Lab 03: Linked Lists

COP 3503

October 13, 2014

A First Object

Lecture 05

```
// a very basic C++ object
class Person
{
	public:
		Person(string name, int age);

	private:
		string name;
		int age;
}
```

- We still have another problem.
  - –How can we actually make use of the class’s data?

Encapsulation

- Since we’ve set the class fields to private, it is necessary to implement some way of accessing its information – one that does not expose the fields.
  - The solution? *Accessor* methods.

A First Object

```
string Person::getName()
{
    return this->name;
}

int Person::getAge()
{
    return this->age;
}
```

A First Object

```
string Person::getName()
{
    return this->name;
}
```

- Suppose we had a “Person p”. The line “p.getName()” would return the value for “name” from the object represented by “p”.

Encapsulation

- First, note that these accessor methods will be set to public – otherwise, they won’t be of use to code outside of the class.
- Secondly, these methods retrieve the data without allowing it to be changed.
  - **In Java**, String’s implementation does not allow its internal data to be changed. C++ differs on this point.

Encapsulation

- What if we need to be able to change one or more of the fields of a class instance?
  - The (first) solution: *mutator* methods.
  - These provide an interface through which outside code may *safely* change the object’s state.

A First Object

```
void Person::setName(string name)
{
    this->name = name;
}

void Person::setAge(int age)
{
    this->age = age;
}
```

Encapsulation

- Is this necessarily the correct solution, though?
  - It depends on the purpose for class design.
- Note that we allow both the “name” of our “Person” and his/her “age” to change, freely.

Encapsulation

- Should we allow a “Person” to change his/her name?
  - It does happen in the real world, but for simplicity, let us suppose that we do not wish to allow people to change names.
  - In such a case, we should remove the setName() method.

Encapsulation

```
void Person::setName(string name)
{
    this->name = name;
}

void Person::setAge(int age)
{
    this->age = age;
}
```

Encapsulation

- However, we shouldn’t stop here. If we wish to make sure that a person may *never* have their name changed, can we make sure that even code from within the class may not change it?
  - Yes: use the const keyword.
  - In Java: “final”.

Encapsulation

```
class Person
{
    private:
        const string name;
        int age;
}
```

- When a field is marked as const, it can only be initialized in a special part of the constructor.

Encapsulation

```
Person::Person(string name, int age)
:name(name)
{
    //this->name = name;
        /* This line would be
           a compiler error!  */

    this->age  = age;
}
```

Encapsulation

```
Person::Person(string name, int age)
:name(name)
{
    //this->name = name;
        /* Thi
            a co

    this->ag
}
```

This is the only valid way to initialize
a **const** variable.

Encapsulation

- Should we allow a “Person” to change his/her age?
  - –Last time I checked, everyone ages.
  - –However, note that a person’s age *cannot* change freely.
  - –Nobody ages in reverse.
  - –A person can only add one year to their age, well, every year.

Encapsulation

```
void Person::setName(String name)
{
    this->name = name;
}

void Person::setAge(int age)
{
    this->age = age;
}
```

Encapsulation

```
void Person::haveABirthday()
{
    this->age++;
}
```

Encapsulation

- At first, encapsulation may seem to be unnecessary.
  - It does add extra effort to using values that you *could* just directly access instead.
  - However, someone else might not know how to properly treat your object and may mess it up if you don’t encapsulate.

Encapsulation

- There are other benefits to encapsulation.
  - What if you later realize there’s an even *better* way to implement your class?
  - You can provide the same methods for accessing object data while changing its internals as needed.

Encapsulation

- Is our current implementation of age “the best”?
  - A possible alternative: track birthdays instead!
  - Birthdays only come once a year, after all, and at known, preset times.

Analysis

- Note that the “inputs” to an object are managed through its constructors and mutator methods.
- The “outputs” are managed through its accessor methods in such a way that the “constraints” are still enforced.

Lecture 06

Value Types vs. Reference Types

- Master the pointers in C++/C
- Importance of memory managment

Memory + Arrays

- An *array*, when actually utilized during execution, is a large, contiguous (undivided) block of memory.
- The array’s starting location – its *address* within memory – is then stored for future *reference*.
  - All of its data can be found given this starting reference and indices.

Memory + Arrays

- The “first” (typically, index “0”) element of the array is stored directly at the starting address of the array.

- Each subsequent element is then stored at a constant offset from this address.

Working with Data in *Java*

- *Reference types* within Java are ***all*** "classes"/"objects," and vice-versa.
  - Only the primitive value types are treated "by value" in Java.
- Note that these objects, or classes, may be composed of multiple value types.
  - These values must be obtained *through* the whole object's **reference**.

Working with Data in C++

- In C++, the programmer may choose which way to handle data.
  - Objects and arrays may be handled by value (within a function) *or* through a pointer.
  - While primitive types default to “by value,” they may be handled by reference!

Working with Data in C++

- By default, most types in C++ will be treated as if they were direct values.
  - The following code will handle both variables “by value.”

```
int i = 4;
Person p(“Harrison Ford”, 72);
```

  - Note the form of the constructor call here – it’s for a “by value” class instance.

Working with Data in C++

- Conversely, any type in C++ can be referred to via a pointer.
  - The following code will handle both variables “by reference.”

```
int *i = new int(4);
Person *p =
   new Person(“Harrison Ford”, 72);
```

- The ‘*’ symbol denotes that the variable stores a *pointer* to that type.

Working with Data in C++

- If a pointer is not presently referring to any active object, it should ***always*** be set to “null” – the address zero (0).

```
int *i = 0;
Person *p = 0;
Person *q; // WARNING: q is not
           // pointing to null!
```

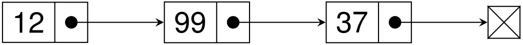
Working with Data in C++

- A pointer can be obtained for *any* value – including pointers!
  - This is done with the & operator.

```
Person p(“Harrison Ford”, 72);
Person *pPtr = &p;
Person **pPtrPtr = &pPtr;
// Yes, pointers to pointers
// are completely legal.
```

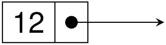
Definition

- Linked lists are a popular and simple data structure
- They are made up of a bunch of *nodes* linked together
- Together these nodes represent a sequence of some sort of *data*



The Node Structure

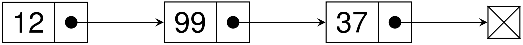
- A node is made up of some value and a pointer to the next node in the list. The data can be a primitive type or even a pointer to another class!



node

The above node has a value of 12 and presumably points to another node.

- If no node follows a given node, we point that node to *NULL* or *nullptr*. This usually signfies the end of the list

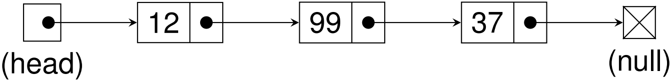


(null)

The above list starts with 12 and ends with 37

The List Structure

- The only thing we need to store for a list is a pointer to the first node
- Once we have this, the idea is to follow the next pointers until you hit the NULL pointer, in which case you know the list is over
- This is typically known as the head of the list



In Code

- We usually break the linked list into two classes: Node and LinkedList.

```
class Node{
  private:
    typename data; // Typename is any data type
    Node * next; // Notice it is a pointer!
  public:
    // Methods and Stuff
}

class LinkedList{
  private:
    Node * head; // Again, we have a node pointer!
  public:
    // Methods and stuff
}
```

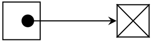
Some Common Methods

- Most methods with linked lists can be done either recursively or iteratively.
- Common methods are:
  - Insert - Add a value to the list
  - Remove - Remove a value from the list
  - Traverse - Traverse the list (visit every single node), doing something useful at each node
  - Member - Check to see if a given value is in the list
  - Split - Split a given list into two seperate lists

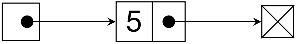
Insert

At a high level, inserting a value just involves finding the end of a list and pointing it to a new node:

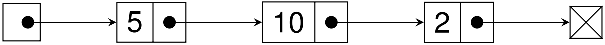
```
LinkedList * ll = new LinkedList();
```



```
ll.insert(5);
```



```
ll.insert(10);
ll.insert(2);
```



Insert Code: Two Ways

Recursive:

```cpp
void LinkedList::insert(int value){
  if(head == NULL) {  // List is empty
    head = new Node(value);
    return;
  }
  head->insert(value); // recurse starting with head
}

void Node::insert(int value){
  if(next == NULL) { // Base case: value goes after me!
    next = new Node(value);
    return;
  }
  next->insert(value); // I'm not the end of the list.
                       // Keep going
}
```

Insert Code: Two Ways

Iterative:

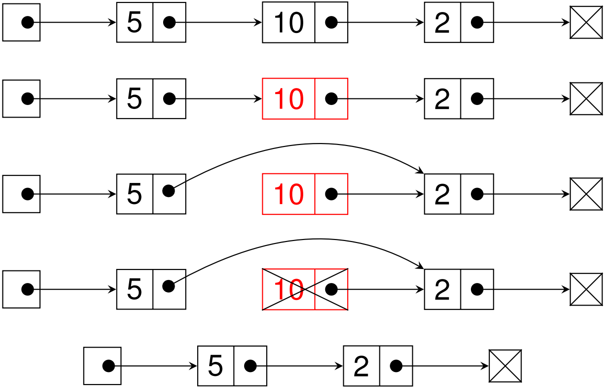
```
void LinkedList::insert(int value){
  if(head == NULL) {  // List is empty
    head = new Node(value);
    return;
  }
  Node * temp = head;
  while(temp->next != NULL) { // While we aren't at end
    temp = temp->next;
  }
  // temp now points at the final node in the list
  temp->next = new Node(value);
}
```

Remove Overview

- Base case: Head is null. There is nothing to remove, so the function ends!
- Find the node you want to remove (either iteratively or recursively).
- Once found, you point the node's predecessor's pointer to the node's successor (i.e. previous->next = current->next).
- Then, you must *delete* the node you removed!

Example

```
ll->remove(10);
```



Command Line Arguments

- Command line arguments are a way to pass information to your program when you run it about how it should operate.

  ```
  g++ -c list.cpp -o list.o
  ```

- g++ is a program, just like any that you write
- The other things on the line are arguments telling g++ how it should run
  - -c means just compile the source code into an object file
  - -o tells g++ what file to stick the result of its compilation in
  - The other two (list.cpp and list.o) are filenames passed to the program
- In your project, you need to take in an argument that can either be best or worst

Command Line in C++

- In C++, we use paramters passed to main to get command line arguemnts

```cpp
int main(int argc, char ** argv){
  for(int i=0; i < argc; i++)
    cout << "argv[" << i << "] = " << argv[i] << endl;
  return 0;
}
```

- argc is the number of command line arguments passed to the program

- argv is an array of strings that contains each argument

- Note that the name of the executable is the first string passed in the program, so argc is always >= 1

Example Runs

```
int main(int argc, char ** argv){
  for(int i=0; i < argc; i++)
    cout << "argv[" << i << "] = " << argv[i] << endl;
  return 0;
}
```

```
> ./a.out one two three
argv[0] = ./a.out
argv[1] = one
argv[2] = two
argv[3] = three
> ./a.out worst
argv[0] = ./a.out
argv[1] = worst
```

Other Things

- The strings passed to your program are called cstrings. This is different than the string type in C++
- When using cstrings, you need to use functions like strcmp (for comparison), strlen (for length of a string), and strcat (to concatenate strings). These can all be found in the library <cstring>
- It is important that you check the arguments passed to your program are valid arguments. In this case, that means that there is only one and it is either best or worst

Exercises

Implement a linked list class with the methods we discussed during the lab:

- Insertion
- Removal
- Traversal (print the list out)
- Search

Questions

???