Reference Counting

- Each object has an associated count of the references (pointers) to it.
  - This field is usually in the object's hidden header.
  - Each time a reference to the object is created (e.g., a pointer copy), the object's count is incremented.
  - When an existing reference to an object is eliminated, the count is decremented.
  - Notice that this requires collaboration from the compiler to generate the necessary code for both incrementing and decrementing the count.

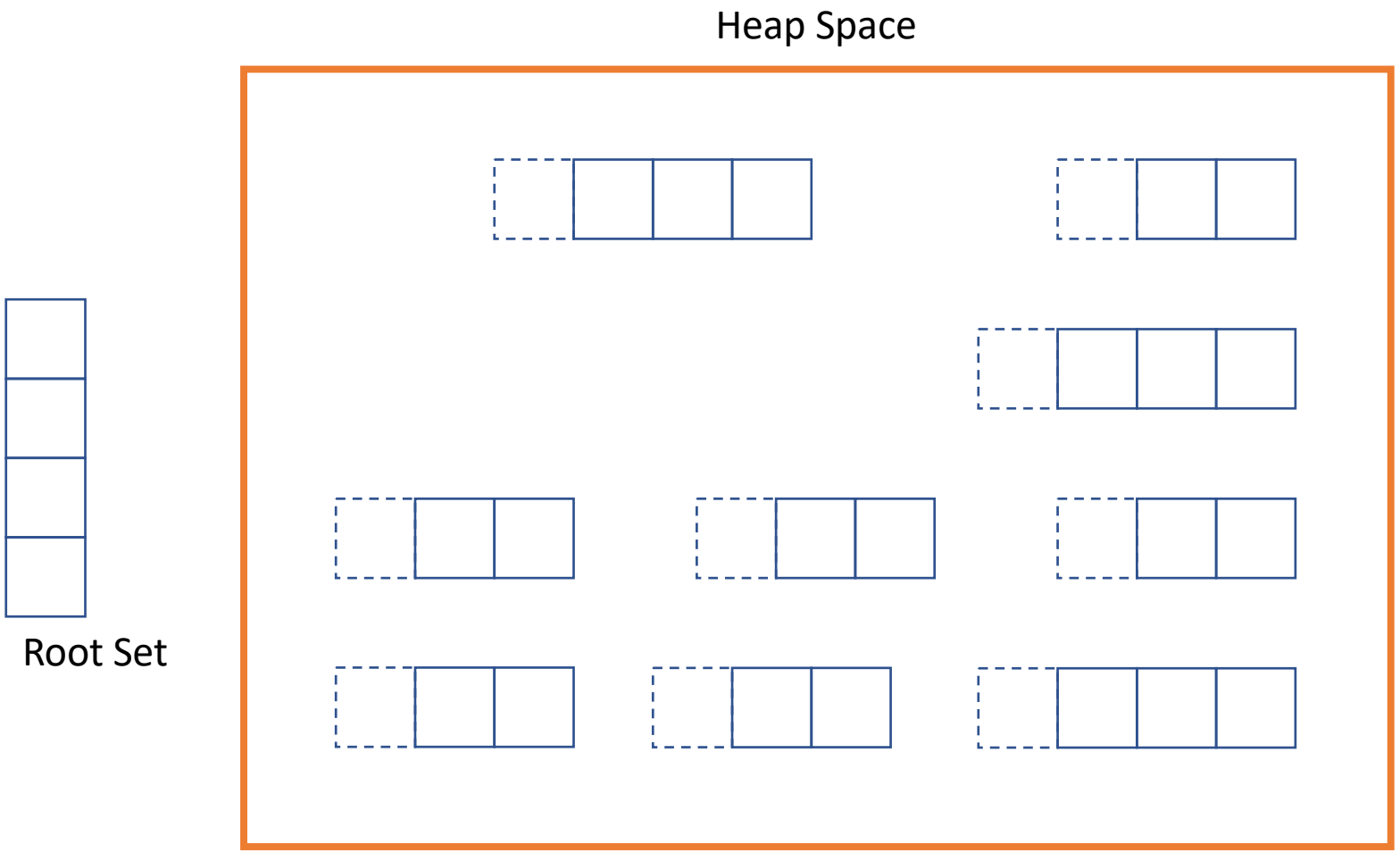
Reference Counting

- Each object has an associated count of the references (pointers) to it.
  - This field is usually in the object's hidden header.
  - Each time a reference to the object is created (e.g., a pointer copy), the object's count is incremented.
  - When an existing reference to an object is eliminated, the count is decremented.
  - Notice that this requires collaboration from the compiler to generate the necessary code for both incrementing and decrementing the count.
- An object is considered to be garbage and subject to reclamation when its count equals zero.

Reference Counting

- Each object has an associated count of the references (pointers) to it.
  - This field is usually in the object's hidden header.
  - Each time a reference to the object is created (e.g., a pointer copy), the object's count is incremented.
  - When an existing reference to an object is eliminated, the count is decremented.
  - Notice that this requires collaboration from the compiler to generate the necessary code for both incrementing and decrementing the count.
- An object is considered to be garbage and subject to reclamation when its count equals zero.
- When an object is reclaimed, its pointer fields are examined, and any objects it holds pointers to also have their reference counts decremented.
  - This may result in a cascade of count decrements and reclamations.

Reference Counting: An Example



Mapping To The Two-Phase Abstraction

- Garbage detection, i.e., distinguishing live objects from garbage objects through some means.
  - The adjustment and checking of reference counts implements this phase.

Mapping To The Two-Phase Abstraction

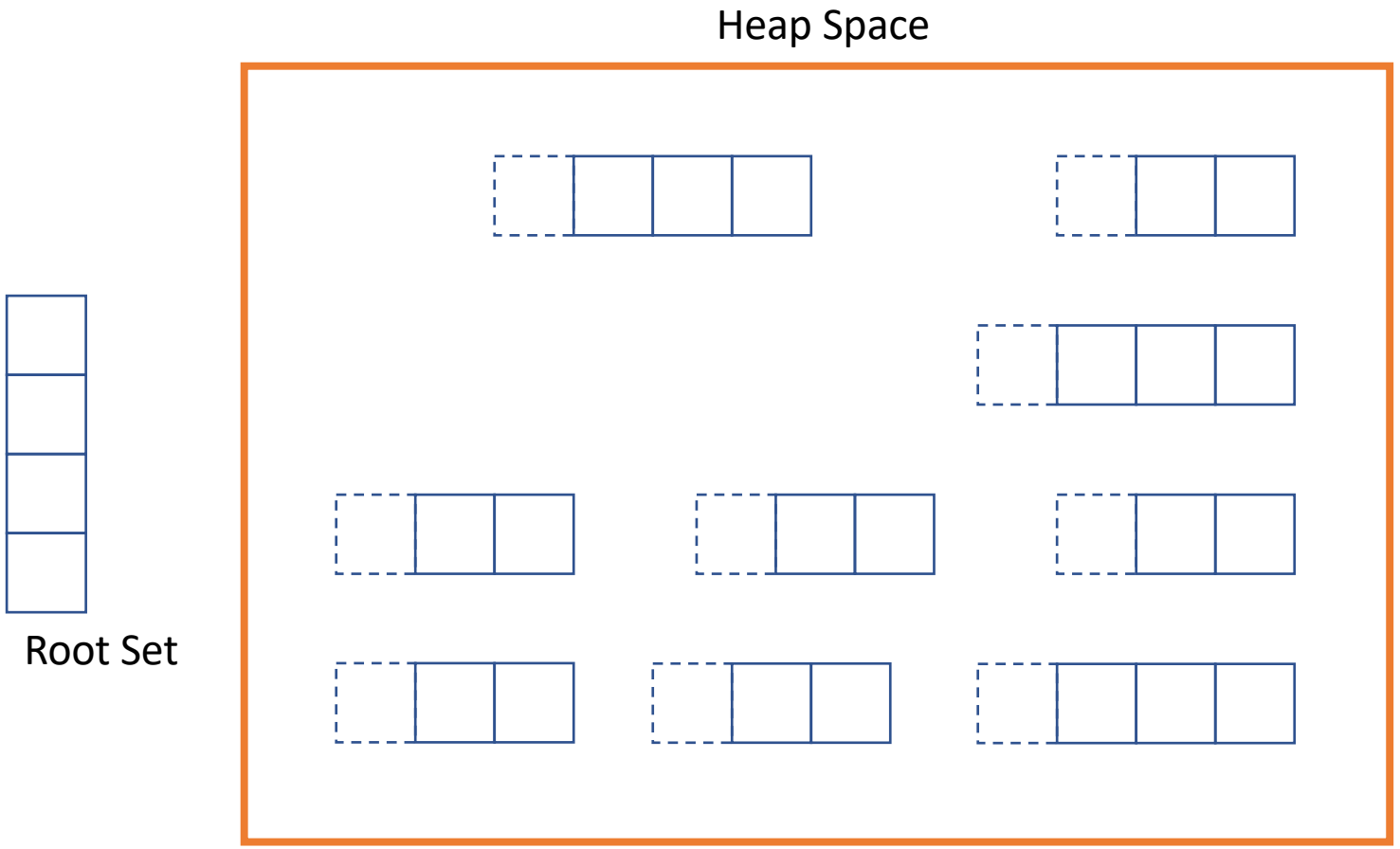
- Garbage detection, i.e., distinguishing live objects from garbage objects through some means.
  - The adjustment and checking of reference counts implements this phase.
- Storage reclamation, i.e., reclaiming the storage of the garbage objects so that it is available for use by the mutator.
  - Happens when the reference count reaches zero.

Mapping To The Two-Phase Abstraction

- Garbage detection, i.e., distinguishing live objects from garbage objects through some means.
  - The adjustment and checking of reference counts implements this phase.
- Storage reclamation, i.e., reclaiming the storage of the garbage objects so that it is available for use by the mutator.
  - Happens when the reference count reaches zero.
- Operations are interleaved with the execution of the mutator.
  - Reference count adjustments are intrinsically incremental in nature.
  - Can also be made real time, i.e., performing at most a small and bounded amount of work per unit of mutator execution.
  - The transitive reclamation of objects can be deferred by keeping a list of freed objects whose reference counts have become zero but which haven't yet been processed.

The Effectiveness Problem: Cycles

- Reference counting fails to reclaim circular structures.
  - If the pointers in a set of objects create a directed cycle, the reference counts of the objects are never reduced to zero, even if there is no path to the objects from the root set.



The Effectiveness Problem: Cycles

- Reference counting fails to reclaim circular structures.
  - If the pointers in a set of objects create a directed cycle, the reference counts of the objects are never reduced to zero, even if there is no path to the objects from the root set.
- This is a fundamental limitation of reference counting.
  - The problem arises because reference counting determines a conservative approximation of true liveness.
  - Systems using reference counting usually include some other kind of garbage collector as well to handle this problem.

The Efficiency Problem

- The cost of reference counting is that its cost is generally proportional to the amount of work done by the mutator, with a fairly large constant of proportionality.
  - The count of an object must be adjusted when a pointer to it is created or destroyed.
  - If a variable's value is changed from one pointer to another, the counts of two objects must be adjusted.
  - Reclaiming an object when its reference count reaches zero incurs overheads identical to those for explicit deallocation.
  - Short-lived stack variables (e.g., method arguments) can incur a great deal of overhead, especially in leaf methods.

The Efficiency Problem

- The cost of reference counting is that its cost is generally proportional to the amount of work done by the mutator, with a fairly large constant of proportionality.
  - The count of an object must be adjusted when a pointer to it is created or destroyed.
  - If a variable's value is changed from one pointer to another, the counts of two objects must be adjusted.
  - Reclaiming an object when its reference count reaches zero incurs overheads identical to those for explicit deallocation.
  - Short-lived stack variables (e.g., method arguments) can incur a great deal of overhead, especially in leaf methods.
- Deferred reference counting avoids adjusting reference counts for short-lived pointers from the stack.
  - Reference counts are now not a safe indicator of garbage.
  - Reference counts must be updated periodically by considering roots from the stack frames. This period can be tuned.

The Efficiency Problem

- The cost of reference counting is that its cost is generally proportional to the amount of work done by the mutator, with a fairly large constant of proportionality.
  - The count of an object must be adjusted when a pointer to it is created or destroyed.
  - If a variable's value is changed from one pointer to another, the counts of two objects must be adjusted.
  - Reclaiming an object when its reference count reaches zero incurs overheads identical to those for explicit deallocation.
  - Short-lived stack variables (e.g., method arguments) can incur a great deal of overhead, especially in leaf methods.
- Deferred reference counting avoids adjusting reference counts for short-lived pointers from the stack.
  - Reference counts are now not a safe indicator of garbage.
  - Reference counts must be updated periodically by considering roots from the stack frames. This period can be tuned.
- A reference counting system may perform with little degradation when almost all of the heap space is occupied by live objects.