CS 213 -- Lecture #23

"Late Night Guide to C++"
Chapter 12 pg. 330 - 338

SPACE ALLOCATION

Administrative...

- Remember, second prelim on 11/23!
  - In class, closed book, similar to prelim #1
  - Will cover entire semester, main focus on second half of year.
  - Slightly more "coding"
- Last two lecture will be "fun" lectures:
  - I will take attendance :-)
  - Official course eval
  - Fun and games

Memory Allocation

- No matter what system or platform you are programming for, there is one constant…
- Memory allocation is slow!
- The general problem is that no matter how good the generic memory management algorithm is behind the `new` operator, it is burdened by the fact that it must be prepared to allocate and manage blocks of memory which are of varying sizes.
- Now, please keep in mind that *slow* is a *relative term*. When looked at individually, even the slowest memory allocation is still fairly fast under a modern OS and/or modern hardware.
- Under MacOS, memory management is particularly slow due to there being an extra layer of indirection present (handles).
- So, what a perfect platform to talk about memory management on!

Demonstration 1

Lots o' Dynamic Allocation

Memory Management (cont)

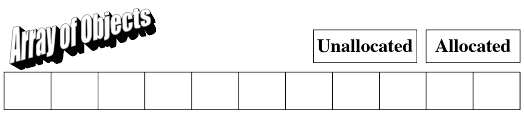
- As you can see, even though memory allocation is slow, it still takes *a lot* of memory allocations to slow us down.
- But, in the "real world", you'd be surprised at how quickly repeated memory allocation can slow down your program.
- Acknowledging this, C++ lets you do something truly bizarre…
- You can overload the `new` operator and define your own memory management routines.
- Just *try* doing that in Java!!!!
- Of course, if you can overload `new` you must be able to overload delete as well.
- Overloading the `new` and `delete` operators is done just like any other operator overload, except there is a slightly special syntax:
  - `void *operator new(size_t);`
  - `void operator delete(void *);`

Memory Management (cont)

- OK, but that means I actually have to *implement* some memory management routines!
- *WHY* would I want to do that?
- Because the generic memory management routines are slow!
- They're slow because they have to be able to deal with any size object.
- If you know you will need to dynamically allocate a lot of the same objects during the course of your program you can define your own memory management routines for allocations of that object type.
- A typical memory management routine to handle repeated allocations of the same type can be implemented in a way that will be more efficient than the generic routine.
- Let's do it!

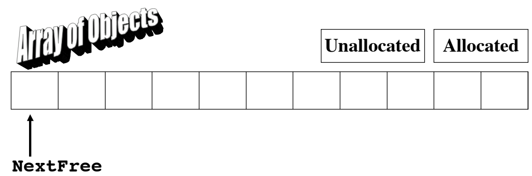
Memory Management (cont)

- The idea is somewhat simple, when you think about it.
- You allocate a block of memory (using the generic routine) to form an "array" of the objects in question.
- When `YourObject::operator new` is called, you "give out" one of your previously allocated objects.
- That means you need to keep track of what's allocated and what is not!
- When you start out, you'll have something like this:



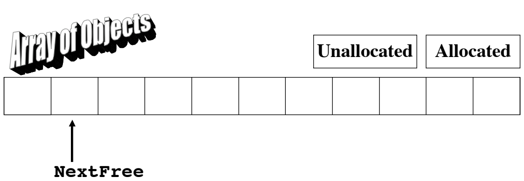
Memory Management (cont)

- But how do you "dole out" pre-allocated objects in an orderly fashion?
- First, you'll need a pointer variable which will be used to point to consecutive items in your array. We'll call it `NextFree`.
- It will, of course, start at the beginning of the array.



Memory Management (cont)

- When one object is allocated via a call to `YourObject::operator new`, we'll return the object that `NextFree` is pointing at and then increment `NextFree` to the next item in the array.
- That would look like this:



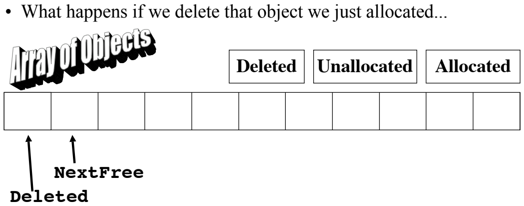
Memory Management (cont)

- But what happens when `YourObject::operator delete` is called?
- We need to keep track of deleted objects as well.
- We do this in the form of a "pseudo" linked list.
- I say pseudo because in a regular linked list, the implication is that each element is allocated only when necessary.
- What we're going to do is make each object in our pre-allocated array similar to a linked list `Node` with two fields, the first is an instance of `YourObject`, the second is a "next" pointer.
- It might look like this:

```
class Element {
public:
  YourObject  theObject;
  Element     *nextElement;
};
```

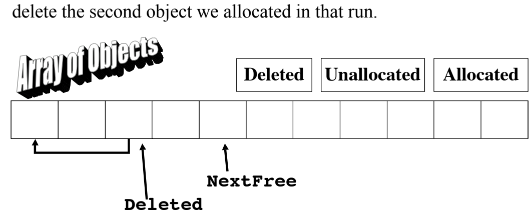
Memory Management (cont)

- Now, when we delete an object, we'll put it at the beginning of a linked list of "deleted" objects.
- Whenever `YourObject::operator new` is called, we'll first check to see if there are any objects on the "deleted objects" list.
- If there are, we'll return the first object from that list instead of the object pointed at by `NextFree`.
- What happens if we delete that object we just allocated...



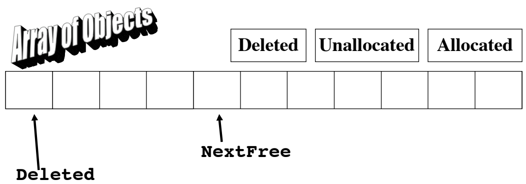
Memory Management (cont)

- OK, now let's say that we allocated three more objects and then delete the second object we allocated in that run.



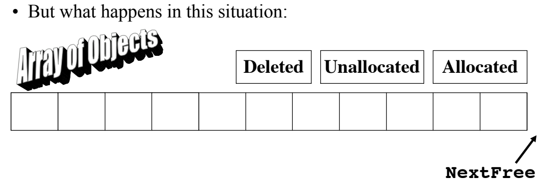
Memory Management (cont)

- Now, let's allocate another object.
- We'll use the first one on our list of deleted objects...



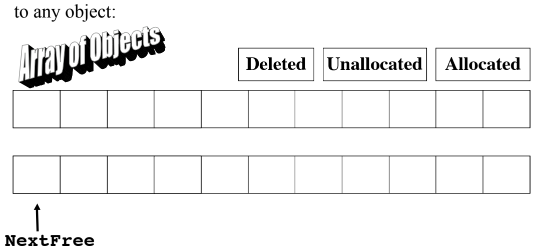
Memory Management (cont)

- But what happens if we allocate all the elements in our array?
- You can always fall back on the default operator new in your YourObject::operator new code.
- As a matter of fact, in YourOperator::operator new, you should always check that the size requested matches the size of the object you've defined the memory handler for, and default to ::operator new if it doesn't.
- But what happens in this situation:



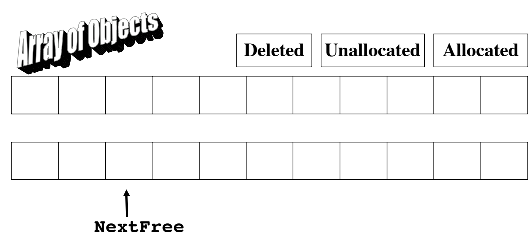
Memory Management (cont)

- It's actually pretty simple.
- We just allocated another array.
- It doesn't matter if it's not contiguous with the previous array, our linked list of deleted objects will mean we'll never "lose reference" to any object:



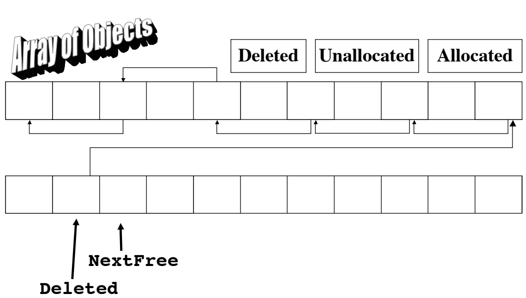
Memory Management (cont)

- Now, let's allocate two more objects:



Memory Management (cont)

- And now delete every other object:



Memory Management (cont)

- Any questions on the memory management algorithm?
- Let's look at some code:

```
void *MyString::operator new(size_t size)
{
  // Make sure size requested matches a MyString size
  if (size != sizeof(MyString))
  {
    return ::new(size);
  }
  // First, check the free list
  if (deletedMyStrings)
  {
    void *freePtr = deletedMyStrings;
    deletedMyStrings = deletedMyStrings->theNextString;
    return freePtr;
  }
```

Memory Management (cont)

```
 // Second, check to see if there are any more "array" elements
  if (nextFreeMyString < lastMyString) {
    void *freePtr = nextFreeMyString++;
    return freePtr;
  }
  // Allocate another chunk
  char *newChunk = new char[100 * sizeof(MyStringFreeList)];
  nextFreeMyString = (MyStringFreeList *) newChunk;
  lastMyString = nextFreeMyString + 100;
  void *freePtr = nextFreeMyString++;
  return freePtr;
}

void MyString::operator delete(void *d)  {
  ((MyStringFreeList *)d)->theNextString =
                    (MyStringFreeList *)deletedMyStrings;
  deletedMyStrings = (MyStringFreeList *)d;
}
```

Demonstration 2

Better Dynamic Allocation???

Allocator Classes

- The fact that default memory management can be overridden and improved for specialized cases is significant in the following scenario.
- Consider a class named allocator which implemented a couple of basic memory allocation member functions:

```
class allocator
{
public:
  void *allocate(size_t size);
  void deallocate(void *);
};
```

Allocator Classes (cont)

- Now consider that this class implements the two member functions in the following way:

```
void *allocator::allocate(size_t size)
{
  return ((void *) new char[size]);
}

void allocator::deallocate(void *ptr)
{
  char *thisPtr = (char *) ptr;
  delete [] thisPtr;
}
```

Allocator Classes (cont)

- Now consider that my fancy memory management algorithm could be implemented to take object size as a parameter and then fit into this same type of allocator class:

```
template<int objectSize>
class RonsAllocator {
public:
  void *allocate(size_t size);
  void deallocate(void *ptr);
private:
  // all necessary private members here...
};
```

Allocator Classes (cont)

- NOW consider that this allocator class could be taken as a template argument itself…
- Hmmm, did I ever tell you about default template arguments?

```
template<class storageType, class A = allocator<storageType> >
class MyCoolListClass
{
public:
  …
  void *operator new(size_t sz)
  {
    return A.allocate(sz);
  }

  void operator delete(void *ptr)
  {
    A.deallocate(ptr);
  }
```

Allocator Classes (cont)

- This concept is what is meant when we talk about *Allocator Classes*.
- The “real” allocator classes are used all over the standard template library.
- They are passed as template arguments to any STL container.
- They have default template arguments of `allocator<T>` which is the default allocator class which just relies on `new` and `delete`.
- So, you could implement your own memory management for type (presumably of the same size) and pass that to any of the STL containers to help make dynamic memory allocation faster.
- The actual allocator classes are a little more complicated, LNG doesn’t even go into them.
- Consult Stroustrup for more information. (Well, his book, not necessarily him)

Final Thoughts

- Assignments #10 due on 11/30
- Prelim #2 given in class next Tuesday
- That’s it for LNG
- Classes on 11/30 and 12/2…