Object Oriented Languages

CHAPTERS 7, 8, AND 10 OF PROGRAMMING LANGUAGES PRAGMATICS

Three fundamental concepts

- Encapsulation:
  - Group data and subroutines in one place, hide irrelevant details from users of the abstraction
- Inheritance:
  - New abstractions defined as refinements or extensions of existing abstractions
- Dynamic Method Binding:
  - Allow a new version of an abstraction to display newly refined behavior, even when used in a context that expects an earlier version

Outline

- **Classes and objects**
- Methods
  - Inheritance, polymorphism
  - Static methods and fields
- Implementation: compilation, allocation
- Types
- Memory and type safety
- Memory Management

Classes

- A class is a blueprint for creating objects
  - Contains all the information the programmer needs to use the abstraction
  - Contains all the information the compiler needs to generate code

```
class Rectangle {
    public double height, width;
    public double area() {
            return height * width;
    }
}
```

  - This is Java code, the equivalent C++ code is very similar
- Class members: methods and fields

Objects

- The central concept of object oriented programming
- In C++ and Java, objects are instances of classes, created through new
  - For example, when the expression new Rectangle() is evaluated, a new object of class Rectangle is created and initialized
  - “instance” = “object”
  - “class X is instantiated” = “an instance of X is created”

Creation of objects

- During the evaluation of x = new Rectangle()
  - A new instance (object) of class Rectangle is created on the heap
  - A reference (pointer) to this new instance is produced
    - This is the result of evaluating the new expression
  - The approptriate constructor of the class is called to initialize the new object
  - x is assigned this reference (pointer) value
    - the value may be the address of the first bye of the objects memory
    - or the value may be some internal handle to the actual object (i.e. an index in some internal table which contains the address of the first byte)

Destruction of objects

- C++: each new must have a corresponding delete
  - x = new Rectangle; ... delete x;
- Java: dead objects are reclaimed automatically by a garbage collector (GC)
  - x = new Rectangle(); // After you stop using the object, the GC (hopefully) figures out it is dead
- C++ destructors: called when the programmer manually destroys the object with delete
  - class Rectangle { ... ~Rectangle() { ... } }
- Java finalizers: called when the object is collected
  - class Rectangle { ... void finalize() { ... } }

Members: fields and methods

- Two separate kinds:
  - Instance members
    - Each instance of the class has a separate copy of this member
  - Static members

```
Rectangle a, b, c;
a = new Rectangle();
b = new Rectangle();
a.height = 1.0; a.width = 3.6;
b.height = 2.2; b.width = 5.0;
c = a;
```

Members: fields and methods (C++)

- C++: x->f is shorthand for (*x).f
  - x evaluates to a pointer value that points to the object
  - *x evaluates to the actual object
  - (*x).f evaluates to the field f of the object

```
Rectangle *a, *b, *c;
a = new Rectangle();
b = new Rectangle();
a->height = 1.0; a->width = 3.6;
b->height = 2.2; b->width = 5.0;
c = a;
```

Outline

- Classes and objects
- **Methods**
  - Inheritance, polymorphism
  - Static methods and fields
- Implementation: compilation, allocation
- Types
- Memory and type safety
- Memory Management

Instance methods

- An instance method operates on objects
  - Is invoked on the object

```
double area() {return height * width; }
```

- In reality, this is shorthand for

```
double area(Rectangle this) {
        return this.height * this.width; } //Java
double area(Rectangle* this) {
        return this->height * this->width; } //C++
```

- There is an implicit formal parameter this: a reference to the object on which the method was invoked
  - Calls like x.area() and x->area() are, essentially, calles to area(x)

Constructors

- Constructors are used to set up the initial state of new objects

```
public Rectangle(double height, double width) {
        this.height = height; this.width = width;
}
```

- x = new Rectangle(1.1, 2.3);
  - A new object is created, with default value 0.0 in Java and undefined values in C++
  - The constructor is invoked on this object; the fields are initialized with 1.1 and 2.3
  - A reference ot the object is assigned to x

Outline

- Classes and objects
- Methods
  - **Inheritance, polymorphism**
  - Static methods and fields
- Implementation: compilation, allocation
- Types
- Memory and type safety
- Memory Management

Inheritance

- class B extends A { ... }
  - Single inheritance: only one superclass (Java)
- class B : public A1, A2, A3 { ... }
  - Multiple inheritance: several superclasses (C++)
- Every member of A is inherited by B
  - If a field f is defined in A, every object of class B has an f field
  - If a method is defined in A, this method can be invoked on an object of class B
- B may declare new members

Example

```
class Rectangle {
	private double height, width;
	public double getHeight() { return height; }
	public double get Width() { return width; }
	public double area() { ... }
}

class SwissRectangle extends Rectangle {
	private int hole_size;
	public SwissRectangle(double h, double w, int hs) { ... }
	public void shrinkHold() { hole_size--; }
	public double area() { ... }
}
```

Constructors and inheritance

- Constructors are not inherited
- A constructor in a subclass B must invoke a constructor in the superclass A
  - This is maybe an oversimplification
- The constructor of superclass A initialized the part of the "object state" that is declared in A
  - Sets up values for fields declared in A and inherited by the subclasses

```
class SwissRectangle extends Rectangle {
        private int hole_size;
        public SwissRectangle(double h, double w, int hs) {
                super(h, w);
                hold_size = hs;
        }
}
```

Inheritance of methods

- If a subclass declares a method with the same name but a different signature, we have overloading
  - Method signature: Some combination of
    - name
    - number and type of arguments
    - return type
  - **Either** method can be invoked on an instance of the subclass
- If a subclass declares a method with the same signature, we have overriding
  - **Only** the new method applies to instances of the subclass

Polymorphism of references

- Reference variables for A objects also may point to B objects
  - A x = new B() in Java
  - A* x = new B() in C++
- Simplistic view: the type of x is pointer (reference) to instance of A
- Correct view: pointer (reference) to instance of A or instances of any transitive subclass of A
  - If **C** is a subclass of B, variable x can also point to instance of **C**
    - **C** is a transitive subclass of A
  - **Poly** (many) **morph** (form) **ism** (state or condition)

Method invocation – compile time

- What happens when we have a method invocation of the form x.m(a, b)?
- Two very different things are done
  - At compile time, by the Java compiler (javac)
  - At run time, by the Java Virtual Machine
- At compile time, a target method is associated with the invocation expression
  - Terms: compile-time target, static target
  - The static target is **based on the declared type of x**

Method invocation – compile time

```
class A {void m(int p, int q) { … } … }
class B extends A { void m(int r, int s) { … } … }
A x;
x = new B();
x.m(1, 2);
```

- Since x has declared type A, the compile-time target is method m in class A
- Javac encodes this in the bytecode
  - virtualinvoke x,<A: void m(int,int)>

Method invocation – run time

- The Java Virtual Machine loads the bytecode and starts executing it
- When the JVM tries to execute the instruction
  virtualinvoke x,<A: void m(int,int)>
  - JVM looks at the class **Z** of the object referenced by x
    - i.e. what x actually is at run time
  - JVM searches **Z** for a method with signature m(int,int) and return type void
    - Searchs for a match to the signature
  - If **Z** does not have it, the JVM goes to **Z**'s superclass to find a match, and continues to search upwards through the class hierarchy until a match is found?
    - Is the JVM guaranteed to find a match?

Method invocation – run time

- The run time (dynamic) target: The "lowest" method that matches the signature of the static target method
  - "Lowest" with respect to the inheritance chain from **Z** to **java.lang.Object**
- Once the JVM determines the run time target method, it invokes it on the object that is referenced by x
- Key term:
  - Virtual dispatch/dynamic dispatch: The process of selecting which implementation of a method to call at run time

Virtual methods in C++

```
class A { virtual void m(int p, int q) { ... } ... }
class B  : public A { virtual void m(int r, int s) { ... } ... }
A* x;
x = new B();
x->m(1,2);
```

- Since x has declared type A*, the compile time target is method m in class A
- The run time target is m in B
  - Without the keyword virtual, the run time target will be the same as the compile time target

Abstract classes

- Abstract class: A class that contains abstract methods
  - abstract void m(int x); // Java
  - virtual void m(int x) = 0; // C++
- Abstract classes allow defining and creating functionality that subclasses can implement or override
- We cannot say new **X**() if **X** is an abstract
  - Why not?
- An abstract method can be the compile time target of a method call
  - But not the run time target
- Sometimes non-abstract classes are referred to as “concrete classes”

Interfaces

- Very similar to abstract classes in which all methods are abstract
  - Functionality can be defined but not implemented
- A reference variable can be of interface type, and can refer to any instance of a class that implements the interface
- An interface method can be the compile time target of a method call
- Java 8 allows default and static methods in interfaces

Example

```
interface X { void m(); }
interface Y { void n(); }
abstract class A implements X {
    void m() { … }
    abstract void m2();
}
class B extends A implements Y {
    void m2() { … }
    void n() {…}
}
X x = new B(); x.m();
Y y = new B(); y.n();
A a = new B(); a.m2();
```

compile-time
targets

Outline

- Classes and objects
- Methods
  - Inheritance, polymorphism
  - **Static methods and fields**
- Implementation: compilation, allocation
- Types
- Memory and type safety
- Memory Management

Static methods and fields

- Static field: A single copy for the entire class
- Static method: not invoked on an object
  - Just like a regular subroutine in a procedural language
- Terminology:
  - Static method/field = class method/field
  - Instance method/field = non-static method/field

Static example (Java)

```
class X { …
    private static int num = 0;
    // constructor
    public X() { num++; }
    public static int numInstances()
        { return num; }
}
```

<u>in main:</u>

```
X x1 = new X(); X x2 = new X();
int n = X.numInstances();
```

→ returns 2

Static example (C++)

```
class X { …
    private: static int num;
    public: X();
    public: static int numInstances();
}
int X::num = 0;
X::X() { num++; }
int X::numInstances() { return num; }
```

<u>in main:</u>

```
X* x1 = new X; X* x2 = new X;
int num = X::numInstances();
```

→ **returns 2**

Example: Singleton pattern (Java)

```
class Logger {
    private Logger() { }
    private static Logger instance = null;
    public static Logger getInstance() {
        if (instance == null)
            instance = new Logger();
        return instance;
    }
}
client code: Logger.getInstance().writeLog(…)
```

Outline

- Classes and objects
- Methods
  - Inheritance, polymorphism
  - Static methods and fields
- **Implementation: compilation, allocation**
- Types
- Memory and type safety
- Memory Management

Implementaiton techniques for Java

- The compiler takes as input source code
  - Oracle/Sun provides a standard compiler; other can build their own compilers if they want
  - Typically, class A is stored in file A.java
    - Exception: nested classes
- Compiler output: Java bytecode
  - A.java -> A.class
  - A standardized platform independent representation of Java code
  - Essentially, a programming language that is understood by the JVM
    - Why not have the JVM directly interpreter Java source?

Rectangle.class

```
class Rectangle extends
  java.lang.Object {
   public double height;
   public double width;
   Rectangle();
   public double area();
}
```

```
Rectangle()
   0 aload_0
   1 invokespecial #3 <Method
    java.lang.Object()>
   4 return
double area()
   0 aload_0
   1 getfield #4 <Field double height>
   4 aload_0
   5 getfield #5 <Field double width>
   8 dmul
   9 dreturn
```

Execution model

- Java bytecode is executed by a Java Virtual Machine (JVM)
  - Oracle/Sun provide several kinds of JVMs for various platforms
    - e.g. Windows, Linux, MacOS, Solaris, ect)
  - Several other vendors also provide JVMs
    - e.g. IBM sells a JVM that is performance-tuned for enterprise server applications
- Platform independence: as long as there are JVMs available, the exact same Java bytecode can be executed anywhere

JVM

- There are two ways to execute the bytecode
- Interpretation: The VM just executes each bytecode instruction itself
  - Initial JVMs used this model
- Compilation: The VM uses its own internal compiler to translate bytecode to native code for the platform
  - The native code is executed by the platform
  - Faster than interpretation

Compilation inside a VM

- Just-in-time: The first time some bytecode needs to be executed, it is compiled to native code on the fly
  - Typically done at method level, the first time a method is invoked the compiler kicks in
  - Problems: compilation has overhead, and the overall running time may actually increase
- Profile-driven compilation
  - Start executing through interpretation, but track “hot spots” (frequently executed methods) and after a certain threshold is reached compile them

Outline

- Classes and objects
- Methods
  - Inheritance, polymorphism
  - Static methods and fields
- Implementation: compilation, allocation
- **Types**
- Memory and type safety
- Memory Management

Types

- Organization of untyped values
  - Untyped universes: bit strings, S-expr, …
  - Categorize based on usage and behavior
- Type = set of computational entities with uniform behavior
- Constraints to enforce correctness
  - Check the applicability of operations
    - Should not try to multiply two strings
    - Should not use a character value as a condition of an if-statement
    - Should not use an integer as a pointer

Examples of Type Checking

- Built-in operators should get operands of correct types
- Type of left-hand side must agree with the value on the right-hand side
- Procedure calls: check number and type of actual arguments
- Return type should match returned value

Static Typing

- Statically typed languages: expressions in the code have static types
  - static type = claim about run-time values
  - Types are either declared or inferred
  - Examples: C, C++, Java, ML, Pascal, Modula-3
- A statically typed language typically does some form of static type checking
  - E.g., at compile time Java checks that the [] operator is applied to a value of type “array”
  - May also do dynamic (run-time) checking
    - e.g., Java checks at run time for array indices out of bounds and for null pointers

Dynamic Typing

- Dynamically-typed languages: entities in the code do not have static types
  - Examples: Lisp, Scheme, CLOS, Smalltalk, Perl, Python
  - Entities in the code do not have declared types, and the compiler does not try to infer types for them
- Dynamic type checking
  - Before an operation is performed at run time
  - E.g., in Scheme: **(+ 5 #t)** fails at run time, when the evaluation expects to see two numeric atoms as operands of +

Type (and memory) safety

- Type-safe languages: type-incorrect operations are not performed at run time
  - Things cannot “go wrong”: no undetected type errors
  - Certain run-time errors are possible but clearly marked as such
    - i.e. array index out of bounds, null pointer
  - C/C++: type unsafe
  - Java: type safe
- Independent of static vs. dynamic
  - Lisp, Scheme, Python: dynamically typed; type safe
  - Forth: dynamically typed; type unsafe

Examples of Types

- Integers
- Arrays of integers
- Pointers to integers
- Records with fields **int x** and **int y**
  - e.g., “struct” in C
- Objects of class C or a subclass of C
  - e.g., C++, Java, C#
- Functions from any list to integers

Numeric Types

- Varied from language to language
- C does not specify the ranges of numeric types
  - Integer types: char, short, int, long, long long
    - Includes “unsigned” versions of these
  - Floating-point types: float, double, long double
- Java specifies the ranges of numeric types
  - byte: 8-bit signed two's complement integer [-128,+127]
  - short: 16-bit signed two's complement integer [-32768,+32,767]
  - int: 32-bit signed two's complement integer
    - [-2147483648,+2147483647]
  - long: 64-bit signed two's complement integer
    - [-9223372036854775808, +9223372036854775807]
  - float/double: single/double-precision 32-bit IEEE 754 floating point
  - char: single 16-bit Unicode character; minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65535)

Enumeration Types

- C: a set of named integer constant values
  - Example from the C specification

```
enum hue { chartreuse, burgundy, claret=20, winedark };
/* the set of integer constant values is { 0, 1, 20, 21 } */
enum hue col, *cp;
col = claret; cp = &col;
if (*cp != burgundy) …
```

- Java: a fixed set of named items (not integers)

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY }
```

  - In reality, it is like a class: e.g., it can contain methods

Types as Sets of Values

- Integers
  - Any number than can be represented in 32 bits in signed two's-complement
  - "**type int**" = { -2^{31}, …, $2^{31} - 1$ }
- Class type (not the same as a class)
  - Any object of class C or a subclass of C
  - "**type C**" = set of all instances of C or of any transitive subclass of C ("**class C**" is just a blueprint for objects)
- **Subtypes are subsets**: T2 is a **subtype** of T1 if T2's set of values is a subset of T1's set of values

Monomorphism vs. Polymorphism

- Greek:
  - mono = single
  - poly = many
  - morph = form
- Monomorphism
  - Every computational entity belongs to exactly one type
- Polymorphism
  - A computational entity can belong to multiple types

Types of Polymorphism

- polymorphism
  - universal
    - parametric
    - inclusion (subset)
  - ad hoc
    - overloading
    - coercion

Types of Polymorphism

- polymorphism
  - universal
    - parametric
    - inclusion (subset)
  - ad hoc
    - overloading
    - **coercion**

Coercion

- Values of one type are silently converted to another type
  - e.g. addition: 3.0 + 4 : converts 4 to 4.0
    - **int × int → int** or **real × real → real**
- In a context where the type of an expression is not appropriate
  - either an automatic coercion (conversion) to another type is performed automatically
  - or if not possible: compile-time error

Coercions

- Widening
  - coercing a value into a “larger” type
  - e.g., **int** to **float**, subclass to superclass
- Narrowing
  - coercing a value into a “smaller” type
  - loses information, e.g., **float** to **int**

Types of Polymorphism

- polymorphism
  - universal
    - parametric
    - inclusion (subset)
  - ad hoc
    - **overloading**
    - coercion

Types of Polymorphism

- polymorphism
  - universal
    - **parametric**
    - inclusion (subset)
  - ad hoc
    - overloading
    - coercion

Parametric Polymorphism: Generics in Java

```
package java.util;
public interface Set<E> extends Collection<E> { …
   Iterator<E> iterator();
   boolean add(E e);
   boolean addAll(Collection<? extends E> c); }
class Rectangle { … }
class SwissRectangle extends Rectangle { … }
Set<Rectangle> s = new HashSet<Rectangle>();
s.add(new Rectangle(1.,2.)); s.add(new SwissRectangle(3.,4.,5));
Set<SwissRectangle> s2 = new TreeSet<SwissRectangle>();
s2.add(new SwissRectangle(6.,7.,8)); s.addAll(s2);
```

Types of Polymorphism

- polymorphism
  - universal
    - parametric
    - **inclusion (subset)**
  - ad hoc
    - overloading
    - coercion

Inclusion (Subset) Polymorphism

- Subtype relationships among types
  - Defined by “Y is subset of X” (i.e., set inclusion)
- A computational entity of a subtype may be used in any context that expects an entity of a supertype
- Typical examples
  - Imperative languages: record types
  - Object-oriented languages: class types

Subtyping in Java

- Recall that **class type C** is the set of all instances of class C or of any transitive subclass of C
- Subtyping between class types

```
class X { int m () { … } }
class Y  extends  X { int m () { … } }
X x = new Y();
int i = x.m();
```

- Interface type: the set of all instances of classes that implements the interface (transitively)

```
interface Z { bool m(); }
class  W implements Z { bool m() { … } }
Z z = new W();  bool b = z.m();
```

Outline

- Classes and objects
- Methods
  - Inheritance, polymorphism
  - Static methods and fields
- Implementation: compilation, allocation
- Types
- **Memory and type safety**
- Memory Management

Memory and type safety

```
int[] a = new int[64];
…
a[82] = …;
```

Memory and type safety

int[] a = new int[64];

…

a[82] = …;

C: undefined behavior → major source of security exploits (how?)

Java: throws ArrayIndexOutOfBoundsException

How? Instrumentation added by JIT compiler (or proved unnecessary)

Memory and type safety

```
MyType* a = new MyType();
…
*((void*)a + 16) = 42;
```

Memory and type safety

```
MyType* a = new MyType();
…
*((void*)a + 16) = 42;
```

C++: Undefined behavior

Java: Pointer arithmetic isn't part of the language

Memory and type safety

```
SomeType* a = …;

b = (IncompatibleType*) a;
b->f = …;
```

Memory and type safety

```
SomeType* a = …;

b = (IncompatibleType*) a;
b->f = …;
```

C: undefined behavior → potential security exploit

Java: throws ClassCastException

How? Instrumentation added by JIT compiler (or proves unnecessary)

Memory and type safety

```
MyType* a = new MyType(); /* or: malloc(sizeof(MyType)) */
…
delete a;  /* or: free(a) */
…
a->f = …;
```

Memory and type safety

```
MyType* a = new MyType();
…
delete a;
…
a->f = …;
```

Memory and type safety

```
MyType* a = new MyType();
…
delete a;
…
a->f = …;
```

C++: Undefined behavior

Java: Garbage collection → no explicit freeing

Memory and type safety

```
MyType* a = new MyType();
…
delete a;
…
delete a;
```

Memory and type safety

```
MyType* a = new MyType();
…
delete a;
…
delete a;
```

C++: Undefined behavior

Java: Garbage collection → no explicit freeing

Memory and type safety

```
MyType* a = new MyType();
container->data = a;
…
container->data = NULL; /* Last ptr to a is lost */
```

Memory and type safety

MyType* a = new MyType();

container->data = a;

…

container->data = NULL; /* **Last** ptr to a is lost */

C++: Memory leak

Java: Garbage collection

How? Knows all reference types and all “roots”; approximates liveness via transitive reachability

Outline

- Classes and objects
- Methods
  - Inheritance, polymorphism
  - Static methods and fields
- Implementation: compilation, allocation
- Types
- Memory and type safety
- **Memory Management**

Outline for memory management

- **Concepts:**
  - Live vs. dead objects
  - Explicit (manual) MM vs. automatic MM (GC)
  - GC overapproximates live set
  - Memory leaks
- Overview of GC, esp. tracing GC
- GC algorithms and strategies:
  - Mark-sweep GC
  - Copying GC
  - Generational GC

Live and dead objects

- **Live** object will be used in the future
- Other objects are **dead**
- Deallocate as soon as possible after last use (but not before!)

Memory management: Deallocate the **dead** objects in a “timely” fashion

Explicit (manual) memory management

- More code to maintain
- Requires global reasoning
- Correctness
  - Free a live object
  - Free a dead object too late (or never)
- Efficiency can be very high
  - Gives programmers more control over the run-time behavior of the program

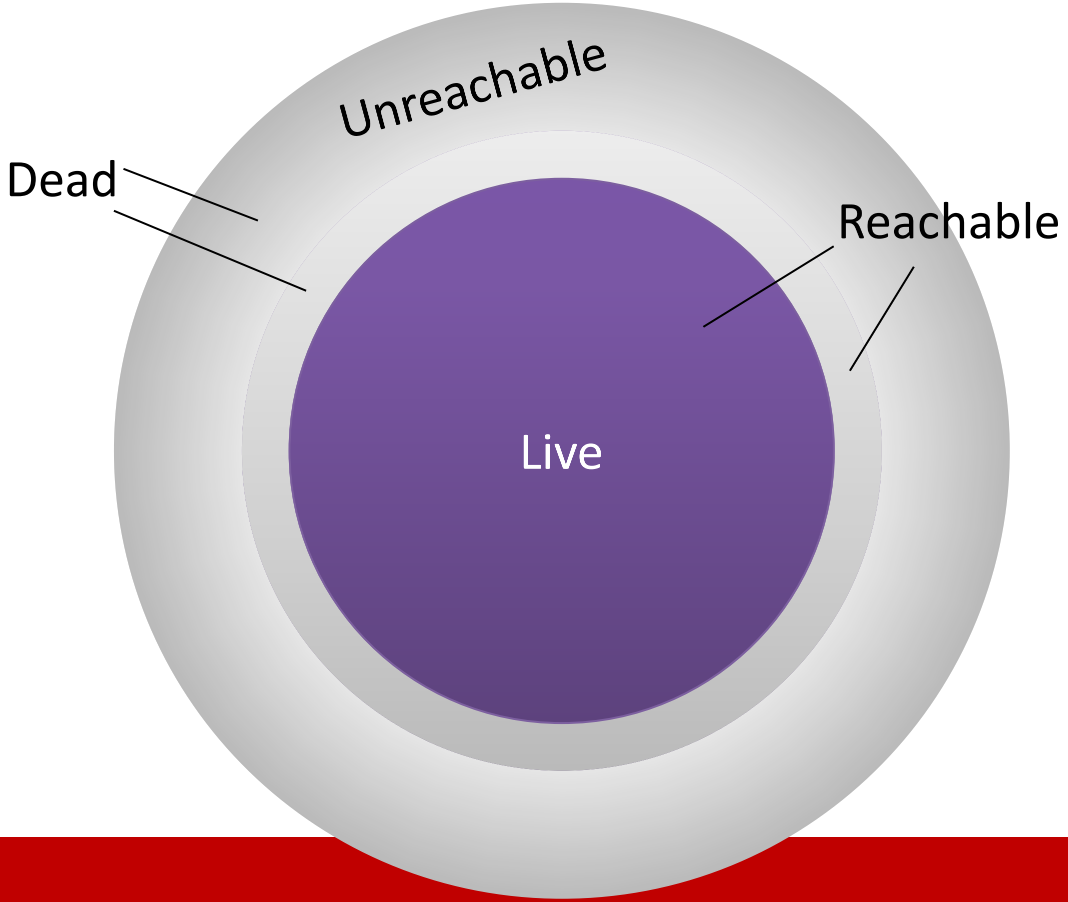
Automatic memory management (garbage collection)

- Integral for memory and type safety
  - Protects against some classes of memory errors (dangling pointers, double frees)
  - Essential for Java, C#, PHP, JavaScript, …
- Reduces programmer burden
- Not perfect, memory can still leak
  - Programmers still need to eliminate all pointers to objects the program no longer needs
- (A mostly solved) challenge: performance

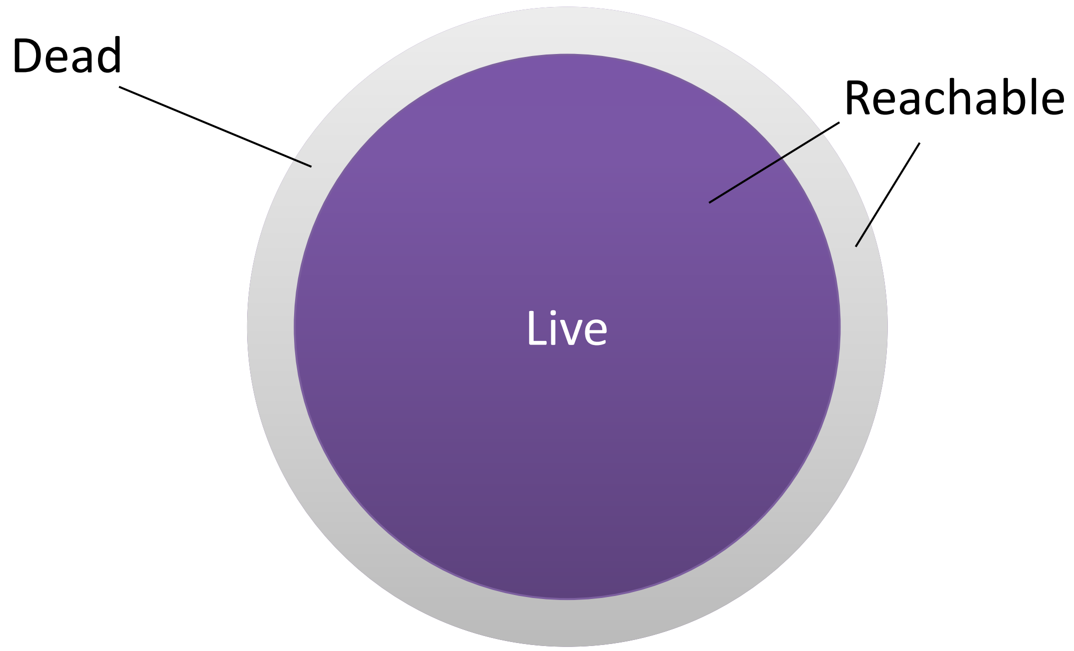
What is Garbage?

- In theory, any object the program will never reference again
  - But compiler & runtime system cannot figure that out
- In practice, any object the program cannot reach is garbage
  - Approximate **liveness** with **reachability**
- Managed languages couple GC with “safe” pointers
  - Programs may not access arbitrary addresses in memory (e.g., Java/C# vs. C/C++)
  - The compiler can identify and provide to the garbage collector all the pointers, thus enforcing “Once garbage, always garbage”
  - Runtime system can move objects by updating pointers

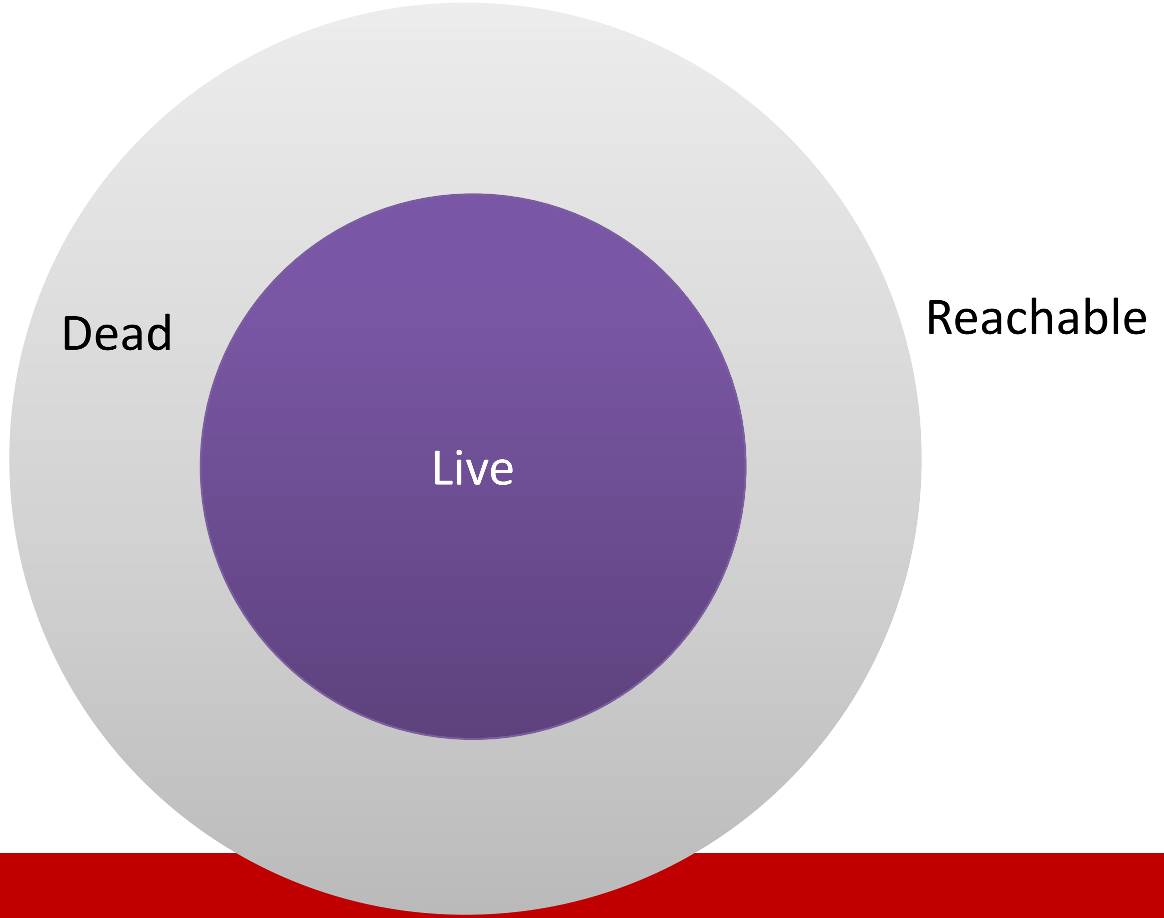
Liveness under-approximates reachability
(Reachability over-approximates liveness)



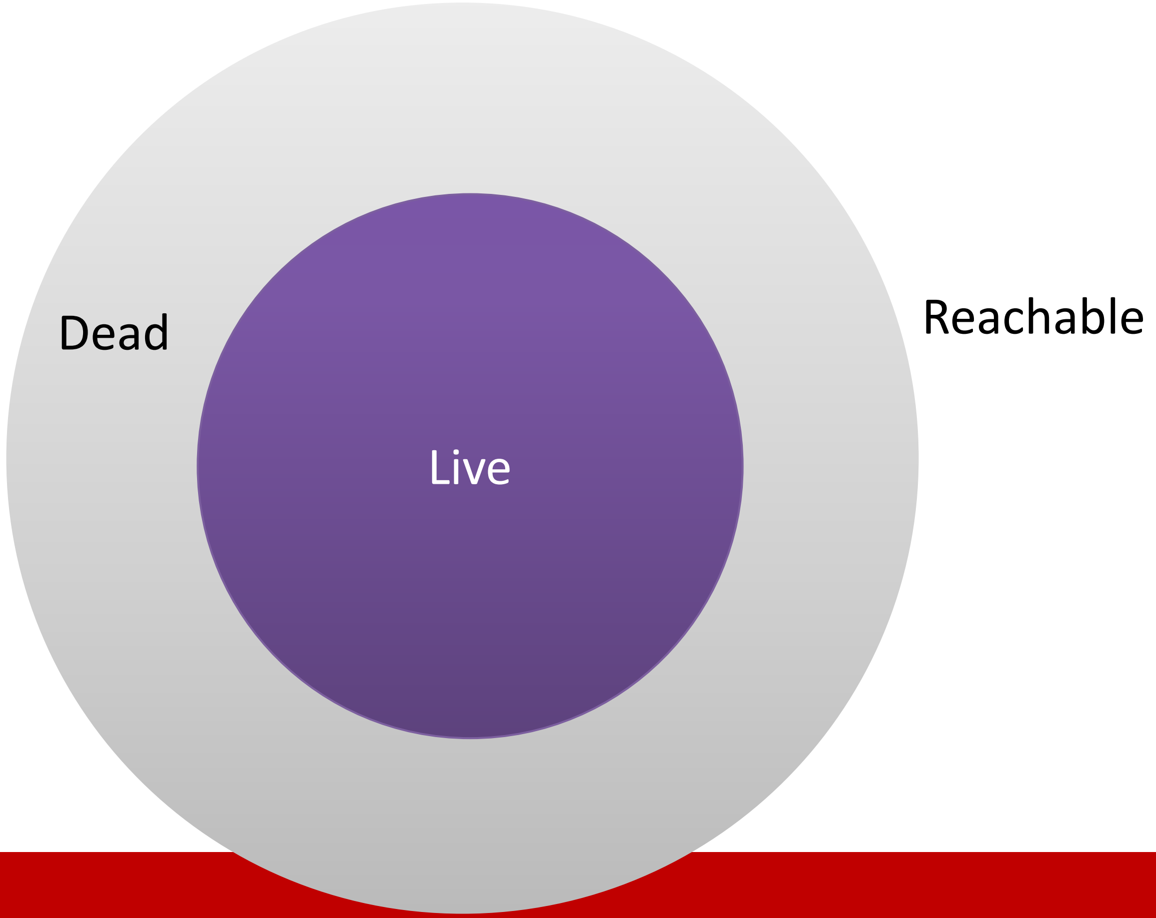
Liveness under-approximates reachability
(Reachability over-approximates liveness)



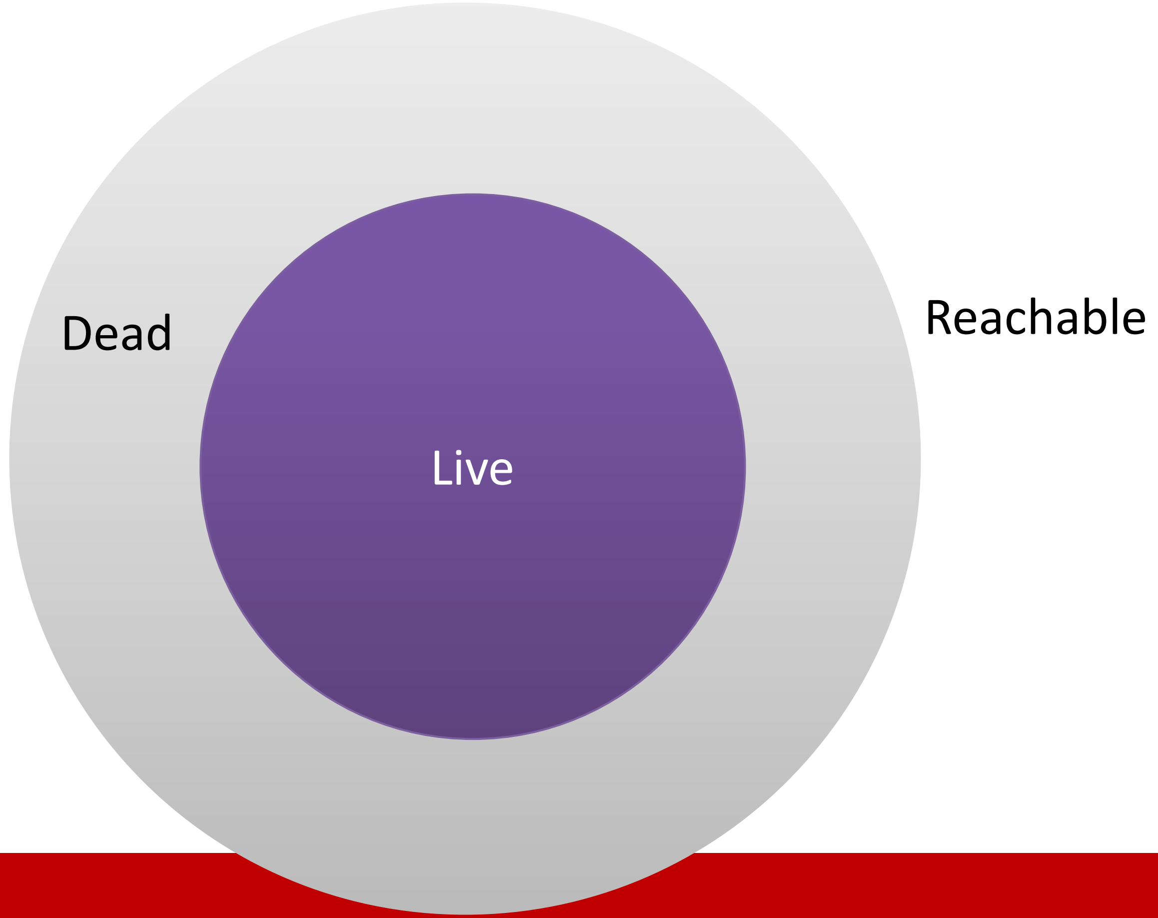
- Can leaks happen in GC'd languages?



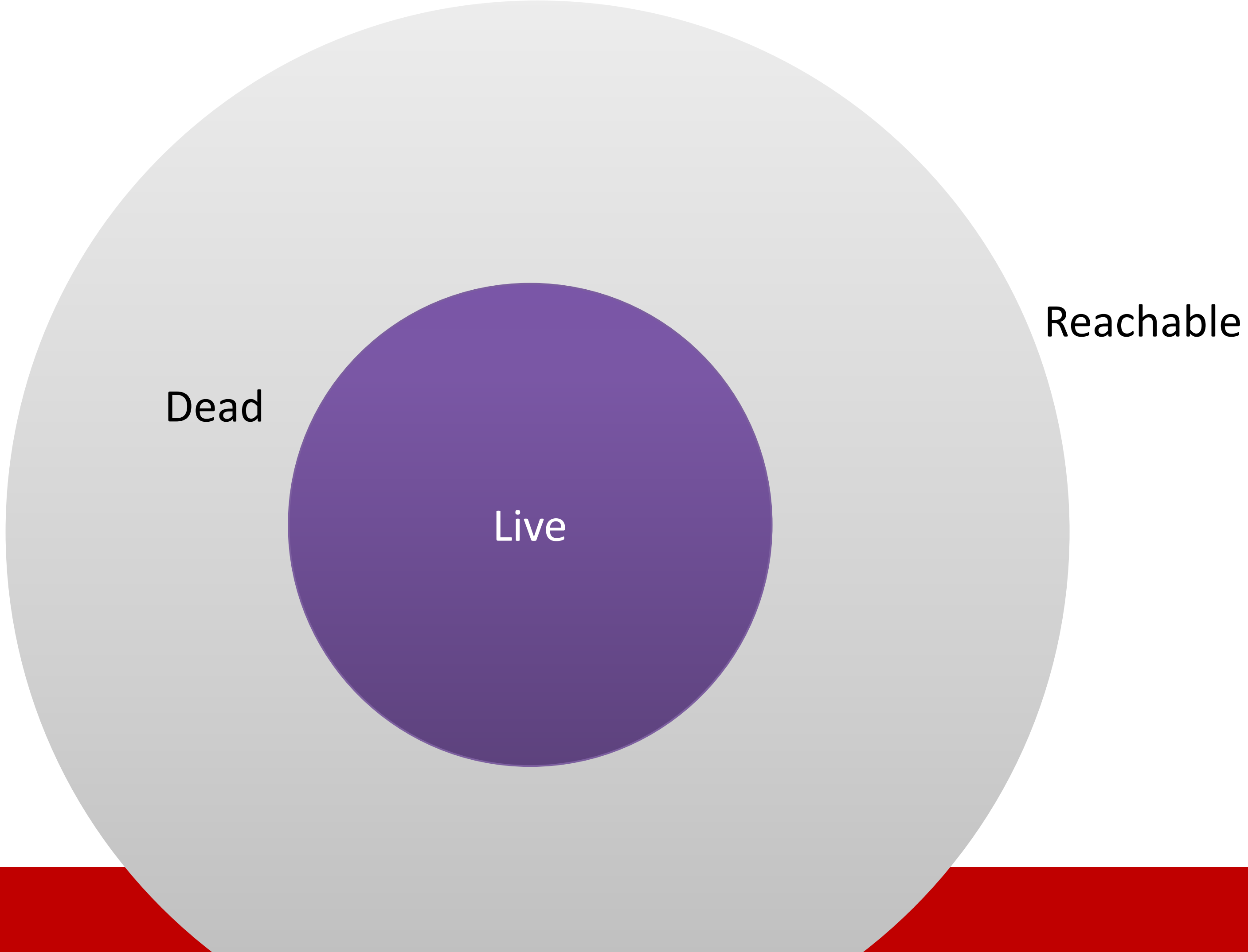
- Can leaks happen in GC'd languages?



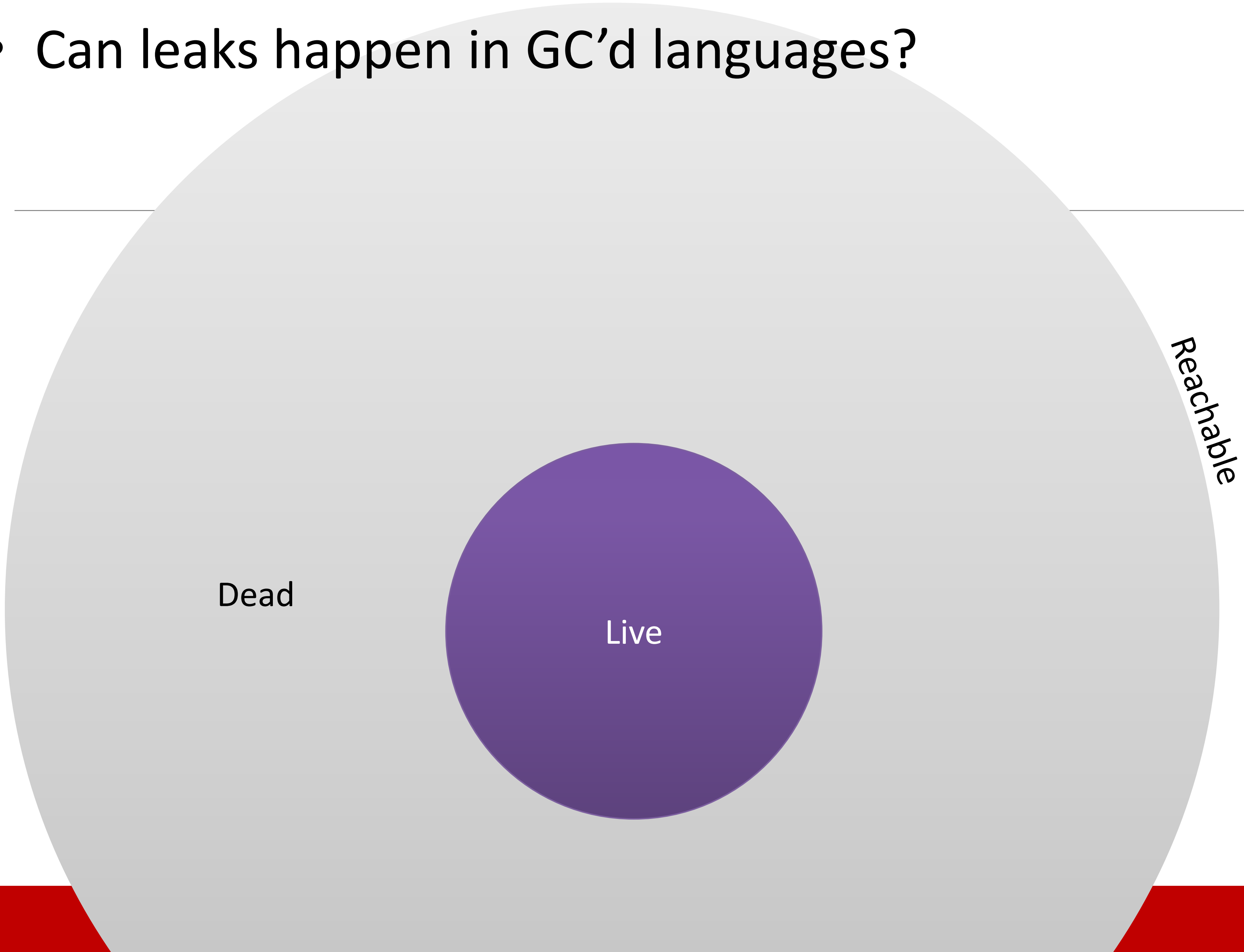
- Can leaks happen in GC'd languages?



- Can leaks happen in GC'd languages?



- Can leaks happen in GC'd languages?



- Can leaks happen in GC'd languages?

Dead

Memory leak example

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx

- Driverless truck
  - 10,000 lines of C#
- Leak: past obstacles remained reachable

Memory leak example

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx

- Driverless truck
  - 10,000 lines of C#
- Leak: past obstacles remained reachable
- No immediate symptoms

  “This problem was pernicious because it only showed up after 40 minutes to an hour of driving around and collecting obstacles.”

Memory leak example

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx

- Driverless truck
  - 10,000 lines of C#
- Leak: past obstacles remained reachable
- No immediate symptoms

  "This problem was pernicious because it only showed up after 40 minutes to an hour of driving around and collecting obstacles."
- Quick "fix": restart after 40 minutes

Memory leak example

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx

- Driverless truck
  - 10,000 lines of C#
- Leak: past obstacles remained reachable
- No immediate symptoms

  "This problem was pernicious because it only showed up after 40 minutes to an hour of driving around and collecting obstacles."
- Quick "fix": restart after 40 minutes
- Environment sensitive
  - More obstacles in deployment
  - Unresponsive after 28 minutes

Outline for memory management

- Concepts:
  - Live vs. dead objects
  - Explicit (manual) MM vs. automatic MM (GC)
  - GC overapproximates live set
  - Memory leaks
- **Overview of GC, esp. tracing GC**
- GC algorithms and strategies:
  - Mark-sweep GC
  - Copying GC
  - Generational GC

GC basics

Two types (duals of each other):

- Reference counting
  - Work proportional to dead objects
  - Memory freed immediately
  - Cycles are problematic
- Tracing
  - Work proportional to live objects
  - Freeing postponed
  - Can be concurrent

How does tracing GC work?

Roots

- Local variables: registers & stack locations
- Static variables

Transitive closure

Memory & type safety ensure GC knows the roots and references exactly

How does tracing GC work?

When does it happen?

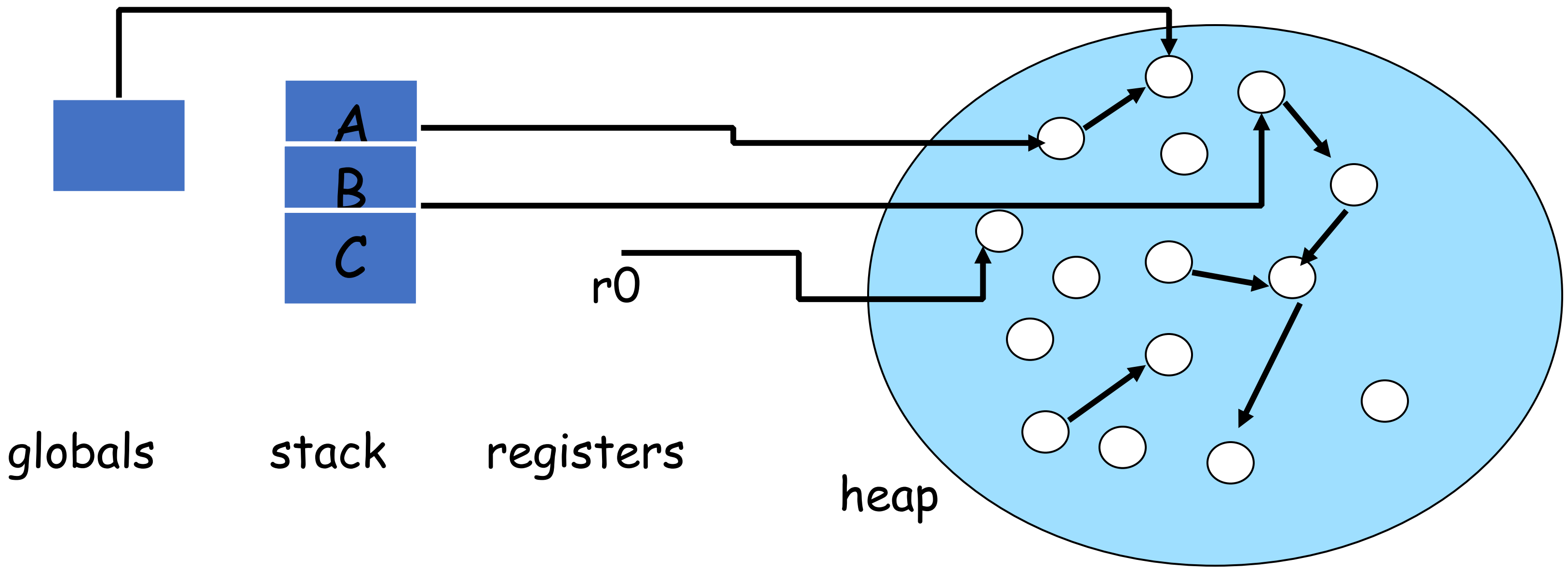
- Stop-the-world: safe points inserted by VM
- Concurrent
- Incremental

How many GC threads?

- Single-threaded
- Parallel

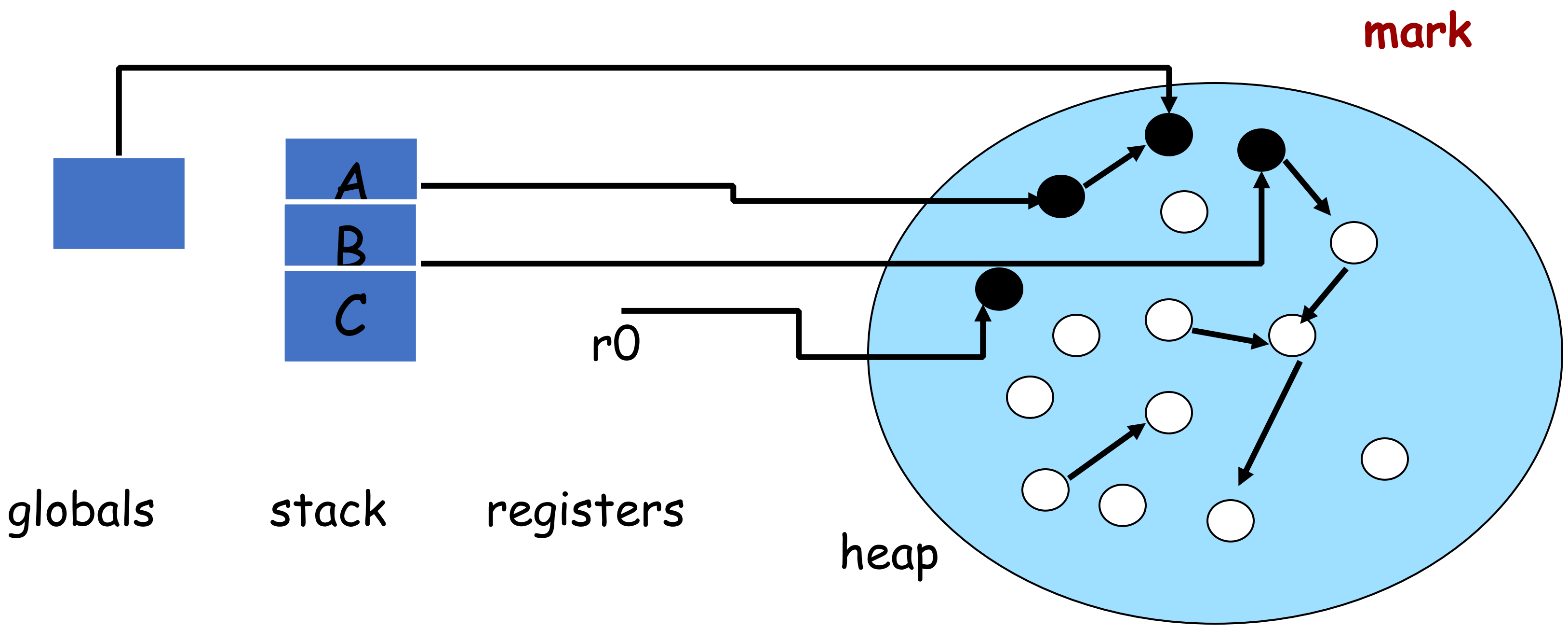
Reachability

- The runtime memory management system examines all global variables, stack variables, and live registers that could refer to heap objects (i.e., the **roots** of reachability)
- GC threads can **trace** these pointers through the heap (following object fields that themselves point to heap objects) to find all reachable objects



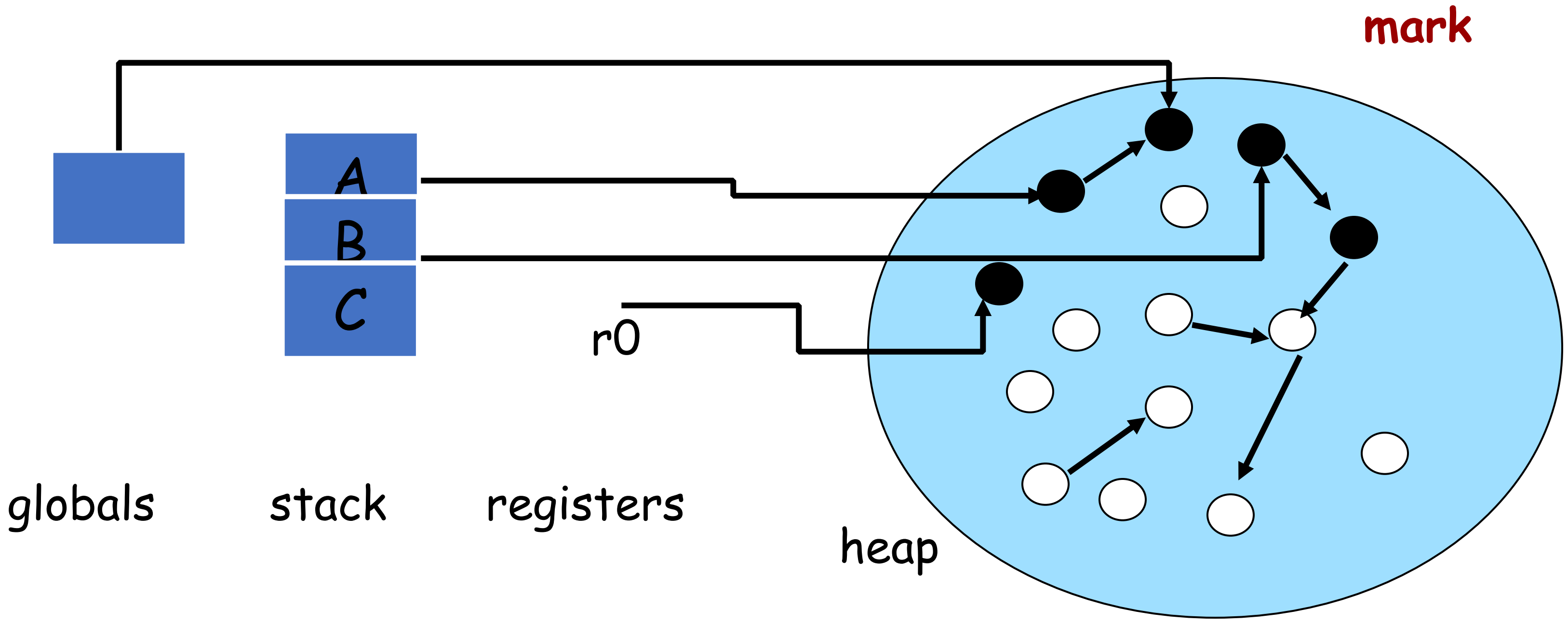
Reachability

- Tracing collector
  - Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them



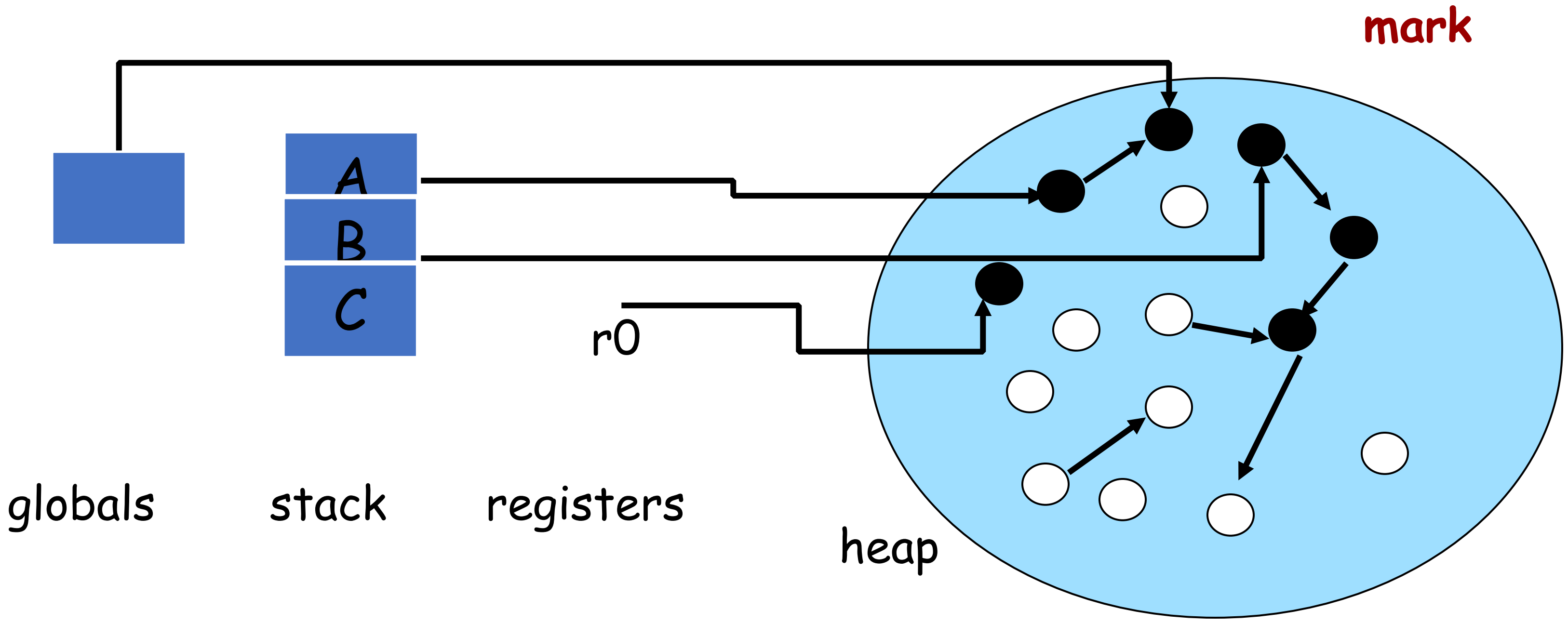
Reachability

- Tracing collector
  - Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them



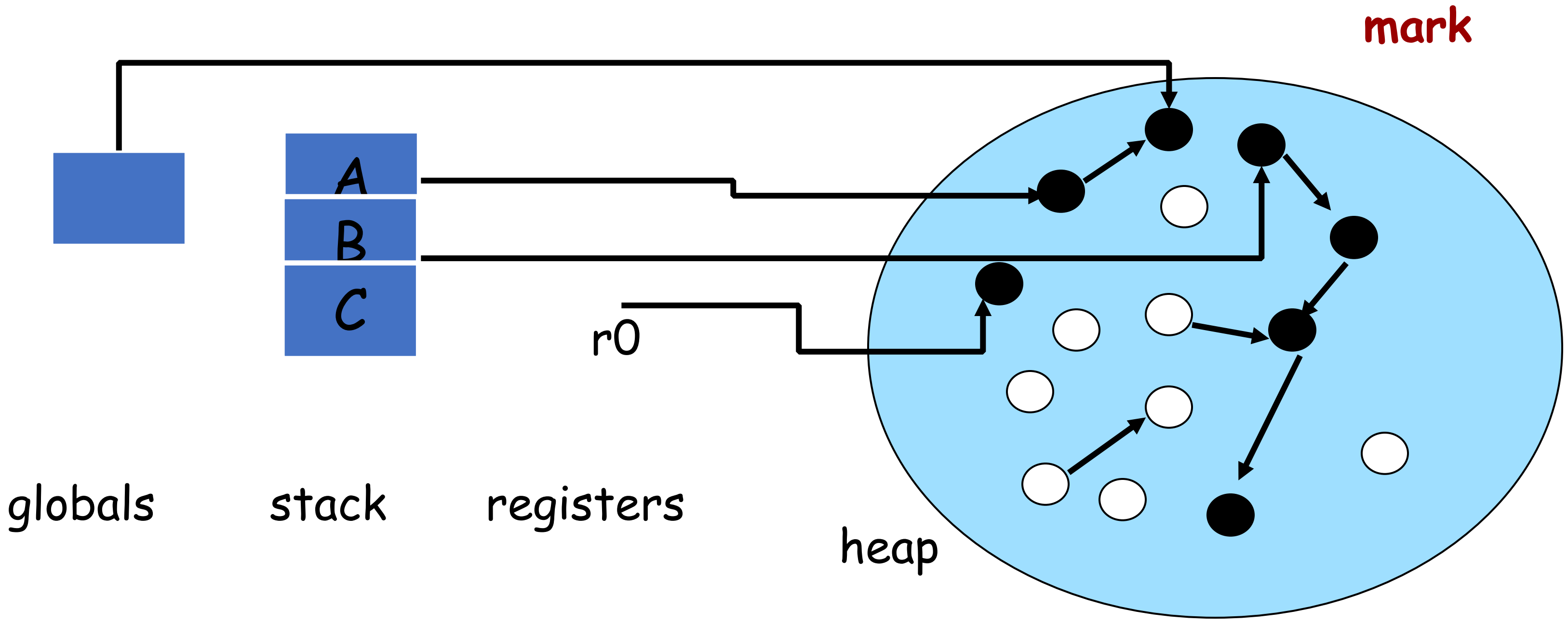
Reachability

- Tracing collector
  - Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them



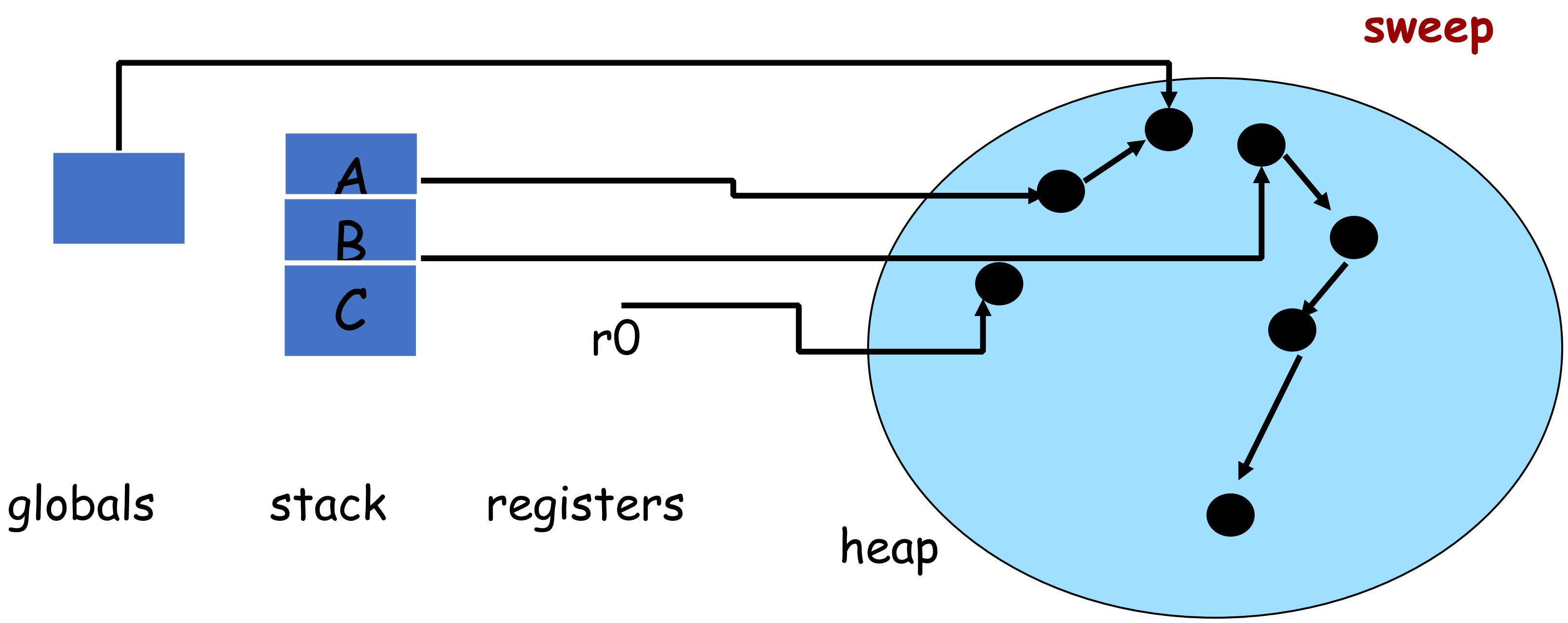
Reachability

- Tracing collector
  - Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them
- All unmarked objects are **dead**



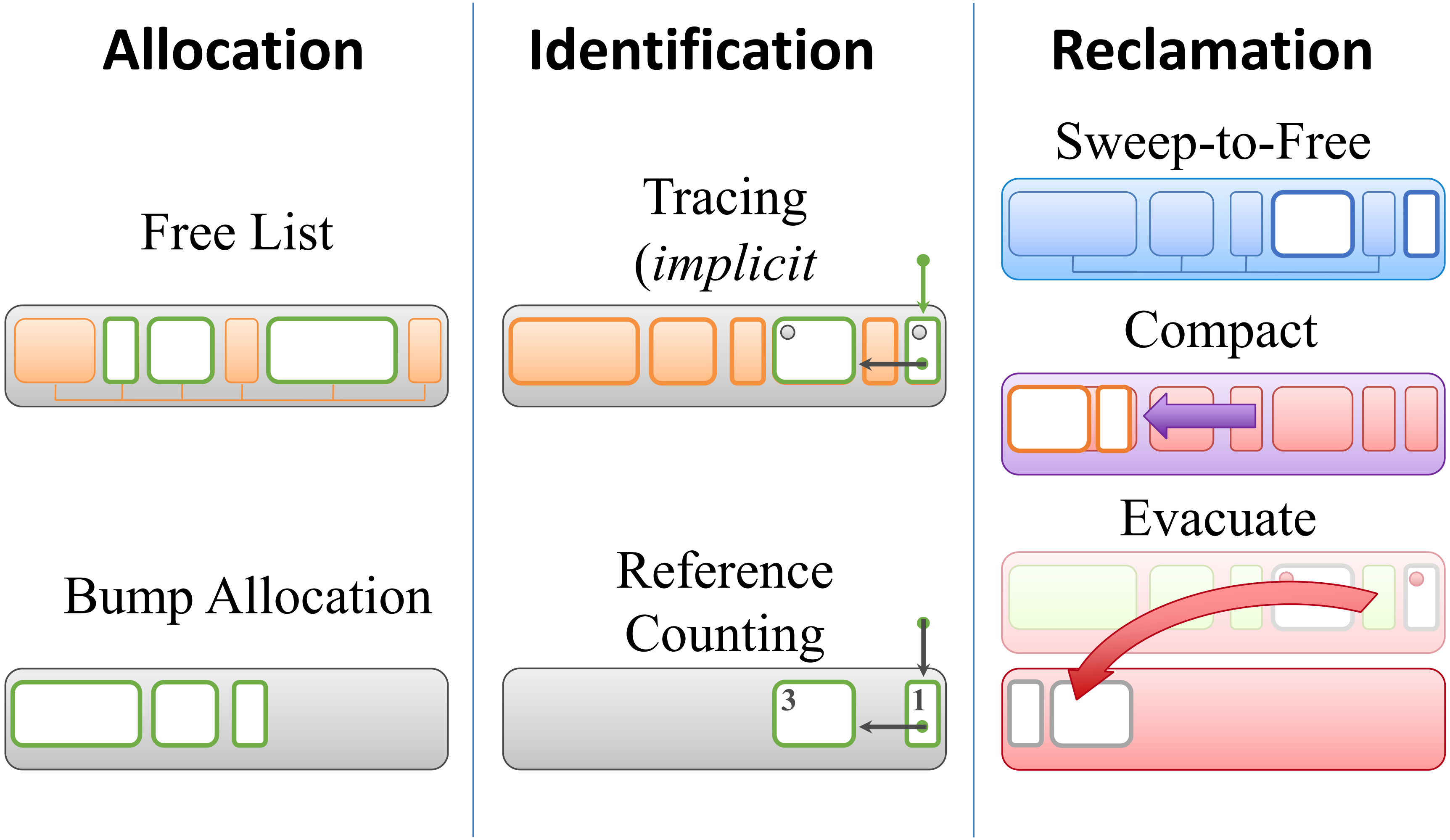
Reachability

- Tracing collector
  - Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them
- All unmarked objects are **dead**



The Big Picture

- Heap organization; basic algorithmic components

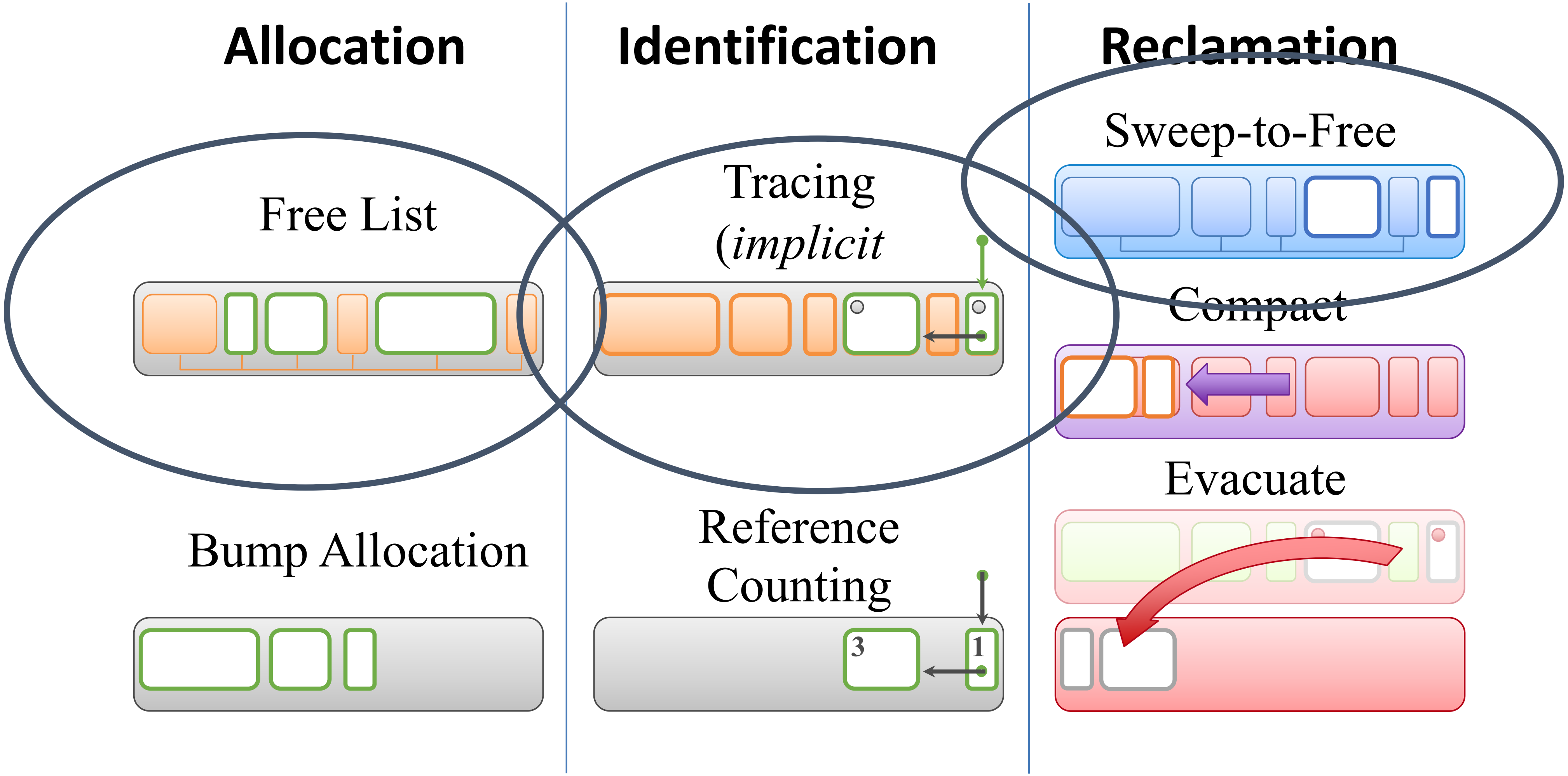


Outline for memory management

- Concepts:
  - Live vs. dead objects
  - Explicit (manual) MM vs. automatic MM (GC)
  - GC overapproximates live set
  - Memory leaks
- Overview of GC, esp. tracing GC
- GC algorithms and strategies:
  - **Mark-sweep GC**
  - Copying GC
  - Generational GC

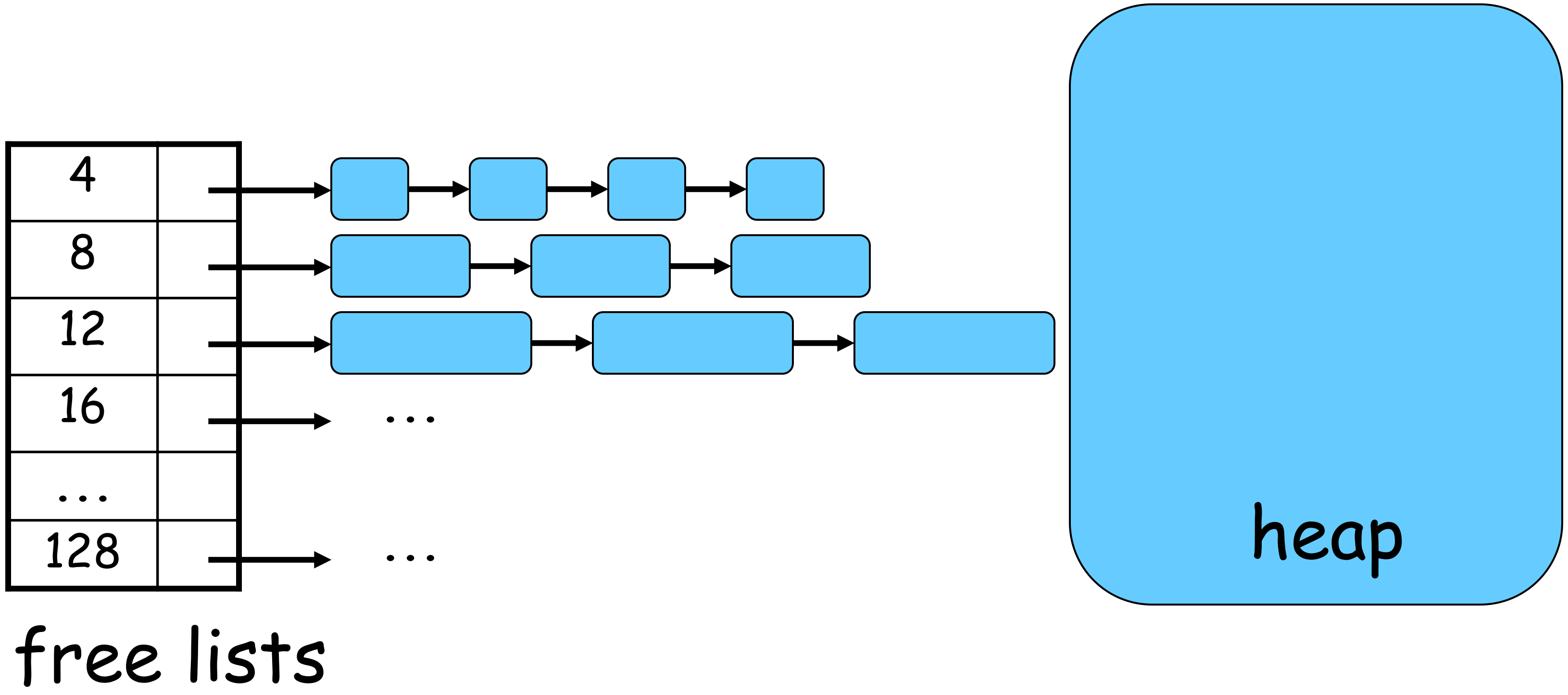
The Big Picture

- Heap organization; basic algorithmic components



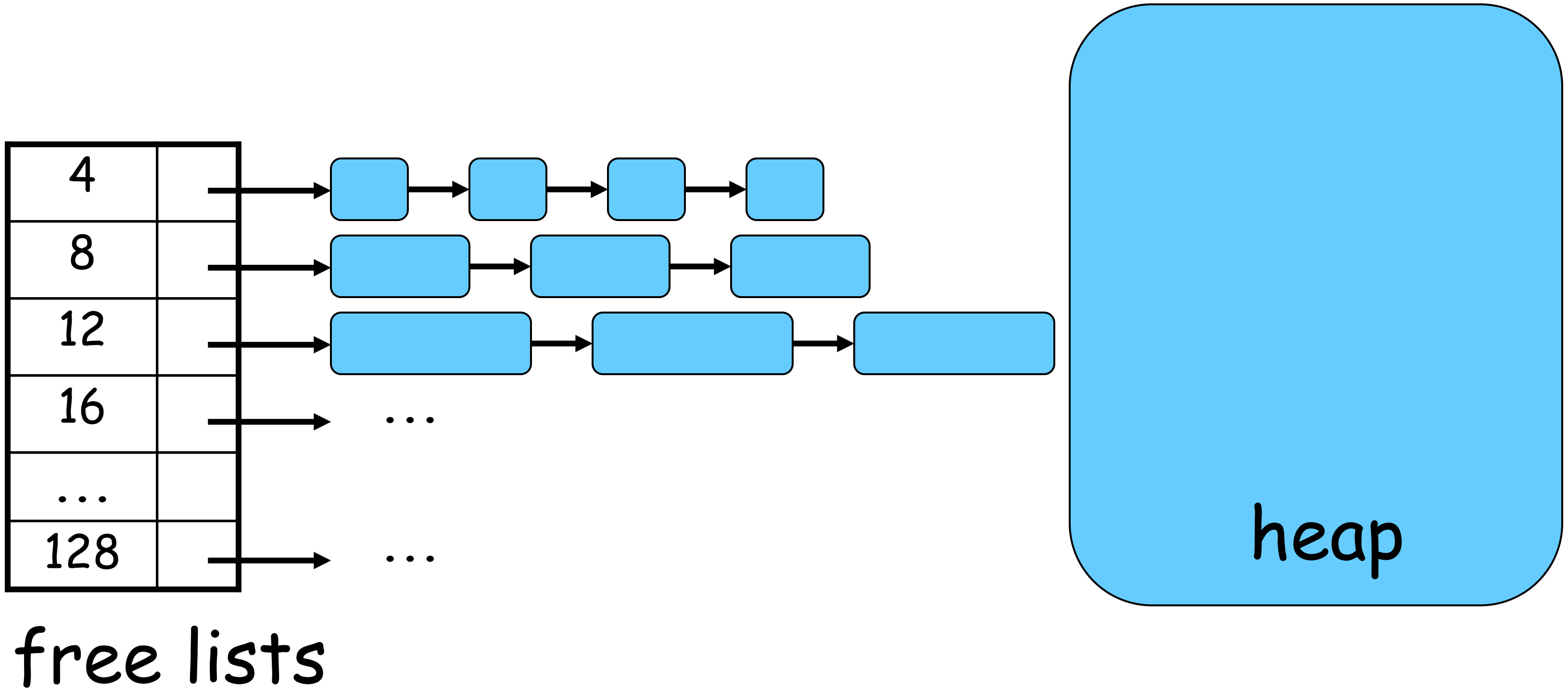
Mark-and-Sweep Implementation

- Free-lists organized by size
  - blocks of same size, or
  - individual objects of same size
- Most objects are small < 128 bytes



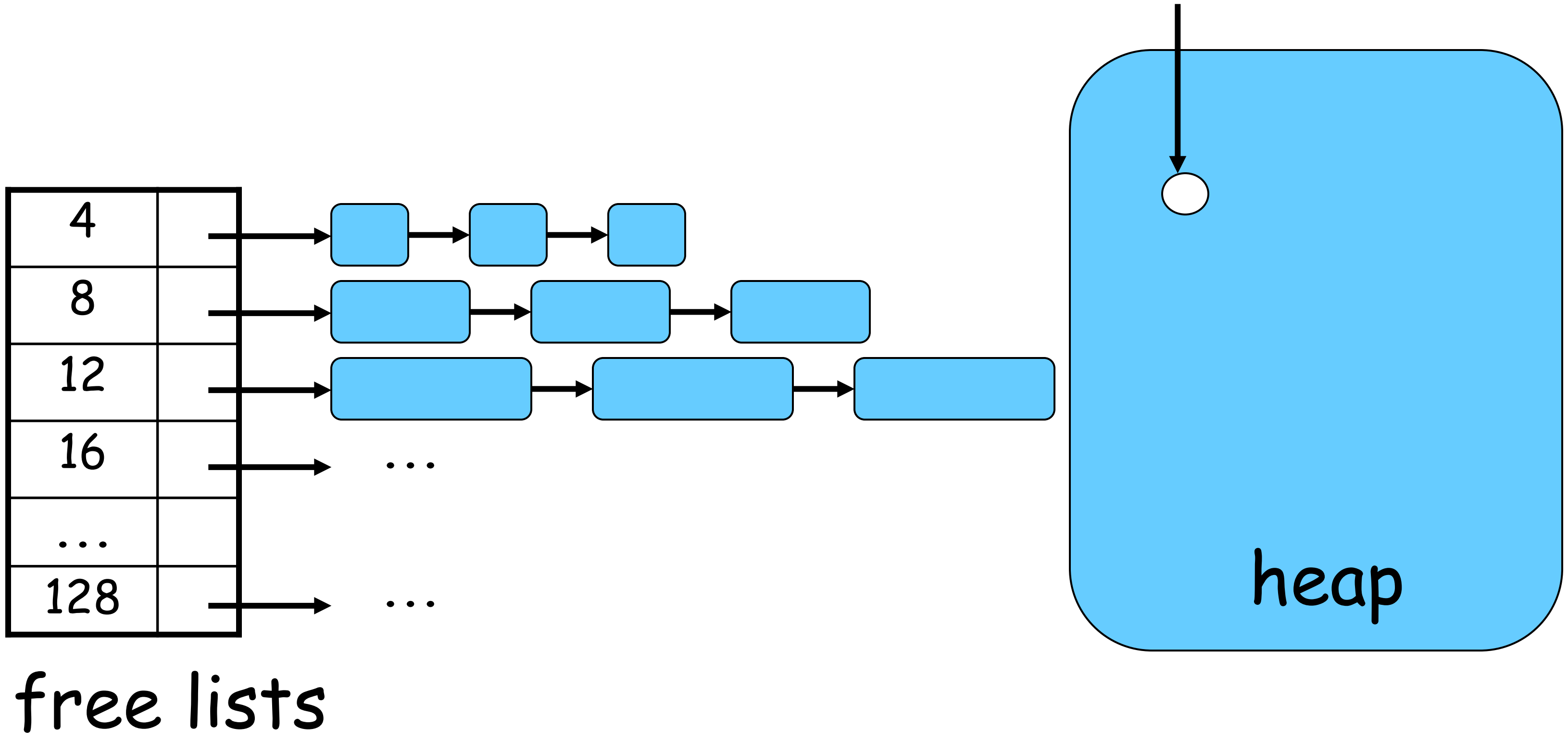
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list



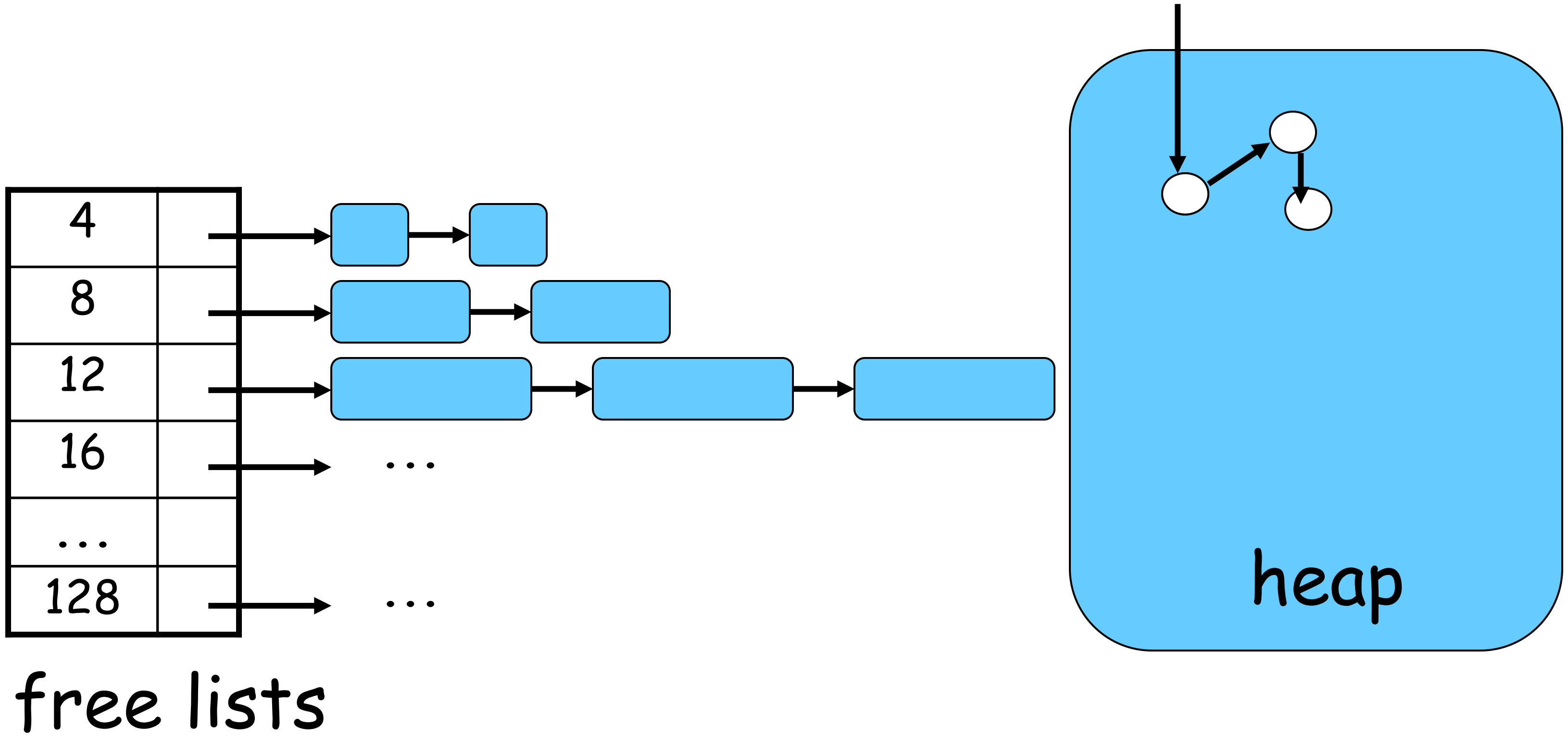
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list



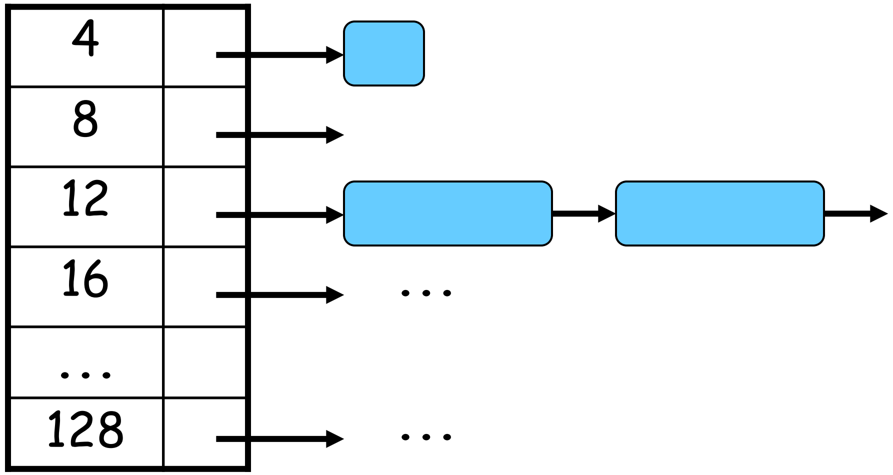
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list

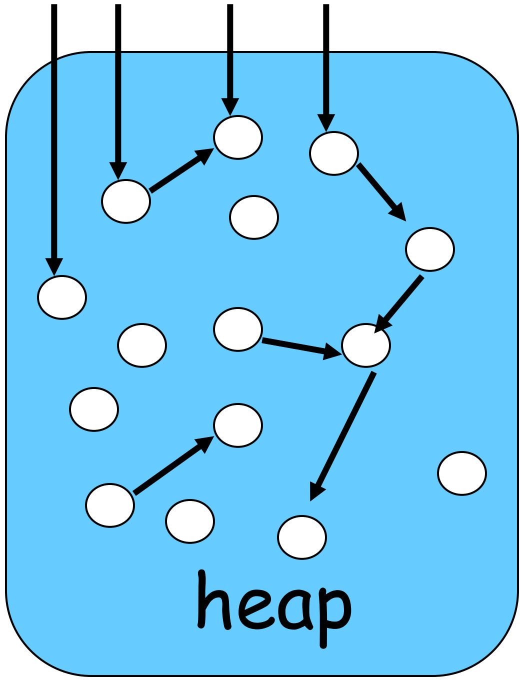


Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list
  - If there is no more memory of the right size, a garbage collection is triggered
  - Mark phase - find the live objects
  - Sweep phase - put free ones on the free list

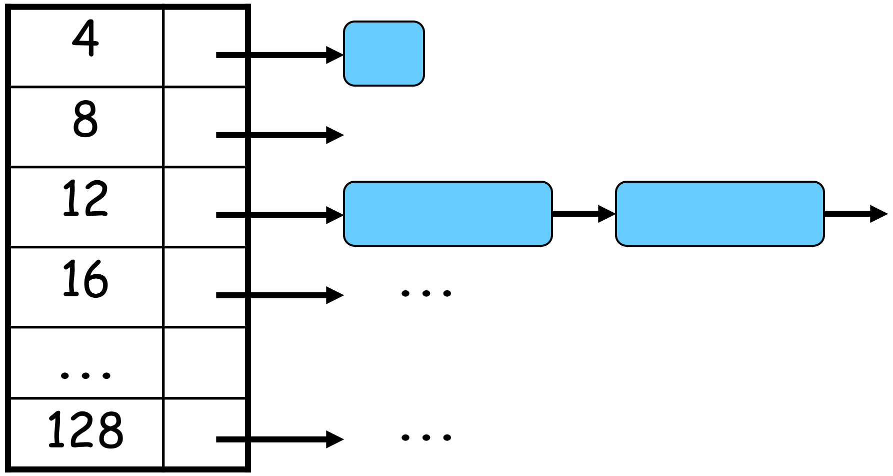


free lists

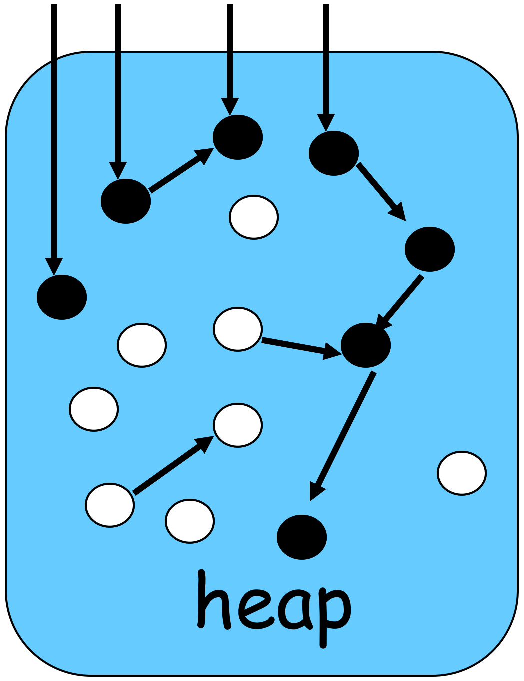


Mark-and-Sweep Implementation

- Mark phase
  - Reachability computation on the heap, marking all live objects
- Sweep phase
  - Sweep the memory for free objects, and populate the free lists

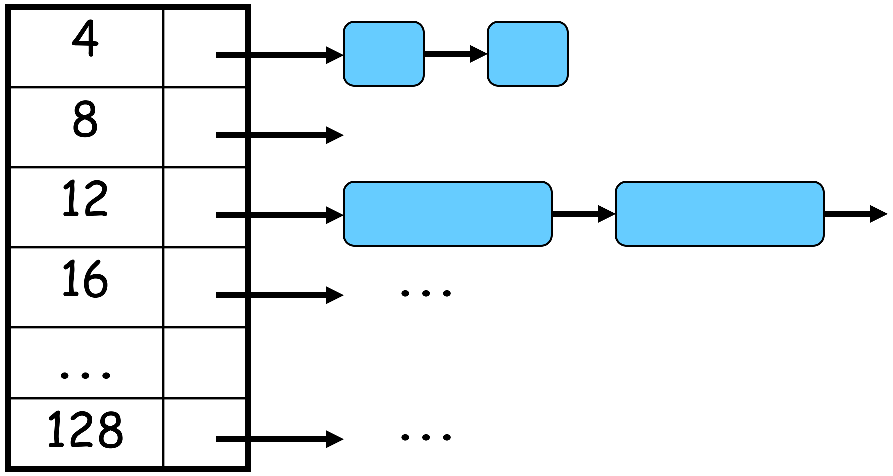


free lists

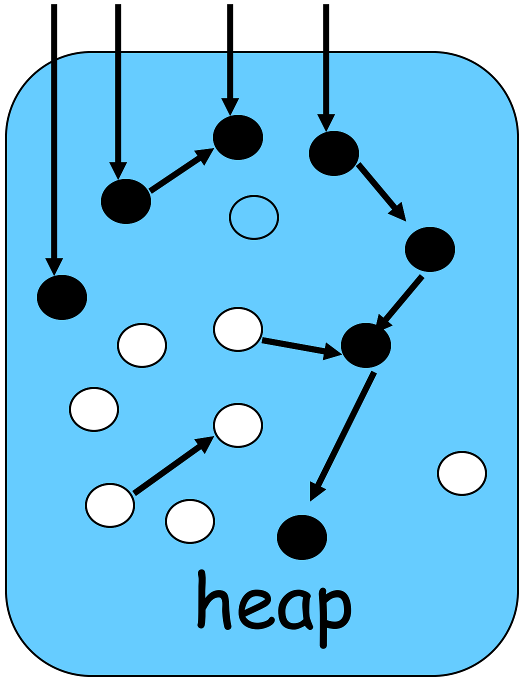


Mark-and-Sweep Implementation

- Mark phase
  - Reachability computation on the heap, marking all live objects
- Sweep phase
  - Sweep the memory for free objects, and populate the free lists

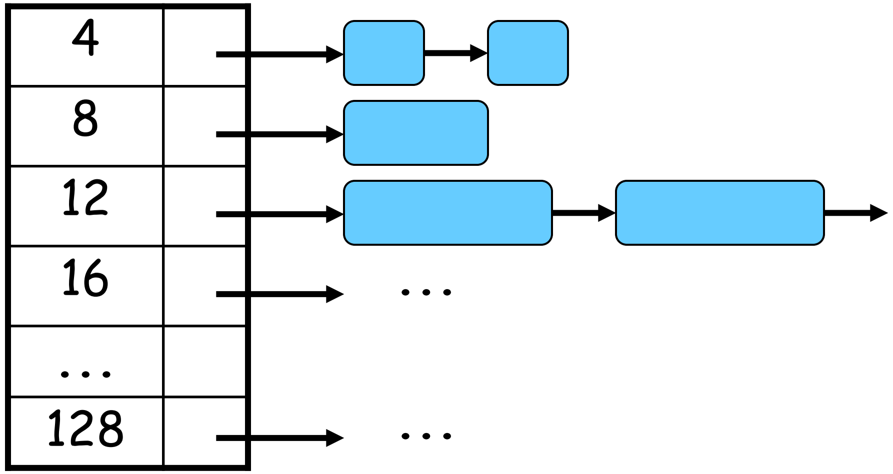


free lists

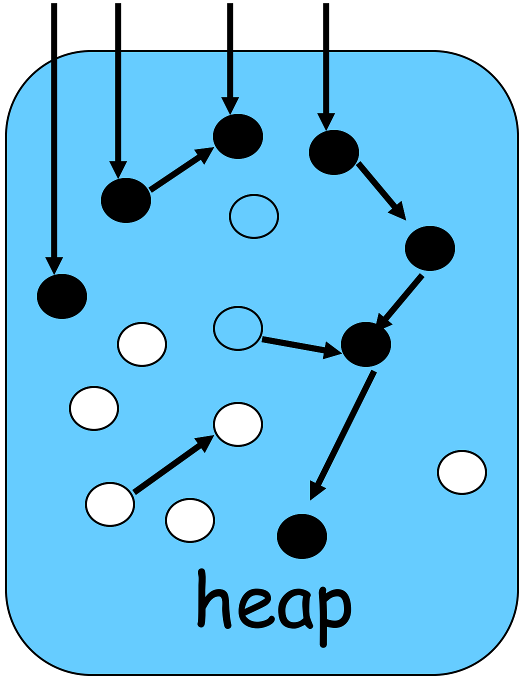


Mark-and-Sweep Implementation

- Mark phase
  - Reachability computation on the heap, marking all live objects
- Sweep phase
  - Sweep the memory for free objects, and populate the free lists

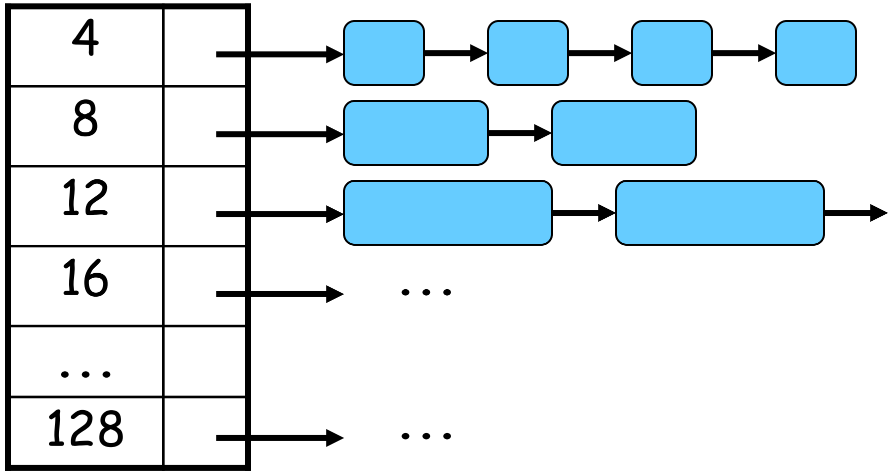


free lists

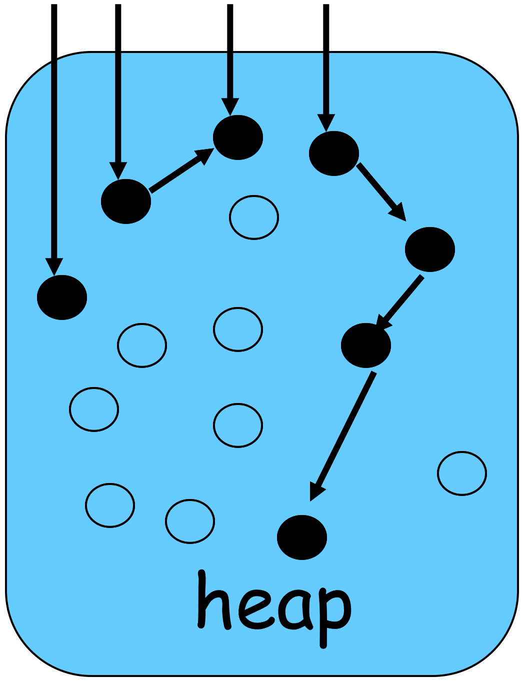


Mark-and-Sweep Implementation

- Mark phase
  - Reachability computation on the heap, marking all live objects
- Sweep phase
  - Sweep the memory for free objects, and populate the free lists



free lists

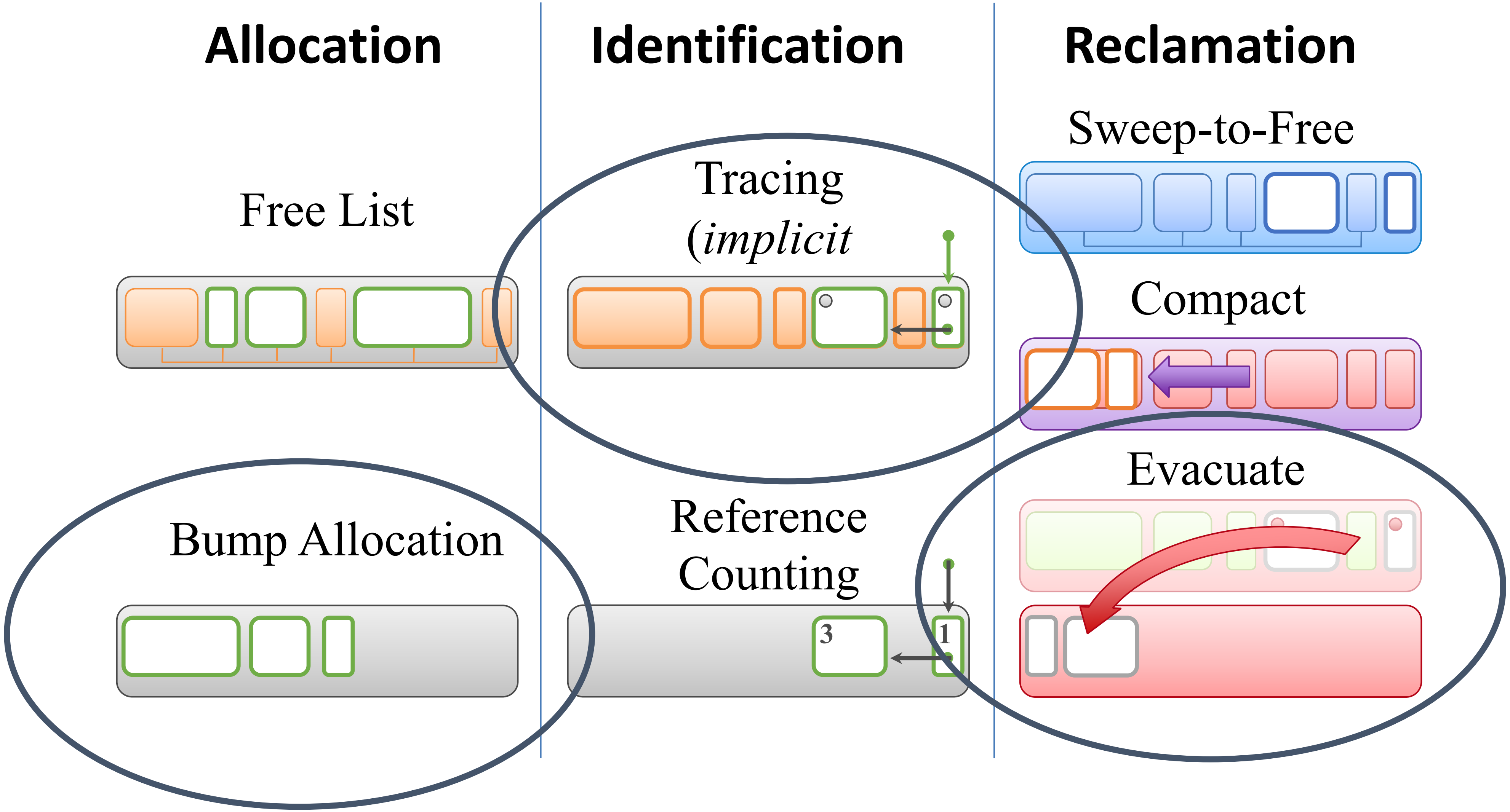


Outline for memory management

- Concepts:
  - Live vs. dead objects
  - Explicit (manual) MM vs. automatic MM (GC)
  - GC overapproximates live set
  - Memory leaks
- Overview of GC, esp. tracing GC
- GC algorithms and strategies:
  - Mark-sweep GC
  - **Copying GC**
  - Generational GC

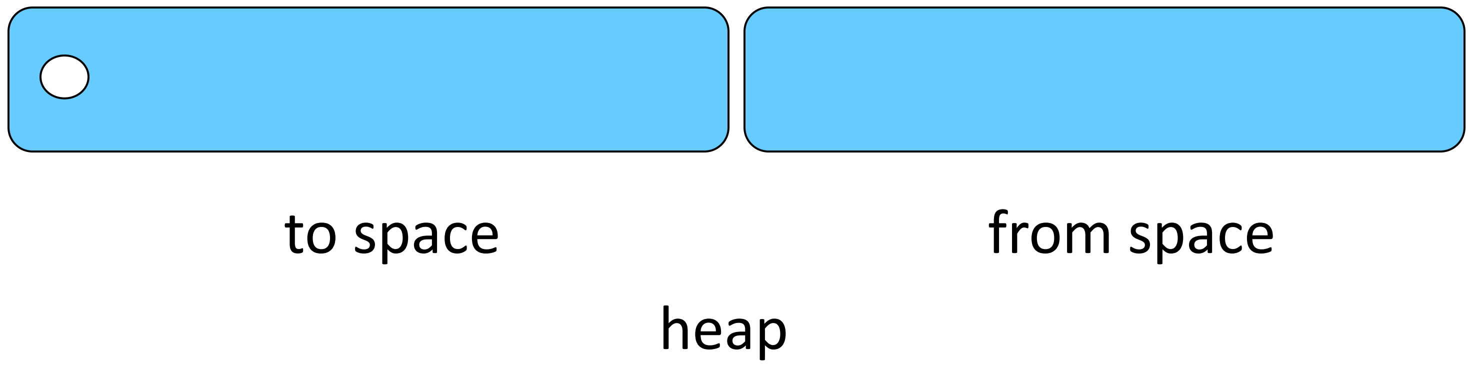
The Big Picture

- Heap organization; basic algorithmic components



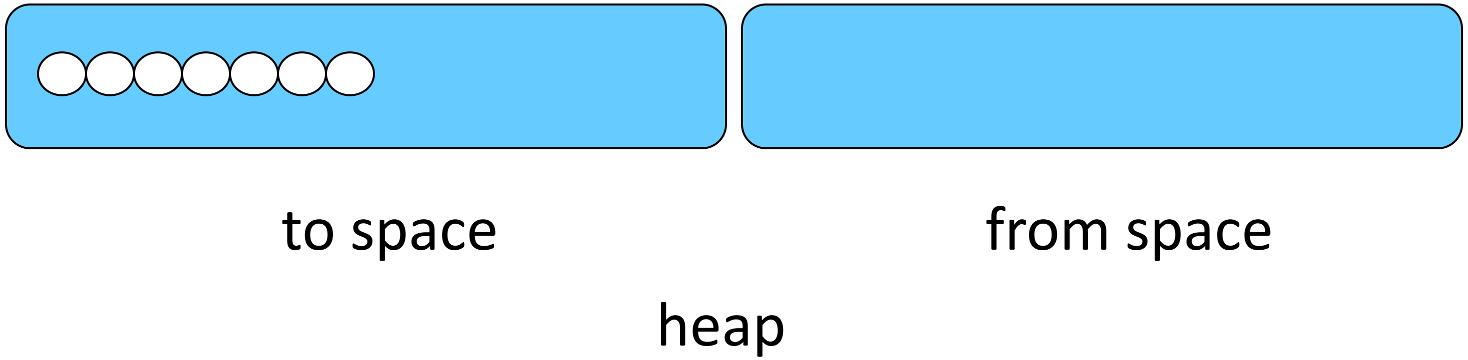
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live



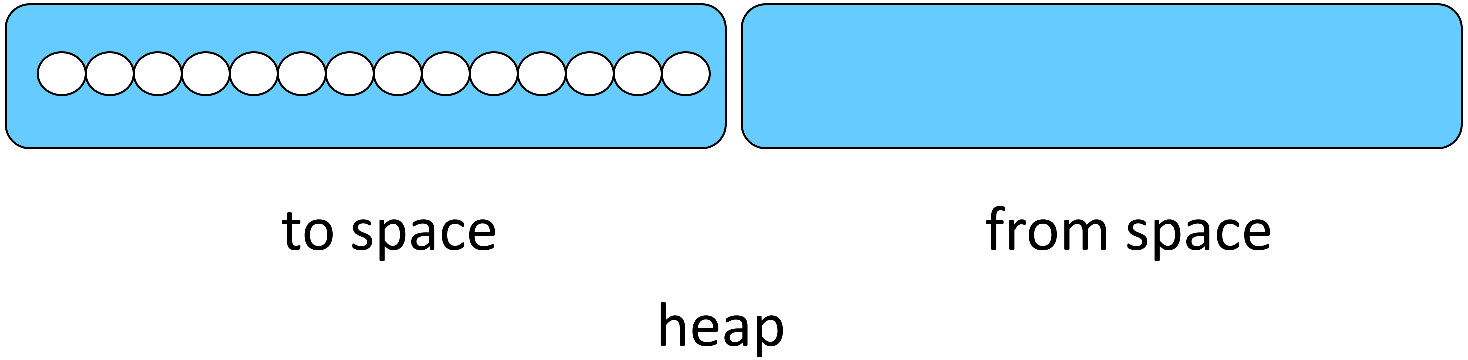
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live



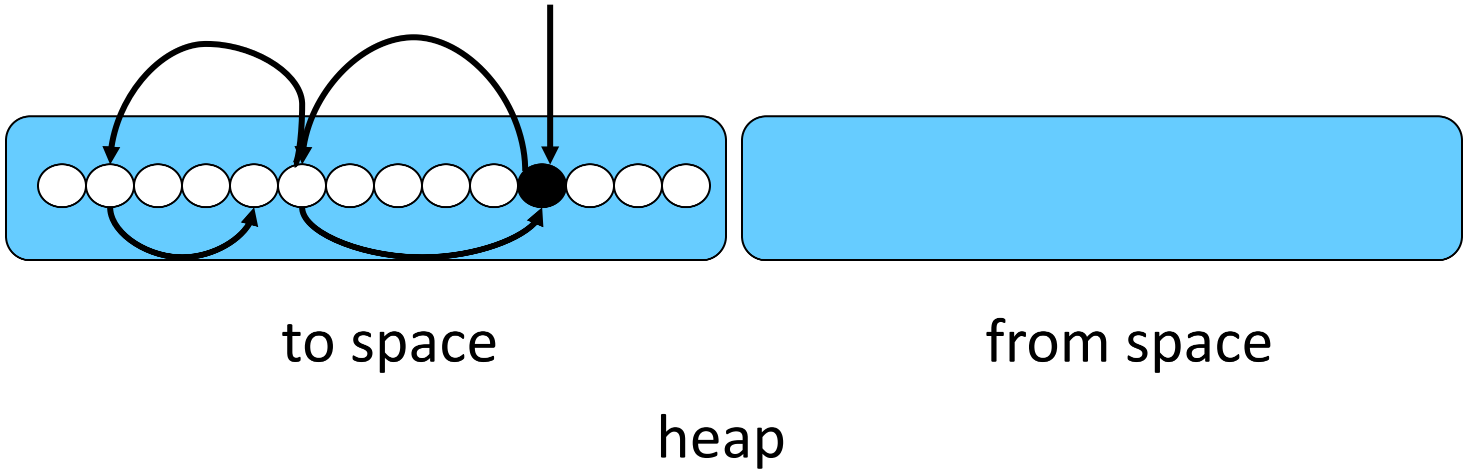
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live



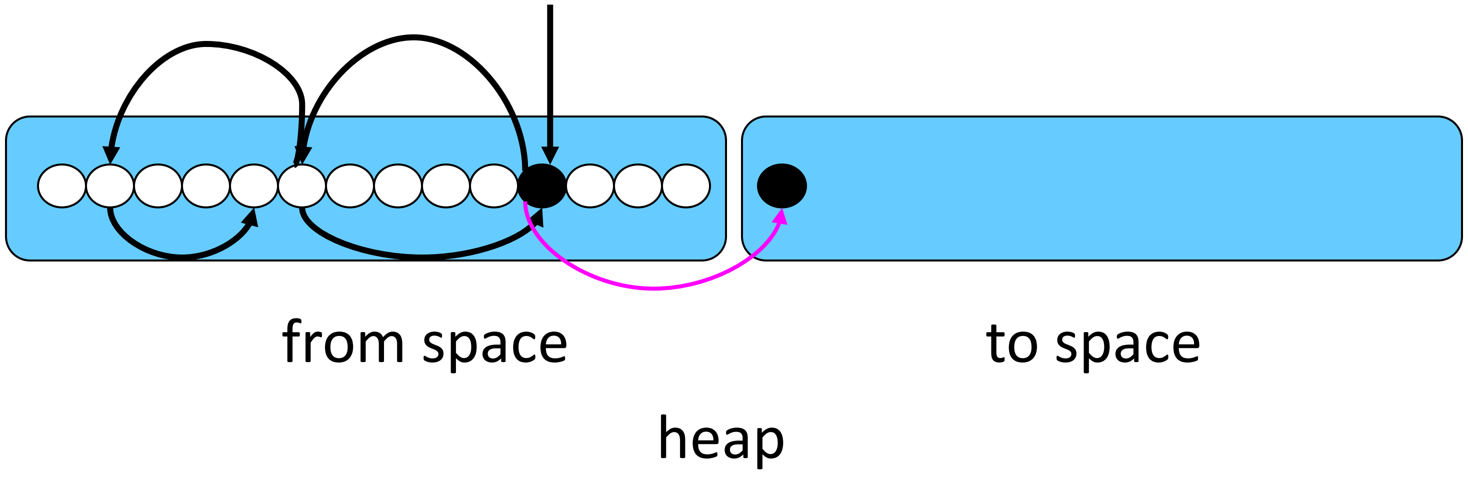
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live



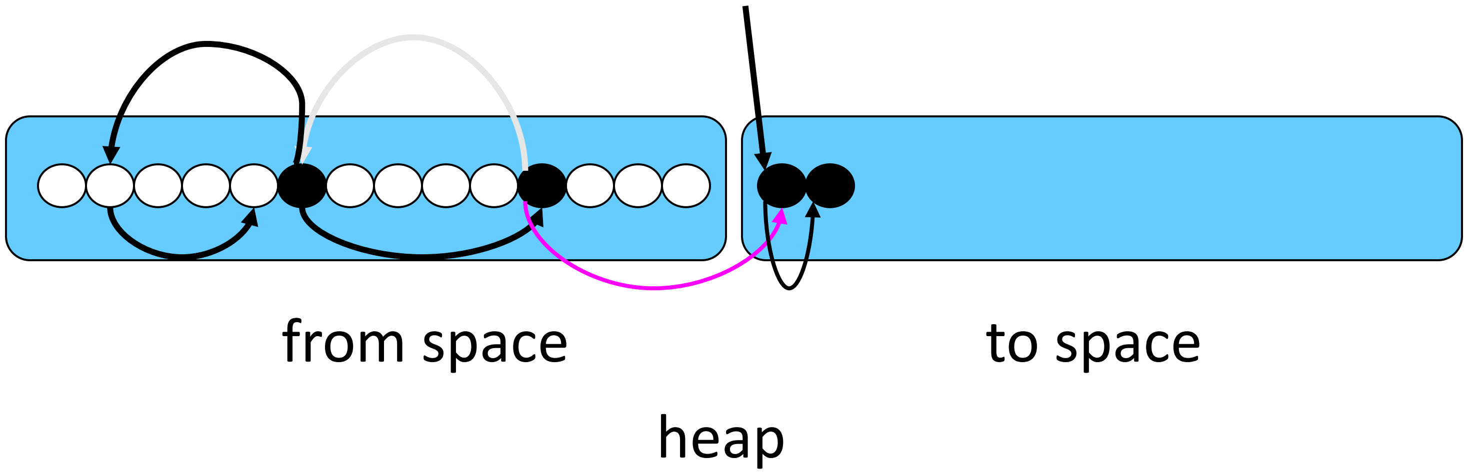
Semispace

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**



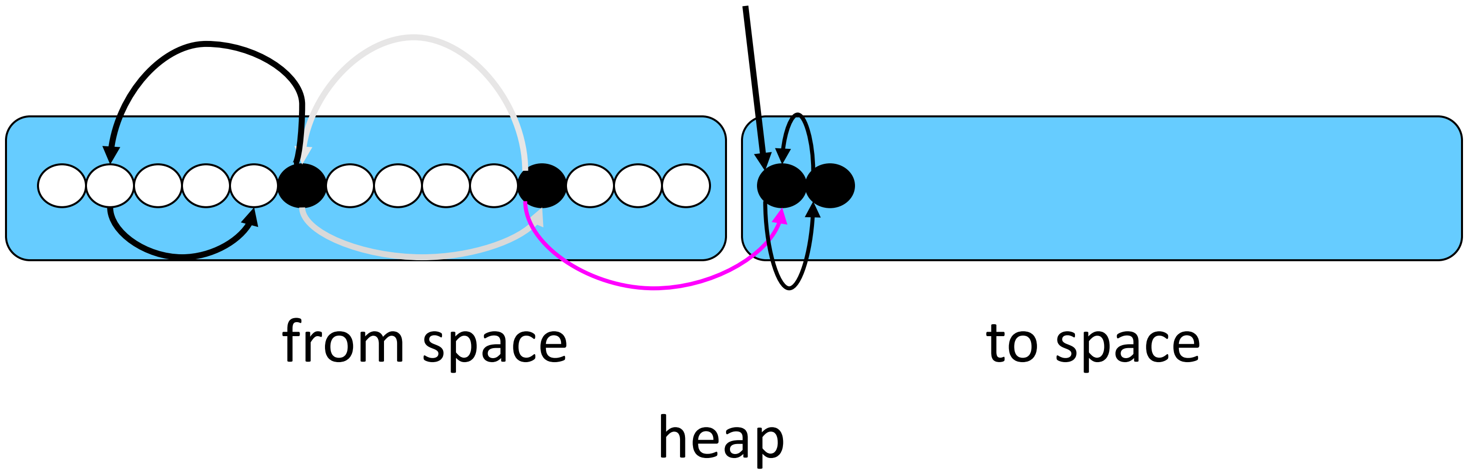
Semispace

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes



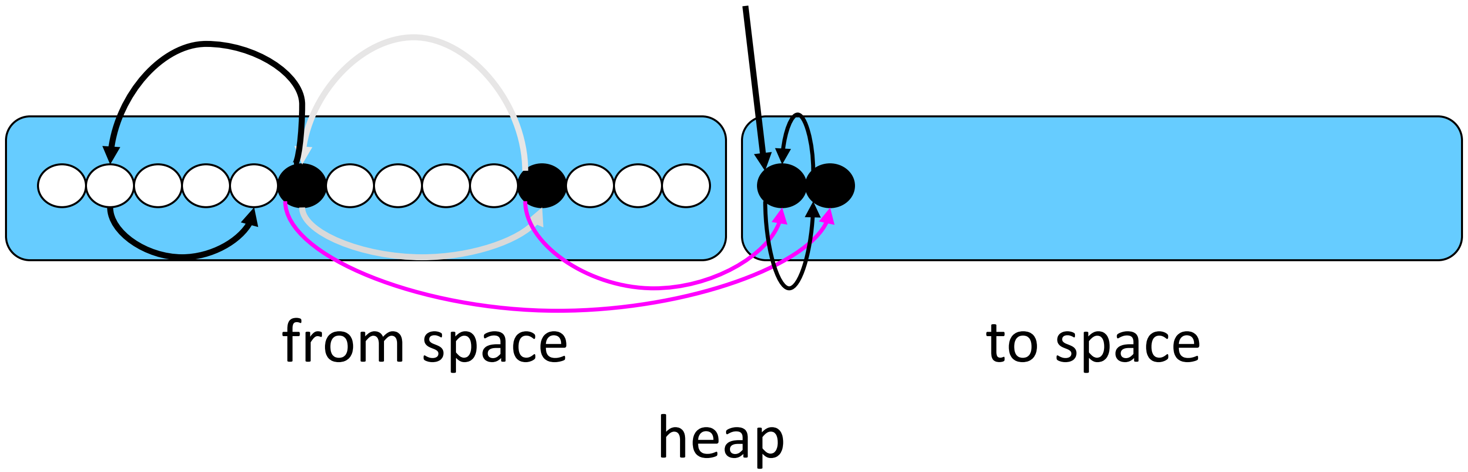
Semispace

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes



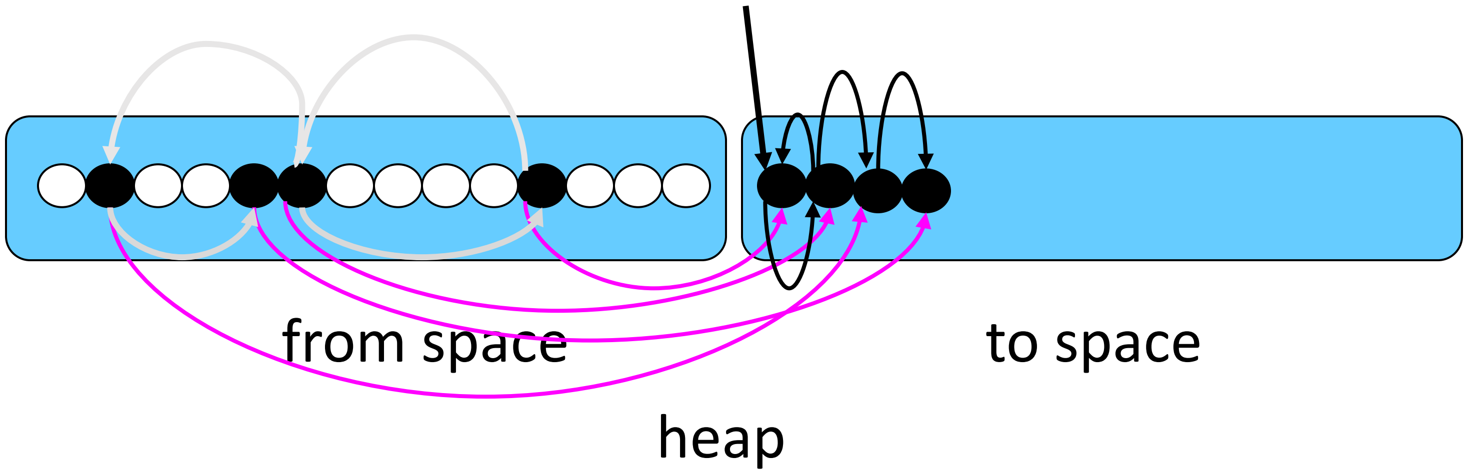
Semispace

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes



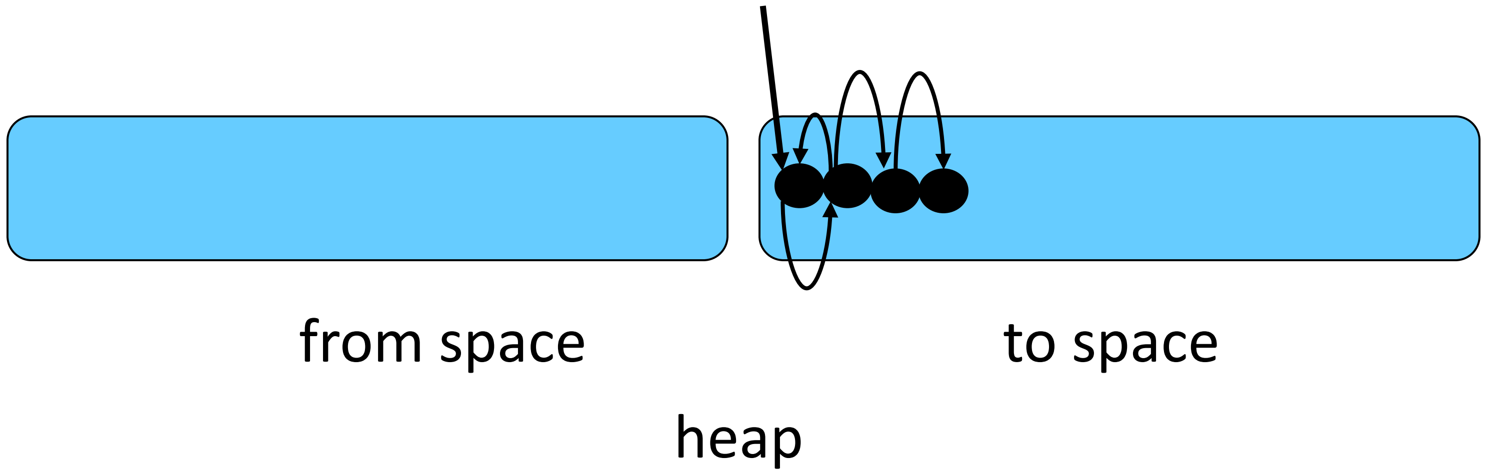
Semispace

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes
  - reclaims “from space” en masse



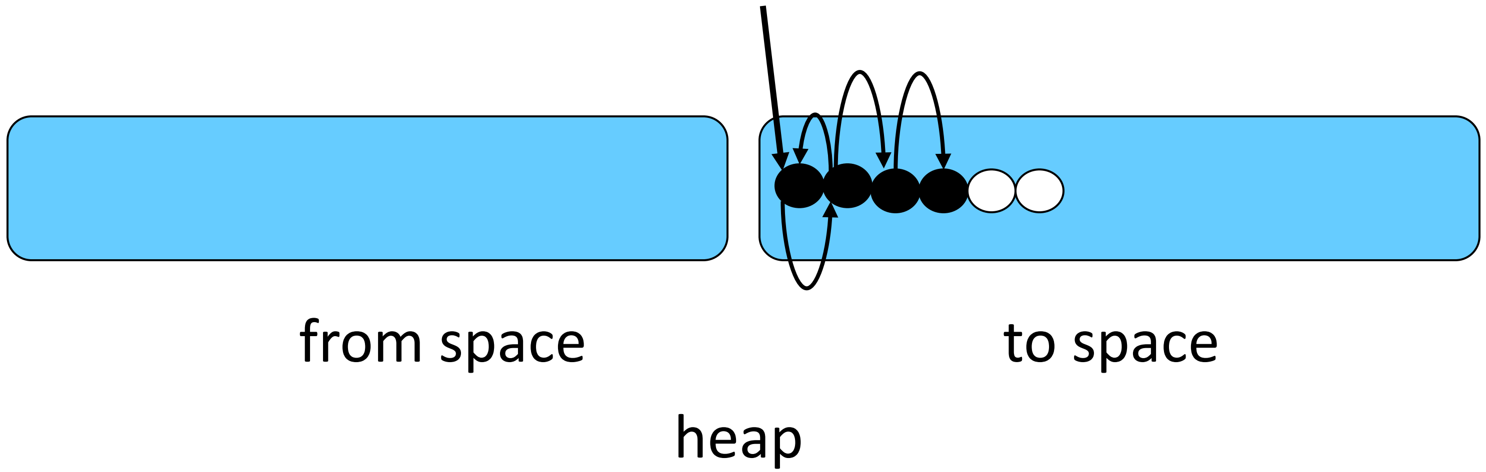
Semispace

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes
  - reclaims "from space" en masse
  - start allocating again into "to space"



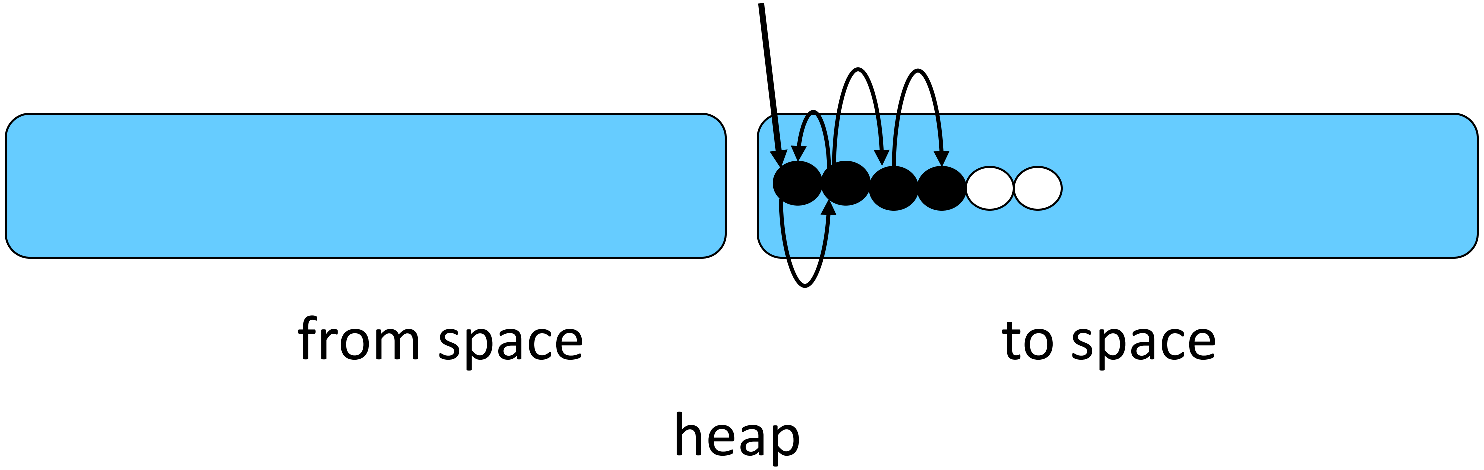
Semispace

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes
  - reclaims “from space” en masse
  - start allocating again into “to space”



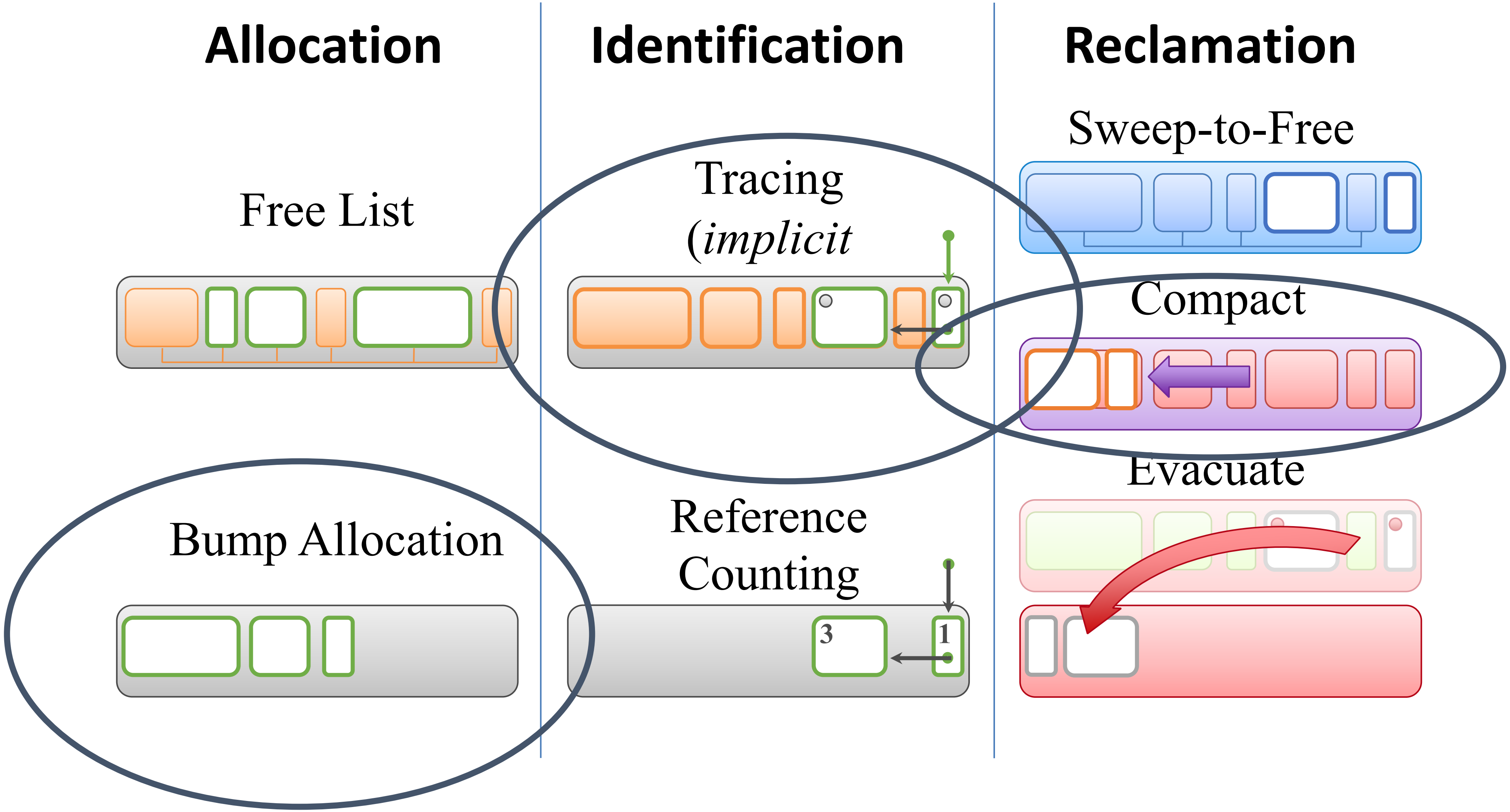
Semispace

- Notice:
  - fast allocation
  - locality of contemporaneously allocated objects
  - locality of objects connected by pointers
  - wasted space



The Big Picture

- Heap organization; basic algorithmic components



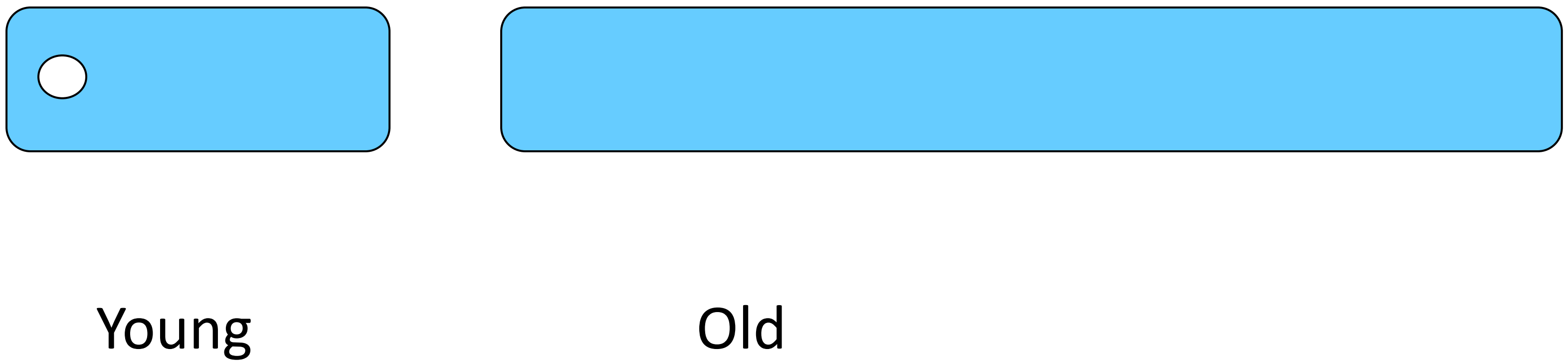
Generational Collection

What objects should we put where?

- **Generational hypothesis**
  - **young objects die more quickly than older ones [Lieberman & Hewitt’83, Ungar’84]**
  - most pointers are from younger to older objects [Appel’89, Zorn’90]
- Organize the heap in to young and old, collect young objects preferentially

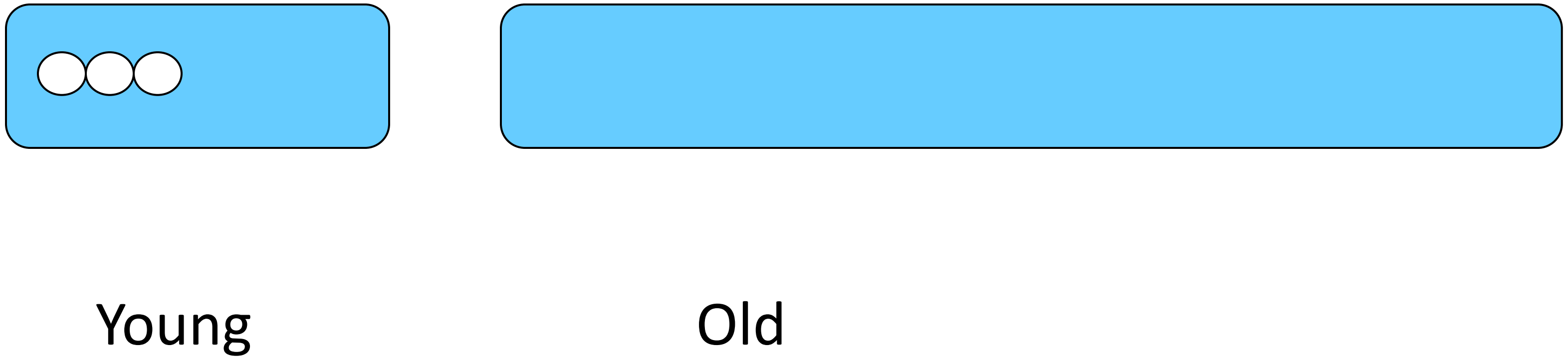
Generational Heap Organization

- **Divide the heap in to two spaces: young and old**
- Allocate in to the young space
- When the young space fills up,
  – collect it, copying into the old space
- When the old space fills up
  – collect both spaces
  – Generalizing to m generations
    - if space n < m fills up, collect n through 1



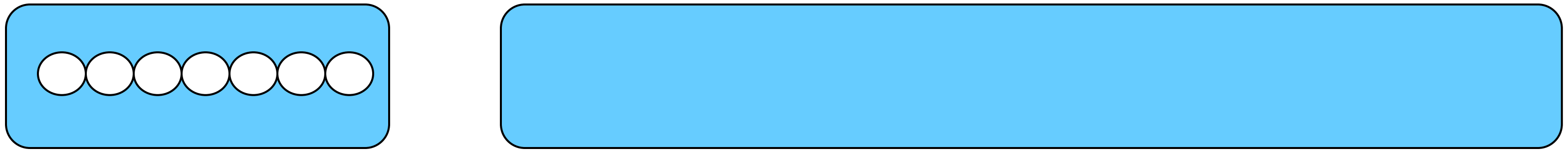
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- **Allocate in to the young space**
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



Generational Heap Organization

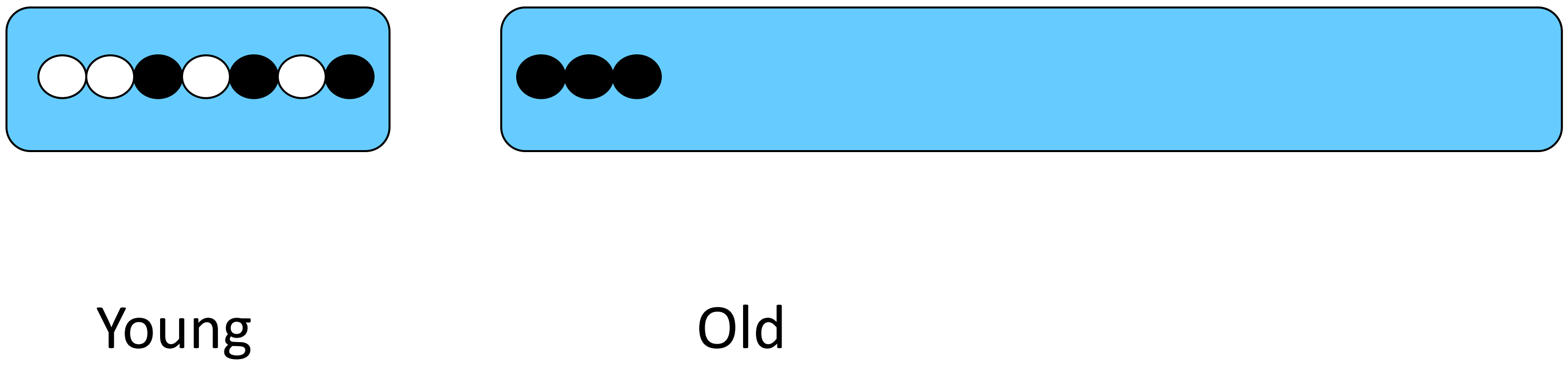
- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



Young Old

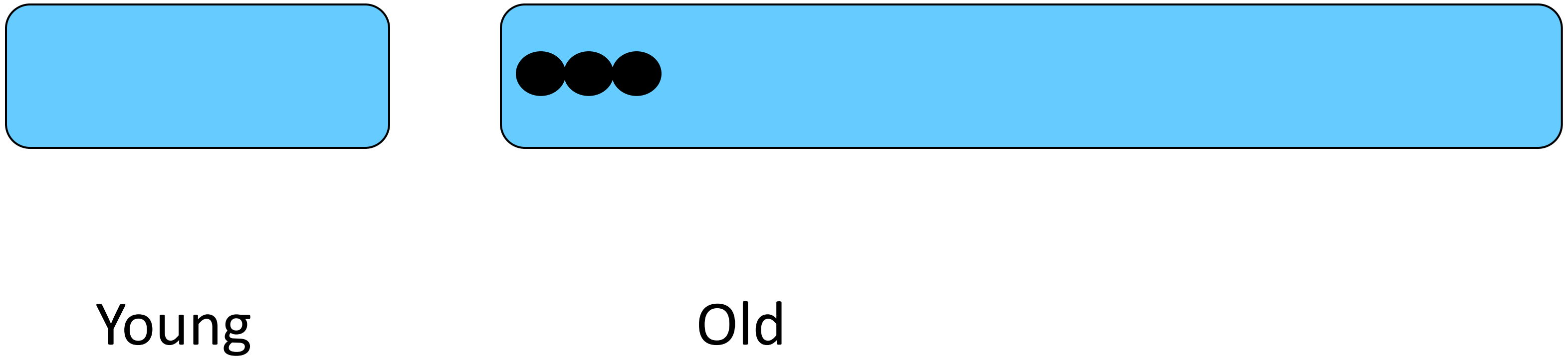
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



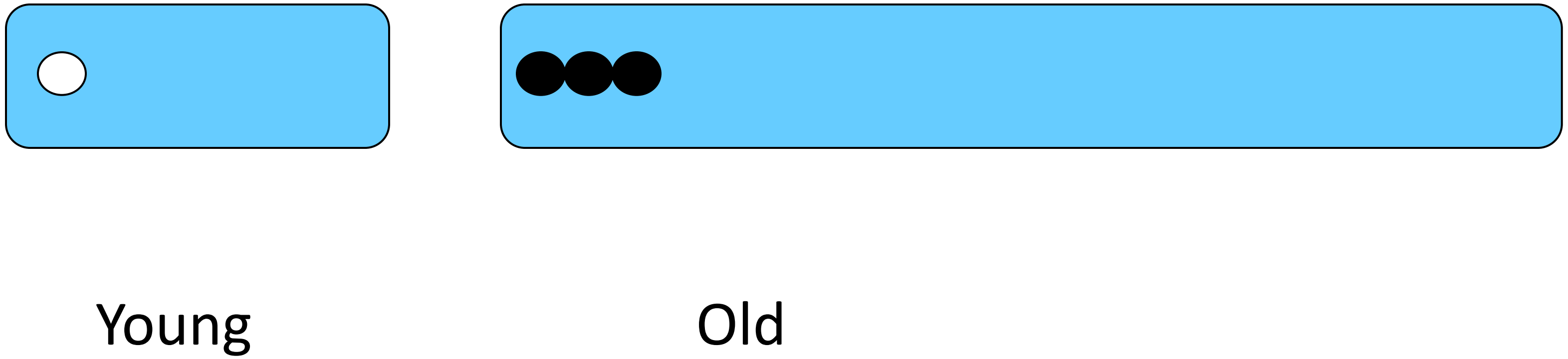
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



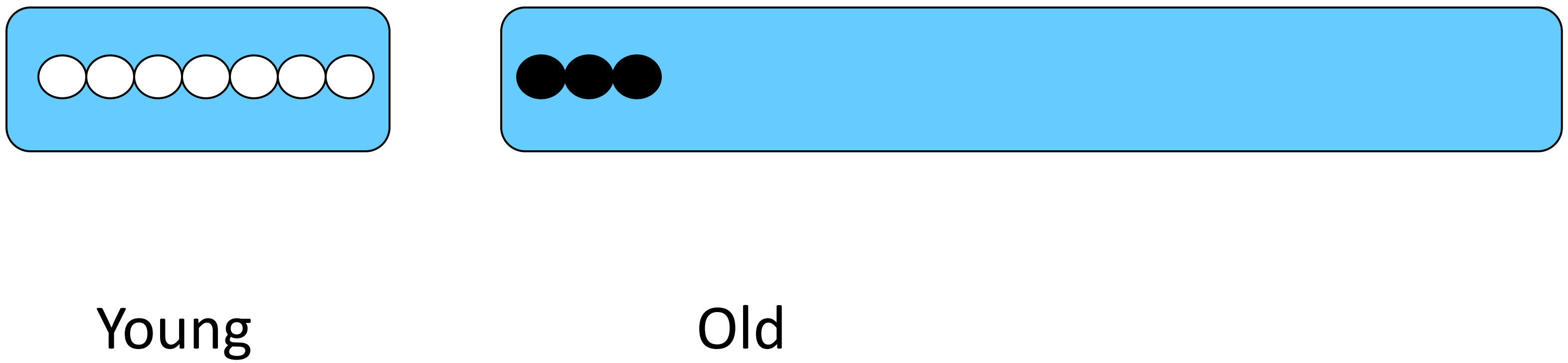
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- **Allocate in to the young space**
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



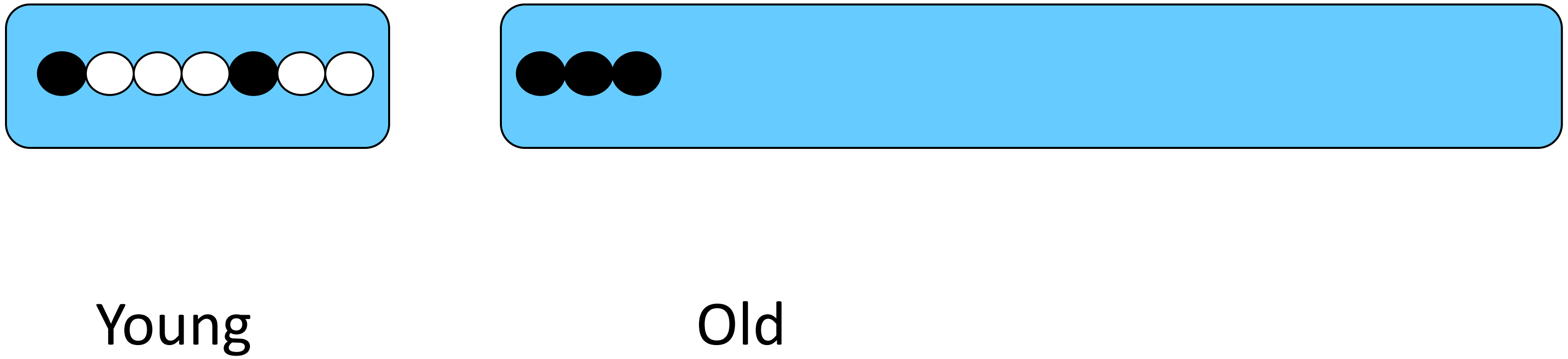
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



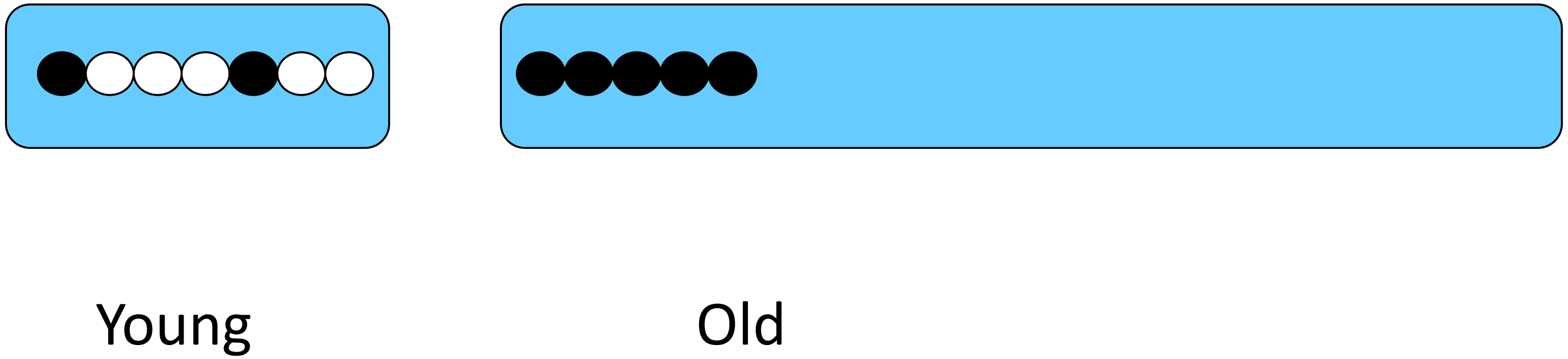
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



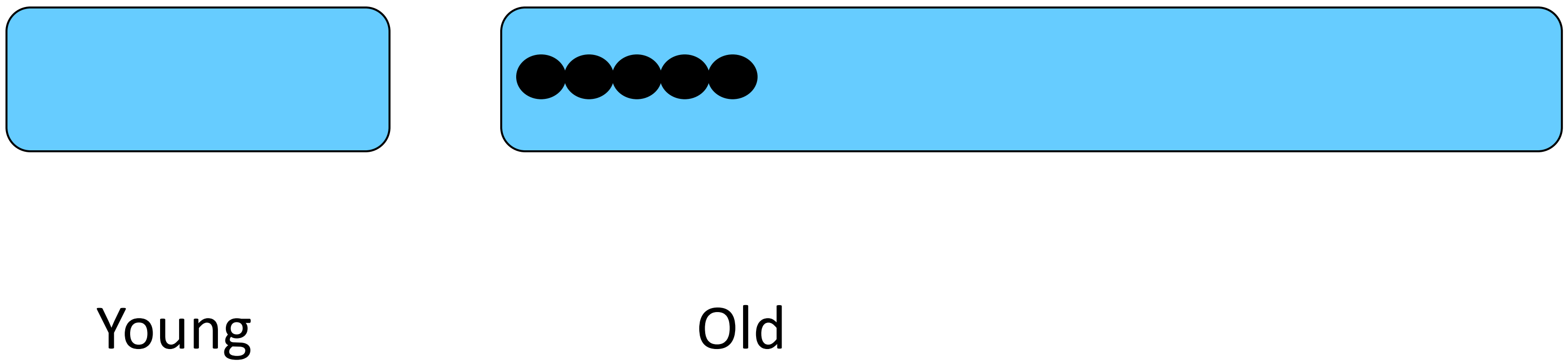
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



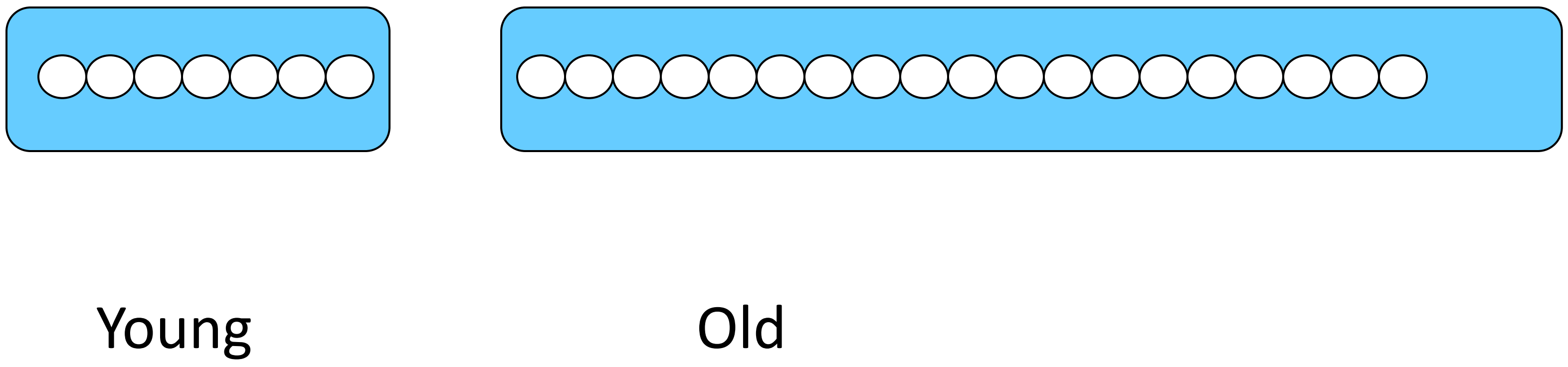
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



Generational Heap Organization

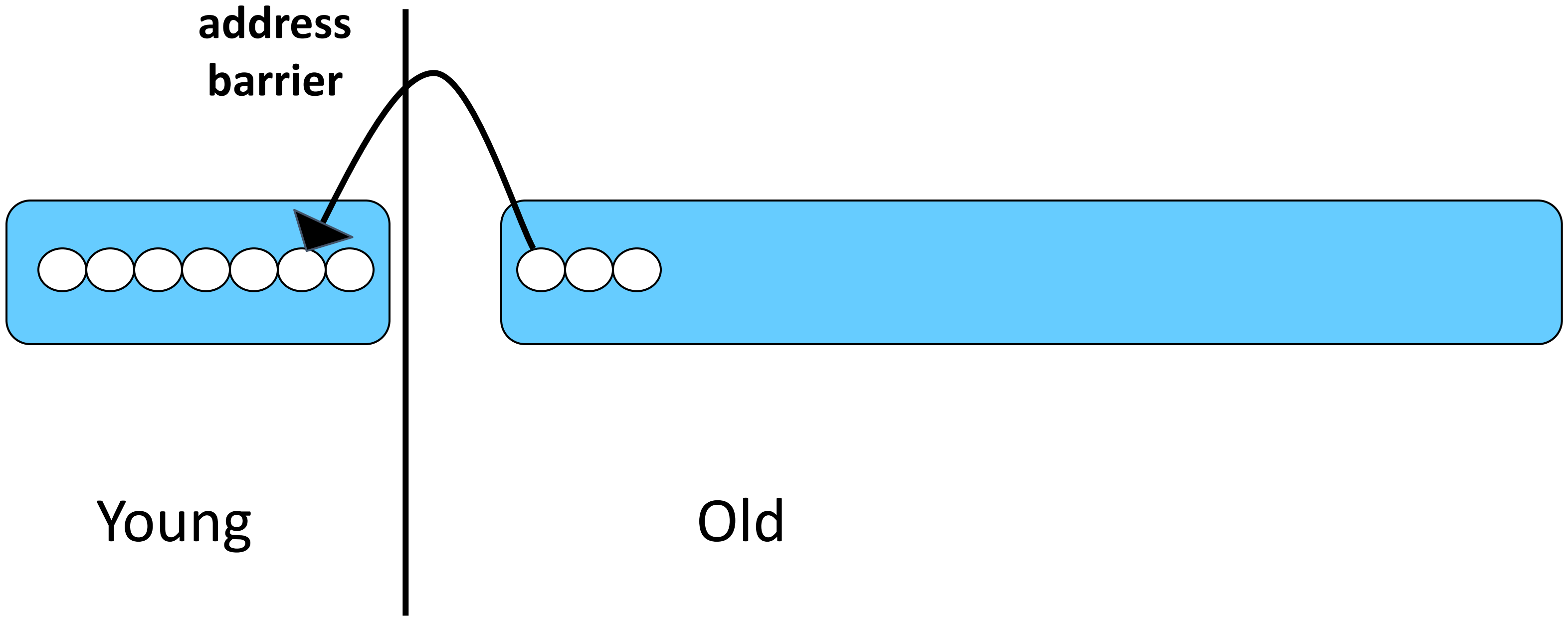
- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- **When the old space fills up**
  - **collect both spaces**
  - Generalizing to m generations
    - if space n < m fills up, collect n through 1



Generational Write Barrier

Unidirectional barrier

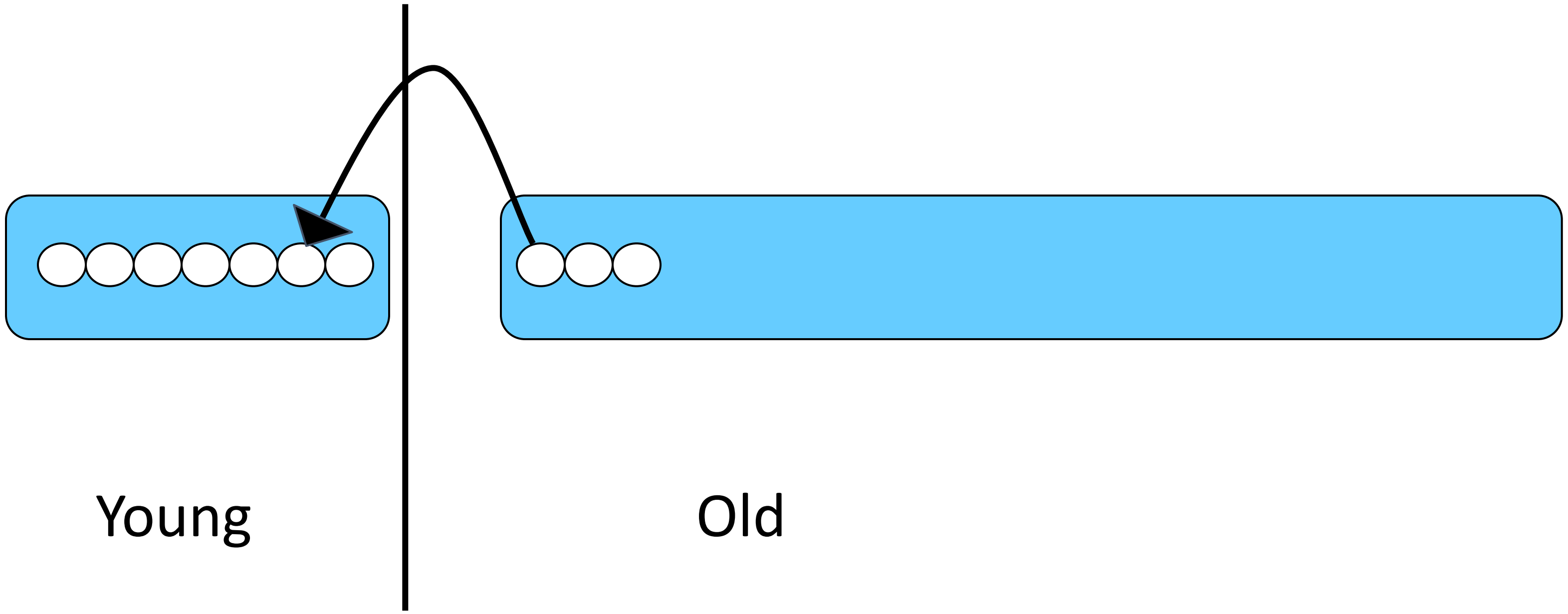
- **record only older to younger pointers**
- **no need to record younger to older pointers, since we never collect the old space independently**
- most pointers are from younger to older objects [Appel'89, Zorn'90]
- track the **barrier** between young objects and old spaces



Generational Write Barrier

unidirectional boundary barrier

```
// original program    // compiler support for incremental collection
p.f = o;               if (p > barrier && o < barrier) {
                           remset_nursery = remset_nursery U &p.f;
                       }
                       p.f = o;
```



Generational Write Barrier

Unidirectional

- **record only older to younger pointers**
- **no need to record younger to older pointers, since we never collect the old space independently**
- most pointers are from younger to older objects [Appel'89, Zorn'90]
- most mutations are to young objects [Stefanovic et al.'99]