the gamedesigninitiative
at cornell university

Lecture 13

Memory in C++

Sizing Up Memory

Primitive Data Types

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

Not standard
May change

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

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 same (w/ trailing null)
- **Struct**: sum of 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 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++) {
   CULog("Point %4.2, %4.2",
            points[ii].x, points[ii].y);
}
```

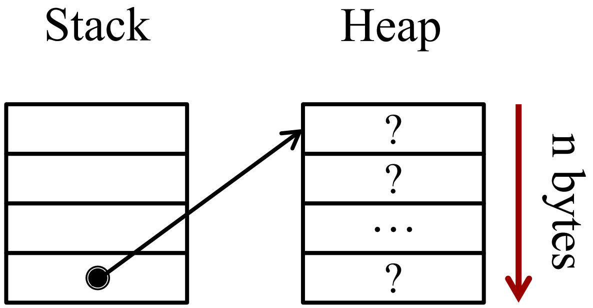
Two Main Concerns with Memory

- *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

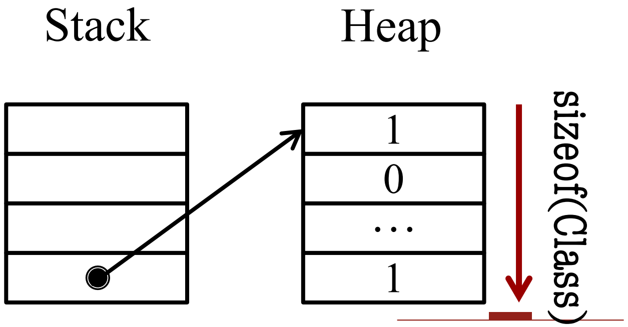
malloc

- 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();`



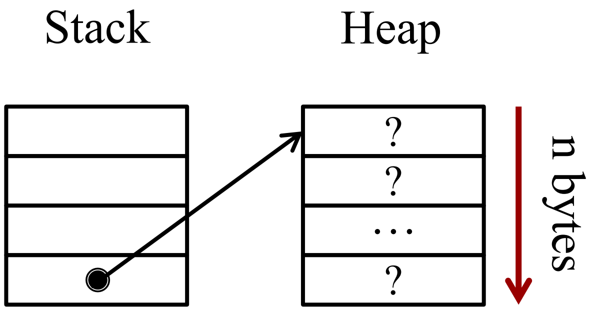
C/C++: Allocation Process

malloc

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

Preferred in C

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

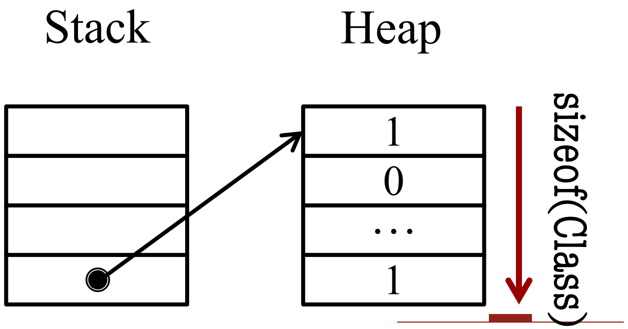


new

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

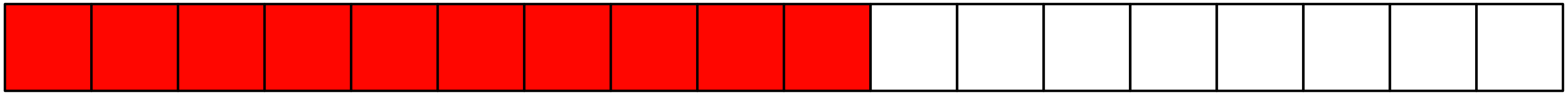
Preferred in C++

`Point* p = new Point();`



Custom Allocators

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



Start **Free** **End**

- **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

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

Custom Allocators 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

Custom Allocators 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);

    /** Creat                                  ct. */
    static shared_ptr< Texture> allocWithData(const void *data, int w, int h);

private:
    /** Creat                                   /
    Sprite();

    /** 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);
};
```

Customizable allocation

Standard allocation

Allocation & initialization

Allocation only

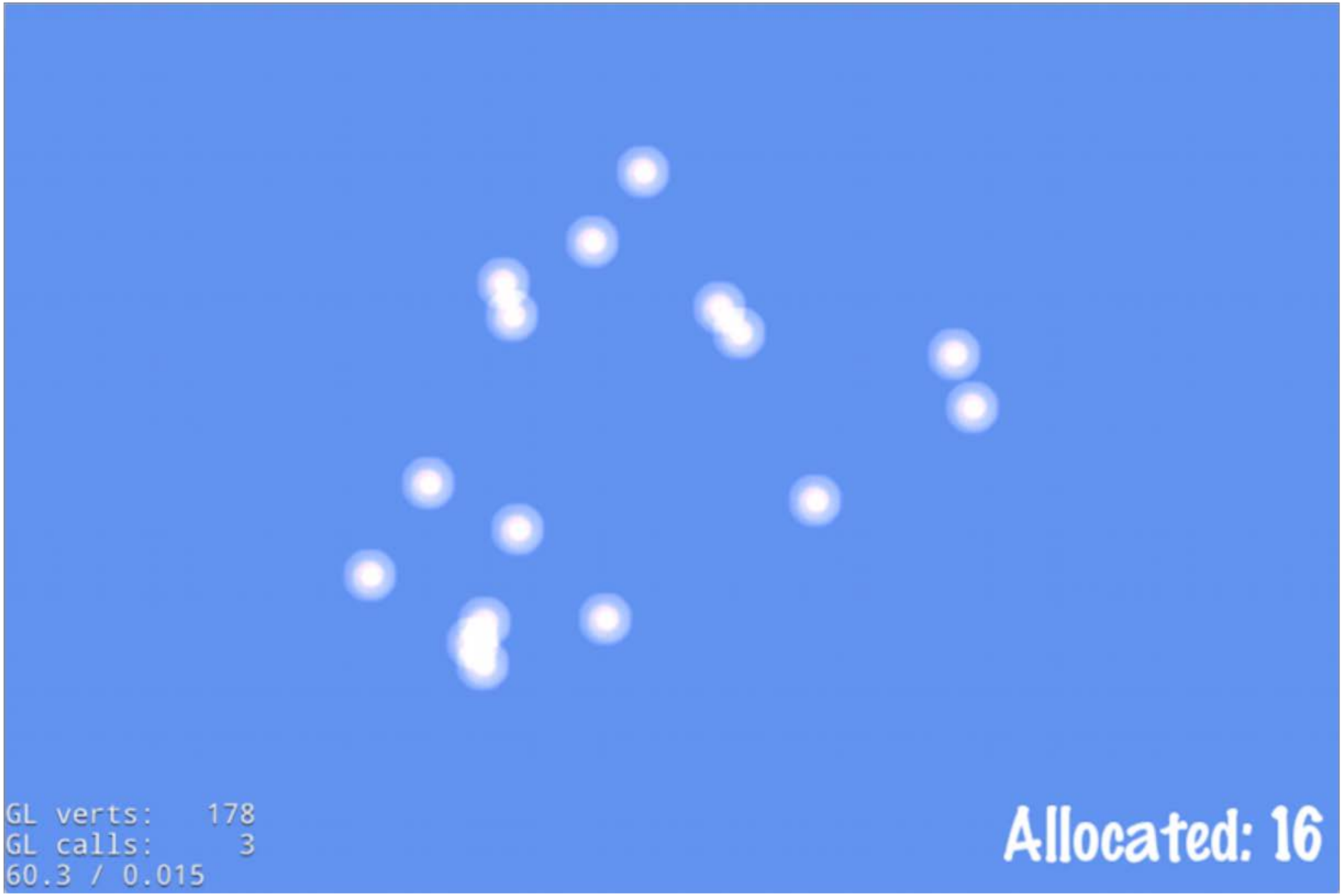
Initialization only

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(…)
```

Particle Pool Example



```
GL verts:    178
GL calls:      3
60.3 / 0.015
```

Allocated: 16

Particle Pool Example

```
GL verts:    178
GL calls:      3
60.3 / 0.015
```

Allocated: 16

See FreeList and GreedyFreeList

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;
}
```

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

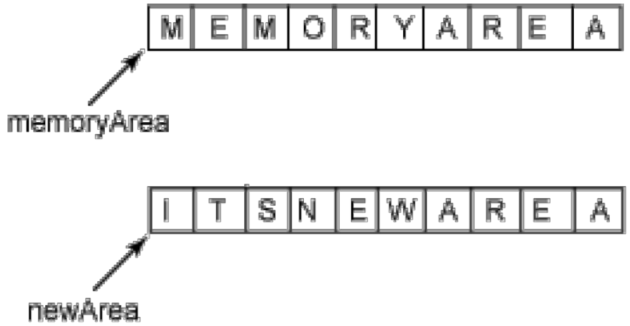
- Basic format:

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

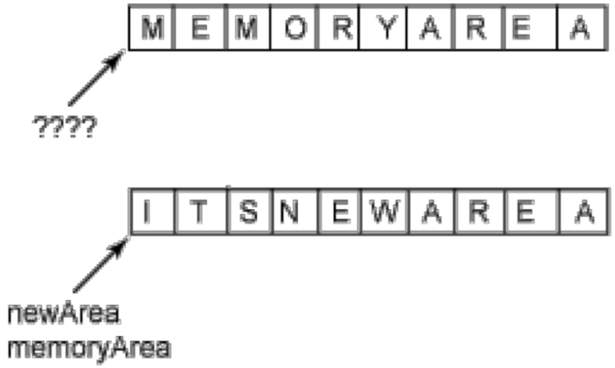
- Example:
  - `int* array = new int[5];`
  - `Point* p = new Point[7];`
- Forget [] == memory leak

Memory Leaks

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



```
memoryArea = newArea;
```



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

Easy: Will ignore

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

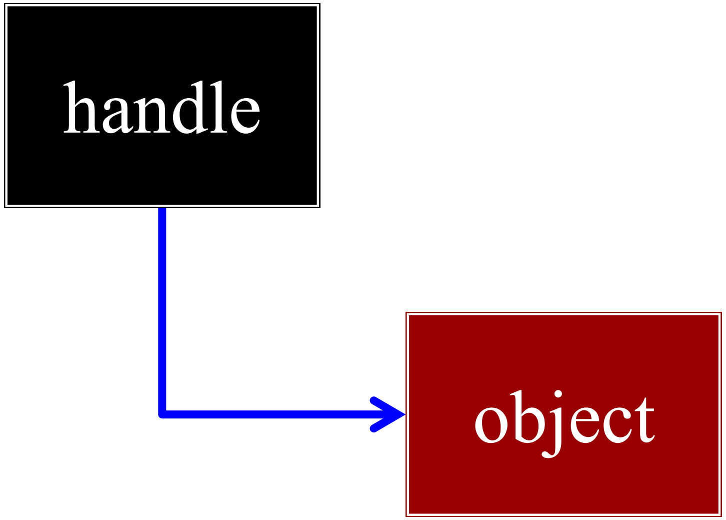
- Reference asserts ownership
  - Cannot delete referred object
  - Assign to NULL to release
  - Else assign to another object
- Can use reference **directly**
  - No need to copy reference
  - Treat like a normal object
- Standard type of reference

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?

C++11 Support: Shared Pointers

- C++ can override **anything**
  - Assignment operator =
  - Dereference operator ->
- Use special object as pointer
  - Has field to reference object
  - Tracks ownership of object
  - Uses *reference counting*
- What about deletion?
  - Smart pointer is on *stack*
  - Stack releases ownership



```
Foo* object = new Foo();
shared_ptr<Foo> handle(object);
...
handle->foo();   //object->foo()
```

C++11 Support: Shared Pointers

```
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 reference, delete if last
    p2 = nullptr;   // empty pointer and decrement
}
```

C++11 Support: **Weak** Pointers

```
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)
}
```

Passing Smart Pointers

- Shared pointers are objs
  - They are not the pointer
  - They contain the pointer
- Copy increases reference
  - What to avoid if possible
  - So reference smart pointer
- But make reference `const`
  - Keep from modifying ptr
  - 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
}
```

Summary

- Memory usage is always an issue in games
  - Uncompressed images are quite large
  - Particularly a problem on mobile devices
- Limit **allocations** in your animation frames
  - **Intra-frame** objects: **cached objects**
  - **Inter-frame** objects: **free lists**
- Must track **ownership** of allocated objects
  - The owner is responsible for deletion
  - C++11 **smart pointers** can manage this for us