Data Structures with C++ : CS189

Lecture 14-1: Memory Management

Details about Memory

Recap - Dynamic Memory

- All of your variables and all of your code are combined to make the exe in stack memory
  - The exe can't change size so dynamic memory comes from the "heap"
- A pointer to a spot in the heap could point at an object, deleted memory, or null
  - The only way to tell is to use nullptr without fail
- Allocating memory and then losing the pointer to it is a memory leak
  - 179's smart pointers fight that
  - Otherwise, anything you new you must delete

Recap - Call Stack

- Whenever you call a function, a small amount of memory is used to keep track of where you were, and all the variables in the new function
    - Doesn't actually make a copy of the assembly language out of the exe
- This call "stack" is a fixed amount of memory allocated at program start
    - I've seen 1 MB as the standard size, not fixed
- Since the call stack is pre-allocated, calling a function just moves a pointer along that block

Stack Overflow

- The block of memory holding a function's local variables and a pointer to the caller is a "stack frame"
- We've seen stack overflow caused by infinite recursion
  - Every stack frame takes up space, so eventually you burn through the 1MB Stack allocation
- Since local variables (like an array of size ten million) are in the stack frame, calling what is effectively a very large function will also overflow the stack

The "this" Pointer and Move

Hey, add file analogy for pointers in to heap

- When in any method of an object, "this" points at the address the object starts at
  - Could be to the stack or heap, as above
- (*this) is then actually the address the memory for the object starts
  - Think of "this" as "start of object"
- It follows that like any other variable, that memory could be stack or heap
- So you know how much and where the memory is
- And that's why "Move" works
  - = would have taken a different chunk of memory and copied all the fields over

Arrays and Placement new

```
struct Rock { int var; };
int "main"() {
int mem[3] = {1, 2, 3};

Rock* X = new(mem)
Rock;
Rock* Y = new(mem + 1)
Rock;

cout << X->var << endl;
cout << Y->var << endl;
}
```

- Whether stack or heap, an array is a single block
  - Size being just Count * Size of one
- So you aren't actually allocating 5 ints, you are allocating one block the size of 5 ints with no inherent meaning
- You can tell the 'new' command where you want an object to be
- Combine those two ideas and you are controlling your own memory
- DO NOT delete those rocks. No new allocation took place for them

Note on Polymorphism

- A Destructor must be virtual or else if you delete using a baseclass pointer you only delete that part
    - Famous bugfix of the last memory leak in game
    - More importantly, delete is not virtual
- sizeof is a command that tells you the size of a variable and is also not virtual
    - Ask for sizeof basclass and you get only the size of that part
- If you find you need to know the size of a Dog, you'd need a virtual "int GetSize()" on Animal and Dog

Object Creation

Why Memory Management

- The goal is to speed up new/delete calls, and to prevent new/delete from breaking memory up into little pieces
- For speeding up, you can't do better than no-time-at-all, so making your objects at program launch is a great idea
- For "fragmentation" of memory, keeping track of what's bein used and giving out the most appropriately sized piece is where we're going.

Memory Pool for Objects

```
RockPool {
vector<Rock> mRocks;
vector <bool> mInUse;
};

Rock *X =
TheRockPool.GetRock();

RockPool.ReturnRock(X);
```

- A program might have a particular type of object that is used very dynamically
  - Particle effects or bullets are created and deleted constantly. Sometimes many every second
- So let's premake a bunch of that particular class, and just keep reusing them
- Just need to track which ones are "in use"
- Caller must always go through the pool for new and delete

Optimize Memory Pool

- Since this is 189, we know about Big O
- That solution above is O(n) because we might have to check every Rock to find one not in use
- How about if someone asks for a Rock and we don't have any free, we make one, and when they return it we put it in a queue of free Rocks
  - Fixes previous over-allocation
  - Causes stutter the first time something happens
- Correct pick depends on program
  - This is indeed 189

Abstract Memory Pool

```
char *where =
ThePool.Request(
sizeof(Rock));

Rock *myRock = new
(where) Rock;
```

- Instead of only tracking the clases we specifically want to pool, we could pre-allocate blocks of arbitrary size
- When someone wanted memory they would ask the pool using sizeof to tell it how much was needed
- The pool would return a pointer to the block, and the caller would be the one to use placement-new to put it there
  - Good cohesion. Pool knows pool code only

Managing the Heap

Fragmentation

```
Starts empty
------------------------------
Fill with small items
++++++++++++++++++
Remove every other
-+-+-+-+-+-+-+-+-+-+-+-
Try to add one that should
fit, but can't find a spot
++++++!
```

- Whether you are using memory pools or one giant block of a heap after a time the performance will degrade
  - I'm not talking about the normal OS heap since we can't control that. Heap here means the program allocated 4MB on load
- Picture a 10KB heap, then allocate 10 1KB objects, then remove every other one.
- Trying to ask for a 4KB object next would fail. We have the space overall, but no one spot big enough

Pointer Management

- We can't compress all the used memory spaces to the left unless our system is in charge of every pointer, star, and arrow
  - If you had a pointer to memory and I moved the memory without telling you you'd crash
  - Every pointer call would have to go through the system to check for redirects
  - Like Java
- But we can work to try to keep fragmentation from consolidating our free space

Sequential Fit

10 Spots used

++++++++++

2 in middle freed

++++--++++

Where do we put the next 1?

++++--++++?

++++?-++++

- Say you have made 10KB worth of allocations, and then free the middle 2KB
- If someone asked for something 1KB or smaller, it would be better if we could put it in that hole instead of continuing from the end of the 5
- Sequential fit means instead of keeping a write pointer, we scan from the left and return the first block they fit in

How

```
struct MemNode {
	int mStartAddress;
	int mSize;
	bool mFree
};

class Memory {
list<MemNode> mNodes;
char *Request(int tSize);
};
```

- Start with one giant node for the whole heap marked free that tracks its own size and start position
    - The size will change as we break pieces of it
    - Each node can go back and forth from used and not
- Scan from the left. If we are looking at a free node big enough to hold us, insert a new busy node of the requested size and push up the start address of the free node by that amount

Optimize Sequential Fit

- This delays fragmentation, but after a while every single node near the front is going to be chopped to pieces because we're doing first-fit we find
- So add to the algorithm: If we are looking at a free MemNode, if the next node is also free, combine them and start over
    - Combining would just be adding their size to yours and deleting them

Sequential Fit Variations

- After running your program a few times and analyzing the end result of your heap you might consider a different choice than First-Fit
  - Maybe you have only big requests or only small requests or some other pattern
- Best-Fit: Scan all the nodes once and then pick the one closest to your size
- Next-Fit: Like First, but start at the last insert instead of the beginning
- Worst-Fit: Opposite of Best-Fit
- Last-Fit: Scan from the end, but still bite off from the left

Non-Sequential Buddy System

0: List of 1024s
1: List of 512
2: List of 256
3: List of 128
.
.

Called "Left-fit" as in the vector is the left-most branch and each list is a different right branch.

- Picture this like a demented tree
- Start with one giant block and pick a minimum size (power of 2)
- Build a vector of lists where each vector index is a block size, and inside each list is all the blocks of that size
- When someone asks for a block, if you don't have the closest size reach up to next highest size you can find, break it in half, and move it down one space
- Now finding a block of some size is log n instead of n

Garbage Collection

Smart Pointers

This topic is an entire week of 179 so it won't fit here

- When memory is allocated but there is no longer any pointers to it that is a memory leak
  - OS thinks it is yours, but you don't know where it is
- Garbage Collection means finding all the memory leaks and freeing them
- A Smart Pointer is one method - it is a pointer that shares a reference count of how many smart pointers there are for the object
- When the last one lets go, the object is deleted

Garbage Collection

- True garbage collection would be a system that finds and frees lost objects with no extra work
- In C++ it is darn near impossible because we have pointers.
- A system that moved memory around to stop fragmentation would invalidate every pointer
- To make that work, every single pointer would have to secretly be a pointer to a table that redirects it to the real address
    - Like Java