Chapter 3.5
Memory and I/O Systems

Memory Management

- Only applies to languages with explicit memory management (C, C++)
- Memory problems are one of the leading causes of bugs in programs
  - Leaks
  - Buffer Overflows (Improper Access)
  - Fragmentation

Memory Management

We have 3 goals when working with memory:

1. Safety
   – Find fix leaks
   – Protect data
2. Knowledge
   – Who, what, where, how much
3. Control
   – Location → Caching → Speed

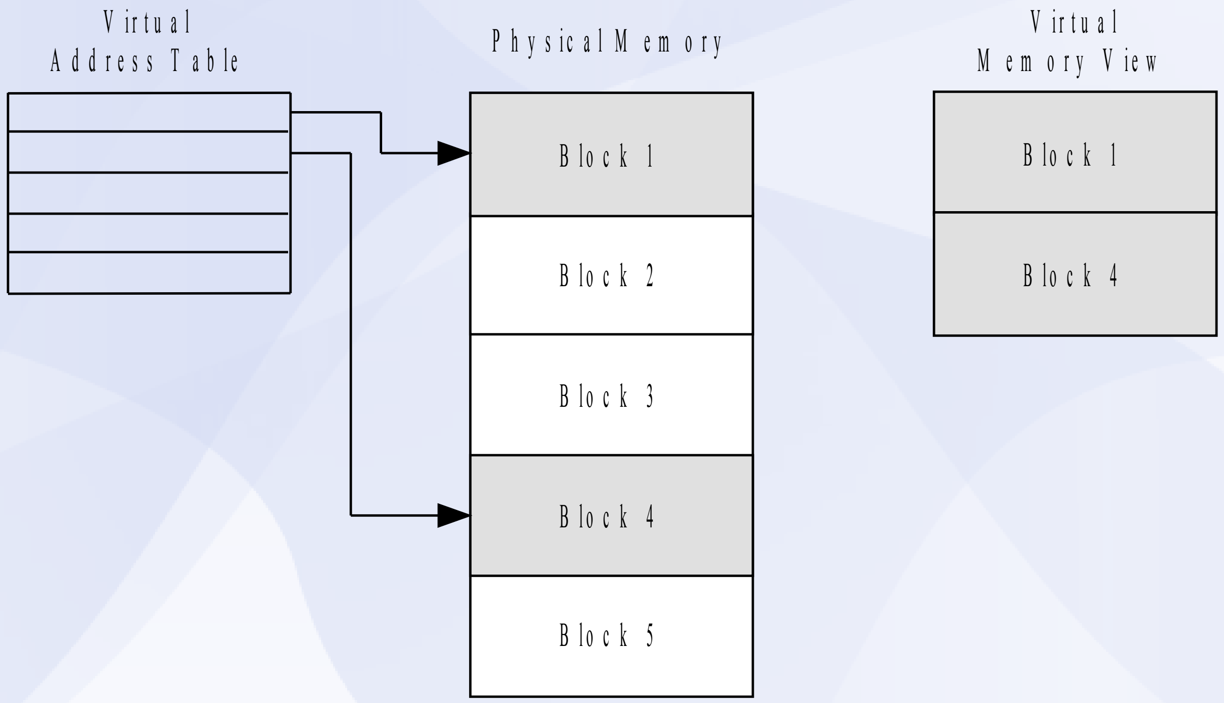
Memory Fragmentation

- Free physical memory is only available in small sections
- Can prevent an allocation from completing successfully even if there's plenty of free memory
- Virtual addressing will greatly help with fragmentation problems

Virtual Addressing

- Virtual addressing

"Lying to your applications since 1961"



Static Allocation

Static memory allocation: If all memory is statically allocated at start there will be very few problems:

- No - Leaks (no malloc, new, delete, free)
- No - Fragmentation
- No - Running out of memory

Has disadvantages though:

- Very restrictive (everything predeclared)
- Lots of wasted memory
- Old games used to be done this way

Static Allocation Example

```
// Create a fixed number of pathnodes
#define MAX_PATHNODES     4096
AIPathNode max_PathNodes[MAX_PATHNODES];

// Create a fixed size butter for Geometry calculations
// 8MB in size
#define GEOMSIZE  (8*1024*1024)
byte* s_GeomBuffer[GEOMSIZE ];
```

Dynamic Allocation

Dynamic allocation is much more flexible than static allocations:

- But has lots of potential problems
  - Leaks,
  - Miss-allocations
  - Tracking difficulties
- Need to override/replace new and delete to take control over allocations

Key Concept #1

1. "new" is actually broken down by the compiler into many sub-steps.

```
MyClass *data = new MyClass();

// Equivalent too
MyClass *data = malloc(sizeof(MyClass));
*data -> MyClass();

// malloc() in turn is broken down into several calls.
```

Key Concept #2

2. “new” and “delete” are just operators (like +, <, =) and operators can be changed and overloaded.

```
CVector::CVector (int a, int b) {  x = a;   y = b; }

CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}
```

Key Concept #3

3. We can overload new (and delete) so that they create and manage receipts (allocation headers) for all dynamically created objects.

```
void * operator new (size_t size, Heap *pHeap);

Struct AllocHeader {
    int nSignature, nAllocnum, nSize;
    Heap *pHeap;
    AllocHeader * pNext;
    AllocHeader * pPrev;
}
```

Key Concept #4

3. We can overload new (and delete) so that return/release a pointer to an existing (statically defined) memory location [using Alloc()], rather then using malloc to create a new one.

Class MemoryPool {

…

*Void * Alloc (size_t nSize)*

Memory Manager

- Heaps are collections of data sorted by size.
- We can use a heap to monitor memory allocations
- We will need to override operator new and delete
- We can add some simple error-checking schemes to prevent memory overrun problems

Memory Management

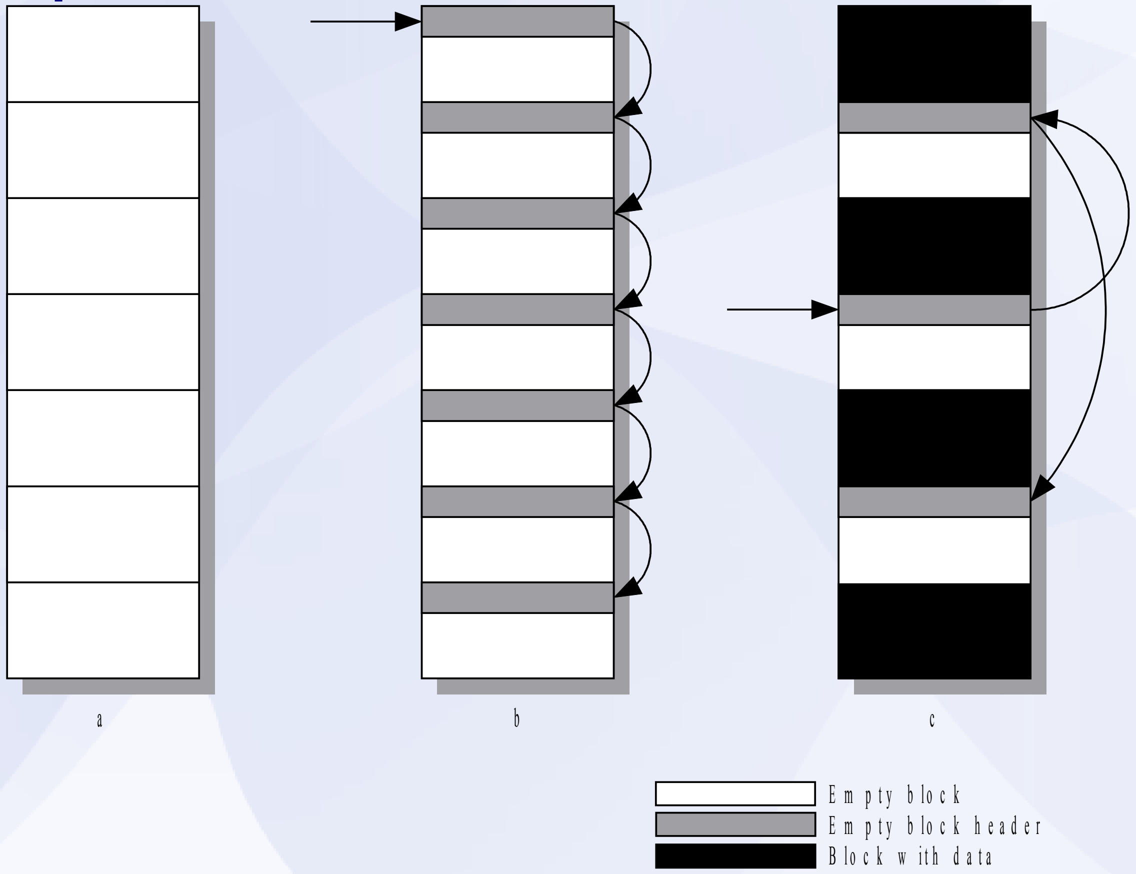
- Memory leaks
  - A memory leak is a memory allocation that is no longer necessary and was not released by the program
  - Memory leaks can cause the game to use up resources and run out of memory
  - To find all memory leaks, we look for all the allocations that occurred between two particular points in time

Memory Management

- Memory pools
  - Memory pools are contiguous blocks of memory used to allocate objects
  - Allocations are extremely fast
  - There is no fragmentation
  - They will usually have some unused space
  - We can allocate objects from specific classes with a simple macro

Memory Management

Memory pools



File I/O

Sometimes full file I/O not available on all platforms

- Different ways of accessing different devices
- Different speeds of access
    - Memory cards, network, etc

Usually have no control over physical disk allocation

- Which can lead to long load times

Ultimate goal is too simplify and speed-up File I/O

File I/O

Desire unified file system

- Platform independent
- Provide same interface to different types of media
- Based around FileSystem class
- Allow us to “buffer data” as desired
    - Including moving data to main memory

File I/O

File system class

- Uses concept of streams
- Stream can be the source of data

Possible sources

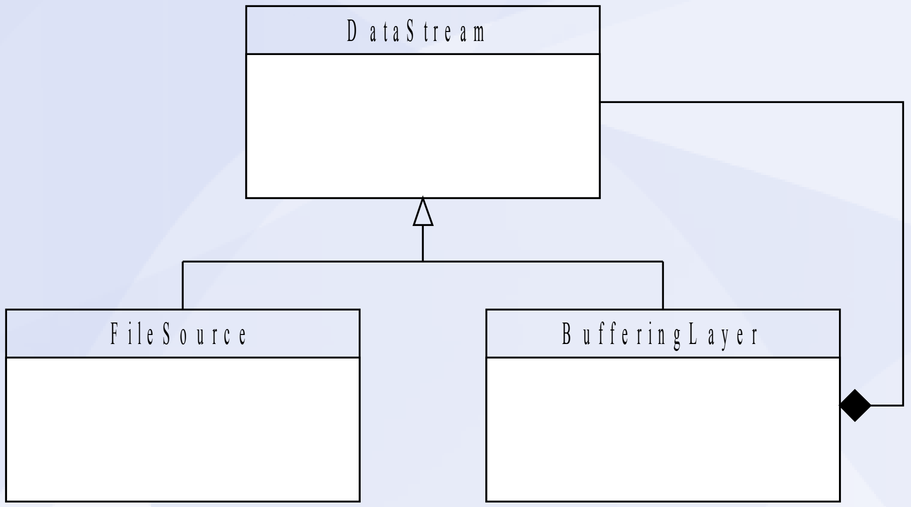
- File, memory, network
- Or it can be a layer that adds some properties

Possible layers

- Buffering, compression, range, etc

File I/O

File system



Pack Files

- Fast Processor & Slow File I/O
  - Perfect time to use compression
  - Cab, zip, rar or custom.
- Pack files can yield caching advantages
  - Grouping related data
  - Improve cached hit if large sections buffered

Game Resources

- A game resource (or asset) is anything that gets loaded that could be shared by several parts of the game
  - A texture, an animation, a sound, etc
- We want to load and share resources quickly and easily avoid File I/O
- There will be many different types of resources in a game

Game Resources

- Resource manager
  - Uses registering object factory pattern
  - Can register different types of resources
  - All resource creation goes through the resource manager
  - Any requests for existing resources don't load it again
  - Pre-caching possible/desirable

Game Resources

- Resource lifetime
  - If resources are shared, how do we know when we can destroy them?
    - All at once
    - At the end of the level
  - Explicit lifetime management
  - Reference counting
  - Smart Pointers

Game Resources

- Resources and instances
  - Resource is the part of the asset that can be shared among all parts of the game
  - Instance is the unique data that each part (object) of the game needs to keep
  - Pointers can be used to keep instance data that references shared resource data

Serialization

- Every game needs to save and restore some game state
  - Even for check point saves
- Level editing and creation could be implemented as a saved game
- How do we effectively store state information for objects?

Serialization

- Saving
  - ISerializable interface (pure virtual functions) for Read and Write operations
  - Each class implements the Write() function
  - Saving pointers is a problem
    - They won't be the same when we load
  - Save raw pointers to other objects
    - Translate them later
  - Save UID information (always valid)

Serialization

- Loading
  - Create object types through an object factory
  - Read() function takes care of loading the stored data from the stream
  - Pointers are then loaded into a translation table
  - Second pass fixes up all the pointers

The end