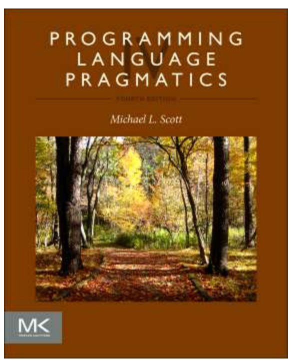
Chapter 7:: Data Types

- CSE307/526: Principles of Programming Languages
- https://ppawar.github.io/CSE307-F18/index.html

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Copyright © 2016 Elsevier & Dr. Pravin Pawar - SUNYK

Classification of Types

- What has a type? – Things that have values
  - –Constants, variables, fields, parameters, subroutines, objects
- Classification of types:
  - –Constructive: A type is built-in (e.g. Integer, Boolean) or composite (records, arrays).
  - –Denotational: A type is a set or collection of values. (e.g. enum)
  - –Abstraction-based: A type is defined by an interface, the set of operations it supports. (e.g. List)

ELSEVIER

Definition of Types

- Defining a type has two parts:
  - A type's declaration introduces its name into the current scope.
  - A type's definition describes the type (the simpler types it is composed of).

Here are some examples of declarations that are not definitions, in C:

```
extern char example1;
extern int example2;
void example3(void);
```

Here are some examples of declarations that are definitions, again in C:

```
char example1; /* Outside of a function definition it will be initialized to zero.  */
int example2 = 5;
void example3(void) { /* definition between braces */ }
```

ELSEVIER

Type System

- Mechanism for defining types and associating them with operations that can be performed on objects of each type:
  - –Built-in types with built-in operations
  - –Custom operations for built-in and custom types
- A type system includes rules that specify
  - –Type equivalence: Do two values have the same type? (Structural equivalence vs name equivalence)
  - –Type compatibility: Can a value of a certain type be used in a certain context?
  - –Type inference: How is the type of an expression computed from the types of its parts?

Common Kinds of Type Systems

- Strongly typed
  - Prohibits the application of an operation to any object not supporting this operation.
- Weakly typed
  - The type of a value depends on how it is used
- Statically typed
  - Strongly typed and type checking is performed at compile time (Pascal, C, Haskell, SML, …)
- Dynamically typed
  - Strongly typed and type checking is performed at runtime (LISP, Smalltalk, Python, …)
- In some statically typed languages (e.g., SML), the programmer does not specify types at all. They are inferred by the compiler.

ELSEVIER

Type Systems: Examples

- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically (for instance, for dynamic binding):

```
String a = 1;                //compile-time error
int i = 10.0;                //compile-time error
Student s = (Student)(new Object());// runtime
```

- Python is strong dynamic typed:

```
a = 1;
b = "2";
a + b          run-time error
```

- Perl is weak dynamic typed:

```
$a = 1
$b = "2"
$a + $b               no error.
```

ELSEVIER

Trade-offs

- There is a trade-off here:
  - Strong-static: verbose code (everything is typed), errors at compile time (cheap)
  - Strong-dynamic: less writing, errors at runtime
  - Weak-dynamic: the least code writing, potential errors at runtime, approximations in many cases

ELSEVIER

Orthogonality

- A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined
- For example:
  - Prolog is more orthogonal than ML (because it allows arrays of elements of different types, for instance)
  - Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance)
- Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

ELSEVIER

Type Checking

- A ***type system*** has rules for:
  - –*type equivalence*: when are the types of two values the same?
    - •***Structural equivalence***: two types are the same if they consist of the same components
  - –*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?

```
a : int      b : int
--------------------
     a + b : int
```

Type and Equality Testing

- What should **`a == b`** do?
  - –Are they the same object?
  - –Bitwise-identical?
- Languages can have different equality operators:
  - –Ex. Java's **`==`** vs **`equals`**

Type Casts

- Two casts: converting and non-converting
  - *Converting cast*: changes the meaning of the type in question
    - cast of **`double`** to **`int`** in Java
  - *Non-converting casts*: means to **<u>interpret</u>** the bits as the same type

```
Person p = new Student(); // implicit non-converting
Student s = (Student)p; // explicit non-converting cast
```

- *Type coercion*: May need to perform a <u>runtime semantic check</u>
  - Example: Java references:

```
Object o = "...";
String s = (String) o;
        // maybe after if(o instanceOf String)...
```

ELSEVIER

Type Coercion

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

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

- Type coercion refers to automatic, implicit conversion of one type to the expected type.

Type Coercion Examples

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

ELSEVIER

Type Checking

- Make sure you understand the difference between
  - type conversions (explicit)
  - type coercions (implicit)
  - sometimes the word 'cast' is used for conversions (C is guilty here)

Type Checking

- Two major approaches:
- structural equivalence
  - based on notion of meaning behind those declarations
- Name equivalence
  - based on declarations
  - Name equivalence is more fashionable these days

Structural equivalence

- ***Structural equivalence***: most languages agree that the **format of a declaration should not matter**:

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

is the same as the type:

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

Name equivalence

`TYPE new_type = old_type; (*Modula-2)`

`new_type` is said to be an alias for **`old_type`**.

–aliases to the same type

```
TYPE stack_element = INTEGER;       (* alias *)
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
    ...
PROCEDURE push(elem : stack_element);
    ...
PROCEDURE pop() : stack_element;
    ...
```

stack is meant to serve as an abstraction that allows the programmer, to create a stack of any desired type (in this case INTEGER).

A language in which aliased types are considered equivalent is said to have ***loose name equivalence***

Name equivalence

–there are times when aliased types should probably **<u>Not</u>** be the same:

```
TYPE celsius_temp = REAL,
     fahrenheit_temp = REAL;
VAR  c : celsius_temp,
     f : fahrenheit_temp;
f := c; (* this should probably be an error *)
```

A language in which aliased types are considered distinct is said to have ***strict name equivalence***

ELSEVIER

Classification of types

- Types can be discrete (countable/finite in implementation):
  - **`boolean`**:
    - in C, 0 or not 0
  - **integer types:**
    - different precisions (or even multiple precision)
    - different signedness
    - Why do we define required precision? Leave it up to implementer
  - **floating point numbers:**
    - only numbers with denominators that are a power of 10 can be represented precisely
  - **decimal types:**
    - allow precise representation of decimals
    - useful for money: Visual Studio .NET:

```
decimal myMoney = 300.5m;
```

ELSEVIER

Classification of types

– **character**
  - often another way of designating an 8 or 16 or 32 bit integer
  - Ascii, Unicode (UTF-16, UTF-8), BIG-5, Shift-JIS, latin-1
– **subrange numbers**
  - Subset of a type (`for i in range(1:10)`)
  - Constraint logic programming: `X in 1..100`
– **rational types:**
  - represent ratios precisely
– **complex numbers**

Classification of types

- Types can be composite :
  - **records (unions)**
  - **arrays**
    - **Strings** (most languages represent Strings like arrays)
      - list of characters: null-terminated
      - With length + get characters
  - **sets**
  - **pointers**
  - **lists**
  - **files**
  - **functions, classes**, etc.

Records

- A record consists of a number of fields:
  - Each has its own type:

```
struct MyStruct {
   boolean ok;
   int bar;
};
MyStruct foo;
```

- There is a way to access the field:

`foo.bar;` <- C, C++, Java style.

`bar of foo` <- Cobol/Algol style

`person.name` <- F-logic *path expressions*

Records

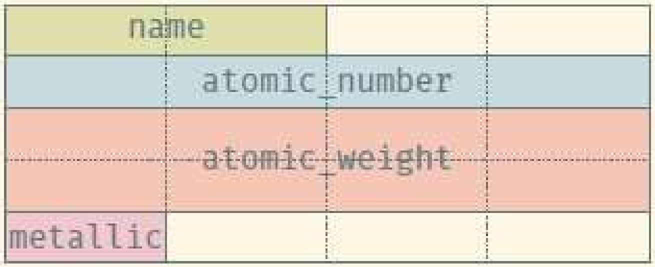
- Nested record definition in Pascal:

```
type ore = record
name : short_string;
element_yielded : record
name : two_chars;
atomic_n : integers;
atomic_weight : real;
metallic : Boolean
end
end;
```

- Accessing fields:
  - ore.element_yielded.name
  - name of element_yielded of ore

Memory Layout of Records

Aligned (fixed ordering):

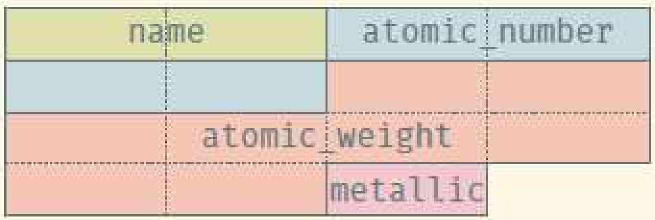


– Potential waste of space

+ One instruction per element access

+ Guaranteed layout in memory
(Good for systems programming)

Packed:

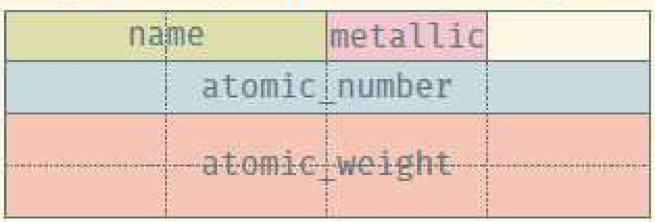


+ No waste of space

– Multiple instructions per element access

+ Guaranteed layout in memory
(Good for systems programming)

Aligned (optimized ordering):



± Reduced space overhead

+ One instruction per element access

– No guarantee of layout in memory
(Bad for systems programming)

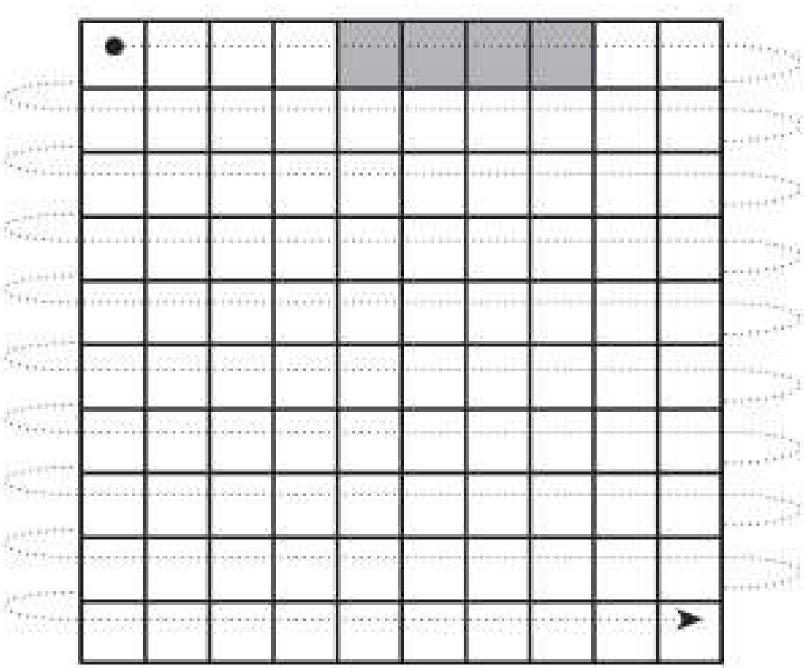
Arrays

- Arrays = areas of memory of the same type.
  - Stored consecutively.
    - Element access (read & write) = O(1)
  - Possible layouts of memory:
    - Row-major and Column-major:
      - storing multidimensional arrays in linear memory
      - Example: `int A[2][3] = { {1, 2, 3}, {4, 5, 6} };`
        - »Row-major: A is laid out contiguously in linear memory as: **`123456`**
        - »Column-major: A is laid: **`142536`**
      - Row-major order is used in C, PL/I, Python and others.
      - Column-major order is used in Fortran, MATLAB, GNU Octave, R, Rasdaman, X10 and Scilab.
    - Row pointers: Java

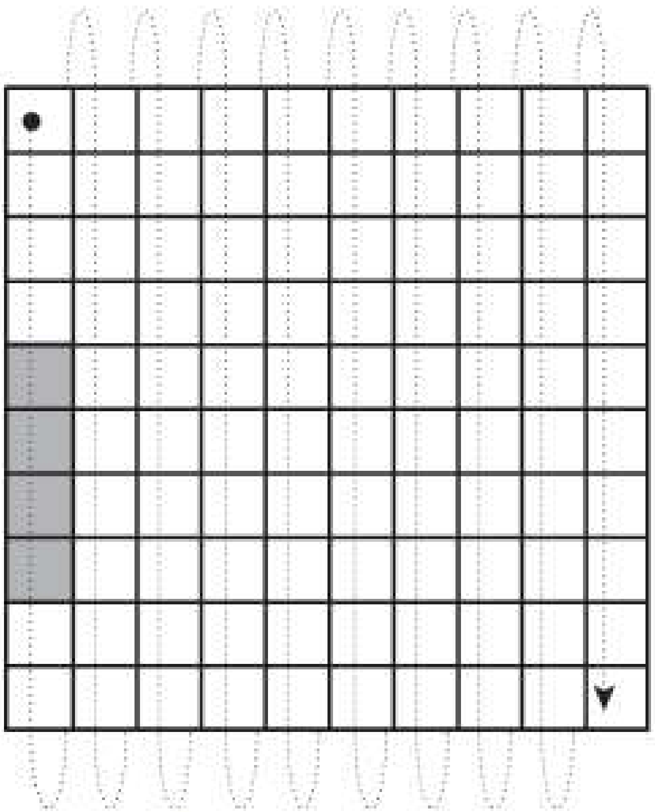
ELSEVIER

Arrays

- Row-major and Column-major:



Row-major order

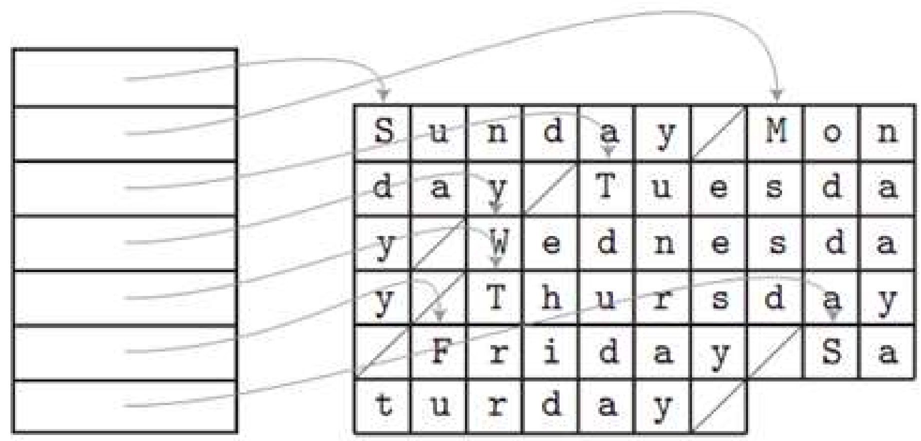


Column-major order

Arrays

- Row pointers:
  - –Allows rows to be put anywhere
  - –Good for big arrays on machines with segmentation problems
  - –requires extra space for the pointers
  - –nice for matrices whose rows are of different lengths

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



ELSEVIER

Strings

- Strings are one-dimensional structures
- In imperative languages, strings are really just arrays of characters
- In functional languages, strings are lists of characters

Sets

- Set: contains **distinct** elements without order.
  - –Pascal supports sets of any discrete type, and provides union, intersection, and difference operations:

```
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}*)
```

- Bag: Allows the same element to be contained inside it multiple times.
- Dictionary/Hashmap: Maps keys to values
- Multimap: Maps keys to set of values

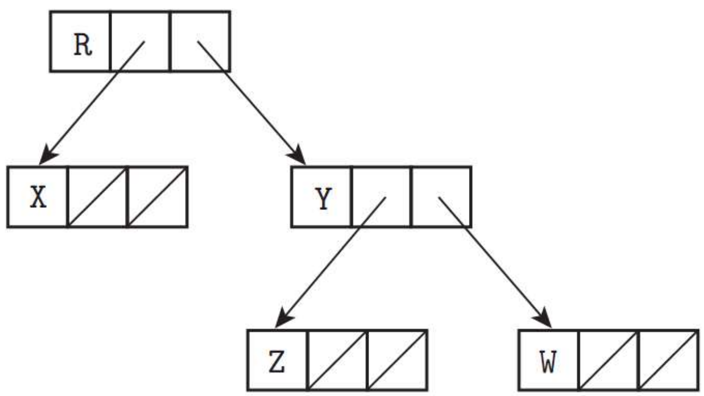
ELSEVIER

Lists

- Prolog-style Linked lists (same with SML) vs. Python-style Array lists:
  - Prolog: matching against lists
    - Head
    - Tail
  - Python lists: Array-lists are efficient for element extraction, doubling-resize

Pointers/Reference Types

- Pointers serve two purposes:
  - –efficient (and sometimes intuitive) access to elaborated objects (as in C)
  - –dynamic creation of linked data structures, in conjunction with a heap storage manager
  - –Recursive types – like trees:



  - –Several languages (e.g. Pascal, Ada 83) restrict pointers to accessing things in the heap

Pointers/Reference Types

- Pointers serve two purposes:
  - –efficient (and sometimes intuitive) access to elaborated objects (as in C)
  - –dynamic creation of linked data structures, in conjunction with a heap storage manager
  - –Recursive types – like trees:

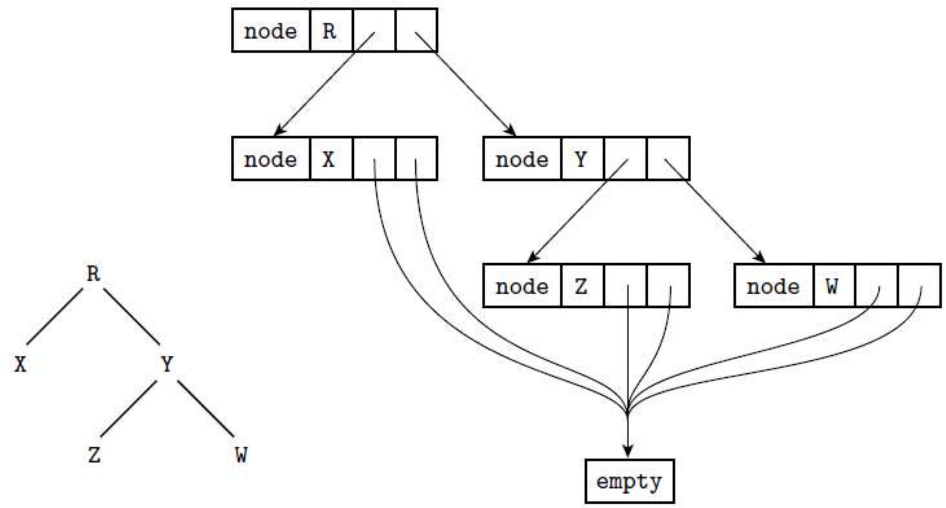


Figure 7.10 Implementation of a tree in ML. The abstract (conceptual) tree is shown at the lower left.

  - –Several languages (e.g. Pascal, Ada 83) restrict pointers to accessing things in the heap

Pointers/Reference Types

- Pointers and arrays are closely linked in C.

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

- Now all of the following are valid:

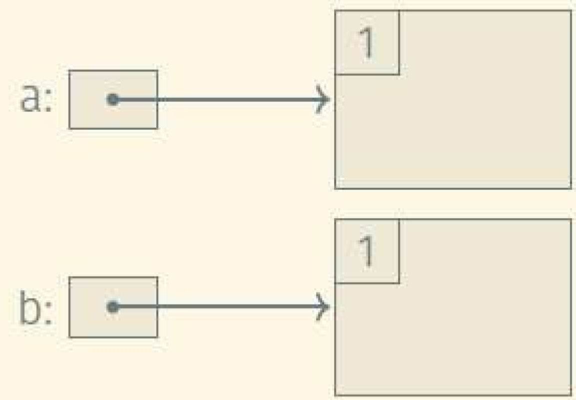
```
a = b; /* make a point to the initial element of b */
n = a[3];
n = *(a+3); /* equivalent to previous line */
n = b[3];
n = *(b+3); /* equivalent to previous line */
```

- Pointers tend to allow pointer arithmetic
  - –Only useful when in an array: Leave the bounds of your array, and you can have security holes
- In Java, references are assigned an object, and don't allow pointer arithmetic.

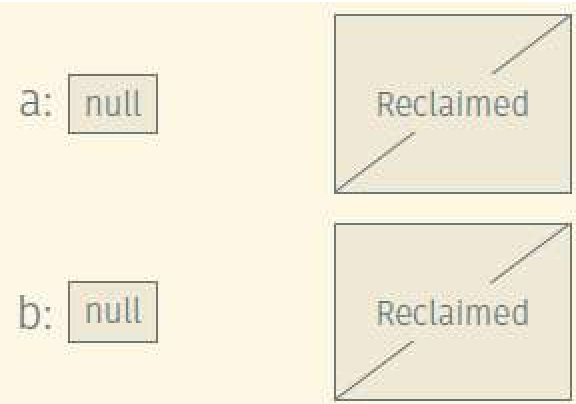
ELSEVIER

Garbage Collection – Reference counts

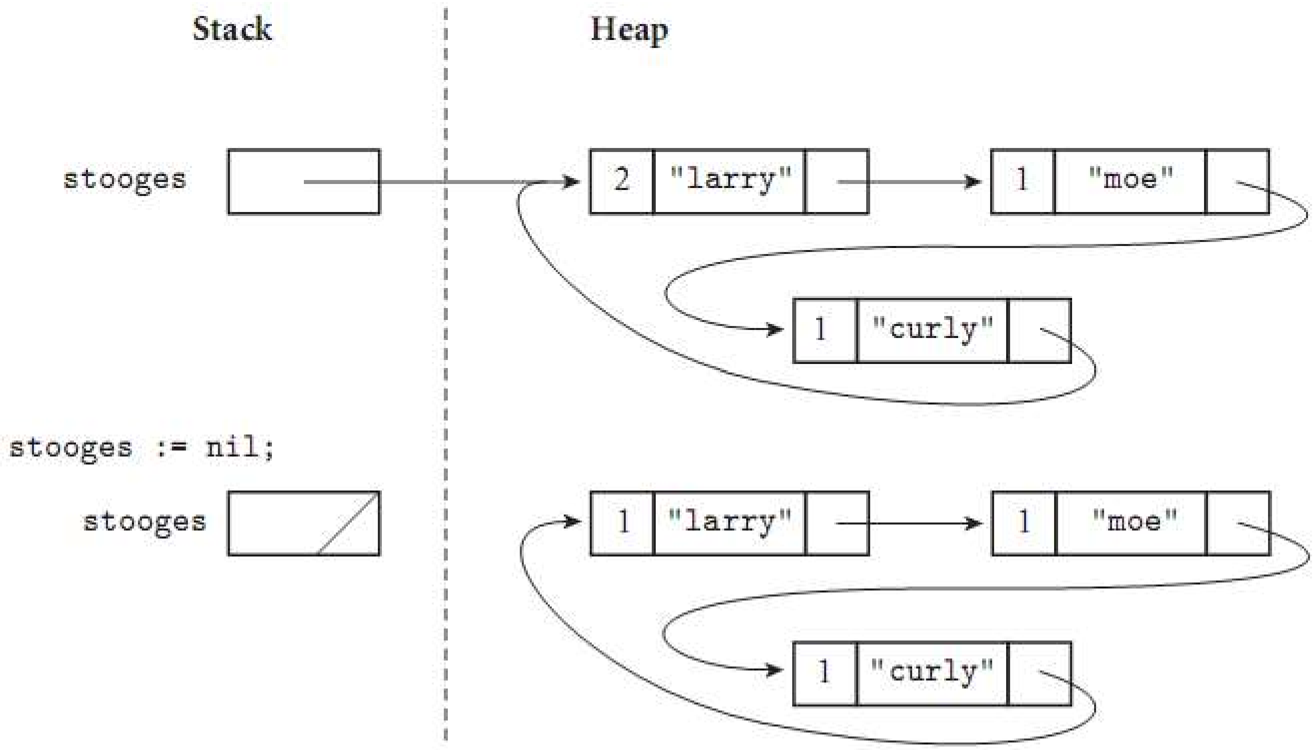
```
a = new Obj();
b = new Obj();
```




```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```



Problem with Reference counts



- The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

Garbage Collection - Mark and Sweep

- Mark every allocated memory block as useless.
- For every pointer in the static address space and on the stack, mark the block it points to as useful.
- For every block whose status changes from useless to useful, mark the blocks referenced by pointers in this block as useful. Apply this rule recursively.
- Reclaim all blocks marked as useless.

- Animation:
- https://www.youtube.com/watch?v=0CMm8GkkuzY