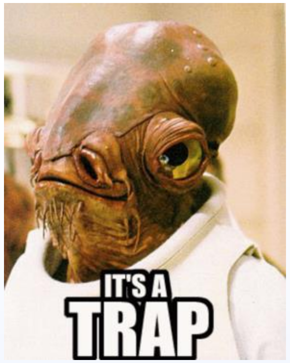
EECS 280 – Lecture 14

Managing Dynamic Memory

Review: Memory Leaks/Errors



- **Memory Leaks**

  You're not using dynamic memory, but you never free up the space for it.
- **Orphaned Memory**

  You lose the address of a heap object, meaning it will inevitably be leaked.
- **Double Free**

  You try to free heap memory too many times.
- **Non-Heap Delete**

  Use `delete` with a pointer to a non-heap object.
- **Wrong Delete**

  Getting `delete` and `delete[]` mixed up.
- **Use a Dead Object**

  You can keep around the address of a dead heap object and accidentally use it.

Example: Memory Leak

- This function allocates two arrays.
  - However, it does not use the `delete` operator.
  - Although the pointers `arr` and `arr2` are cleaned up when they go out of scope, the arrays they point to are not.

```
void doSomeArrayThing(int size) {
  int *arr = new int[size];
  arr[0] = 5;

  int *arr2 = new int[2 * size];
  arr2[0] = 5;
}
```

- But wait, why can't the compiler add this rule?

"When a pointer goes out of scope…

…just delete what it points to."

Pointers and the Compiler

- Why can't the compiler clean up the array for you?
- **Proposal**: *Whenever a pointer object goes out of scope, the compiler deletes the object it points to.*

```
void doSomeArrayThing(int size) {
A   int *arr = new int[size];
B   arr[0] = 5;

C   int *arr2 = new int[2 * size];
D   arr2[0] = 5;
}
```

- **Exercise**: Find a way to break this scheme by adding or modifying one line of code above.

Note: The STL provides "smart pointers", allow you to opt-in to something like this for certain pointers. They're pretty cool, but beyond the scope of the course (for now).

Example: Managing Dynamic Arrays

- General principle: *Prefer to package* ***new/delete*** *inside an ADT rather than use them manually.*

```
void doSomeArrayThing(int size) {
  // Allocate memory
  int *arr = new int[size];
  int *arr2 = new int[2 * size];

  // Use it
  arr[0] = 5;
  arr2[0] = 5;

  // Free memory
  delete[] arr;
  delete[] arr2;
}
```

```
void doSomeArrayThing(int size) {
  // Constructor - Create ADTs
  // Memory allocated automatically
  DynamicIntArray arr(size);
  DynamicIntArray arr2(2 * size);

  // Use it
  arr[0] = 5;
  arr2[0] = 5;

  // Free memory?

}
```

No new/delete here! Let's see how it works…

Example: Managing Dynamic Arrays

```
class DynamicIntArray {
private:
  int *arr;
  int arr_size;

public:
  DynamicIntArray(int size)
    : arr(new int[size]), arr_size(size) { }

  ~DynamicIntArray() {
    delete[] arr;
  }

  int size() const {
    return arr_size;
  }

  int & operator[](int i) { return arr[i]; }
  const int & operator[](int i) const { return arr[i]; }

};
```

Constructor allocates a dynamic array.

Destructor cleans up the dynamic array.

Member functions and operators make up the interface.

The const version can be used for DynamicIntArrays declared as const.

Example: Managing Dynamic Arrays

- General principle: *Prefer to package **`new/delete`** inside an ADT rather than use them manually.*

```
void doSomeArrayThing(int size) {
  // Allocate memory
  int *arr = new int[size];
  int *arr2 = new int[2 * size];

  // Use it
  arr[0] = 5;
  arr2[0] = 5;

  // Free memory
  delete[] arr;
  delete[] arr2;
}
```

```
void doSomeArrayThing(int size) {
  // Constructor - Create ADTs
  // Memory allocated automatically
  DynamicIntArray arr(size);
  DynamicIntArray arr2(2 * size);

  // Use it
  arr[0] = 5;
  arr2[0] = 5;

  // Destructor - Memory freed
  // automatically when arr and
  // arr2 go out of scope
}
```

No new/delete here!

RAII: Resource Acquisition Is Initialization

- Problem:
  The compiler is bad at managing resources.
  - If we use `new` to create dynamic memory, the compiler can't clean it up for us.
- Observation:
  The compiler can manage the lifetime of local objects.
  - Automatic storage duration.
  - Cleaned up when they go out of scope.
- Idea:
  Wrap up resource management in a class and use local instances of the class to access the resources.
  - **Ctor:** The resource is acquired when the object is born.
  - **Dtor:** The resource is released when the object dies.

Upgrading `UnsortedSet`

```
template <typename T>
class UnsortedSet {
public:
  // Maximum capacity of a set.
  static const int ELTS_CAPACITY = 10;

  ...

  // REQUIRES: size < ELTS_CAPACITY
  // EFFECTS:  adds v to the set
  void insert(T v);

  ...
};
```

Let's remove the fixed capacity restriction. We'll use dynamic memory in the implementation of `UnsortedSet` to make it work.

Idea

- Dynamically allocate the array…
- Need more space? Make a new, larger array, copy over the elements, and throw away the old one!

Using a Dynamic Array Instead

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;
```

Instead of directly having the array as a member, we use a pointer to a dynamically allocated array.

The lifetime of the array is now independent from the `UnsortedSet`.

```
  int capacity;
```

Current capacity of the dynamic array. This no longer has to be the same for all `UnsortedSets`.

```
  int elts_size;
```

Number of elements in the set. (Number of valid cells in the array.)

```
  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow();
};
```

UnsortedSet Destructor

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet()
    : elts(new T[DEFAULT_CAPACITY]),
      capacity(DEFAULT_CAPACITY),
      elts_size(0) {}

  ~UnsortedSet() {
    delete[] elts;
  }
  ...

private:
  T *elts;
  int capacity;
  int elts_size;
};
```

Allocate dynamic array in the constructor. elts points to it.

Clean up memory for dynamic array in the destructor.

Destructors

- We used a destructor to take care of this in `DynamicIntArray`.

- What is the role of the destructor?
  - Common misconception:
    The destructor destroys the object…No!
  - It gives the object a chance to put its affairs in order before dying.
  - In the two examples today, this means cleaning up the dynamic array so it isn't leaked.

Calling grow() as Needed

```
template <typename T>
class UnsortedSet {
public:
  void insert(T v) {
    if (contains(v)) { return; }
    if (elts_size == capacity)) { grow(); }

    elts[elts_size] = v;
    ++elts_size;
  }

private:
  T *elts;
  int capacity;
  int elts_size;

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow();
};
```

If we are attempting to add an element, but there is no more room, call grow() to allocate a larger dynamic array.

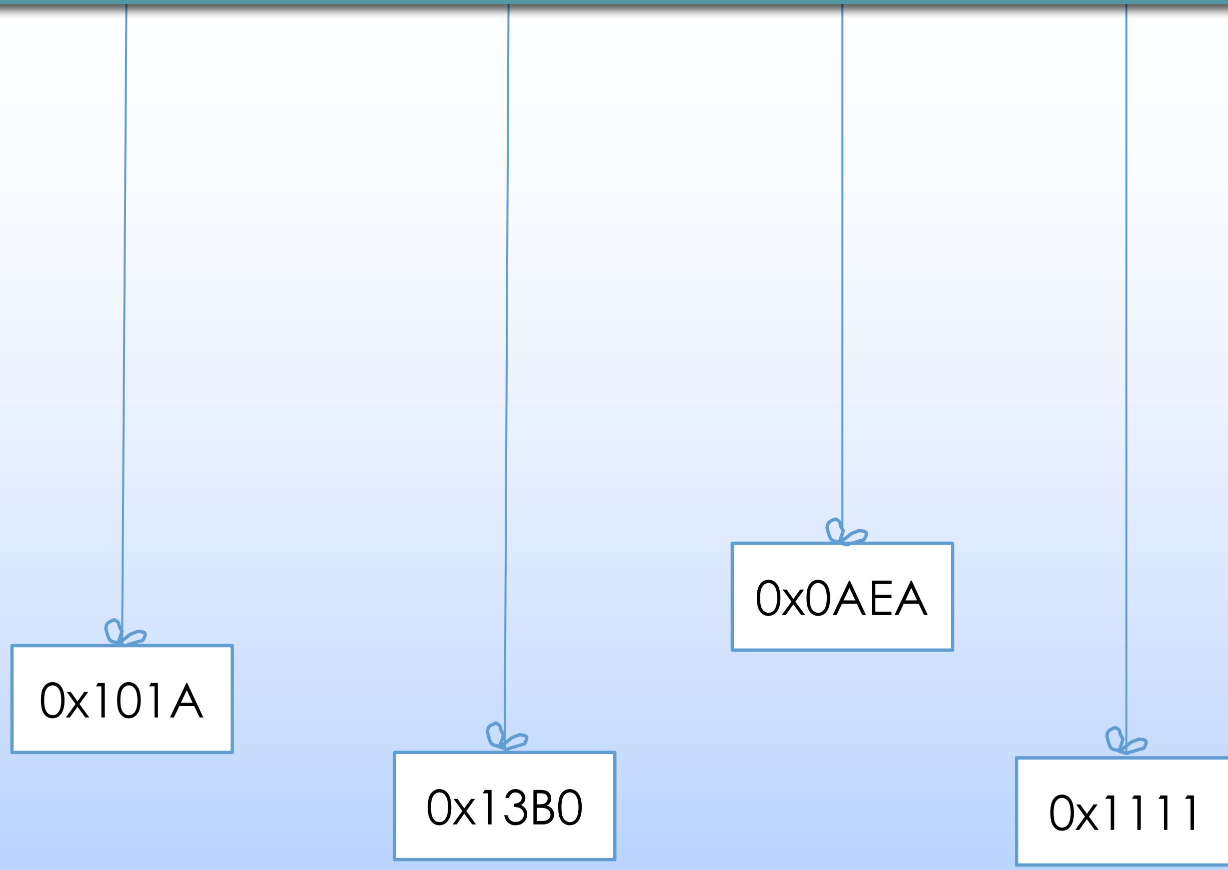
Exercise: grow()

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;
  int capacity;
  int elts_size;

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow() {
    // TODO: WRITE YOUR CODE HERE


  }
};
```

In order to grow…

1. **Make a new array with twice as much capacity**
2. **Copy elements over**
3. **Update capacity**
4. **Destroy old array**
5. **Point elts to the new array**

Break Time!



They're dangling pointers.

Okay, maybe it's not that funny.

Break Time!

We'll start again in three minutes.

Exercise

Question

Which of these grow functions are correct?

```
void grow() {
  T *newArr = new T[2 * capacity];
  for (int i = 0; i < elts_size; ++i) {
    newArr[i] = elts[i];
  }
  capacity *= 2;
  elts = newArr;
}
```

A

```
void grow() {
  T *oldArr = elts;
  elts = new T[2 * capacity];
  for (int i = 0; i < elts_size; ++i) {
    elts[i] = oldArr[i];
  }
  capacity *= 2;
  delete[] oldArr;
}
```

C

```
void grow() {
  T *newArr = new T[2 * capacity];
  for (int i = 0; i < elts_size; ++i) {
    newArr[i] = elts[i];
  }
  capacity *= 2;
  elts = newArr;
  delete[] elts;
}
```

B

```
void grow() {
  T *oldArr = elts;
  elts = new T[2 * capacity];
  delete[] oldArr;
  for (int i = 0; i < elts_size; ++i) {
    elts[i] = oldArr[i];
  }
  capacity *= 2;
}
```

D

Solution: grow()

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;
  int capacity;
  int elts_size;

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow() {
    T *newArr = new T[2 * capacity];
    for (int i = 0; i < elts_size; ++i) {
      newArr[i] = elts[i];
    }
    capacity *= 2;
    delete[] elts;
    elts = newArr;
  }
};
```

In order to grow…

1. Make a new array with twice as much capacity
2. Copy elements over
3. Update capacity
4. Destroy old array
5. Point elts to the new array

The Power of Indirection

- In `UnsortedSet`, we've decoupled the lifetime of the object (i.e. the set itself) from the dynamic array it uses.
- This is possible because the array is not **directly** a member of the class.
- Instead, we use a pointer to work with the array **indirectly**.
  - We can just make a new one if we need it to be bigger (and free up the old one)

Note: The lifetime of the member `elts` itself is tied to the whole object, but `elts` is just the pointer, not the array!

`new` and `delete`

- Generally, it's a good strategy to think of matching up `new` and `delete` in your code.
  - Allocate with `new`, clean up with `delete`.
  - This doesn't mean you literally have the same number of `new` and `delete`.
  - A single `new/delete` can be used many times if it's in a loop, a function called several times, etc.
- All objects that come from a `new` must eventually meet a `delete`!

- Use this pattern for constructors and destructors.
  - If you use `new` in the constructor, almost certainly you need to use `delete` in the destructor.

Dynamic Resource Invariant

```
template <typename T>
class UnsortedSet {
...
private:

  // INVARIANT: An UnsortedSet manages a dynamically
  //            allocated array. During its lifetime
  //            there is exactly one such array,
  //            pointed to by elts.
  T *elts;
  int capacity;
  int elts_size;

  int indexOf(int v) const {
    for(int i = 0; i < elts_size; ++i){
      if(elts[i] == v){ return i; }
    }
    return -1;
  }
};
```

Add an invariant to guarantee elts is always safe to use.

Because elts is now just a pointer, do we have to worry it might be uninitialized or null?

Dynamic Resource Invariant


```
template <typename T>
class UnsortedSet {
...
private:
  // INVARIANT: An UnsortedSet manages a dynamically allocated
  // array. During its lifetime there is exactly one such array,
  // pointed to by elts.
  T *elts;

  UnsortedSet() : elts(new T[DEFAULT_CAPACITY]),
                  capacity(DEFAULT_CAPACITY), elts_size(0) {}

  ~UnsortedSet() { delete[] elts; }

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow() {
    T *newArr = new T[2 * capacity];
    for (int i = 0; i < elts_size; ++i) { newArr[i] = elts[i]; }
    capacity *= 2;
    delete[] elts;
    elts = newArr;
  }
};
```

Establish invariant.

Clean up.

Break invariant (temporarily).

Restore invariant.

Exercise

Question

Which of these code snippets leak memory?

```
template <typename T>
class UnsortedSet {
private:

  T *elts;

  UnsortedSet() : elts(new T[DEFAULT_CAPACITY]),
                  capacity(DEFAULT_CAPACITY), elts_size(0) {}

  ~UnsortedSet() { delete[] elts; }
};
```

A

```
void func() {
  UnsortedSet<int> s1;
  s1.insert(2);
  s1.insert(3);
}
```

C

```
void func() {
  UnsortedSet<int*> s2;
  s2.insert(new int(2));
  s2.insert(new int(3));
}
```

B

```
void func() {
  UnsortedSet<int> *s3
    = new UnsortedSet<int>;
  s3->insert(2);
  s3->insert(3);
}
```

D

```
void func() {
  UnsortedSet<int> *s4
    = new UnsortedSet<int>;
  s4->insert(2);
  s4->insert(3);
  delete s4;
}
```

Recall: Working with Objects Indirectly

- Using a pointer or reference to keep track of an object is really powerful.
  - You get *reference semantics*.
    (i.e. avoid making a copy)
  - Use objects across different scopes.
  - Enable *subtype polymorphism*.
  - Keep track of objects in **dynamic memory**.

- However, it's also tricky…

But wait, there's more!

```
int main() {
  UnsortedSet<int> s1;
  s1.insert(2);
  s1.insert(3);
  cout << s1 << endl; // prints {2, 3}

  UnsortedSet<int> s2 = s1;
  cout << s2 << endl; // prints {2, 3}

  s2.insert(4); // will cause a grow

  cout << s2 << endl;
  // prints {2, 3, 4}, ok cool

  cout << s1 << endl; // EXPLODE
}
```