Object-Oriented Languages

Chapter 9

Classes

- A **class** is a blueprint for creating objects

```
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, they are **instances** of classes, created through **new**
  – E.g., when 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”

References in Java/Pointers in C++

- Objects are manipulated indirectly through object references (pointers)

```
main(…) { // Java code
  Rectangle x;
  x = new Rectangle(); // 1) Create a Rectangle object in memory
                       // 2) Produce a reference value which is
                       //      a handle to this object
                       // 3) Assign this reference value to x
  x.width = 3.14;      // 1) Use the r-value of x to get to the object
}                      // 2) Assign based on the l-value of field width
```

  - **x** is a variable of type “reference to Rectangle objects”
- **C++: Rectangle* x; x = new Rectangle();**
  - **x** is a variable of type “pointer to Rectangle objects”

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 instance is produced
    - This is the result of evaluating the **new** expression
  - The appropriate **constructor** of the class is called to initialize the new object
  - **x** is assigned this reference (pointer) value
    - e.g. the value may be the **address** of the first byte of the object's memory
    - or the value may be some **internal handle** to the actual object (e.g., index in some internal table, which itself contains the address of the first byte)

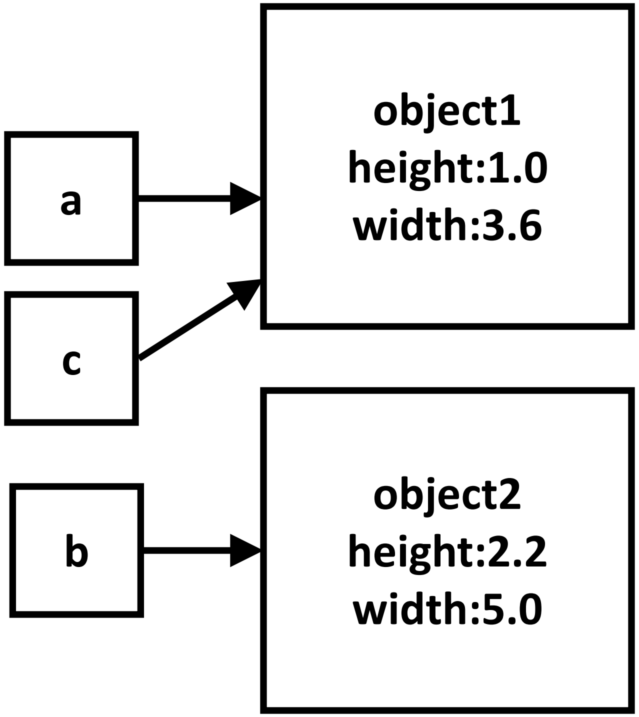
Destruction of Objects

- 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, GC may figure out it is dead**
- C++ destructors: called when the programmer manually destroys the object with **delete**
  - **class Rectangle { … ~Rectangle() {…} // destructor }**
- Java finalizers: called when the object is collected
  - **class Rectangle { … void finalize() {…} // finalizer }**

Members: Fields and Methods

- Two separate kinds: **instance** members and **static** members
  - Instance members: each instance of the class has **a separate copy of this member**

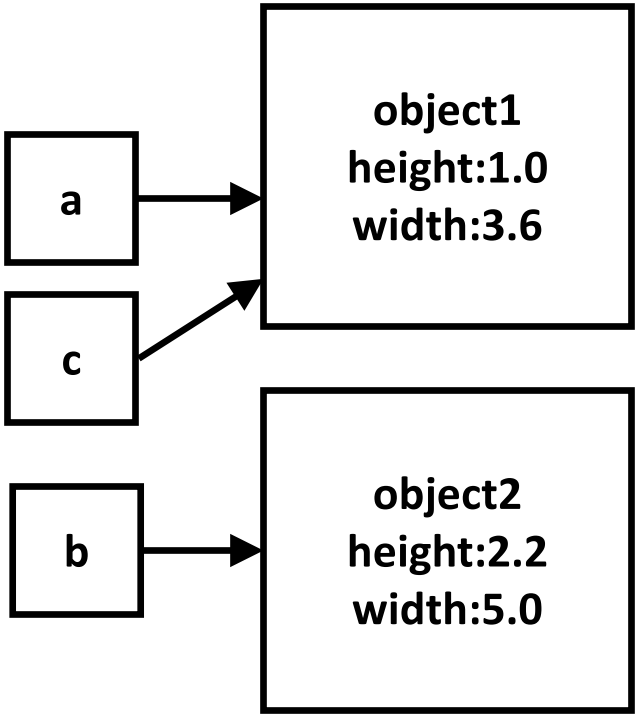
```
Rectangle a, b, c;
a = new Rectangle();
b = new Rectangle();
a.height = 1.0; a.width = 3.6;
b.height = 2.2; b.width = 5.0;
c = a;
```



Members: Fields and Methods (C++)

- C++: **x->f** is shorthand (syntactic sugar) for **(*x).f**
  - Expression **x** evaluates to pointer value that points to the object; expression ***x** evaluates to the actual object; ***x->f** evaluates to the field **f** of that object (**f** is **not** static – why?)

```
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;
```



Instance Methods

- An instance method operates on objects
  - Method **m** is invoked on the object

double area() { return height*width;}

in reality, this is syntactic sugar for

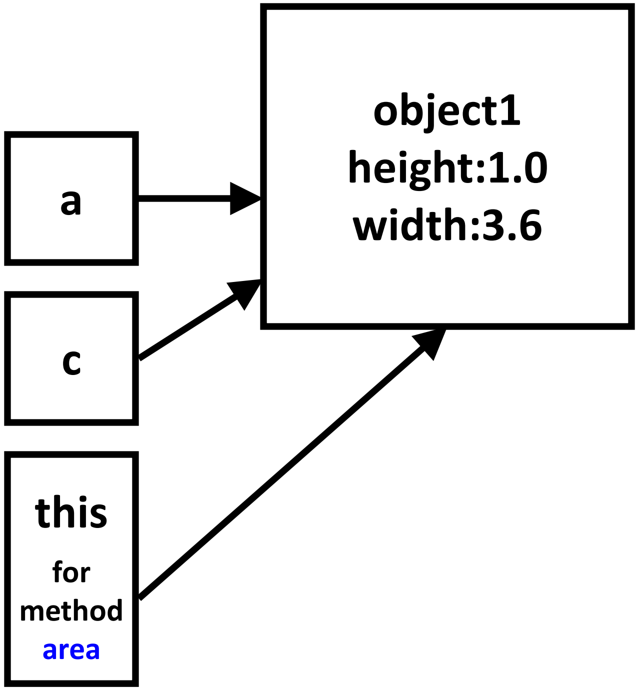
```
double area(Rectangle this) { // Java
        return this.height * this.width; }
double area(Rectangle* this) { // C++
        return this->height * this->width; }
```

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

Methods Calls

- Calling an instance method: there is an object on which we are calling it
  - **x.m()** in Java, **x->m()** in C++

```
Rectangle  *a, *b, *c;
a = new Rectangle();
a->height = 1.0;
a->width = 3.6;
c = a;
double result = c->area();
```



Method Overloading (Java)

- A class may have more than one method with the same name
  - the name is “overloaded”
- All overloaded methods must have different signatures
  - Signature: method name + types of formal parameters
- **double area() { … }** → signature **area()**
- **double area(int precision)** { … } → signature **area(int)**

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 values 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 to the object is assigned to x

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 **m** 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 Rectangle(double h, double w) { … }
  public double getHeight() { return height; }
  public double getWidth() { return width; }
  public double area() { … } }

class SwissRectangle extends Rectangle {
  private int hole_size;
  public SwissRectangle(double h, double w, int hs) { … }
  public void shrinkHole() { hole_size--; }
  public double area() { … } // overridden }
```

Constructors and Inheritance

- Constructors are not inherited
- A constructor in a subclass **B** must invoke a constructor in the superclass **A**
  - (this is a bit of an oversimplification)
- The constructor of superclass **A** initializes 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);  hole_size = hs; }
```

Inheritance of Methods

- If a subclass declares a method with the same name but **a different signature**, we have **overloading**
  - 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 points 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 instances of **A**
- Correct view: pointer to instances of **A** or instances of any subclass of **A**
  - If **C** is a subclass of **B**, variable **x** can also point to instances of **C**
  - Poly (many) morph (form) ism

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 (classname.class)
  - **virtualinvoke x,<A: void m(int,int)>**

Method Invocation – Run Time

- The Java virtual machine loads the bytecode and starts executing it
- When it tries to execute instruction **virtualinvoke x,<A: void m(int,int)>**
  - Looks at **the class Z of the object referenced by x**
  - Searches **Z** for a method with signature **m(int,int)** and return type **void**
  - If **Z** does not have it, goes to **Z**'s superclass, and so on upwards, until a match is found

Method Invocation – Run Time

- The **run-time (dynamic) target**: "lowest" method that matches the signature and the return type 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**
- Terms: **virtual dispatch**, **method lookup**

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

Terminology

- Invocation **x.m(a,b)** "sends a message **m**" to the object referenced by **x**
  - This object is the receiver object
  - The method that contains call **x.m(a,b)** belongs to the sender object
- Dynamic binding of the message/call (virtual dispatch): mapping the message (i.e., the call) to a method
- Polymorphic call: more than one possible run-time target

Abstract Classes and Methods

- Abstract class: instances of it cannot be created
  - Only instances of its subclasses
- Abstract methods
  - No code: just name, parameter types, and return type
  - Abstract methods must be **overridden** in subclasses, by concrete methods
    - “concrete” = “non-abstract”

Abstract Classes

- Abstract class: class that contains abstract methods
  - **abstract void m(int x);** // Java
  - **virtual void m(int x) = 0;** // C++
- We cannot say **new X()** if X is abstract. Why?
- An abstract method can be the **compile-time target** of a method call
  - But not the run-time target, obviously
- Sometimes non-abstract classes are referred to as "concrete classes"

Interfaces in Java

- Very similar to abstract classes in which <u>all</u> methods are abstract
- A Java class has only one superclass, but can implement many interfaces
  - **class Y extends X implements A, B { … }**
- 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

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

Static Methods and Fields

- **Static field**: a single copy for the entire class
- **Static method**: **not** invoked on an object
  - Just like a regular procedure (function) in a procedural language (e.g.. C, Pascal, etc.)
- Terminology
  - static method/field = **class** method/field
  - instance method/field = **non-static** method/field

Classic Example (Java)

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

in main:

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

returns 2

Classic 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(…)

Objects in C++: Pointers vs. Values

```
main() { …
    Rectangle* x; // Pointer variable on the call stack
    x = new Rectangle(2.3,7.8); // New object on the heap
    Rectangle y(4.5,0.1); // Object variable on the call stack
    // y's constructor called when execution reaches the declaration
    double z = f(x,y);
    // y's destructor called at the end of the method
}
double f(Rectangle* a, Rectangle b) {
    // a: Pointer variable on the call stack
    // b: Object variable on the call stack
    // Parameter passing: the copy constructor of b is called
    // Equivalent to a call b.Rectangle(y)
    return a->width + b.height;
}
```

- A default copy constructor provided by the compiler: copies field-by-field
- The programmer may choose to implement her own copy constructor
  - **Rectangle(Rectangle& other) { … }**

Implementation Techniques for Java

- The compiler takes as input source code
  - Oracle/Sun provides a standard compiler; others 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 Java Virtual Machine

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 provides several kinds of JVMs for various platforms (e.g., Solaris, Wintel, etc.)
  - Several other vendors for 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" (e.g., frequently executed methods), and after a certain threshold is reached, point compile them

Lifetimes and Memory Management

- **Static allocation**: address determined once and retained throughout the execution of the program
  - E.g., **static** fields in C++, Java
- **Stack-based allocation**: local variables of methods, plus the formal parameters (incl. **this**)
- **Heap-based allocation**: space allocated and deallocated manually by the programmer
  - C: **A* a = (A*)malloc(sizeof(A)); … free (a);**
  - C++: **A* a = new A(); … delete a;**
  - Java: **A a = new A();** but deallocation is done automatically, through **garbage collection**

Garbage Collection

- Slides based on course materials by Prof. Kathryn McKinley, UT Austin and Microsoft Research
- Explicit (manual) memory management
  - More code to maintain
  - Correctness
    - Free an object too soon - crash
    - Free an object too late - waste space
    - Never free - at best waste, at worst fail
  - Efficiency can be very high
  - Gives programmers more control over the run-time behavior of the program

Garbage Collection

- Automatic management through garbage collection
  - Reduces programmer burden: less user code compared to manual memory management
  - Eliminates sources of errors
    - Less user code to get correct
    - Protects against some classes of memory errors: no free(), thus no premature free(), no double free(), or forgetting to free()
  - Not perfect, memory can still leak
    - Programmers still need to eliminate all pointers to objects the program no longer needs
  - Integral to modern object-oriented languages
    - Java, C#, PHP, JavaScript
  - Mainstream
  - Challenge: performance

Key Issues

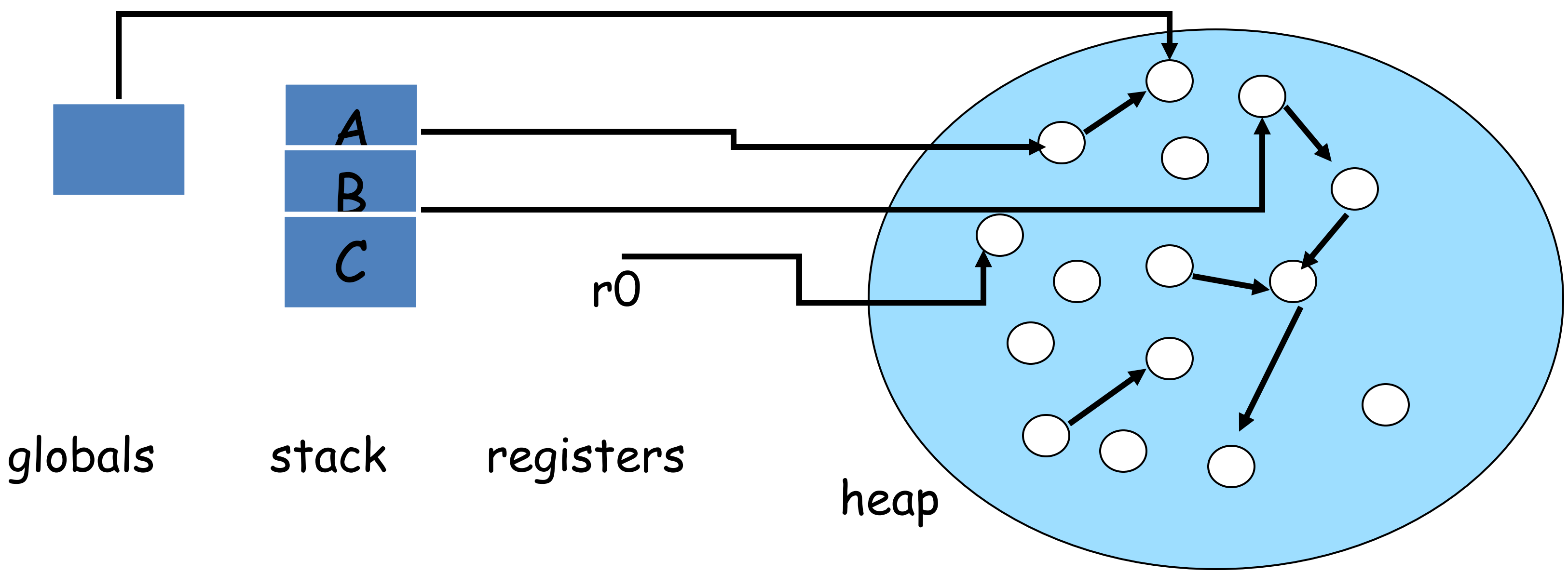
- For both mechanisms
  - Fast allocation
  - Fast reclamation
  - Low fragmentation (wasted space)
  - How to organize the memory space
- Garbage collection
  - Discriminating live objects and garbage
    - **Live** object will be used in the future
    - Prove that object is **not live (i.e., dead)**, and deallocate it
    - Deallocate as soon as possible after last use

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

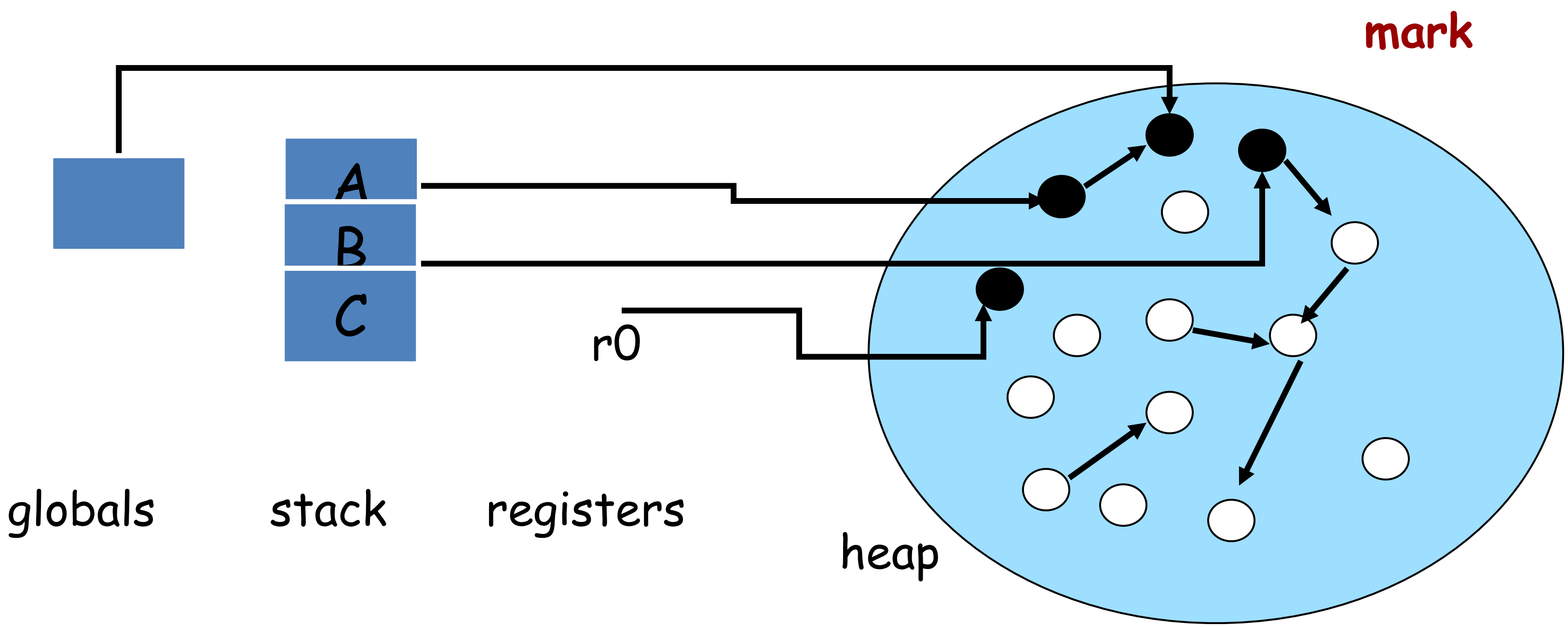
Reachability

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



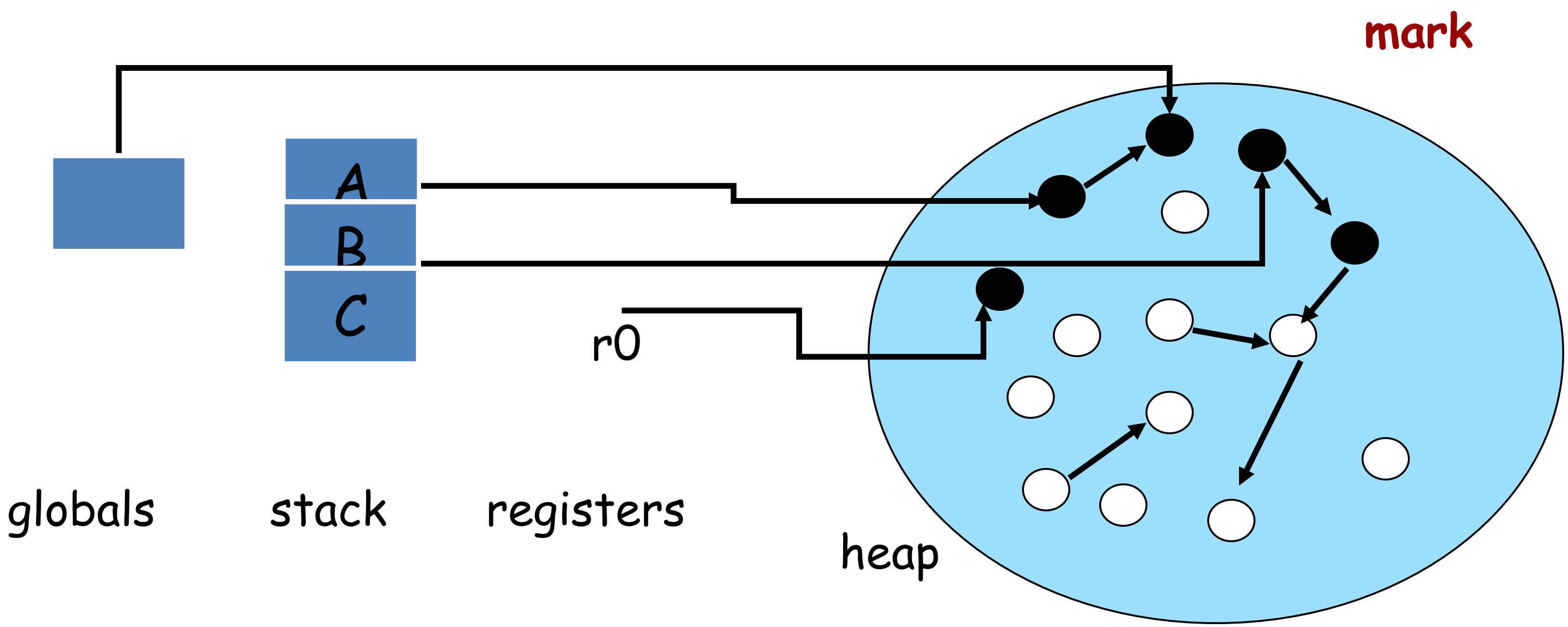
Reachability

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



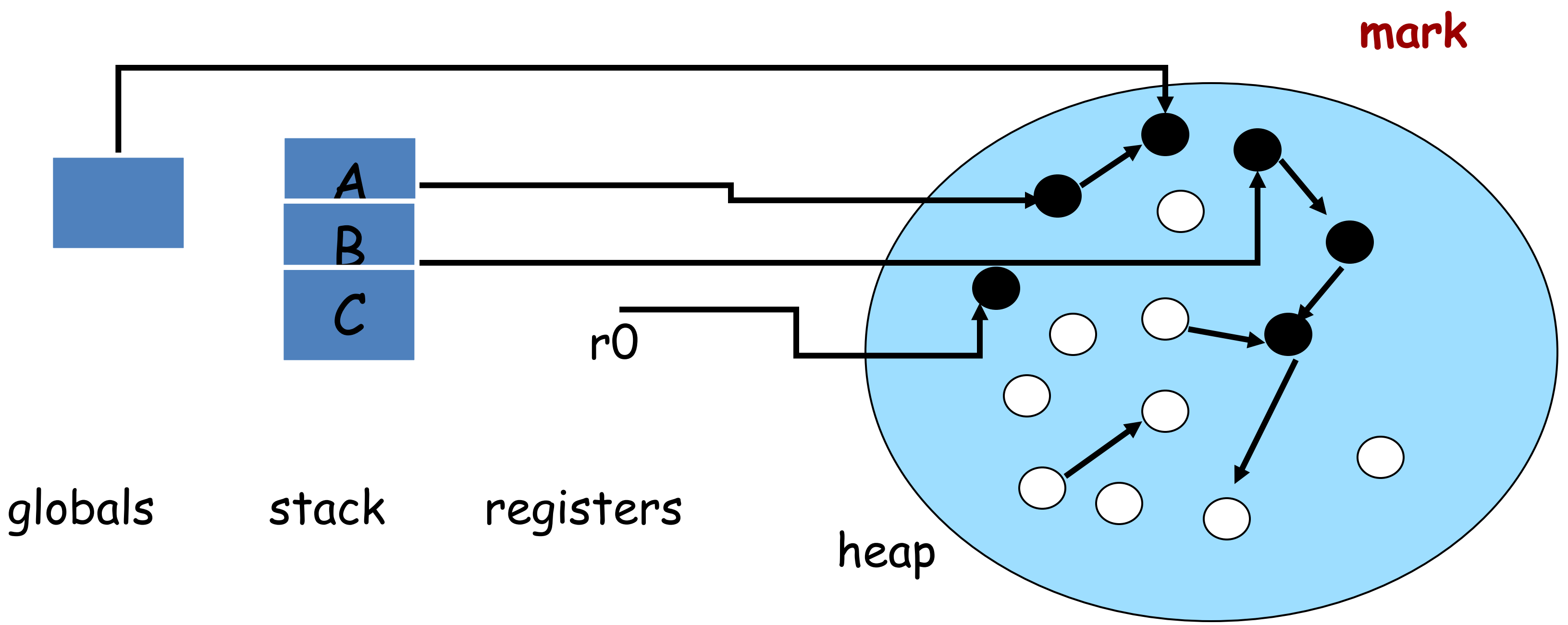
Reachability

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



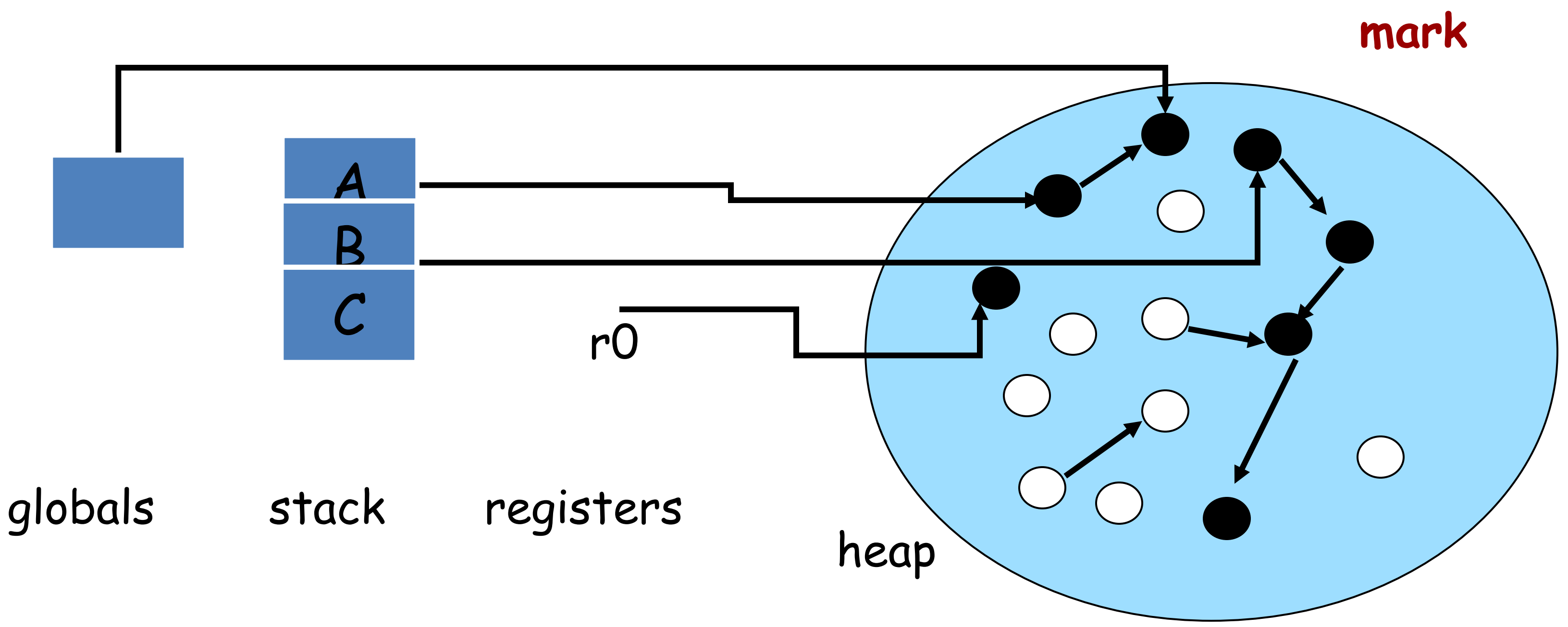
Reachability

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



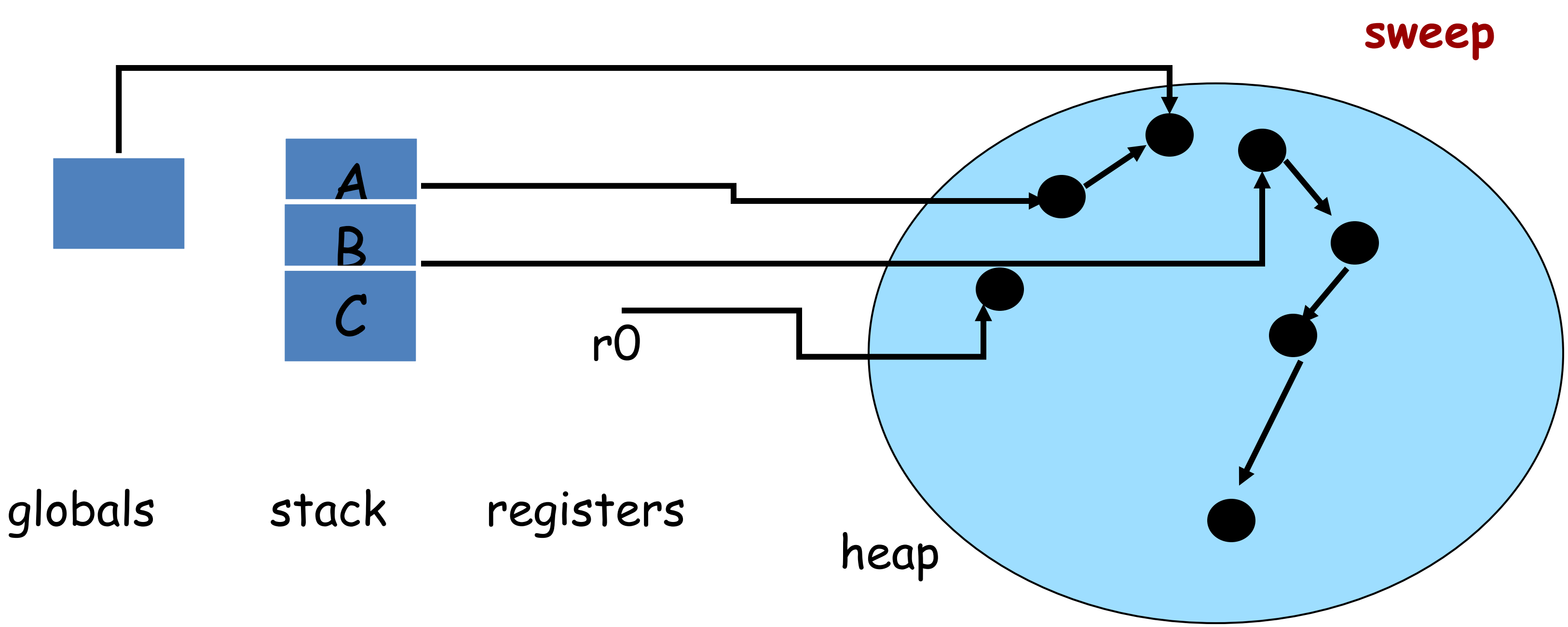
Reachability

- 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**



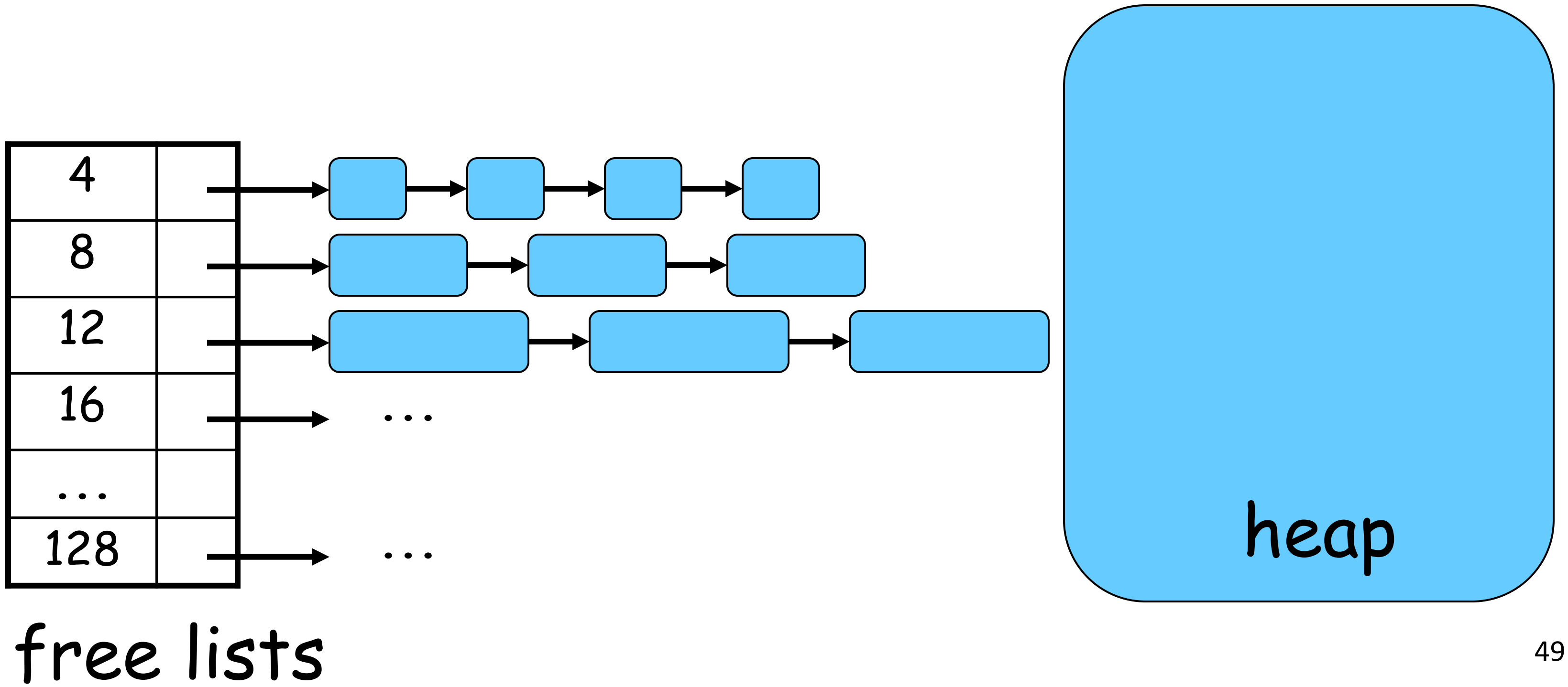
Reachability

- 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**



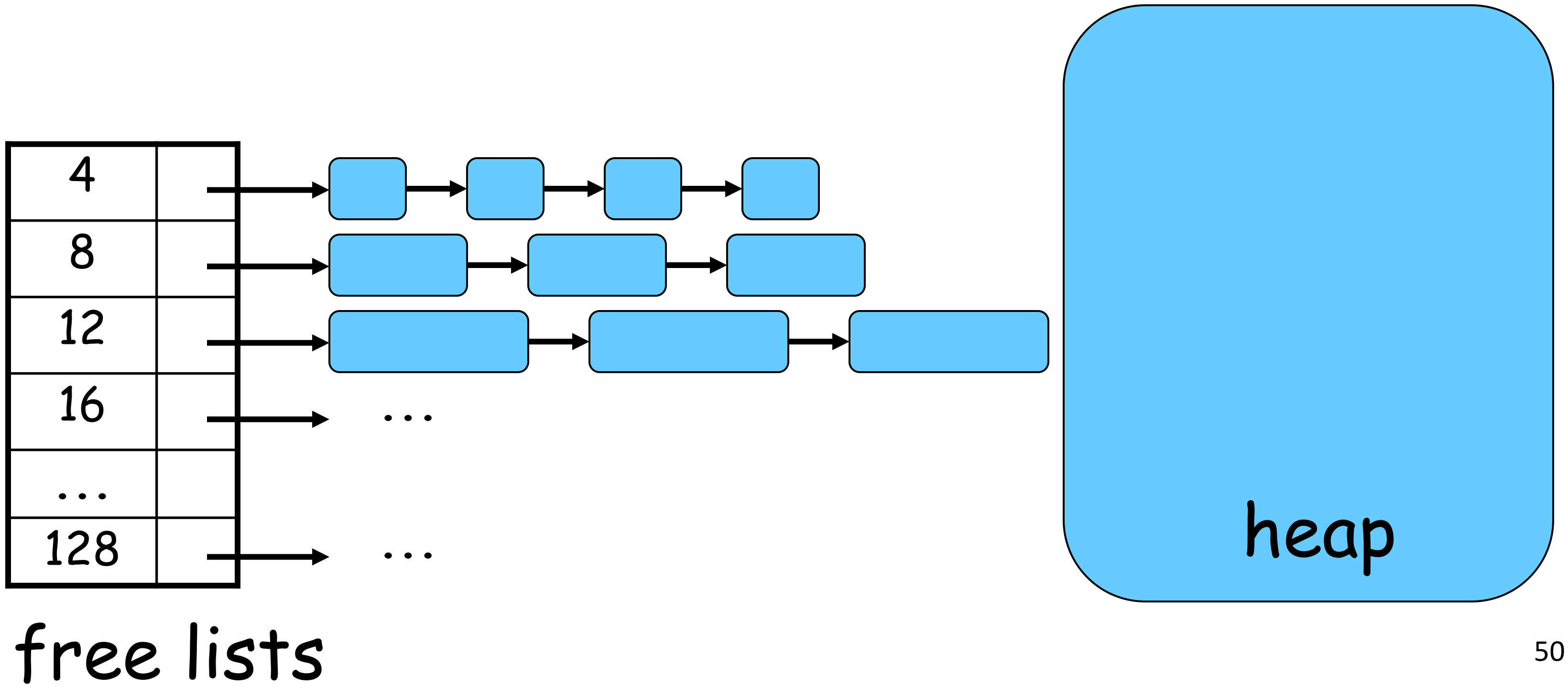
Mark-and-Sweep Implementation

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



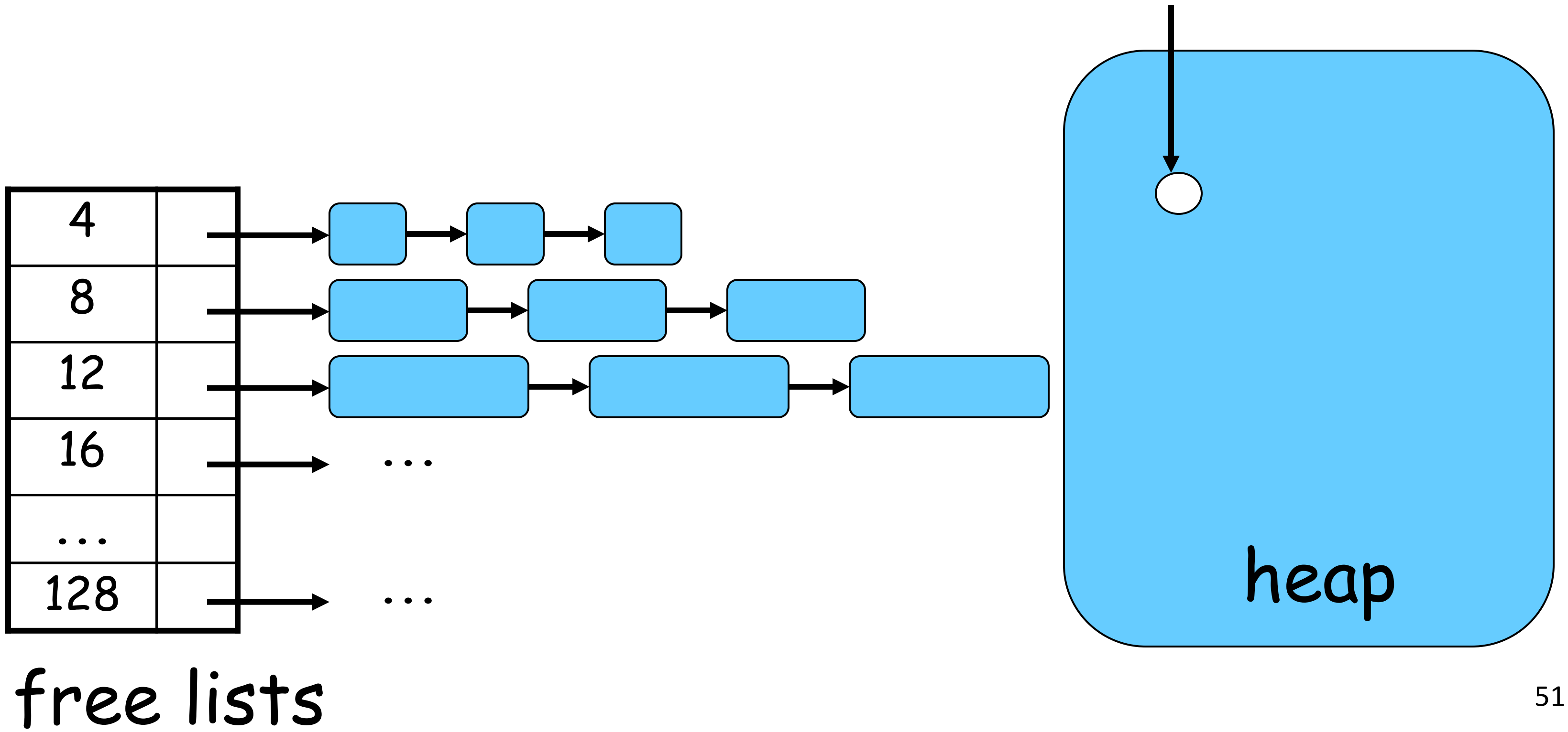
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list



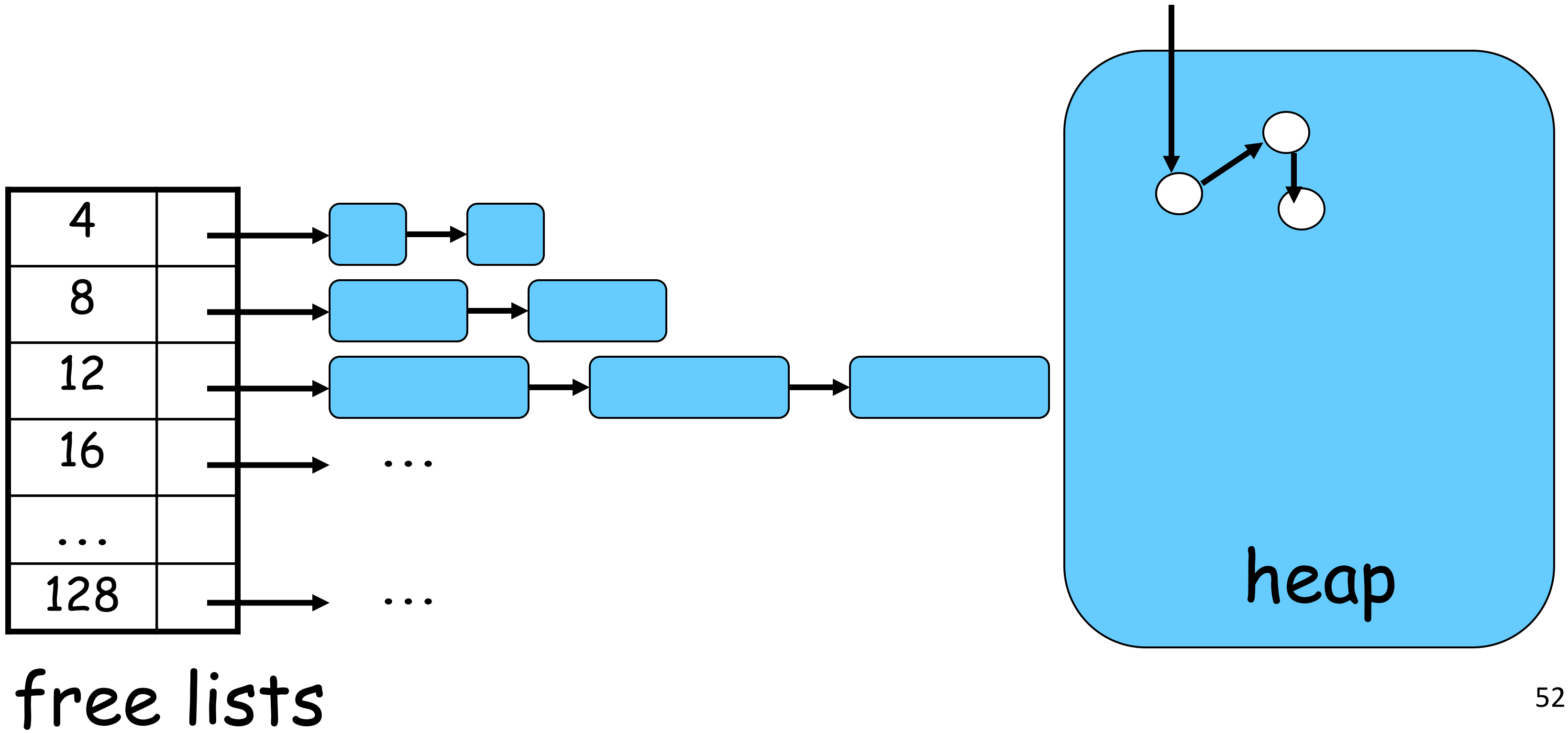
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list



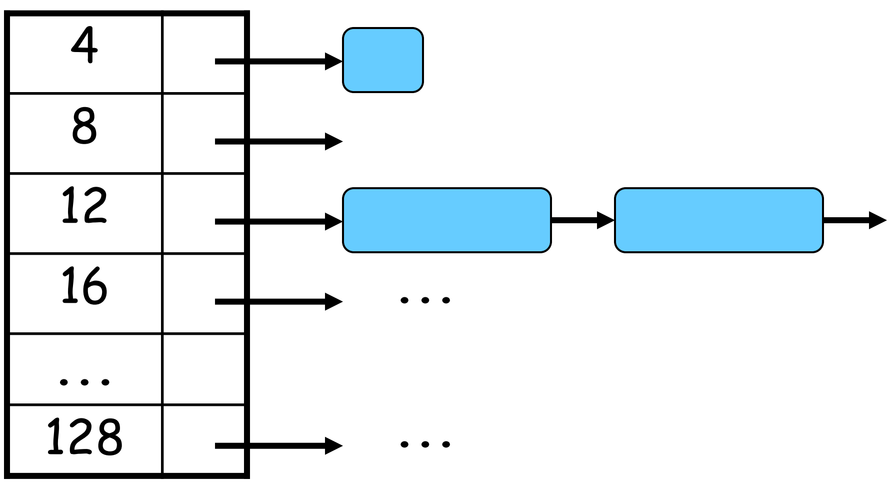
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list

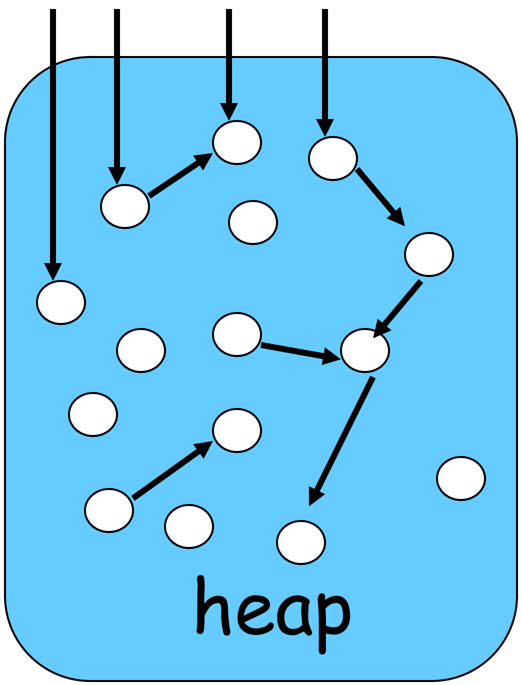


Mark-and-Sweep Implementation

- 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

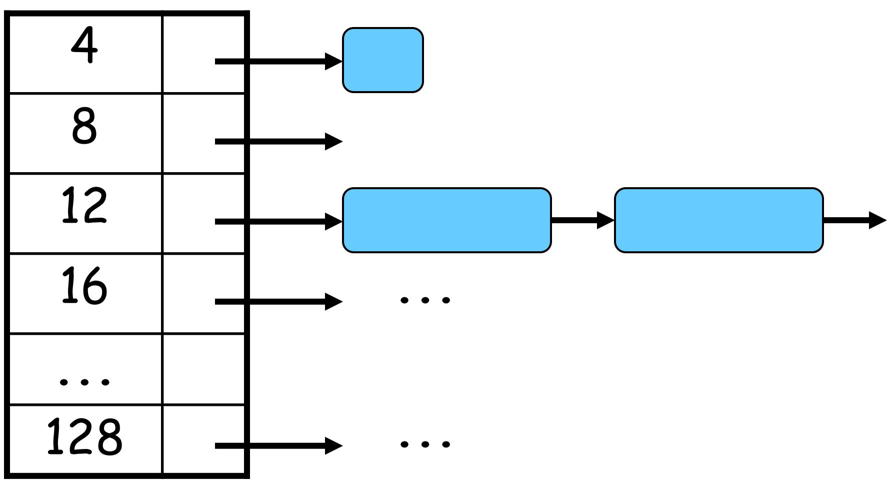


free lists

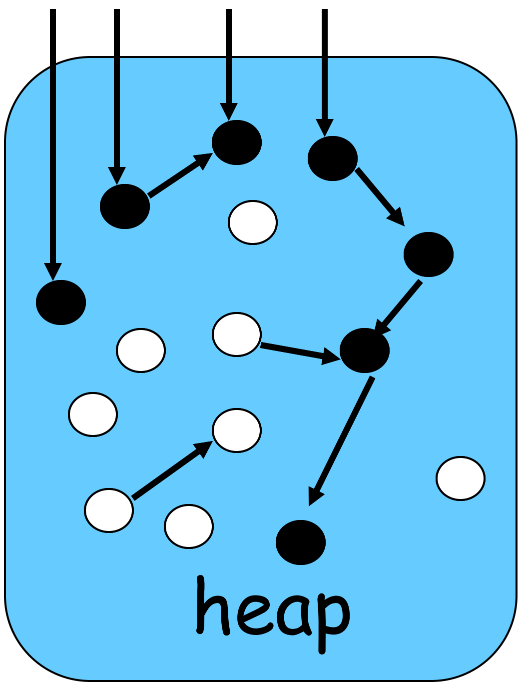


Mark-and-Sweep Implementation

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

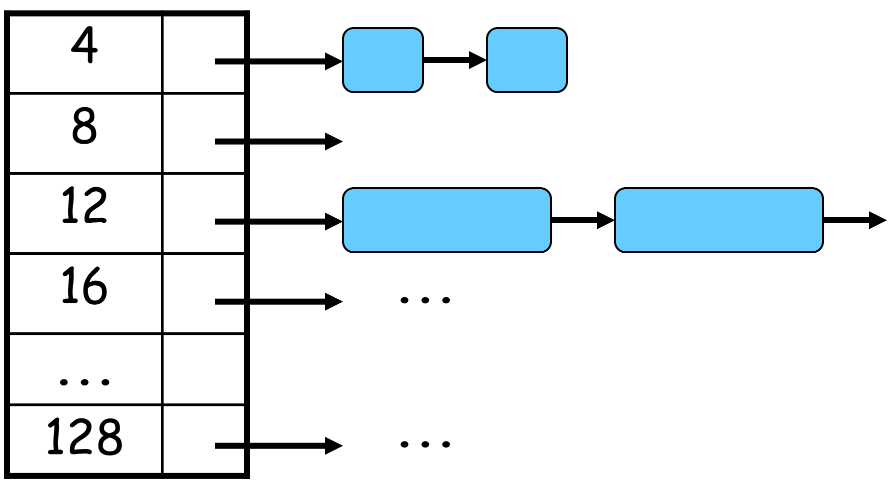


free lists

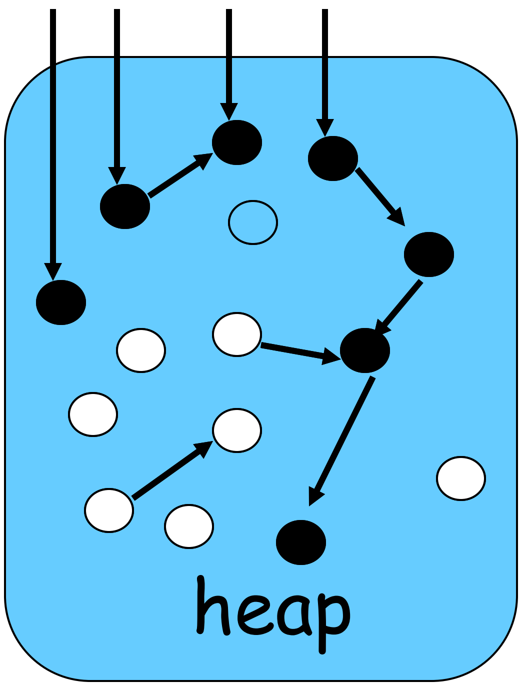


Mark-and-Sweep Implementation

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

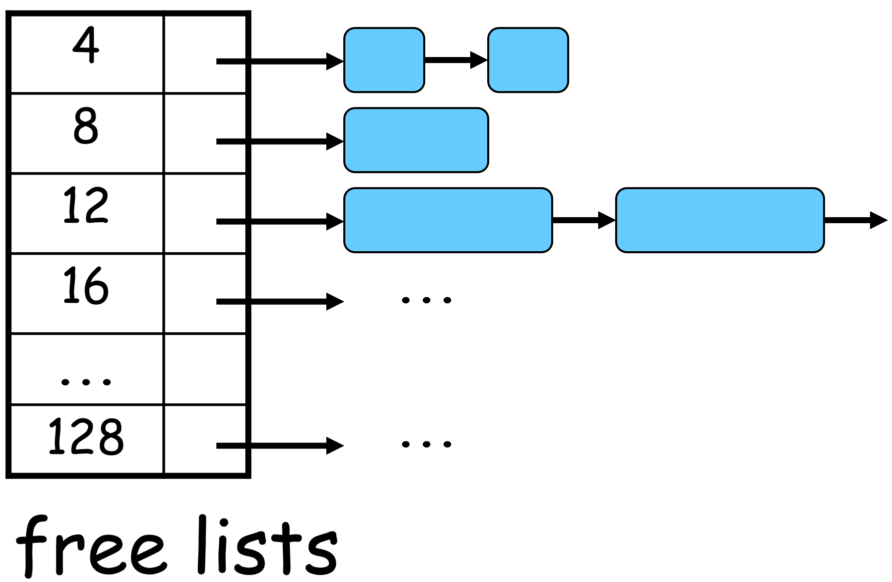


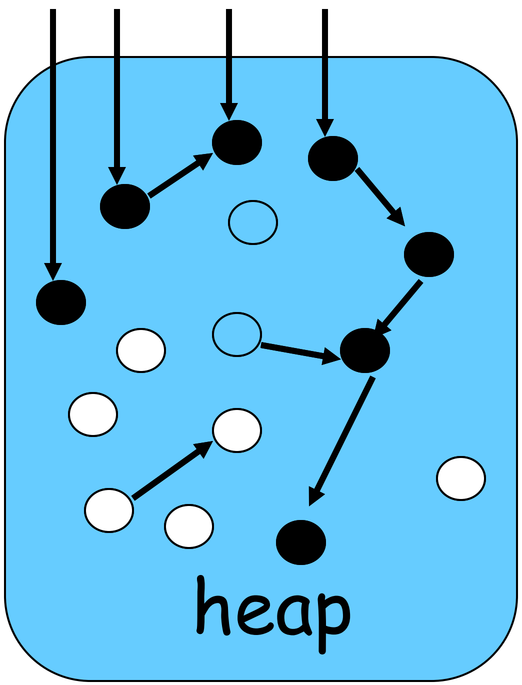
free lists



Mark-and-Sweep Implementation

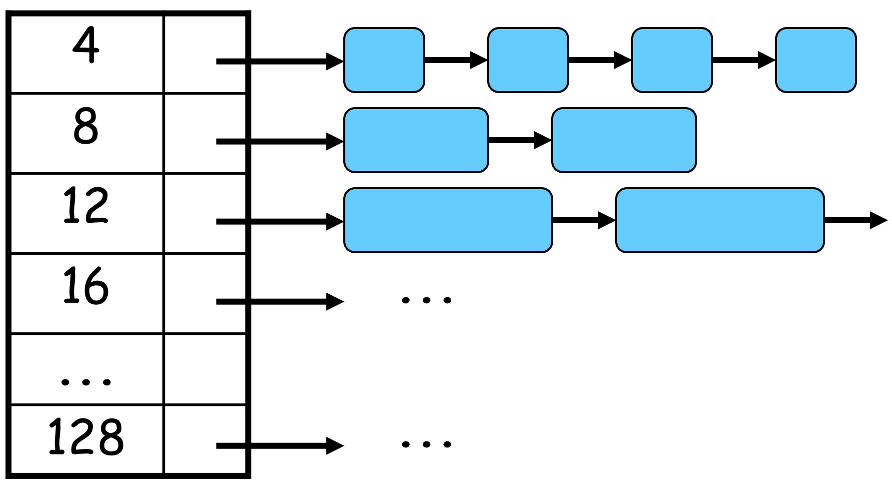
- Mark phase
  - Reachability computation on the heap, marking all live objects
- Sweep phase
  - Sweep the memory for free objects, and populate the free lists



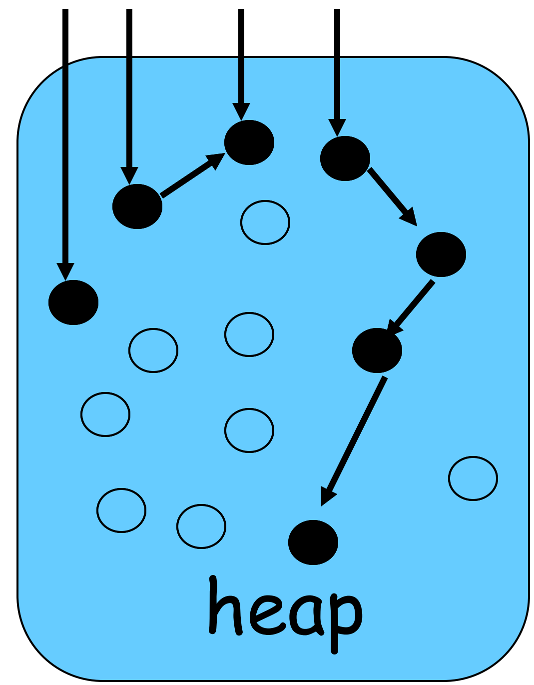


Mark-and-Sweep Implementation

- 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



The Big Picture

- Heap organization; basic algorithmic components

Allocation **Identification** **Reclamation**