CISC 3130
Project 1

Lists like Lisp

Introduction

The programming language Lisp (originally called LISP) is nearly 60 years old (the second-oldest high-level language that we're still using). Originally, its name was an abbreviation for "LISt Processor"—in the early forms of the language, linked lists were its primary composite data type (just like arrays and structures are the primary composite data types in C++). These days, variants of Lisp (like Scheme and Clojure) are a bit more complex... Scheme is used as the introductory programming language at many colleges; Clojure has substantial adoption in industry; and everyone's favorite editor is written (almost entirely) in its own dialect, Emacs Lisp. In this project, you'll add some behaviors to the book's `SLList` to make that class act a bit more like Lisp lists.

Note: you don't need to read/understand any of the Wikipedia pages linked above to do this project; they're just there for your information/curiosity.

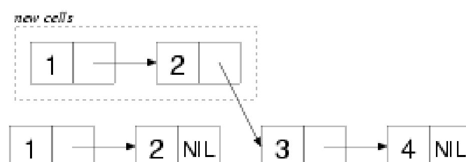
Implementation

First, make a copy of the book's `SLList` files and rename them (and the class) to `LispList`. Be sure to leave a comment in both files crediting the author and linking to the code source. Your code should not belong to the `ods` namespace, so remove those directives. (And be sure to fix the `Node` constructor so that the value is set to `x0`, not `0`!)

Before we add any exciting new behaviors, we need to be able copy lists (or parts of lists). First, write a `protected` (helper) method `copy(Node *n)` which returns a new `LispList` that is a copy of the "sublist" that starts at `n` (and ends at `NULL`). Use this helper method to write a `LispList` copy constructor and overloaded assignment operator (`operator=`).

Lisp provides two basic operations on lists: **first** and **rest**. **first** returns the *value* of the first element in the list; **rest** returns a list containing all the values except the first (that is, the rest of the list). Note that from this description, you can't tell whether the return value of rest should contain new nodes or just use existing nodes. For this implementation, do not create any new nodes.

Lisp provides an **append** operation that returns a new list containing the elements of two other lists. So **append** applied to (1, 2) and (3, 4) would return (1, 2, 3, 4). How many new nodes should append create, if any? It can't use the original nodes from the first argument (why not?), but it *can* use the original nodes from the second argument:



So, implement that operation.

Finally, the **map** operation applies a function to each element of the list and returns a new list containing the return values. So applying **map** with the x^2 function to (1, 2, 3, 4) would result in (1, 4, 9, 16). Or, applying **map** with `std::string::size()` to ("dog", "cat", "elephant") would result in (3, 3, 8).

Implement this operation! But this is going to take some C++ techniques, discussed below.

Finally, implement a `print()` method that prints the contents of a list to `cout`. Use the syntax I've been using, where the list is enclosed in parentheses and the elements are separated by a comma and a space. (You don't need to print out the quotes around strings, so `(dog, cat, elephant)` is fine.) Be sure the last element is followed directly by parentheses!

Your Main Program

Your main program will conduct some limited testing of your `LispList` behaviors. It should expect input from `cin` like this (the stuff after // are my descriptions; it's not part of the input!):

4 // a positive int value
dog // then a batch of std::string values of that size
cat
elephant
zebra
3 // a non-negative number of std::strings to read (another batch of strings)
Brooklyn
Queens
Manhattan

You should store those 3 sets of input in 3 `LispLists`, of course.

The program's output comes in two sections. First, for each of the `strings` in the first batch, output the initial substring with length given by the initial `int` input. (Of course, do this with the `map()` method.) In this example, the output would be:

dog
cat
elep
zebr

Then, using the `append()` method and the `map()` method, output the size of all the strings provided in input. In this example, the output would be:

3
3
8
5
8
6

There should be no blank lines in your output (unless one of the strings was empty).

What to Turn In

Email me a zipfile named familyname_givenname.zip. (So I would send `dexter_scott.zip`). You may also send a `.tgz` file if you're using Unix/Linux. Do not send a RAR or other kind of compressed file; those will get a grade of 0 points. Make sure the subject of your email is *CISC 3130 Project 1.*The zipfile should contain:

- Your makefile, which produces an executable file called `project1`
- All `.h` and `.cpp` files in your project.
- Two textfiles containing test input and the resulting output, named familyname_givenname.in and familyname_givenname.out.

I should be able to extract files from your zip, run `make`, and run `project1`. If any part of that fails, you will not earn full points. I will also be providing my own test data: if your program does not produce the expected output, you will not earn full points. I will also be compiling your LispList class against my own main program and running additional tests (e.g. of your methods, using other types of data). If your program does not compile, run, and produce the expected output, you will not earn full points.

On Style

You do not need to change the style of the code from the book. But your code should follow best-practice style guidelines—probably [Google's C++ Style Guide]() is the most comprehensive. Pay particular attention to the sections on Naming, Comments, and Formatting. If I have to work hard to figure out what you're trying to do, you will not earn full points.

Implementing the `map` Method

The idea of passing a function as a parameter to another function is a pretty old one, dating back at least to the C language. The basic idea is that a function is really just a memory location—the address in memory where the function's machine code is stored. So passing a "function pointer" isn't much different from passing any other kind of pointer. In C++, this idea has developed along with the facilities for generic and polymorphic code. In particular, the [`functional`]() library of the STL includes a templated class `function` that can represent pretty much anything that can be called like a function. For example, if I've declared a function

```
void print_num(int i) { std::cout << i << '\n'; }
```

then I can use that to initialize a `function` object:

```
std::function<void(int)> f_display = print_num;
```

where the `void` indicates the return type of the `function`, and the types listed in the `()` indicate the parameter types of the `function`. Then I can use `f_display` just like a "regular" function:

```
f_display(-9);
```

Of course, this is not a very interesting application of this idea! More interesting is to write a function that can take a `function` parameter:

```
void foo(std::function<int(int)> func) {
     for (i=0; i<100; i++)
          std::cout << func(i) << std::endl;
}
```

Then when you call `foo()`, you can pass the right kind of `function` object (just make sure you actually pass a `function` object—like `f_display` above—rather than the name of an "actual" function—like `print_num` above—or else the compiler will be grumpy).

The great thing for this project is that the types inside the `function<>` can be templated; you'll probably have to think a *little* about how to use this fact in your code. In general (ignore templates), the prototype for map should be

```
SLList map(function f, SLList l)
```