# CS107: Memory Management

**Guannan Wei**
guannan.wei@tufts.edu

April 9, 2026

Spring 2026

Tufts University

# Last time

Last time:

- Reference counting
- Mark & sweep GC
- Copying GC

Today:

- Cheney's copying GC
- Generational GC
- Other variants: incremental, concurrent, or parallel GC

# Allocation In A Copying GC

In a copying GC, there is no free list to maintain, and no search to perform in order to find a free block.

Memory is allocated *linearly* in from-space.

# Allocation In A Copying GC

In a copying GC, there is no free list to maintain, and no search to perform in order to find a free block.

Memory is allocated *linearly* in from-space.

- Allocation in a copying GC is therefore very fast - as fast as stack allocation.
- All that is required is a pointer to the border between the allocated and free area of from space.

# Forwarding Pointers

Before copying an object, a check must be made to see whether it has already been copied. If this is the case, it must not be copied again. Rather, the already-copied version must be used.

How can this check be performed?

# Forwarding Pointers

Before copying an object, a check must be made to see whether it has already been copied. If this is the case, it must not be copied again. Rather, the already-copied version must be used.

How can this check be performed?

- By storing a **forwarding pointer** in the object in from-space, after it has been copied.
- Since the object in from-space is not needed anymore, its memory can be reused to store the forwarding pointer.

# Cheney's Copying GC

The copying GC algorithm presented before does a depth-first traversal of the reachable graph. When it is implemented using recursion, it can lead to stack overflow.

How to avoid using worklist/queue/stack?

# Cheney's Copying GC

The copying GC algorithm presented before does a depth-first traversal of the reachable graph. When it is implemented using recursion, it can lead to stack overflow.

How to avoid using worklist/queue/stack?

**Cheney's copying GC** is an elegant GC technique that does a breadth-first traversal of the reachable graph, requiring only one pointer as additional state.

# Cheney's Copying GC

In any breadth-first traversal, one has to remember the set of nodes that have been visited, but whose children have not been.

The basic idea of Cheney's algorithm is to use to-space to store this set of nodes, and tracks the status using two pointers in to-space:

# Cheney's Copying GC

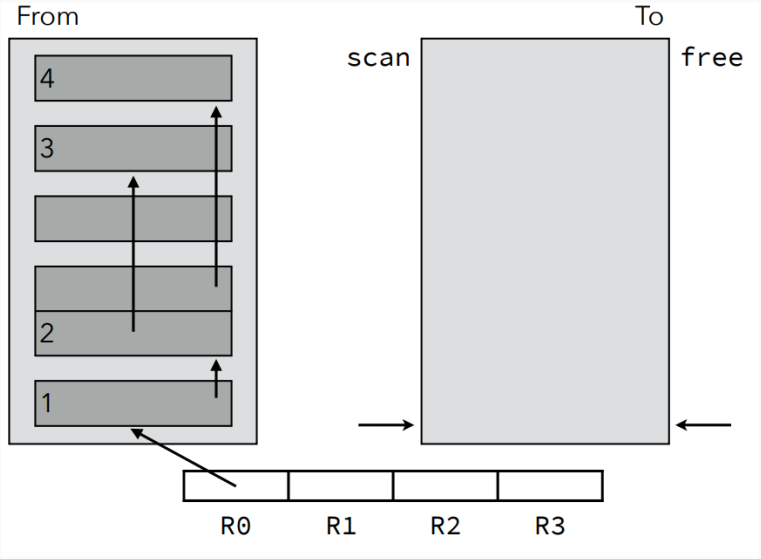
In any breadth-first traversal, one has to remember the set of nodes that have been visited, but whose children have not been.

The basic idea of Cheney's algorithm is to use to-space to store this set of nodes, and tracks the status using two pointers in to-space:

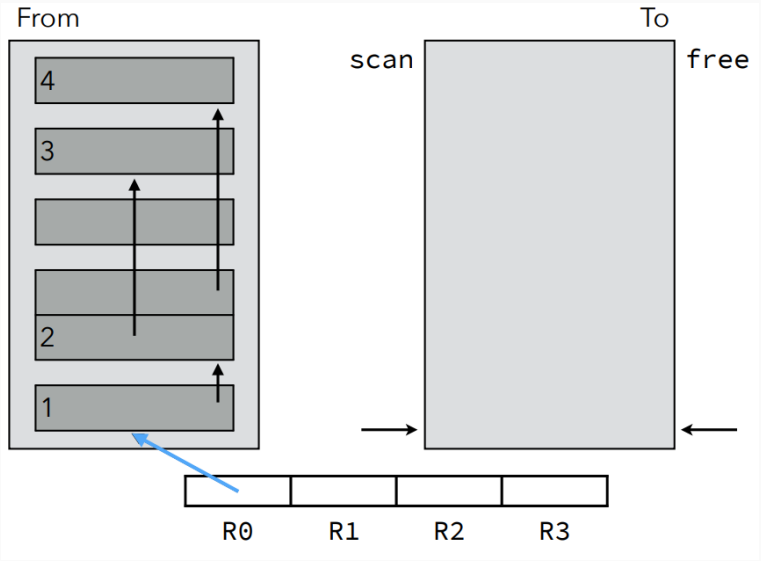
- scan: points to the next copied object whose fields still need to be scanned
- free: points to the end of the copied objects, where the next object will be copied to

The "scan'' pointer partitions to-space in two parts: the nodes whose children have been visited, and those whose children have not been visited.

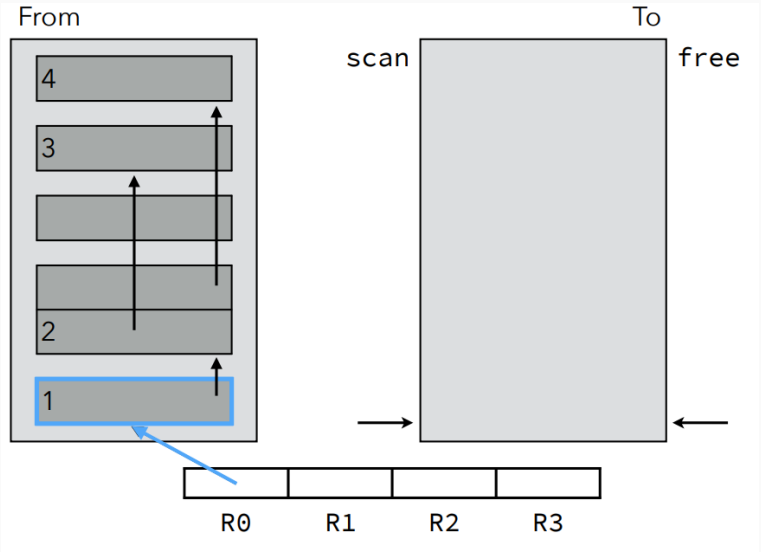
# Cheney's Copying GC



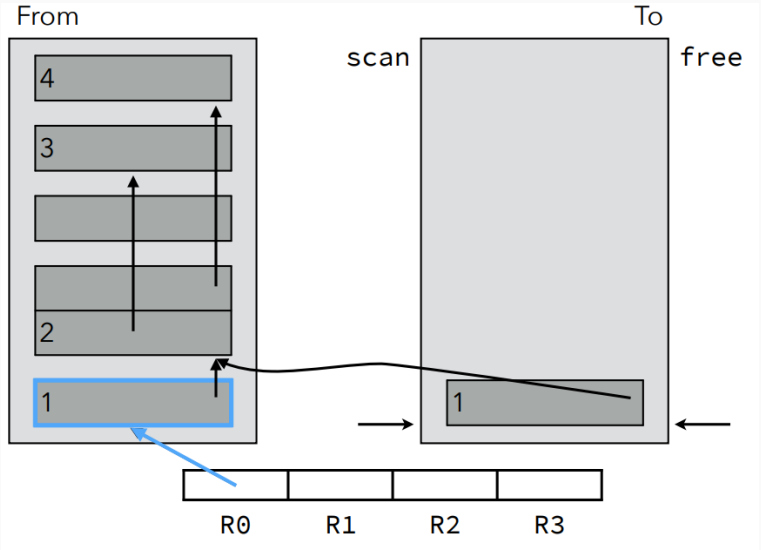
# Cheney's Copying GC



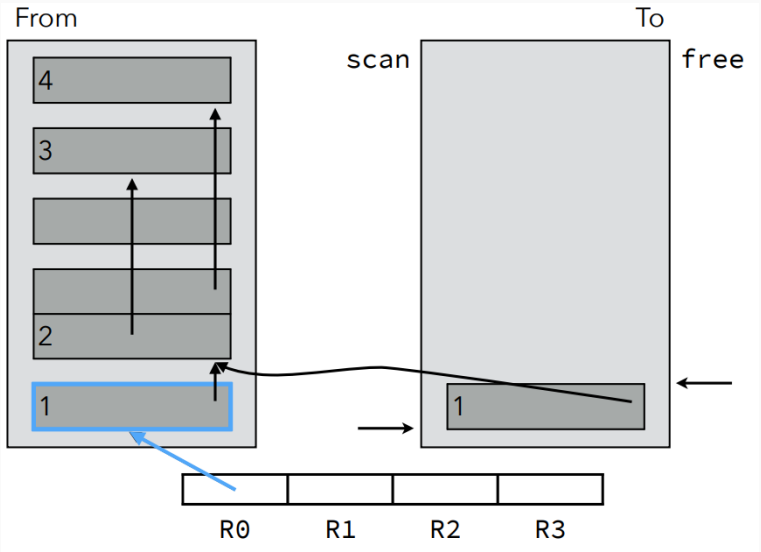
# Cheney's Copying GC



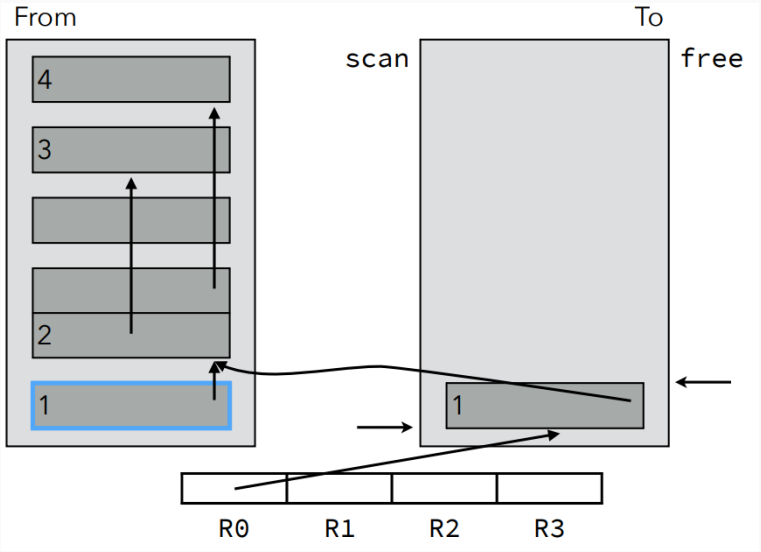
# Cheney's Copying GC



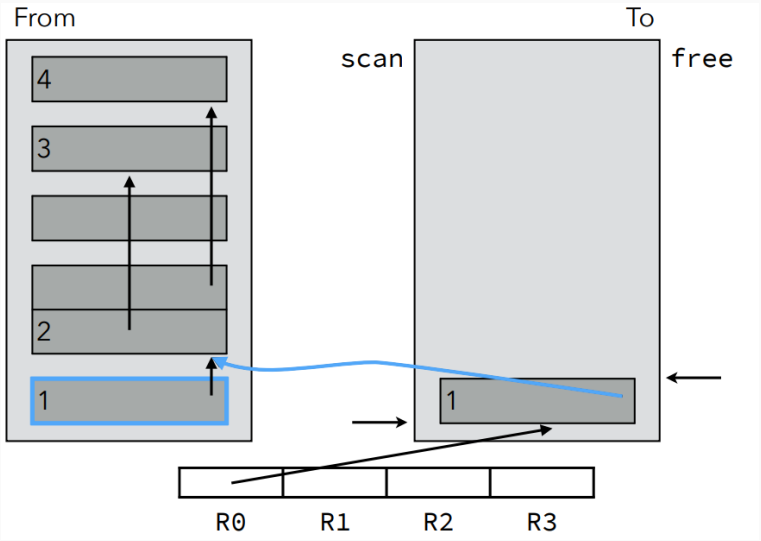
# Cheney's Copying GC



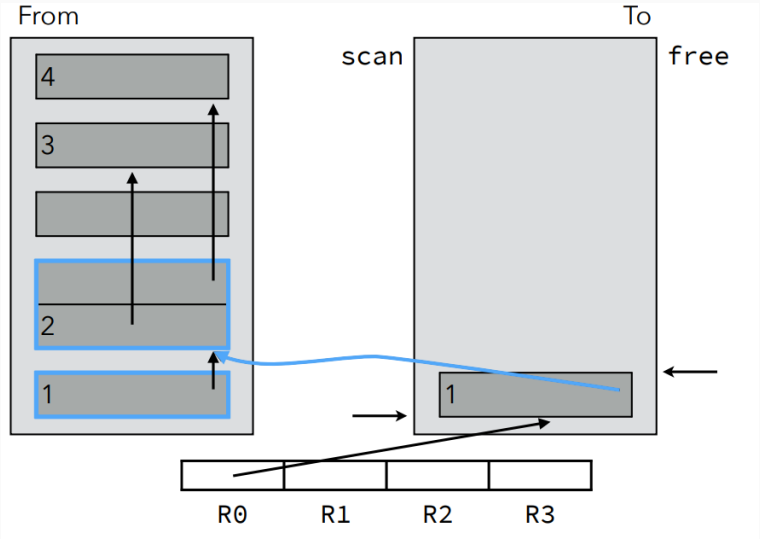
# Cheney's Copying GC



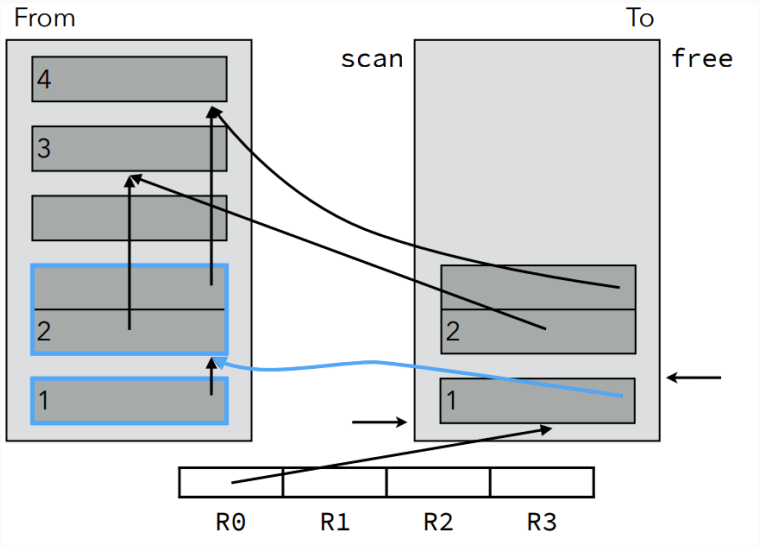
# Cheney's Copying GC



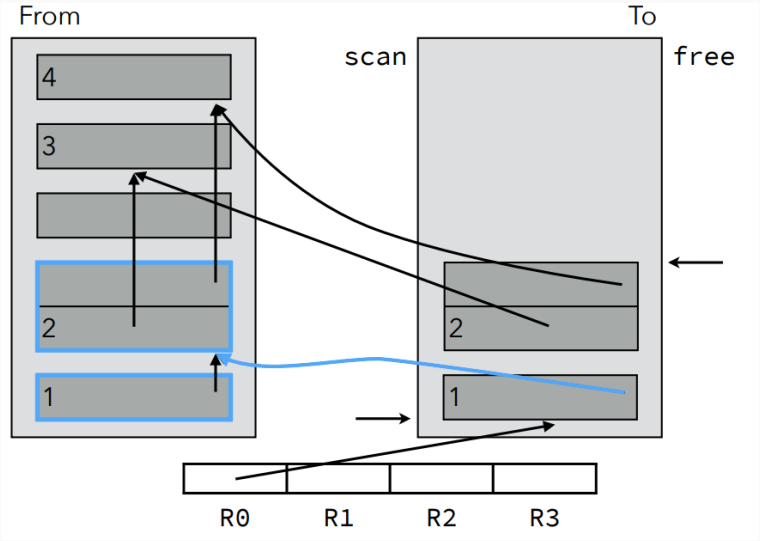
# Cheney's Copying GC



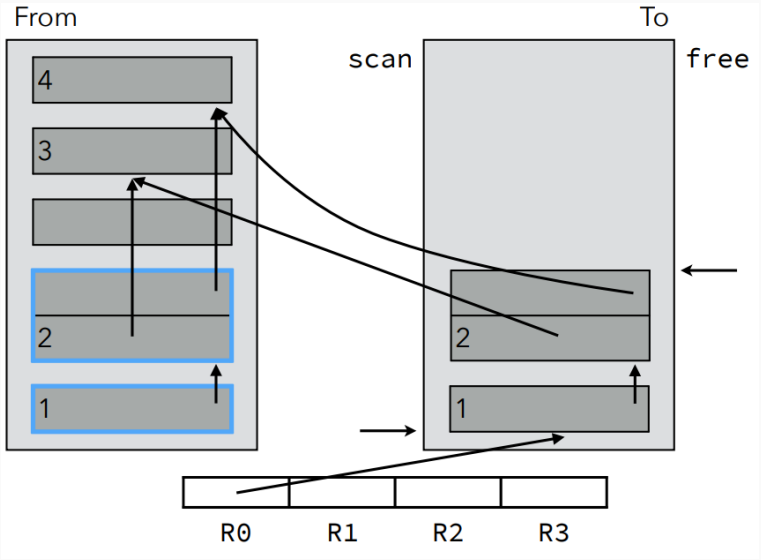
# Cheney's Copying GC



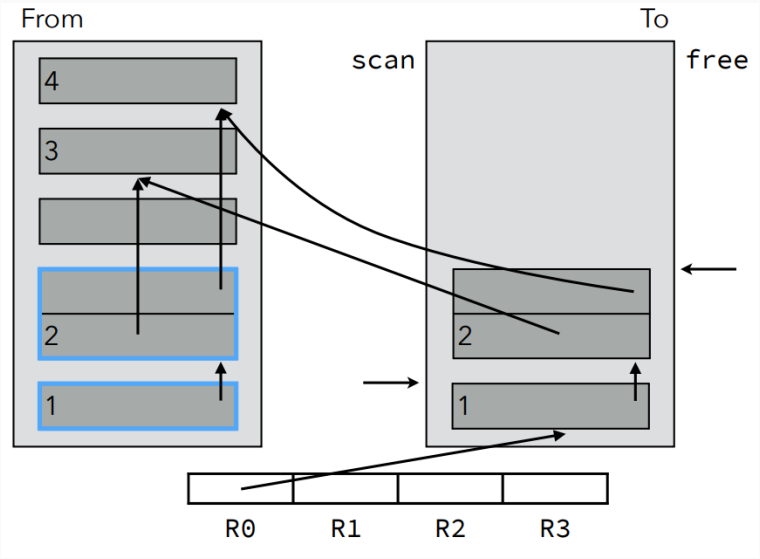
# Cheney's Copying GC



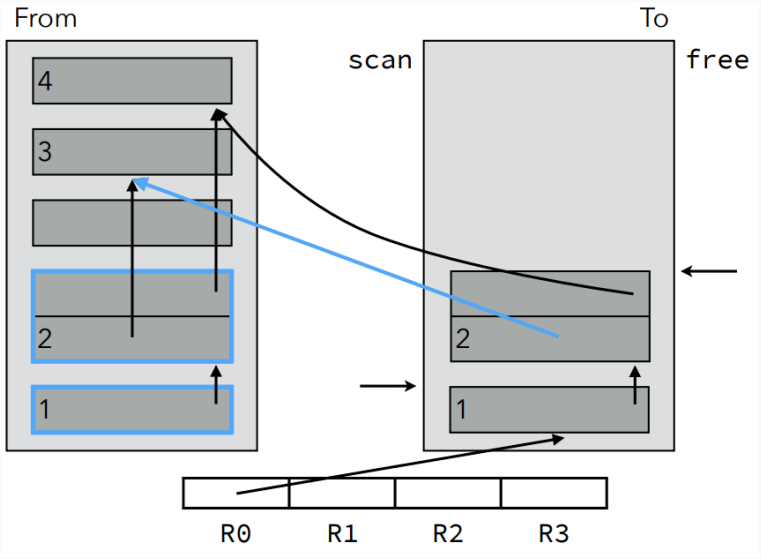
# Cheney's Copying GC



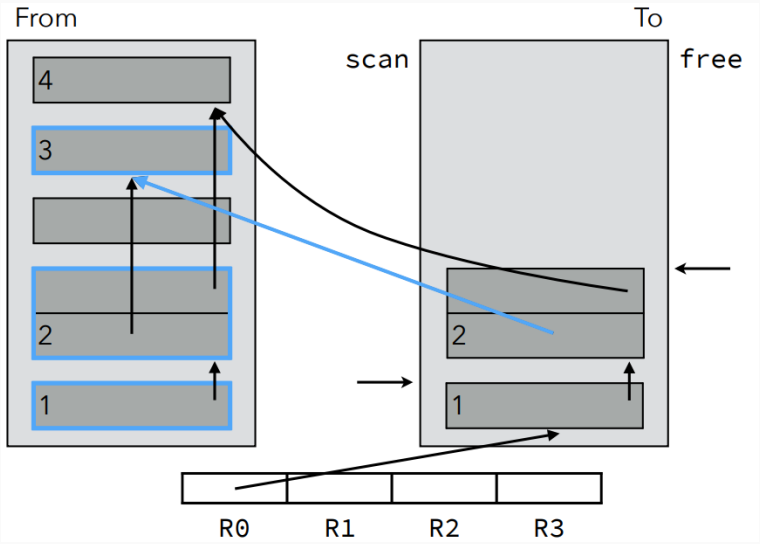
# Cheney's Copying GC



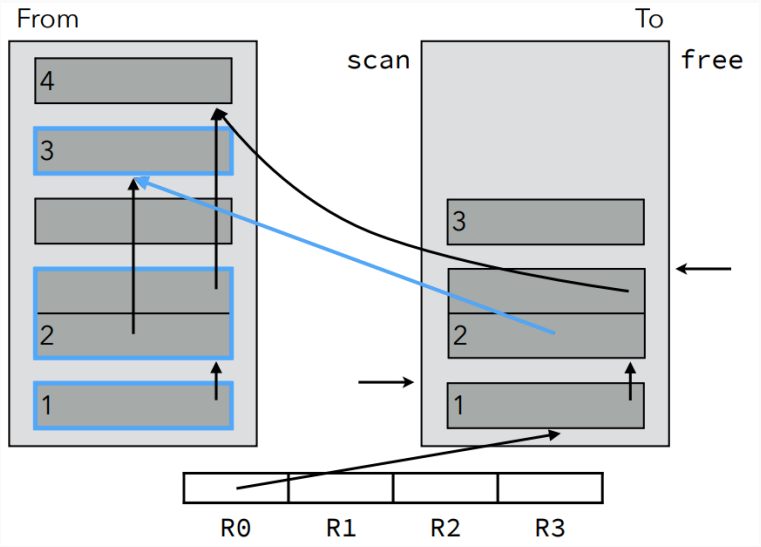
# Cheney's Copying GC



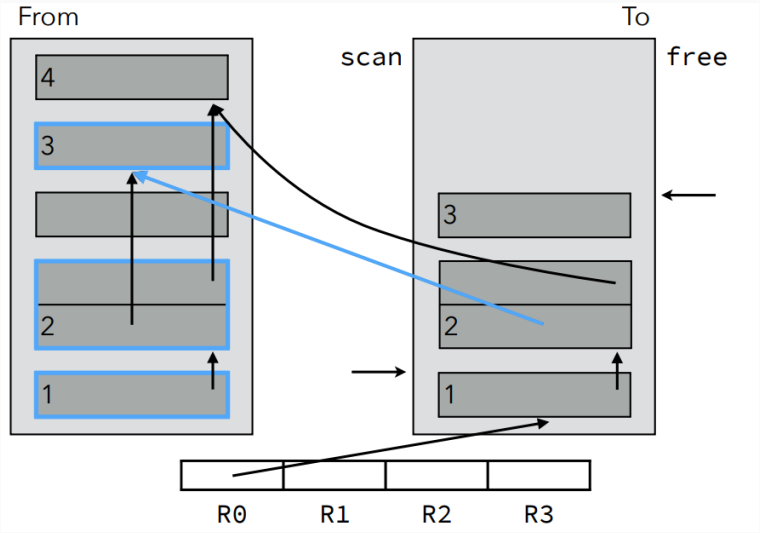
# Cheney's Copying GC



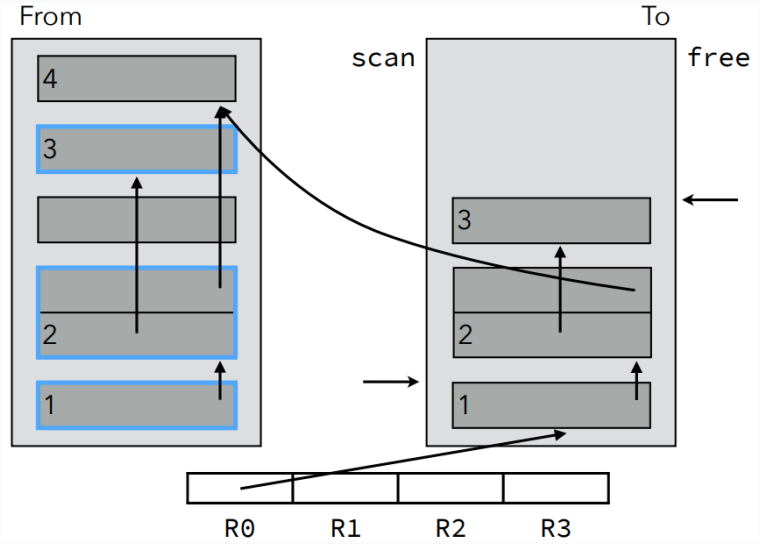
# Cheney's Copying GC



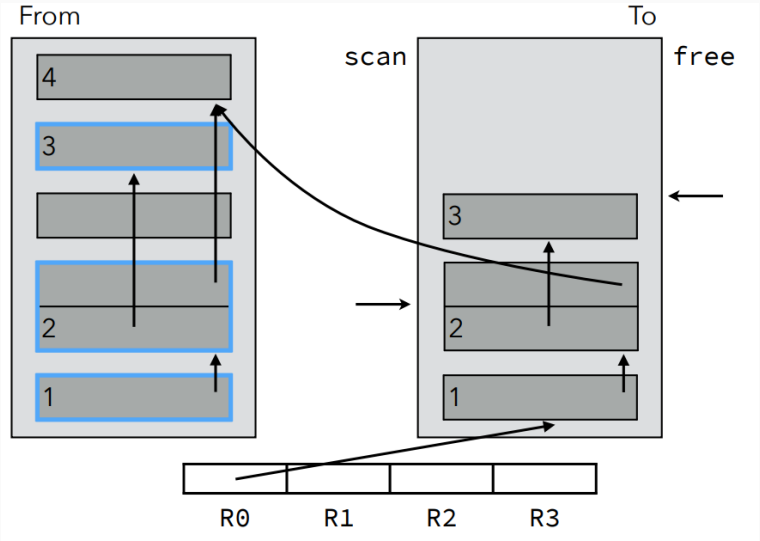
# Cheney's Copying GC



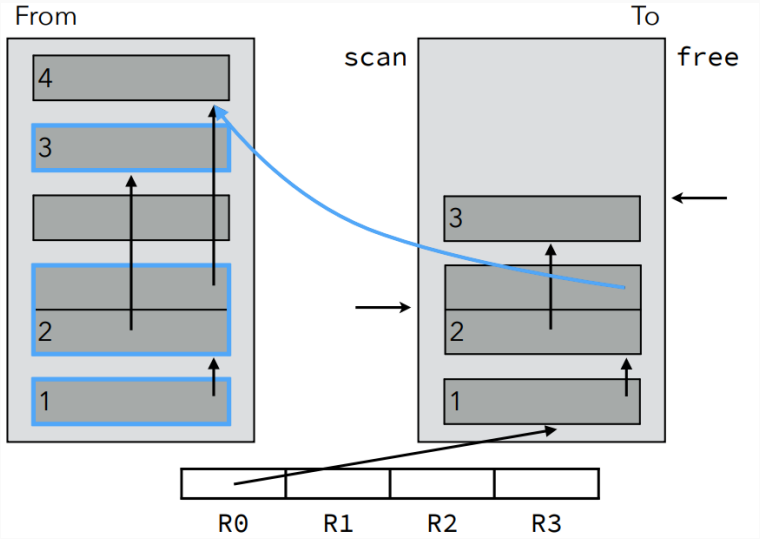
# Cheney's Copying GC



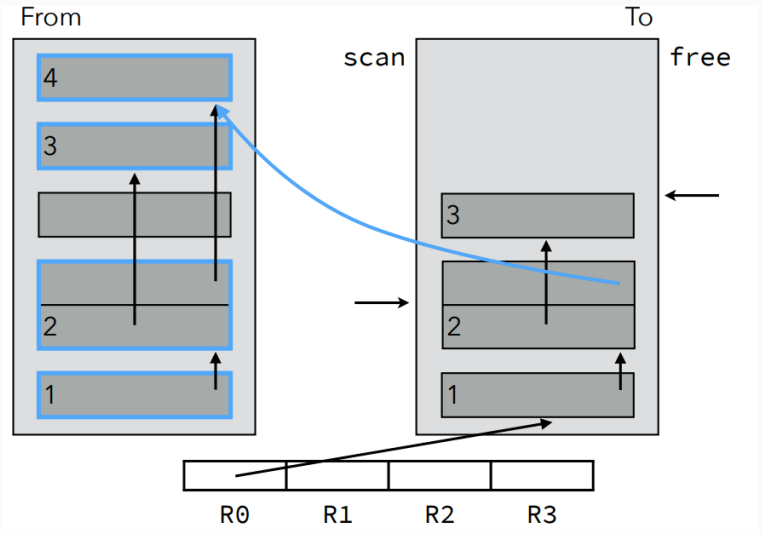
# Cheney's Copying GC



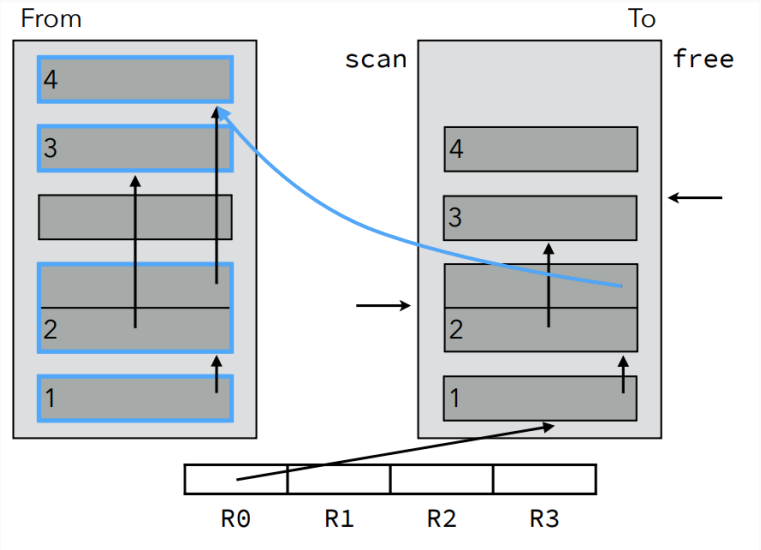
# Cheney's Copying GC



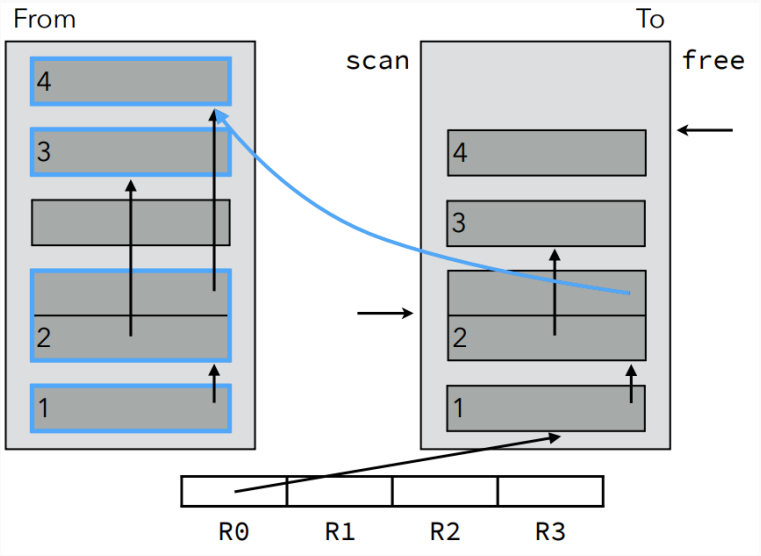
# Cheney's Copying GC



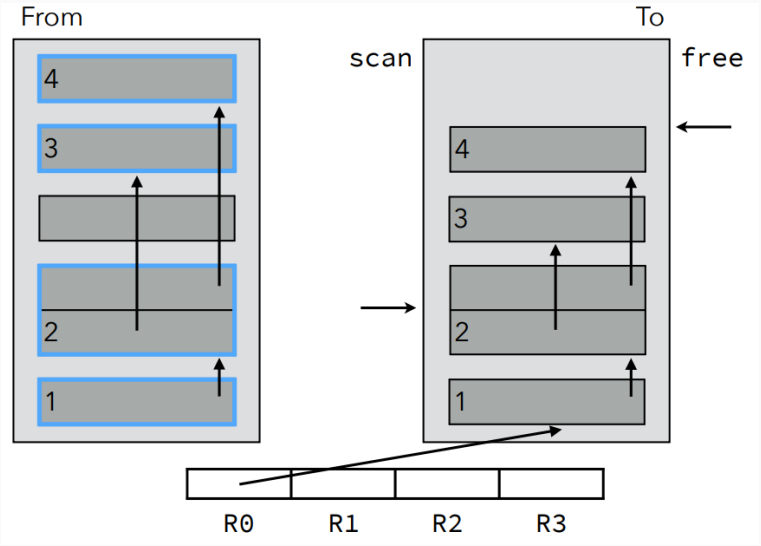
# Cheney's Copying GC



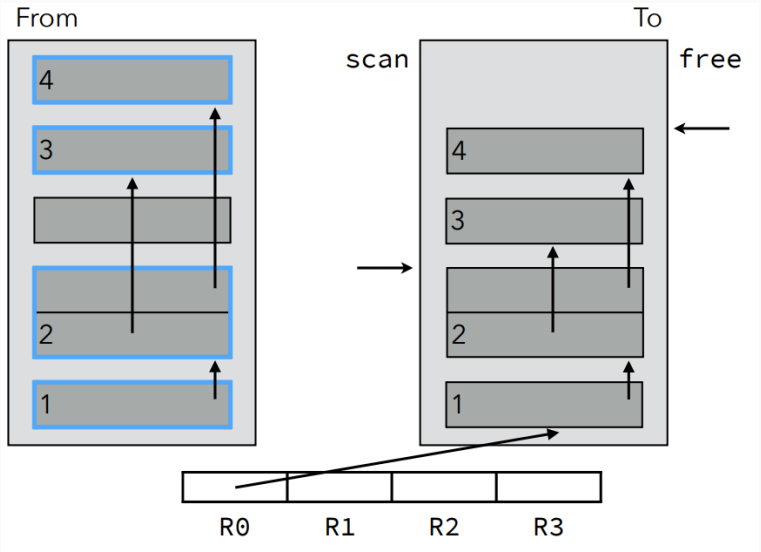
# Cheney's Copying GC



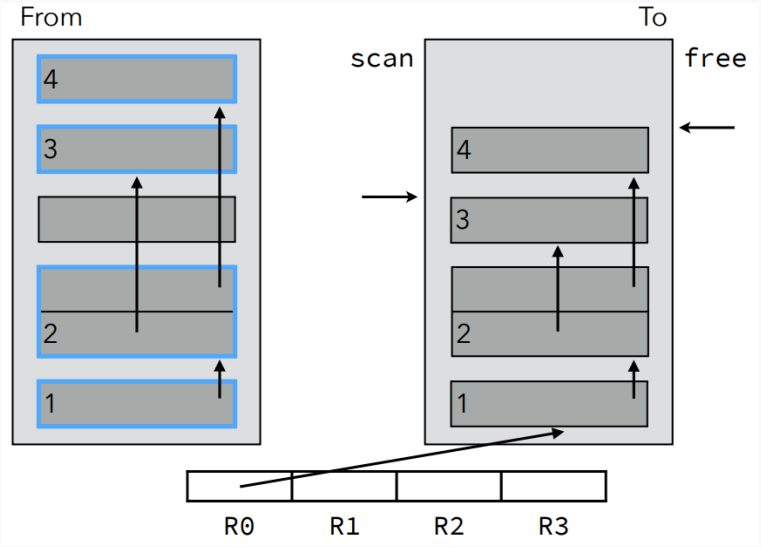
# Cheney's Copying GC



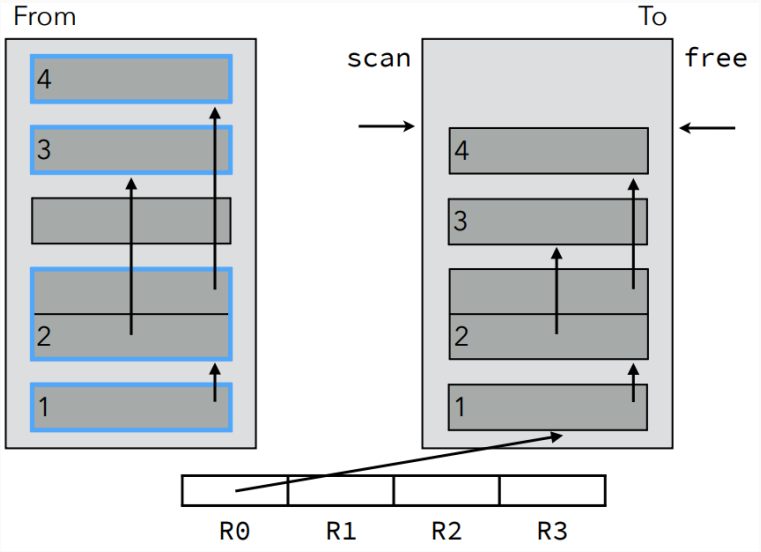
# Cheney's Copying GC



# Cheney's Copying GC



# Cheney's Copying GC



# Cheney's Copying GC

Summary:

- Once we copy an object, we increase the free pointer.
- If we have copied the children of an object, we can move the scan pointer forward.

# Copying vs Mark & Sweep

The pros and cons of copying GC, compared to mark & sweep.

Pros:

- No external fragmentation
- Very fast allocation
- No transversal of dead objects

# Copying vs Mark & Sweep

The pros and cons of copying GC, compared to mark & sweep.

Pros:

- No external fragmentation
- Very fast allocation
- No transversal of dead objects

Cons:

- Uses twice as much (virtual) memory
- Copying can be expensive
- Requires precise identification of pointers

# Copying vs Mark & Sweep

The pros and cons of copying GC, compared to mark & sweep.

Pros:

- No external fragmentation
- Very fast allocation
- No transversal of dead objects

Cons:

- Uses twice as much (virtual) memory
- Copying can be expensive
- Requires precise identification of pointers

Why it requires precise identification of pointers?

# Copying GC and Ambiguous Pointers

- Consider there are ambiguous pointers that look like a heap location, but they are actually just data (e.g. integer values).

```
int x = 0x40000; // x is an integer, but it looks like a pointer to the GC
return x + 1;
```

# Copying GC and Ambiguous Pointers

- Consider there are ambiguous pointers that look like a heap location, but they are actually just data (e.g. integer values).

```
int x = 0x40000; // x is an integer, but it looks like a pointer to the GC
return x + 1;
```

- If we conservatively trace it and copy the object to the new space, we also need to modify the ambiguous pointer to point to the new location, which is wrong since it is not a pointer.

```
int x = 0x80000; // modified by the GC pointing to new location
return x + 1; // now wrong result
```

# Copying GC and Ambiguous Pointers

- Consider there are ambiguous pointers that look like a heap location, but they are actually just data (e.g. integer values).

```
int x = 0x40000; // x is an integer, but it looks like a pointer to the GC
return x + 1;
```

- If we conservatively trace it and copy the object to the new space, we also need to modify the ambiguous pointer to point to the new location, which is wrong since it is not a pointer.

```
int x = 0x80000; // modified by the GC pointing to new location
return x + 1; // now wrong result
```

Copying GC requires precise identification of pointers, which is not always possible - e.g. because of untagged integers in registers.

# Mostly-Copying GC

A copying GC can also be used in situations where not all pointers can be identified unambiguously. This is the idea of **mostly-copying GC**, due to Joel F. Bartlett (1988).

# Mostly-Copying GC

A copying GC can also be used in situations where not all pointers can be identified unambiguously. This is the idea of **mostly-copying GC**, due to Joel F. Bartlett (1988).

In such a GC, objects are partitioned in two classes:

1) those for which some pointers are ambiguous, usually because they appear in the stack or registers,
2) those for which all pointers are known unambiguously.

# Mostly-Copying GC

A copying GC can also be used in situations where not all pointers can be identified unambiguously. This is the idea of **mostly-copying GC**, due to Joel F. Bartlett (1988).

In such a GC, objects are partitioned in two classes:

1) those for which some pointers are ambiguous, usually because they appear in the stack or registers,
2) those for which all pointers are known unambiguously.

Objects of the first class are **pinned**, i.e. left where they are, while the others - the vast majority, generally - are copied as usual.

# Mostly-Copying GC

If the from- and to-spaces are organized as two separate areas of memory, objects cannot be pinned, as the from-space must be completely empty after GC.

Therefore, a mostly-copying GC organizes memory in **pages/blocks** of fixed size, tagged with the space to which they belong.

# Mostly-Copying GC

If the from- and to-spaces are organized as two separate areas of memory, objects cannot be pinned, as the from-space must be completely empty after GC.

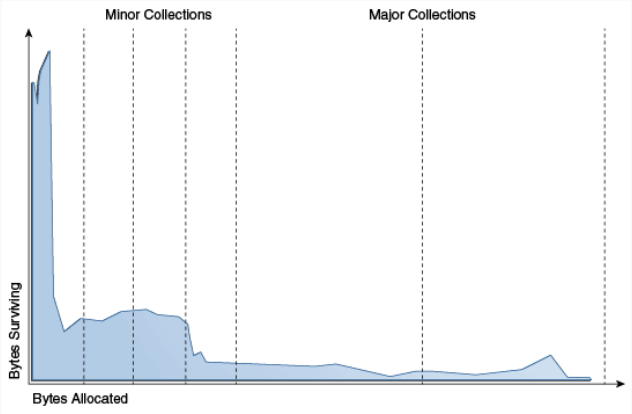
Therefore, a mostly-copying GC organizes memory in **pages/blocks** of fixed size, tagged with the space to which they belong.

Then, during GC :

- pinned objects are left on their page, whose tag is updated to "move" them to to-space,
- other objects are copied (and compacted) as usual.

# GC Technique #4: Generational GC

Empirical observation suggests that a large majority of the objects die young, while a small minority lives for very long.




Source: HotSpot Virtual Machine Garbage Collection Tuning Guide

# GC Technique #4: Generational GC

The idea of **generational garbage collection** is to partition objects in generations based on their age (e.g., young vs old). Then collect the young generation(s) more often than the old one(s).

This should improve the amount of memory collected per objects visited. In a copying GC, this also avoids repeatedly copying long-lived objects.

Note: The principles of generational garbage collection will be presented here in the context of copying GCs, but can also be applied to other GCs like mark & sweep.

# Generational GC

In a generational GC, objects are partitioned into a given number (often 2) of generations. The younger a generation is, the smaller the amount of memory reserved to it.

# Generational GC

In a generational GC, objects are partitioned into a given number (often 2) of generations. The younger a generation is, the smaller the amount of memory reserved to it.

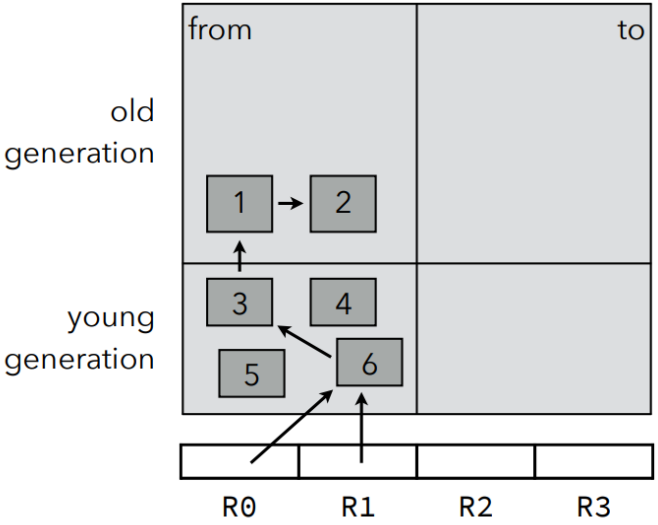
- All objects are initially allocated in the youngest generation.
- When it is full, a **minor collection** is performed, to collect memory in that generation only. Some of the surviving objects are promoted to the next generation, based on a **promotion policy**.

# Generational GC

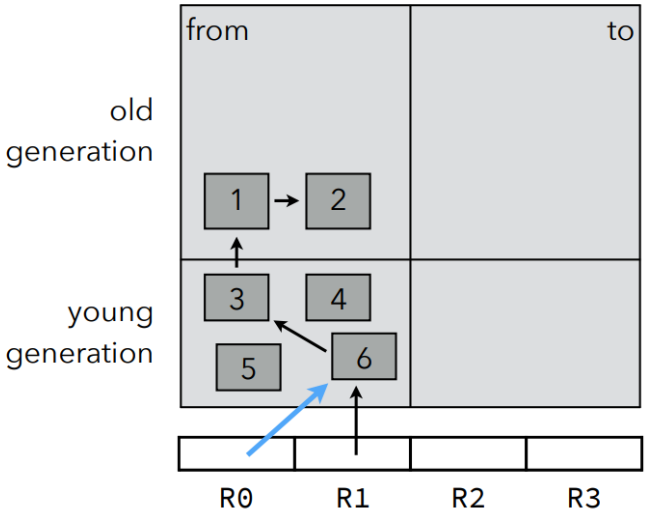
In a generational GC, objects are partitioned into a given number (often 2) of generations. The younger a generation is, the smaller the amount of memory reserved to it.

- All objects are initially allocated in the youngest generation.
- When it is full, a **minor collection** is performed, to collect memory in that generation only. Some of the surviving objects are promoted to the next generation, based on a **promotion policy**.
- When an older generation is itself full, a **major collection** is performed to collect memory in that generation and all the younger ones.

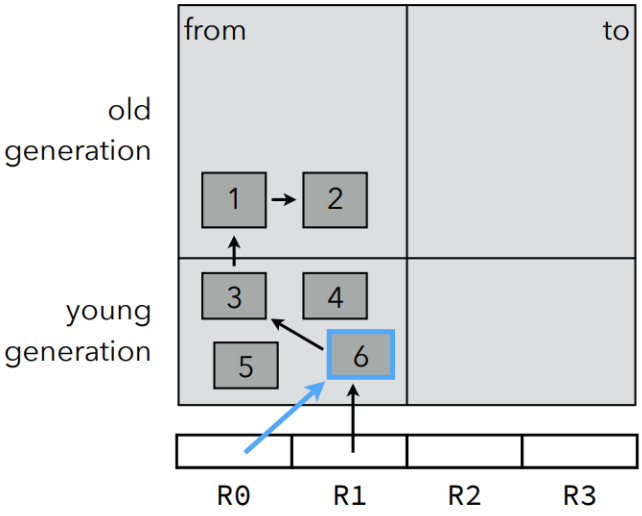
# Minor Collection Example



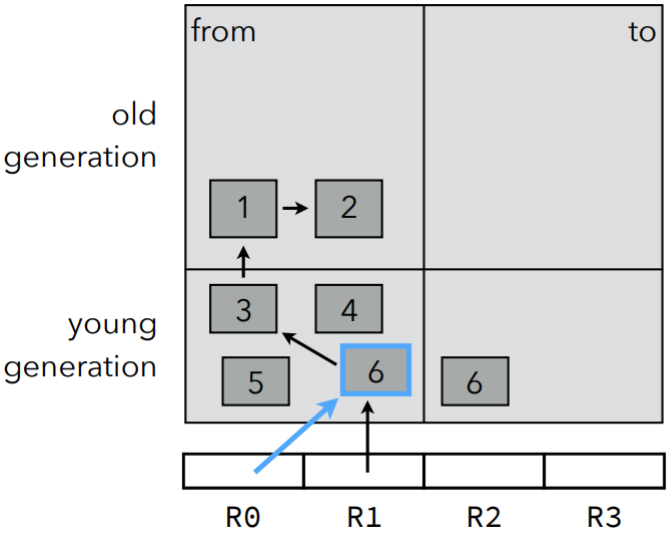
# Minor Collection Example



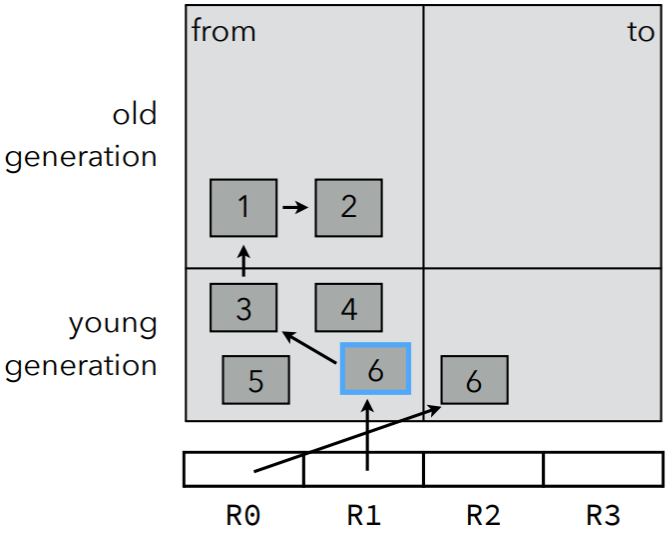
# Minor Collection Example



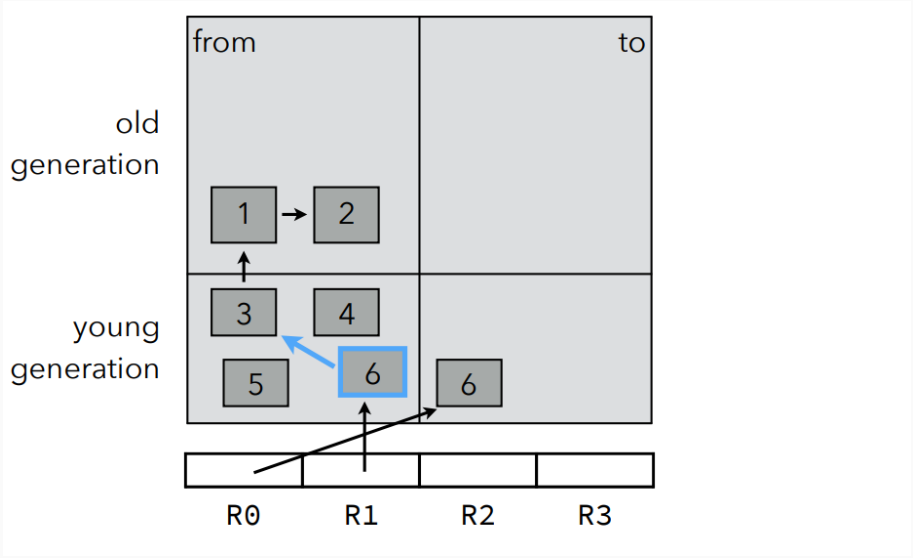
# Minor Collection Example



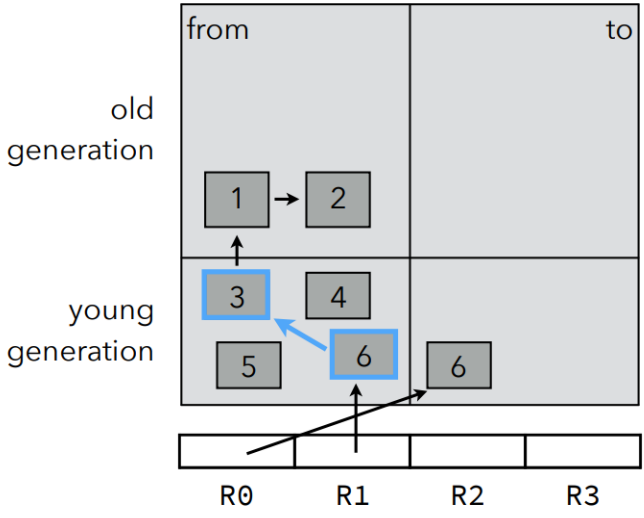
# Minor Collection Example



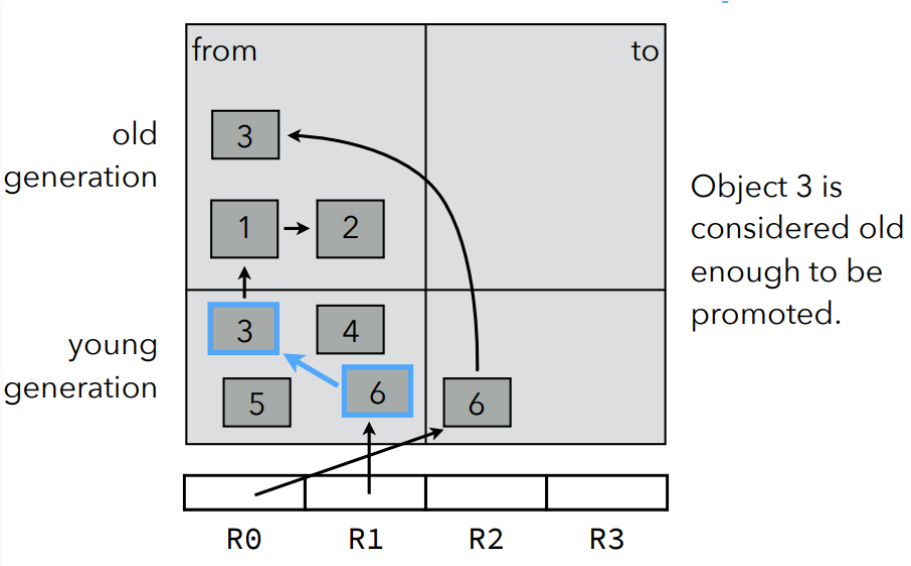
# Minor Collection Example



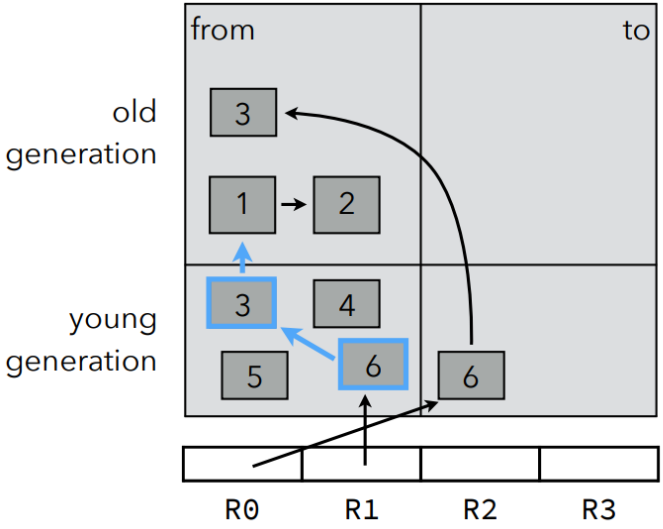
# Minor Collection Example



# Minor Collection Example

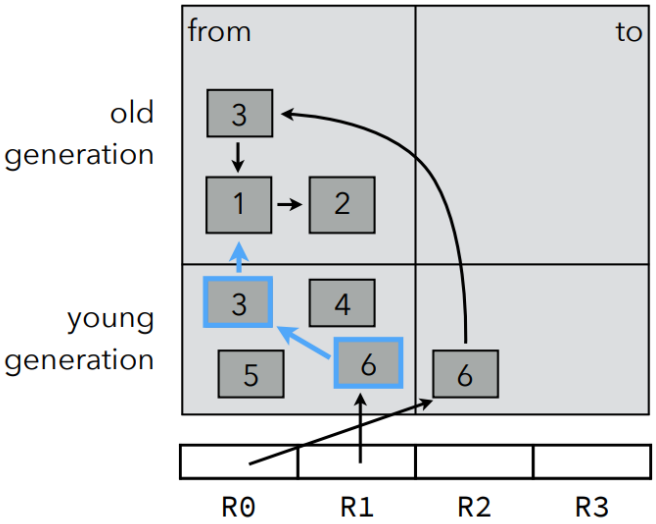


# Minor Collection Example



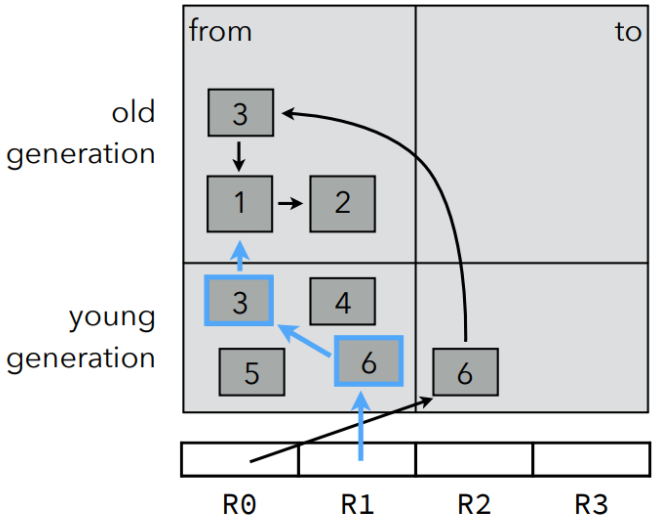

Object 3 is considered old enough to be promoted.

# Minor Collection Example



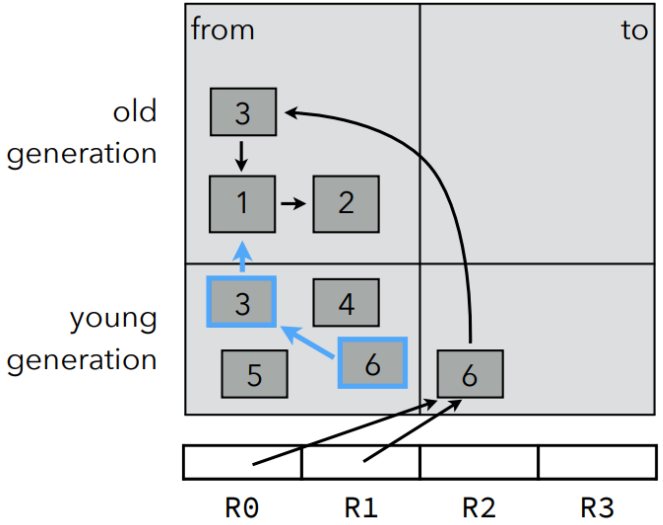

Object 3 is considered old enough to be promoted.

# Minor Collection Example



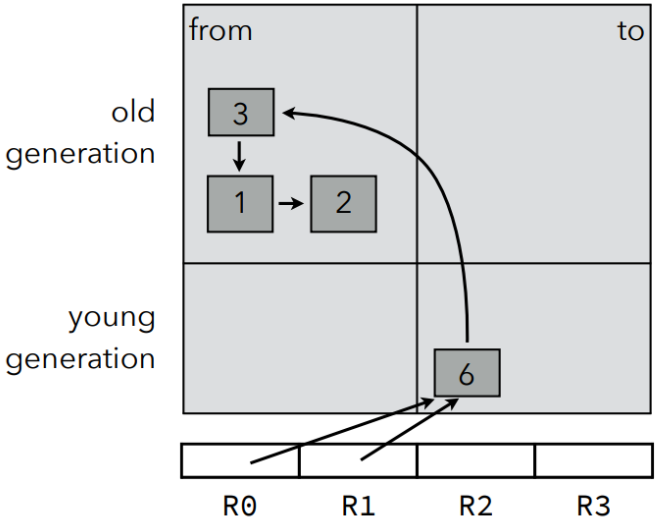

Object 3 is considered old enough to be promoted.

# Minor Collection Example



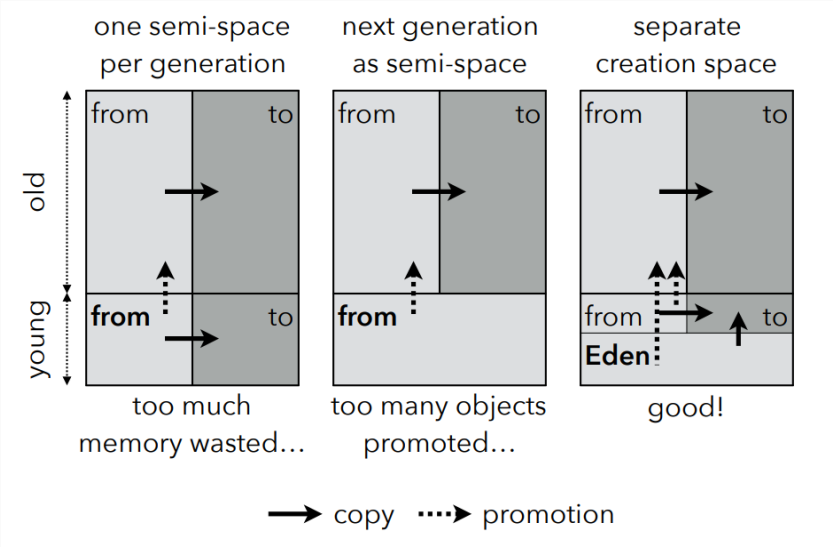

Object 3 is considered old enough to be promoted.

# Minor Collection Example




Object 3 is considered old enough to be promoted.

# Heap Organization



# Hybrid Heap Organization

Instead of managing all generations using a copying algorithm, it is also possible to manage some of them - the oldest, typically - using a mark & sweep algorithm.

# Promotion Policies

Generational GCs use a **promotion policy** to decide when objects should be advanced to an older generation.

# Promotion Policies

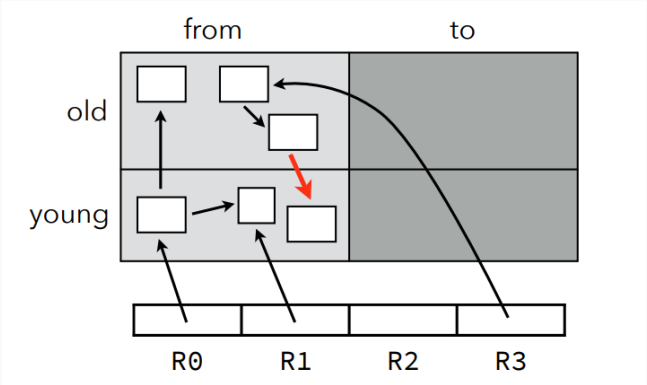
Generational GCs use a **promotion policy** to decide when objects should be advanced to an older generation.

- The simplest one - all survivors are advanced - can promote very young objects, but is simple as object age does not need to be recorded.
- To avoid promoting very young objects it is sufficient to wait until they survive a second collection before advancing them.

# Minor Collection Roots

The roots used for a minor collection must also include all pointers from older generations to younger ones.

Otherwise, objects reachable only from the old generation would incorrectly get collected!



# Inter-Generational Pointers

Pointers from old to young generations, called **inter-generational** pointers.

```
A.field = B // A is in old gen, B is in young gen
```

Two possible ways to handle that:

1) by scanning - without collecting - older generations during a minor collection.
   Problem: expensive.

# Inter-Generational Pointers

Pointers from old to young generations, called **inter-generational** pointers.

```
A.field = B // A is in old gen, B is in young gen
```

Two possible ways to handle that:

1) by scanning - without collecting - older generations during a minor collection. Problem: expensive.
2) by detecting pointer writes using a write barrier - implemented either in software or through hardware support - and remembering inter-generational pointers.

# Remembered Set

A **remembered** set contains all old objects pointing to young objects.

The write barrier maintains this set by adding objects to it if and only if:

- the object into which the pointer is stored is not yet in the remembered set, and
- the pointer is stored in an old object, and points to a young one - although this can also be checked later by the collector.

# Card marking

**Card marking** is another technique to detect inter-generational pointers.

Memory is divided into small, fixed sized areas called cards.

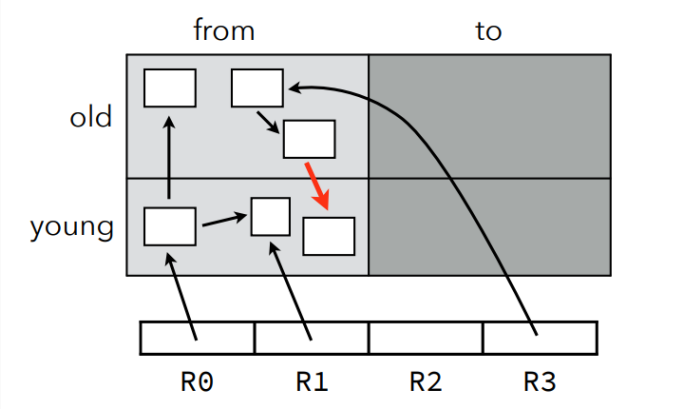
A card table remembers, for each card, whether it potentially contains inter-generational pointers.

On each pointer write, the card is marked in the table, and marked cards are scanned for inter-generational pointers during collection.

# Nepotism

Since old generations are not collected as often as young ones, it is possible for dead old objects to prevent the collection of dead young objects.

This problem is called **nepotism**.



# Pros and Cons

- Generational GC tends to reduce GC pause times since only the youngest generation - which is also the smallest - is collected most of the time.
- Combined with copying GCs, the use of generations also avoids copying long-lived objects over and over.

# Pros and Cons

- Generational GC tends to reduce GC pause times since only the youngest generation - which is also the smallest - is collected most of the time.
- Combined with copying GCs, the use of generations also avoids copying long-lived objects over and over.
- The only problems of generational GCs are the cost of maintaining the remembered set and nepotism.

# A Few Other Variants

# Other Kinds Of Garbage Collectors: Incremental/concurrent GC

An **incremental garbage collector** can collect memory in small, incremental steps, thereby reducing the length of GC pauses - a very important characteristic for interactive applications.

# Other Kinds Of Garbage Collectors: Incremental/concurrent GC

An **incremental garbage collector** can collect memory in small, incremental steps, thereby reducing the length of GC pauses - a very important characteristic for interactive applications.

Incremental GCs must be able to deal with modifications to the reachability graph made by the main program - called the **mutator** - while they attempt to compute the graph and collect memory.

# Other Kinds Of Garbage Collectors: Incremental/concurrent GC

An **incremental garbage collector** can collect memory in small, incremental steps, thereby reducing the length of GC pauses - a very important characteristic for interactive applications.

Incremental GCs must be able to deal with modifications to the reachability graph made by the main program - called the **mutator** - while they attempt to compute the graph and collect memory.

This is usually achieved using a write barrier that ensures that the reachability graph observed by the GC is a valid approximation of the real one. Several techniques, not covered here, exist to guarantee the validity of this approximation.

# Other Kinds Of Garbage Collectors: Parallel GC

Some parts of garbage collection can be sped up considerably by performing them in parallel on several processors. This is becoming important with the popularization of multi-core architectures.

For example, the *marking phase* of a mark & sweep GC can easily be done in parallel by several processors.

(Remember that parallelism and concurrency are separate and orthogonal concepts! A parallel GC does not have to be concurrent, and a concurrent GC does not have to be parallel.)

# Other Kinds Of Garbage Collectors: Virtual-Memory-Aware GC

The GCs presented until now are oblivious to the OS virtual memory manager. Unfortunately, this can lead them to perform badly when little physical memory is available: by traversing all live objects, even those residing on pages evicted to disk, they can incur considerable paging activity.

# Other Kinds Of Garbage Collectors: Virtual-Memory-Aware GC

The GCs presented until now are oblivious to the OS virtual memory manager. Unfortunately, this can lead them to perform badly when little physical memory is available: by traversing all live objects, even those residing on pages evicted to disk, they can incur considerable paging activity.

**Bookmarking GC** is an example of a GC that avoids this problem. Its basic idea is to bookmark memory-resident objects that are referenced by evicted objects. These bookmarked objects are then considered reachable, and garbage collection is performed without looking at - and therefore loading - evicted objects.

# Additional GC Features: Finalizers

Finalizers are functions that are called when an object is about to be collected. They are generally used to free "external" resources associated with the object about to be freed.

Since there is no guarantee about when finalizers are invoked, the resource in question should not be scarce.

Some GCs make it possible to associate **finalizers** with objects.

# Finalizer Issues

Finalizers are tricky for a number of reasons:

- What should be done if a finalizer makes the finalized object reachable again - e.g. by storing it in a global variable?
- How do finalizers interact with concurrency - e.g. in which thread are they run?
- How can they be implemented efficiently in a copying GC, which doesn't visit dead objects?

# Finalizer Issues

Finalizers are tricky for a number of reasons:

- What should be done if a finalizer makes the finalized object reachable again - e.g. by storing it in a global variable?
- How do finalizers interact with concurrency - e.g. in which thread are they run?
- How can they be implemented efficiently in a copying GC, which doesn't visit dead objects?

Java used to have finalizers ( `finalize()` method), but they were deprecated in Java 9 and removed in Java 18, due to their unpredictability and performance issues.

# Summary

- Cheney's copying GC
- Mostly-copying GC due to ambiguous pointers
- Generational GC
- Other issues
  - Incremental, concurrent, or parallel GC
  - Virtual-memory-aware GC
  - Finalizers