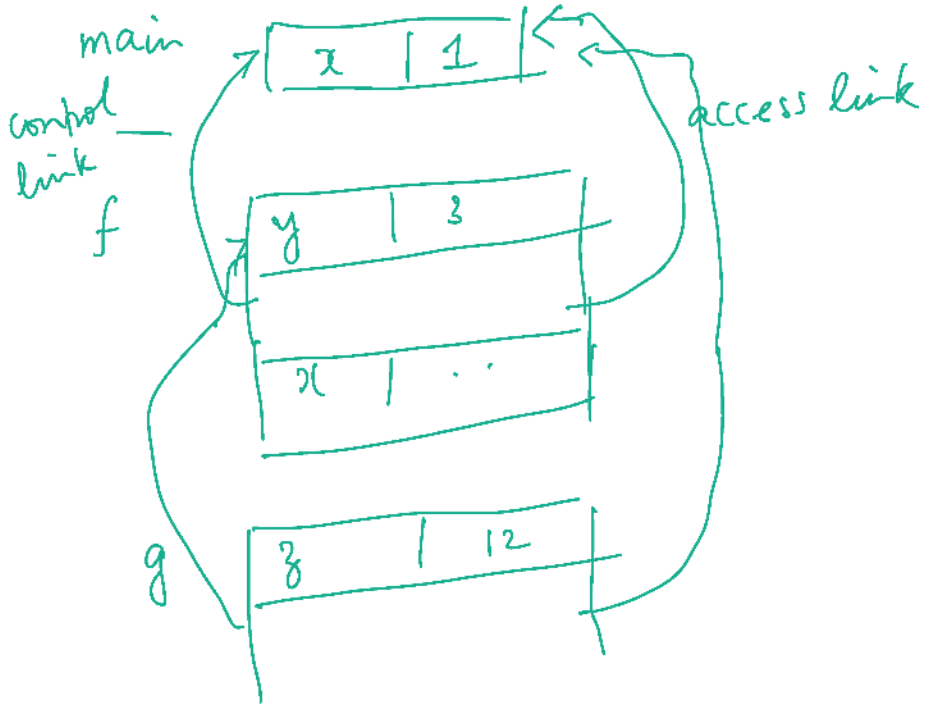
Previous class: run-time support for functions

stack allocation.

```
main {
    int x = 1;
    int g(int z) {
        return x+z;
    }
    int f(int y) {
        int x = y+1;
        return g(y*x);
    }
    f(3);
}
```

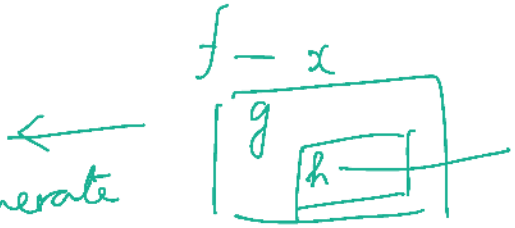


in the code for g, whenever there is a reference to x

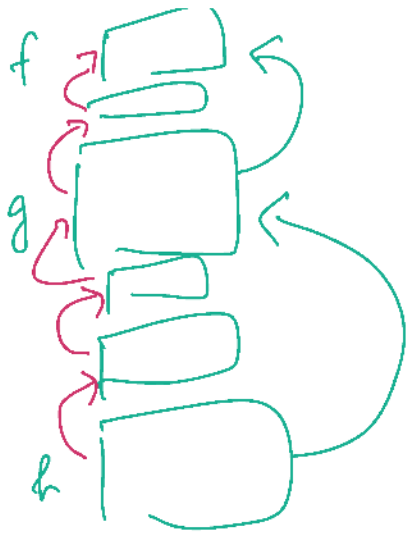
```
w = x + 1;
```

LOAD the value of x
ADD 1
STORE it to w.

go to access link; calculate some offset for x;
LOAD the value at that location
ADD 1
STORE w.

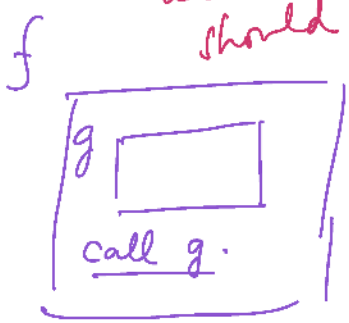


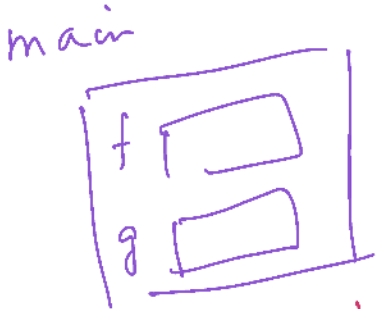
the compiler will generate code to follow the chain of access links twice.



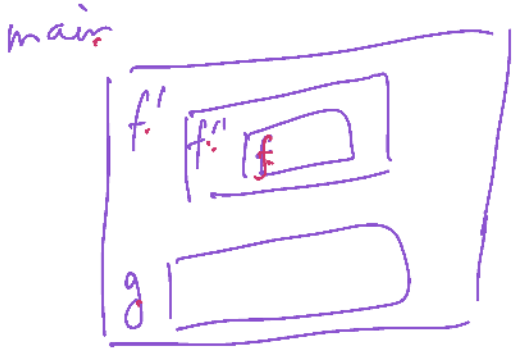
how to determine where the access link should point to.

f calls g.





access link of g should point to wherever access link of f points to.



access link should point to closest enclosing function (which is guaranteed to be on the stack).

access

Dynamic allocation.

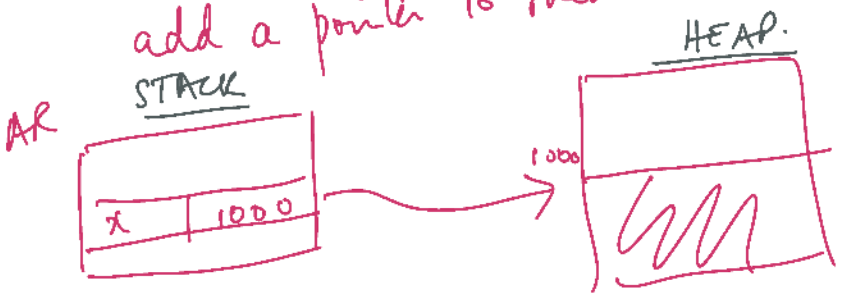
heap allocation.

functions handle complex data types.

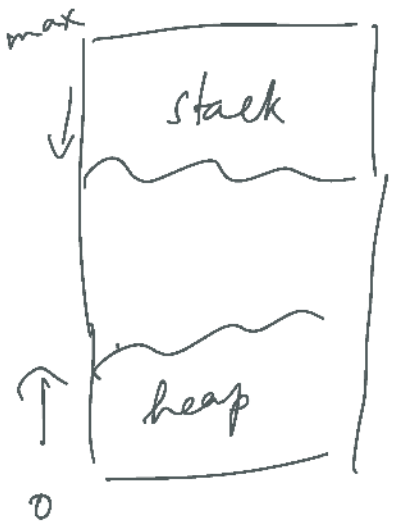
- array.
- classes
- dynamic data structures (no pre-specified bound on the number of elements)
  - linked lists, graphs

```
int f ( int n ) {
   int array [n];
}
```

when we need to allocate space for such objects, we put them on the heap and add a pointer to them on the stack.



heap - a portion of memory
(there is no structured access).

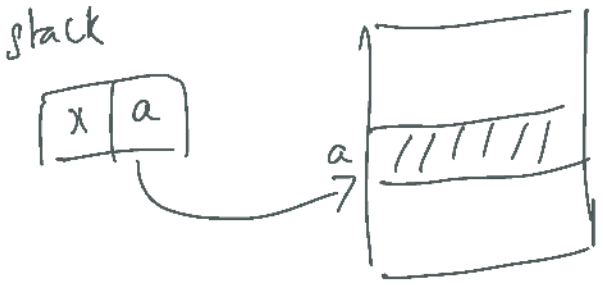



```
int f (int n) {
  int arr[n];
  ...
}
```

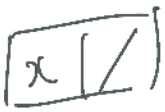
Suppose arr is stored on the heap and
pointer is maintained on the stack.
when the function exits the stack, the memory allocated
for arr is no longer accessible.
= reachable from the stack.

allocated but unreachable
memory locations = garbage.

```
int *x = malloc(...)      free.   C.
x = NULL;                         C++ new/delete
```



C/C++/Java /Swift all create garbage

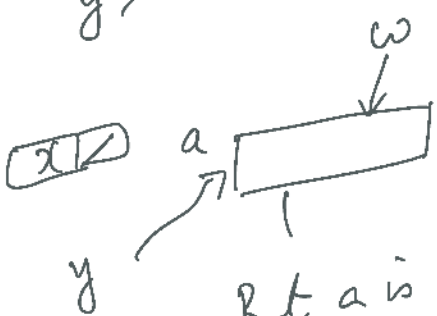


But C/C++ have an additional problem because of explicit deallocation.

dangling pointers

```
int *x = malloc(...)
int *y = x;

free(x); x = NULL;
int *w = malloc(...)
int *z = y.
```



But a is also in the free memory pool.

dangling pointers

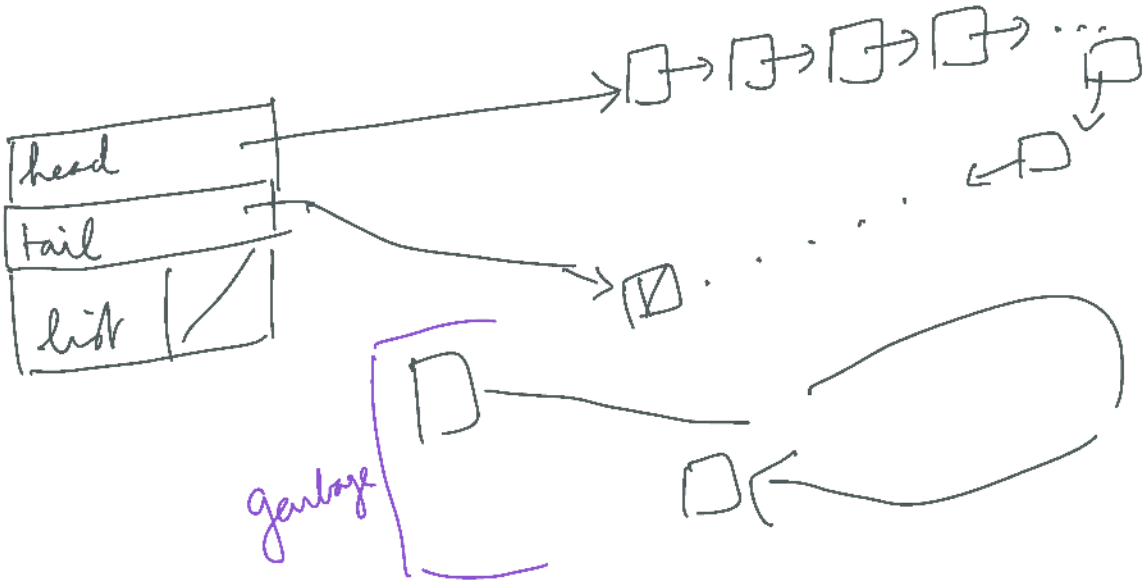
a deallocated memory location is reachable from the stack.

Two main methods for garbage collection.

- mark-and-sweep garbage collection — Java / Android
- reference counting — Objective C / Swift / iOS.

Mark-and-sweep algorithm.

Mark phases: start from the stack



Mark phase: start from the stack.
mark every reachable location.
(graph reachability algorithm. start nodes on the stack.)

Once all reachable "nodes" are marked

enter the sweep phase.

go through mem from 0 to max,

add any unmarked node to free memory pool.

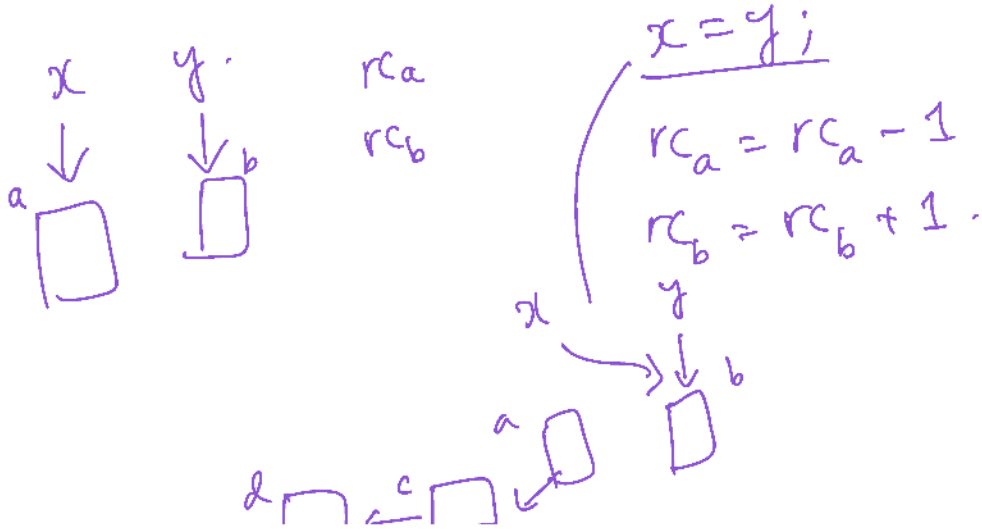
---

reference counting.

each block of memory has a counter that comes with it.

when the block is allocated,

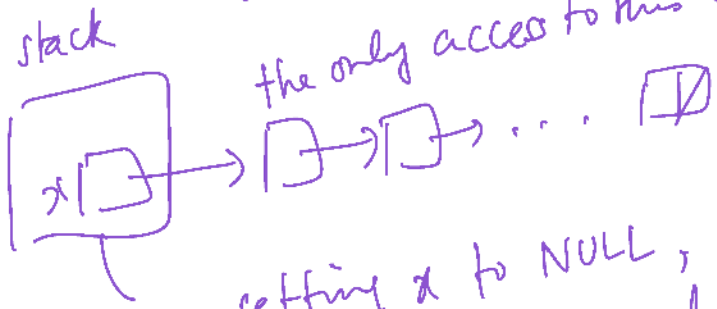
rc = 1.



when $rc_a = 0$,

for every block ω that a points to

$rc_\omega = rc_\omega - 1$.



on setting x to NULL, the entire list will be returned to the free memory pool.

Pros.

typically, some low cost operation

there is no accumulation of garbage.

Con.

Sometimes simple assignment triggers a long chain of deallocations.

Lazy approach might help.

Mark for deletion / delete later.

can happen in parallel.

(because these nodes are anyway inaccessible).

Drawback of reference counting.

Cyclic references.

doubly linked lists.

```
class Person {
    House myhouse;
}
```

```
class House {
    weak Person tenant;
}
```

Swift. strong references
weak references.

if all incoming references to a block are weak, deallocate.