Data Types

In Text: Chapter 6

Outline

- What is a type?
- Primitives
- Strings
- Ordinals
- Arrays
- Records
- Sets
- Pointers

Data Types

- Two components:
  - Set of objects in the type (domain of values)
  - Set of applicable operations
- May be determined:
  - Statically (at compile time)
  - Dynamically (at run time)
- A language's data types may be:
  - Built-in
  - Programmer-defined
- A **declaration** explicitly associates an identifier with a type (and thus representation)

Design Issues for All Data Types

- How is the domain of values specified?
- What operations are defined and how are they specified?
- What is the syntax of references to variables?

Primitive Data Types

- A **primitive** type is one that is not defined in terms of other data types
- Typical primitives include:
  - Boolean
  - Character
  - Integral type(s)
  - Fixed point type(s)
  - Floating point type(s)

Boolean

- Used for logical decisions/conditions
- Could be implemented as a bit, but usually as a byte
- Advantage: readability

Integer

- Almost always an exact reflection of the hardware, so the mapping is trivial
- There may be many different integer types in a language
  - Short, Int, Long, Byte
- Each such integer type usually maps to a different representation supported by the machine

Fixed Point (Decimal) Types

- Originated with business applications (money)
- Store a fixed number of decimal digits (coded)
- Advantage: accuracy
  - 0.1
- Disadvantages: limited range, wastes memory

Floating Point Types

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types; sometimes more
  - Float, Double
- Usually exactly like the hardware, but not always; some languages allow accuracy specs in code (e.g., Ada):

```
type Speed is
    digits 7 range 0.0..1000.0;
type Voltage is
    delta 0.1 range -12.0..24.0;
```

- See book for representation (p. 237)
  - Sign, Exponent, Fraction

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Is the length of objects fixed or variable?
- Operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Concatenation
  - Substring reference
  - Pattern matching

Examples of String Support

- Pascal
  - Not primitive; assignment and comparison only (of packed arrays)
- Ada, FORTRAN 77, FORTRAN 90 and BASIC
  - Somewhat primitive
  - Assignment, comparison, concatenation, substring reference
  - FORTRAN has an intrinsic for pattern matching
  - Examples (in Ada)
    - N := N1 & N2 (catenation)
    - N(2..4) (substring reference)
- C (and C++ for some people)
  - Not primitive
  - Use char arrays and a library of functions that provide operations

Other String Examples

- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Perl
  - Patterns are defined in terms of regular expressions
  - A very powerful facility!
  - /[A-Za-z][A-Za-z₩d]*/
- Java and C++ (with std library)
  - String class (not array of char)

String Length Options

- **Fixed (static) length** (fixed size determined at allocation)
  - FORTRAN 77, Ada, COBOL
  - A FORTRAN 90 example:

```
CHARACTER (LEN = 15) NAME;
```

- **Limited dynamic** length (fixed maximum size at allocation, but actual contents may be less)
  - C and C++ char arrays: actual length is indicated by a null character
- **Dynamic** length (may grow and shrink after allocation)
  - SNOBOL4, Perl

Evaluation of String Types

- Supporting strings is an aid to readability and writability
- As a primitive type with fixed length, they are inexpensive to provide—why not have them?
- Dynamic length is nice, but is it worth the expense?
- Implementation:
  - **Static length** — compile-time descriptor
  - **Limited dynamic length** — may need a run-time descriptor for length (but not in C and C++)
  - **Dynamic length** — need run-time descriptor; allocation/deallocation is the biggest implementation problem

User-Defined Ordinal Types

- An **ordinal type** is one in which the range of possible values can be easily associated with the set of **positive integers**

- Two common kinds:
  - Enumeration types
  - Subrange types

Enumeration Types

- The user enumerates all of the possible values, which are symbolic constants
- Design Issue: Should a symbolic constant be allowed to be in more than one type definition?
- Examples:
  - Pascal — cannot reuse constants; ETs can be used for array subscripts, for variables, case selectors; no input or output; can be compared
  - Ada — constants can be reused (overloaded literals); can be used as in Pascal; input and output supported
  - C and C++ — like Pascal, except they can be input and output as integers
  - Java does not include an enumeration type

Subrange Types

- An ordered, contiguous subsequence of another ordinal type
- Design Issue: How can they be used?
- Examples:
  - Pascal—subrange types behave as their parent types; can be used as for variables and array indices

```
type pos = 0 .. MAXINT;
```

  - Ada—subtypes are not new types, just constrained existing types (so they are compatible); can be used as in Pascal, plus case constants

```
subtype Pos_Type is
    Integer range 0 ..Integer'Last;
```

Evaluation of Ordinal Types

- Aid readability and writeability
- Improve reliability — restricted ranges add error detection abilitiy
- Implementation of user-defined ordinal types:
  - Enumeration types are implemented as integers
  - Subrange types are the parent types; code may be inserted (by the compiler) to restrict assignments to subrange variables

Arrays

- An **array** is an **aggregate of homogeneous data** elements in which an individual element is **identified by its position**
- Design Issues:
  - What types are legal for subscripts?
  - Are subscript values range checked?
  - When are subscript ranges bound?
  - When does allocation take place?
  - What is the maximum number of subscripts?
  - Can array objects be initialized?
  - Are any kind of slices allowed?

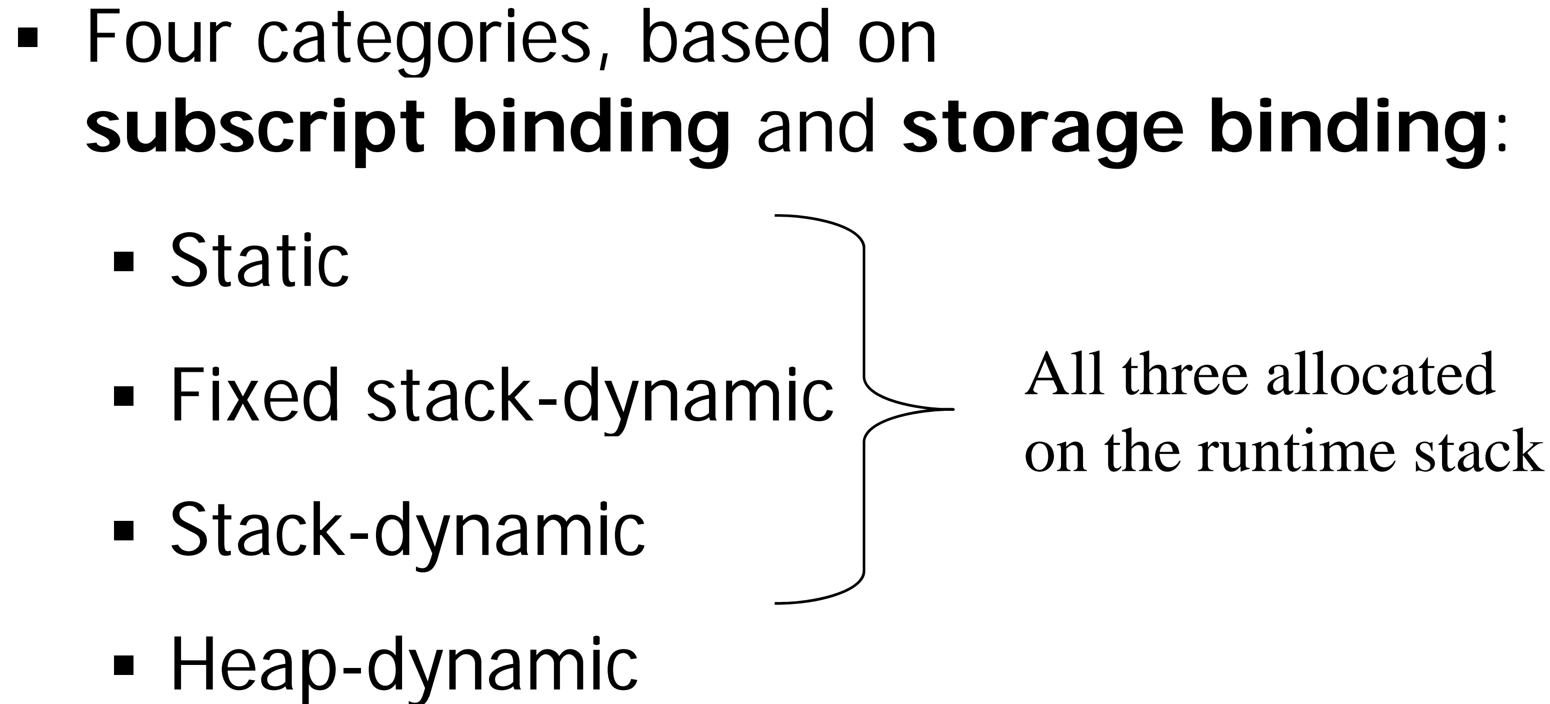
Array Indexing

- **Indexing** is a mapping from indices to elements
- Syntax
  - FORTRAN, PL/I, Ada use parentheses
  - Most others use brackets

Array Subscript Types

- What type(s) are allowed for defining array subscripts?
- FORTRAN, C — int only
- Pascal — any ordinal type
  - int, boolean, char, enum
- Ada — int or enum
  - including boolean and char
- Java — integer types only

Four Categories of Arrays



Static Arrays

- Range of subscripts and storage bindings are **static**
  - Range : compile time
  - Storage : initial program load time
- Examples: FORTRAN 77, global arrays in C++, Static arrays (C/C++), some arrays in Ada
- Advantage:
  - Execution efficiency
  - no need for explicit allocation / deallocation
- Disadvantages:
  - Size must be known at compile time
  - Bindings are fixed for entire program

Fixed Stack-Dynamic Arrays

- Range of subscripts is **statically bound**, but storage is bound at **elaboration time**
- Examples:
  - Pascal locals, C/C++ locals
  - Think: C/C++ arrays declared in callable procedures
- Advantages:
  - Space efficiency
  - Supports recursion
- Disadvantage:
  - Must know size at compile time

Stack-Dynamic Arrays

- Range and storage are dynamic, but fixed for the remainder of the time its defining procedure is active
- Examples: Ada locals in a procedure or block and having range specified by a variable
  - List : array (1..X) of integer
- Advantage:
  - Flexibility — size need not be known until the array is about to be used
- Disadvantage:
  - Once created, array size is fixed

Heap-Dynamic Arrays

- Subscript range and storage bindings are dynamic and and can **change** at any time
- Examples: FORTRAN 90, **Ada**, APL, Perl
  - List : array [*] of integer
    - List = (1, 2, 3, 4)
    - List = (1, 5, 6, 10, 19, 4 , 7, 8)
- Advantage:
  - Ultimate in flexibility
- Disadvantages:
  - More space (may be) required
  - Run-time overhead

Summary: Array Bindings

- Binding times for Array

	subscript range	storage
static:	compile time	compile time
fixed stack dynamic:	compile time	decl. elaboration time
stack dynamic:	runtime, but fixed thereafter	
dynamic:	runtime	runtime

Number of Array Subscripts

- FORTRAN I allowed up to three
- FORTRAN 77 allows up to seven
- Some languages have no limits
- Others allow just one,
  but elements themselves can be arrays

Array Initialization

- List of values put in array in the order in which the array elements are stored in memory
- Examples:
  - FORTRAN—uses the DATA statement, or
  - put the values in / ... / on the declaration
- C and C++ — put the values in braces at declaration
  - lets compiler count them (`int stuff [] = {2, 4, 6, 8};`)
- Ada — positions for the values can be specified:

```
SCORE : array (1..14, 1..2) :=
      (1 => (24, 10), 2 => (10, 7),
       3 =>(12, 30), others => (0, 0));
```

- Pascal and Modula-2 do not allow array initialization

Array Operations

- APL
  - Reverse elements in vector
  - Reverse rows or columns
  - Transpose (switch rows and colums), invert elements
  - multiply or compute vector “dot product”
- Ada
  - In assignment
    - RHS can be an aggregate, array name, or slice
    - LHS can also be a slice
  - Catenation for single-dimensioned arrays
  - Equality/inequality operators (= and /=)

Array Slices

- A slice is some **substructure** of an array
  - nothing more than a referencing mechanism

- Slice Examples:
  - FORTRAN 90

```
INTEGER MAT (1 : 4, 1 : 4)
MAT(1 : 4, 1)
```

The first column (rows 1 – 4 of column 1)

```
MAT(2, 1 : 4)
```

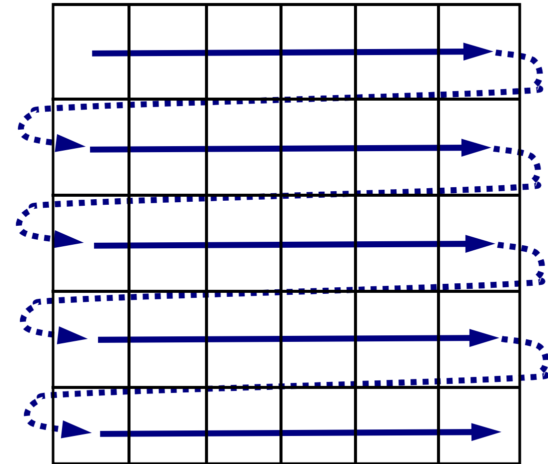
The second row (2nd row of columns 1 – 4)

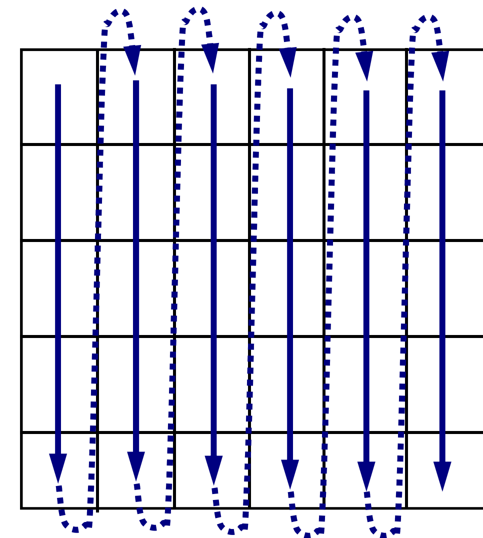
  - Ada — single-dimensioned array slice:

```
LIST(4..10)   (elements 4-10)
```

Implementation of Arrays

- Storage is LINEAR
- Row major: data elements are stored in linear memory by ROWS
  - C/C++, Ada
- Column major order: data elements are stored in linear memory by columns
  - FORTRAN





Access Functions for Arrays

- Access function maps subscript expressions to a address corresponding to intended position in array

- A: array [15] of X

  Compute location of a[k]

  (Addr of a[1]) + ((k-1) * size of (X))

Access Functions for Arrays

- A: array [8,9] of X
  - Location of a[i,j] assuming **row major** storage
  - (Addr of a[1,1]) +
    (((number of rows above i^{th} row) * (size of row)) +
    (number of elements to left of jth column)) * size of (X)
  - (Addr of a[1,1]) + (((I-1) * 9) + (j-1)) * size of (X)
- A: array [8,9] of X
  - Location of a[i,j] assuming **column major** storage
  - (Addr of a[1,1]) +
    (((number of columns to left of jth column) * (size of col)) +
    (number of elements above i^{th} row)) * size of (X)
  - (Addr of a[1,1]) + (((j-1) * 8) + (i-1)) * size of (X)

Associative Arrays

- An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called keys
  - (key, data element) pair

- Design Issues:
  - What is the form of references to elements?
  - Is the size static or dynamic?

Perl Associative Arrays
(Hashes)

- Names begin with %
- Literals are delimited by parentheses

```
%hi_temps = ("Monday"  => 77,
             "Tuesday" => 79,…);
```

- Subscripting is done using braces and keys

```
$hi_temps{"Wednesday"} = 83;
```

- Elements can be removed with delete

```
delete $hi_temps{"Tuesday"};
```

Records

- A **record** is a possibly **heterogeneous aggregate** of data elements in which the individual elements are **identified by names**

- Design Issues:
  - What is the form of references?
  - What unit operations are defined?

Record Definition Syntax

- COBOL uses level numbers to show nested records; others use recursive definitions

In C:

```
typedef struct {
    int   field1;
    float field2;
    . . .
} MyRecord;
```

In Ada:

```
type MyRecord is record
    field1 : Integer;
    field2 : Float;
    . . .
end record;
```

Record Field References

- COBOL:

```
field_name OF record_name_1 OF ...
      OF record_name_n
```

- Others (dot notation):

```
record_name_1.record_name_2. ...
      .record_name_n.field_name
```

- **Fully qualified references** must include all record names
- **Elliptical references** allow leaving out record names as long as the reference is unambiguous
- Pascal and Modula-2 provide a `with` clause to abbreviate references

Record Operations

- Assignment
  - Pascal, Ada, and C++ allow it if the types are identical
- Initialization
  - Allowed in Ada, using an aggregate
- Comparison
  - In Ada, = and /=; one operand can be an aggregate
- MOVE CORRESPONDING
  - In COBOL - it moves all fields in the source record to fields with the same names in the destination record

Unions

- A **union** is a type whose **variables** are allowed to store **different types of values** at different times during execution

- Design Issues for unions:
  - What kind of type checking, if any, must be done?
  - Should unions be integrated with records?

Union Example

- *discriminated* union
  - tag stores type of current value
- e.g., Pascal variant record

```
type rec =
    record
        case flag : bool of
            true : (x : integer;
                    y : char);
            false : (z : real)
    end

var ex : rec;
ex.flag := true;
ex.x := 5
```

Type Checking Issues

- **System must check value of flag before each variable access**

```
ex.flag := true;
ex.x := 10;
:
print(ex.z);      -- error
```

```
type rec =
    record
        case flag : bool of
            true : (x : integer;
                    y : char);
            false : (z : real)
    end
```

- **Still not good enough!**

```
ex.flag := true;
ex.x := 5;
ex.y := 'a';
ex.flag := false;
print (ex.z);    -- this should be an error, but how to check
```

- **Problem is that *USER* can set tag independently**

Free Unions

- Pascal Declaration

```
type rec = record
   case bool of
      true : . . .
      false : . . .
   end
```

- No tag variable is required
- No storage for tag, so union is inherently unsafe.
- So Pascal's union type is insecure in at least two ways.
- C/C++ have free unions (No tags)

Ada Union Types

- Similar to Pascal, except
  - No free union
    - Tag MUST be specified with union declaration
  - When tag is changed, all appropriate fields must be set too.

```
ex := (flag => false,
          z => 1.5)
```

- So Ada union types are safe.
  - Ada systems required to check the tag of all references to variants

Algol 68 Union Types

- **Declaration**

```
union (int, real) ir1, ir2
```

Can assign either type .

```
ir1 := 5;
...
ir1 := 3.4;
```

but need conformity clause to access value

```
real x;
int count;
...
count := ir1;        -- illegal

case ir1 in
   (int i) : count := i;
   (real r) : x := r;
esac
```

Union Type-Checking

- Problem with Pascal's design:
  - Type checking is ineffective
  - User can create inconsistent unions (because the tag can be individually assigned)
  - Also, the tag is optional (Free Union)!

- Ada Discriminated Union
  - Tag must be present
  - All assignments to the union must include the tag value — tag cannot be assigned by itself
  - It is impossible for the user to create an inconsistent union

Evaluation of Unions

- Useful
- Potentially unsafe in most languages
- Ada, Algol 68 provide safe versions

Pointers

- A **pointer type** is a type in which the range of values consists of memory addresses and a special value, nil (or null)

- Uses:
  - Addressing flexibility
  - Dynamic storage management

Pointer Design Issues

- What is the scope and lifetime of pointer variables?
- What is the lifetime of heap-dynamic variables?
- Are pointers restricted to pointing at a particular type?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should a language support pointer types, reference types, or both?

Pointers

- Should be able to point to only one type of object
- Dereferencing
  - explicit
  - implicit
- Used for
  - dynamic vars only
  - any kind of variable

Dangling Reference

- **Pointer to variable that has been deallocated.**

Pascal:

```
var p,q : ^cell;
begin
   new(p);
   q := p;
   dispose(p);
```

-- q is a dangling ref.

C:

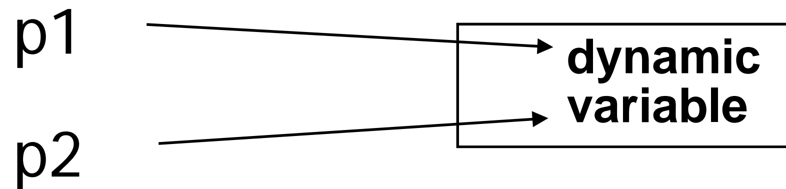
```
int *p;
int fun1(); {
   int x;
   p = &x;
    …
 }
 main ()  {
 fun1 ();
 }
```

-- *p is a dangling ref.

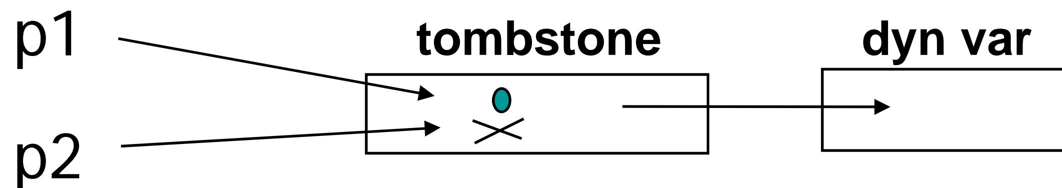
Preventing Dangling References

- **Tombstones**
  - Pointers can't point directly to a dynamic variable

without tombstone:



with tombstone: extra level of indirection called a *tombstone*.



Safe, but add space and time overhead

Preventing Dangling References

- **Locks and Keys**
  - Additional information stored with pointers and dynamic variables:
    - pointer ≡ (key, address)
    - dynamic variable ≡ (lock, var)
  - A pointer can only access a dynamic variable if its key matches the lock.
  - When a dynamic variable is deallocated, its lock is changed.
  - Again, space and time overhead.

Garbage: Memory Leaks

- An object is garbage if it is stored in an inaccessible memory address.
  - Pascal:
    - var p,q : ^cell;
    - begin
    -    new(p);
    -    new(q);
    -      ...   -- assuming no dispose or reassign
    -    p := q;
  - original p^'s storage is now garbage
  - Wasteful, but not dangerous.

Aliasing

- When two pointers refer to the same address
- Pointers need not be in the same program unit
- Changes made through one pointer affect the behavior of code referencing the other pointer
- When unintended, may cause unpredictability, loss of locality of reasoning

Pointer Examples

- Pascal: used for dynamic storage management only
  - Explicit dereferencing
  - Dangling pointers and memory leaks are possible
- Ada: a little better than Pascal
  - Implicit dereferencing
  - All pointers are initialized to null
  - Similar dangling pointer and memory leak problems for typical implementations

Pointer Examples (cont.)

- C and C++: for both dynamic storage management and addressing
  - Explicit dereferencing and address-of operator
  - Can do address arithmetic in restricted forms
  - Domain type need not be fixed (void *)
- FORTRAN 90 Pointers
  - Can point to heap and non-heap variables
  - Implicit dereferencing
  - Special assignment operator for non-dereferenced references

Evaluation of Pointers

- Dangling pointers and dangling objects are problems, as is heap management
- Pointers are like goto's—they widen the range of cells that can be accessed by a variable, but also complicate reasoning and open up new problems
- Pointers services are necessary—so we can't design a language without them

Heap Management

- *Allocation*
  - Maintain a free list of available memory cells

- *Deallocation* (Reclamation)
- method 1: Reference Counting
  - Each cell has a tag with # of pointers to that cell.
  - When reference count = 0 => deallocate cell.
  - Advantage:
    - cost is distributed over time
  - Disadvantages:
    - space/time overhead in maintaining reference counts
    - won't collect circular structures

Heap Management

- Method 2a: Garbage Collection with mark-and-sweep
  - Each cell has a mark bit.
  - Mark and Sweep:
    - set all mark bits to "garbage"
    - for each pointer into the heap, mark all reachable cells "not garbage"
    - look at each cell in memory; if marked "garbage," reclaim.
  - Advantages:
    - reclaims all garbage
    - little space/no time overhead *during normal execution*
  - Disadvantages:
    - must stop execution to garbage collect
    - fragmentation
    - time ~ memory size

Heap Management

- Method 2b: Garbage Collection with copying
  - Start with two heaps of same size
    - *working heap* + *other heap*
  - Copying:
    - allocate new cells in *working heap*
    - when *working heap* is full,
      - for each pointer into *working heap*, copy all reachable cells into *other heap*.
    - *other heap* is new *working heap*, *working heap* is new *other heap*
  - Advantages:
    - both advantages of mark & sweep (reclaims all garbage, little space/no time overhead during normal execution)
    - time ~ used cells, not total memory size
    - automatic compaction (ameliorates fragmentation)
  - Disadvantages
    - stopping to copy still a problem
    - need twice as much memory for heap => only practical with virtual memory systems

Heap Management

- This is all much easier for fixed-size allocate/deallocate than for variable-size:
  - Fixed size (& format) Lisp
    - Know where pointers are within cells
    - Fragmentation not a problem
  - Variable size (& format)
    - Need header for each object in memory telling:
      - its size
      - where it may contain pointers to other objects
    - Fragmentation is a problem--need <u>compaction</u>