CS 246 Spring 2018 — Tutorial 6

June 13, 2018

Summary

1 Visibility

- So far, we've had classes where everything is visible 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 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 visible within the class, i.e. within methods. Everything which is `public` will be visible to everyone as before.

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

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

```
};
```

- The visibility restriction can be by passed 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 are **not allowed** to 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 Const Methods

- Suppose we want to get the size of a `LinkedList`
- Because the field `numNodes` inside `LinkedList` is private, we have to write an **accessor**

```
class LinkedList {
    ...
    int size() {
        return numNodes;
    }
    ...
}

LinkedList list(4,5); // linked list with four 5's
cout << list.size() << endl; // prints 4
```

- Suppose our `LinkedList` has the `const` keyword

```
const LinkedList list(4,5);
cout << list.size() << endl; // Does not compile!
```

- The reason this code doesn't compile is because in our `size` method, we could be changing `list` but it is `const`
- Solution: Convert `size` into a `const` method

```
class LinkedList {
    ...
    int size() const {
        return numNode;
    }
    ...
}
```

- Now we can call `size` on `const LinkedLists`
- Like with variables, make as many of your methods `const` as possible

4 Member Operators

- Recall that we can overload operators

  ```
  // appends the linked list on the rhs to the end of the lhs
  LinkedList &operator+=(LinkedList &lhs, const LinkedList &rhs);
  ```

- We can actually make these operator overloads into methods

  ```
  class LinkedList {
      ...
      LinkedList &operator+=(const LinkedList &rhs);
      ...
  }
  ```

- As a method, the left hand side is referred to by the pointer `this`
- We can combine this with `const` methods as well

  ```
  class LinkedList {
      ...
      // creates a new linked list that is the result of appending
      // the rhs to the end of the lhs
      LinkedList operator+(const LinkedList &rhs) const;
      ...
  }
  ```

5 Static Fields and Methods

- Suppose we wanted to know how many `LinkedList`s were created over the course of a program. We can use static fields and methods to accomplish this
- A static field is a field that is associated with a class, as opposed to an instance of an object

  ```
  // LinkedList.h:
  class LinkedList {
      ...
      static int count;
      ...
      LinkedList() {
          ...
          count++;
      }
      ...
  }

  // LinkedList.cc:
  ```

```
// must be given value from outside of the class in the .cc file
// the type is mandatory, this is an initialization, not assignment
int LinkedList::count = 0; // before any ctors are called
```

- We can have static methods as well, they may only call static methods or access/mutate static variables

```
class LinkedList {
    ...
    static int getCount() {
        return count;
    }
    ...
}
```

- Now we can do

```
LinkedList list1;
LinkedList list2;
LinkedList* list3 = new LinkedList;
delete list3;

cout << LinkedList::getCount() << endl; // prints 3
```

- **Exercise:** How would you make it so that `count` keeps track of how many instances of `LinkedLists` are active (i.e. the above example would output 2 instead of 3)? Hint: You may want to modify the destructor

6 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++, iterator 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

7 Exception Basics

- The header `<stdexcept>` contains exceptions (in the form of classes) that can be thrown

```
#include <stdexcept>

int foo(int x) {
    if (x != 246) {
        throw invalid_argument("Not 246!");
    }
    return x;
}
```

- The constructor is called with a string (C or C++ style) which determines the message the exception holds
- When a `throw` statement occurs, normal program execution is interrupted, and the stack begins unwinding
- The unwinding process continues until either `main` is unwound and the program ends abruptly

```
int main() {
    foo(245); // this call will cause an exception to be thrown
    foo(246); // will not get to here
}
```

- Or when the exception reaches a `catch` statement

```
int main() {
    int x = 0;
    try {
        // causes an exception
        // x will still have a value of 0
        // because foo never returned
        x = foo(245);
    }

    catch (invalid_argument& ia) {
        // catches the exception
        // program flow continues here
        // what() method retrives the string the exception
        // was constructed with
        cout << ia.what() << endl;
    }

    // program flow continues
    foo(246); // will be executed
}
```

- **Note:** As the stack unwinds, the objects on the stack have their destructors called

8 Vim Tips of the Week: Insert and Command Mode

- Vim is designed to be used where you stay in normal mode most of the time. Nevertheless you probably are still using insert mode most of the time.
- So here's a few commands that can only be used in insert mode:

`C-w`	deletes the word before cursor
`C-u`	deletes from cursor to the start of the line
`C-p`	completes the current word, searching upwards
`C-n`	completes the current word, searching downwards
`C-x s`	completes spelling corrections
`C-x C-f`	completes file paths

- You probably also only used the command mode to save and quit vim. But it can do a lot more:

`<range>!<cmd>`	runs `<cmd>` in shell. If `<range>` is not empty, send text within range as input, and replace text within range with the output of the command
`<range>sort`	sorts the lines in `<range>`
`read <file>`	reads the content of `file` and inserts after current line
`read !<cmd>`	runs `<cmd>` in a shell and inserts the output after current line

- And one particularly powerful command to substitute `<pat>` to `<text>`:

  `<range>s/<pat>/<text>/<flags>`

  - If `<range>` is omitted, only substitute on the current line.
  - If `<flags>` contain `g`, substitute on all occurrences on a line. Otherwise only substitute the first occurrence.
  - If `<flags>` contain `i`, ignore case for pattern.

- For the commands above, `<range>` can be:

`%`	the whole file
`'<,'>`	the current selection. Appears automatically when you press `:` in visual mode
`<a>,<b>`	from line `<a>` to line `<b>` (inclusive)