ECE 590.01
C++ Programming, Data structures, and Algorithms
OOP and ADTs revisited

Admin
- Hwk 2
  - Due next week (Mon)
- Project Proposals
  - Due next week (Wed)

Remind us where we left off last time?
- Who can remind us what we talked about?

OOP Revisited
- Last time: many questions about abstract classes
  - Take a few minutes here to revisit OOP in general
  - First: OOP is for programming in the large
    - What does this mean?
    - What is programming in the small?

Programming in the Large
- Programming in the small:
  - One or a few functions
  - What you have done on your hwk so far
- Programming in the large:
  - Big programs/complete systems
  - Abstraction becomes key
  - Interfaces between components
  - OOP is aimed at this
    - Why? How?
### OOP: Programming in the Large

- **Abstraction**
  - Hide implementation details within an object
  - Public interface
  - Private fields/methods/classes

- **Polymorphism + dynamic dispatch**
  - Treat things the same (Dogs, Cats, Fish: all Animals)
  - Retain specific behaviors of sub-types even in general algorithms

- **Encapsulation**
  - Data and methods packaged up together

### OOP Example

- Suppose we are making some e-commerce program with multiple different shipping methods
  - USPS
  - FedEx
  - UPS

- Suppose each of these has different ways to compute cost (as a function of weight, distance etc).

### Without OOP (e.g., C)

```c
int getCost(package_t * pck) {
    switch(pck->type) {
        case TYPE_USPS:  return ...;
        case TYPE_FedEx: return ...;
        case TYPE_UPS: return ...;
        default: error("Unknown package type");
    }
}
```

- Add other shipping types?
- Other functions? Change them all too to accommodate the new types

### OOP Design

- **Abstract class**
  - Can have some concrete functionality (get/set Weight/Dest)
  - But leave other parts abstract

```c
AbstractPackage
getCost() { .. }
```

### Using abstract classes

```c
int totalCost(AbstractPackage ** pckgs, int n) {
    
}
```

- **Add new shipping method**
  - Add a subclass
  - Abstract methods force programmer to implement methods

- An array of Pointers to AbstractPackages
Using abstract classes

```java
int totalCost(AbstractPackage **pckgs, int n) {
    int total = 0;
    for (int i = 0; i < n ; i++) {
        total += pckgs[i]->getCost();
    }
    return total;
}
```

AbstractPackage guarantees that all subclasses have `getCost()`
Even though AbstractPackage doesn’t specify an implementation
Dynamic dispatch automatically makes this use the right method

OOP Design

```java
AbstractPackage
    setWeight(){}
    setDestination(){}
    getWeight(){}
    getDestination(){}
    getCost() abstract

USPSPackage
    getCost(){}

FedExPackage
    getCost(){}

USPShandPackage
    getCost(){}

HandCourierPackage
    getCost(){}

ECE 590.01 (Hilton): OOP/ADT
```

Extensibility "vertically" is easier too
- E.g., more specific types of our existing types
- Contrast with non-OOP (completely new "type" vs sub-cases?)

OOP and Software Design

- Remember two design approaches
  - Top-down:
    - Start with the main algorithm
    - Find you need other sub-algorithms
    - Repeat
    - Write algorithm, make interfaces for objects, then implement
    - Repeat until done
    - Test with simple implementations/scaffolds
  - Bottom-up:
    - Make small building blocks
    - Build larger things from them
    - Build "bottom" classes (test), build "next layer"
    - Repeat until done

ADTs (con’td): Sets

- Next ADT: Set<T>
  (think: mathematical set)
  - Add element to set
  - Test for membership (contains)
  - Union
  - Intersect
  - Count elements
  - ...
  - ...

Set: Implement with LL?

- Could implement a Set with a LinkedList
  - Add: add to list (seen that before)
  - Contains: iterate through list checking
    - Requires equality checking
  - Union: add all items from each set
  - Intersection:
    - Make new set S
    - Iterate through X, for each item Z
      - Check if Z is in Y if so, add to S
    - Return S
Set intersection

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    LLNode<T> curr = list->head;
    while (curr != NULL) {
        if (y->contains(curr->data)) {
            ans->add(curr->data);
        }
        curr = curr->next;
    }
    return ans;
}
```

Set intersection

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    LLNode<T> curr = list->head;
    while (curr != NULL) {
        if (y->contains(curr->data)) {
            ans->add(curr->data);
        }
        curr = curr->next;
    }
    return ans;
}
```

Set intersection

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    LLNode<T> curr = list->head;
    while (curr != NULL) {
        if (y->contains(curr->data)) {
            ans->add(curr->data);
        }
        curr = curr->next;
    }
    return ans;
}
```

Set intersection

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    LLNode<T> curr = list->head;
    while (curr != NULL) {
        if (y->contains(curr->data)) {
            ans->add(curr->data);
        }
        curr = curr->next;
    }
    return ans;
}
```

Set intersection

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    int n = list->size();
    for (int i = 0; i < n; i++) {
        T & d = list->get(i);
        if (y->contains(d)) {
            ans->add(d);
        }
    }
    return ans;
}
```

What is bad here?

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    LLNode<T> curr = list->head;
    while (curr != NULL) {
        if (y->contains(curr->data)) {
            ans->add(curr->data);
        }
        curr = curr->next;
    }
    return ans;
}
```

Should really have LLNodes hidden inside the linked list, not visible outside

What don't we like about this implementation?
Set intersection

Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    int n = list->size();
    for (int i = 0; i < n; i++) {
        T & d = list->get(i);
        if (y->contains(d)) {
            ans->add(d);
        }
    }
    return ans;
}

What don’t we like about this implementation?
Slower: why?

Getting a particular item out of a list by index takes iterating through that many elements.

How fast is something?
• How fast is a piece of code?
  • It depends (on what?)

Big-O: Asymptotic Behavior
• Observation: small things generally don’t matter
  • Computers: pretty fast
  • Difference between 20 and 50 instructions: you won’t notice

• Instead, think about behavior on large inputs
  • How does the execution time scale as input size grows?
  • Use Big-O to approximate within some formal rules
    • Upper bound, ignoring constant factors

Big-O
T * get(int n) {
    LLNode<T> * curr = head;
    while (curr != NULL && n > 0) {
        curr = curr->next;
        n --;
    }
    if (curr == NULL) { print_error_and_abort(); } else {
        return curr->data;
    }
}

We call this O(N)
The time it takes is linear in N
Don’t care if its 3N + 2 or 4N + 7 or ..., all of these are O(N)
**Big-O**

- Formalism:
  \[ f(x) \text{ is } O(g(x)) \text{ iff exists } x_0, c \text{ .} \forall x > x_0. \quad f(x) \leq c \times g(x) \]

**Not Big-O**

- Consider
  - \[ f(x) = x^2 \]
  - \[ g(x) = x \]
  
- \[ f(x) \text{ is not } O(g(x)) \]

**Big-O: continued**

- \( O(1) \): constant time
  - Run time doesn’t vary with input/data structure size
  - Example: add to front of linked list

- \( O(N) \): linear time
  - Double elements, double runtime
  - Example: iterate through all elements of array/list
**Big-O: continued**

- **O(1): constant time**
  - Run time doesn’t vary with input/data structure size
  - Example: add to front of linked list
- **O(N): linear time**
  - Double elements, double runtime
  - Example: iterate through all elements of array/list
- **O(N^2): quadratic time**
  - Double elements, quadruple runtime
  - Example: our intersection algorithm from before
- **O(N^3): cubic time**
- **O(2^N): Exponential Time**
  - Increase elements by 1, double runtime
  - Examples: Exhaustive search of binary possibilities

---

**Set intersection**

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T>();
    int n = list->size();
    for (int i = 0; i < n; i++) {
        T & d = list->get(i);  // O(N)
        if (y->contains(d)) {  // O(N)
            ans->add(d);  // O(1)
        }
    }
    return ans;
}
```

---

**Set LL implementation Big-Os**

- **Next ADT: Set<T>** (think: mathematical set)
  - Add element to set O(1)
  - Test for membership O(N)
  - Union O(N)
  - Intersect O(N)
  - Count elements O(N) or O(1)
    - Explicitly count LL elements O(N)
    - Track number of elements in counter O(1)
  - ...
  - ...

---

**A bit more about performance**

- In some situations, performance really matters
  - Big-O ignores constant factors: these can matter in the real world
    - Set insert: both versions O(N^2)
      - N+1 things inside loop (times N)
      - N+N+1 things inside loop (times N)
  - Hardcore performance optimization?
    - Understand your hardware
    - Understand your compiler
    - Profile your code
    - Tools that find where your code is spending time
    - Let's you optimize the common case
    - Amdahl's Law

---

**Performance vs Abstraction?**

- Seems like we have to make a tradeoff:
  - Performance (expose the internals of the list)
  - Abstraction (hide the internals of the list and access slowly)
- Can we have our cake and eat it too?
**Cake + Eating (C++ style)**

- C++ style solution: expose an iterator
  - Public inner class
    - Encapsulates a position in the structure + access to it

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T> ();
    LinkedList<T>::iterator it = list->begin();
    while (it != list->end()) {
        if (y->contains(*it)) {
            ans->add(*it);
        }
        ++it;
    }
    return ans;
}
```

**ECE 590.01 (Hilton): OOP/ADT**

---

**Cake + Eating (C++ style)**

- C++ style solution: expose an iterator
  - Public inner class
    - Encapsulates a position in the structure + access to it

```cpp
Set<T> * intersect(const Set<T> * y) {
    Set<T> * ans = new Set<T> ();
    LinkedList<T>::iterator it = list->begin();
    while (it != list->end()) {
        if (y->contains(*it)) {
            ans->add(*it);
        }
    ++it;
    return ans;
}
```

**ECE 590.01 (Hilton): OOP/ADT**

---

**Implementing Iterator**

- Nothing magic: just an inner class that encapsulates the state

```cpp
template<class T> class LinkedList {
  ...
    class iterator {
        LLNode<T> * ptr;
        T & operator *() { return ptr->data; }
        iterator operator++() {
            ptr = ptr->next;
            return *this;
        }
    ...
};
```

**ECE 590.01 (Hilton): OOP/ADT**

---

**Not the only approach**

- C++’s approach: Iterators
  - Decent way to do this in OO languages
  - Balances encapsulation vs performance
- Functional languages:
  - Pass a function to the data structure to say what to do
  - Let data structure handle iteration internally
  - Puts "iteration boilerplate" inside rather than outside
  - Reducing boilerplate code = good
Map: the most useful ADT ever

- Sets are incredibly useful
- Maps are even more useful
  - Track mappings from keys to values
  - Probably the most ubiquitous thing we do in programs

- Interface
  - add(key, value)
  - find(key)
  - delete(key) [maybe?]
  - iterators [maybe?]

Maps with Linked Lists

- Could do Linked List implementation
  - add(key, value) \(O(1)\)
  - find(key) \(O(N)\)
  - delete(key) \(O(1)\)

- Is this good?
  - \(O(N)\) sounds good, but can actually do better
  - \(O(\lg N)\): Double input size, add 1 to runtime.
  - Pretty good: go from 2 Billion to 4 Billion, runtime 31 -> 32
  - We'll delve into details next time

Summary

- OOP
  - Brief examination of design
  - Abstract class example

- ADTs: continued
  - Sets
  - Maps
  - Iterators

- Performance
  - Big-O