Chapter 7:: Data Types

1

Programming Language Pragmatics

Michael L. Scott

1

Types

2

- most programming languages include a notion of types for variables, expressions, functions, etc.
- two main purposes of types
  - give a context for operations
    - example: a + b means mathematical addition if a and b are integers
  - limit the set of operations that can be performed and thereby prevents mistakes
    - type checking cannot prevent all invalid operations, but it catches a lot of them

2

Types

3

- types provide meaning and structure
- computer only sees bits in memory and does not know about types
- bits can be interpreted in many ways
  - instructions
  - addresses
  - characters, integers, floats, etc.
- bits are untyped, but higher-level languages associate types with values
- types guide the interpretation of the bits
  - for operator context performed at the machine level
    - e.g., integer addition vs. floating point addition
  - error checking
    - e.g., are we adding values of compatible types?

3

Type Checking

4

- type checking ensures a program obeys the type compatibility rules of the language
  - type error: violation of type rules
  - type safety: extent to which a language prevents type errors
- different levels of type checking
  - strongly typed if it prohibits an operation on an object for which the operation is not defined
  - statically typed if it is strongly typed and type checking can be performed at compile time
  - dynamically typed if type checking is performed at run time
- no standard definition of strongly typed

4

Type Checking

5

- some languages have sophisticated type systems
  - Haskell and ML
  - possible to determine the type of every expression at compile time
  - strongly statically typed
- some languages perform implicit or explicit conversions
  - C and Fortran
  - undermines type safety
  - weakly strongly typed
- Python is dynamically typed
- some languages are completely untyped
  - assembly languages

5

Type Checking

6

- data may need to be changed to a different type before an operation can be completed
- two different types of type changes
  - type conversions (explicit)
    - sometimes the word 'cast' is used for conversions (e.g., C)
  - type coercions (implicit)

6

Type Checking

7

- 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
c : celsius_temp;     -- as in Example 7.19
...
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
```


7

Type Checking

8

- 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

```
int a;
 float b, c;
       ...
 c := a + b;
```

  - 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

8

Type Checking

9

- C has lots of coercion, but with simple rules
  - all **`floats`** in expressions become **`doubles`**
  - **`short int`** and **`char`** become **`int`** in expressions
  - if necessary, precision is removed when assigning into LHS

9

Type Checking

10

- example in C, which does a bit of coercion

```
short int s;
unsigned long int l;
char c;     /* may be signed or unsigned -- implementation-dependent */
float f;    /* usually IEEE single-precision */
double d;   /* usually IEEE double-precision */
...
s = l;  /* l's low-order bits are interpreted as a signed number. */
l = s;  /* s is sign-extended to the longer length, then
           its bits are interpreted as an unsigned number. */
s = c;  /* c is either sign-extended or zero-extended to s's length;
           the result is then interpreted as a signed number. */
f = l;  /* l is converted to floating-point.  Since f has fewer
           significant bits, some precision may be lost. */

d = f;  /* f is converted to the longer format; no precision lost. */
f = d;  /* d is converted to the shorter format; precision may be lost.
           If d's value cannot be represented in single-precision, the
           result is undefined, but NOT a dynamic semantic error. */
```


10

Type Checking

11

- in effect, coercion rules are a relaxation of type checking
  - recent thought is that this is probably a bad idea
  - some languages, such as Ada, do not permit coercions
  - C++, however, goes hog-wild with them
  - they're one of the hardest parts of the language to understand

11

Type Checking

12

- 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?

12

Type Checking

13

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

13

Type Checking

14

- 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;
}
```


14

Type Checking

15

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

15

Type Checking

16

- 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

16

Type Checking

17

- 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 schemes, we begin by putting every type description in a standard form that takes care of "obviously unimportant" distinctions like those above

17

Type Systems

18

- 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

18

Data Types

19

- types reflect the internal structure of data, down to the level of a small set of fundamental types
  - structural approach to types
- other approaches are more mathematical
  - denotational: collection of values from a domain
  - abstraction: operations that can be applied to objects of a type
- types can be specified in different ways
  - Fortran: implicitly by the spelling of variable names, explicitly in variable declarations
  - C, C++: explicitly in variable declarations
  - Python: by the type of the right-hand side of an assignment

19

Data Types

20

- 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
  - floating point
- languages also supply composite types built from the basic data types
  - C: structure, array, union
  - C++, Python: class
  - Python: list, dictionary

20

Data Types

21

- universal reference types
  - a reference that can point to anything
  - C, C+: void
  - Java: Object
  - C#: object

21

Classification of Types

22

- numeric types
  - C and Fortran distinguish between different lengths of integers
  - C, C++, C#, Modula-2: signed and unsigned integers
  - differences in float precision cause unwanted behavior across machine platforms
  - some languages provide complex, rational, or decimal types

22

Classification of Types

23

- 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;
```


23

Classification of Types

24

- 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};
```


24

Classification of Types

25

- 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

25

Classification of Types

26

- 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

26

Classification of Types

27

- 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)

27

Polymorphism

28

- 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

28

Polymorphism

29

- 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

29

Records and Variants

30

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

30

Records and Variants

31

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

31

Records and Variants

32

- 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;
```


32

Records and Variants

33

- 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

33

Records and Variants

34

- 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;
```


34

Records and Variants

35

- 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

35

Records and Variants

36

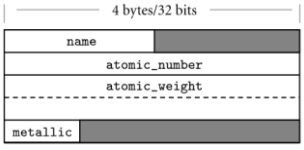
- records
  - usually laid out contiguously
  - addresses are computed as base + offset
  - possible holes for alignment reasons
  - smart compilers may re-arrange fields to minimize holes (C compilers promise not to)
  - implementation problems are caused by records containing dynamic arrays

36

Records and Variants

37

- records example
  - each element must begin on proper boundary
    - int on 4-byte boundary
    - short on 2-byte boundary
    - char on 1-byte boundary
    - user-defined type begins and ends on word boundary
  - holes for alignment



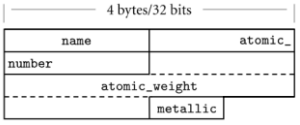
37

Records and Variants

38

- some languages, such as Pascal, allow the data to be packed
  - boundary alignment overridden
  - generally inefficient
- compiler should optimize for space over speed

```
type element = packed record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;
```



- less space, but system takes longer to access elements

38

Records and Variants

39

- some languages, such as C, allow record assignment

```
my_element = copper;
```

- or record tests for equality

```
if (my_element == copper) …
```

- most other languages do not

39

Records and Variants

40

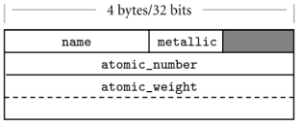
- record equality can be done field by field
- to save time, a block compare function can be used
  - compares bytes
  - holes need to be filled with 0's to allow block comparisons to work correctly

40

Records and Variants

41

- holes in records waste space
  - packing helps, but with performance penalties
- smart compilers rearrange fields automatically (not C compilers)
  - programmer unaware
  - some systems-level applications may depend on non-rearranged fields



41

Records and Variants

42

- 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

42

Records and Variants

43

- Pascal

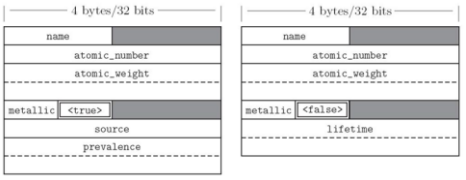
```
type long_string = packed array [1..200] of char;
type string_ptr = ^long_string;
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
    case naturally_occurring : Boolean of
      true : (
        source : string_ptr;
            (* 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;
```


43

Records and Variants

44

- variant part must be at end



44

Records and Variants

45

- Modula-2: variant part not required to be at end

45

Orthogonality

46

- 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)
- for example
  - Pascal is more orthogonal than Fortran because it allows arrays of anything
- orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

46

Arrays

47

- 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
- index
  - most languages require the index to be an integer at least discrete type
  - index usually inside square brackets [ ]
  - some languages use parentheses ( )
- non-discrete type
  - associative array
  - Python: dictionary
  - require hash table for lookup

47

Arrays

48

- 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;
```


48

Arrays

49

- multidimensional arrays

```
matrix : array (1..10, 1..10) of real;    -- Ada
real, dimension (10,10) :: matrix         ! Fortran
```

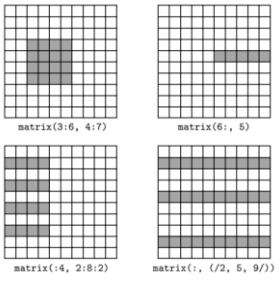
- accessed as matrix [3][4]
  - sometimes as matrix [3, 4]
- Ada: matrix (3, 4)

49

Arrays

50

- a slice or section is a rectangular portion of an array



50

Arrays

51

- slicing
  - some languages allow operations on arrays
  - Ada: lexicographic ordering: if A < B
  - Fortran: more than 60 intrinsic functions
    - arithmetic operations, logic functions, bit manipulation, trigonometric functions

51

Arrays

52

- dimensions, bounds, and allocation
  - global lifetime, static shape — if the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in static global memory
  - local lifetime, static shape — if the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine's stack frame at run time
  - local lifetime, shape bound at elaboration time

52

Arrays

53

- dope vector
  - collection of information about an array
    - dimensions
    - bounds (lower and upper)
  - record
    - offset to each field
  - good for runtime checks
  - can enhance efficiency by avoiding computations
  - origin: "having the dope on (something)"

53

Arrays

54

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

54

Arrays

55

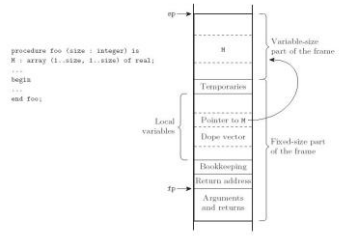


Figure 7.7: **Allocation in Ada of local arrays whose shape is bound at elaboration time.** Here `M` is a square two-dimensional array whose width is determined by a parameter passed to `foo` at run time. The compiler arranges for a pointer to `M` to reside at a static offset from the frame pointer. `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.

55

Arrays

56

- heap allocation
  - dynamic arrays
  - dimensions not known until runtime
  - dope vector can be maintained dynamically

56

Arrays

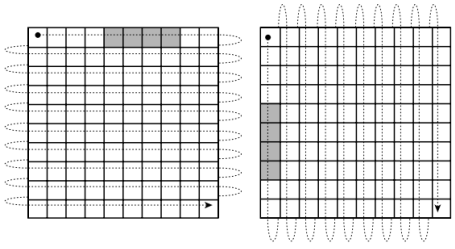
57

- 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

57

Arrays

58



58

Arrays

59

- 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

59

Arrays

60

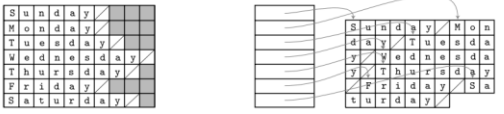
- two layout strategies for arrays
  - contiguous elements
  - row pointers
- row pointers
  - an option in C
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication
  - nice for matrices whose rows are of different lengths
    - e.g.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

60

Arrays


```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```



Arrays

- address calculation
- can compute some portions before execution

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

Sets

- 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

Pointers

- 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

Pointers

- 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

Pointers

67

- reference model
  - ML example

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

  - example tree

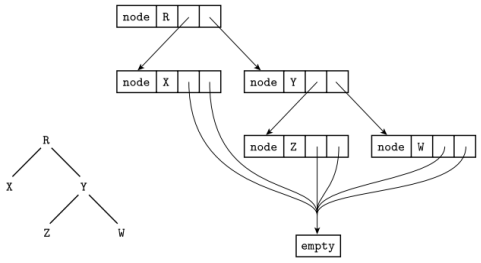
```
(#"R", node (#"X", empty, empty), node
(#"Y", node (#"Z", empty, empty), node (#"W", empty, empty)))
```

  - block storage from heap

67

Pointers

68



68

Pointers

69

- reference model
  - Lisp example

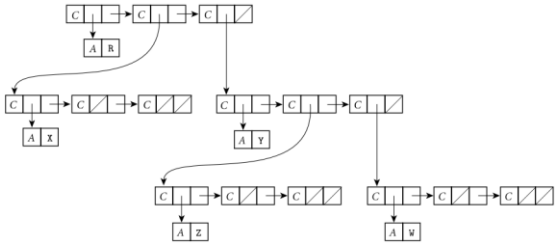
```
'(#\R (#\X ()()) (#\Y (#\Z ()()))
```

  - 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

69

Pointers

70



70

Pointers

71

- 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

71

Pointers

72

- 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;
```


72

Pointers

73

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

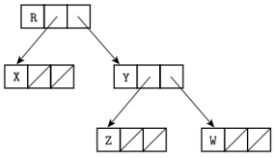
```
struct chr_tree {
    struct chr_tree *left, *right;
    char val;
};
```

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

73

Pointers

74



74

Pointers

75

- 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 );
```


75

Pointers

76

- accessing objects pointed to by references
  - Pascal

```
my_ptr^.val := 'X';
```

  - C

```
(*my_ptr).val = 'X';
```

    - or for structs

```
my_ptr->val = 'X';
```


76

Pointers

77

- most languages blur the distinction between l-values and r-values by implicitly dereferencing variables on right-hand side of assignment

77

Pointers

78

- 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 */
```


78

Pointers

79

- subscript operator [ ]
  `E1[E2]`
- equivalent to
  `(*((E1)+(E2)))`
  `(*((E2)+(E1)))`
- order does not matter
  `A[3]`
  `3[A]`

79

Pointers

80

- 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
```


80

Pointers

81

- 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
```


81

Pointers

82

- 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

82

Pointers

83

- 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

83

Pointers

84

- dangling reference
  - live pointer that no longer points to a valid object
  - only in languages that have explicit deallocation
- 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

84

Garbage Collection

85

- 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

85

Garbage Collection

86

- several types of garbage collection
  - reference counts
  - mark-and-sweep
  - pointer reversal
  - stop-and-copy
  - others

86

Garbage Collection

87

- 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

87

Garbage Collection

88

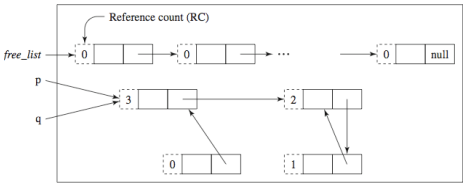
- reference counts (cont.)
  - reference count also decremented upon subroutine return if local pointers involved
  - system must run recursively, so that any pointers in reclaimed space are accounted for
  - pointers must otherwise be initialized to null
  - works well with strings, which have no circular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

88

Garbage Collection

89

- 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

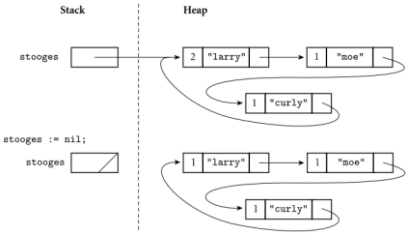


89

Garbage Collection

90

- reference counts may miss circular structures



90

Garbage Collection

91

- disadvantages of reference counts
  - 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

91

Garbage Collection

92

- 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

92

Garbage Collection

93

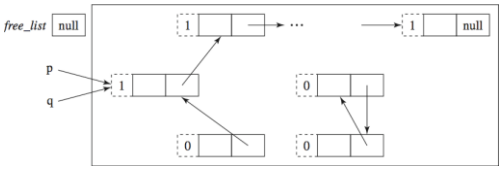
- mark-and-sweep
  - 3 main steps executed when space is low
    1. all blocks in the heap marked with 0
    2. follow pointers outside the heap and recursively explore all heap space
       - each newly discovered block marked with 1
       - if already marked with 1, stop following that chain
    3.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

93

Garbage Collection

94

- triggered by `q = new node()` and free_list = null
- all accessible nodes are marked 1

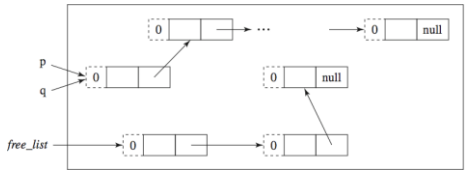


94

Garbage Collection

95

- now free_list is restored and
- the assignment `q = new node()` can proceed



95

Garbage Collection

96

- advantages over reference counting
  - reclaims all garbage in the heap
  - only invoked when the heap is full
- issues with mark-and-sweep
  - system must know beginning and ending points of blocks
    - each block must contain a size
  - collector must be able to find pointers contained in each block
  - recursive nature requires stack space
  - can be time-intensive

96

Garbage Collection

97

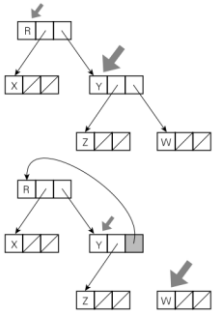
- pointer reversal
  - to avoid large stack during recursive search, use pointers already in heap space to work back up the chain
  - such pointers oriented in the wrong direction
  - can be reversed to point backward in the "stack"
  - collector keeps track of current block and the block from whence it came

97

Garbage Collection

98

- heap exploration using pointer reversal



98

Garbage Collection

99

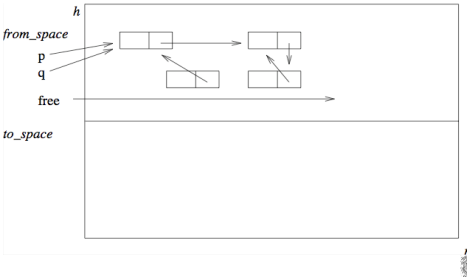
- stop-and-copy (aka copy collection)
  - heap space divided into two halves
    - only one is active at any given time
  - when one half nearly full, collector recursively explores heap, copying all valid blocks to other half
  - defragments while copying
  - pointers to blocks change during copy
 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

99

Garbage Collection

100

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

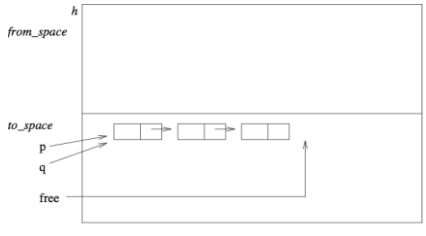


100

Garbage Collection

101

- accessible nodes copied to other half and defragmented
- the accessible nodes are packed, orphans are returned to the free_list, and the two halves reverse roles
- free_list always points to starting point of available memory



101

Garbage Collection

102

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

102

Garbage Collection

103

- other techniques
  - generational collection
    - since most dynamically allocated blocks are short-lived, newer ones stored in half the heap
    - this half 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

103

Garbage Collection

104

- other techniques (cont.)
  - 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

104

Garbage Collection

105

- 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

105

Lists

106

- 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

106

Lists

107

- 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

107

Lists

108

- 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)
```


108

Lists

109

- 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]
```


Files and Input/Output

110

- 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

Equality Testing and Assignment

111

- 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

Equality Testing and Assignment

112

- 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