CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Data Types

Language Specification

- General issues in the design and implementation of a language:
  - Syntax and semantics
  - Naming, scopes and bindings
  - Control flow
  - Data types
  - Subroutines

Data Types

- Why are types useful?
- Implicit context

  a + b
  - integer or floating-point addition, depending on the types of operands

  var p ^integer;

  new (p);
  - allocate the "right size", depending on the pointer type
  - in C++: call the appropriate constructor

- Checking
  - make sure that certain meaningless operations do not occur
  - cannot check everything, but enough to be useful

Type Systems

- A type system consists of:
  - Mechanism for defining types (syntax, semantics)
  - 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?)

- Type checking - ensure that a program obeys the compatibility rules
- Type clash - violation of compatibility rules

Type Systems

- A language may have:
  - Strong typing - language prevents you from applying an operation to data for which it is not appropriate
  - Static typing - language has strong typing, and it does type checking at compile time
  - Dynamic typing - type checking done at run time
- Examples:
  - Scheme - strong typing, but dynamic
  - Smalltalk - strong typing, but dynamic (operations <=> messages sent to objects)
  - Ada - strong and static typing
  - Pascal - "almost" strong and "almost" static typing
  - C - weaker than Pascal, but static (not much done at run time)
  - Java - strong typing, part done at compile time, part at run time

Definition of Types

- Several ways to think about types:
  - Denotational - collection of values for a domain
  - Constructive - internal structure of data, described down to the level of fundamental types
    - built-in types (integer, character, real, etc), also called primitive or predefined types
    - composite types (array, record, set, etc)
  - Abstraction - interface that specifies the operations that can be applied to objects
- Denotational – formal way of thinking → denotational semantics
- Constructive – widely used, introduced by Algol
- Abstraction – object-oriented way of thinking, introduced by Simula-67 and Smalltalk

Classification of Types

- Discrete (ordinal) types
  - countable domains
  - well defined notions of predecessor and successor
  - character, integer, boolean, enumeration, subrange
- Non-discrete
  - real, rational, complex

- Scalar (simple) types
  - each value conceptually corresponds to one number (either discrete or not)
  - discrete types, reals
- Composite types
  - have internal structure, defined in terms of simpler types
  - arrays, records, sets, lists

Scalar Types

- Boolean
  - implemented on one byte, 1 → true, 0 → false
- Character
  - one byte - ASCII encoding
  - two bytes ("wide characters") - Unicode character set (Java)
- Integer
  - length may not be specified by language - vary with implementation
  - signed or unsigned varieties
- Fixed point
  - represented as integers, but with implied decimal point at a specified position
  - currency values: 129.99

Scalar Types

- Floating point (real)
  - internally represented as sign *s*, mantissa *m* and exponent *exp*
  - value = $(-1)^s \times m \times 2^{exp}$
- Complex
  - pair of floating point numbers - real and imaginary parts
- Rational
  - pair of integers - numerator and denominator

Enumeration Types

- Introduced in Pascal:

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
```

- Values are ordered - allows for comparisons: mon < tue
- Predecessor and successor: tomorrow := succ (today);
- Enumeration-controlled loops:

```
for today := mon to fri do begin ...
```

- Index in arrays:

```
var daily_attendance : array [weekday] of integer;
daily_attendance [mon] := 40;
```

- Ordinal value:

```
ord (sun)        =>      1
```

Subrange Types

- Subrange – contiguous subset of values from a discrete base type
- Also introduced in Pascal:

```
type test_score = 0..100;
     workday = mon..fri;
```

- Why are subranges useful (why not just use integers)?
  - easier to read/document programs
  - semantic checks to ensure values are within range
  - efficient representation
    - test_score can be represented on a single byte

Composite Types

- Record (structure)
  - introduced in Cobol
  - collection of fields, with (potentially different) simpler types
  - mathematical formulation: type of a record → Cartesian product of field types
- Variant record
  - only one field is valid at any moment
  - type of a variant record → union of field types
- Array
  - components have the same type
  - type of an array → function that maps an index type to a component type

Composite Types

- Set
  - collection of distinct elements of a base type
- Pointer
  - reference to an object of the pointer's base type
- Lists
  - sequence of elements, no indexing
  - implemented as linked lists
- Files
  - notion of current position
  - usually accessed in sequential order

Type Checking

- Relation between an object type and the context where it is used:
  - Type equivalence
  - Type compatibility
  - Type inference
- Type equivalence vs. type compatibility:
  - equivalence - are the types the same?
  - minimal implementation - use an object only if the object type and the type expected by the context are equivalent
  - too restrictive → use compatibility
- Compatibility issues:
  - conversion (casting) - explicit
  - coercion - implicit
  - non-converting cast - does not change the bits, just interpret them as another type

Type Equivalence

- Two ways to define type equivalence:
  - Structural equivalence - same components, put together in the same way
  - Name equivalence - each definition introduces a new type

- Structural equivalence
  - Algol 68, Modula-3, C, ML
  - early Pascal

- Name equivalence
  - more popular lately
  - Java, standard Pascal, Ada

Structural Equivalence

- Are the following equivalent?

```
type T1 = record
    a, b : integer
      end;

type T2 = record
    a : integer
    b : integer
      end;

type T3 = record
    b : integer
    a : integer
      end;
```

- T1 and T2 - yes
- T2 and T3 – in ML no, in most other languages yes

Structural Equivalence

- Implementation-level way of thinking about types:

```
type student = record
    name, address : string
    age : integer
type school = record
    name, address : string
    age : integer

x : student
y : school

x := y;
```

- Should this be an error?
  - probably yes, although the types are structurally equivalent

Name Equivalence

- In general: different names → different types
  - Problem - handling type aliases:

```
TYPE stack_element = INTEGER;

MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
PROCEDURE push (elem : stack_element);
  ...
PROCEDURE pop () : stack_element;
  ...

VAR x : stack_element;
x = pop + 1;
```

  - Here *stack_element* and *integer* should be equivalent

Name Equivalence

- Same name $\leftrightarrow$ same type
- Problem - handling type aliases:

```
TYPE celsius_temp = REAL;
          fahrenheit_temp = REAL;
VAR  c : celsius_temp;
         f : fahrenheit_temp;
...
f := c;
```

```
TYPE college = school;
VAR  c : college;
         s : school;
...
s := c;
```

  - Here *celsius_temp* and *fahrenheit_temp* should not be equivalent
  - But *college* and *school* may be

Name Equivalence

- Variants:
  - Strict name equivalence - aliased types considered distinct
  - Loose name equivalence - aliased types considered equivalent
- Most languages use loose name equivalence (Pascal)
- Ada - allows the programmer to decide whether an alias is a derived type or a subtype:

```
subtype stack_element is integer;

type celsius_temp is new real;
type fahrenheit_temp is new real;
```

  - Subtype - equivalent to its parent (base) type, and to its sibling types
  - Derived type - a new type

Type Equivalence

- Example:

```
type alink = pointer to cell;
type blink = alink;
p, q  : pointer to cell;
r     : alink;
s     : blink;
t     : pointer to cell;
u     : alink;
```

- Structural equivalence
  - all six variables have same type
- Strict name equivalence
  - p, q, t have same type
  - r, u have same type
- Loose name equivalence
  - p, q, t have same type
  - r, s, u have same type

Announcements

- Readings
  - Chapters 7, 8

Type Conversion and Casts

a := expr

a + b

f (arg1, arg2, ... argN)

- Assume that the type provided and the type expected are required to be the same
- Need to use casts - explicit conversions
- Does it involve additional code at run-time?

- Several cases:
  - Types would be structurally equivalent, but language uses name equivalence
    - types use same representation
    - conceptual conversion → no additional code
  - Types have different sets of values, but same representation
    - subranges
    - if the provided type has some values that the expected type does not
      - checks at run-time, but no conversion of representation
  - Types have different representations
    - conversion of representation required
    - checks sometimes required (overflow, etc)

Type Conversion and Casts

- Example (Ada):

```
n : integer;
r : real;
t : test_score;                -- 0..100
c : celsius_temp;              -- real

t := test_score (n)            -- run-time check required
n := integer (t)               -- nothing
r := real (n)                  -- conversion
n := integer (r)               -- conversion and check
r := real (c)                  -- nothing
c := celsius_temp (r)          -- nothing
```

Type Conversion and Casts

- Non-converting type cast - change of type without altering the underlying bits
  - interpret the bits of a value as if they were of another type

- Examples:

- Memory allocation algorithms
  - heap → a very large array of characters (bytes)
  - reinterpret portions of the heap as various data structures

- High-performance numeric computations
  - reinterpret a float as an integer or record
  - extract the sign, mantissa, exponent

Type Conversion and Casts

- Non-converting type cast - example:

- In C – interpret the bits in an integer as a float:

```
int   n;
float f;
...
f = *((float *) &n); // can achieve this effect with pointers
```

Type Compatibility and Coercion

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

- Most languages do not require equivalence of types in every context, but just compatibility
- Coercion - implicit conversion, if types are compatible

- In Ada:
  - type S is compatible with type T if and only if:
    - S and T are equivalent
    - one is a subtype of the other
    - both are subtypes of the same type
    - both are arrays, with same numbers and types of elements in each dimension
- C
  - lots of coercions, some involve loss of precision (truncation)

Type Inference

- What is the type of an expression, given the types of the operands?

- Simple case
  - Same type as the (coerced) operands

- Complex case
  - A new type
  - Operations on subranges
  - Operations on composite objects

- More complex case
  - Types are not declared at all, so they need to be inferred
  - ML, Miranda, Haskell

Records

- Records – allow data of heterogeneous types to be manipulated together
  - Called structures in Algol 68, C, C++, Common Lisp
  - C++ structures – special form of classes (with all members public)
  - Java – no structures, only classes
- Example (Pascal):

```
type two_chars = packed array [1..2] of char;
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;

var copper : element;
```

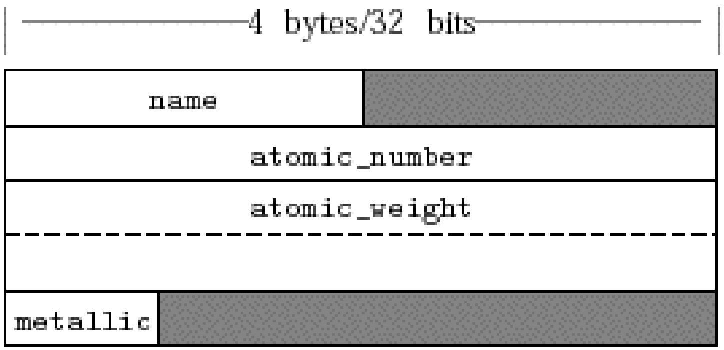
- Access to fields:

Pascal, C:	copper.name
Fortran:	copper%name
Cobol, Algol 68:	name of copper
ML:	#name copper
Common Lisp:	(element-name copper)

Records

- Memory layout – fields are usually stored next to each other
- Access – in the symbol table, for each field keep its offset from beginning of structure

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

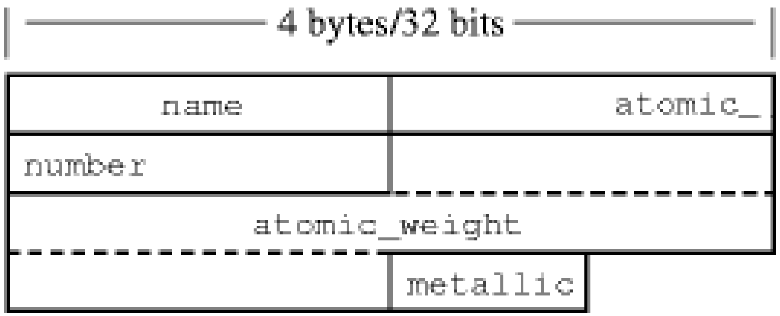


- Alignment restrictions → generate "holes"
- Size of the record in memory – 20 bytes

Records

- Packed records – available in Pascal, to eliminate holes

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



- Size of the record in memory – 15 bytes

- Drawback:
  - sacrifice speed for space
  - need multiple instructions (in target code) to access atomic_number and atomic-weight

Records

- Assignment – allowed in most languages (bit-by-bit copy):

  my_element := copper;

- Problem:
  - if some fields are pointers to dynamically allocated objects, only the pointers are copied, not the objects

- Comparison – generally not allowed
- Problem – why a bit-by-bit comparison will not work?
  - holes may contain random garbage
- Potential solution:
  - fill all holes with zero at allocation time – time consuming

With Statements

- Consider the following:

```
ruby.chemical_composition.elements[1].name := 'Al';
ruby.chemical_composition.elements[1].atomic_number := 13;
ruby.chemical_composition.elements[1].atomic_weight := 26.98154;
ruby.chemical_composition.elements[1].metallic := true;
```

- Simplify with with (in Pascal):

```
with ruby.chemical_composition.elements[1] do
begin
   name := 'Al';
   atomic_number := 13;
   atomic_weight := 26.98154;
   metallic := true
end;
```

With Statements

```
with ruby.chemical_composition.elements[1] do
begin
   name := 'Al';
   atomic_number := 13;
   atomic_weight := 26.98154;
   metallic := true
end;
```

- How would you do this in C (no with)?
  - Use a pointer:

```
p = & ruby.chemical_composition.elements[1];
p->name = 'Al';
p->atomic_number = 13;
p->atomic_weight = 26.98154;
p->metallic = 1;
```

With Statements

- Problems with with (in Pascal):
  - cannot "open" two records of the same type simultaneously
  - naming conflicts, if field names coincide with variable names
  - lack of clarity, if nested with statements are used

With Statements

- Modula-3 – introduces an alias:

```
WITH e = ruby.chemical_composition.elements[1] DO
    e.name := 'Al';
    e.atomic_number := 13;
    e.atomic_weight := 26.98154;
    e.metallic := true;
END;
```

- Several records can be accessed simultaneously:

```
WITH e = whatever, f = whatever DO
    e.field1 := f.field1;
    e.field2 := f.field2;
END;
```

- Modula-3 also allows with statements to create aliases for other objects than records:

```
WITH d = complicated_expression DO
    IF d <> 0 then val := n/d ELSE val := 0
```

Variant Records

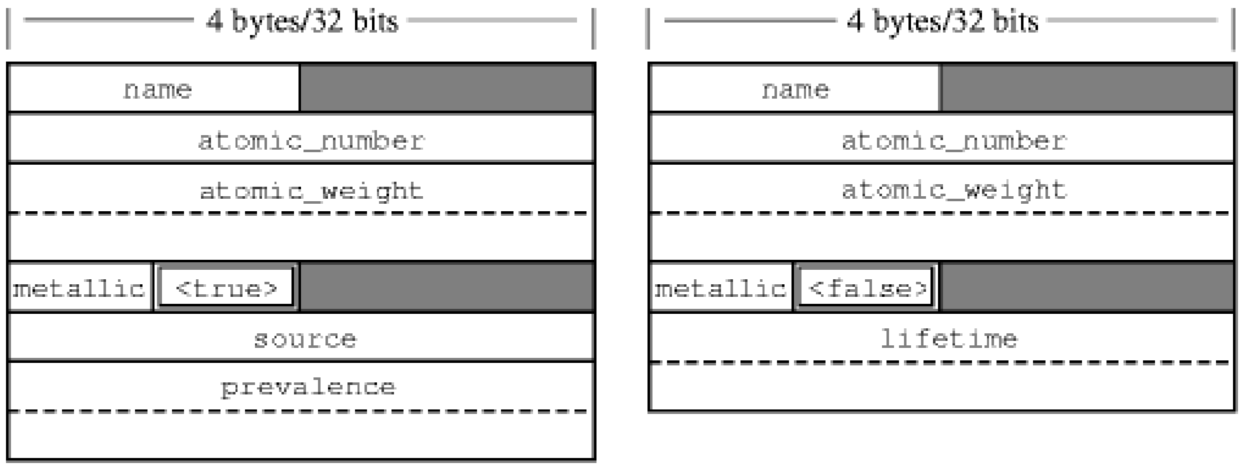
- Variant record – provides several alternative fields or collections of fields (only one is valid at any time)
- In Pascal:

```
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;
            (* percentage, by weight, of Earth's crust *)
      );
      false : (
        lifetime : real;
            (* half-life in seconds of the most stable known isotope *)
      )
end;
```

  - The field naturally_occuring is called tag or discriminant
  - Each alternative is called a variant

Variant Records

- Memory layout:



- Variants can share space
- Access to fields – similar to records:

    copper.atomic_weight

    copper.source

Variant Records

- In C – unions:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    char metallic;
    char naturally_occurring;
    union {
        struct {
            char *source;
            double prevalence;
        } natural_info;
        double lifetime;
    } extra_fields;
} copper;
```

- Access to fields:

  copper.atomic_weight

  copper.extra_fields.natural_info.source

- Historically – equivalence statements in Fortran:

  integer i

  real r

  logical b

  equivalence (i, r, b)

  – Inform the compiler that i, r, b will never be used simultaneously → can share the same memory space

Variant Records

- Safety issues:
  - lack of tag (discriminant) → you don't know what is there
  - if a tag is provided → ability to change the tag and the fields independently

Arrays

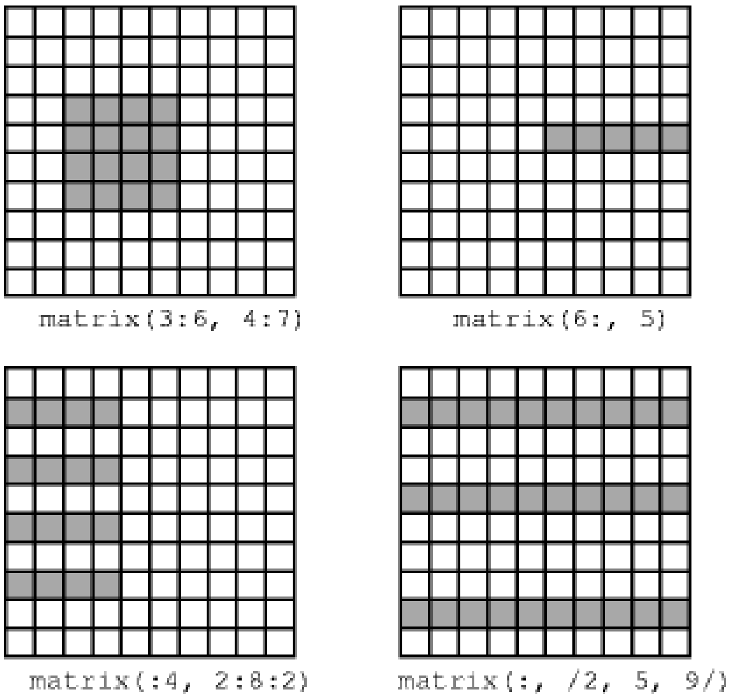
- Array
  - sequence of elements that have the same type
  - mathematically – function that maps the index type to the element type

- In most languages – the index type must be discrete
- Awk and Perl – allow non-discrete index types
  - associative arrays – implemented with hash tables

- Access to elements
  - Pascal, C: A[3]
  - Fortran, Ada: A(3)

Arrays

- Slices – rectangular portions of an array
  - Fortran 90 offers extensive facilities for slicing:



matrix(3:6, 4:7) columns 3-6, rows 4-7
matrix(6:, 5) columns 6-end, row 5
matrix(:4, 2:8:2) columns 1-4, every other row from 2-8
matrix(:, /2, 5, 9/) all columns, rows 2, 5, and 9

- Can be extracted, assigned into each other, etc, as if they were smaller arrays

Arrays

- Array operations:
- Selection of an element and assignment – all languages
- Operations performed on entire arrays (or slices) - Fortran 90 and Ada:
  - equality test
  - comparison for lexicographic ordering
  - bitwise operations
  - arithmetic operations
  - mathematical functions

Arrays

- Shape of an array – the number of dimensions and bounds (for each dimension)
  - may not be known at compile time
- Impact on allocation – several cases:

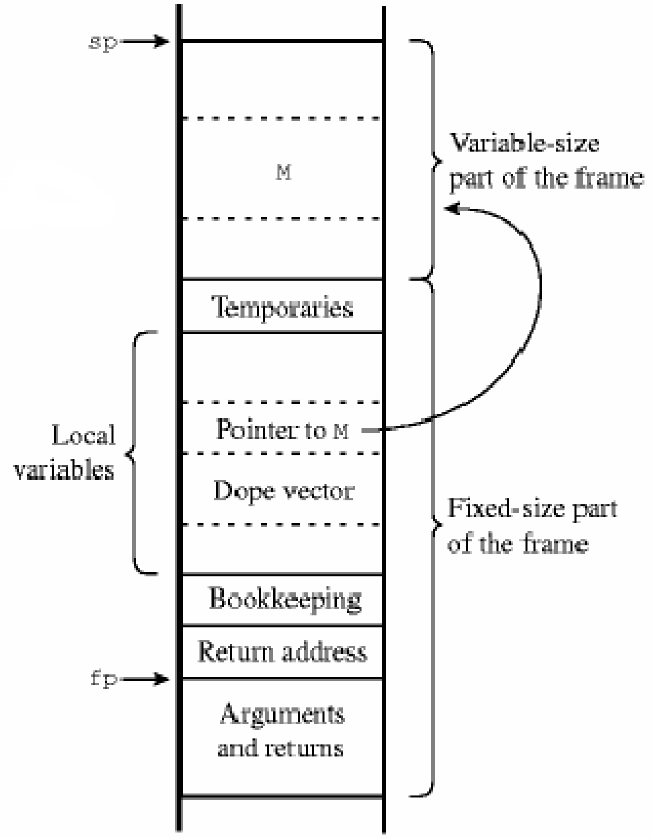
- Global lifetime, static shape
  - global variables in C, Pascal
  - allocated statically

- Local lifetime, static shape
  - local variables in C, Pascal
  - allocated on stack

Arrays

- Allocation (cont.):

```
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
...
end foo;
```



- Local lifetime, shape bound at elaboration time
  - local variables in Ada
  - allocated on stack, but with an extra level of indirection
  - the stack frame is divided into a fixed-size part and variable-size part

Arrays

- Allocation (cont.):

- Arbitrary lifetime, shape bound at elaboration time
  - Java arrays (allocated with new)
  - once allocated, shape remains the same until deallocation
  - dynamic allocation on the heap

- Arbitrary lifetime, dynamic shape
  - Perl arrays
  - shape of the array can be changed during its lifetime
  - if size is increased → allocate a larger block, copy data, deallocate the old block
  - dynamic allocation on the heap

Arrays

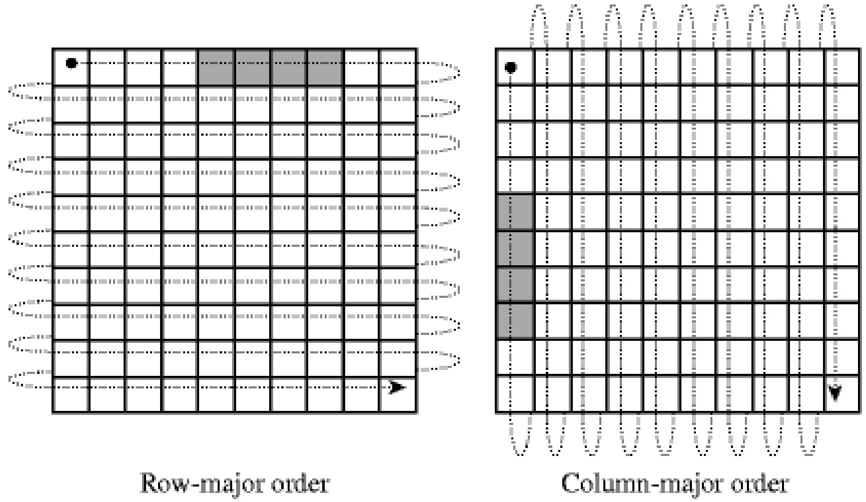
- Memory layout:

- One-dimensional arrays
  - contiguous allocation

- Multi-dimensional arrays → two strategies:
  - contiguous allocation
  - row pointers

Arrays

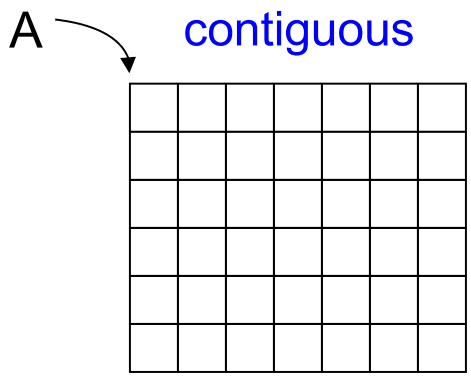
- Multi-dimensional arrays – contiguous allocation
  - column-major order (in Fortran)
  - row-major order (all other languages)

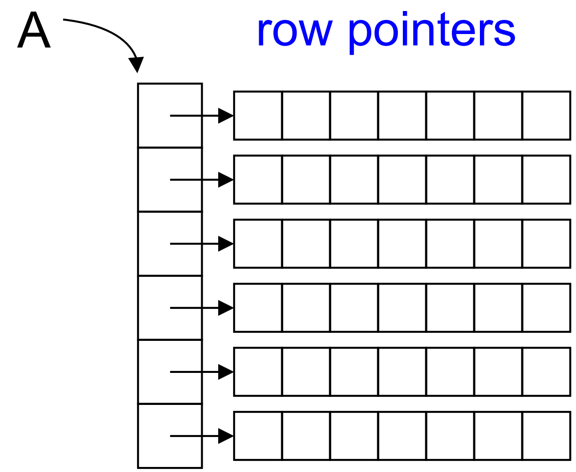


  - Row-major order – makes array [a..b, c..d] equivalent to array [a..b] of array [c..d]
  - Efficiency – important to traverse arrays along consecutive elements (along rows in C) to maximize cache hits

Arrays

- Multi-dimensional arrays – row pointers





  - Disadvantage
    - requires extra space for pointers
  - Advantages
    - allows rows to be allocated anywhere
    - faster access, avoids multiplication - good for 1970s machines with slow multiplication instructions
    - can accommodate rows of different lengths (array of strings)

Arrays

- Address calculations – contiguous allocation:

  A : array [L1..U1] of array [L2..U2] of elem_type;

  - Let:

    S2 = size of elem_type

    S1 = (U2-L2+1) * S2

  - The address of A[i,j] is:

    address of A + (i - L1) * S1 + (j - L2) * S2

  - Rewrite as:

    (i * S1) + (j * S2) + address of A
    - [(L1 * S1) + (L2 * S2)]

  - When bounds are known at compile time, part of the calculation can be done in advance
    - the part between [ ] can be computed at compile time

Arrays

- Address calculations – row pointer allocation:

```
-- assume i is in r1, j is in r2
r4 := &A
r4 := *r4[r1]                -- load
r5 := *r4[r2]                -- load
```

  - No multiplication, but more loads from memory

Announcements

- Readings
  - Chapters 7, 8

- Homework
  - HW 5 out – due on April 11
  - Submission
    - at the beginning of class
    - with a title page: Name, Class, Assignment #, Date
    - preferably typed

Strings

- Strings - in general, implemented as arrays of characters

- Special case → easier to implement additional operations
  - always one-dimensional
  - one-byte elements
  - never contain references

- Operations
  - Assignment
  - Comparison (=, >, etc)
  - Concatenation
  - Substring reference
  - Pattern matching

Strings

- Length – actually two issues:
  - how much memory space is allocated for a string variable?
  - what is the length of the actual string?

- Length descriptor
  - C – null character after the actual string
  - Pascal – the first character (at index 0) stores the length of the string; the string begins at index 1
    - advantage – easier to get the length (no need to search)
    - disadvantage – cannot have strings longer than 255

Sets

- Sets – introduced by Pascal

```
var A, B, C : set of char;
    D, E : set of weekday;
...
A := B + C;      (* union; A := {x | x is in B or x is in C} *)
A := B * C;      (* intersection; A := {x | x is in B and x is in C} *)
A := B - C;      (* difference; A := {x | x is in B and x is not in C} *)
```

- Other operations: equality test (=), superset test (>=), subset test (<=), membership test (in)
- Type of the set elements - base (universe) type

Sets

- Implementation:
  - Bit vector – represent the set:

(monday wednesday friday)

0	1	0	1	0	1	0
s	m	t	w	t	f	s

- length in bits = number of possible values of the base type
- k^{th} bit is 1 → the k^{th} value is present in the set
- k^{th} bit is 0 → the k^{th} value is not present in the set
- union → bit-wise or
- intersection → bit-wise and
- difference (A - B) → A & !B
- a set of characters – need a bit vector with 256 bits = 32 bytes
- problem – how do you represent a set of integers?
  - will need about 500 MB
  - Pascal - sets are available only for limited base types (256 values)
  - alternatives - arrays, linked lists

Pointers and Recursive Types

- Pointers – serve several purposes:
  - efficient and intuitive access to objects
  - creation of recursive data structures (linked lists)
  - dynamic storage management
- Note – in general, pointers are NOT the same thing as addresses
  - pointers – high-level concept (abstraction)
  - addresses – low-level concept (implementation)
  - examples:
    - segmented memory: segment ID and offset within segment
    - catch dangling references: address and access key

Pointers and Recursive Types

- Design issues:
  - Are pointers restricted to pointing at particular types?
  - Are pointers used for dynamic storage management, indirect addressing, or both?

Pascal - pointers can only refer to objects in the heap:

```
var p, r : ^integer;
new (p);                (* allocate from heap *)
r := p;                 (* point to the same object in the heap *)
```

C - pointers can also refer to non-heap objects ("address of" operator):

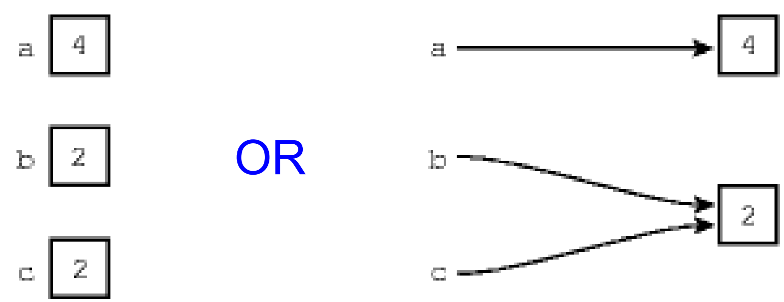
```
int x = 5;
int *p;
p = &x;          /* points to object x (may be statically */
/* allocated, or allocated on the stack) */
```

Syntax and Operations

- Operations
  - allocation
  - deallocation
  - assignment
  - "address of"
  - dereferencing
- Depend on the variable model:
- Consider the code:

```
b := 2;
c := b;
a := b + c;
```

- Implementation:



Value model   Reference model

Syntax and Operations

- Value model

  A = B; // put the value of B into A

  A = B; // if A and B are (explicit) pointers →

  // → make A refer to the object referred by B

- Reference model

  A = B; // make A refer to the object referred by B

  A = 3; // should we really make a pointer to 3?

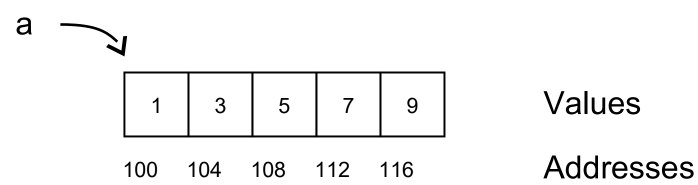
  - Implementation in Clu, Smalltalk, Java:
    - Immutable objects (integers, floats, characters) – keep the actual value (same as in value model)
    - Mutable objects (everything else) – keep a reference to the object

Pointer-Array Duality in C

- One-dimensional arrays:

```
int *a;
int b[5] = {1, 3, 5, 7, 9};

a = b;
```



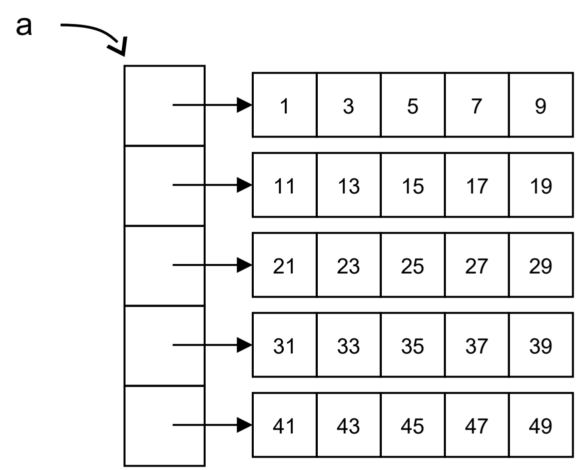
a	=>	100
a[0]	=>	1
&a[0]	=>	100
a+2	=>	108
*(a+2)	=>	5
(a+2)[1]	=>	7

Pointer-Array Duality in C

- Multi-dimensional arrays (row pointer allocation):

```
int **a;
int rows = 5, cols = 5;

a = (int**) malloc ( rows * sizeof(int*) );
for ( i = 0 ; i < rows ; i++ )
    a[i] = (int*) malloc ( cols * sizeof(int) );
```




```
(*(a+3))[2]   =>   35
*(a[3]+2)     =>   35
*(*(a+3)+2)   =>   35
a[3][2]       =>   35
```

- To compute a[i][j] – need only i and j

Pointer-Array Duality in C

- Multi-dimensional arrays (contiguous allocation as one-dimensional arrays):

```
int *a;
int rows = 3, cols = 4;

a = (int*) malloc ( rows * cols * sizeof(int) );
```

a

1	3	5	7	9	11	13	15	17	19	21	23

- Compute the equivalent of a[2][1]:

a[2*cols+1] => 19

- To compute a[i][j] – need i, j and cols

Pointer-Array Duality in C

- Multi-dimensional arrays (contiguous multi-dimensional allocation):

int a[3][4];

a

1	3	5	7
9	11	13	15
17	19	21	23

<=>

a

1	3	5	7	9	11	13	15	17	19	21	23

a[2][1] => 19

- Memory layout is actually identical to the previous case
- To compute a[i][j] – programmer needs only i and j, but compiler also needs cols

Pointer-Array Duality in C

- Passing parameters - always a pointer is passed

- One-dimensional array:

```
int a[4];

void f (int *s)
{
    s[i] = ...
}
```

- Multi-dimensional array (row pointer allocation):

```
int **a;

void f (int **s)
{
    s[i][j] = ...
}
```

Pointer-Array Duality in C

- Passing parameters (cont.)

- Multi-dimensional array (contiguous multi-dimensional allocation):

```
int a[3][4];

void f (int **s)          // wrong - compiler must know the number of columns

void f (int s[][4])       // correct
{
   s[i][j] = ...
}
```

Pointer-Array Duality in C

- Passing parameters (cont.)

- Multi-dimensional array (allocated as one-dimensional):

```
int *a;


void f (int *s)          //correct syntactically, but unusable inside f

void f (int *s, int rows, int cols)          // at least cols must be given
{
   s[i*cols+j] = ...
}
```

Pointer-Array Duality in C

- Iteration over an array:
- Using an index:

```
int a[5];
int i;

for (i = 0 ; i < 5 ; i++)
   a[i] = ...
```

- Using a pointer:

```
int a[5];
int *p;

for (p = a ; p < a+5 ; p++)
   *p = ...                 equivalent to p[0] ≠/ ...
```

Pointer-Array Duality in C

- The sizeof operator:

```
int a[5];
int *p;
p = (int *) malloc (5 * sizeof(int))
int **s;
s = (int **) malloc ... // allocate a 5x5 matrix of integers
```

```
sizeof (int)     =>   4
sizeof (a)       =>   20
sizeof (p)       =>   4   (address encoded on 4 bytes)
sizeof (s)       =>   4   (address encoded on 4 bytes)
```

- sizeof does not generate any code - is evaluated at compile time

Announcements

- Readings
  - Chapters 7, 8

C Declarations

- C declaration rule – start at the variable name, read right as far as you can (subject to parentheses), then left, then go out a level of parentheses and repeat

- Examples:

int *a[n];	array of n pointers to integers
int (*a)[n];	pointer to array of n integers
int (*a) (int *);	pointer to function taking pointer to integer as argument, and returning integer

Dangling References

- How are objects deallocated?
- Storage classes:
  - static - never deallocated, same lifetime as the program
  - stack - deallocated automatically on subroutine return
  - heap - explicit or implicit deallocation
- Explicit deallocation in Pascal:

  dispose (my_ptr);
- In C:

  free (my_ptr);
- In C++:

  delete my_ptr; //before deallocation, also calls the destructor for the object
- Implicit deallocation
  - garbage collection

Dangling References

- Dangling reference - a pointer that no longer points to a live object

- Produced in the context of heap allocation:

```
p = new int ;
r = p ;
delete r ;
*p = 3;                 // crash!!
```

- Produced in the context of stack allocation (not in Pascal - has only pointers to heap objects):

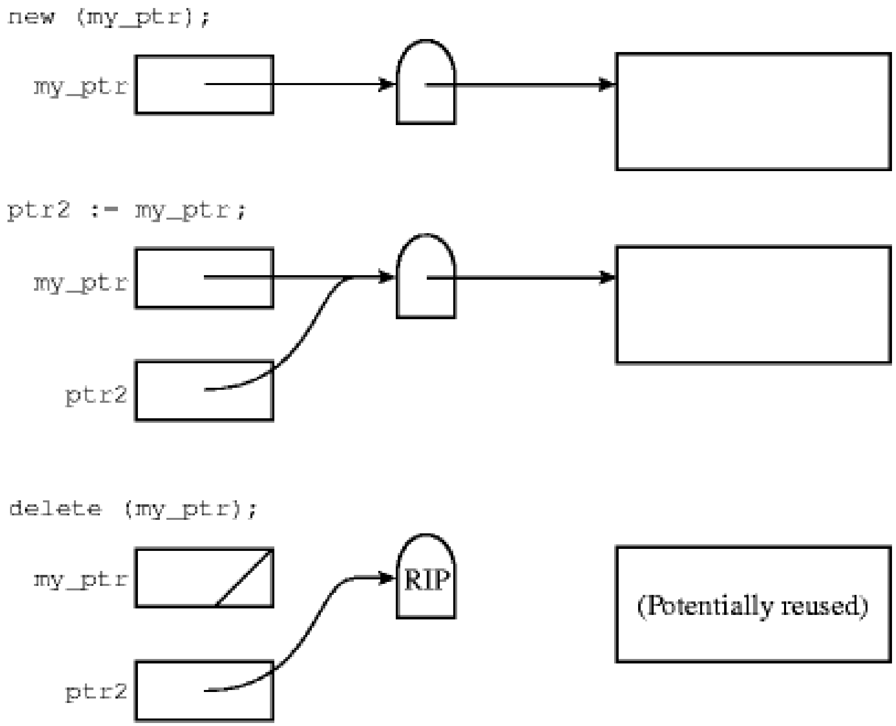
```
int* f ()
{
    int x;
    return &x;
}
```

```
void main ()
{
    int *p;
    p = f();
    *p = 3;             // crash!!
}
```

Dangling References

- Dangling references can only be produced in languages with explicit deallocation. Why?
  - Programmer may deallocate an object that is still referred by live pointers
  - Garbage collection will check if an object still has references to it before deleting it
- Why do we need to detect dangling references?
  - Need to warn the programmer - generate an error, not silently retrieve some garbage
- Mechanisms to detect dangling references
  - Tombstones
  - Locks and keys

Tombstones



- For each object allocated dynamically - also allocate a tombstone

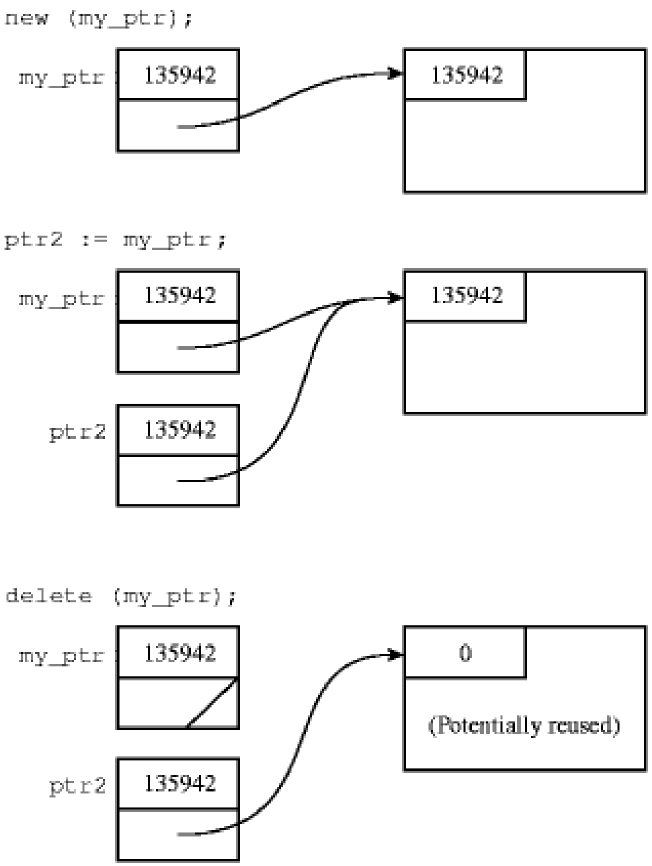
- Extra level of indirection
  - the pointer points to the tombstone
  - the tombstone points to the object
- Deallocate an object - put a special value in the tombstone

Tombstones

- Properties
  - catch all dangling references
  - handle both heap and stack objects
  - make storage compaction easier – why?

    when moving blocks, need to change only addresses in tombstones, not in the pointers
- Time complexity - overhead:
  - creation of tombstones when allocating objects
  - checking validity on every access
  - double indirection
- Space complexity - need extra space for tombstones
  - when are they deallocated?
  - two approaches:
    - never deallocate them
    - add a reference count to each tombstone - deallocate when count is zero

Locks and Keys



- Allocation - generate a number, store it in the object (lock) and in the pointer (key)
- Access - check if the key matches the lock
- Deallocation - put a special value in the lock

Locks and Keys

- Properties
  - do not guarantee to catch all dangling references - why?
    - a reused block may get the same lock number - however, it is a low probability
  - used to handle only heap objects – why?
    - to catch stack objects - would need to have locks on every local variable and argument
- Time complexity - overhead:
  - comparing locks and keys on every access
  - however, no double indirection
- Space complexity
  - extra space for locks in every heap object
  - extra space for keys in every pointer
  - however, no additional entities (tombstones) to be deallocated

Garbage Collection

- Explicit deallocation of heap objects
  - Advantages: implementation simplicity, speed
  - Disadvantages: burden on programmer, may produce garbage (memory leaks) or dangling references
- Automatic deallocation of heap objects (garbage collection)
  - Advantages: convenience for programmer, safety (no memory leaks or dangling references)
  - Disadvantages: complex implementation, run-time overhead
- Garbage collection
  - essential for functional languages - frequent construction and return of objects from functions
  - increasingly used in imperative languages (Clu, Ada, Java)
  - mechanisms:
    - reference counts
    - mark-and-sweep

Reference Counts

- How do we know when a heap object is no longer useful?
  - when there are no pointers to it

- Solution
  - in each object keep a reference counter = the number of pointers referring the object
  - when allocating the object:

    p = new int; // refcnt of new object ← 1
  - when assigning pointers:

    p = r; // refcnt of object referred by p --

    // refcnt of object referred by r ++
  - when the reference count is zero → can destroy it

Reference Counts

- Problems:
  - Objects that contain pointers:

```
p = new chr_tree;
...
p = r;
```

if the//object referred by p can be deleted (refcnt = 0), need to recursively follow pointers within it to decrement refcnt for those objects, and delete them as well if their refcnt is zero

  - Pointers declared as local variables:

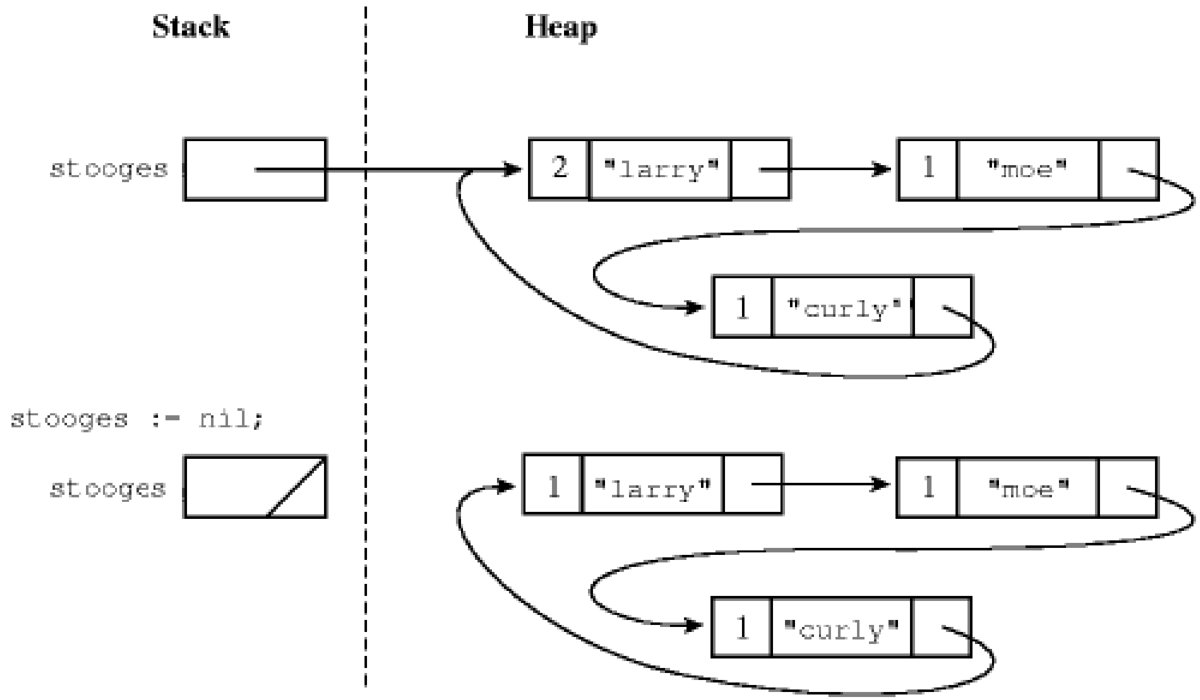
```
void f ( int * x )
{
    int i;
    int * p = x;        // refcnt of object referred by x ++
    return;
}
```

all loca//l variables will die here - must find all those which are pointers (like p) and decrement refcnts

  - Solution - use type descriptors
    - at the beginning of each record - specify which fields are pointers
    - in each stack frame - specify which local variables are pointers

Reference Counts

- Problem - circular lists:



- The objects in the list are not reachable, but their reference counts are non-zero

Mark-and-Sweep Collection

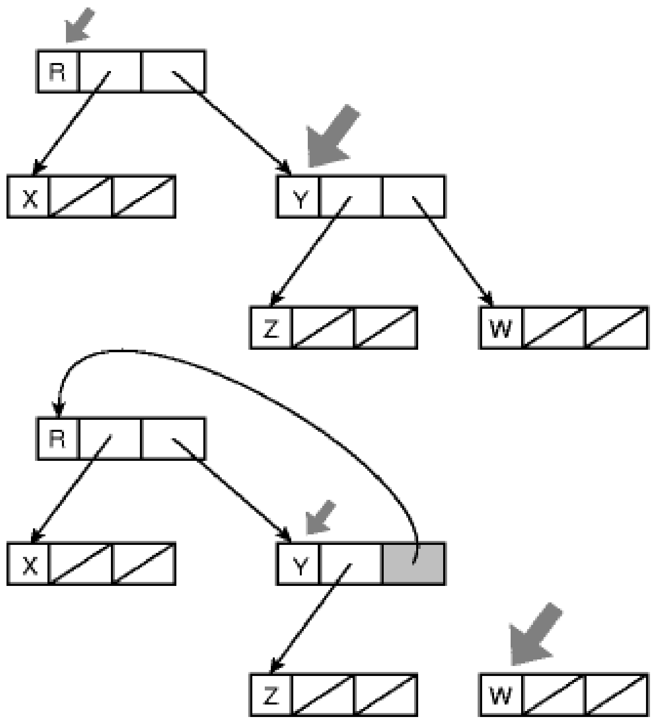
- When the free space becomes low:
  1. walk through the heap, and mark each block (tentatively) as "useless"
  2. recursively explore all pointers in the program, and mark each encountered block as "useful"
  3. walk again through the heap, and delete every "useless" block (could not be reached)

- To find all pointers - use type descriptors

- To explore recursively - need a stack (to be able to return)
  - maximum length of stack = the longest chain of pointers
  - problem - maybe not enough space for a stack (the free space is already low)
  - solution - exploration via pointer reversal

Mark-and-Sweep Collection

- Exploration via pointer reversal:



- Before moving from current to next block, reverse the pointer that is followed, to refer back to previous block
- When returning, restore the pointer
- During exploration, the currently explored path will have all pointers reversed, to trace the way back

Storage Compaction

- Storage compaction - reduce external fragmentation
  - elegant solution – stop-and-copy
- Stop-and-copy
  - achieve compaction and garbage collection and simultaneously eliminate steps 1 and 3 from mark-and-sweep
  - divide the heap in two halves
  - all allocations happen in first half
  - when memory is low
    - recursively explore all pointers in the program, move each encountered block to the second half, and change the pointer to it accordingly
    - swap the notion of first and second half

Garbage Collection

- Stop-and-copy – advantage over standard mark-and-sweep
  - no more walks ("sweeps") through the entire heap
  - overhead proportional to the number of used blocks, instead of the total number of blocks

- Significant difference between reference counts strategies and mark-and-sweep strategies
  - reference counts - when an object is no longer needed, it can be immediately reclaimed
  - mark-and-sweep - "stop-the-world" effect: pause program execution to reclaim all the garbage

Announcements

- Readings
  - Chapters 7, 8