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 byte
}
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

// 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);
}
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:**
  ```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**
C/C++: Allocation Process

**malloc**
- Based on memory size
  - Give it number of **bytes**
  - Typecast result before using it

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**: contents may vary
- **Heap**: fixed size for **n bytes**
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**
- **Can emulate malloc**
  - Create a char (byte) array
  - Arrays not initialized
  - Typecast after creation
- **Example:**
  ```c
  Point* p = (Point*)(new char[8])
  ```

Memory Details

Stack

Heap

n bytes
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 `alloc()` method in CUGL objects

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

Memory Details

[Diagram showing pre-allocated array with start, free, end labels]
class PolygonNode : public Node {
public:
    /** Creates, but does not initialize node */
    Sprite();

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

    /** Initializes a node with a texture. */
    virtual bool initWithTexture(const shared_ptr<Texture>& texture);

    /** Creates a node with an image filename. */
    static shared_ptr<Sprite> allocWithFile(const string& filename);

    /** Creates a node with a Texture object. */
    static shared_ptr<Sprite> allocWithTexture(const shared_ptr<Texture>& texture);
};
Separating Allocation and Initialization

Pre-allocated Array

- Array contains *objects*, not pointers
  - Allocating the array will allocate objects
  - No initialization until user grabs an object

- Makes it easy to implement factory pattern
  - Hides choice of underlying allocation
  - Uses standard initialization independent of allocation

These are not pointers!
class PolygonNode : public Node {
public:
    /** Creates, but does not initialize node */
    Sprite();

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

    /** Initializes a node with a texture. */
    virtual bool initWithTexture(const shared_ptr<Texture>& texture);

    /** Creates a node with an image filename. */
    static shared_ptr<Sprite> allocWithFile(const string& filename);

    /** Creates a node with a texture object. */
    static shared_ptr<Sprite> allocWithTexture(const shared_ptr<Texture>& texture);
};
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

```cpp
// 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(...)
```
Particle Pool Example

Memory Details

GL verts: 178
GL calls: 3
60.3 / 0.015
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

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();

};
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 re-use. */
    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

```c
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
A Question of Ownership

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;
}

Memory Details

Memory Leak

Not a Leak
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);
    table.put(key, o);
    o = null;
    return;
}
A Question of Ownership

Thread 1

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

Thread 2

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

“Owners” of obj

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
- 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>• Only use <strong>indirect</strong> references</td>
</tr>
<tr>
<td>• Can use reference <strong>directly</strong></td>
<td>• Copy to local variable first</td>
</tr>
<tr>
<td>• No need to copy reference</td>
<td>• Compute on local variable</td>
</tr>
<tr>
<td>• Treat like a normal object</td>
<td>• Be prepared for NULL</td>
</tr>
<tr>
<td>• Standard type of reference</td>
<td>• Reconstruct the object?</td>
</tr>
<tr>
<td></td>
<td>• Abort the computation?</td>
</tr>
</tbody>
</table>

### Memory Details

**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?
Reference Counting

• Every object has a **counter**
  • Tracks number of “owners”
  • No owners = memory leak
• Increment when assign reference
  • Historically an explicit method call
  • Method often called `retain()`
• Decrement when remove reference
  • Method call is `release()`
  • If makes count 0, delete it
# References vs. Garbage Collectors

## Reference Counting

<table>
<thead>
<tr>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deallocation is immediate</td>
</tr>
<tr>
<td>Works on non-memory objects</td>
</tr>
<tr>
<td>Ideal for real-time systems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead on every assignment</td>
</tr>
<tr>
<td><strong>Cannot easily handle cycles</strong> (e.g. object points to itself)</td>
</tr>
<tr>
<td>May require training to use</td>
</tr>
</tbody>
</table>

## Mark-and-Sweep

<table>
<thead>
<tr>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>No assignment overhead</td>
</tr>
<tr>
<td>Can handle reference cycles</td>
</tr>
<tr>
<td>No specialized training to use</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection can be expensive</td>
</tr>
<tr>
<td>Hurts performance when runs</td>
</tr>
<tr>
<td>Usually triggered whenever the memory is close to full</td>
</tr>
</tbody>
</table>
Smart Pointers

- `std::shared_ptr<T>`
  - Templatized type in C++11
  - Provides reference counting
  - Need to include `<memory>`

- Counting is automatic
  - Uses overloaded operators
  - Increments at creation/copy
  - Decrements when destroyed

- Smart pointer is on stack!
  - Passing as a parameter copies
  - So often pass by reference
Smart Pointers

- `std::shared_ptr<T>`
  - Templatized type in C++11
  - Provides reference counting
  - Need to include `<memory>`

- Counting is automatic
  - Uses overloaded operators
  - Increments at creation/copy
  - Decrements when destroyed

- Smart pointer is on stack!
  - Passing as a parameter copies
  - So often pass by reference

```cpp
#include <memory>
#include "A.h"
using namespace std;

/** Okay, but inefficient */
void foo(shared_ptr<A> var)
{
    ...
}

/** Preferred approach */
void foo(const shared_ptr<A>& var)
{
    ...
}
```
Smart Pointers

- `std::shared_ptr<T>`
  - Templatized type in C++11
  - Provides reference counting
  - Need to include `<memory>`

- Counting is automatic
  - Uses overloaded operators
  - Increments at creation/copy
  - Decrements when destroyed

- Smart pointer is on stack!
  - Passing as a parameter copies
  - So often pass by reference

```cpp
#include <memory>
#include "A.h"
using namespace std;

/** Okay, but inefficient */
void foo(shared_ptr<A> var)
{
    ...
}

/** Preferred approach */
void foo(const shared_ptr<A>& var)
{
    ...
}
```

- Smart pointers are on stack!
- Passing by reference copies
  - So often pass by reference
Smart Pointers

- `std::shared_ptr<T>`
  - Templatized type in C++11
  - Provides reference counting
  - Need to include `<memory>`
- Counting is automatic
  - Uses overloaded operators
  - Increases at creation/copy
  - Decreases when destroyed
- Smart pointer is on stack!
  - Passing as a parameter copies
  - So often pass by reference

```c++
#include <memory>
#include "A.h"

using namespace std;

/** Okay, but inefficient */
void foo(shared_ptr<A> var)
{
    ...
}

/** Preferred approach */
void foo(const shared_ptr<A>& var)
{
    foo claims ownership
    foo can change A, not smart pointer
    foo wants usage only
}
```
Smart Pointers

- `std::shared_ptr<T>`
  - Templatized type in C++11
  - Provides reference counting
  - Need to include `<memory>`

- Counting is automatic
  - Uses overloaded operators
  - Increments at creation/copy
  - Decrements when destroyed

- Smart pointer is on stack!
  - Passing as a parameter copies
  - So often pass by reference

```cpp
#include <memory>
#include "A.h"
using namespace std;

/** Good */
shared_ptr<A> foo(void) {
  shared_ptr<A> result ...
  return result;
}

/** BAD!!! */
shared_ptr<A> & foo(void) {
  shared_ptr<A> result ...
  return result;
}
```
**Smart Pointers**

- `std::shared_ptr<T>`
  - Templatized type in C++11
  - Provides reference counting
  - Need to include `<memory>`

- Counting is automatic
  - Uses overloaded operators
  - Increments at creation/copy
  - Decrements when destroyed

- Smart pointer is on stack!
  - Passing as a parameter copies
  - So often pass by reference

```cpp
#include <memory>
#include "A.h"
using namespace std;

/** Good */
shared_ptr<A> foo(void) {
  shared_ptr<A> result ...
  return result;
}

/** BAD!!! */
shared_ptr<A>& foo(void) {
  shared_ptr<A> result ...
  return result;
}
```

**Memory Details**

- Smart pointers are on the stack.
- Passing by reference results in a copy.
- Passing by value results in a pointer that is deleted when the stack frame is removed.
- Reference counting manages the lifetime of objects.

---

36 Memory Details
Recall: Smart Pointers and Allocation

Heap Allocation

```c++
void func() {
    Point* p = new Point(1,2,3);
    ...
    delete p;
}
```

- Must remember to delete
- Otherwise will **memory leak**

Smart Pointer

```c++
void func() {
    shared_ptr<Point> p;
    p = make_shared<Point>(1,2,3);
    ...
}
```

- Deletion is not necessary
- Sort-of garbage collection

Same arguments as constructor
class PolygonNode : public Node {
public:
    /** Creates, but does not initialize node */
    Sprite();

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

    /** Initializes a node with a texture. */
    virtual bool initWithTexture(const shared_ptr<Texture>& texture);

    /** Creates a node with an image filename. */
    static shared_ptr<Sprite> allocWithFile(const string& filename);

    /** Creates a node with a Texture object. */
    static shared_ptr<Sprite> allocWithTexture(const shared_ptr<Texture>& texture);
};

Smart pointer & initialization
**Recall: Reference Strength**

**Strong Reference**
- `shared_ptr<A>`
  - Always safe to use
  - Held until all deleted

**Weak Reference**
- `weak_ptr<A>`
  - Not always safe to use
  - Returns null if deleted

```
std::shared_ptr<T>

Pointer to T
Ptr to Control Block
```

```
std::shared_ptr<T>

Pointer to T
Ptr to Control Block
```

BAD!
Recall: Reference Strength

**Strong Reference**
- `shared_ptr<A>`
  - Always safe to use
  - Held until all deleted

**Weak Reference**
- `weak_ptr<A>`
  - Not always safe to use
  - Returns null if deleted
Weak Pointers vs. Raw Pointers

- Weak pointers are still managed!
  - weak_ptr<A> and A* not the same thing
  - Weak pointer has reference to the control block
  - Will null the base pointer when no longer valid

- But sometimes you want a raw pointer A*
  - May be required by 3rd party APIs
  - Can get this from strong, weak pointers: var.get()
  - Caching this value past a function call is unsafe
## Recall: Typecasting and Smart Pointers

### Normal Pointers

```cpp
B* b;  // The super class
A* a;  // The subclass
```

**Acceptable:**

```cpp
b = new B();
a = (A*)b;
```

**Better:**

```cpp
b = new B();
a = dynamic_cast<A*>(b);
```

### Smart Pointers

```cpp
shared_ptr<B> b;  // Contains B*
shared_ptr<A> a;  // Contains A*
```

**Bad:**

```cpp
b = make_shared<B>();
a = (shared_ptr<A>)b;
```

**Good:**

```cpp
b = make_shared<B>();
a = dynamic_pointer_cast<A>(b);
```
# Recall: Typecasting and Smart Pointers

### Normal Pointers

<table>
<thead>
<tr>
<th>B* b; // The super class</th>
<th>A* a; // The subclass</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Acceptable:</strong></td>
<td></td>
</tr>
<tr>
<td>b = new B();</td>
<td></td>
</tr>
<tr>
<td>a = (A*)b;</td>
<td></td>
</tr>
<tr>
<td><strong>Better:</strong></td>
<td></td>
</tr>
<tr>
<td>b = new B();</td>
<td></td>
</tr>
<tr>
<td>a = dynamic_cast&lt;A*&gt;(b);</td>
<td></td>
</tr>
</tbody>
</table>

### Smart Pointers

<table>
<thead>
<tr>
<th>shared_ptr&lt;B&gt; b; // Contains B*</th>
<th>shared_ptr&lt;A&gt; a; // Contains A*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bad:</strong></td>
<td></td>
</tr>
<tr>
<td>b = make_shared&lt;B&gt;();</td>
<td></td>
</tr>
<tr>
<td>a = (shared_ptr&lt;A&gt;)b;</td>
<td></td>
</tr>
<tr>
<td><strong>Good:</strong></td>
<td></td>
</tr>
<tr>
<td>b = make_shared&lt;B&gt;();</td>
<td>Must acquire control block!</td>
</tr>
<tr>
<td>a = dynamic_pointer_cast&lt;A&gt;(b);</td>
<td></td>
</tr>
</tbody>
</table>
Platform Specific Issues

- **Android**: JNI interface issues
  - May need to call Java method from C++
  - Doing so requires pointers/references to Java
  - This requires a special allocator/deallocator

- **Apple**: Reference counting issues
  - Objective C has its own reference counting
  - Works with normal raw pointers so easier to use
  - But this requires specialized compiler support
  - Obj-C++ does not enable this support!
Android: Calling Java from inside C++

// Retrieve the JNI environment.
JNIEnv* env = (JNIEnv*)SDL_AndroidGetJNIEnv();

// Retrieve the Java instance of the SDLActivity and get its class
jobject activity = (jobject)SDL_AndroidGetActivity();
jclass clazz(env->GetObjectClass(activity));

// Find the identifier of the method to call
jmethodID method_id;
method_id = env->GetMethodID(clazz, "myMethod", "()V");

// Effectively call the Java method
env->CallVoidMethod(activity, method_id);

// Clean up the local references.
env->DeleteLocalRef(activity);
env->DeleteLocalRef(clazz);

See SDL API for more info

Memory leak if forget this

Memory Details
Apple: Reference Issues in Obj-C++

// Note Objective separates allocation, initialization
// Reference var has reference count of 1
A* var = [[A alloc] init];

// Increments reference count to 2
[var retain];

// Decrements reference count to 1
[var release];

// Decrements reference count to 0 AND deletes
[var release];
Apple: Reference Issues in Obj-C++

// Note Objective separates allocation, initialization
// Reference var has reference count of 1
A* var = [[A alloc] init];

// Increments reference count to 2
[var retain];

// Decrements reference count to 1
[var release];

// Decrements reference count to 0 AND deletes
[var release];

Do this instead of free or delete
Summary

- Must control *allocation* of heap objects
  - Preallocate objects when it makes sense
  - Use free-lists to recycle objects when possible
  - Use the CUGL factory pattern to support these

- Must track *ownership* of allocated objects
  - Know who is responsible for deleting
  - True even when using smart pointers
  - Pay attention to platform specific issues