Pointer Semantics

CSE 340 FALL 2021

Rida Bazzi

Assignment Semantics

Assignment semantics is concerned with the meaning of

```
a = expr
```

where

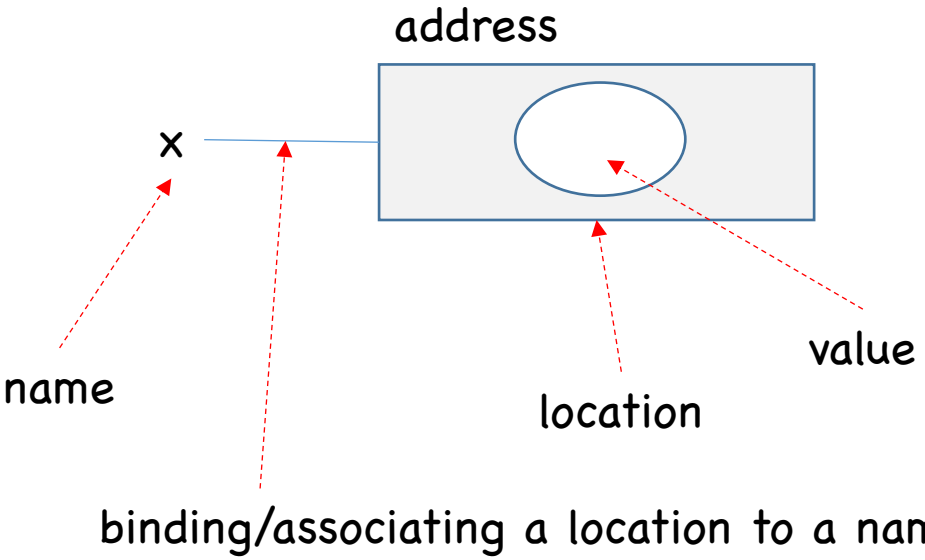
- a can be a variable or, more generally, an expression and
- expr is an expression.

In general there are two kinds of semantics used by programming languages

1. Copy semantics. This semantics is used in C, C++ and is used for basic types in Java
2. Reference semantics. This semantics is use by Java for assigning object values

In what follows I will concentrate on copy semantics, but I will later explain reference semantics

Box-Circle Diagram



A box-circle diagram makes clear the distinction between a name and the location that that is associated with the name. In general, a name can have a location associated with it, as is the case for a variable in C, or might not have a location associated with it, such as the name MAX_INT which is the name of a constant.

Also, we make the distinction between the location and the value stored in the location. A location can store different values at different times.

Finally, we make the distinction between a location and the address of the location. The location itself can be thought of as a physical location but the address is just a number that can be used to describe the "position" of the location in memory. The address itself is not a name of the location but can be used in naming the location. For example 1024 is an address and a name for the location whose address is 1024 is "The location at address 1024"!! This is not playing on words. The distinction is real.

We say that the location (the box) is *associated* with the name. The line between the name and the location represents this association (which is also called binding)

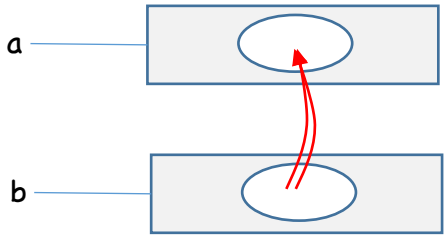
In general, a name need not be a simple variable name. We also, treat more involved expressions as names. For example a[i] where a is an array is the name of a location (that depends on the value of i).

In general, we distinguish between expressions that have names associated with them from other expressions. Expressions that have locations associated with them are called l-values. Expressions that have values but no locations associated with them are called r-values. This is further described next.

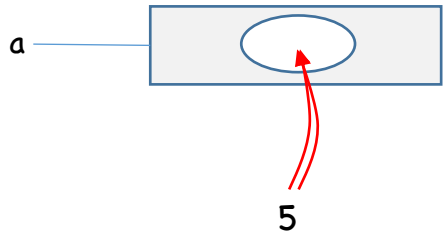
Assignment under copy semantics

There are two general forms of assignment. Each assignment we will consider can be reduced to one of the following two forms

1. a = b — copy value in location associated with b to location associated with a



2. a = 5 — copy value 5 to location associated with a



In both forms, a value is copied to a location. The difference is where the value comes from.

l-values and r-values

l-value — is an expression that has a location associated with it

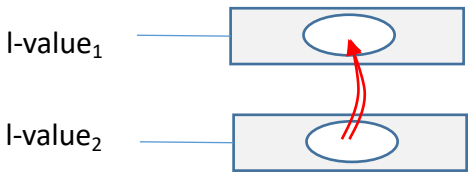
Examples — a, b[i+j], b[i+b[i]], *p, **q

r-value — is an expression that does not have a location associated with it, but has a value associated with it

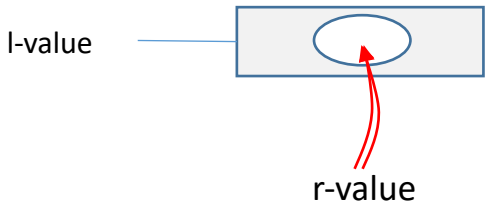
Examples — 5, i+j, 2*a

Possibilities for assignment

1. l-value$_1$ = l-value$_2$ — copy value in location associated with l-value$_2$ to location associated with l-value$_1$



2. l-value = r-value — copy value of the r-value to the location associated with l-value



3. r-value = l-value — not possible

4. r-value = r-value$_2$ — not possible

Pointer Semantics in C

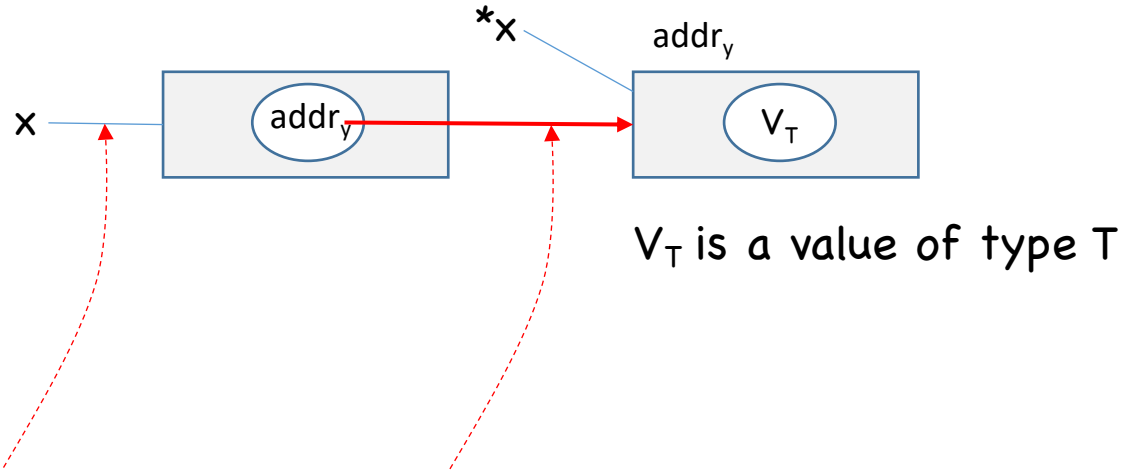
Pointer declaration has the form

```
T * x;
```

where T is a type. We read the declaration as "the type of x is pointer to T"

Examples
```
int * x;     // type of x is pointer to int
int * *x;    // type of x is pointer to int *
```

If `T * x;` is a declaration for x, the location associated with x stores a value which is the address of a location that stores a value of type T



binding to illustrate that location is associated with name x. This is not a pointer

points to: to illustrate that value of location associated with x is address of location pointed to

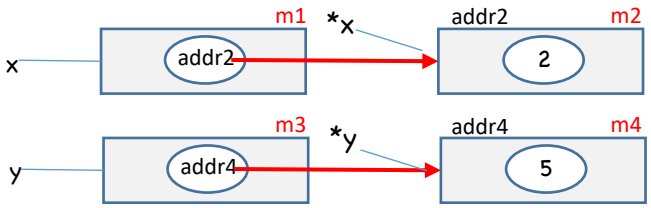
The picture shows that the location "pointed to" by x is associated with the name *x. More specifically, if x is a pointer variable, *x is an l-value. The location associated with *x is the location who address is equal to the value in the location associated with x. More simply, the location associated with *x is the location "pointed to" by x.

Pointer Semantics Examples

We consider two pointer variables x and y and assume

```
int *x;
int *y;
...
// point 1
```

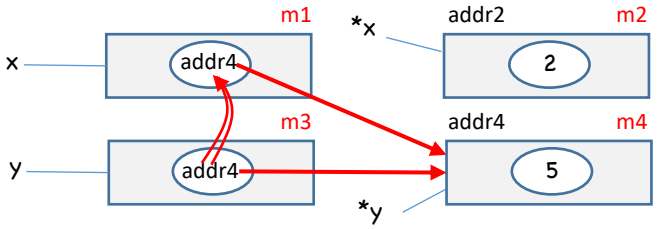
The ... represents some missing code that is not shown and we assume that at point 1, the box-circle diagram is the following



In the diragram, m1, m2, m3 and m4 are used to refer to the boxes without using program variables. Notice how in the diagram location m2 is associated with *x and location m4 is associated with *y.

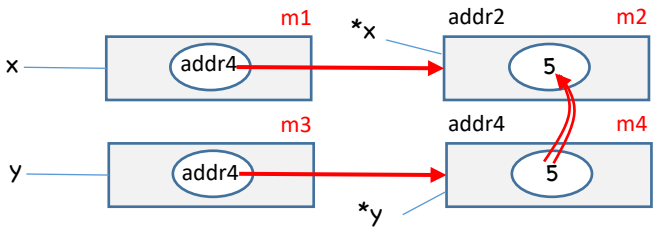
Assuming the situation is as show above, we consider the effects of various assignments.

1. x = y : this will copy the value in the location associated with y to the location associated with x. The situation becomes as follows



notice how the value in the location associated with y (addr4) is copied to the location associated with x. The result is that x points to m4.

2. *x = *y : this will copy the value from the location associated with *y to the location associated with *x (remember that this is being applied to the situation above).



malloc(): memory allocation function

malloc() :	input:	integer which specifies the "size" in bytes of the memory to be allocated
	output:	r-value which is the address of a location
	type:	the returned value has type void *, which only means that it is a pointer. In order to assign the returned value to a variable, it needs to be typecast.

malloc() allocates memory whose size is equal to the "size" parameter and returns the address of the first byte of the allocated memory

The allocated memory is allocated on the heap and is not initialized by malloc()

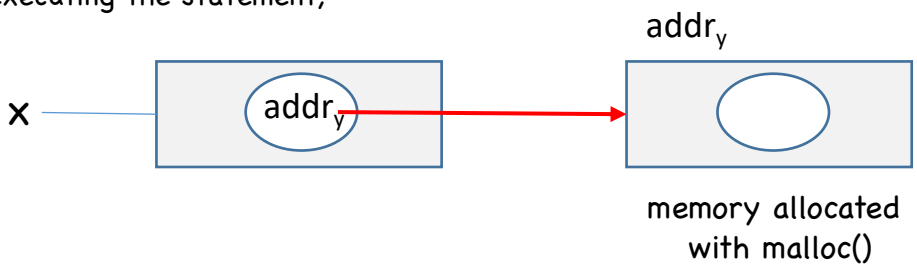
Example

If x has type T *, we allocate memory with malloc() as follows

```
x = (T *) malloc(sizeof(T));
```

The call to malloc(), allocated memory whose size is the size of a value of type T. The returned value has type void * . That is why we use type casting (T *) when we assign the value to x.

After executing the statement,



Note C does not require that a value of type void * be typecast in order to assign it to a variable of type T *. The type casting is implicitly done. Nevertheless, it is good practice to have an explicit type case in this case. It makes the code more readable and potentially easier to detect mistakes. This is why C++ requires typecasting with malloc(). Remember that code that you write is read much more often than it is written!

free(): memory de-allocation function

free() : input: pointer to memory that was previously allocated

output: no output

free() de-allocates the memory by making it available for future allocation.

The input to free() must have a value which is the address of a previously allocated memory. If the value passed to free() does not satisfy this requirement, its behavior is undefined which means that you cannot rely on what will happen.

The size of the memory to be freed is not specified. The memory manager knows the size because it stores that information when malloc() was previously called.

& : address operator

& & is a unary operator

the operand of & must be an l-value

the result is an r-value

Value The value of & l-value is equal to the address of the location associated with the l-value

Example The value of &x is the address of the location associated with x

Type If x is of type T, then &x has type T * (pointer to T)

Example If x has type int, &x has type int *

* : dereference operator

* * is a unary operator

operand l-value or r-value

result always l-value

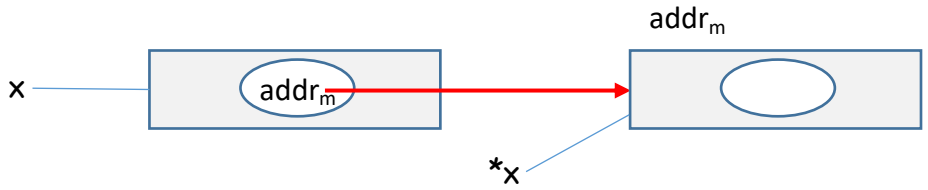
Location the location associated with *(expr), where expr is an expression that is either an l-value or an r value is given as follows

* r-value the location associated with * r-value is the location whose address is equal to the value of the r-value

Illustration

* l-value the location associated with * l-value is the location whose address is the value in the location associated with the l-value

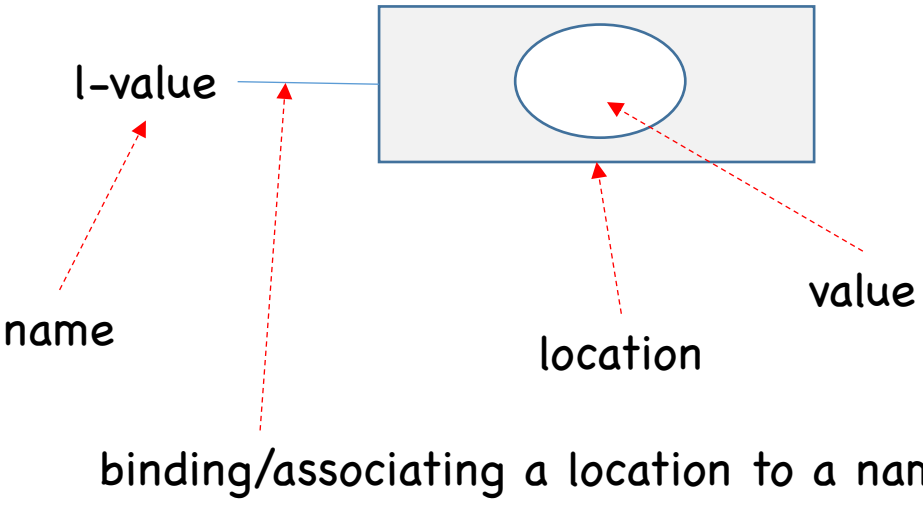
Illustration



Informally, if x is a pointer, the memory associated with *x is the memory "pointed to" by x.

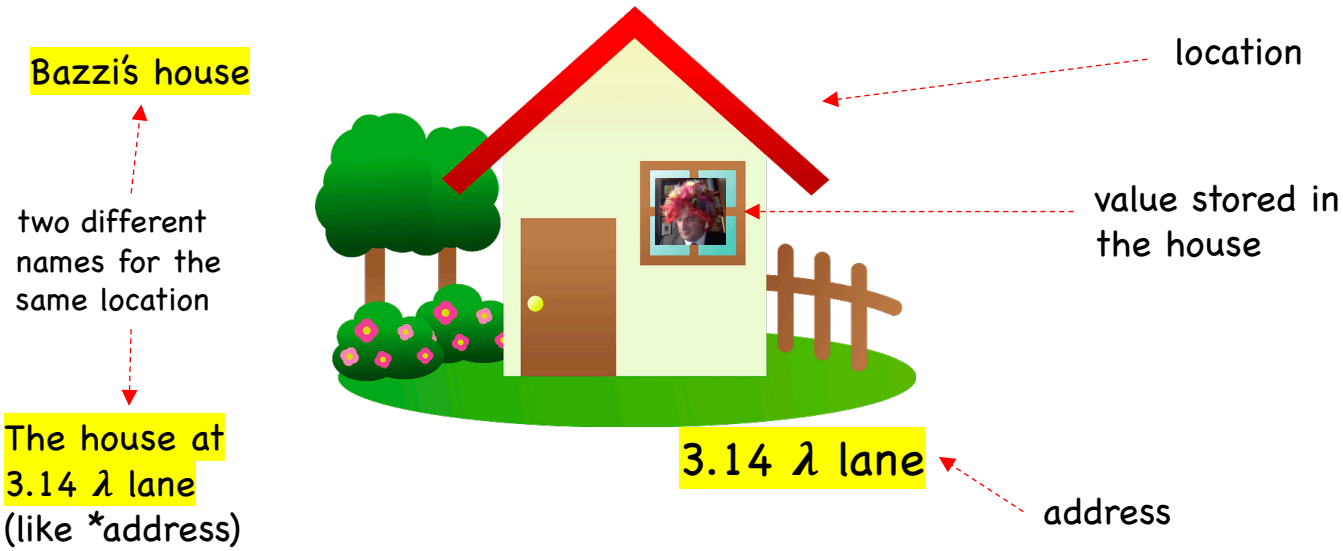
Type If x has type T *, *x has type T

Box-circle diagram revisited



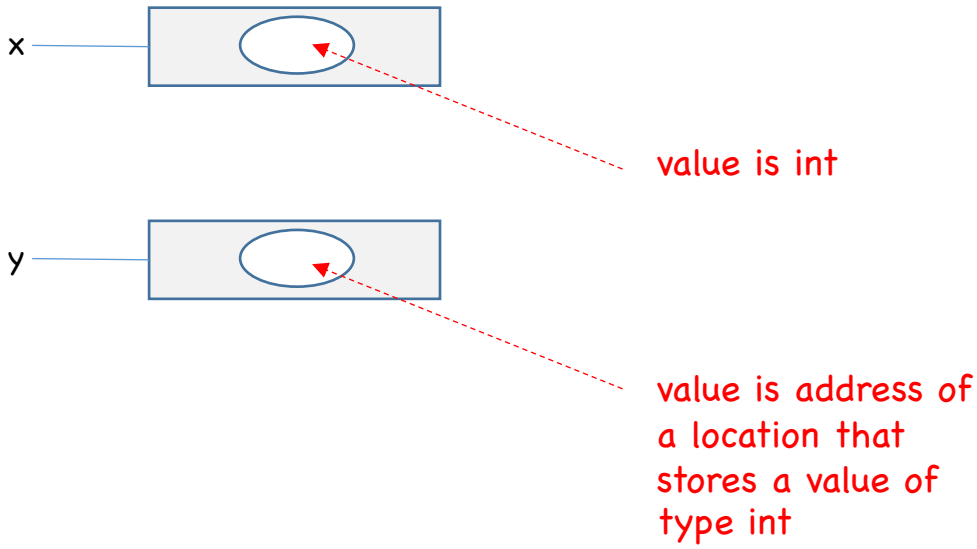
- An l-value is an expression that has a location associated with it.
- The l-value itself is the name of the location.
- The address of the location associated with an l-value is given by &l-value.
- The location associated with the l-value contains a value
- For simplicity, we can call the value in a location associated with an l-value, the value of the l-value
- The address of the location associated with an l-value is not the name of the location

An analogy can help in making the distinctions clearer

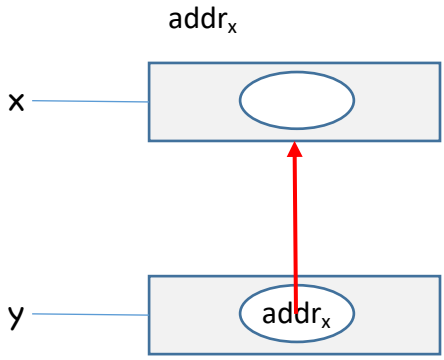


Example

```
int x;
int *y;
```

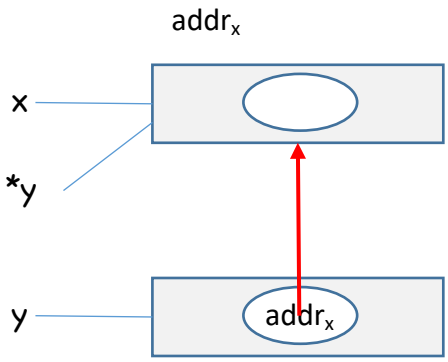



```
y = &x;     // &x = address of location associated with x
            // equivalent to y = addr_x where addr_x is the address
            // of location associated with x

            // note the arrow below point to box not to value inside
            // the box
```



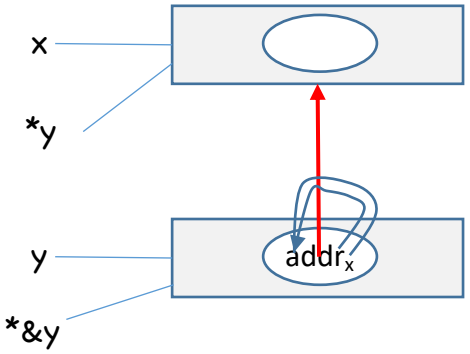
Example continued

```
y = &*y;          // *y is an l-value
                  // the location associated with *y is the
                  // location whose address is the value in
                  // the location associated with y (the value
                  // of y)
```




```
                  // &*y is the address of the location
                  // associated with *y = addr_x
```

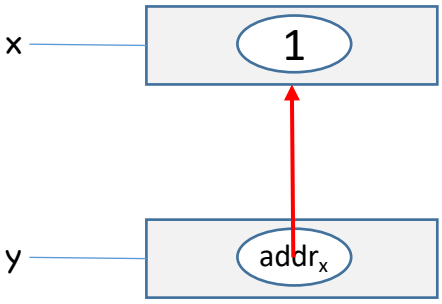
```
y = *&y;          // *&y is an l-value

                  // the location associated with *&y is the
                  // location whose address is equal to
                  // &y = addr_y
```



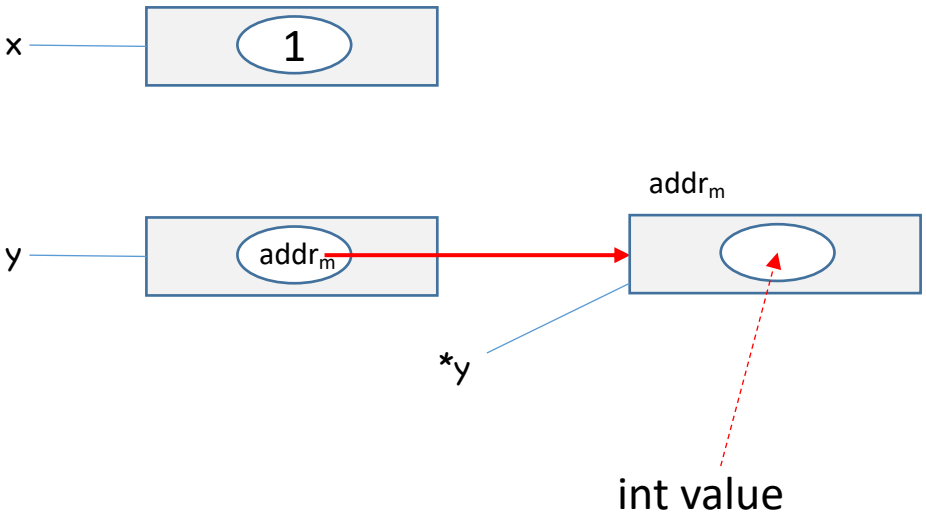

```
                  // y = *&y  is equivalent to y = y
                  // so the value of y does not change
```

Example continued

```
x = 1;
```

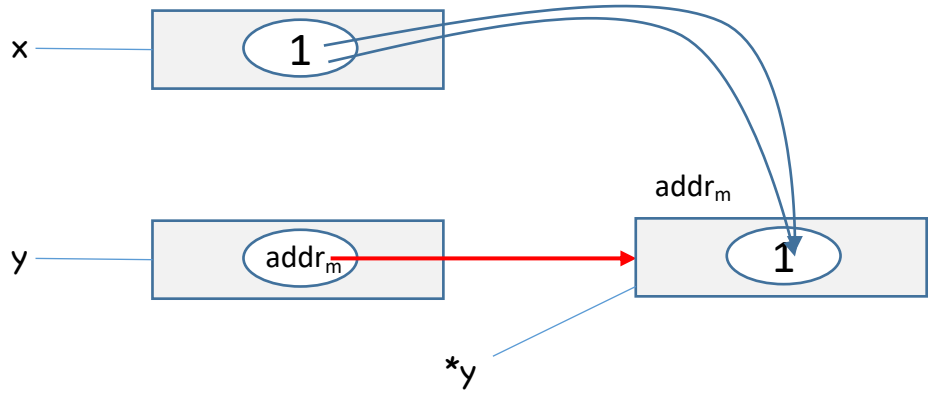



```
y = (int *) malloc(sizeof(int));        // y = addr_m address of memory
                                        // location allocated by malloc()
```



Example continued

```
*y = x;      // copy value in location associated with x to location
             // associated with *y
```

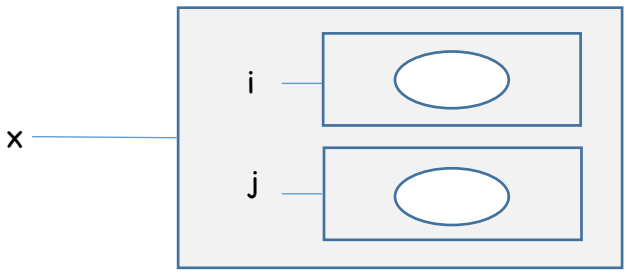


Structures

When we declare a structure

```
struct {

    int i, j;
} x;
```

We represent the box-circle diagram for the structure as follows:



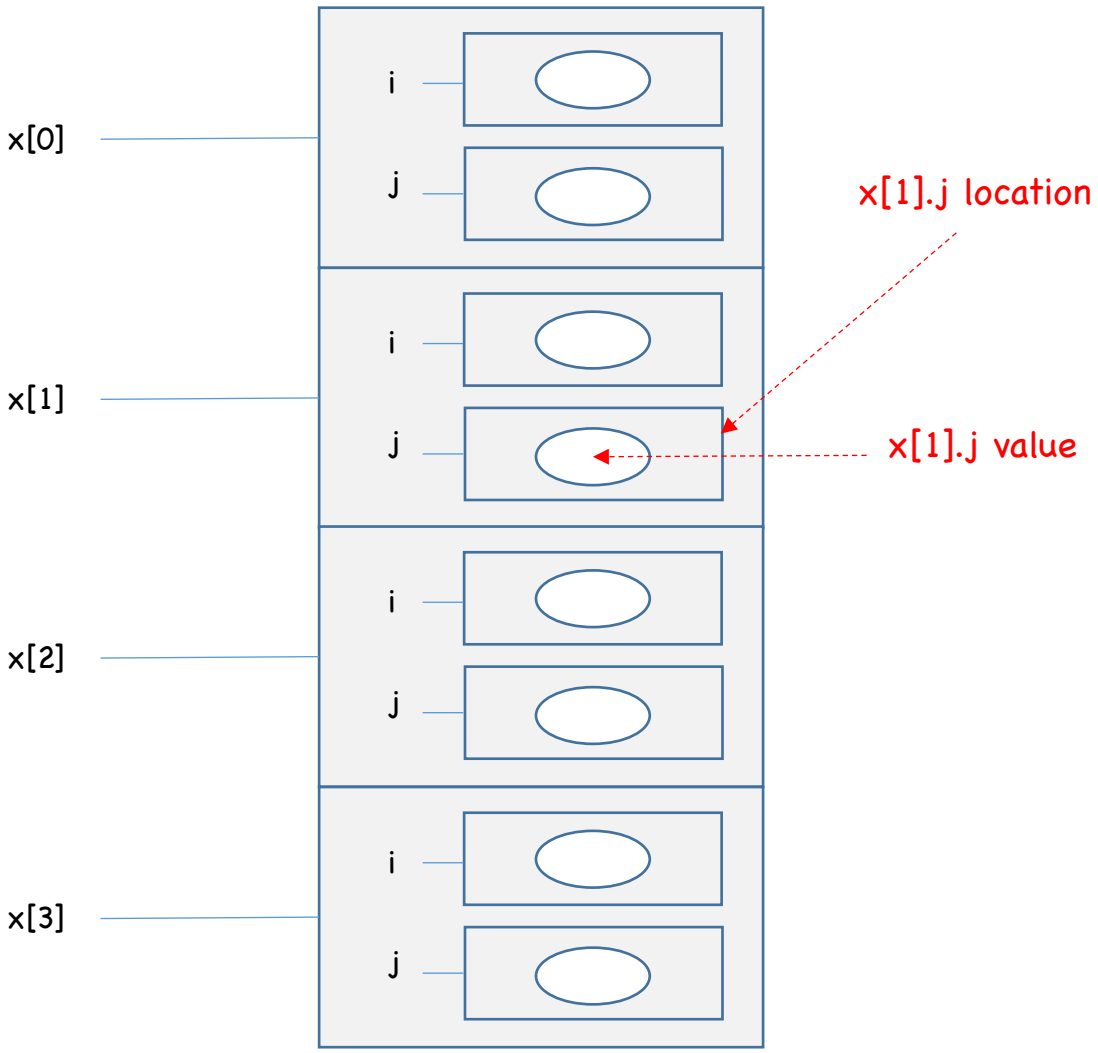
Note how the locations for x.i and x.j are inside the location we associate with x.

Arrays

When we declare an array, each entry in the array will have a corresponding box. I will give an example with an array of structures

```
struct {

    int i, j;
} x[4];
```

We represent the box-circle diagram for the array of structures as follows:

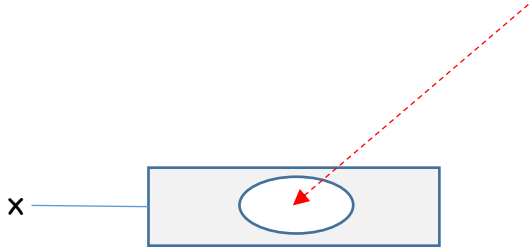


Pointers with Structures

When we declare a structure

```
struct st {
     int i;
     struct st * next;
} * x;
```

depending on situation,
the value can be initially NULL
or uninitialized (wild pointer)

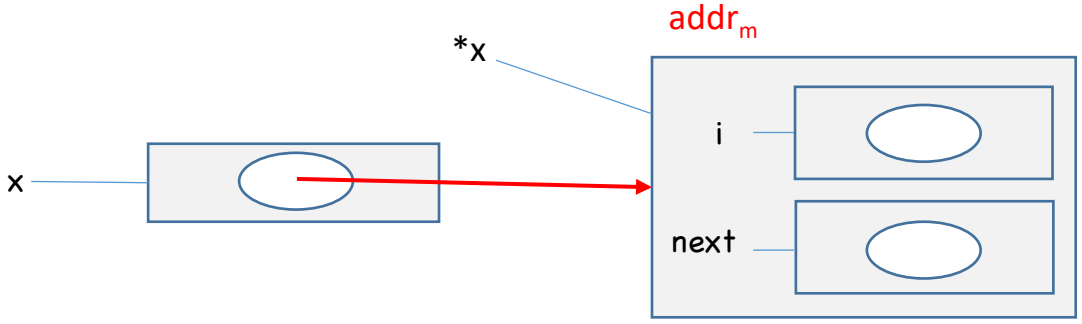


Notice here that x is a pointer and the value in the location associated with x is the address of a location that stores a value of type st.

If we execute

x = (struct st *) malloc(sizeof(struct st)); // $addr_m$ returned by call

we get



Aliases

Two expressions are aliases of each other if they have the same location associated with them.

We have already see that x and *&x are aliases of each other

In general, there are multiple ways to obtain aliases

1. **pointers**

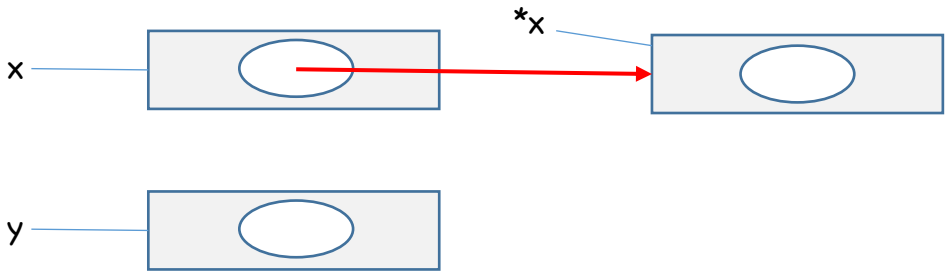
2. **arrays**

3. **pass by reference** (we will describe that later)

4. **assignment with reference semantics** (Java)

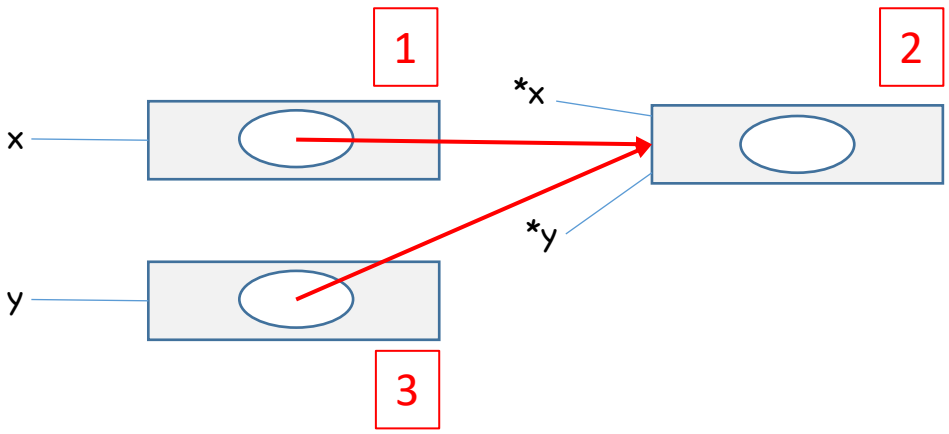
Pointer Aliases: Example

```
int * x;
int * y;

x = (int *) malloc(sizeof(int));
```



at this point y's value is uninitialized (assuming it is a local variable. If it is a global variable, it will be initialized to 0)

If we execute y = x, we get



*y is an alias of *x because they have the same location ([2]) associated with them

y is NOT an alias of x because they have different locations ([1] and [3]) associated with them

Array Alias Example

```
int a[10];
int i,j;

i = 5;
j = 3;
```

a[i-2] is an alias of a[j]

Dangling Reference

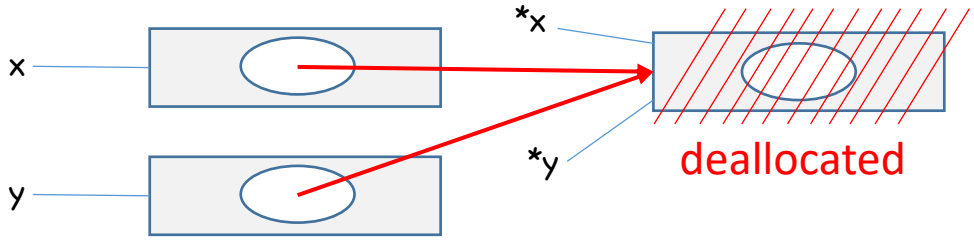
Definition A pointer is a dangling reference if its value is the address of a location that:

- has been allocated and
- has been deallocated

Dangling Reference: Example 1

pointer to deallocated memory

```
int *x;
int *y;
x = (int *) malloc(sizeof(int));
y = x;
free(x);

// at this point both x and y are dangling references
```

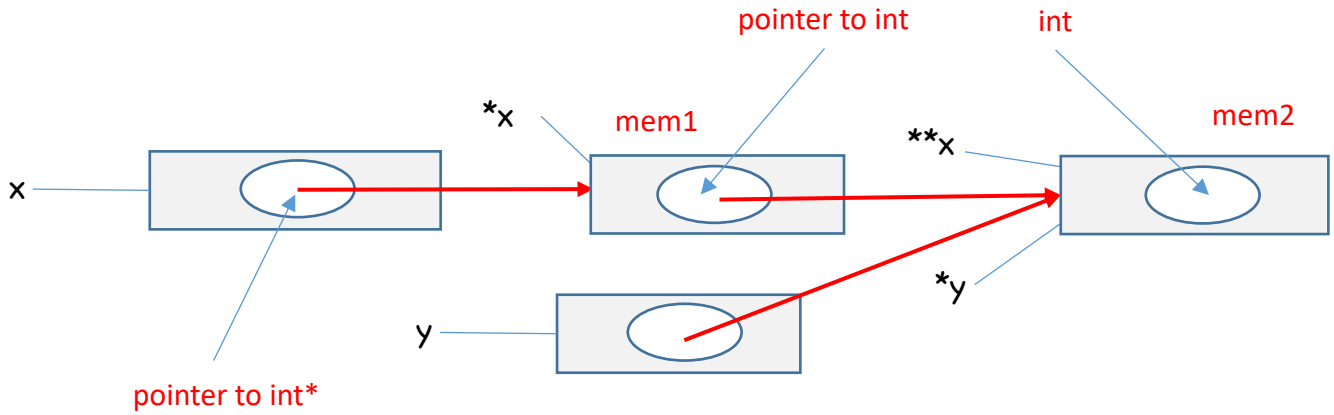


Note. free(x) frees the location "pointed to" by x.
free(x) does not change the value of x.

Dangling Reference: Example 2

pointer to deallocated memory

```
int **x;
int * y;
x = (int **) malloc(sizeof(int *));     // mem1
*x = (int *) malloc(sizeof(int));       // mem2
y = *x;
```



For the code above, remember that if T * x; is a declaration, then we allocate memory for x as follows

```
x = (T * ) malloc(sizeof(T))
```

In the example above T is int * which explains the code

```
x = (int * *) malloc(sizeof(int *));
```

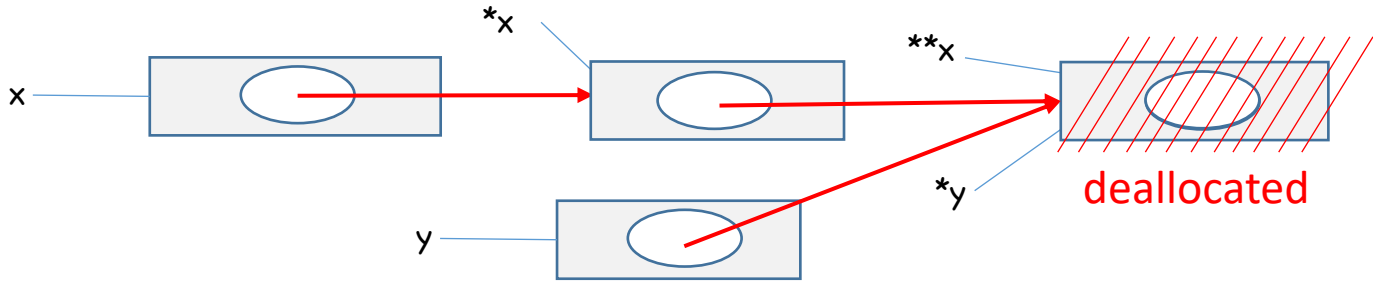
Also, recall that if T * x; is a declaration *x is of type T. In the code above, T is int *, so we allocate for *x as follows

```
*x = (int *) malloc(sizeof(int))
```

Dangling Reference: Example 2

pointer to deallocated memory

```
int **x;
int * y;
x = (int **) malloc(sizeof(int *));
*x = (int *) malloc(sizeof(int));
y = *x;
free(*x);
```



Now *x and y are dangling references

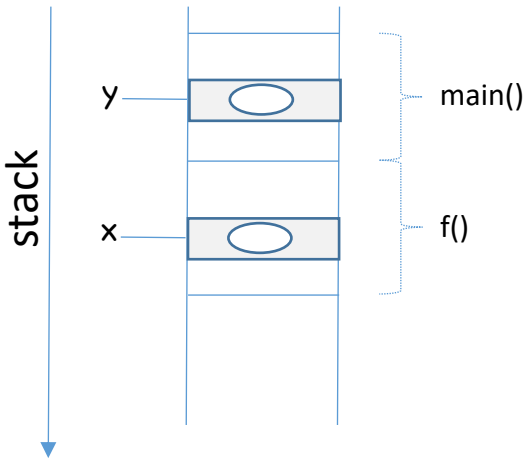
Note free(*x) frees the memory pointed to by *x
free(*x) does not modify the value of *x

Dangling Reference: Example 3

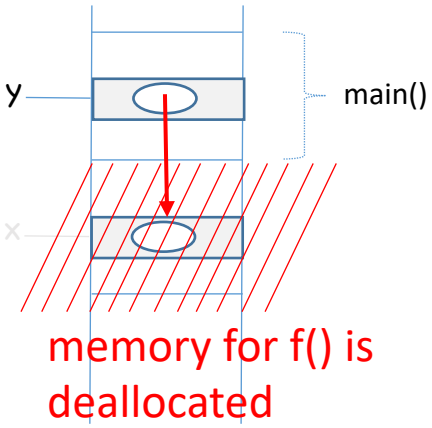
pointer to local variable of a function that exited

```
int * f()
{  int x; // memory for x allocated on stack
   // point 1
   return &x;
}

main()
{
   int *y;
   y = f();              // memory for x deallocated when function
                         // returns but y still point to it
  // point 1

}
```

// point 1



// point 2



memory for f() is deallocated

when f() exits, its memory on the stack is de-allocated, and the value of y is the address of de-allocated memory, so y is a dangling reference

Dangling Reference: Example 4

pointer to variable from outside its scope

```
{ int *x;

    {   int y;

        x = &y; // point 1
    }
    // point 2
 }
```

At point 1, the value of x is the address of the local variable y

At point 2, y is no longer accessible and its memory is reclaimed (de-allocated) but x still points to the memory previously associated with y

Note. In practice, the memory allocated for y might not be reclaimed, but since y is out-of-scope, its memory can be de-allocated by the compiler and it should be treated as de-allocated memory

Dangling References

- Possible in C
- Possible in Ada if unchecked_deallocation_package is used
- Not possible in Java

Garbage (aka memory leak not )

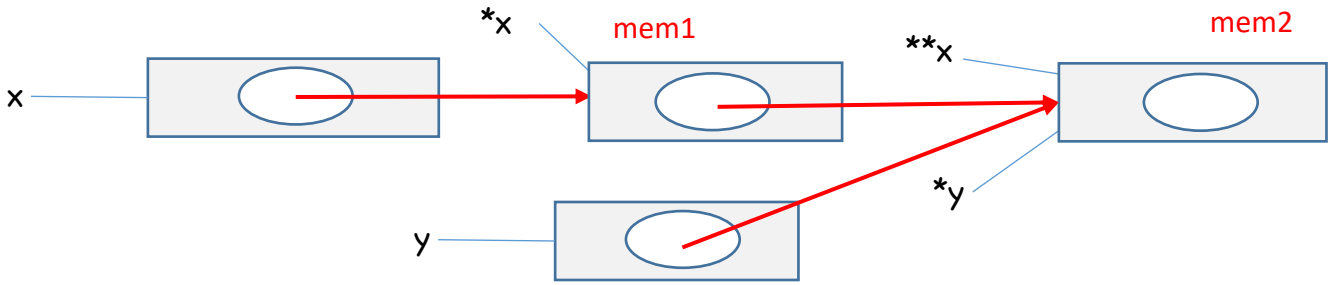
A location is garbage if

- The location has been allocated and
- The location has not been deallocated and
- The location is no longer accessible by the program

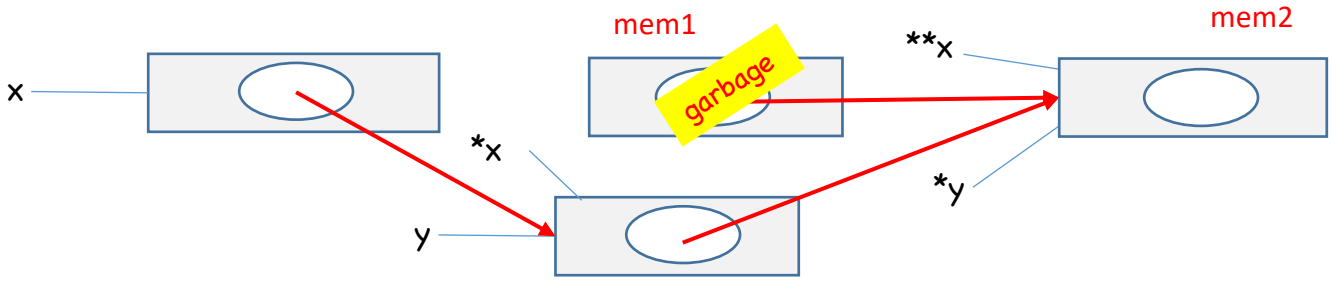
What does no longer accessible mean?

It means that you cannot refer to it. Or that the memory does not have a name

Example



If we execute x = &y



mem1 has no name. It is garbage.
The name of mem2 is *y or **x. It is not garbage

Why is garbage a problem?

Consider a long-running server that continuously processes incoming requests.

```
repeat
        receive input
        call f(input) to process input
        produce output
forever
```

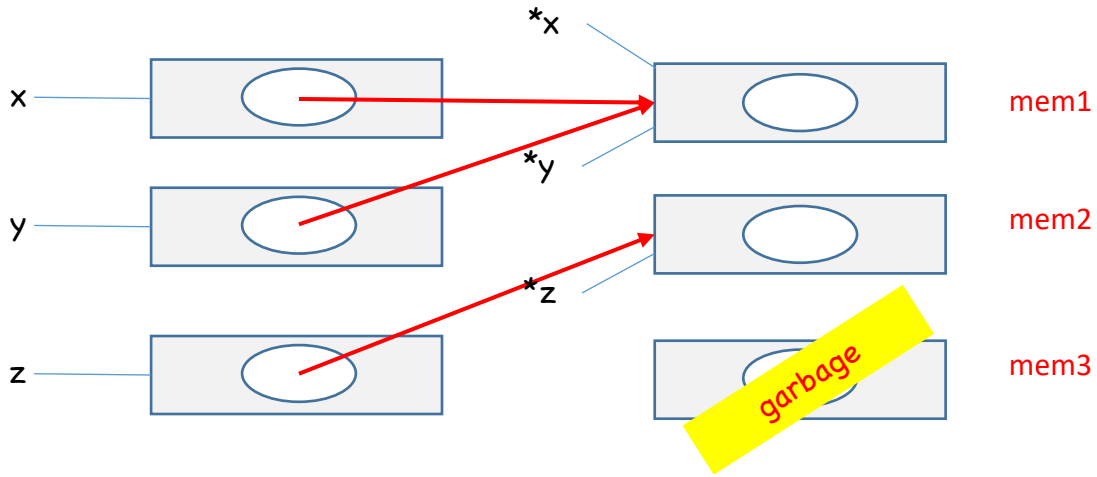
If the function f() allocates some memory that is not needed after the call, then as time goes by more and more memory gets allocated without it being deallocated (which creates a big leak as shown on the next page). At some point, there will be no more heap memory to allocate and the program will fail.


BIG LEAK

Garbage: Example 1

```
{ int * x;
  int * y;
  int * z;

  x = (int *) malloc(sizeof(int));  // mem 1 allocated
  y = (int *) malloc(sizeof(int));  // mem 2 allocated
  z = (int *) malloc(sizeof(int));  // mem 3 allocated
  z = y;
  y = x;

  // point 1
}
```

At point 1, the box-circle diagram looks as follows



mem1 is garbage because there is no way to refer to it in the program. mem2 and mem3 are not garbage because they can be referred to in the program: mem1 is associated with *x and *y and mem2 is associated with *z

Garbage: Example 2

exiting a function before free

```
f()
{ int * x;
  x = (int *) malloc(sizeof(int));  // memory 1 allocated
}

// memory 1 is garbage when function exit if free() is not called
```

In C, heap memory is not deallocated unless it is deallocated explicitly using free()

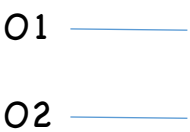
Stack memory is automatically deallocated when the function returns or when the local scope is exited

Reference Semantics for Assignment: Java

In Java, object assignment does not have copy semantics.

If O1 and O2 are two objects of class C

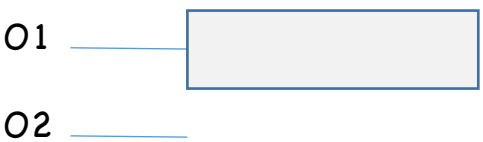
Initially, there is no location associated with O1 and O2.



If we execute

```
O1 = new A;
```

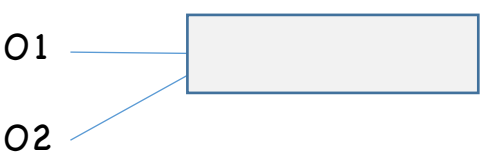
we get



If we execute

```
O2 = O1;
```

we get



At this point, O2 is an alias of O1