?
```

  - behavior

```
(eq? #t #t)            ⟹  #t (true)
(eq? 'foo 'foo)        ⟹  #t
(eq? '(a b) '(a b))    ⟹  #f (false); created by separate cons-es
(let ((p '(a b)))
  (eq? p p))           ⟹  #t; created by the same cons
(eq? 2 2)              ⟹  unspecified
(eq? "foo" "foo")      ⟹  unspecified
```

- deep assignments rare in any language

Copyright © 2005 Elsevier