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

## 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 same (w/ trailing null)
- **Struct**: sum of fields
  - Same rule for classes
  - Structs = classes w/o methods

---

**Memory Details**

- **Not standard**: May change
- **IEEE standard**: Won’t change
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 and Pointer Casting

- C++ allows **ANY** cast
  - Is not “strongly typed”
  - Assumes you know best
  - But must be **explicit** cast

- **Safe** = aligns properly
  - Type should be same size
  - Or if array, multiple of size

- **Unsafe** = data corruption
  - It is all your fault
  - Large cause of seg faults

```cpp
// Floats for OpenGL
float[] lineseg = {0.0f, 0.0f, 2.0f, 1.0f};

// Points for calculation
Vec2* points

// Convert to the other type
points = (Vec2*)lineseg;

for(int ii = 0; ii < 2; ii++) {
    CCLOG("Point %4.2, %4.2",
          points[ii].x, points[ii].y);
}
```
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!
• 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:**
```c
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:**
```c
Point* p = new Point();
```

---

**Memory Details**

Stack | Heap
---|---
? | ?
? | ...
? | ?

Stack | Heap
---|---
1 | 0
| ... |
| 1 |
C/C++: Allocation Process

**malloc**
- Based on memory size
  - Give it number of bytes
  - Typecast result and use it
- 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();`

---

**Memory Details**

Stack | Heap
---|---

Preferred in C:

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

Preferred in C++:

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

```
<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
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])
```

---

**Memory Details**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
Custom Allocators

Pre-allocated Array (called Object Pool)

- **Idea**: Instead of `new`, get object from array
  - Just reassign all of the fields
  - Use **Factory pattern** for constructor
  - See `create()` method in Cocos2D-x objects

- **Problem**: Running out of objects
  - We want to reuse the older objects
  - Easy if deletion is FIFO, but often isn’t
Free Lists

- Create an object **queue**
  - Separate from preallocation
  - Stores objects when “freed”

- To allocate an object...
  - Look at front of free list
  - If object there take it
  - Otherwise make new object

- Preallocation unnecessary
  - Queue wins in long term
  - Main performance hit is deletion/fragmentation

// Free the new particle
freelist.push_back(p);
...

// Allocate a new particle
Particle* q;
if (!freelist.isEmpty()) {
  q = freelist.pop();
} else {
  q = new Particle();
}
q.set(...)

Memory Management
Particle Pool Example

Memory Details

---

GL verts: 178
GL calls: 3
60.3 / 0.015

Allocated: 16
Particle Pool Example

class ParticlePool {
public:
    /** Creates a ParticlePool with the given capacity. */
    ParticlePool(int capacity);
    /** Returns a new OR reused object from this pool. */
    Particle* obtain();
    /** Marks object as eligible for reuse. */
    void free(Particle* object);

private:
    /** Allocates a new object from the pool. */
    Particle* alloc();
};
Particle Pool Example

```cpp
class ParticlePool {

public:

    /** Creates a ParticlePool with the given capacity. */
    ParticlePool(int capacity);

    /** Returns a new OR reused object from this pool. */
    Particle* obtain();

    /** Marks object as eligible for reuse. */
    void free(Particle* object);  // Use instead of delete

private:

    /** Allocates a new object from the pool. */
    Particle* alloc();

};
```

Memory Details
Particle Pool Example

```cpp
class ParticlePool {
public:
    /** Creates a ParticlePool with the given capacity. */
    ParticlePool(int capacity);
    /** Returns a new OR reused object from this pool. */
    Particle* obtain();
    /** Marks object as eligible for reuse. */
    void free(Particle* object);

private:
    /** Allocates a new object from the pool. */
    Particle* alloc();
};
```

Use instead of `new`

Use instead of `delete`

What to do if nothing free
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

```cpp
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 heap
  - Only reference is moved
- 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
void foo() {
    MyObject* o = new MyObject();
    o.doSomething();
    o = null;
    return;
}

void foo(int key) {
    MyObject* o = table.get(key);
    o.doSomething();
    o = null;
    return;
}
A Question of Ownership

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

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

Thread 1

Thread 2

"Owners" of obj

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

Who deletes obj?

void run() {
    o.doSomething2();
}
# 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”

---

Memory Details
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”

Easy: Will ignore
# Reference Strength

<table>
<thead>
<tr>
<th>Strong Reference</th>
<th>Weak Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Reference asserts ownership</td>
<td>• Reference != ownership</td>
</tr>
<tr>
<td>• Cannot delete referred object</td>
<td>• Object can be deleted anytime</td>
</tr>
<tr>
<td>• Assign to NULL to release</td>
<td>• Often for <em>performance caching</em></td>
</tr>
<tr>
<td>• Else assign to another object</td>
<td></td>
</tr>
<tr>
<td>• Can use reference <strong>directly</strong></td>
<td>• Only use <strong>indirect</strong> references</td>
</tr>
<tr>
<td>• No need to copy reference</td>
<td>• Copy to local variable first</td>
</tr>
<tr>
<td>• Treat like a normal object</td>
<td>• Compute on local variable</td>
</tr>
<tr>
<td>• Standard type of reference</td>
<td>• Be prepared for NULL</td>
</tr>
<tr>
<td></td>
<td>• Reconstruct the object?</td>
</tr>
<tr>
<td></td>
<td>• Abort the computation?</td>
</tr>
</tbody>
</table>

**Memory Details**

<table>
<thead>
<tr>
<th>Reference Strength</th>
<th>Weak Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Reference</td>
<td>Weak Reference</td>
</tr>
<tr>
<td>Reference asserts ownership</td>
<td>Reference != ownership</td>
</tr>
<tr>
<td>Cannot delete referred object</td>
<td>Object can be deleted anytime</td>
</tr>
<tr>
<td>Assign to NULL to release</td>
<td>Often for <em>performance caching</em></td>
</tr>
<tr>
<td>Else assign to another object</td>
<td></td>
</tr>
<tr>
<td>Can use reference <strong>directly</strong></td>
<td>Only use <strong>indirect</strong> references</td>
</tr>
<tr>
<td>No need to copy reference</td>
<td>Copy to local variable first</td>
</tr>
<tr>
<td>Treat like a normal object</td>
<td>Compute on local variable</td>
</tr>
<tr>
<td>Standard type of reference</td>
<td>Be prepared for NULL</td>
</tr>
<tr>
<td></td>
<td>• Reconstruct the object?</td>
</tr>
<tr>
<td></td>
<td>• Abort the computation?</td>
</tr>
</tbody>
</table>
Reference Counting

- Every object has a **counter**
  - Tracks number of “owners”
  - No owners = memory leak
- Increment when assign reference
  - Often an explicit method call
  - Historically called **retain()**
- Decrement when remove reference
  - Method call is **release()**
  - 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?

```cpp
class Container {
public:
    RCObject* object;

    Container() {
        // Initial allocation; ownership
        object = new RCObject();
        object->retain();
    }

    void setObject(RCObject o) {
        if (object != null) {
            object->release();
            o->retain();
            object = o;
        }
    }
};
```
// create a new instance
Sprite* sprite = new Sprite();
sprite->retain(); Reference count 1

// Add the sprite to scene graph
rootnode->addChild(sprite); Reference count 2

// Release the local reference
sprite->release(); Reference count 1

// Remove from scene graph
scene->removeChild(sprite); Reference count 0
sprite is deleted
Reference Counting in Cocos2d-x

// create a new instance
Sprite* sprite = new Sprite();
sprite->retain();  // Reference count 1

// Add the sprite to scene graph
rootnode->addChild(sprite);  // Reference count 2

// Do not release the local reference

// Remove from scene graph
scene->removeChild(sprite);  // Reference count 1

Memory Leak!
Which Is Correct?

Sprite* foo(float x, float y) {

    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // free memory
    sp->release();

    // return it
    return sp;
}

Sprite* foo(float x, float y) {

    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // DO NOTHING

    // return it
    return sp;
}
Which Is Correct?

Sprite* foo(float x, float y) {
    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");
    // set the position
    sp.setPosition(Vec2(x, y));
    // free memory
    sp->release();
    // return it
    return sp;
}

Sprite* foo(float x, float y) {
    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");
    // set the position
    sp.setPosition(Vec2(x, y));
    // DO NOTHING
    // return it
    return sp;
}
Neither Choice is Optimal

Sprite* foo(float x, float y) {

    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // free memory
    sp->release();

    // return it
    return sp;
}

Sprite* foo(float x, float y) {

    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // DO NOTHING

    // return it
    return sp;
}

Memory Details

Neither Choice is Optimal

Sprite* foo(float x, float y) {

    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // free memory
    sp->release();

    // return it
    return sp;
}

Sprite* foo(float x, float y) {

    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // DO NOTHING

    // return it
    return sp;
}

Memory Details
Neither Choice is Optimal

Sprite* foo(float x, float y) {
    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // free memory
    sp->release();

    // return it
    return sp;
}

Sprite* foo(float x, float y) {
    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // DO NOTHING

    // return it
    return sp;
}

One possibility: make ownership transfer part of the specification

Reference kept. Who will release this?

Memory Details

Neither Choice is Optimal

Sprite* foo(float x, float y) {
    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // free memory
    sp->release();

    // return it
    return sp;
}

Sprite* foo(float x, float y) {
    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // DO NOTHING

    // return it
    return sp;
}

One possibility: make ownership transfer part of the specification

Reference kept. Who will release this?

Memory Details
Ownership in Specifications

/**
 * Creates a sprite at (x,y)
 *
 * @release object ownership passes to the caller
 *
 * @return a new sprite
 */

Sprite* foo(float x, float y) {
    ...
}

/**
 * Stores the given sprite
 *
 * @retain container acquires ownership of sprite
 *
 * @param sp sprite to store
 */

void foo(const Sprite* sp) {
    ...
}
An Alternate Solution

Sprite* foo(float x, float y) {

    // create a new instance
    Sprite* sp = new Sprite();
    sp->initWithFile("image.png");

    // set the position
    sp.setPosition(Vec2(x,y));

    // free memory
    sp->autorelease();

    // return it
    return sp;
}

Autorelease

- Places the object in a pool
- Marked for deletion later
- OS releases all in pool
- When is object deleted?
  - iOS: defined manually
  - Cocos2d: at end of frame?
- Must retain immediately
  - Otherwise, inter-frame obj

Delay release until later.
class Sprite : public Node, public TextureProtocol {
public:
    /** Creates a sprite with an image filename. */
    static Sprite* create(const string& filename);

    /** Creates a sprite with a Texture2D object. */
    static Sprite* createWithTexture(Texture2D *texture);

private:
    /** Creates, but does not initialize sprite */
    Sprite();

    /** Initializes a sprite with an image filename. */
    virtual bool initWithFile(const string& filename);

    /** Initializes a sprite with a texture. */
    virtual bool initWithTexture(Texture2D *texture);
};
class Sprite : public Node, public TextureProtocol {
public:
    /** Creates a sprite with an image filename. */
    static Sprite* create(const string& filename);

    /** Creates a sprite with a Texture2D object. */
    static Sprite* createWithTexture(Texture2D *texture);

private:
    /** Creates, but does not initialize sprite */
    Sprite();

    /** Initializes a sprite with an image filename. */
    virtual bool initWithFile(const string& filename);

    /** Initializes a sprite with a texture. */
    virtual bool initWithTexture(Texture2D *texture);
};
C++11 Analogue: Shared 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

```c++
Foo object = new Class();
shared_ptr<Foo> handle(object);
...
handle->foo();  //object->foo()
```
C++11 Analogue: Shared Pointers

```cpp
void foo() {
    shared_ptr<Thing> p1(new Thing);  // Allocate new object
    shared_ptr<Thing> p2 = p1;       // p1 and p2 share ownership
    shared_ptr<Thing> p3(new Thing); // Allocate another Thing

    p1 = find_some_thing();          // p1 might be new thing
    p3->defrangulate();              // call a member function
    cout << *p2 << endl;             // dereference pointer

    // "Free" the memory for pointer
    p1.reset();                      // decrement count, delete if last
    p2 = nullptr;                    // empty pointer and decrement
}
```
Where Does the Count Go?

Non-Intrusive Pointers

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

Intrusive Pointers

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

[Images courtesy of Kosmas Karadimitriou]
Where Does the Count Go?

Non-Intrusive Pointers

- Count inside smart pointer
- Advantage:
  - Easy to mix with raw pointers (and hence any stdlib code)
- Disadvantage:
  - Combining with raw pointers

Intrusive Pointers

- Count inside referred object
- Advantage:
  - Requires custom base object
- Disadvantage:

[Images courtesy of Kosmas Karadimitriou]
## References vs. Garbage Collectors

### Reference Counting

<table>
<thead>
<tr>
<th><strong>Advantages</strong></th>
<th><strong>Disadvantages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Deallocation is immediate</td>
<td>Overhead on every assignment</td>
</tr>
<tr>
<td>Works on non-memory objects</td>
<td><strong>Cannot easily handle cycles</strong></td>
</tr>
<tr>
<td>Ideal for real-time systems</td>
<td>(e.g. object points to itself)</td>
</tr>
<tr>
<td></td>
<td>Requires training to use</td>
</tr>
</tbody>
</table>

### Mark-and-Sweep

<table>
<thead>
<tr>
<th><strong>Advantages</strong></th>
<th><strong>Disadvantages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>No assignment overhead</td>
<td>Collection can be expensive</td>
</tr>
<tr>
<td>Can handle reference cycles</td>
<td>Hurts performance when runs</td>
</tr>
<tr>
<td>No specialized training to use</td>
<td>Usually triggered whenever the memory is close to full</td>
</tr>
</tbody>
</table>
Summary

- Must control **allocation** of heap objects
  - Preallocate objects when it makes sense
  - Use free-lists to recycle objects when possible

- Must track **ownership** of allocated objects
  - Know who is responsible for deleting
  - True even with Cocos2d reference counting
  - **Rule of Thumb**: Use getters to retain/release