CS 421 Lecture 13

- Execution of dynamic languages
  - Sun HotSpot run-time system for Java
    - Tags, just-in-time compilation, reflection
  - Memory management
    - Memory layout; definition of "garbage"
    - Reference-counting
    - Garbage collection
      - Non-compacting (mark-and-sweep)
      - Compacting

Dynamic languages

- Automatic memory management
- Tagged values (for gc, run-time type-checking, reflection)
- *Sometimes*: dynamic type-checking
- *Sometimes*: reflection
- *Usually*: execute virtual machine code

- Will use Sun HotSpot Java virtual machine as example.

Java HotSpot run-time system

- Developed around 1999 – replaced existing widely-used Java VM
- Described in several places, e.g. http://java.sun.com/products/hotspot/whitepaper.html
- HotSpot is VM used in java program, and embedded in many browsers

(Note re: above document: word "compiler" used to refer to translator from Java bytecode to native machine code, not translator from source code.)

Java HotSpot run-time system

- Garbage collection
- Two-word object headers
- Executes .class files (Java VM code)
  - "Just-in-time" compilation
- Meta-objects represented as objects

Meta-objects represented as objects

- Class and Method are classes
- Each class corresponds to a Class object
  - Methods of class Class include getDeclaredMethods(), getFields(), …
- Each method corresponds to a Method object
  - Methods of class Method include getParameterTypes, getReturnType, …
- E.g. can invoke methods that are detected dynamically – e.g. search all objects reachable from one object, and invoke method print on any object whose class contains a print method.

Two-word headers

- Every object in heap is preceded by two words
  - First word is pointer to Class object of this method’s class (which gives layout of object)
  - Second word contains g.c. info
  - Arrays contain third word giving length

Just-in-time compilation

- Methods obtained in bytecode form (.class files) translated to native machine code on the fly
- Numerous optimizations employed
  - Very important optimization: inlining
- Level of optimization determined by monitoring execution
  - Heavily used methods are optimized, and possibly re-optimized more aggressively
- Because this is most innovative aspect of HotSpot, it is main topic of HotSpot papers.

Automatic memory management

- Memory in heap consists of objects containing pointers to other objects.
- Objects in heap are accessed in program by using pointers stored in local variables, which are on stack.
- Therefore, only heap objects that matter are *reachable* either directly from stack, or from fields of other reachable heap objects
- Objects that are not reachable are called *garbage*.
- **Automatic memory management attempts to make garbage cells available for allocation.**

Creation of garbage

- Example:

```
let f n y = let x = numbers 1 n  (* list [1;2;…;n] *)
                   in x@y
```

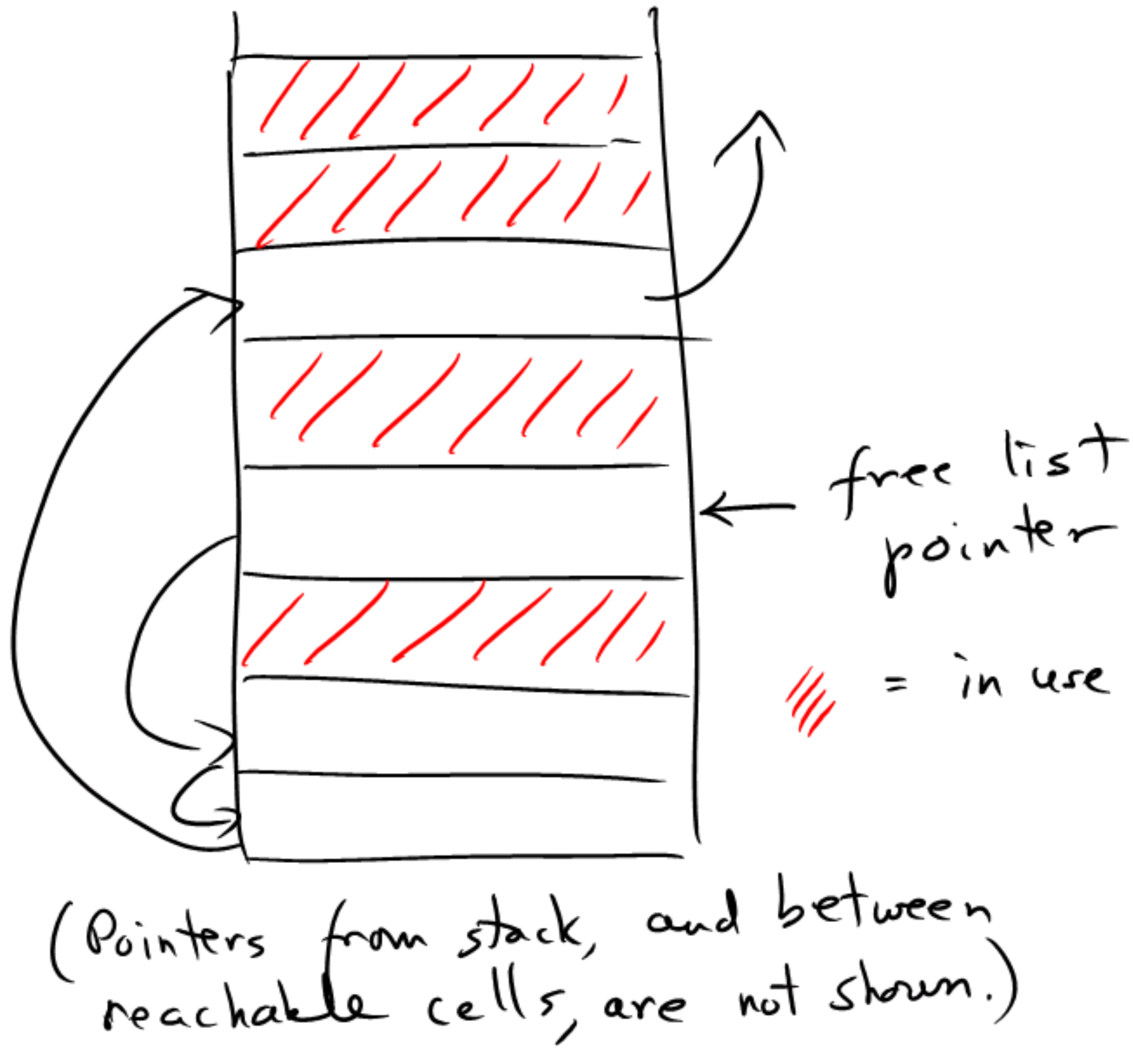
Creates n "cons cells" of garbage, because x@y makes a copy of x.

Representing free memory

- Alternatives: **free list** or **free area**
  - Free list: Free memory is placed on a linked list. Request for memory iterates over list looking for big enough memory area.
  - Free area: Unused area of memory reserved for allocation. Memory allocated from bottom of this area.
- Will discuss free list representation first

History of a heap object, using "free list" system

Heap contains data that have been allocated and data on free list



History of a heap object, using “free list” system

Program executes:

x = new C(); *or*

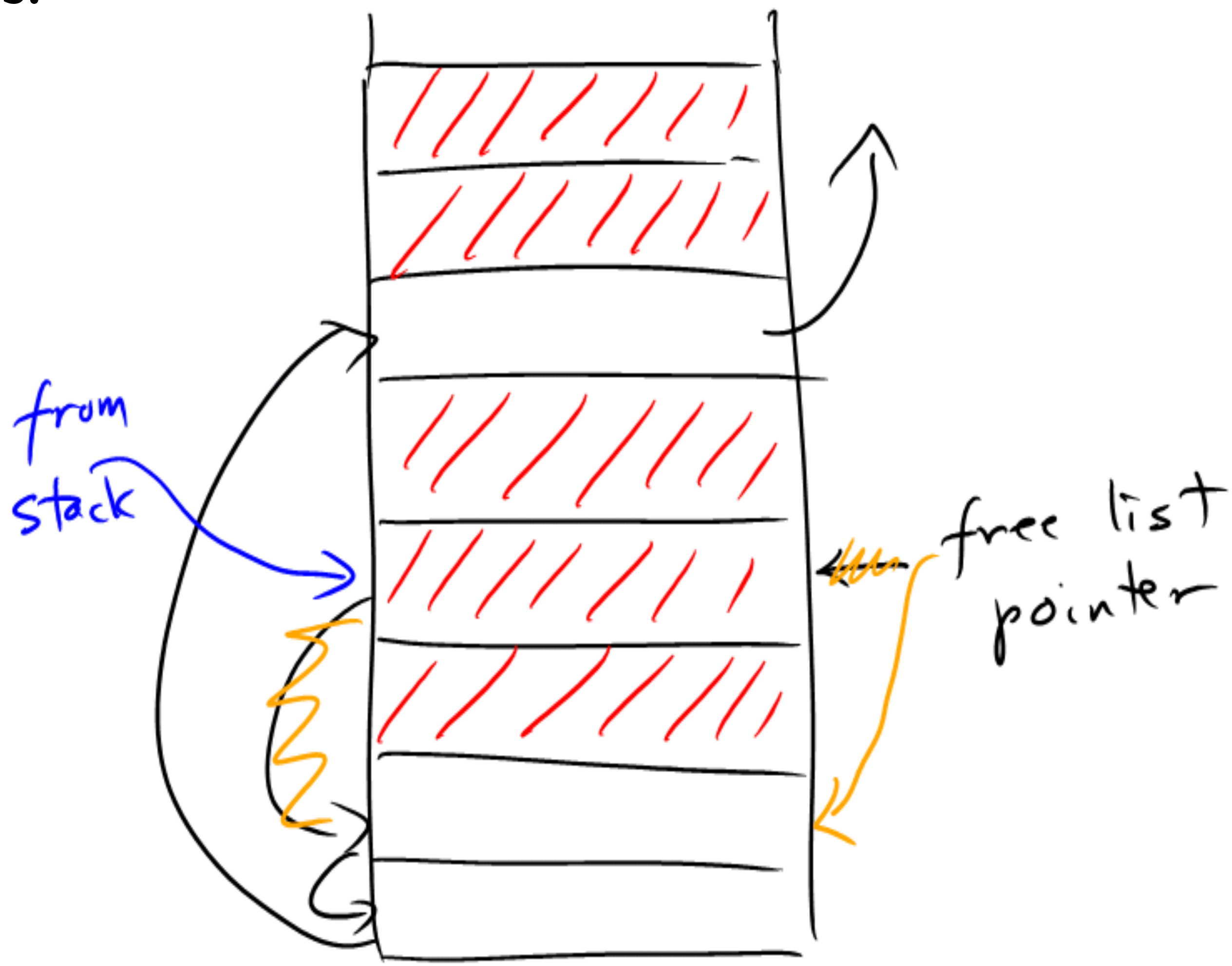
x = malloc(); *or*

x = a::b

(x a local

variable of

function f)

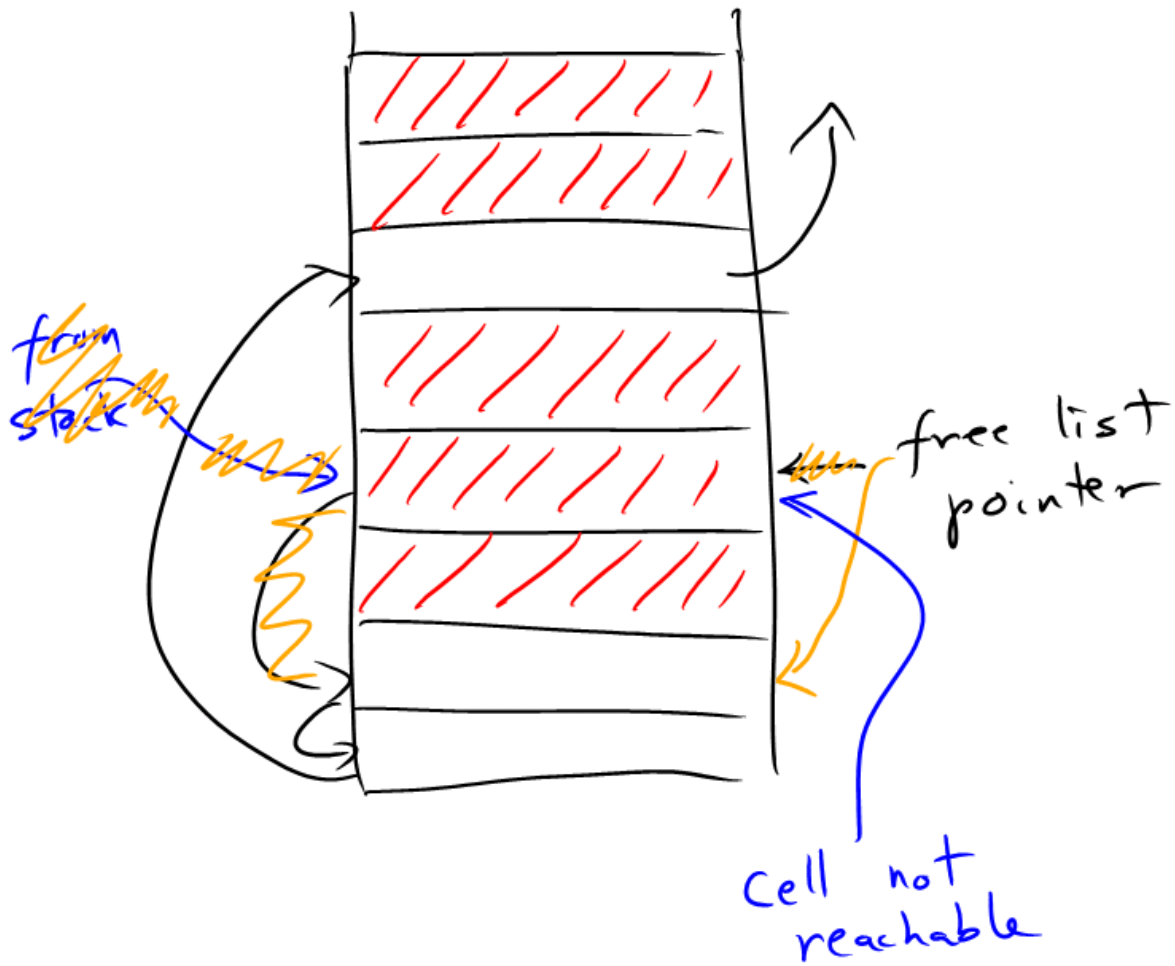


History of a heap object, using "free list" system

Return from f.

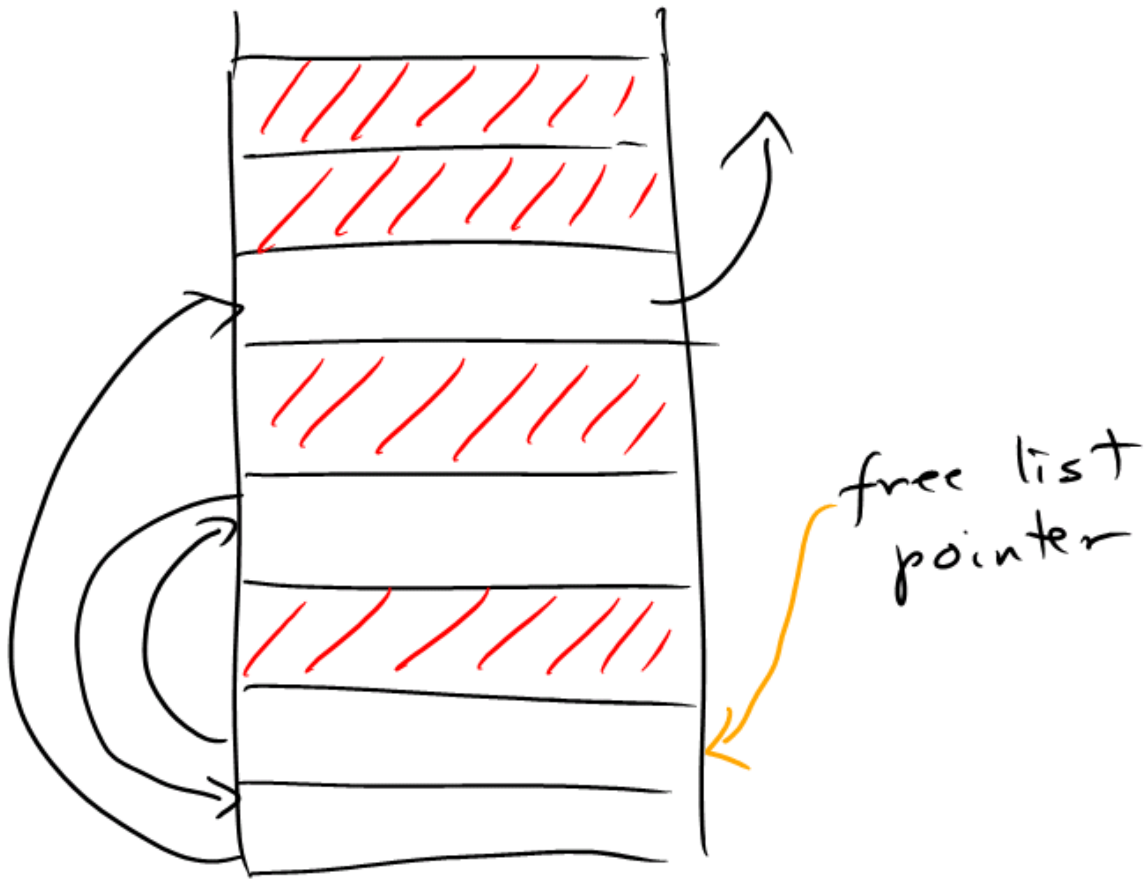
Assume no other objects point to the new object.

New object no longer reachable (but not allocatable either)



History of a heap object, using "free list" system

Eventually, object is returned to free list.



Three types of memory cells

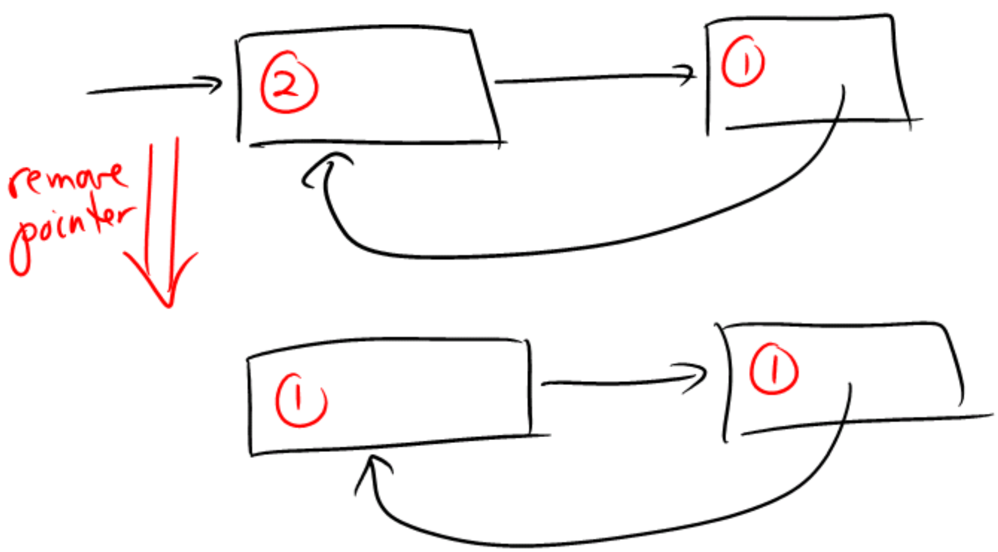
- *Allocatable* – i.e on free list
  - Initially contains all cells
- *Reachable*
  - Obtained by request for heap memory
  - Still reachable from stack (possibly via other heap objects)
- *Neither*
  - Once reachable, now not – e.g. was reachable from a local variable of function f, but have returned from f
  - Was not returned to free list
- "Neither" category is most interesting – memory could be made allocatable.

Reference counting

- Use free list
- Track number of pointers to every object
- Adjust count each time a pointer is copied/assigned
  - "p = q": Increment refcnt(*q)
    Decrement refcnt(*p)
    if refcnt(*p)=0 then return *p to free list
    and decrement refcnt of all
    objects that *p points to
- *All objects go to free list as soon as they are non-reachable – no "neither" category*

Reference counting (cont.)

- Advantages
  - Cost spread out over computation
- Disadvantages
  - Cannot easily handle cycles among objects (which occur a lot)



Garbage collection

- Two methods
  - Non-copying (mark-and-sweep)
    - Uses free list representation
  - Copying
    - Uses free area representation
- Unlike reference-counting:
  - Cells go into "neither" category temporarily
  - Are recovered all at once
  - Costs vary according to method, but happen all at once – "g.c. pause" – and are not amortized

Non-copying garbage collection

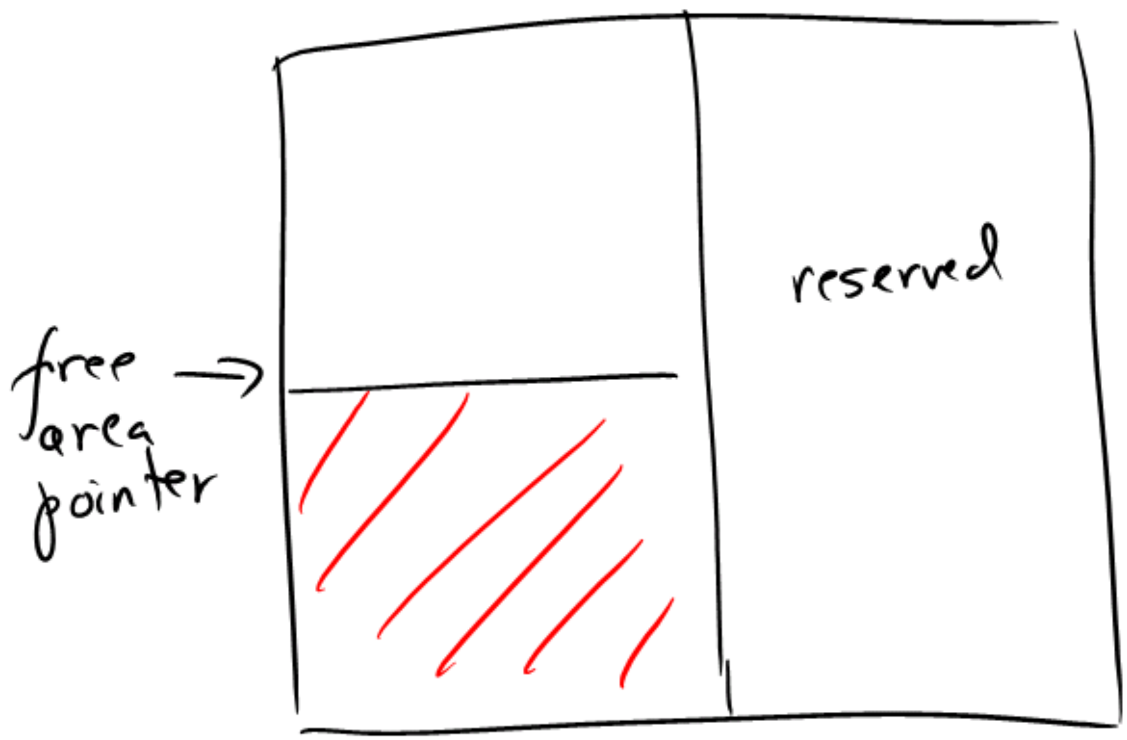
- Use free list
- Reserve one bit in each object header, called the "reachable" bit
- Start with reachable bit zero in every header
- Traverse *reachable* data, setting reachable bit
- Iterate over *entire heap*. If reachable bit is 1, reset it; if it is zero, place that memory chunk on free list
- *Observations*
  - Reachable data is not moved
  - Reachable data remains spread across memory
  - Cost is linear in total size of heap

Copying garbage collection

- Use free area
- Half of memory is reserved (!); all allocation happens in other half, called half-in-use.
- Half-in-use is divided into used area and free area
- Allocate memory from bottom of free area. When free area is exhausted, do g.c.
- G.C.: Traverse reachable object, moving them to reserved area and adjusting all pointers. Reserved area now becomes half-in-use. Free area is area on top of moved objects.

History of a heap object, using "free area" system

Heap contains data that have been allocated – some reachable, some not



History of a heap object, using "free area" system

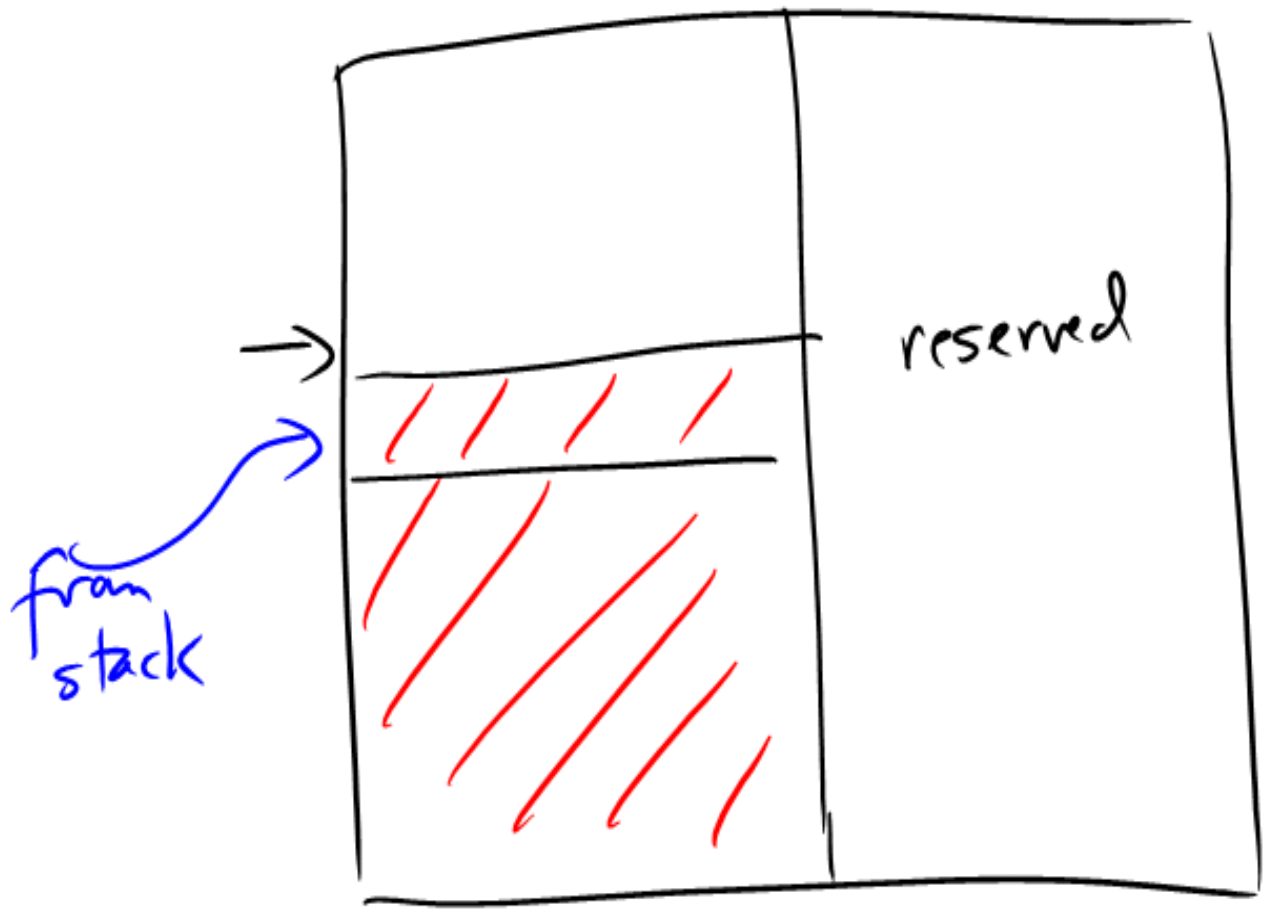
Program executes:

x = new C(); *or*

x = malloc(); *or*

x = a::b

(x a local variable of function f)

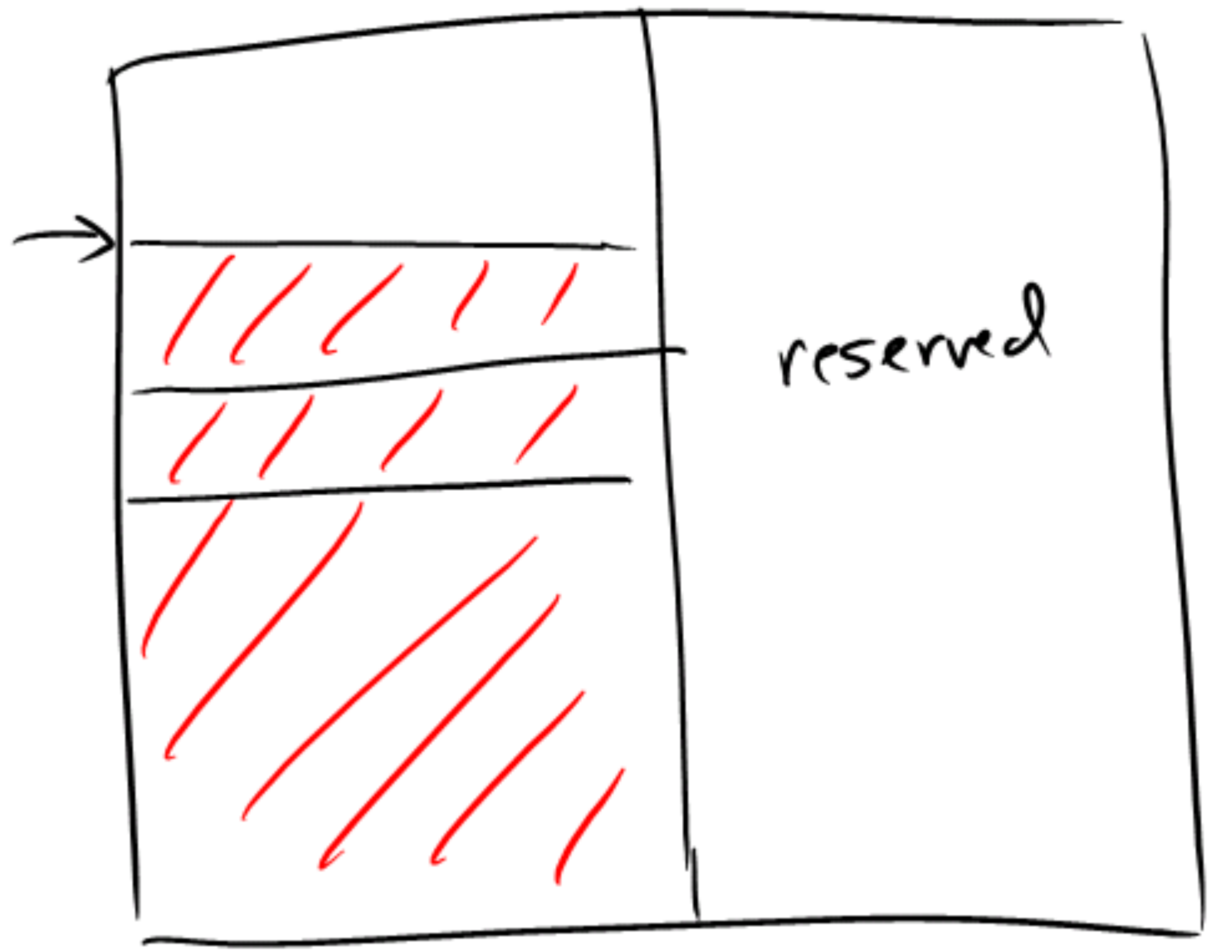


History of a heap object, using "free area" system

Return from f.

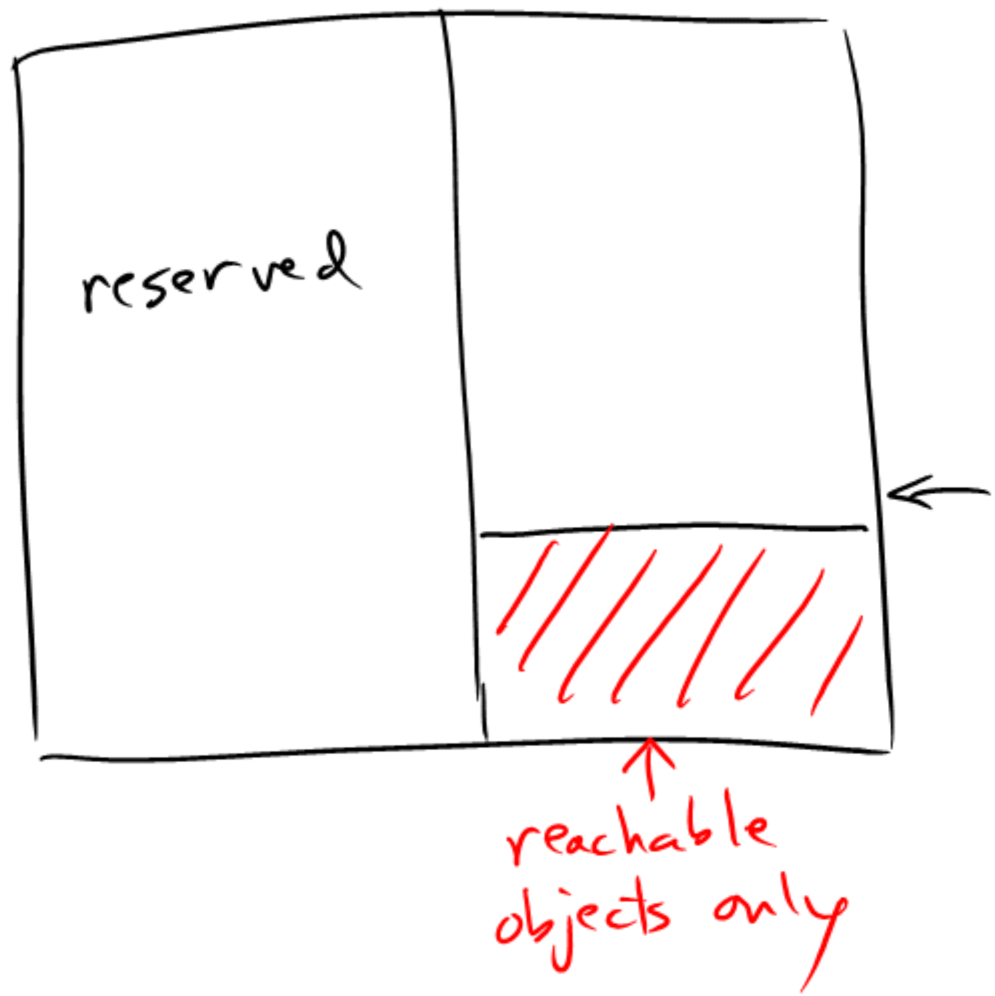
Assume no other objects point to the new object.

New object no longer reachable (but not allocatable either)



History of a heap object, using "free area" system

Eventually, g.c. is done and moves reachable data to reserved memory area.



Copying garbage collection (cont.)

- *Observations*
  - Data is moved; all pointers must be adjusted
    - Works only if garbage collector knows which values are pointers.
  - Reachable data are compressed
  - Cost is linear in size of *reachable* data
  - Traversal normally done breadth-first

Generational garbage collection

- Variant of copying collector
- Most data either long-lived or short-lived
- Both methods described spend a lot of time traversing and/or copying *long-lived* data
- To avoid this, divide memory into *four* spaces:
  - Young-in-use
  - Young reserved
  - Old-in-use
  - Old reserved
- Start allocating from young-in-use, proceed as for regular copying g.c.

Generational garbage collection (cont.)

- When a g.c. does not succeed in recovering memory for young space, move data from young space to old-in-use. Continue to allocate from young-in-use.
- When old-in-use fills up, copy to old reserve.
- *Observations*
  - Copying of old space a rare event
  - GC in young space inexpensive because most young memory is garbage
  - Can extend idea to more than two "generations"

Java HotSpot run-time system g.c.

- HotSpot uses two-generation collector
- Young generation uses copying collector
- Old generation uses mark-and-compact method – compact in place