the gamedesigninitiative at cornell university

C++: Memory

Key Memory Issues for CUGL

- **Memory Size**
  - *Reinterpretting* data types
  - Performing *arithmetic* on pointers
- **Allocation and Deallocation**
  - Understanding the *basic syntax*
  - Understanding the *problems* and *challenges*
- **Modern C++ Features**
  - Understanding *shared pointers*
  - Understanding *memory pools*

Sizing Up Memory

Primitive Data Types

- **char**: 1 byte (8 bits)
- **bool**: 1 byte (*sorry*)
- **short**: 2 bytes
- **int**: 4 bytes
- **long**: 8 bytes

(int, long) Not standard
May change

- **float**: 4 bytes
- **double**: 8 bytes

(float, double) IEEE standard
Won't change

Complex Data Types

- **Pointer**: platform dependent
  - 4 bytes on 32 bit machine
  - 8 bytes on 64 bit machine
- **Array**: data size * length
  - Strings too (w/ trailing null)
- **Struct**: sum of fields
  - Same rule for classes
  - Struct = class w/o methods

Memory Example

`class Date {`	
`short year;`	2 byte
`char day;`	1 byte
`char 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

```
// 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;

// Use the new type
for(int ii = 0; ii < 2; ii++) {
  CULog("Point %4.2, %4.2",
        points[ii].x, points[ii].y);
}
```

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,

// Points for c
Vec2* points

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

// Use the new type
for(int ii = 0; ii < 2; ii++) {
  CULog("Point %4.2, %4.2",
            points[ii].x, points[ii].y);
}
```

This is safe.

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 c
Vec2* points

points =
  reinterpret_cast<Vec2*>(lineseg);

// Use the new type
for(int ii = 0; ii < 2; ii++) {
  CULog("Point %4.2, %4.2",
           points[ii].x, points[ii].y);
}
```

This is better!

Pointer Arithmetic

- sizeof(type) is size in bytes
  - sizeof(char) is 1
  - sizeof(float) is 4
- Pointer arith uses sizeof
  - Suppose p address is 4
  - p+1 is 5 if p is char*
  - p+1 is 8 if p is int*
- Why is this important?
  - Some funcs require char*
  - Reinterpret cast the pointer

```
int x;
int* array = new int[4];
char* ref = (char*)array;
// These are same
x = array[3];
x = *(array+3)
x = *((int*)(ref+3*sizeof(int))
// But these are NOT
x = *(ref+3*sizeof(int))
x = *((int*)(ref+3)
```

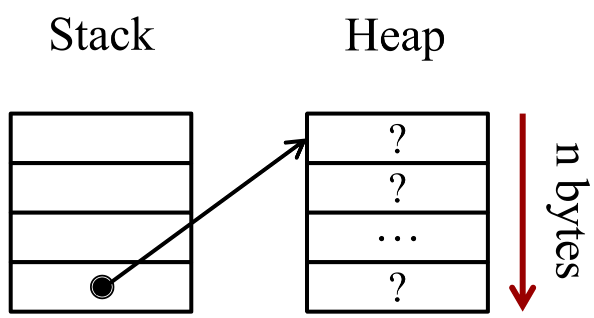
Key Memory Issues for CUGL

- **Memory size and alignment**
  - *Reinterpretting* data types
  - *Aligning* arrays of data
- **Allocation and Deallocation**
  - Understanding the *basic syntax*
  - Understanding the *problems* and *challenges*
- **Modern C++ Features**
  - Understanding *shared pointers*
  - Understanding *memory pools*

C/C++: Allocation Process

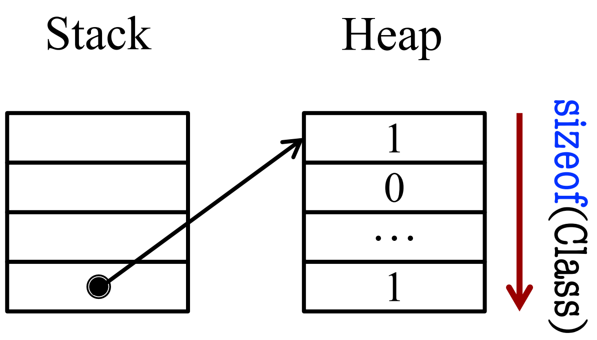
malloc

- 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

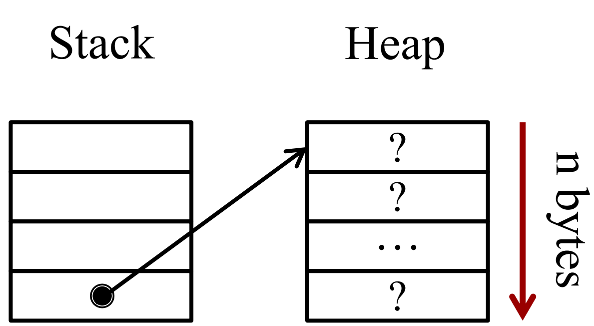
malloc

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

Preferred in C

- **E**…

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



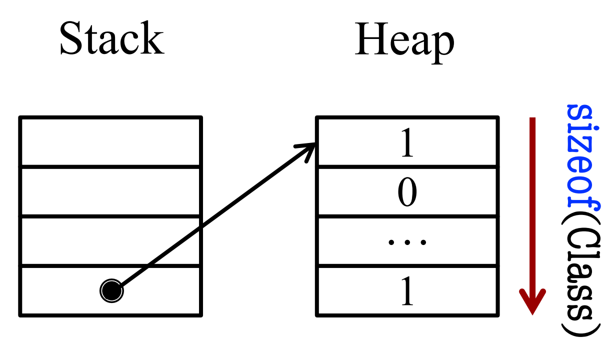
new

- Based on data type
  - Give it a data type
  - If a class, c… tor
  - … ion

Preferred in C++

- **E**…

Point* p = new Point();



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;
}
```

Recall: Allocation and Deallocation

Not An Array

- Basic format:

```
type* var = new type(params);
...
delete var;
```

- Example:
  - `int* x = new int(4);`
  - `Point* p = new Point(1,2,3);`
- One you use the most

Arrays

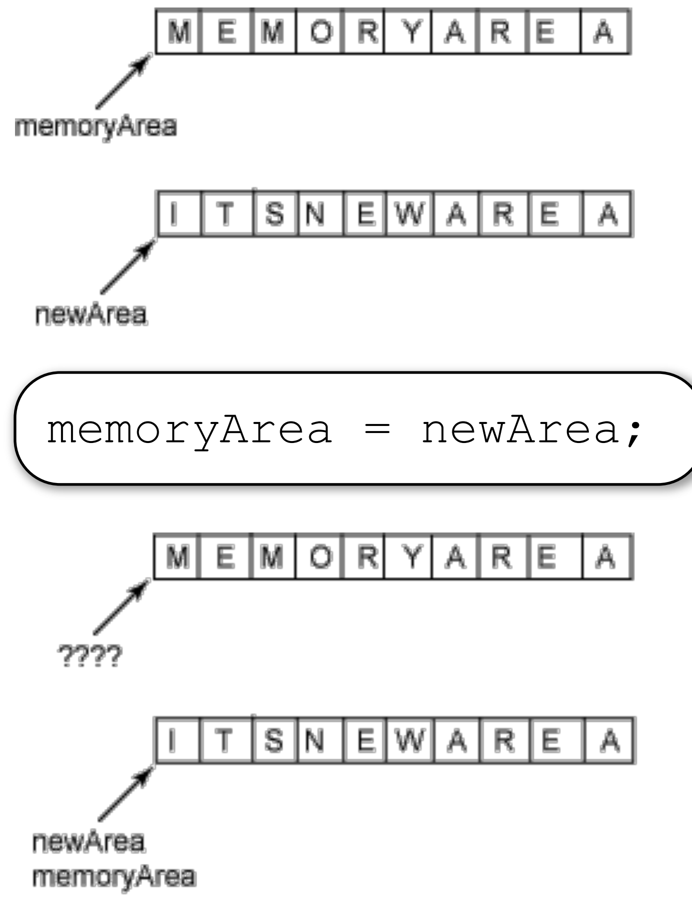
- Basic format:

```
type* var = new type[size];
...
delete[] var; // Different
```

- Example:
  - `int* array = new int[5];`
  - `Point* p = new Point[7];`
- Forget [] == memory leak

Memory Leaks

- **Leak**: Cannot release memory
  - Object allocated on heap
  - Only reference is moved
- Consumes memory fast!
  - Especially if inter-frame
- Can even happen in Java
  - JNI supports native libraries
  - Method may allocate memory
  - Need another method to free
  - **Exmp**: dispose() in LibGDX



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

```
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 *rarely transferred*
  - May pass to other functions
  - Part of the specification
- Really a **stack-based object**
  - Active as long as allocator is
  - So we can avoid the heap

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… *ed*
  - …specification

Easy: Will ignore

- Really a **stack-based object**
  - Active as long as allocator is
  - So we can avoid the heap

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"

Key Memory Issues for CUGL

- **Memory Size**
  - *Reinterpretting* data types
  - Performing *arithmetic* on pointers
- **Allocation and Deallocation**
  - Understanding the *basic syntax*
  - Understanding the *problems* and *challenges*
- **Modern C++ Features**
  - Understanding *shared pointers*
  - Understanding *memory pools*

Reference Strength

Strong Reference

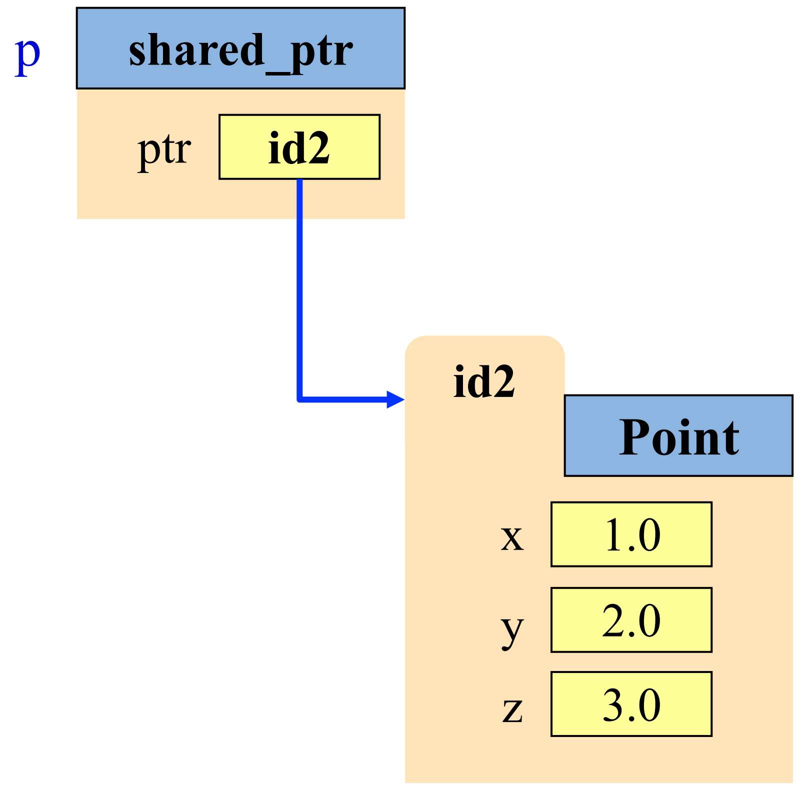
- Reference asserts ownership
  - Cannot delete referred object
  - Assign to NULL to release
  - Else assign to another object
- Can use reference **directly**
  - No need to copy reference
  - Treat like a normal object
- Standard type of reference

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?

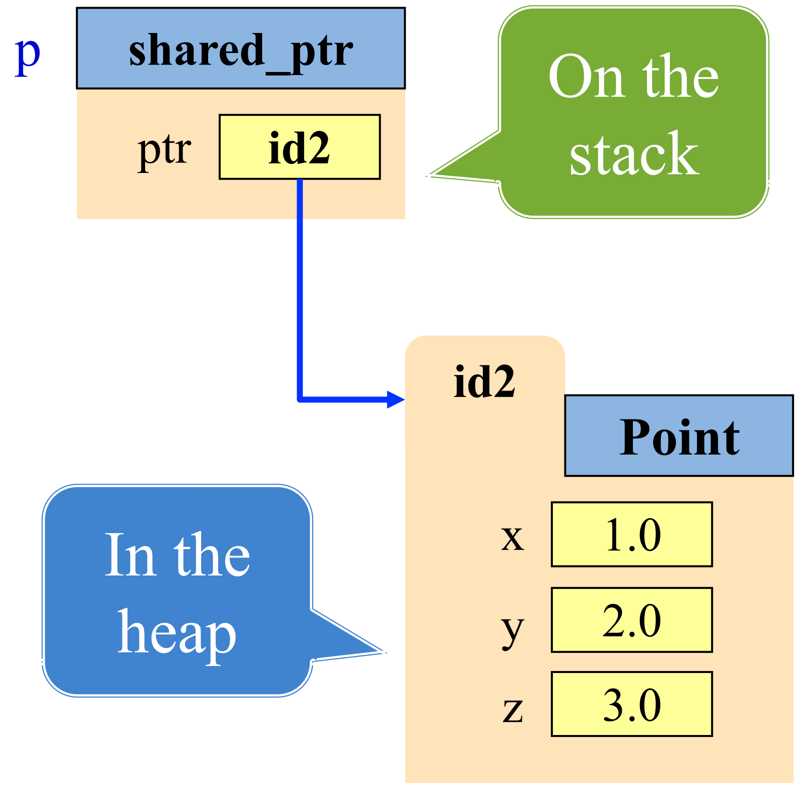
Recall: Shared Pointers (C++11)

- C++ can override **anything**
  - Assignment operator =
  - Dereference operator ->
- Class that *holds* a pointer
  - Tracks the pointer usage
  - Can delete pointer for you
  - Access pointer with `get()`
- Type is *templated* type
  - `std::shared_ptr<Point>`
  - `std::shared_ptr<Font>`



Recall: Shared Pointers (C++11)

- C++ can override **anything**
  - Assignment operator =
  - Dereference operator ->
- Class that *holds* a pointer
  - Tracks the pointer usage
  - Can delete pointer for you
  - Access pointer with get()
- Type is *templated* type
  - std::shared_ptr<Point>
  - std::shared_ptr<Font>



Shared Pointers in C++11

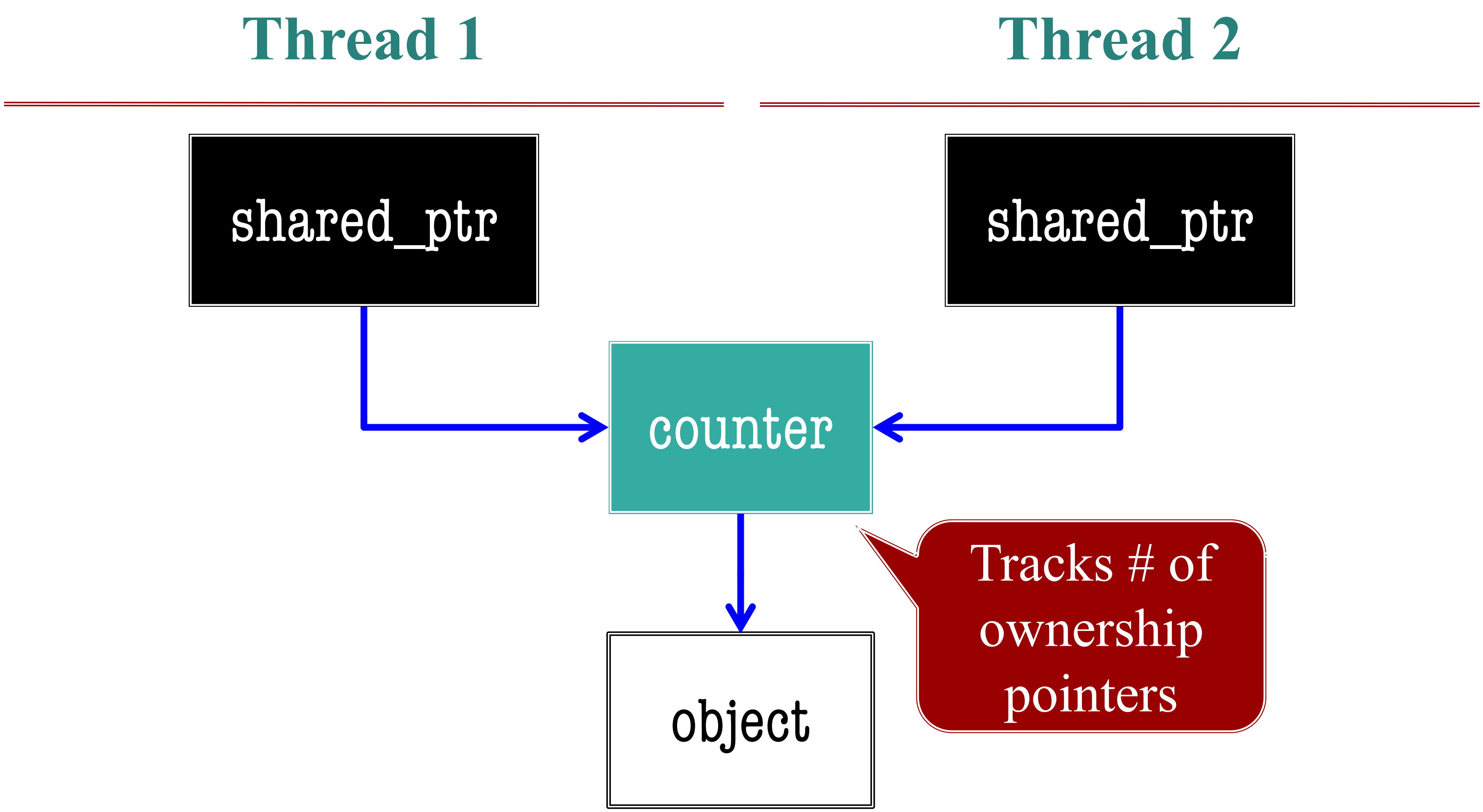
```
void foo() {
   shared_ptr<Thing> p1(new Thing());   // Allocate new object
   shared_ptr<Thing> p2=p1;             // p1 and p2 share ownership
   shared_ptr<Thing> p3 = make_shared<Thing>();   // Allocate another
   ...
   p1 = find_some_thing();   // p1 might be new thing
   p3->defrangulate();       // call a member function
   cout <<*p2 << endl;       // dereference pointer
   ...
   // "Free" the memory for pointer
   p1.reset();     // decrement reference, delete if last
   p2 = nullptr;   // empty pointer and decrement
}
```

Shared Pointers in C++11

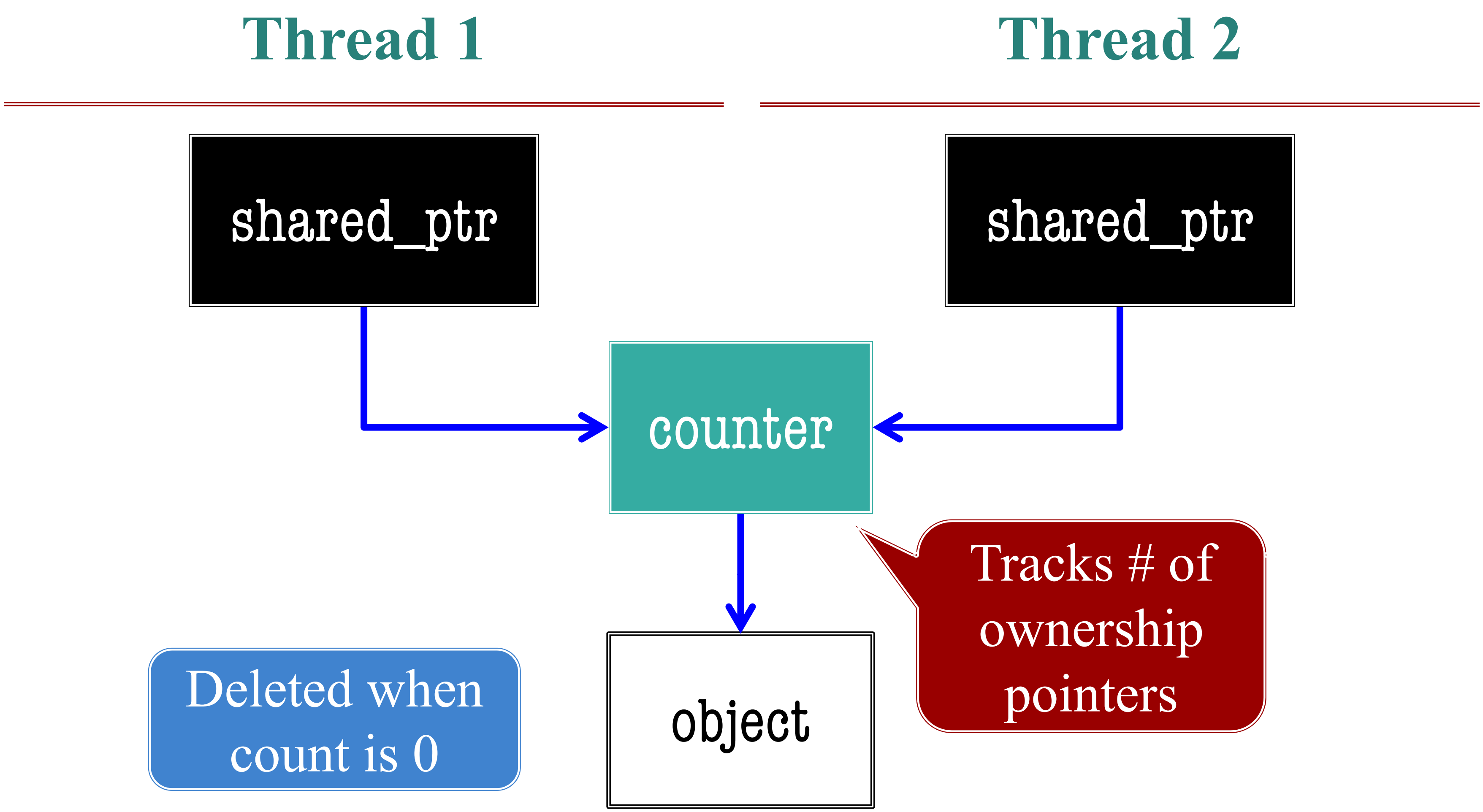
```
void foo() {
    shared_ptr<Thing> p1(new Thing());   // Allocate new object
    shared_ptr<Thing> p2=p1;             // p1 and p2 share ownership
    shared_ptr<Thing> p3 = make_shared<Thing>();   // Allocate another
    ...
    p1 = find_some_thing();   // p1 might be new thing
    p3->defrangulate();       // call a member function
    cout <<*p2 << endl;       // dereference pointer
    ...
    // "Free" the memory for pointer
    p1.reset();     // decrement reference, delete if last
    p2 = nullptr;   // empty pointer and decrement
}
```

All Deleted

Solving the Thread Problem



Solving the Thread Problem



Passing Shared Pointers

- Shared pointers are objs
  - They are **not** the pointer
  - They **contain** the pointer
- Copy increases reference
  - Want to avoid if possible
  - Reference shared pointer!
- But make reference `const`
  - Cannot modify *pointer*
  - Can still modify *object*

```
void foo(shared_ptr<A> a) {
    // Creates new reference to a
}
```

```
void foo(shared_ptr<A>& a) {
    // No new reference to a
    // But can modify pointer
}
```

```
void foo(const shared_ptr<A>& a){
    // The preferred solution
}
```

Shared Pointers in CUGL

```
class Texture : : public enable_shared_from_this<Texture> {
public:
    /** Creates a sprite with an image filename. */
    static shared_ptr<Texture> allocWithFile(const string& file);

    /** Creates a sprite with a Texture2D object. */
    static shared_ptr< Texture> allocWithData(const void *data, int w, int h);

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

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

    /** Initializes a sprite with a texture. */
    virtual bool initWithData(const void *data, int w, int h);
};
```

Allocation & initialization

Allocation only

Initialization only

Shared Pointers in CUGL

```
class Texture : : public enable_shared_from_this<Texture> {
public:
    /** Creates a sprite with an image filename. */
    static shared_ptr<Texture> … file);

    /** Creates a sprite with a …
    static shared_ptr< Texture> allocWithData(const void *data, int w, int h);

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

    /** Initializes a sprite with an image filename. */
    virtual bool initWithFile(const …

    /** Initializes a sprite with …
    virtual bool initWithData(const … data, int w, int h);
};
```

If going in heap

Allocation & initialization

Allocation only

If going on stack

Initialization only

Shared Pointers in CUGL

```
class Texture : : public enable_shared_from_this<Texture> {
public:
    /** Creates a sprite with an i
    static shared_ptr<Texture> al

    /** Creates a sprite with a Te
    static shared_ptr< Texture> a                              t w, int h);

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

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

    /** Initializes a sprite with a texture. */
    virtual bool initWithData(const void *data, int w, int h);
};
```

Allows object to turn this into shared_ptr

Reference Strength

Strong Reference

- Reference asserts ownership
  - Cannot delete referred object
  - Assign to N

Shared Pointers

- …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 Pointers in C++11

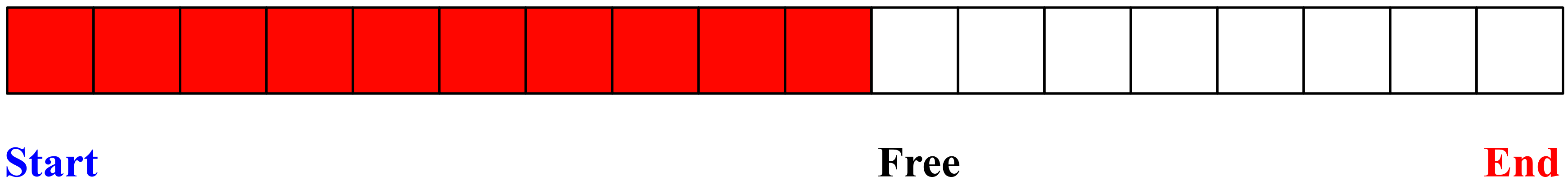
```
void foo() {
    shared_ptr<Thing> p1(new Thing);   // Allocate new object
    weak_ptr<Thing> p2=p1;              // p2 is a weak reference
    ...
    p1 = find_some_thing();   // p1 might be new thing
    auto p3 = p2.lock();          // Must lock p2 to dereference
    cout <<*p3 << endl;          // dereference pointer
    ...
    // "Free" the memory for pointer
    p1.reset();      // decrement reference, delete if last
    p2 = nullptr;   // empty pointer (but does not decrement)
}
```

Challenges of Shared/Weak Pointers

- Additional overhead acceptable, but significant
  - Updating references is not cheap
  - Two dereferences instead of one each time
- Ideal for **inter-frame** objects
  - Objects that persist for a long time
  - Smart pointers do not proliferate
- But what about **intra-frame** objects?
  - Have high churn (creation/deletion)
  - **Example:** particle systems

Custom Allocators

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



- **Idea**: Instead of `new`, get object from array
  - Cuts down on allocation mid-frame
  - Just reassign all of the fields
  - Use **Factory pattern** for constructor
- **Problem**: Running out of objects
  - We want to reuse the older objects
  - Easy if deletion is FIFO, but often isn't

Easy if only one object **type** to allocate

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(...)
```

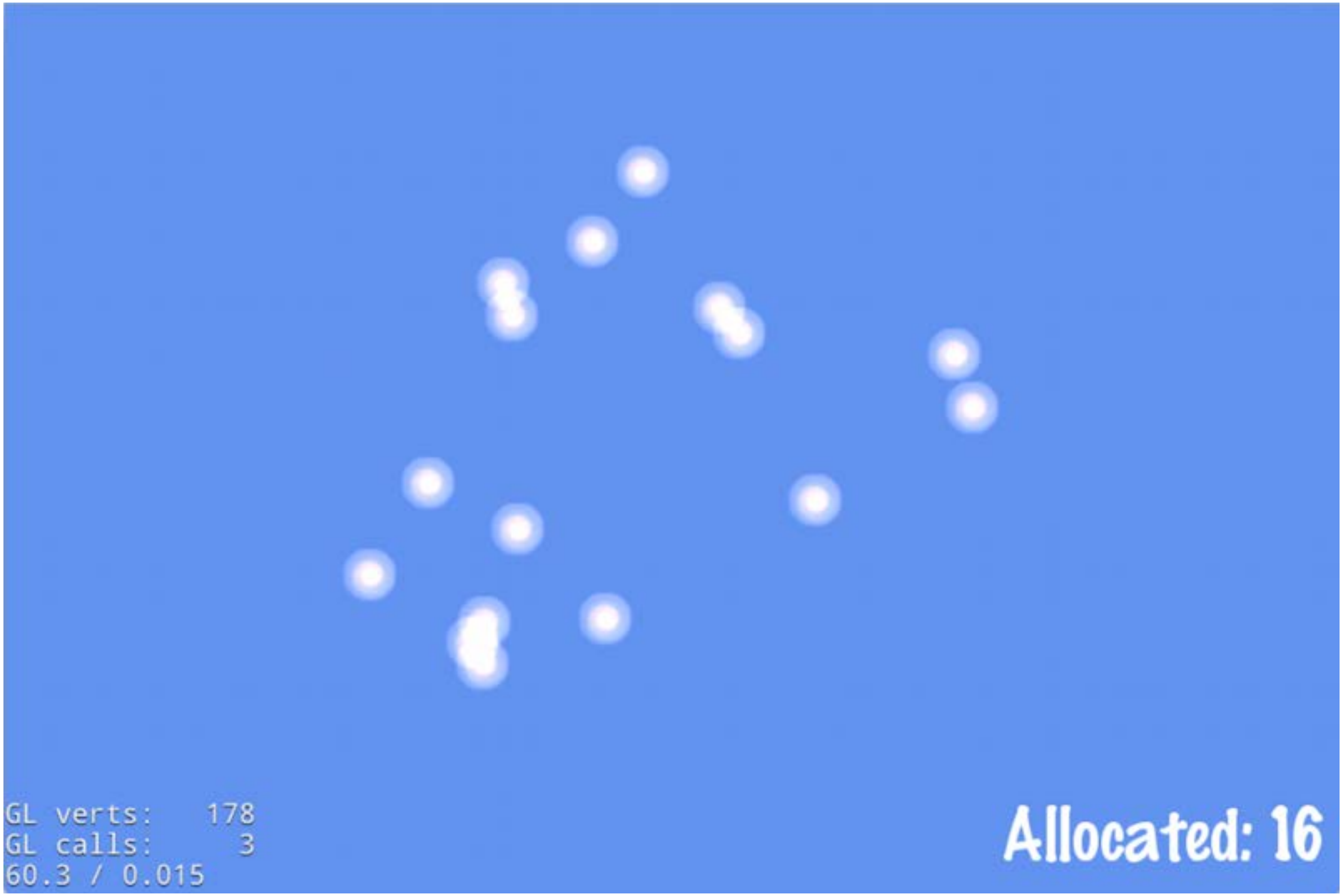
CUGL Support: FreeList

- Manages memory pool for “arbitrary” classes
  - Requires class have reset() method
  - Only supports default constructor
- **Example:**

```
FreeList<Thing> freelist;
freelist.init(CAPACITY);          // Creates obj array
Thing* t = freelist.malloc();     // Allocates object. MAY FAIL!
freelist.free(t)                  // Recycles object
```

- GreedyFreeList: malloc() is never null.

Particle Pool Example



Summary

- Pointer type-casting is very powerful
  - Allows you to impose structure on raw data
  - But requires you understand **memory sizes**
- Memory deallocation is very tricky
  - Must track **ownership** of allocated objects
  - The owner is responsible for deletion
- CUGL has some tools to make this simple
  - **Shared pointers** manage ownership issues
  - **Free lists** better for short-lived objects