Runtime interface and language specific concerns

Oleg Dobkin

2/12/2014

Table of Contents

Object allocation and initialization sequence

- Allocate a block of requested size
  - Memory manager
- System initialization
  - System headers (i.e. virtual table in C++)
  - Object metadata
- Secondary initialization
  - User-defined object fields

Language examples

- C: includes only the first step
  - Programmer is responsible to initialize everything
- Java: first two steps result in empty but type-safe object. Third step applies constructor
  - User defined constructor 'sees' type-safe but empty object
- Haskell: values for all fields must be provided to the constructor
  - New object is fully initialized before it becomes accessible to the program

Runtime allocator post-conditions

- No initialization
- Allocated block is zeroed
- Object header is initialized
- Object header is initialized and block is zeroed
- Object is fully initialized

Zeroing - allocation guarantee

- C - **weak allocation guarantee**
  - No zeroing
- Java requires zeroing
- Functional languages - **strong allocation guarantee**
  - All fields values must be specified
  - Do not strictly need zeroing

Zeroing - when to zero

- At allocation
  - Increases allocation time
- After collection
  - Increases collection pause
  - Best from debugging point of view
- Best time - ahead of the allocator

Table of Contents

Pointer finding

Where to look for pointers

- Heap
- Objects
- Stack
- Registers
- Dynamic code

Types of pointer finding techniques

- Conservative - non-pointer values may be treated as a pointers, if they appear to refer to an allocated object
- Accurate - pointer locations are found exactly

Conservative pointer finding - main idea

Definition

Ambiguous pointer - each contiguous pointer-sized sequence of bytes that looks like a pointer

Finding ambiguous pointers

- Scan heap, stacks, registers
- Collector knows the address and size of heap and the addresses and sizes of allocated blocks inside the heap

Conservative pointer finding - interior pointers

Definition

Interior pointers - pointers that reference inside of an object

Handling interior pointers

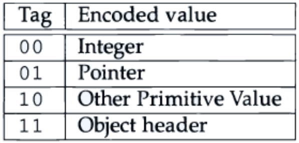
- Some languages require the pointers to refer to the first byte of the object
  - Instead of looking for allocated blocks - look in the list of objects
  - Caveat: even in such languages some pointers may still be interior

Conservative pointer finding - known implementation tricks

- Black-listing - avoid using heap regions (blocks), when their addresses correspond to previously found non-pointer values
- Avoid allocating blocks on addresses with many zeros
- Support for non-pointer blocks (bitmaps)

Accurate pointer finding using tagged values

- **Bit stealing** - use several bits to indicate object's type - usually to differentiate between pointers and non-pointers

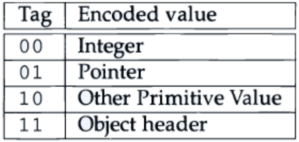


Tag	Encoded value
00	Integer
01	Pointer
10	Other Primitive Value
11	Object header

Table 11.2: Tag encoding for the SPARC architecture

Accurate pointer finding using tagged values

- **Bit stealing** - use several bits to indicate object's type - usually to differentiate between pointers and non-pointers



Tag	Encoded value
00	Integer
01	Pointer
10	Other Primitive Value
11	Object header

Table 11.2: Tag encoding for the SPARC architecture

- **Big bags of pages** - type information is associated with entire blocks
  - There are blocks in the heap containing only integers or only floating point numbers, ...

Accurate pointer finding in objects

- With reflection
  - Object metadata is stored in object's header
- Without reflection
  - **Bit vector**
  - **Vector of offsets** of pointer fields
    - Can be changed dynamically to control order of pointer tracing
  - **Partitioning** pointer and non-pointer data
  - Compiler-generated methods for tracing objects

Accurate pointer finding in stacks

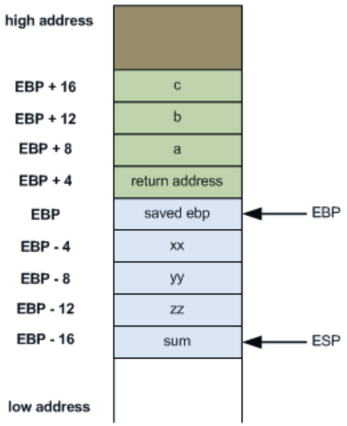
- Finding frames within the stack
  - Usually already provided by the runtime in some way
- Finding pointers within each frame
  - Stack maps
- Dealing with calling conventions

Stack structure

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```



http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86

Accurate pointer finding in registers

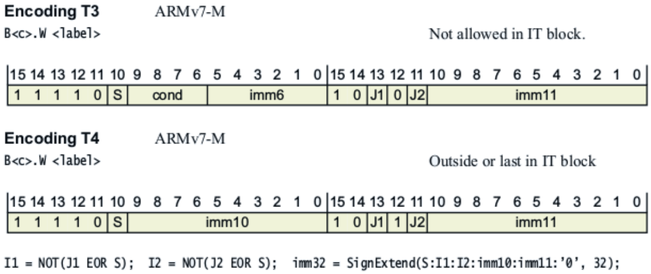
- Usually harder to 'partition' into pointer and non-pointer registers
- Even when the language guarantees no interior pointers, register may at some point refer to the middle of an object
- Harder to supply register map in some calling conventions

Accurate pointer finding in dynamic code (Problems)

- Not often possible to distinguish code from embedded data
- Pointers embedded in instructions may be broken into small pieces
- Embedded pointer value may not refer directly to its target object
- Difficult to update references

Accurate pointer finding in dynamic code (ARM/Thumb example)

- All opcodes are either 16-bit or 32-bit words (in original ARM - all opcodes are 32-bit words)
  - Impossible to store absolute addresses
  - Only relative references are supported
- Some of B - relative (conditional) branch - encodings



Encoding T3 ARMv7-M

`B<c>.W <label>` Not allowed in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	cond				imm6						1	0	J1	0	J2	imm11										

Encoding T4 ARMv7-M

`B<c>.W <label>` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	0	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
```

ARM Reference manual

Accurate pointer finding in dynamic code (Solutions)

- Disassemble code to find pointers
  - Large overhead, but must be done only once
- Generate side table with pointer locations
- Avoid embedded pointers

Handling interior pointers

Computing **standard reference** from interior pointer

- Table with starting addresses of each object
- Heap parsability
- Big bags of pages

Interior pointers with copying collectors

- Update interior pointer appropriately
- Pinning

Handling derived pointers

Definition

Derived pointer is a pointer that is derived from other pointers using arithmetic or logic expression

- $upper_k(p)$ or $lower_k(p)$
- $p \pm c$, such that the result lies outside of p
- $p - q$, distance between two objects

In general case we must have access to the base expression from which the pointer was derived

Table of Contents

Object tables

Definition

Object table - dense array of small records which describe objects

- Instead of mutator holding a direct pointer to an object in heap - let it hold a unique identifier (handle) into object table
- Object table must contain at least contains direct pointer to object
- Handle is:
  - Index into object table
    - Easier to relocate the table
    - Requires calculating entry's address
  - Pointer into table entry

Object tables (cont.)

Object tables advantages

- Easy to find and scan pointers
- Easy to move objects
  - Only the table entries must be scanned and updated
  - To simplify moving, each object should contain a (hidden) self-reference (back-pointer)
- Additional metadata may be stored in object table, for example, class and size information

Object tables (cont.)

Object tables advantages

- Easy to find and scan pointers
- Easy to move objects
  - Only the table entries must be scanned and updated
  - To simplify moving, each object should contain a (hidden) self-reference (back-pointer)
- Additional metadata may be stored in object table, for example, class and size information

Object tables disadvantages

- Performance hit
  - Two memory accesses instead of one (and optionally integer addition)
- No support for interior or derived pointers

Table of Contents

References from external (non-managed) code

Motivation

- Some languages/runtimes support passing heap pointers outside of managed environment - JNI, Python C interface, P/Invoke
- Every runtime must at some point use native OS interface, possibly passing pointers to it

Restraints

- Garbage collector must not free or move objects referenced from external code

References from external (non-managed) code (cont.)

- If the object must survive only for the duration of the call to external code, it's enough to leave a live reference to the object on stack
- Otherwise, external code must explicitly register an object it wishes to reference, and deregister when it's done using it
    - Collector may treat registered objects as additional roots
- Moving collectors can use handles (similar to object tables) instead of direct pointers
    - In cases where external code cannot use handles (OS code), collector may need to pin externally referenced objects

Pinning

- Defer collection, at least of a pinned object's region, while it's pinned
  - Simple, but no guarantee that the object will be unpinned before running out of memory
- Prior to pinning - move object to a non-moving region
- Extend collector to tolerate pinned objects
  - Complicates the collector and may introduce new inefficiencies

Table of Contents

Stack barriers

- Not safe to scan a frame in which a thread is running
- Usually not acceptable to pause a thread long enough to scan its entire stack

Stack barriers

- Not safe to scan a frame in which a thread is running
- Usually not acceptable to pause a thread long enough to scan its entire stack

- It's possible to scan a stack incrementally using stack barriers
  - Divert a thread by altering one of stored return addresses into a custom procedure

Stack barriers

- Not safe to scan a frame in which a thread is running
- Usually not acceptable to pause a thread long enough to scan its entire stack

- It's possible to scan a stack incrementally using stack barriers
    - Divert a thread by altering one of stored return addresses into a custom procedure

- Stack barriers can also mark portions of stack that have not changed
- Stack barriers can handle dynamic code changes

Table of Contents

GC-safe points

Definition

GC-safe point (GC-point) - point in code where GC can suspend the mutator (stop the world) and do its work

Most systems have short sequences of code that must be run entirely to preserve GC invariants

- Initializing a new object
- Setting up a new stack frame
- Write barrier

GC-safe points (cont.)

Collector managed GC points

Stopping a mutator's thread in an unsafe point will require

- Interpreting instruction ahead
- Waking up the thread for a short time, hoping it will stop in a safe point

Mutator managed GC points

- Mutator decides which points are safe and invokes collector explicitly

Table of Contents

Precision of write barriers

Definition

Write barriers detect and record interesting pointers in **remembered sets**.

Precision of write barriers

Definition

Write barriers detect and record interesting pointers in **remembered sets**.

- How accurately should pointer writes be recorded
  - Filtering

Precision of write barriers

Definition

Write barriers detect and record interesting pointers in **remembered sets**.

- How accurately should pointer writes be recorded
  - Filtering
- At what granularity is the location of the pointer to be recorded
  - Address of field into which the pointer was written
  - Address of object containing the field
  - Card tables
  - Virtual memory pages

Precision of write barriers

Definition

Write barriers detect and record interesting pointers in **remembered sets**.

- How accurately should pointer writes be recorded
  - Filtering
- At what granularity is the location of the pointer to be recorded
  - Address of field into which the pointer was written
  - Address of object containing the field
  - Card tables
  - Virtual memory pages
- Should remembered set be allowed to contain duplicates

Card tables

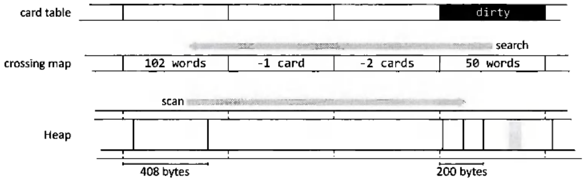
Definition

Card tables record written pointers with card's size granularity.
Implemented as dense array of cards (memory blocks) indexed by dense array of bytes (header), which mark dirty cards.

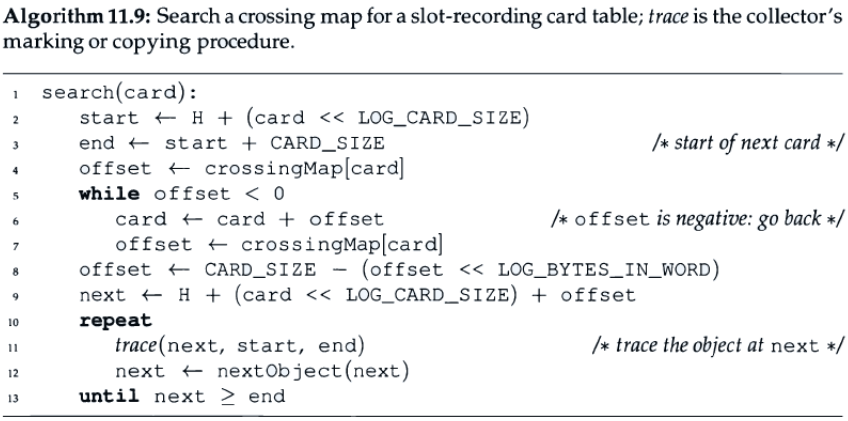
- Typically small cards 128-512 bytes
- Unlike hash tables or store buffers - can't overflow
- More work for the collector
- Commonly used without filtering
- The most compact header implementation - array of bits - not very common

Crossing maps

- Used to ease finding an object's start in a card table
- Crossing map holds as many entries as cards
- Each entry contains the offset to the last object in each card
- Entries corresponding to cards fully occupied by an object contain negative card offset



Crossing maps (cont.)

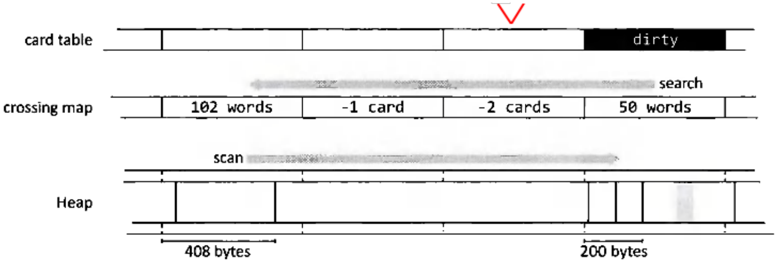


Algorithm 11.9: Search a crossing map for a slot-recording card table; *trace* is the collector's marking or copying procedure.

```
search(card):
    start ← H + (card << LOG_CARD_SIZE)
    end ← start + CARD_SIZE                                /* start of next card */
    offset ← crossingMap[card]
    while offset < 0
        card ← card + offset                       /* offset is negative: go back */
        offset ← crossingMap[card]
    offset ← CARD_SIZE − (offset << LOG_BYTES_IN_WORD)
    next ← H + (card << LOG_CARD_SIZE) + offset
    repeat
        trace(next, start, end)                         /* trace the object at next */
        next ← nextObject(next)
    until next ≥ end
```

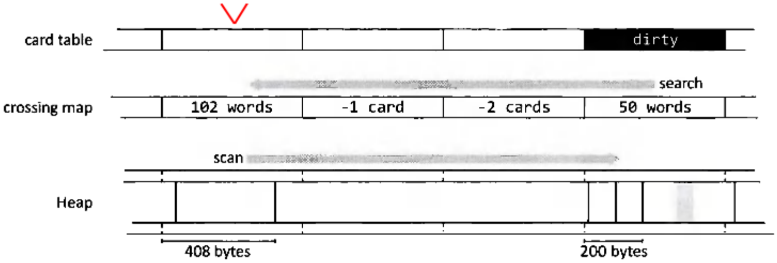
Crossing map - finding object's start 1

Start with one card before the dirty one
Look at the value of crossing map at that card and jump back



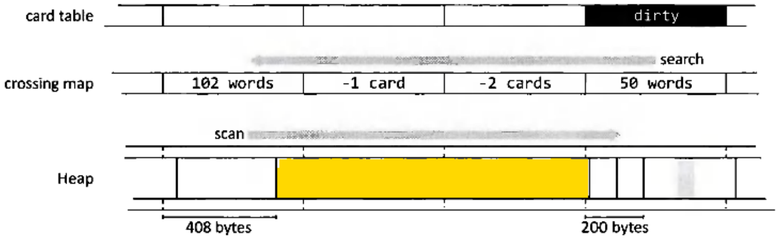
Crossing map - finding object's start 2

If crossing map at current location is positive - start tracing the last object in the card



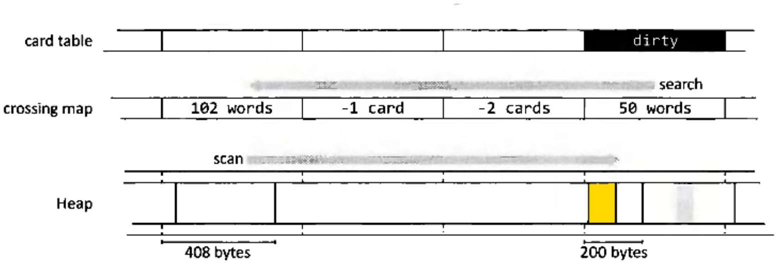
Crossing map - finding object's start 3

Trace objects until the end of the original dirty card



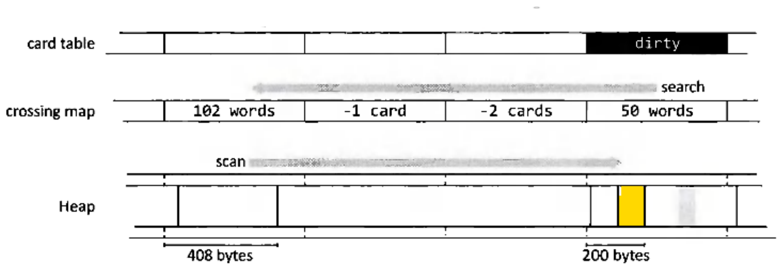
Crossing map - finding object's start 4

Trace objects until the end of the original dirty card



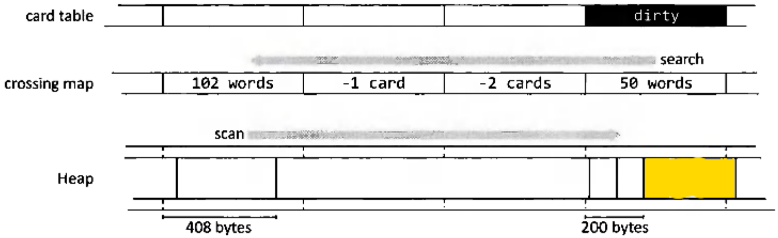
Crossing map - finding object's start 5

Trace objects until the end of the original dirty card



Crossing map - finding object's start 6

Trace objects until the end of the original dirty card



Hardware and virtual memory techniques

In case the compiler is uncooperative, the only choice is to use OS mechanisms for building remembered set

- Virtual memory manager always knows which pages are dirty
- Easily implemented by page protection alteration
- Costs little to mutator
- Reading dirty page information from OS is expensive
- Page protection faults are expensive
- OS pages are much larger than cards - collector needs to have very efficient scanning algorithms

Write barriers summary

- No clear winner amongst different remembered set mechanisms
- Page-based schemes performed worst
  - No choice, if compiler is uncooperative
- Card tables with card size of 512 performed best

Table of Contents

Finalization

Definition
Finalization - releasing unmanaged resources

- Managed object may refer to some external unmanaged resource (file, socket, database, ...)
- GC needs to allow programmer-specified action to finalize the object

Finalization

Definition

Finalization - releasing unmanaged resources

- Managed object may refer to some external unmanaged resource (file, socket, database, ...)
- GC needs to allow programmer-specified action to finalize the object

High-level implementation

- Runtime system maintains table of objects which have finalizers
- During collection pass, before freeing the object, GC invokes finalizer if needed

When do finalizers run

- During collection
  - Might not support general user code (i.e. allocation, global locks)

When do finalizers run

- During collection
  - Might not support general user code (i.e. allocation, global locks)
- After collection: collector marks objects that need finalization
  - How to mark objects
    - Queue objects - may need allocation
    - Mark object header - will need additional pass over all objects
  - During mutator lock - may cause deadlocks, if finalizer communicates with mutator threads

Which thread runs a finalizer

- Multi-threaded system
  - Most natural approach - to use background finalization thread(s)
  - Finalizers run asynchronously with mutator threads - need to be thread-safe
  - Finalizer for a type T might run at the same time as allocation/initialization code for new instance of T
  - Finalizers must be able to run concurrently with each other

Which thread runs a finalizer

- Multi-threaded system
    - Most natural approach - to use background finalization thread(s)
    - Finalizers run asynchronously with mutator threads - need to be thread-safe
    - Finalizer for a type T might run at the same time as allocation/initialization code for new instance of T
    - Finalizers must be able to run concurrently with each other
- Single-threaded system
    - The only feasible and safe way - queue finalizers and have the program to run them under explicit control

Can finalizers access the object that became unreachable

Motivation

- If finalizer does not have access to the object and is provided no additional data - it’s not very useful

Can finalizers access the object that became unreachable

Motivation

- If finalizer does not have access to the object and is provided no additional data - it's not very useful

Consequences

- Object queued for finalization survive the collection cycle
- All referenced objects must also be retained by the collector

Can finalizers access the object that became unreachable

Motivation

- If finalizer does not have access to the object and is provided no additional data - it's not very useful

Consequences

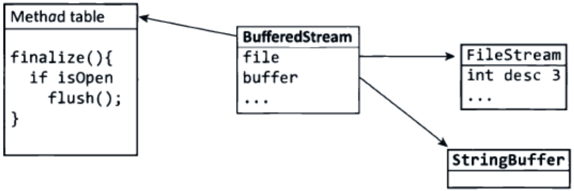
- Object queued for finalization survive the collection cycle
- All referenced objects must also be retained by the collector

Implementation

- Tracing collectors work in two passes
  - First pass: discover finalizable objects
  - Second pass: trace reachable objects
- Reference-counting collectors simply increment finalized object's reference count

Is there any guaranteed order to finalization

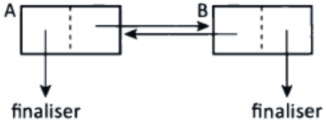
- Finalization order can matter (BufferedStream must be finalized before FileStream and StringBuffer)



- Finalize from higher layers to lower
- Ordered finalization may be slow - one ’level’ at each collection

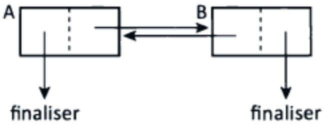
Is there any guaranteed order to finalization (cont.)

- Ordered finalization doesn't handle cycles

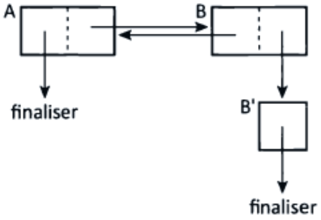


Is there any guaranteed order to finalization (cont.)

- Ordered finalization doesn't handle cycles



- Needs help from programmer
  - Field separation that breaks cycles



  - Weak references - may be tricky

Finalization in particular languages

Java

- `finalize` method of `Object` class
- Finalization order is not guaranteed
- Finalization runs in context starting with no (user-visible) synchronization locks
- Exceptions thrown during finalization are ignored
- Support for programmer-controlled finalization through `java.lang.ref` API

Finalization in particular languages (cont.)

Lisp

- Programmer can register any object with one or more finalization queues
- When object becomes unreachable it enters the finalization queue
- Programmer is responsible to empty the queues
- Order of acyclic references in the queue is guaranteed

Finalization in particular languages (cont.)

C++

- No explicit memory management
- Destructors are used to finalize objects and free memory
- Reference count can be implemented using destructors (smart pointers)

Finalization in particular languages (cont.)

.NET

- Support for both finalizers and destructors
- Finalizers are implemented with the use of `IDisposable` interface and some syntactic sugar
- Destructors are called by the collector

Table of Contents

Weak references

Definition

- **Weak references -** references that refer to their target at least as long as the target is strongly reachable from mutator roots
  - Object is **strongly reachable** if it's reachable via a chain of regular strong references
  - Object is **weakly reachable** if every path from the roots includes at least one weak reference

Weak references

Definition

- **Weak references** - references that refer to their target at least as long as the target is strongly reachable from mutator roots
  - Object is **strongly reachable** if it's reachable via a chain of regular strong references
  - Object is **weakly reachable** if every path from the roots includes at least one weak reference

Motivation

- Runtime can hold references to objects even when mutator doesn't
- Object caches
- Circular references

Weak references with tracing collectors

- First pass: weak references are only recorded but not traced
  - Only strongly reachable objects are found
- Second pass: weak references are examined
  - If weak reference's target was reached in first pass - weak reference is retained
  - Otherwise, it is set to null

Multiple pointer strengths

- Weak references can be generalized to provide multiple levels of weak pointers
- Each level of strength has some collector action associated with it

Multiple pointer strengths

- Weak references can be generalized to provide multiple levels of weak pointers
- Each level of strength has some collector action associated with it
- The best known language that supports multiple flavors of weak references is Java
  - **Strong**: ordinary references
  - **Soft**: collector *can* clear Soft reference at its discretion
  - **Weak**: collector *must* clear Weak reference as soon as the referent becomes weakly reachable
  - **Finalizer**: internal, used for finalization tables
  - **Phantom**: weakest kind, only permits clearing of the referent, can be used to control the order of finalization

Smart pointers in C++ (Boost/C++11)

- boost::scoped_ptr<>/std::unique_ptr<>
  - Simple non-copyable smart pointer
  - Releases object in destructor
  - Doesn't support sharing or transferring ownership
- boost::shared_ptr<>/std::shared_ptr<>
  - Supports shared ownership
  - Uses reference counting, last reference releases the referred object
  - Copy-constructor and `operator=` increase reference count
  - Cycles are not supported
- boost::weak_ptr<>/std::weak_ptr<>
  - Implements weak references
  - `shared_ptr` can be obtained through `weak_ptr` if the object is strongly reachable, otherwise will return empty `shared_ptr` or throw

Discussion

- OS mechanism to detect zero pointers
- Another use for garbage collector

(My) conclusions

- Garbage collection implementation is a very difficult task, depending on various factors:
  - Hardware and OS support (page protection, atomic operations, processor cache)
  - Language requirements (allowing interior pointers, using opaque handles, allowing weak references)
  - Working memory size and layout (long continuous memory patches)
- Good implementations must use deep internal knowledge of the underlying system
  - Useful processor instructions (for example, locked integer increments on x86 system, `InterlockedIncrement` in Windows)
  - Stack structure (calling conventions, stack traces, stack barriers)