COS320: Compiling Techniques

Zak Kincaid

April 23, 2024

Compiling object-oriented languages

Objects

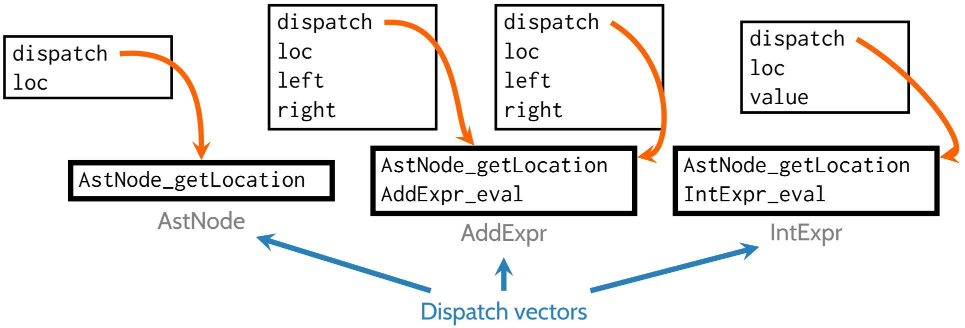
An *object* consists of Data (attributes) and Behavior (methods).

```
public class AstNode {
  location loc;
  public AstNode(location nodeloc)
  {  loc = nodeloc; }
  public location getLocation()
  { return loc; }
}
abstract class Expr extends AstNode {
  public abstract int eval(Env);
  public Expr(location loc) { super(loc); }
}
public class AddExpr extends Expr {
  Expr left, right;
  public AddExpr(int loc, Expr x, Expr y)
  {  super(loc); left = x; right = y; }
  public int eval(Env env)
  {  return left.eval(env) + right.eval(env); }
}
```

```
public class IntExpr extends Expr {
  int value;
  public IntExpr(int loc, int k)
  { super(loc);  value = k; }
  public int eval(int env)
  { return value; }
}
```

Compiling objects

- Compiling OO languages with single inheritance:
  - Each class is associated with a *dispatch vector* (aka virtual table, vtable)
    - dispatch vector = record of function pointers – one for each method
  - Each object is associated with a record, with one field for the dispatch vector of its class, and one field for each attribute



Compiling methods

Each method is extended with an additional parameter for the current object

- Gives the method access to the attributes of the object
- Dispatch vector enables dynamic dispatch

```
location AstNode__getLocation(self) {
    return self.loc;
}
int AddNode__eval(self, env) {
    return self.left.dispatch.eval(self.left, env)
           + self.right.dispatch.eval(self.right, env);
}
int IntNode__eval(self, env) {
     return self.value;
}
```

```
class AstNode { ...
  public location getLocation() { return loc; } }
```

```
public class AddExpr extends Expr { ...
  public int eval(Env env) { return left.eval(env) + right.eval(env); } }
```

```
class IntExpr extends Expr { ...
  public int eval(int env) { return value; } }
```

Subtyping

- Recall the *Liskov substitution priciple*: if s is a subtype of t, then terms of type s can be used as if they have type t without breaking type safety.
  - If class B extends class A, then B is a subtype of A

Subtyping

- Recall the *Liskov substitution priciple*: if s is a subtype of t, then terms of type s can be used as if they have type t without breaking type safety.
  - If class B extends class A, then B is a subtype of A
- This works for the same reason that record width subtyping works:
  - If A has a method *foo*, it appears in the same position in A and B's dispatch vector
  - If A has an attribute *x*, then A objects and B objects place x in the same position in object records

RECORDWIDTH

$$\frac{}{\vdash \{\textit{lab}_1 : s_1; \ldots; \textit{lab}_m : s_m\} <: \{\textit{lab}_1 : s_1; \ldots; \textit{lab}_n : s_n\}} \; n < m$$

Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts

Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation of the class hierarchy

Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation of the class hierarchy
- One solution:
  - The dispatch table serves as a type tag
    (i.e., `typeOf(o) == AddExpr` $\Longleftrightarrow$ `o.dispatch = DispatchVector(AddExpr)`)

Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation of the class hierarchy
- One solution:
  - The dispatch table serves as a type tag (i.e., `typeOf(o) == AddExpr` $\iff$ `o.dispatch = DispatchVector(AddExpr)`)
  - The first member of each dispatch table is a pointer to parent type

Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation of the class hierarchy
- One solution:
  - The dispatch table serves as a type tag
    (i.e., `typeOf(o) == AddExpr` $\iff$ `o.dispatch = DispatchVector(AddExpr)`)
  - The first member of each dispatch table is a pointer to parent type
  - To check `o instanceOf C`, walk up the class hierarchy
    - `o.dispatch = DispatchVector(C)`, or
    - `o.dispatch != DispatchVector(Object)` and `o.dispatch.parent = DispatchVector(C)`, or
    - `o.dispatch != DispatchVector(Object)` and `o.dispatch.parent != DispatchVector(Object)` and `o.dispatch.parent.parent = DispatchVector(C)`, or
    - ...

Testing class membership

- Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
- To implement, we need a run-time representation of the class hierarchy
- One solution:
  - The dispatch table serves as a type tag
    (i.e., `typeOf(o) == AddExpr` $\iff$ `o.dispatch = DispatchVector(AddExpr)`)
  - The first member of each dispatch table is a pointer to parent type
  - To check `o instanceOf C`, walk up the class hierarchy
    - `o.dispatch = DispatchVector(C)`, or
    - `o.dispatch != DispatchVector(Object)` and `o.dispatch.parent = DispatchVector(C)`, or
    - `o.dispatch != DispatchVector(Object)` and `o.dispatch.parent != DispatchVector(Object)` and `o.dispatch.parent.parent = DispatchVector(C)`, or
    - ...
  - Checked downcasting: if `o instanceOf c` then bitcast, otherwise throw run-time exception.

Multiple inheritence

- Some languages (such as C++) support a class extending more than one base class

Multiple inheritence

- Some languages (such as C++) support a class extending more than one base class
- Previous strategy does not work: base classes have conflicting ideas about where methods are stored in vtable

Multiple inheritence

- Some languages (such as C++) support a class extending more than one base class
- Previous strategy does not work: base classes have conflicting ideas about where methods are stored in vtable
- Solution: Use hash tables instead of records
    - Cost can be reduced with optimizing compiler

Multiple inheritence

- Some languages (such as C++) support a class extending more than one base class
- Previous strategy does not work: base classes have conflicting ideas about where methods are stored in vtable
- Solution: Use hash tables instead of records
  - Cost can be reduced with optimizing compiler
- Another solution: For every $A <: B$, create an A-in-B vtable
  - A-in-B is laid out like B's vtable but contains function pointers to A's methods
  - Object = triple of primary vtable pointer + secondary vtable pointer + attribute pointer.
    - Secondary used to resolve method calls!
  - To cast from A to B: allocate a new triple, changing the secondary table to A-in-B

Garbage Collection

Garbage collection

- Many modern languages feature *garbage collectors*, which automatically reclaim memory that was allocated by a program but no longer used

Garbage collection

- Many modern languages feature *garbage collectors*, which automatically reclaim memory that was allocated by a program but no longer used
- A memory location is *garbage* if it will not be used in the remainder of the program

Garbage collection

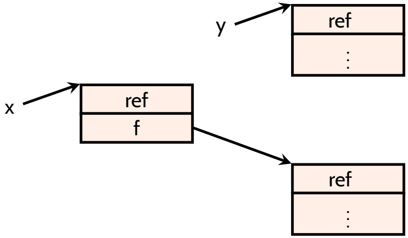
- Many modern languages feature *garbage collectors*, which automatically reclaim memory that was allocated by a program but no longer used
- A memory location is *garbage* if it will not be used in the remainder of the program
- Determining whether or not it will be used is undecidable
  - *But*, we are happy with a conservative approximation: free memory if it *cannot possibly be used* in the remainder of the program

Garbage collection

- Many modern languages feature *garbage collectors*, which automatically reclaim memory that was allocated by a program but no longer used
- A memory location is *garbage* if it will not be used in the remainder of the program
- Determining whether or not it will be used is undecidable
  - *But*, we are happy with a conservative approximation: free memory if it *cannot possibly be used* in the remainder of the program
- Usually not a *static analysis*, but rather a *dynamic analysis*
  - *static analyses* collect information about a program without running it
  - *dynamic analyses* collect information about a program while running it

Reference counting

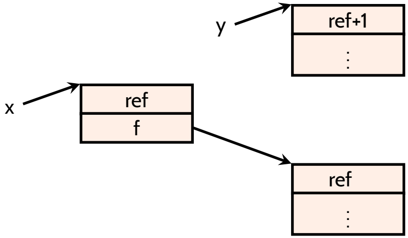
- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store `x->f = y`



Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store `x->f = y`

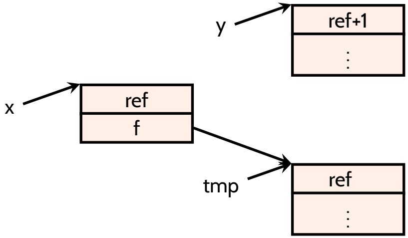
```
y->count ++
```



Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store `x->f = y`

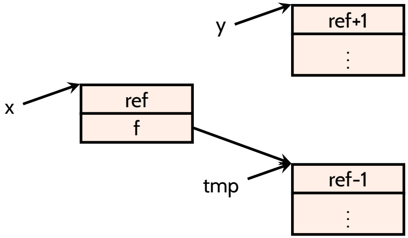
```
y->count ++
tmp = x->f
```



Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store `x->f = y`

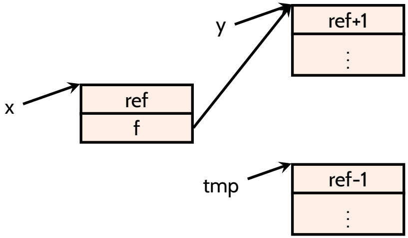
```
y->count ++
tmp = x->f
tmp->count --
if (tmp->count == 0) free(tmp);
```



Reference counting

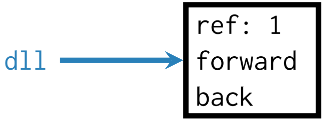
- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- Example: compiling a store `x->f = y`

```
y->count ++
tmp = x->f
tmp->count --
if (tmp->count == 0) free(tmp);
x->f = y
```

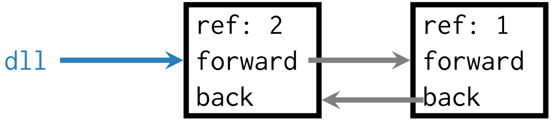


Problem: *cyclic* data structures never get collected

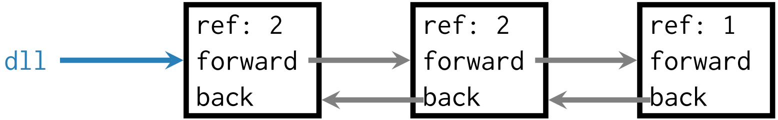
Problem: *cyclic* data structures never get collected



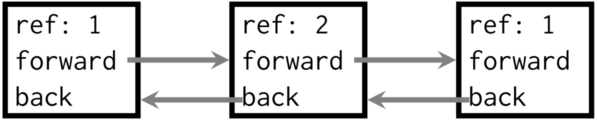
Problem: *cyclic* data structures never get collected



Problem: *cyclic* data structures never get collected



Problem: *cyclic* data structures never get collected



Tracing-based GC

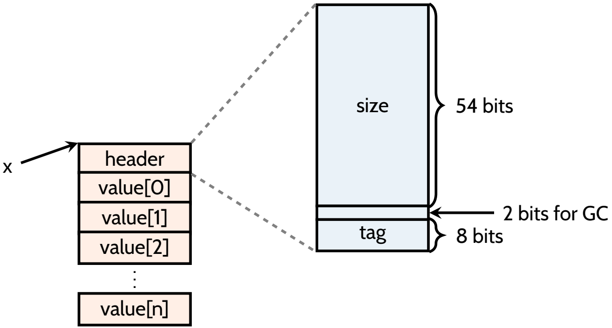
- *Tracing garbage collection*: a memory location is garbage if it is unreachable from the program's *roots*
    - *roots* = registers, stack, global static data

Tracing-based GC

- *Tracing garbage collection*: a memory location is garbage if it is unreachable from the program's *roots*
  - *roots* = registers, stack, global static data
- Mark-and-sweep:
  - Each memory location gets an extra bit to hold a "mark"
  - *Mark*: When there is no remaining free memory, run a DFS search from the roots, marking all memory locations
  - *Sweep*: Traverse the entire heap; unmarked nodes are collected; marked nodes are unmarked

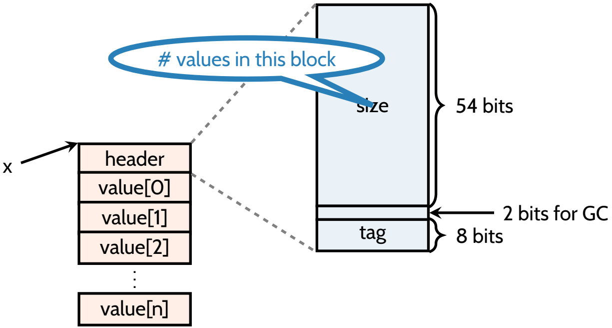
Memory layout

- *Boxing*: **every** value is a pointer to a block of memory that begins with metadata. In OCaml:



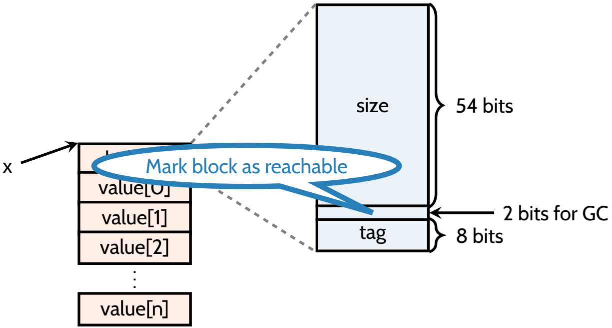
Memory layout

- *Boxing*: **every** value is a pointer to a block of memory that begins with metadata. In OCaml:



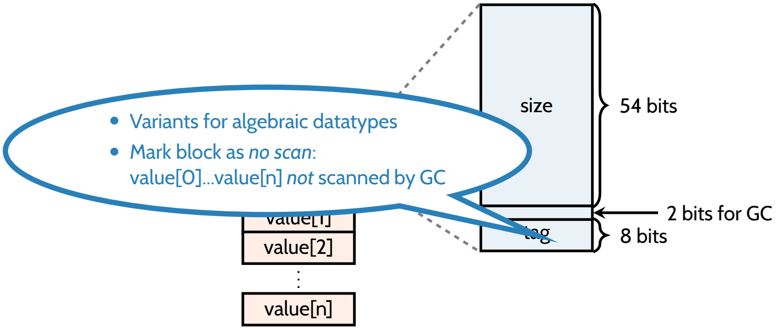
Memory layout

- *Boxing*: **every** value is a pointer to a block of memory that begins with metadata. In OCaml:



Memory layout

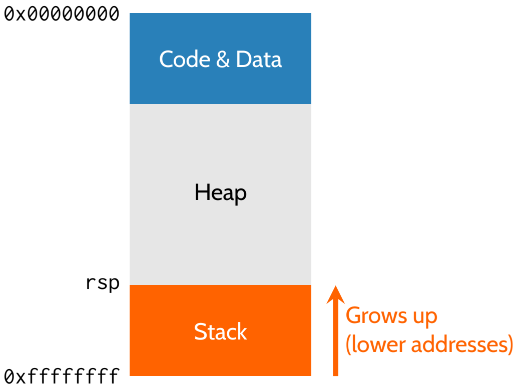
- *Boxing*: **every** value is a pointer to a block of memory that begins with metadata. In OCaml:



Finding roots

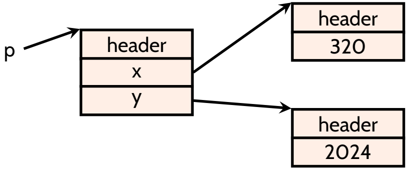
Stack is a sequence of 64-bit values

- Values (pointers in the heap); i.e., roots
- Saved frame pointers (pointers in the stack)
- Saved return addresses (pointers in code)



Tagged pointers

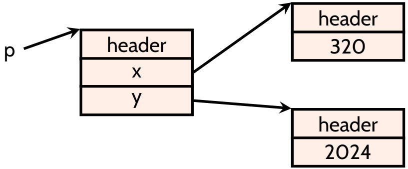
- Boxing has high overhead




```
type point = { x : int; y : int }
```

Tagged pointers

- Boxing has high overhead

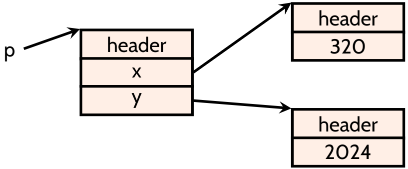



```
type point = { x : int; y : int }
```

- Pointers are *quadword aligned* ⇒ last four (low-order) bits are 0

Tagged pointers

- Boxing has high overhead




```
type point = { x : int; y : int }
```

- Pointers are *quadword aligned* ⇒ last four (low-order) bits are 0
- If values for a type fit into 63 bits, can used *unboxed* value, marked with a last (low-order) bit so GC does not scan
  - Integers are 63 bit: `x` is represented as `x«1 | 1`

Copying GC

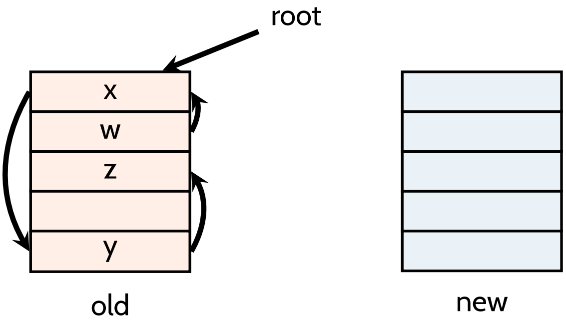
- Mark-and-sweep can lead to memory fragmentation

Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes

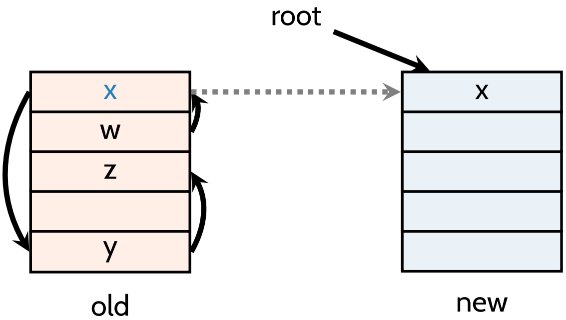
Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes
- *Copying* (or *Moving*) *GC*:
  - Maintain two heaps (roughly equal size), *old* and *new*
  - GC sequentially copies reachable blocks from old heap to new heap



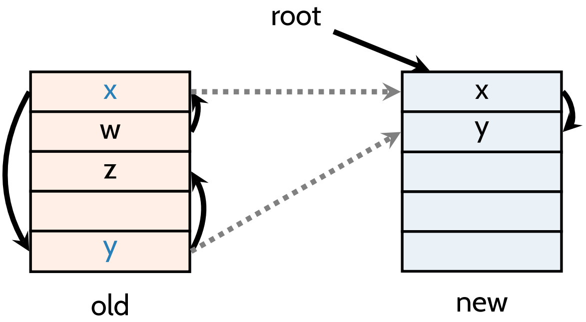
Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes
- *Copying* (or *Moving*) *GC*:
  - Maintain two heaps (roughly equal size), *old* and *new*
  - GC sequentially copies reachable blocks from old heap to new heap



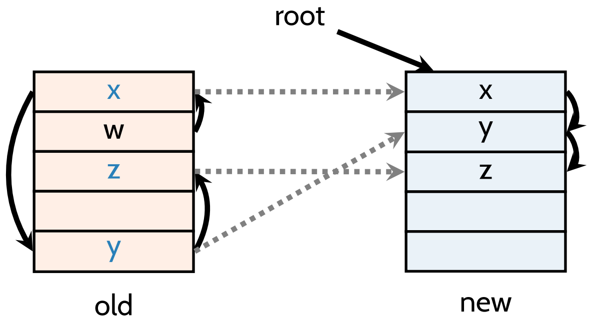
Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes
- *Copying* (or *Moving*) *GC*:
  - Maintain two heaps (roughly equal size), *old* and *new*
  - GC sequentially copies reachable blocks from old heap to new heap



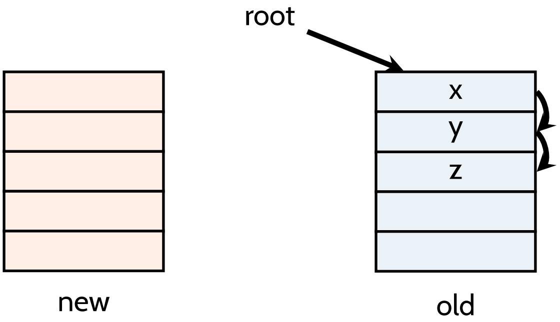
Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes
- *Copying* (or *Moving*) *GC*:
  - Maintain two heaps (roughly equal size), *old* and *new*
  - GC sequentially copies reachable blocks from old heap to new heap



Copying GC

- Mark-and-sweep can lead to memory fragmentation
- Since GC traverses the heap anyway, might as well compact as it goes
- *Copying* (or *Moving*) *GC*:
  - Maintain two heaps (roughly equal size), *old* and *new*
  - GC sequentially copies reachable blocks from old heap to new heap



Generational GC

- Generational hypothesis:
  - Most memory becomes garbage quickly after allocation
  - Memory that does not quickly become garbage is likely to not be garbage for a very long time

Generational GC

- Generational hypothesis:
  - Most memory becomes garbage quickly after allocation
  - Memory that does not quickly become garbage is likely to not be garbage for a very long time
- Generational GC: maintain several heaps ("generations") $G_0, G_1, \ldots$
  - Allocate in G_0, and scan frequently
  - Scan G_1 less frequently, G_2 less frequently than that, ...
  - After collecting garbage in G_i, *non*-garbage is promoted to G_{i+1}

Generational GC

- Generational hypothesis:
  - Most memory becomes garbage quickly after allocation
  - Memory that does not quickly become garbage is likely to not be garbage for a very long time
- Generational GC: maintain several heaps ("generations") $G_0, G_1, \ldots$
  - Allocate in G_0, and scan frequently
  - Scan G_1 less frequently, G_2 less frequently than that, ...
  - After collecting garbage in G_i, *non*-garbage is promoted to G_{i+1}
- Complication: inter-generational pointers (from older to newer generation) are new roots that must be managed

Summary

- Reference counting
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks

Summary

- Reference counting
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks
- Mark-and-sweep GC
  - Low memory requirements
  - Memory fragmentation, long pauses

Summary

- Reference counting
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks
- Mark-and-sweep GC
  - Low memory requirements
  - Memory fragmentation, long pauses
- Copying GC
  - Simple (no free list), Less memory fragmentation
  - Cuts available memory in half, long pauses

Summary

- Reference counting
  - No long pauses (as for tracing GC)
  - Performance penalty for maintaining refcounts, cycles cause leaks
- Mark-and-sweep GC
  - Low memory requirements
  - Memory fragmentation, long pauses
- Copying GC
  - Simple (no free list), Less memory fragmentation
  - Cuts available memory in half, long pauses
- Generational GC
  - Shortens average GC pauses; can combine mark-and-sweep & copying GC
  - Relatively complicated, performance penalty for managing intergenerational pointers