CSE 505

Lecture #4
Sept 10, 2012

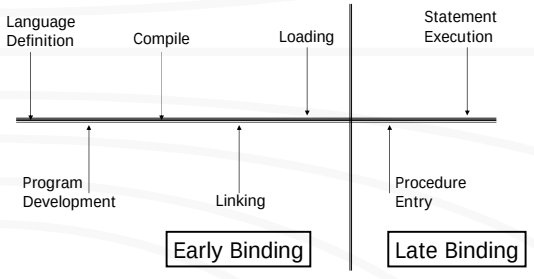
Binding Time

Definition: Binding time is the time at which an attribute is bound, or fixed, to an entity of a programming language.

Entity		Attributes
Variable	⇨	Type, Storage, Value, ...
Procedure	⇨	Body, Storage, ...

Binding Time: ..., compile-time, ..., run-time, ...

Binding Times



Binding Time	Attribute Bound *
Language Definition	Set of well-formed statements
Program Development	Set of variables
Compile	Code generated for expression
Linking	Code for library functions
Loading	Storage locations for program
Procedure Entry	Formal parameter value
Statement Execution	Value of an expression

* These examples don't hold for all languages.

Structured* Types and Data Objects

Simple Types – int, real, bool, ...

Structured Types – array, record/struct, ...

We will examine how contour models can help understand how data objects are allocated and bound to variables, and different forms of assignment and equality rules.

* - Structured type = Concrete Type

Key Terms

Variable, Object, Value, Type

Declaration: <Type> <Variable>

A variable is bound to an object, which is a container of a value of some specified type.

An object has a temporal dimension. An object can be created, shared, updated, destroyed.

Object Allocation Schemes

Allocation Scheme	When Object Allocated
Static	Loading Time
Quasi Dynamic	Procedure-Entry Time
Fully Dynamic	Stmt-Execution* Time

* - Typically done via a statement such as: **new <Type>**

Object Allocation (cont'd)

Allocation Scheme	Where Object Allocated
Static	Outermost Contour
Quasi Dynamic	Contour*
Fully Dynamic	Heap

* – This scheme is also called "Stack Allocation"

How PLs specify Binding Schemes

Fortran - Static allocation for all variables:

e.g. INTEGER A[100,100]

C - Quasi-dynamic is the default.
- Static and Fully-dynamic also permitted:

```
e.g. static int x = 1;
     struct node { int val;   struct node *left, *right;
                 };
```

Java - Fully-dynamic is default for structured types
- Static and Quasi-dynamic also permitted

Object Allocation Schemes

Allocation Scheme	When Object Allocated
Static	Loading Time
Quasi Dynamic	Procedure-Entry Time
Fully Dynamic	Stmt-Execution* Time

* - Typically done via a statement such as: **new(var)**

Object Allocation (cont'd)

Allocation Scheme	Where Object Allocated
Static	Outermost Contour
Quasi Dynamic	Contour*
Fully Dynamic	Heap

* – This scheme is also called "Stack Allocation"

Object Binding – Syntax*

T – Type, V – Variable

Object Binding Scheme	Syntax
Static	T↓ V
Quasi Dynamic	T V
Fully Dynamic	T↑ V

* – This syntax is just for the examples in Chapter 2.

Assignment & Equality

- Assignment-by-Sharing

vs.

- Assigment-by-Copying
  - Shallow Copying
  - Deep Copying

- Equality-by-Reference

vs.

- Equality-by-Value
  - Shallow Equality
  - Deep Equality

Remarks

➢ Assignment-by-Copying is not equivalent to Assigment-by-Sharing.

➢ Assignment-by-Sharing is not compatible with Quasi-Dynamic Binding.

Array Type Definitions

Quasi-dynamic:
int [100]
int [n]

Fully-dynamic:
int []

Sharing is not eqvt to Copying!

```
void main() {
   int a[5], int b[5];
   int i;
   for(i=1; i<=5 ; i++)
         a[i] := i;
   b := a;
   a[5] := 0;
   print b[5];
}
```

Outputs:

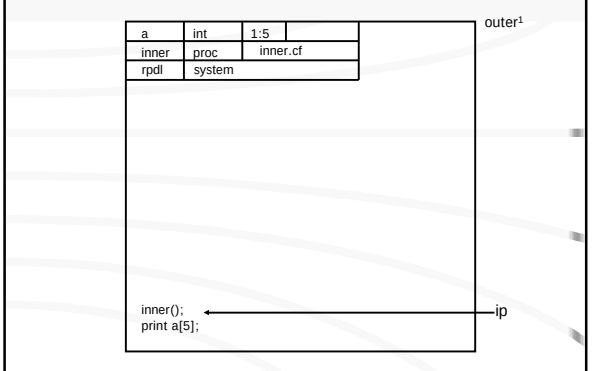
1. copying semantics:
   b[5] ⇨ 5

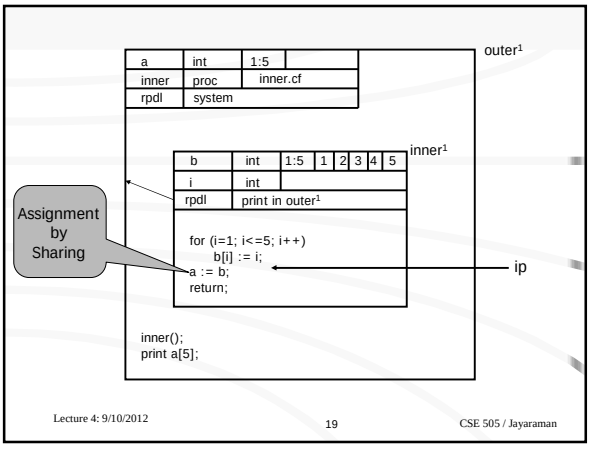
2. sharing semantics:
   b[5] ⇨ 0

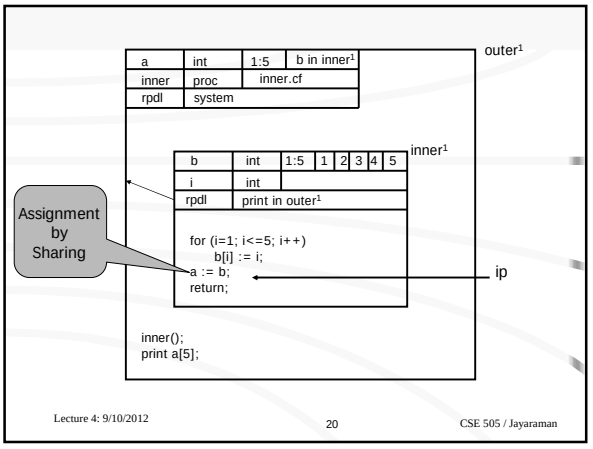
Assignment-by-Sharing is not compatible with Quasi-Dynamic Binding

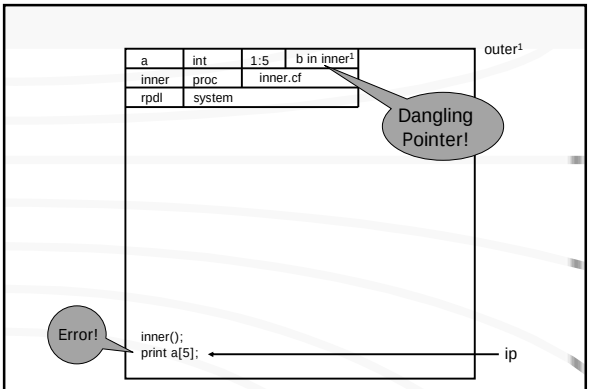
Quasi Dynamic Binding

```
void outer() {
   int a[5];
   void inner(){
      int b[5], int i;
      for(i=1; i<=5 ; i++)
          b[i] := i;
      a := b;
   }
   inner();
   print a[5];
}
```

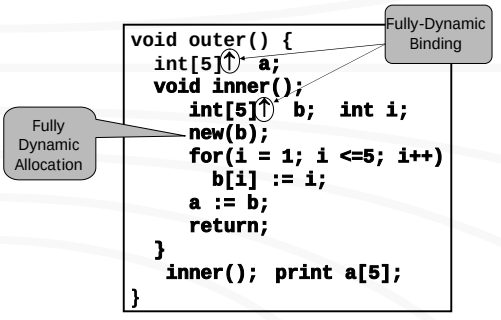



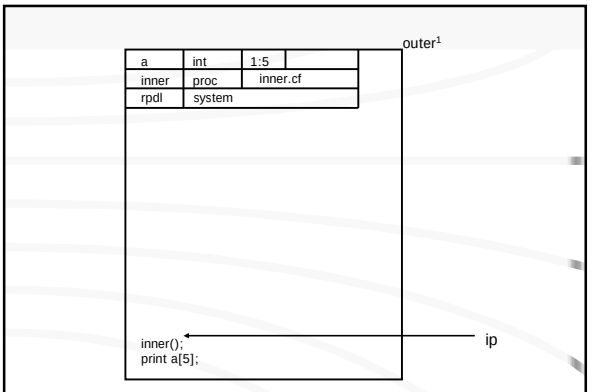


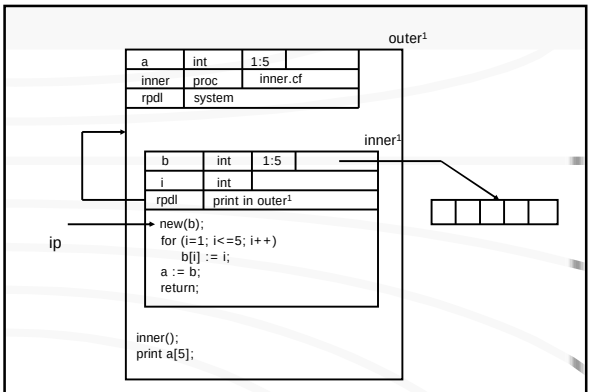


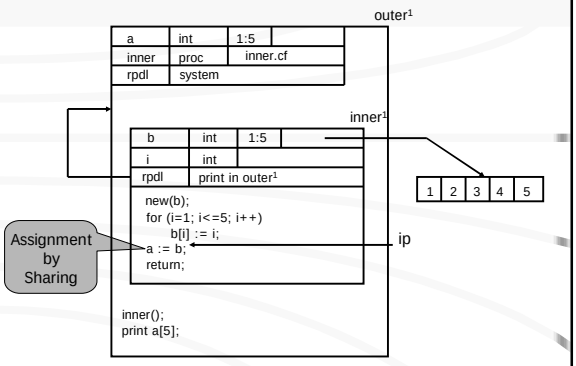


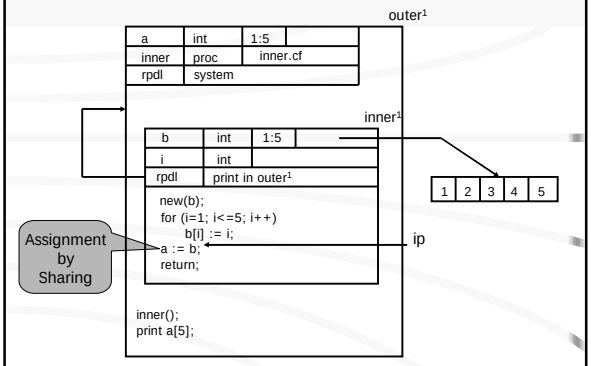
Assignment-by-Sharing needs Fully-Dynamic Binding

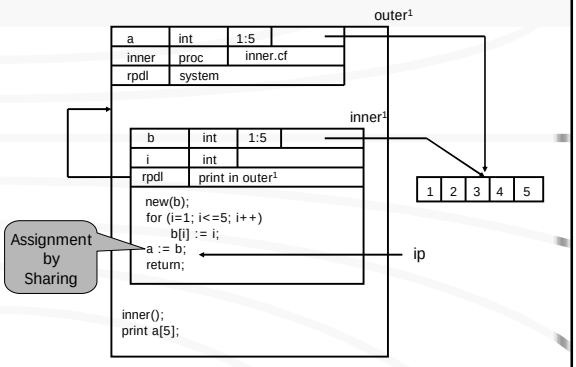


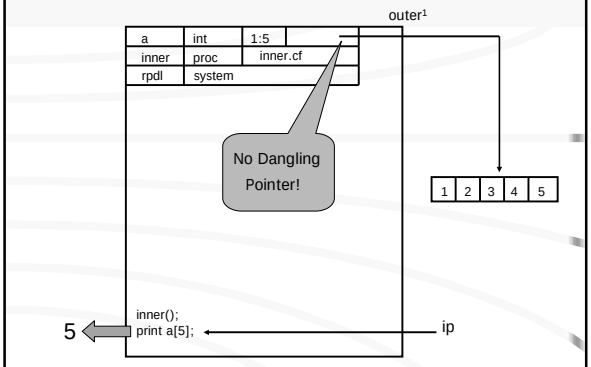










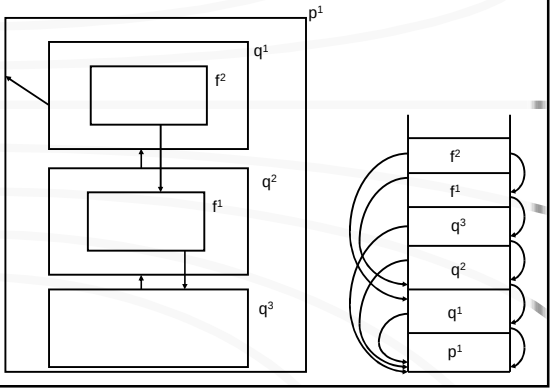


Brief Remarks about

Stacks and Heaps

- Stack Management
  - Relating contours and stack frames
  - Keeping track of static and dynamic links
- Heap Management (discussed later)
  - Reference Counting
  - Garbage Collection
  - Compaction

Mapping Contours to Stack Frames



Stack Storage Management

Each frame is called an 'activation record', which holds the local vars, parameters, together with the return ptr, static and dynamic links.

Stack Management is easy:

- push a.r. when procedure is called
- pop a.r. when procedure completes

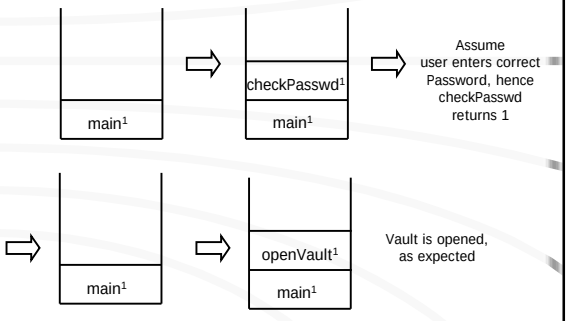
Lately, stack storage format has been exploited to create security violations.

Illustration of Buffer Overflow*

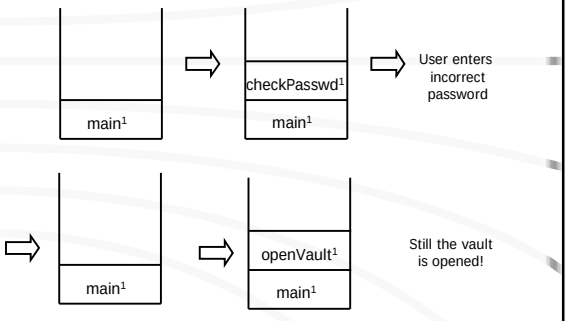
```
int checkPasswd () {
  char pass[16];
  printf ("Enter password: ");
  gets(pass);
  if (strcmp(pass, "opensesame") == 0) return 1;
  else  return 0;
}

void openVault() {
  // opens the vault
}

void main () {
  if (checkPasswd())  openVault();
}
```

* Neil Daswani's book, "Foundations of Security: What Every Programmer Should Know"

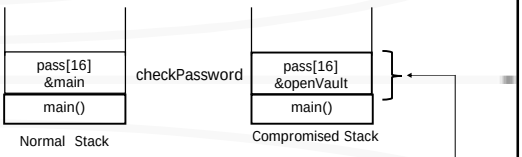
Normal Behavior



Abormal Behavior



Buffer Overflow




```
int checkPasswd () {
  char pass[16];
  printf ("Enter password: ");
  gets(pass);
  if (strcmp(pass, "opensesame") == 0)
        return 1;
  else  return 0;
}
```

A character string consisting of 16 random chars + address of the openVault procedure is entered. C does not check that a larger string is entered, and the return address to main is overwritten.

Assignment & Equality

- Assignment-by-Sharing

vs.

- Assigment-by-Copying
  - Shallow Copying
  - Deep Copying

- Equality-by-Reference

vs.

- Equality-by-Value
  - Shallow Equality
  - Deep Equality

Remarks

- Assignment-by-Copying is not equivalent to Assigment-by-Sharing.
- Assignment-by-Sharing is not compatible with Quasi-Dynamic Binding.

Pascal, C, Java syntax

```
type tree = ↑ node
type node = record
                    int value;
                    tree left;
                    tree right;
                end;
```

Pascal

```
struct node {
    int value;
    struct node* left;
    struct node* right;
}
```

C

```
class Tree {
        int value:
        Tree left;
        Tree right;
        }
```

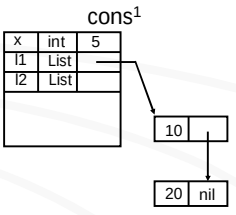
Java

List Constructor - cons

```
class List { int val;   List next; }
```

```
List cons(int x, List l1) {
        List l2;
        new(l2);
        l2.val := x;
        l2.next := l1;
        return l2;
        }
```

ip

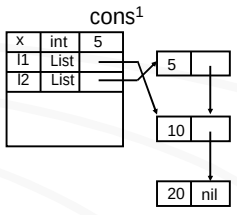


List Constructor - cons

```
class List { int val;   List next; }
```

```
List cons(int x, List l1) {
        List l2;
        new(l2);
        l2.val := x;
        l2.next := l1;
        return l2;
        }
```

ip

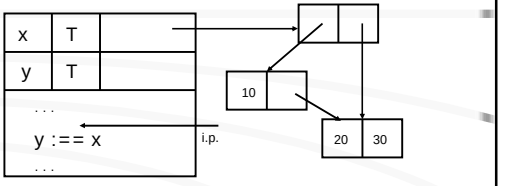


Assignment-
By-Sharing vs By-Copying

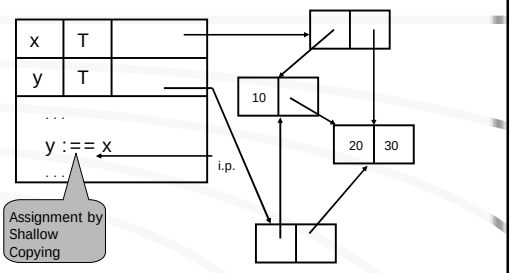
x := y	x :== y x :=== y
By-Sharing	By-Copying

Syntax used in Chapter 2

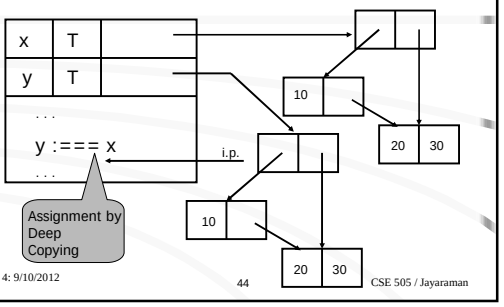
Assignment-by-Copying:
Shallow vs Deep Copying



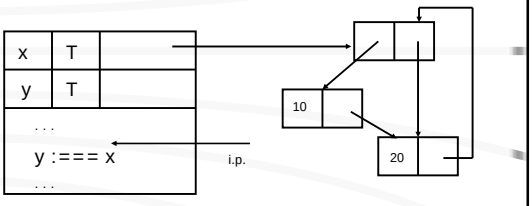
Assignment-by-Copying: Shallow Copying



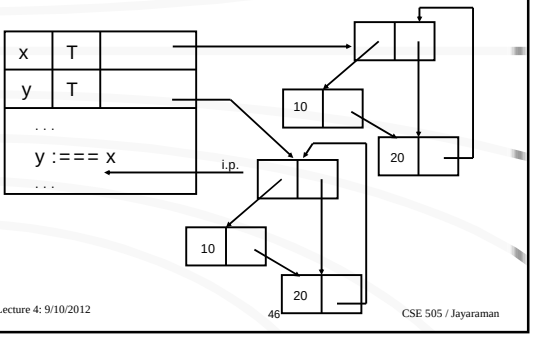
Assignment-by-Copying: Deep Copying



Deep Copying Circular Structures



Deep Copying Circular Structures



Assignment and Binding Schemes

Assignment-by-	Syntax	Which Binding
Sharing	x := y	Fully-Dynamic
Shallow Copying	x :== y	All 3 schemes
Deep Copying	x :=== y	All 3 schemes

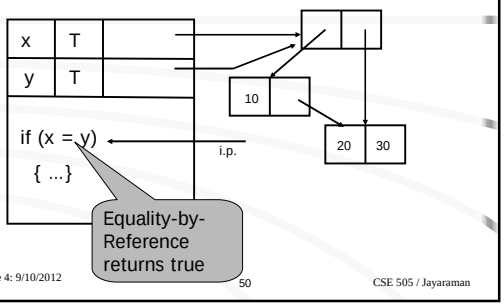
Equality- By-Reference vs By-Value

x = y	x == y x === y
By-Reference	By-Value

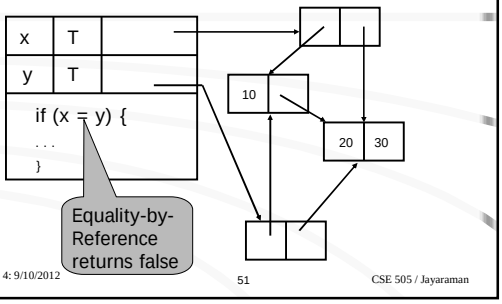
The Three Equalities

- x = y is true iff x and y point to same object.
- x == y is true iff
  (x = y OR x and y point to different objects but with identical content)
- x === y is true iff
  (x = y OR x and y point to isomorphic object structures but with identical primitive values.)

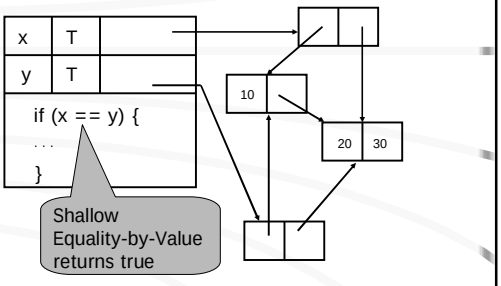
Equality-by-Reference



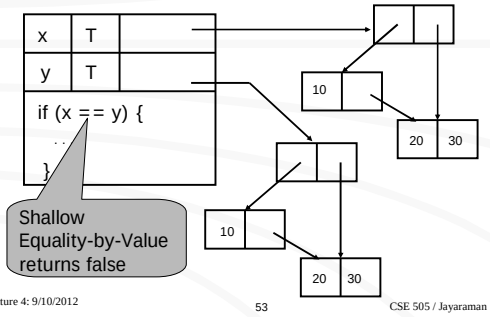
Equality-by-Reference (cont'd)



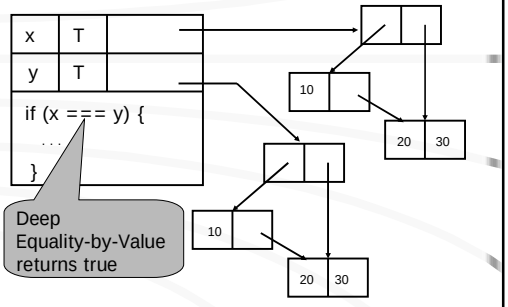
Shallow Equality-by-Value



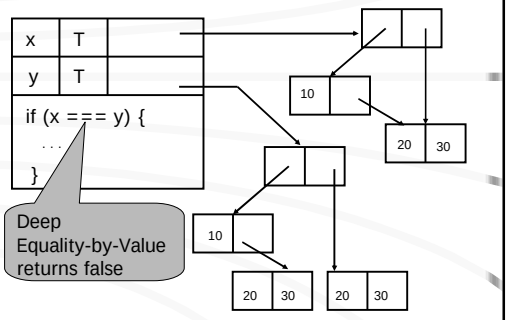
Shallow Equality-by-Value (cont'd)



Deep Equality-by-Value



Deep Equality-by-Value (cont'd)



Discussion

Does (x == y) imply (x = y)? No.

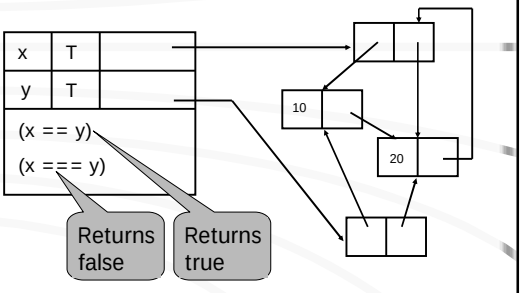
Does (x === y) imply (x == y)? No.

Does (x = y) imply (x == y)? Yes.

Does (x = y) imply (x === y)? Yes.

Does (x == y) imply (x === y)? Maybe!

When == does not imply ===



Discussion

Obviously:

(x = y) implies (x == y)

Less obviously:

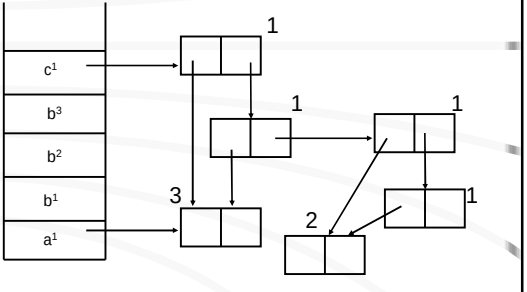
(x == y) does not imply (x === y)

Practical PLs will typically support = and == but not ===, because the latter is harder to implement and may be application-dependent.

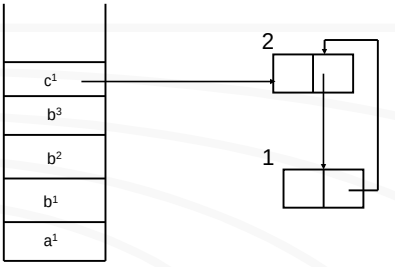
Heap Storage Management

- Reference Counting
  - each heap cell maintains an integer count
  - initialize to 1
  - increment and decrement upon pointer assignment/re-assignment
  - reclaim cell when count reaches 0
  - cannot reclaim circular structures
  - slows down program due to reference mgmt

Ripple effect when count becomes 0



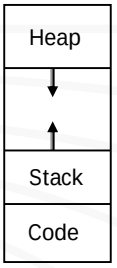
Reference Counting Cannot Reclaim Circular Structures



Heap Storage Management

- Mark-scan Garbage Collection
  - No reference counting
  - When memory runs low, traverse heap objects from stack and mark all reachable objects
  - Sequentially scan memory and reclaim un-marked memory cells
  - Can reclaim circular structures
  - Time taken is inversely proportional to amount of garbage

Stack – Heap Spaces



Unix limit command

```
nickelback % limit

cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    10240 kbytes
coredumpsize 0 kbytes
memoryuse    unlimited
vmemoryuse   unlimited
descriptors  1024
memorylocked 32 kbytes
maxproc      200
```


How much recursion with 10MB stack?

```
#include <stdio.h>

void recurse() {
    int data[262144];        // 1 MB
    recurse();
}

int main(){
    recurse();
}
```

Can change setting: % unlimit stacksize

Testing heap allocation

```
#include <stdio.h>
#include <stdlib.h>

typedef struct list {      // will take about 1MB of storage
      int data[262144];
      struct list *next;
      } LIST;

int main(){
   while (1) {
   LIST* node = (LIST*) malloc(sizeof(LIST));
   if (node == (LIST*) NULL) {
      printf("Ran out of heap space\n");
      exit(1);
   }
   printf("%d\n", sizeof(LIST));
   }
}
```