Lecture 10

Memory Management: The Details

Sizing Up Memory

Primitive Data Types

- **byte**: basic value (8 bits)
- **char**: 1 byte
- **short**: 2 bytes
- **int**: 4 bytes
- **long**: 8 bytes
- **float**: 4 bytes
- **double**: 8 bytes

Not standard
May change

IEEE standard
Won't change

Complex Data Types

- **Pointer**: platform dependent
  - 4 bytes on 32 bit machine
  - 8 bytes on 64 bit machine
  - Java reference is a pointer
- **Array**: data size * length
  - Strings similar (w/ trailing null)
- **Struct**: sum of struct fields
  - Same rule for classes
  - Structs = classes w/o methods

Memory Example

class Date {	
short year;	2 byte
byte day;	1 byte
byte month;	1 bytes
}	4 bytes
class Student {	
int id;	4 bytes
Date birthdate;	4 bytes
Student* roommate;	4 or 8 bytes (32 or 64 bit)
}	12 or 16 bytes

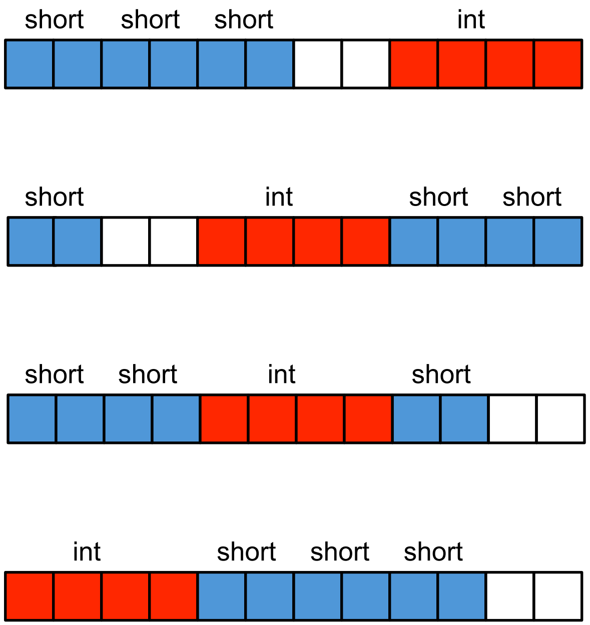
Memory Alignment

```
class Date {
    short year;
    byte day;
    byte month;
}
```

- All data types should align
  - Type starts at multiple of size
  - Shorts at even addresses
  - Ints/words at multiple of 4
  - Longs at multiple of 8
- Structs may require padding
  - Field order matters!
  - Pad between fields to align
  - Worse on 64 bit machines
- **Rule**: Order large to small

Memory Alignment

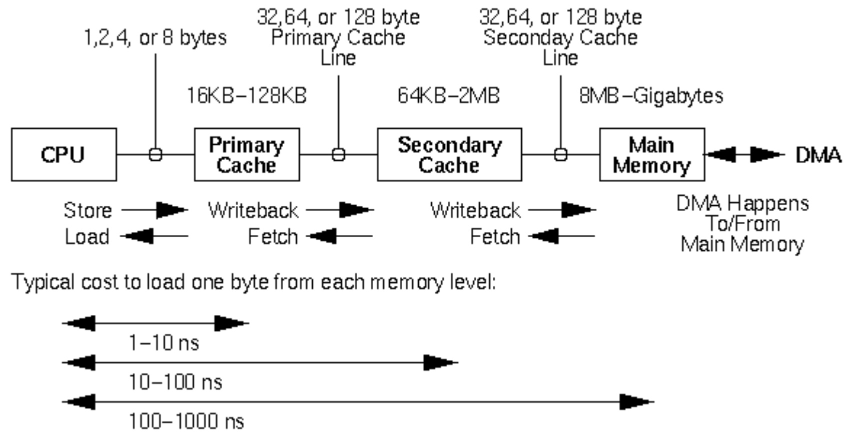
Struct w/ 3 shorts and an int:

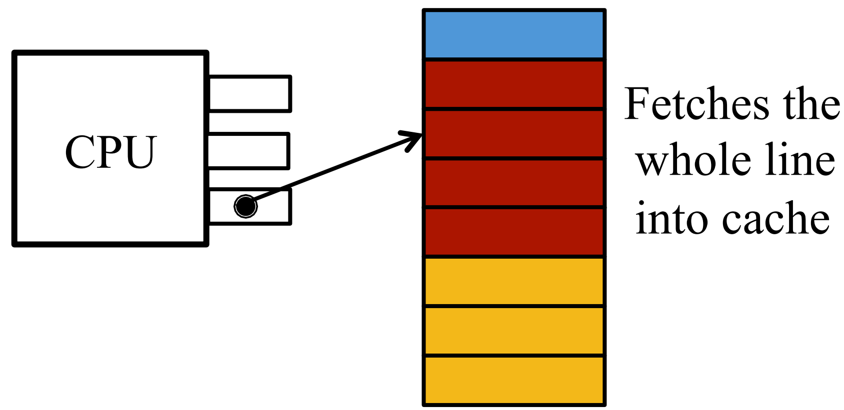


- All data types should align
  - Type starts at multiple of size
  - Shorts at even addresses
  - Ints/words at multiple of 4
  - Longs at multiple of 8
- Structs may require padding
  - Field order matters!
  - Pad between fields to align
  - Worse on 64 bit machines
- **Rule**: Order large to small

Related Topic: Cache Lines

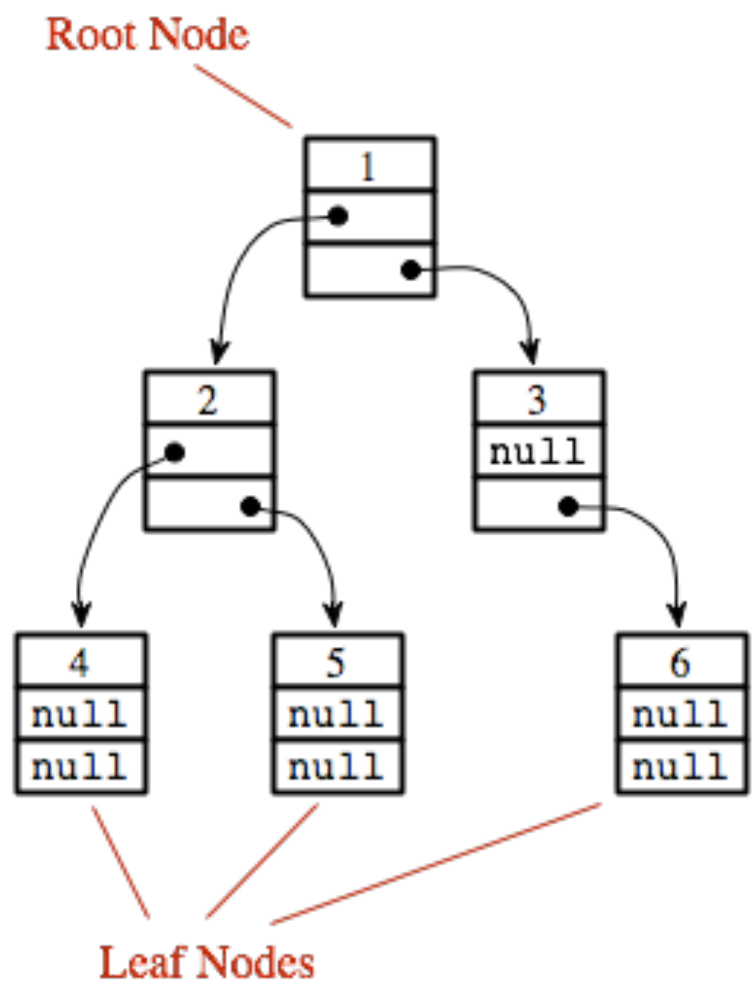
- All CPUs have **caches**
  - A6 (iOS): 32k L1, 1Mb L2
  - Snapdragon (Nexus): 4k L0, 16k L2, 2Mb L2
- Populate with **cache lines**
  - Data block of fixed size
  - Relative to cache size
  - Fetch pulls in whole line
- Can affect performance
  - Accessing neighbors is fast!
  - **Example**: array scanning





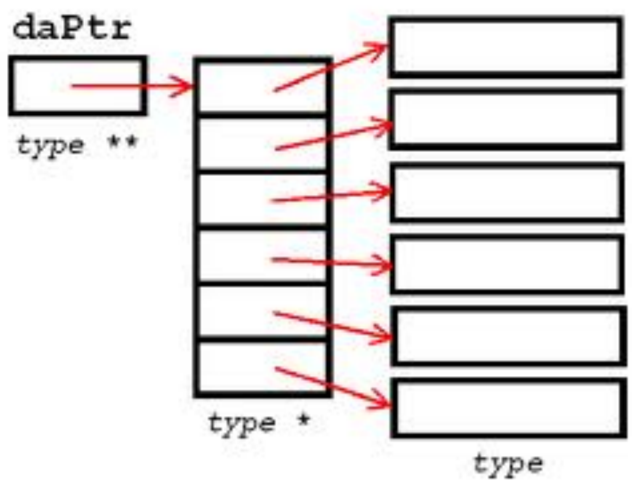
Data Structures and Memory

- Collection types are **costly**
  - Even null pointers use memory
  - Common for pointers to use as much memory as the pointees
  - Unbalanced trees are very bad
- Even true of (pointer) arrays
  - Array uses additional memory
- Not so in **array of structs**
  - Objects stored directly in array
  - But memory alignment!



Data Structures and Memory

- Collection types are **costly**
  - Even null pointers use memory
  - Common for pointers to use as much memory as the pointees
  - Unbalanced trees are very bad
- Even true of (pointer) arrays
  - Array uses additional memory
- Not so in **array of structs**
  - Objects stored directly in array
  - But memory alignment!



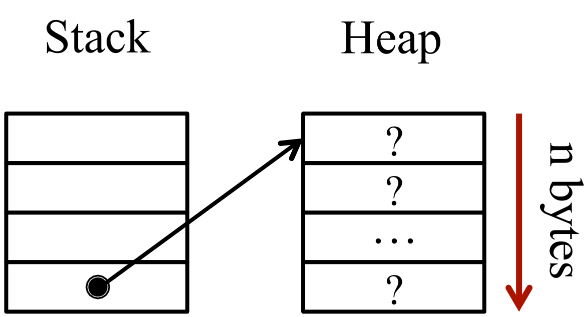
Two Main Concerns with Memory

- *Allocating Memory*
  - With OS support: **standard allocation**
  - Reserved memory: **memory pools**
- *Getting rid of memory* you no longer want
  - Doing it yourself: **deallocation**
  - Runtime support: **garbage collection**

C/C++: Allocation Process

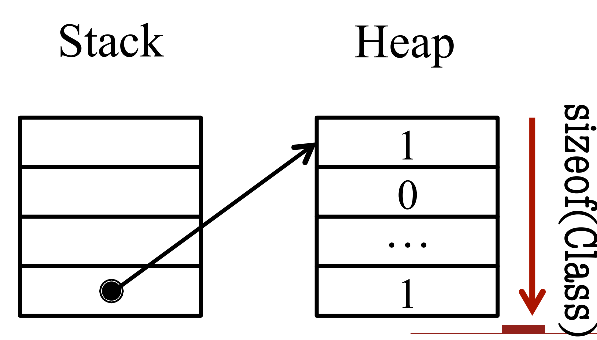
malloc

- Based on memory size
  - Give it number of **bytes**
  - Typecast result to assign it
  - No initialization at all
- **Example**:
  `char* p = (char*)malloc(4)`



new

- Based on data type
  - Give it a data type
  - If a class, calls constructor
  - Else no default initialization
- **Example**:
  `Point* p = new Point();`



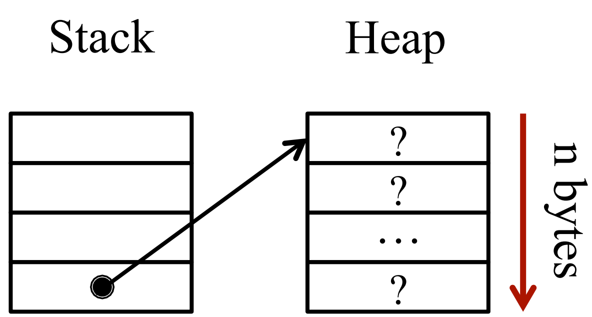
C/C++: Allocation Process

malloc

- Based on memory size
  - Give it number of **bytes**
  - Typecast re... it
  -
- **E**

Preferred in C

```
char* p = (char*)malloc(4)
```

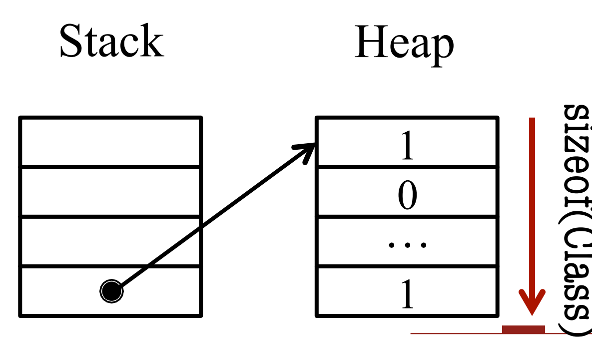


new

- Based on data type
  - Give it a data type
  - If a class, c... tor
  - ...ion
- **E**

Preferred in C++

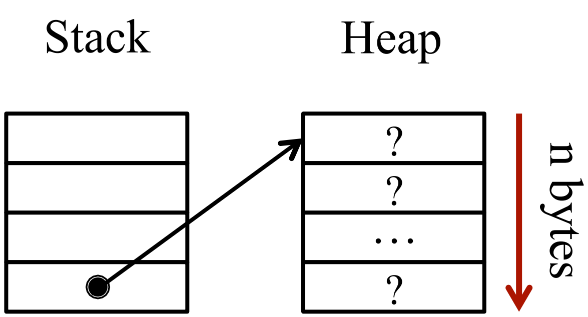
```
Point* p = new Point();
```



C/C++: Allocation Process

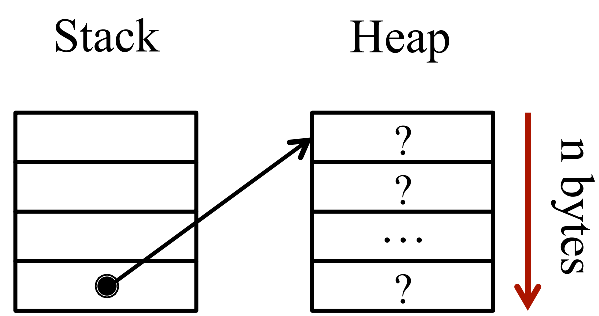
malloc

- Based on memory size
  - Give it number of **bytes**
  - Typecast result to assign it
  - No initialization at all
- **Example**:
  `char* p = (char*)malloc(4)`



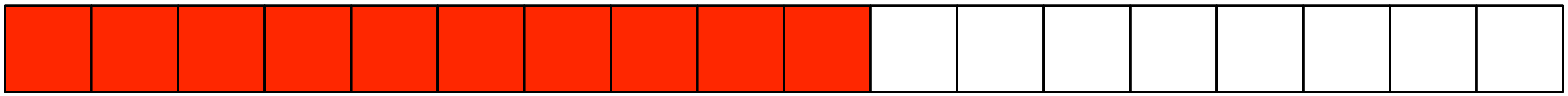
new

- **Can emulate malloc**
  - Create a char (byte) array
  - Arrays not initialized
  - Typecast after creation
- **Example**:
  `Point* p = (Point*)(new char[8])`



Recall: Custom Allocators

Pre-allocated Array (called **Object Pool**)



Start **Free** **End**

- Instead of `new`, get object from array
  - Just reassign all of the fields
  - Use **Factory pattern** for constructor
- Delete all objects at frame end
  - Just reset free pointer to start
  - Do not worry about freeing mid frame

Easy if only one object type to allocate

Anatomy of an Objective-C Constructor

```
NSDate* d1 = [NSDate dateWithString:"2014-03-05"];
```

Static Method
Allocates & Initializes

Anatomy of an Objective-C Constructor

```
NSDate* d1 = [NSDate dateWithString:"2014-03-05"];
```

Static Method
Allocates & Initializes

```
NSDate* d2 = [[NSDate alloc] initWithString:"2014-03-05"]];
```

Static Method
Allocates

Instance Method
Initializes

Custom Allocation in Objective-C

```
@implementation GObject

  static char* mempool = malloc(sizeof(GObject)*AMOUNT);
  static int pointer = 0;

  + (id)alloc {
    if (pointer >= AMOUNT) {
      return Nil;
    }
    pointer += sizeof(GObject);
    return (id)mempool[pointer-sizeof(Gobject)];
  }
```

Custom Allocation in Objective-C

```
@implementation GObject

  static char* mempool = malloc(sizeof(GObject)*AMOUNT);
  static int pointer = 0;

  + (id)alloc {
      if (pointer >= AMOUNT) {
          return Nil;
      }
      pointer += sizeof(GObject);
      return (id)mempool[pointer-sizeof(Gobject)];
  }
```

Fail gracefully

Object Pools In Java

```
public class GObjectFactory {

    private Gobject mempool = new Gobject[AMOUNT];
    private int pointer = 0;

    public GObjectFactory() {
        for(int ii = 0; ii < AMOUNT; ii++) {
            mempool[ii] = new GObject();
        }
    }
…
```

Initialize Pool

Object Pools In Java

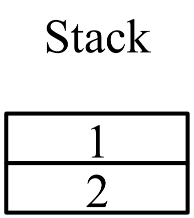
```
public class GObjectFactory {

    …

    public MakeGObject() {
        if (pointer >= AMOUNT) { return null; }
        GObject o = mempool[pointer++];

        // Initialize object here
        return o;
    }
```

Initialization & Allocation Combined

C++: Objects vs. Allocation

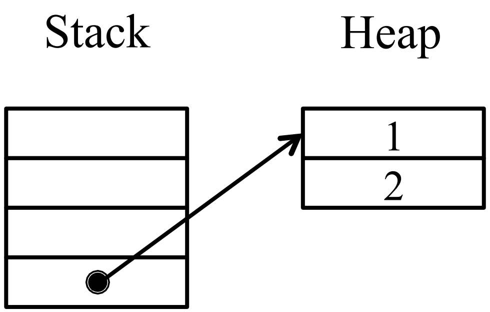
Stack Based Object

- Call with () after variable
  - Calls constructor with args
  - Puts object entirely on stack
  - Deleted when stack popped
- **Example**:
  `Point p(1,2);`



Heap Based Object

- Call with `new` syntax
  - Pointer on stack
  - Object in the heap
  - **Must be manually deleted**
- **Example**:
  `Point p* = new Point(1,2)`

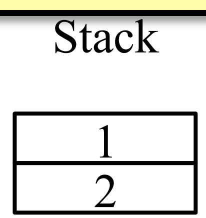


C++: Objects vs. Allocation

Stack Based Object

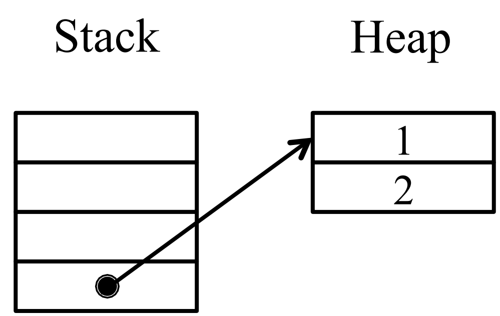
- Call with () after variable
  - Calls constructor with args

- Not in Java, C#, Obj-C
- But C#/Obj-C have **structs**
  - Classes without any methods
  - Can exist on the stack



Heap Based Object

- Call with `new` syntax
  - Pointer on stack
  - Object in the heap
  - **Must be manually deleted**
- **Example**:
  `Point p* = new Point(1,2)`



Two Main Concerns with Memory

- *Allocating Memory*
  - With OS support: **standard allocation**
  - Reserved memory: **memory pools**
- *Getting rid of memory* you no longer want
  - Doing it yourself: **deallocation**
  - Runtime support: **garbage collection**

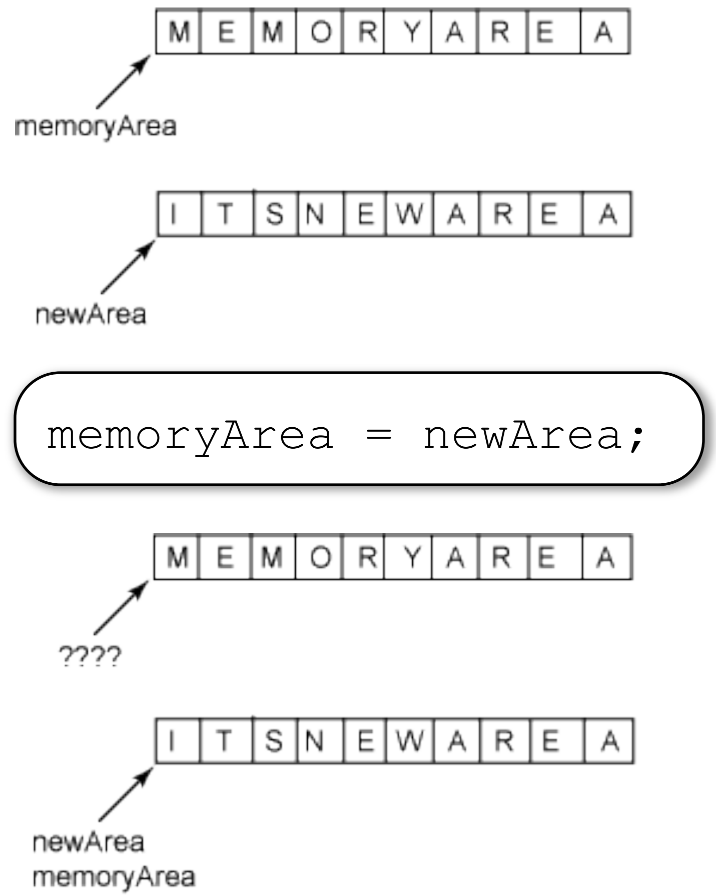
Manual Deletion in C/C++

- Depends on **allocation**
  - malloc: free
  - new: delete
- What does deletion do?
  - Marks memory as available
  - Does **not** erase contents
  - Does **not** reset pointer
- Only crashes if pointer bad
  - Pointer is currently NULL
  - Pointer is illegal address

```
int main() {
    cout << "Program started" << endl;
    int* a = new int[LENGTH];


    delete a;
    for(int ii = 0; ii < LENGTH; ii++) {
        cout << "a[" << ii << "]="
             << a[ii] << endl;
    }
    cout << "Program done" << endl;
}
```

Memory Leaks

- **Leak**: Cannot release memory
  - Object allocated on the heap
  - Only reference is moved
  - No way to reference object
- Consumes memory fast!
- Can even happen in Java
  - JNI supports native libraries
  - Method may allocate memory
  - Need another method to free
  - **Example**: dispose() in JOGL



A Question of Ownership

```
void foo() {
    MyObject* o =
        new MyObject();
    o.doSomething();
    o = null;
    return;
}
```

Memory Leak

```
void foo(int key) {
    MyObject* o =
        table.get(key);
    o.doSomething();
    o = null;
    return;
}
```

Not a Leak

A Question of Ownership

```
void foo() {
    MyObject* o =
        table.get(key);
    table.remove(key);
    o = null;
    return;
}
```

Memory Leak?

```
void foo(int key) {
    MyObject* o =
        table.get(key);
    table.remove(key);
    ntable.put(key,o);
    o = null;
    return;
}
```

Not a Leak

A Question of Ownership

Thread 1

Thread 2

"Owners" of obj

```
void run() {
    o.doSomething1();
}
```

```
void run() {
    o.doSomething2();
}
```

Who deletes obj?

Understanding Ownership

Function-Based

- Object owned by a function
  - Function allocated object
  - Can delete when function done
- Ownership *never transferred*
  - May pass to other functions
  - But always returns to owner
- Really a **stack-based object**
  - Active as long as allocator is
  - But allocated on heap (why?)

Object-Based

- Owned by another object
  - Referenced by a field
  - Stored in a data structure
- Allows *multiple ownership*
  - No guaranteed relationship between owning objects
  - Call each owner a reference
- When can we deallocate?
  - No more references
  - References "unimportant"

Understanding Ownership

Function-Based

- Object owned by a function
  - Function allocated object
  - Can delete when function done
- Owne
  - returns to owner

Easy: Will ignore

- Really a **stack-based object**
  - Active as long as allocator is
  - But allocated on heap (why?)

Object-Based

- Owned by another object
  - Referenced by a field
  - Stored in a data structure
- Allows *multiple ownership*
  - No guaranteed relationship between owning objects
  - Call each owner a reference
- When can we deallocate?
  - No more references
  - References "unimportant"

Reference Strength

Strong Reference

- Reference asserts ownership
  - Cannot delete referred object
  - Assign to NULL to release
  - Else assign to another object
- Can use reference **directly**
  - No need to copy reference
  - Treat like a normal object
- Standard type of reference

Weak Reference

- Reference != ownership
  - Object can be deleted anytime
  - Often for *performance caching*
- Only use **indirect** references
  - Copy to local variable first
  - Compute on local variable
- Be prepared for NULL
  - Reconstruct the object?
  - Abort the computation?

Weak Reference Example in Java

```
public class CombatComponent {
    WeakReference<NPC> target;
    …
    public void attackTarget(AIController ai) {
        NPC theTarget = target.get();
        if (theTarget == null) {    // Be prepared for NULL
            theTarget = ai.pickCombatTargetFor(this);
            target = new WeakReference<NPC>(theTarget);
        }
        // Do stuff with theTarget
        …
    }
}
```

Class in package java.lang.ref

Weak Reference Example in Java

```
public class CombatComponent {
    WeakReference<NPC> target;
    …
    public void attackTarget(AIController ai) {
        NPC theTarget = target.get();
        if (theTarget == null) {    // Be prepared f
            theTarget = ai.pickCombatTargetFor(thi
            target = new WeakReference<NPC>(theT
        }
        // Do stuff with theTarget
        …
    }
}
```

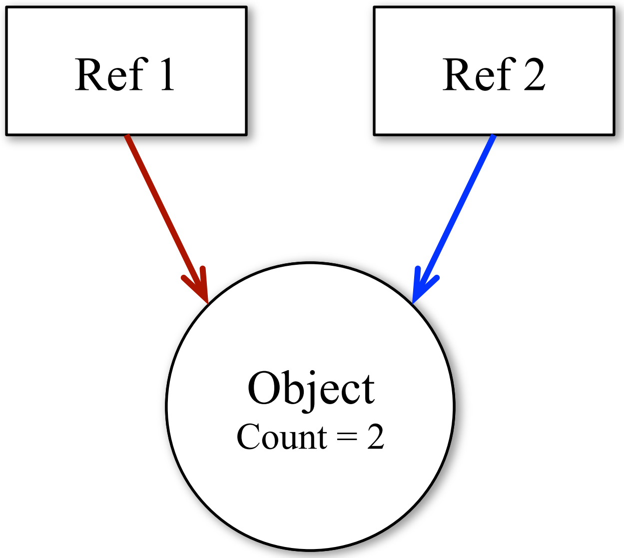
Class in package java.lang.ref

Reference Managers in Java

- WeakReference
  - GC if no standard references left
  - Encourages *eager* GC policies
- SoftReference
  - GC *possible* if no references left
  - But only if space is needed

Reference Counting

- Every object has a **counter**
  - Tracks number of "owners"
  - No owners = memory leak
- Increment when assign reference
  - Can be explicit method call
  - …can we do it automatically?
- Decrement when remove reference
  - Can be explicit or automatic
  - If makes count 0, delete it



When to Adjust the Count?

- On object allocation
  - Initial allocator is an owner
  - Even if in a local variable
- When added to an object
  - Often handled by setter
  - Part of class invariant
- When removed from object
  - Also handled by the setter
  - Release before reassign
- Any other time?

```
public class Container {
    RCObject object;
    public Container() {
        // Initial allocation; ownership
        Object = new RCObject();
    }
    …
    public void setObject(RCObject o) {
        if (object != null) {
            object.decrement();
        }
        o.increment(); object = o;
    }
}
```

Reference Counting in Obj-C

```
// create a new instance
Fraction *frac = [[Fraction alloc] init];
[frac retain];

// set the values and print
[frac setNumerator: 1];
[frac setDenominator: 3];

printf( "The fraction is: " );
[frac print];
printf( "\n" );

// free memory
[frac release];
[frac release];
```

Reference count 1

Reference count 2

Reference count 1

Reference count 0

frac is deleted

Reference Counting in Classic Obj-C

```
// create a new instance
Fraction *frac = [[Fraction alloc] init];
[frac retain];

// set the values and print
[frac setNumerator: 1];
[frac setDenominator: 3];

printf( "The fraction is: " );
[frac print];
printf( "\n" );

// free memory
[frac release];
```

Reference count 1

Reference count 2

Reference count 1

Memory Leak!

Which Is Correct?

```
Fraction* foo(int n, int d) {

    // create a new instance
    Fraction *frac =
        [[Fraction alloc] init];

    // set the values
    [frac setNumerator: n];
    [frac setDenominator: d];

    // free memory
    [frac release];

    // return it
    return frac;

}
```

```
Fraction* foo(int n, int d) {

    // create a new instance
    Fraction *frac =
        [[Fraction alloc] init];

    // set the values
    [frac setNumerator: n];
    [frac setDenominator: d];

    // Do nothing


    // return it
    return frac;

}
```

Which Is Correct?

```
Fraction* foo(int n, int d) {

    // create a new instance
    Fraction *frac =
            [[Fraction alloc] init];

    // set the values
    [frac setNumerator: n];
    [frac setDeno

    // free mem
    [frac release];

    // return it
    return frac;

}
```

Trick Question!

```
Fraction* foo(int n, int d) {

    // create a new instance
    Fraction *frac =
            [[Fraction alloc] init];

    : n];
    ator: d];

    // Do nothing

    // return it
    return frac;

}
```

Neither is Optimal

```
Fraction* foo(int n, int d) {

    // create a new instance
    Fraction *frac =
        [[Fraction alloc] init];

    // set the values
    [frac setNumerator: n];
    [frac setDenominator: d];

    // free memory
    [frac release];

    // return it
    return frac;

}
```

Object freed.
Nothing left to return.

```
Fraction* foo(int n, int d) {

    //

    [frac

    // Do nothing

    // return it
    return frac;

}
```

One possibility: make **ownership transfer** part of the specification

Reference kept.
Who will release this reference?

Objective-C: An Alternate Solution

```
Fraction* foo(int n, int d) {

    // create a new instance
    Fraction *frac =
        [[Fraction alloc] init];

    // set the values
    [frac setNumerator: n];
    [frac setDenominator: d];

    // free memory
    [frac autorelease];

    // return it
    return frac;

}
```

Delay release until later. Handled by the OS.

Autorelease

- Places the object in a pool
  - OS releases all in the pool
  - At the end of event handler
- Games are not event driven!
  - Game loop runs continuously
  - **Pool is not released**
- You can release it manually
  - At end of loop iteration?

Objective-C: An Alternate Solution

```
Fraction* foo(int n, int d) {
    // create a new instance
    Fraction *frac =
        [[Fraction alloc] init];

    // set the values
    [frac setNumerator: n];
    [frac setDenominator: d];

    // free memory
    [frac autorelease];

    // return it
    return frac;
}
```

Delay release until later. Handled by the OS.

Al Demers says:
This is a hack.
Bad Apple, bad.

ames are not event driven!
- Game loop runs continuously
- **Pool is not released**

- You can release it manually
  - At end of loop iteration?

Reference Counting in iOS 5

```
// create a new instance
Fraction *frac = [[Fraction alloc] init];
[frac retain];

// set the values and print
[frac setNumerator: 1];
[frac setDenominator: 3];

printf( "The fraction is: " );
[frac print];
printf( "\n" );

// free memory
[frac release];
```

No-op (does nothing)

No-op (does nothing)

Automated Reference Counting

- Handled by the compiler
  - Inserts retain/release for you
  - Still reference counting, not GC
- Old methods are deprecated
  - Backwards compatibility only
  - No-ops if ARC is turned on

C++ Analogue: Smart Pointers

- C++ can override **anything**
  - Assignment operator =
  - Dereference operator ->
- Use special object as pointer
  - A field to reference object
  - Also a reference counter
  - Assignment increments
- What about decrementing?
  - When smart pointer deleted
  - Delete object if count is 0

```
void foo(){
    // Create smart pointer
    smart_ptr<MyObject> p();

    // Allocate & assign it
    p = new MyObject();

    p->doSomething();

    // NO LEAK

}
```

C++ Analogue: Smart Pointers

- C++ can override **anything**
  - Assignment operator =
  - Dereference operator ->
- Use special object as reference
  - A field to reference object
  - Also a reference counter
  - Assignment increments
- What about decrementing?
  - When smart pointer deleted
  - Delete object if count is 0

```
void foo() {
  // Create smart pointer
  smart_ptr<MyObject> p();
  // Allocate & assign it
  p = new MyObject();
  p->doSomething();
  // NO LEAK
}
```

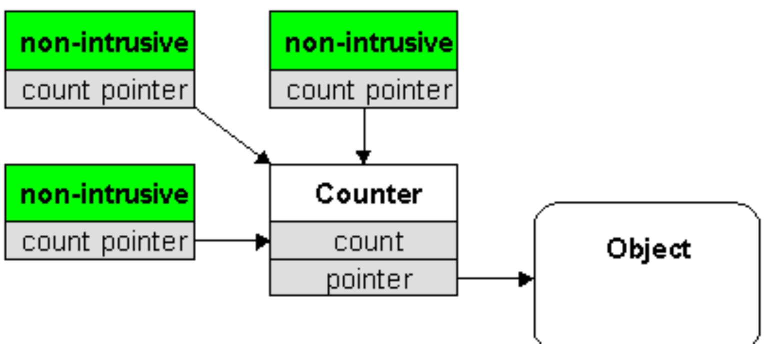
Stack object; not in heap.

Stack released; Smart pointer is disposed.

Where Does the Count Go?

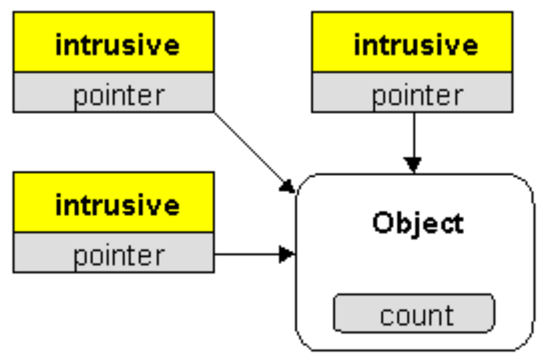
Non-Intruisive Pointers

- Count inside smart pointer
- **Advantage**:
  - Works with any class
- **Disadvantage**:
  - Combining with raw pointers (and hence any stdlib code)



Intruisive Pointers

- Count inside referred object
- Advantage:
  - Easy to mix with raw pointers
- Disadvantage:
  - Requires custom base object



[Images courtesy of Kosmas Karadimitriou]

References vs. Garbage Collectors

Reference Counting

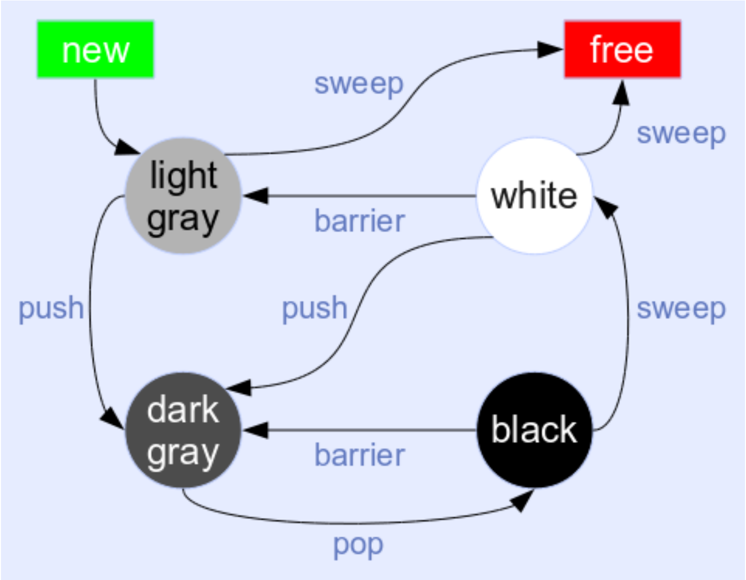
- **Advantages**
  - Deallocation is immediate
  - Works on non-memory objects
  - Ideal for real-time systems
- **Disadvantages**
  - Overhead on every assignment
  - **Cannot easily handle cycles** (e.g. object points to itself)
  - Requires training to use

Mark-and-Sweep

- **Advantages**
  - No assignment overhead
  - Can handle reference cycles
  - No specialized training to use
- **Disadvantages**
  - Collection can be expensive
  - Hurts performance when runs
  - Usually triggered whenever the memory is close to full

Incremental Mark-and-Sweep

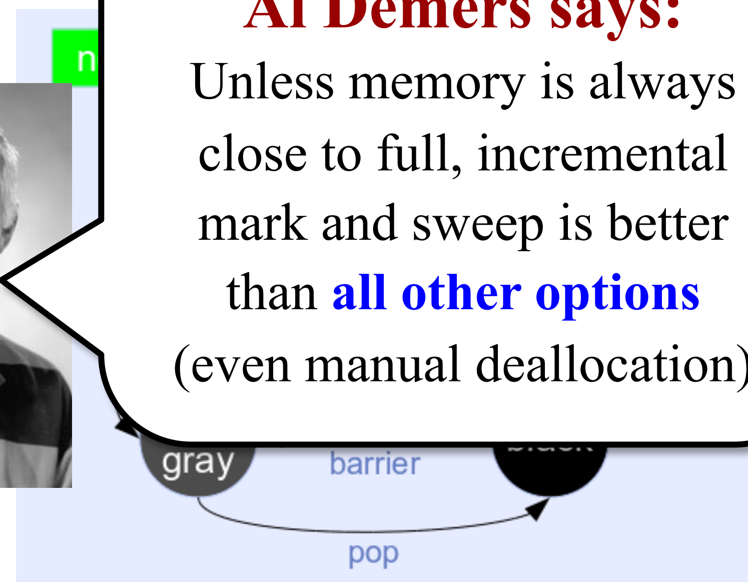
- Objects have multiple "colors"
  - Indicate where in GC process
  - At GC, change *some* colors
  - Free objects of certain color
- Natural for game loops
  - Give GC a time budget
  - Do at end of game loop
  - Stop when done or time up
- See online reading for more



Lua 3.0: Quad-Color M&S

Incremental Mark-and-Sweep

- Objects have multiple "colors"
  - Indicate where in GC process
  - At GC, change *some* colo
  - Free objects of certain co
- Natural for game loops
  - Give GC a time budget
  - Do at end of game loop
  - Stop when done or time up
- See online reading for more

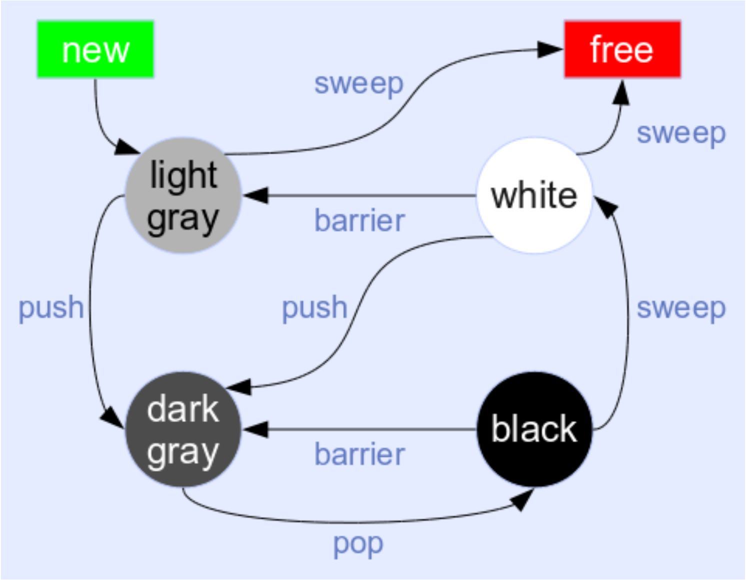


Al Demers says:
Unless memory is always close to full, incremental mark and sweep is better than **all other options** (even manual deallocation)

Lua 3.0: Quad-Color M&S

Incremental Mark-and-Sweep

- Problem is **availability**
  - Not available in Java
  - Objective-C uses ARC
  - No good C/C++ libraries
- You need to **implement**!
  - Create a singleton allocator
  - Has (weak) references to all objects that it allocates
- Only for complex games



Lua 3.0: Quad-Color M&S

Summary

- Memory management a major challenge
  - Manual deallocation is difficult
  - Garbage collection is better on paper
  - But not all techniques easily available
- Custom allocators common in AAA games
  - Organize memory in pools and budget use
  - Could theoretically support GC as well