CS 246 Fall 2018 — Tutorial 7

October 31, 2018

Summary

1 Accessibility

- So far, we've had classes where everything is accessible to anyone using our classes. This is not ideal because it means that individuals will likely use our classes in ways we did not intend.

  In other words, we need to be able to ensure that our invariant is always true.

- We can restrict what someone outside of the class can see using the `private` and `public` keywords:

```
struct Node{
    int val;
private:
    Node* next;
}
```

- Everything after `private:` and before `public:` will only be accessible with in the class, i.e. within methods. Everything which is `public` will be accessible to everyone as before.

- C++ has the `class` keyword, which is like a struct, but the default accessibility is `private`:

```
class Node {
    Node* next;
public:
    int val;
};
```

- The accessibility restriction can be bypassed with the `friend` keyword

```
class LinkedList {
    ...
    // Pretty prints the LinkedList
    // the friend keyword allows access of LinkedList's private fields
    friend std::ostream &operator<<(std::ostream &out, const LinkedList &rhs);
}
```

- In CS 246, you should not use the `friend` keyword unless instructed otherwise or we provide you code that uses `friend`
- In real life, you should restrict the use of the `friend` keyword as much as possible

2 Nested Classes

- We may want to create a class which doesn't make sense to exist on its own. An example is implementing a wrapper class for some structure to restrict others' ability to alter the class.
- A good example of this would be creating a wrapper class around the `Node` class we implemented.

```
class LinkedList {
    struct Node {
        int val;
        Node* next;
    }

    int numNodes;

    Node *head;
    Node *tail;

public:
    LinkedList();
    LinkedList(int amount, int what); // fill constructor

    void insertHead(int value);
    void insertTail(int value);

    void remove(int index);
};
```

- Since `Node` is declared within `LinkedList`, when we refer to the `Node` struct in our source code, we will refer to it as `LinkedList::Node`. This states that we are using the `Node` struct which is part of the `LinkedList` class.
- Note that since the `Node` struct is private within the `LinkedList` class, we will not be able

to create instances of Nodes outside of the `LinkedList` class. Our Nodes are safe from others tampering with our `LinkedList` Nodes.

3 Iterators

- Iterators are used to traverse containers in some order.
- Such order can be specified by the definition of the iterator, or not specified in any order.
- In C++, iterators are usually implemented with the following functions with in the container class (`Container`) and the nested iterator class (`Container::Iterator`):
  - `Container::begin()` — returns the iterator that represents the beginning of the iteration sequence.
  - `Container::end()` — returns the iterator that represents the end of the iteration sequence (which is NOT included in the elements being iterated)
  - `Container::Iterator::operator++()` (prefix) — increment the iterator by one, and return the incremented iterator.
  - `Container::Iterator::operator!=(const Iterator &other)` — returns true if two iterators does not represent the same element, false otherwise.
  - `Container::Iterator::operator*()` (unary) — return a reference of the element being represented by the iterator. It could also be a const reference, but what would you lose?
- The above is crucial to support the range-based for loop for objects:

```
// Assume that Container iterator iterates through objects of type T
Container c;
for (T &v : c) {
    // You can change v here.
    v.modify();
    cout << v << endl;
}
```

- See lecture material for implementation of iterators for linked lists

4 Class Relationships

- There are three types of relationships between classes which we typically discuss:
  - **Composition(owns-a):** class A *owns an* instance of class B. This means that class A is responsible for deleting the instance of class B when an object of class A is destroyed.
    - Under composition, instances of B cannot be shared.

- **Aggregation(has-a):** class A *has an* instance of class B. This means that class A is not responsible for deleting the instance of class B.
  - if an object of class A is deleted, the instance of class B associated to it lives on.
  - multiple objects of class A can have the same instance of class B.
- **Inheritance (is-a):** class B *is a* class A. This means that an instance of class B can be used in any situation where an instance of class A can be used.

  Note:
  - the converse is not true. That is, an instance of class A cannot always be used where an instance of class B can be used.
  - for this course, we are mainly concerned with public inheritances.

- **Note:** If a class A has a pointer to an instance of class B, you cannot know if the relationship is composition or aggregation without looking at documentation.

```
class B {
    ...
};

class A {
    B b; // This is composition
    B *b2; // This could be composition or aggregation
};
```

5 Inheritance

- Example:

```
class A {
    int a;

public:
    A(int a) : a{a} {}
};

class B : public A {
    int b;

public:
    B(int a, int b) : A{a}, b{b} {}
};
```

- In this example, B **inherits** from A (this is what the "`: public A`" is for). This means that every instance of B has the fields and methods which an instance of A has.
- Note the constructor for the B class. The first element of the MIL is `A{a}` which is calling the constructor for the A portion of the B object.

6 Encapsulation and Inheritance

- If A has fields which are private, B cannot access these fields (as they are private).
- What are some benefits of an inherited class not having direct access to the fields of the superclass?
  - Other people may inherit from our classes and this means they'd have access to the fields of the superclass in their implementation of their class, and *this breaks encapsulation.*
- However, we often want to give subclasses "special access" to the class.
  - For instance, we might want to have some accessor methods so that subclasses can access fields in a way that we choose, but we don't want to let everyone have access to these members.
- For this purpose, we can use the third type of privacy: `protected`.
- Members which are `protected` can be accessed directly by subclasses but cannot be accessed by the public.
- **Note:** Most of the time you should not make fields `protected` as this also breaks encapsulation.