Introduction to Memory Management & Garbage Collection

or How to Live with Memory Allocation Problems

OOPSLA 2000 Tutorial no. 70
Tuesday 17 October 2000

Richard Jones
Computing Laboratory
University of Kent at Canterbury

Eric Jul
DIKU Department of Computer Science
University of Copenhagen

What is the problem?

Memory is a scarce and precious resource

Some applications can manage with a bounded amount of memory using static allocation combined with stack allocation.

Others use dynamic allocation of memory because:

- Some objects live longer than the method that creates them.
- Recursive data structures such as lists and trees.
- Avoids fixed hard limits on data structure sizes.

If we had unbounded amounts of memory, we'd never worry.

The *PROBLEM* is that we don't have *unbounded* memory.

What can we do?

Dynamically allocate and deallocate memory.

***REUSE* deallocated memory.**

Dynamical memory allocation is available in many languages, e.g., using languages features:

- **New** allocates a new object
- **Delete** *X* deallocated the object *X*

Such features allows programmer to handle allocation themselves.

Objects that no longer are needed are called *garbage.*

Garbage Collection

Identifying garbage and deallocating the memory it occupies is called *garbage collection.*

We can try to handle the garbage collection housekeeping chores related to object allocation and deallocation ourselves.

Such housekeeping can be simple but for many applications the chores become complex – and error prone.

Can we do it ourselves? – Or should it be automatic??

Why AUTOMATIC Garbage Collection?

Because human programmers just can't get it right.

Either

too little is collected leading to **memory leaks**, or

too much is collected leading to **broken programs**.

Space leaks?

Human programmers can:

- Forget to delete an object when it is no longer needed.
- Return a newly allocated object – but when will it be deallocated?
- Not figure out when a shared objects should be deleted.

Sharing is a significant problem

Can be handled by using the principle *last one to leave the room turns off the light*.

However, this is easily forgotten, and, worse, in a large building, it can be close to impossible to detect that you are the last!

Dangling Pointers

Eager human programmers can delete objects too early leading to *dangling pointers*

Consider an object that is shared between two different parts of a program each having its own pointer to the object.

If one of the pointers is deleted then the other pointer is left pointing to a *non-existent object* – we say that it is a *dangling pointer.*

(My wife, who is effective at throwing things out, introduced me to the concept of dangling pointers quite early in our marriage.)

Sharing is a Real Problem

Sharing is a significant problem

Memory leaks and dangling pointers are two sides of the same coin:

The difficulty is managing objects in the presences of sharing.

Example

Consider the principle: *last one to leave the room turns off the light*.

However, this is easily forgotten, and, worse, in a large building, it can be close to impossible to detect that you are the last to leave!

Part 2: Object Allocation

In the following, we review how objects are allocated

- Object & Machine Model?
- Explicit Allocation
- Dangers of Explicit Allocation

Allocation

Allocation means finding a free piece of memory in the heap and reserving it for the representation of an object.

Deallocation means changing the status of a piece of memory from allocated to free.

Liveness An object is live as long as it still is reachable from some part of the program's computation.

Objects in Memory

Objects are typically allocated in the heap.

An *Object Reference* is a pointer to an object (typically merely the heap address of the start of the object).

Variables contain object references (ignoring primitive data).

Each object can contain a number of variables and thereby reference other objects.

Static & dynamic allocation

Static allocation — allocation takes place when a program starts – basically memory is laid out by the compiler

Dynamic allocation — allocate new objects while the program is executing.

A simple form of dynamic allocation is *stack allocation* where objects are allocated on the program stack and deallocated using a stack discipline.

Heap allocation is the most general form of allocation: objects are allocated in the heap.

Dangers of Explicit Deallocation

With explicit deallocation the programmer ends up:

Doing too little

- Garbage objects are not deallocated and slowly but surely clutters memory and so the program runs out of memory (such a failure to delete garbage objects is called a *memory leak).*

Doing too much

- Throwing away a non-garbage object. Subsequent use of a live reference to the object will cause the program to fail in inexplicable ways. Such a reference is a *dangling reference*.
- Throwing away a garbage object *twice!* Likely to break the memory manager.

The *REAL* bad thing about explicit deallocation

The problems of:

- Memory leaks
- Dangling references
- Double deallocation

are real and omnipresent in explicit deallocation systems and they cause the real problem:

Wasting huge amounts of debugging time!

and despite this, programs may still fail in mysterious ways long after being put into production.

Finding and fixing MM bugs can account for 40% of debug time.

Part 4: Automatic Memory Management

Automatic memory management including garbage collection handles the most significant of the problems that we tried to solve until now.

Doing it yourself has is cumbersome to do – and quite error prone.

In the following, we present automatic memory management

What *is* garbage?

Almost all garbage collectors assume the following definition of live objects called **liveness by reachability**: if you can get to an object, then it is live.

More formally: An object is **live** if and only if:

it is referenced in a predefined variable called a **root**,

or

it is referenced in a variable contained in a live object

(i.e. it is transitively referenced from a root).

Non-live objects are called *dead* objects, i.e. **garbage**.

Graphs & Roots

The objects and references can be considered a *directed graph*:
The live objects of the graph are those reachable from a **root**.
The process executing a computation is called the **mutator** because it is viewed as dynamically changing the object graph.

What are the roots of a computation?

Determining roots is, in general, language-dependent

In common language implementations roots include

- words in the **static area**
- **registers**
- words on the execution **stack** that point into the heap.

Why garbage collect?

Language requirement

- many OO languages assume GC, e.g. allocated objects may survive much longer than the method that created them

Problem requirement

- the nature of the problem may make it very hard/impossible to determine when something is garbage

Composing components

Modules should be reusable in different contexts

- Cohesive
- Loosely-coupled
- Communicate with as few other modules as possible and exchange as little information as possible [Meyer]
- Interfaces should be simple and well-defined

But…

Liveness is a global property

- An object is live if it can be used by *any* part of the program
- This cannot (in general) be determined by inspection of a single code fragment

Adding MM book-keeping clutter to interfaces

- Weakens abstractions
- Reduces extensibility of modules

Liveness

An object is only live if it can effect future computation

- Must be able to load it (or a part of it) into registers.
- Well-behaved programs that do not access random addresses in memory.

What data is known (and can be manipulated)?

- Global data held in static areas
- Local variables, parameters and compiler temporaries that may be held on the stack or in machine registers

Hence the program may also use

- Any objects that can be reached by way computations on known objects.

Liveness by reachability

Almost all garbage collectors assume the following definition of live objects called **liveness by reachability**:

if you can get to an object, then it is live.

More formally: An object is **live** if and only if:

it is referenced in a predefined variable called a **root**,

or

it is referenced in a variable contained in a live object

(i.e. it is transitively referenced from a root).

Non-live objects are called *dead* objects, i.e. **garbage**.

A conservative estimate

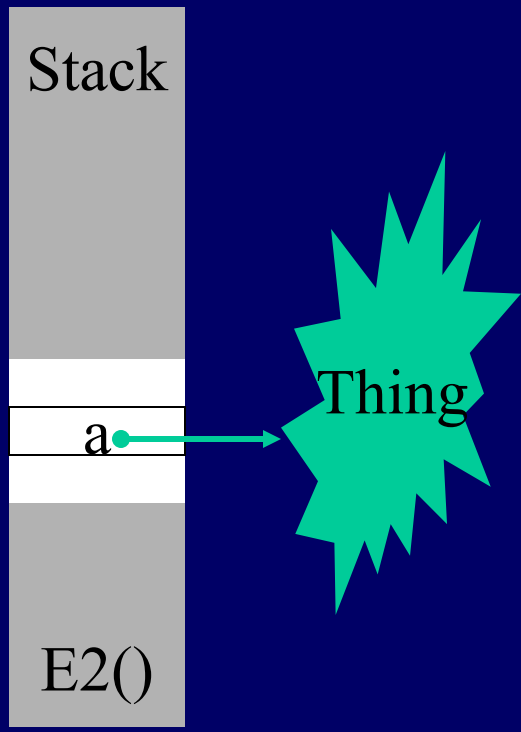
'Liveness by reachability' provides a *conservative* estimate of the set of live objects.

- Contains all objects that could be used by a well-behaved program
- May contain objects that will never be used again.

```
Thing a = someComputation();
if(a.property())
    E1();
else
    E2();
```

Reference to a may be held on stack — hence considered reachable — until E1/E2 has completed.

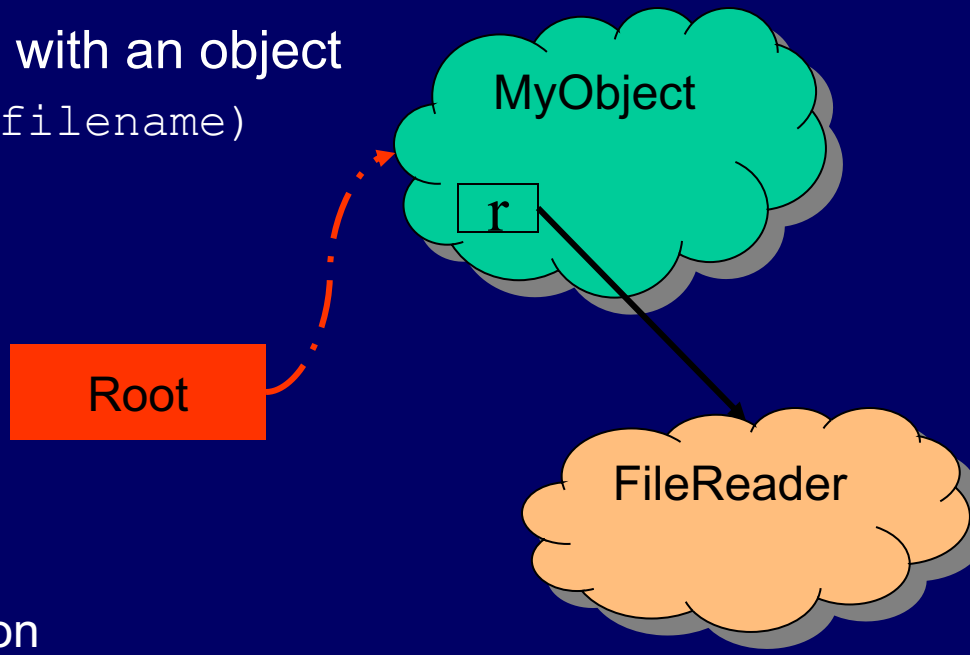
But static analysis may reveal that a could be discarded after the conditional test.



Help the collector

Help the GC that you are finished with an object

```
r = new FileReader(filename)
// use the reader
…
reader.close();
reader = null;
```



- This is a simple, *local*, decision
- *Don't null* the reference if it is about to disappear (e.g. local variable in a method that's about to return),
- *Do dispose* of components when you have finished with them if your framework (e.g. AWT) that requires you to.
  e.g. `myWindow.dispose();`

Cost Metrics for GC

Execution time

- total execution time
- distribution of GC execution time
- time to allocate a new object

Memory usage

- additional memory overhead
- fragmentation
- virtual memory and cache performance

Delay time

- length of disruptive pauses
- zombie times

Other important metrics

- comprehensiveness
- *implementation* ***simplicity*** *and* ***robustness***

No silver bullet

Often not necessary for simple programs

- But beware reuse of simple code

Hard real-time code

GC doesn't cure problem of data structures that grow without limit

- Surprisingly common e.g. caching
- Benign in small problems, bad for large or long running ones
- Java's References model

Abstraction may hide concrete representations

- E.g. stack as an array
  [Problem is that this assumes concept of tracing GC. Put it later?]

The basic algorithms

- **Reference counting:** Keep a note on each object in your garage, indicating the number of live references to the object. If an object's reference count goes to zero, throw the object out (it's dead).

- **Mark-Sweep:** Put a note on objects you need (roots). Then recursively put a note on anything needed by a live object. Afterwards, check all objects and throw out objects without notes.

- **Mark-Compact:** Put notes on objects you need (as above). Move anything with a note on it to the back of the garage.
  Burn everything at the front of the garage (it's all dead).

- **Copying:** Move objects you need to a new garage. Then recursively move anything needed by an object in the new garage. Afterwards, burn down the old garage (any objects in it are dead)!

Reference counting

A mechanism to share ownership

Goal

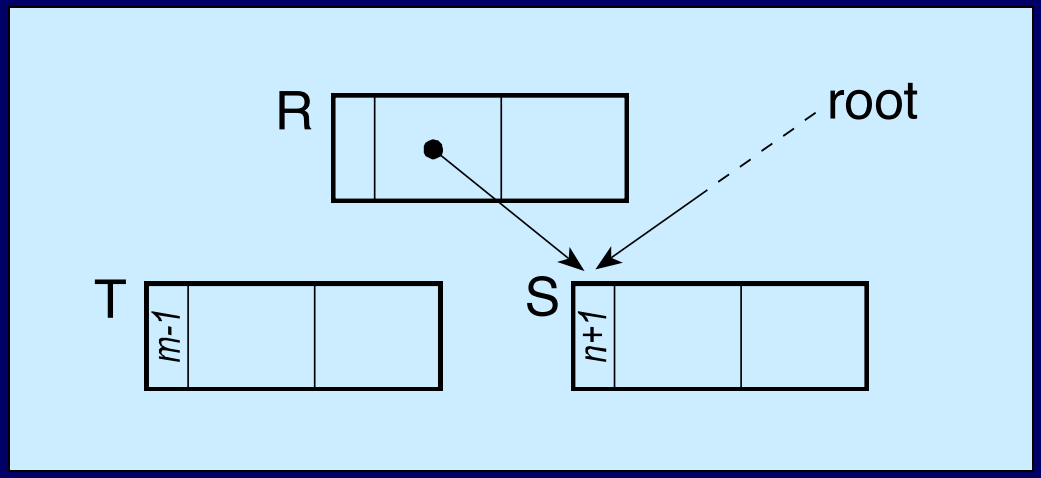
- identify when you are the *only* owner
- You can make the disposal decision.

Basic idea: count the number of references from live objects.

Reference counting: principle

Each object has a **reference count (RC)**

- when a reference is copied, the referent's RC is incremented
- when a reference is deleted, the referent's RC is decremented
- an object can be reclaimed when its RC = 0



`Update(left(R), S)`

Advantages of reference counting

- ✓ Simple to implement
- ✓ Costs distributed throughout program
- ✓ Good **locality of reference**: only touch old and new targets' RCs
- ✓ Works well because few objects are shared and many are short-lived
- ✓ **Zombie time** minimized: the zombie time is the time from when an object becomes garbage until it is collected
- ✓ Immediate **finalisation** is possible (due to near zero zombie time)

Disadvantages of reference counting

- Not **comprehensive** (does not collect all garbage):
  cannot reclaim cyclic data structures
- High cost of manipulating RCs:
  cost is ever-present even if *no* garbage is collected
- Bad for concurrency; RC manipulations must be atomic — need `Compare&Swap` operation
- Tightly coupled interface to mutator
- High space overheads
- **Recursive freeing** cascade is only bounded by heap size

Tracing GC: idea

We can formalise our definition of reachability:

$$live = \{ N \in Objects \mid (\exists\, r \in Roots \,.\, r \rightarrow N) \;\vee\; (\exists\, M \in live \,.\, M \rightarrow N) \}$$

We can encode this definition simply

- Start at the roots; the live set is empty
- Add any object that a root points at to our live set
- Repeat
  Add any object a live object points at to the live set
  Until no new live objects are found
- Any objects not in the live set are garbage

Mark-Sweep

Mark-sweep is such a **tracing** algorithm — it works by following (*tracing)* references from live objects to find other live objects.

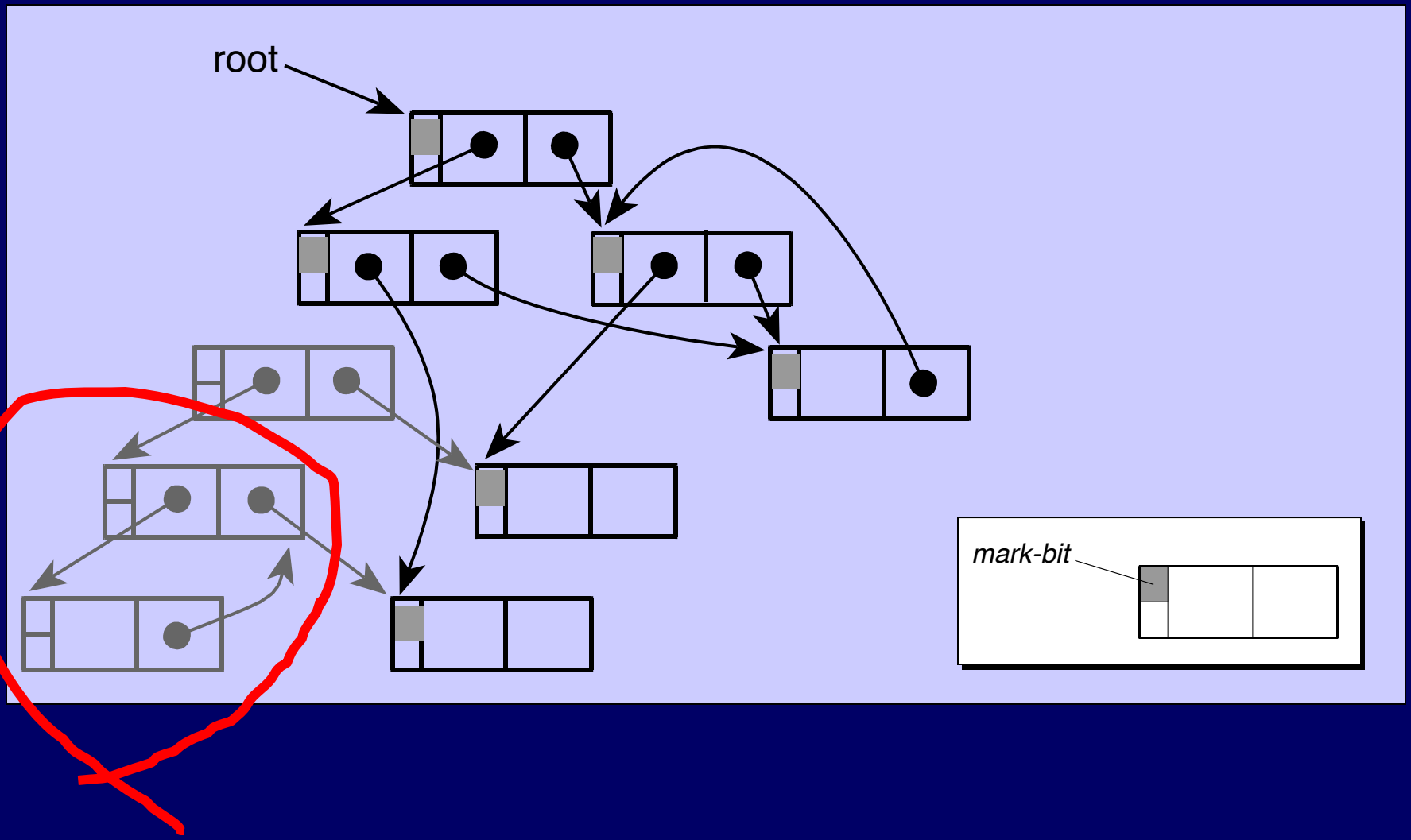
Implementation of the live set:
Each object has a **mark-bit** associated with it, indicating whether it is a member of the live set.

There are two phases:

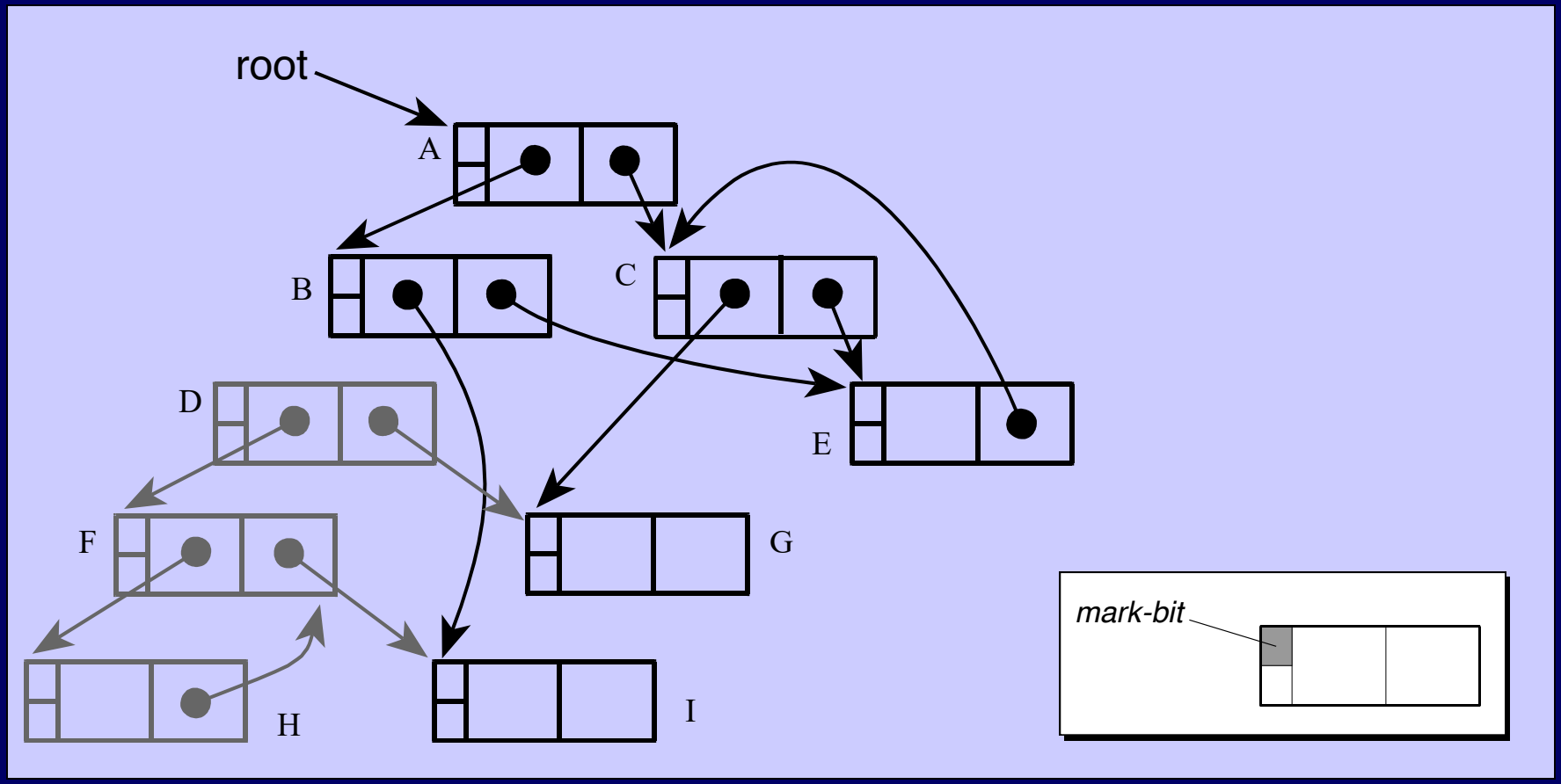
- **Mark phase:** starting from the roots, the graph is traced and the mark-bit is set in each unmarked object encountered.

  At the end of the mark phase, *unmarked* objects are garbage.

- **Sweep phase:** starting from the bottom, the heap is swept
  - mark-bit not set: the object is reclaimed
  - mark-bit set: the mark-bit is cleared


root
mark-bit

Marking exercise



”Must do” for pointer members in C++

For all pointer members check:

- Initialisation in *each* constructor
- Deletion in *assignment* operator
- Deletion in *destructor*
- Does copy constructor create shared objects?
- Is **creation** paired with **deletion**?

General advice C++

In general:

- Exploit the concept of ownership
- Check return value of **new** – it *may* be null!
- Adhere to convention, e.g., write **delete** if you write **new**
- Consider passing and returning objects by value.
- Writing a function that returns a dereferenced pointer is a memory leak waiting to happen!

The *Smart Pointer* Concept

Basic idea: allow the programmer to write code that is executed every time a pointer is manipulated:

- **Creation**
- **Assignment**
- **Copy constructor**

Smart pointers is a powerful language concept that can be used for many purposes including Garbage Collection.

The point: Smart pointers can be thought of as adding a level of indirection: Instead of having a reference to an object, you get a reference to a *smart pointer object* which executes some code every time you use the original reference. The smart pointer object contains a reference to the real object in question.

RC with smart pointers

Common C++ technique:

The basic idea is that the smart pointer object maintains a reference count together with the object reference count.

```
Template<typename T> class shared_ptr {
      T *ptr;
      long *rc;
   public:
      shared_ptr(T* p) : ptr(p) {
         rc = new long;
      }
      ~shared_ptr() { delete ptr; delete rc;}
```

Using Smart Pointers for RC

More smart pointer RC implementation

```
T& operator*()          { return *ptr;}
T* operator->()         { return ptr;}
shared_ptr& operator= (other object r) {
   if (--*rc == 0) { delete ptr;}  // last reference to object
   increment reference count for r
}
```

Smart Pointers comments

Smart pointers are ingenious but their actual implementation is quite gory.

However, they work and can be utilise for many purporses including RC GC.

ADVICE: start by using a publicly available implementation such as the one given in the notes.

Part 6: Advanced collectors

Simple tracing collectors suffer from several drawbacks

- disruptive delays
- repeated work on long-lived objects
- poor spatial locality

We now outline the approaches taken by sophisticated garbage collectors.

Generational GC

Weak generational hypothesis

"Most objects die young"

It is common for 80-95% objects to die before a further megabyte has been allocated

- 95% of objects are 'short-lived' in many Java programs
- 50-90% of CL and 75-95% of Haskell objects die before they are 10kb old
- SML/NJ reclaims 98% of any generation at each collection
- Only 1% Cedar objects survived beyond 721kb of allocation

Not a universal panacea

Generational GC is a successful strategy for many but not all programs.

There are common examples of programs that do not obey the weak generational hypothesis.

It is common for programs to retain most objects for a long time and then to release them all at the same time.

Generational GC imposes a cost on the mutator:

- ✗ pointer writes become more expensive

Incremental garbage collection

Incremental garbage collection

- runs collector in parallel with mutator
- attempts to bound pause time
- many soft real-time solutions
- but no general hard real-time solutions yet

Making GC incremental

Sequential GC can be made incremental by interleaving collection with allocation.

At each allocation, do a small amount of GC work.

Tune the rate of collection to the rate of allocation to prevent mutator running out of memory before collection is complete

Synchronisation

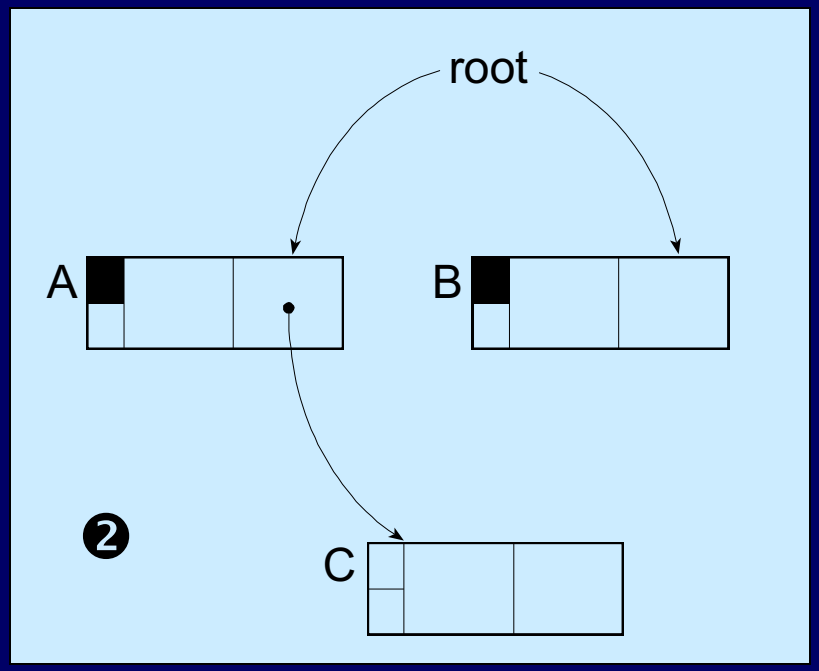
Asynchronous execution of mutator and collector introduces a coherency problem. For example, in the marking phase

Collector marks A

```
Update(right(B), right(A))
right(A) = nil                    ❶
```

Collector scans A

```
Update(right(A), right(B))
right(B) = nil                    ❷
```



Collector marks B

Collector scans B

Two ways to prevent disruption

There are two ways to prevent the mutator from interfering with a collection by writing white pointers into black objects.

1) Ensure the GC sees objects before the mutator does
   - when mutator attempts to access a white object, the object is visited by the collector
   - protect white objects with a **read-barrier**

2) Record where mutator writes pointers so that the GC can (re)visit objects
   - protect objects with a **write-barrier**

In conclusion

Points:

- Garbage collection is useful
- You can live without – albeit that can be painful
- Automatic mechanisms for GC are better – even at a slight extra execution time cost
- Conservative collectors actually work
- Classic algorithms reviewed
- There are many advanced collector available

Final Remarks

Garbage collection is a relatively mature technology.

But hard problems remain.

Commercial deployment of collector technology is still at an early stage. There are few players, and they use a small set of solutions.

There is no one magic solution to all problems: know your application!

Resources

- www.cs.ukc.ac.uk/people/staff/rej/gc.html

Copying GC Example