Encapsulation

* Based on notes by Brad Lushman, used with ... I'll ask later.

- Encapsulation is the binding of data together with the methods that operate on the data. Also, how we limit access to the data through the provided methods.
- Want clients to treat objects as *capsules* - similar to black boxes.
- Clients only need to understand what functionality is provided not how it is implemented; i.e. a client only needs an "abstraction" of how it works.
- Want to avoid: clients writing code dependent on an implementation (that could change), using code in a way that violates how it was intended to be used (violating invariants), etc.

Encapsulating Linked Lists - Interface

Wrapper class `List` has exclusive access to the underlying Node objects.

```
// Interface: list.cc

export class List {
   struct Node;  // Private nested class
   Node *theList = nullptr;

 public:
   void addToFront(int n);
   int &ith(int i);
   ~List();
   ...
};
```

Encapsulating Linked Lists - Implementation

```
// Implementation: list-impl.cc
struct List:: Node {   // Nested class
   int data;
   Node *next;
   ...
   ~Node() { delete next; }
};

List:: ~List() { delete theList; }

void List::addToFront(int n) {
   theList = new Node{n, theList};
}

int &List::ith(int i) {
   Node *cur = theList;
   for (int j = 0; j < i; ++j, cur = cur->next);
   return cur->data;
}
```

Encapsulating Linked Lists

- `stuct Node` is under the private section in `List`
- `Node *theList` is also private in `List`

⇒ The nodes are only accessible inside `List` and only `List` can directly manipulate `Node` objects (Encapsulation).

⇒ This allows us to guarantee the invariant: `next` is either `nullptr` of a `Node` allocated by `new` - since we control the `Node` objects.

How do we traverse the linked list?

Encapsulating Linked Lists

- `stuct Node` is under the private section in `List`
- `Node *theList` is also private in `List`

⇒ The nodes are only accessible inside `List` and only `List` can directly manipulate `Node` objects (Encapsulation).

⇒ This allows us to guarantee the invariant: `next` is either `nullptr` of a `Node` allocated by `new` - since we control the `Node` objects.

How do we traverse the linked list?

Use `List::ith(i)` for `i=0, 1, 2, ..., n-1`
⇒ **Runtime**: $O(n^2)$ to traverse the list!

Many of the operations we want to implement will traverse the list and we don't want them to take $O(n^2)$, we want $O(n)$.

Iterator Pattern

We want to maintain properties of encapsulation:

- hide implementation details $\Rightarrow$ if the class implementation changes, client code should be uneffected
- design to prevent client misuse; limiting access and exposure of internal details; such as `Node` structure

Iterator Pattern

We want to maintain properties of encapsulation:

- hide implementation details ⇒ if the class implementation changes, client code should be uneffected
- design to prevent client misuse; limiting access and exposure of internal details; such as `Node` structure

We also want to allow a client to traverse the `List` in an efficient and safe way (limit access to the nodes details such as memory addresses, etc)
⇒ create a class that manages access to nodes.

- Need an abstraction of a pointer.
- Functions that will allow client to walk the list without exposing the pointers.

⇒ **Iterator Pattern**

Iterator Pattern

What do we want to be able to do?

- Move from one item in the List to another ("increment the pointer").
- Access the data at the current location ("dereference the pointer").
- Have a starting point: `begin()`.
- Have a finishing point: `end()`.
- Be able to check if we are at the end of not: `operator!=`

```
class List {
   struct Node;
   Node *theList;
 public:
```

Iterator Pattern - Iterator class (nested in List)

```
 public:
   class Iterator {
      Node *p;   // Private
    public:
      explicit Iterator(Node *p): p{p} {}
      int &operator*() { return p->data; }
      Iterator &operator++() {
         p = p->next;
         return *this;
      }
      bool operator!=(const Iterator &other) const {
         return (p != other.p);
      }
   };
   Iterator begin() const { return Iterator{theList}; }
   Iterator end() const { return Iterator{nullptr}; }
};
```

Iterator Pattern - client usage

```
int main() {
  List lst;
  lst.addToFront(1);
  lst.addToFront(2);
  lst.addToFront(3);

  //  What type is auto here?
  for (auto it = lst.begin(); it != lst.end(); ++it) {
    cout << *it << endl;
  }
}
```

Iterator Pattern - client usage

```
int main() {
  List lst;
  lst.addToFront(1);
  lst.addToFront(2);
  lst.addToFront(3);

  //  List::Iterator
  for (auto it = lst.begin(); it != lst.end(); ++it) {
    cout << *it << endl;
   }
}
```

Note: at each step `operator++` returns an Iterator that is copied into `it`. We then use `operator*` to get the node data.

Built-in Support for the Iterator Pattern

Class Requirements:

- Methods `begin` and `end` that return `Iterator`s

Iterator Requirements:

- Must support prefix `operator++`, `operator!=` and unary `operator*`

Range-based for loop (C++11)

```
// access by value (makes a copy) of variable n, of type int
  for (auto n : lst) {
    cout << n << endl;  // implicit: n = *it
  }
```

Built-in Support for the Iterator Pattern

Class Requirements:

- Methods `begin` and `end` that return `Iterator`s

Iterator Requirements:

- Must support prefix `operator++`, `operator!=` and unary `operator*`

Range-based for loop (C++11)

```
// access by value (makes a copy) of variable n, of type int
  for (auto n : lst) {
    cout << n << endl;  // implicit: n = *it
  }
```

Recall: `int &operator*() { return p->data; }` gives access to mutate.

```
// access by reference to be able to mutate
  for (auto &n : lst) {
    n = ...;  // e.g. ++n
  }
```

More Encapsulation

The ctor for Iterator is in the public section.
⇒ A client of `List` can directly create `Iterator` objects violating encapsulation. For example:

```
auto it = List::Iterator{nullptr};
```

We want the client to use `begin` and `end`.

Should we make `List::Iterator`'s ctor private?

More Encapsulation

The ctor for Iterator is in the public section.
⇒ A client of `List` can directly create `Iterator` objects violating encapsulation. For example:

```
auto it = List::Iterator{nullptr};
```

We want the client to use `begin` and `end`.

Should we make `List::Iterator`'s ctor private?

- Client's can't call `List::Iterator{...}`

More Encapsulation

The ctor for Iterator is in the public section.
⇒ A client of `List` can directly create `Iterator` objects violating encapsulation. For example:

```
auto it = List::Iterator{nullptr};
```

We want the client to use `begin` and `end`.

Should we make `List::Iterator`'s ctor private?

- Client's can't call `List::Iterator{...}`
- But then neither can `List`

We want to give `List` access but restrict others.

Hello Friend!

Make `List` a *friend* to `Iterator`:

- As a friend, `List` has access to all members of `Iterator`

```
class List {
   ...
 public:
   class Iterator {
     Node *p;
     explicit Iterator(Node *p);  // ctor moved to private

    public:
      ...
      friend class List;  // Can be placed anywhere in
   }; ...                 //    class Iterator
};
```

Only have friends you trust!

Clients are now forced to create iterators through `begin` and `end` since the Iterator ctor is private.

`List` can create iterators as a friend.

Friendships weaken encapsulation - classes should have as few friends as possible.

Accessors and Mutators

Often create member functions to provide access to private fields.

```
class Vec {
   int x, y;

 public:
   ...
   int getX() const { return x; }   // accessor
   int setY(int z) { y = z; }   // mutator
}
```

What about `operator«`?

Needs access to private fields `x` and `y` but can't be a member function.

- Can use accessors, `getX` and `getY`, if defined.

What about operator«?

Needs access to private fields x and y but can't be a member function.

- Can use accessors, getX and getY, if defined.
- If no accessors, make operator« a *friend* function

```
class Vec {
  ...
  friend std::ostream &operator<<(std::ostream &out, const Vec &v);
};

ostream &operator<<(std::ostream &out, const Vec &v) {
  // friends can access private members
  return out << v.x << ' ' << v.y;
}
```