Lab 4: On Lists and Light Sabers

Due: March 19th at 11:59pm

Overview

The goal of this lab is to familiarize yourself with the usage of Lists and their implementations, Array List and Linked. To do so, you will complete an ascii-mation movie player first using an Array List, implementing an Array List Iterator, and then implementing a Linked List and a Linked List Iterator. Of course, along the way, you'll enjoy some choice scenes from Star Wars. **This is group lab, groups of two, but you must work with someone in your lab section.**

Deliverables

Your submission should include requisite code necessary to run your player, but the following files will be the focus of grading:

- `Makefile`
- `README`
- `player.cpp`
- `movie.[c|h]`
- `string_list.h`
- `array_string_list.cpp`
- `linked_string_list.cpp`
- `iterator.[c|h]`
- `test_list.cpp`

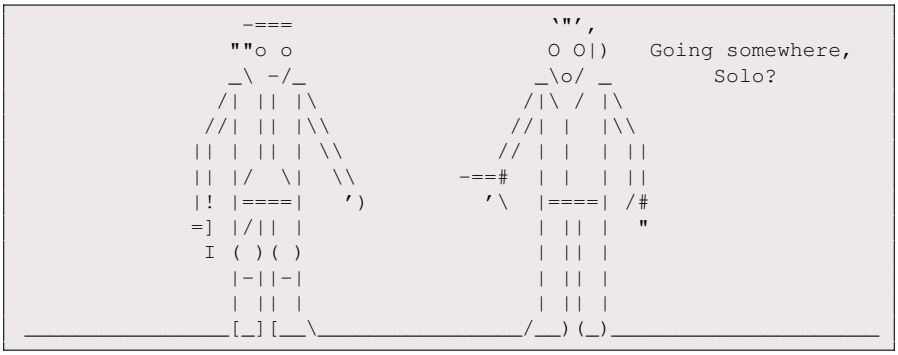
All starter code is provided via `update35`.

Submission

You will submit using `handin35`. Be sure to place all relevant files in `cs35/labs/04`. If you are attempting the extra credit, you should submit a sub-folder labeled `ExtraCredit` within the `labs/04` folder.

ASCII-Mation Star Wars Player

This lab is inspired by a late-90s Internet phenomenon known as ASCII-mation (google "ASCIImation" for more examples). Most of you are familiar with ASCII art, where basic characters and symbols are pieced together to represent an image. ASCII-mation takes this a step further by placing a series of ASCII images in animation. For example, here is one frame of the movie:



By iterating over frames, the action comes to life (and Han shoots Guido).

In this lab, each frame of the movie will be stored as a string within a List. The challenge is to complete the movie player by first using the provided Array List implementation of a List, and then by implementing a Linked List implementation of a List. In addition, you will implement List Iterators for each implementation type so that you will have efficient playback regardless of the list implementation.

Starter's Abstract This is a fairly involved lab that requires reading and accessing multiple source/header files. There is a lot going on, and it can be overwhelming at first. So, before we going through the lab in detail, here is an abstract of the required work in the form of a plan of action. (Note that a complete description of all provided files can be found at the end of this document.)

1. Implement the Array List Iterator in `iterator.cpp` so that basic movie play back works.
2. Complete the additional playback functionality for `player.cpp` by editing movie.cpp
3. Add test to `test_list.cpp` to make sure your Array List Iterator works as you intend. Even if the movie plays back, it's worth doing the tests (and is part of your grade).
4. Implement the Linked List in `linked_string_list.c`, test its functionality in `test_list.cpp`.
5. Implement the Linked List Iterator in `iterator.cpp` and add test code in `test_list.cpp`.
6. Switch over the iterator in `player.cpp`, pop some popcorn, and enjoy.
7. (**Extra Credit**) Implement Template forms of each the list implementations and integrate into the movie player.

Finally, this is the first group lab, and here is some basic advice on group work. Happy groups meet early and often. **Working together means coding together**: that doesn't mean you must work on the same computer, although that works for some, but rather work in the same room at the same time. You'll find life much easier when you have someone you can bounce an idea off of — your lab partner. You must trust your partner and you must work together. If you would like to use a version control repository, please let me know. Both GIT and SVN are available.

Lists

We will use `string` based lists throughout this lab. That is, the lists are designed to hold a single data type, a `string`. Since each frame in the movie is represented as a string, this makes sense; however, a more general list can be created using templates, which is a task for extra credit.

The abstract list class, `StringList`, is defined in `string_list.h`. In addition, the class definitions for `ArrayStringList` and `LinkedStringList`. To complete the `LinkedStringList`, you will need to define an additional class, a `StringNode`, which stores a single string and is lined together to form a list.

Iterators

An Iterator is a special class used to efficiently iterate over a list. Below is the abstract class deffinition, which can be found in `iterator.h`:

```
/**
 * Abstract Iterator Class for iterating over a StringList
 *
 **/
class StringListIter{

 public:

  virtual ~StringListIter(){/* do nothing */}

  /**
   *Checks if there is more items to iterate over
   *
   *
   *@return bool : true if iterator has ended, and false otherwise
   *
   **/
  virtual bool end() = 0;

  /**
   *Increment the iterator to the next value
   *
   *@error runtime_error: no more items in the list
   *
   **/
  virtual void incr() = 0;

  /**
   * Retrieve the value referenced by this iterator
   *
   *@return string : the list item the iterator points to
   **/
  virtual string get() = 0;

  /**
   * Retrieve the current index of the iterator
   *
   *@return int: the current index of the iterator
   */
  virtual int curIdx() = 0;
};
```

The reason you need an iterator is that depending on the implementation, iteration over a list can be grossly inefficient. For example, consider that `get(i)` for a Linked List requires traversing from the head

of the list `i` iterations, while this only requires a single access for an Array List. Most operations over list require iteration, e.g., using a for or while loop, and a key part of the List data structures is providing efficient iteration that is implementation agnostic. **You are required to implement both an Array List Iterator and Linked List Iterator to iterate over the list with $O(n)$ operations, where n is the size of the list**. That is, the following loop should be equally costly for both implementation types:

```
StringListIter * iter;
/* Instatiate either ArrayListIter or LinkedListIter */

string cur;
while(! iter->end()){
   string cur = iter->get()

   /* Do something totally awesome! */

   iter->incr();
}
```

As a comparison, consider what happens if you were to use standard retrieval `get(i)` with a Linked List, like as follows:

```
LinkedStringList * list = new LinkedStringList();
/* Propogate the list with a bunch of strings */

for(int i = 0; i < list->getSize() ; i ++){
   string cur = list->get(i);

   /* Do something really crazy! */
}
```

Not only do you iterate over the indexes, but each `get(i)` requires an iteration of the nodes of the Linked List. A seemingly $O(n)$ operations is actually $O(n^2)$. A well designed iterator for Link Lists will avoid this peril, and is a major part of this lab assignment.

Switching from Array List to Linked List implementation

There is one line of code that needs to change to switch between implementations. If you implement everything well, then your Linked List should function as a complete drop in replacement. Here is the line of code in `player.cpp` (line number 38):

```
//TODO: Eventually Switch to LinkedStringList()!!
//StringList * movie = new LinkedStringList();
StringList * movie = new ArrayStringList();
```

Testing

Developing reasonable tests for your Iterators and Linked List implementation is a required part of this lab, and **you must describe your testing strategy in your** `README`. I've provided a series of tests for the Array List implementation in `test_list.cpp`, and you should adapt and expand on the testing environment to include tests for new parts of the lab. The test library also makes extensive use of the `assert()` function which checks the validity of its argument, e.g. `assert(1 == 2)` fails, and if the argument is not valid, the program stops executes and prints a useful message. I strongly recommend you use `assert()`. You can compile `test_list.cpp` with `make test`.

Grading Rubric

This lab will be graded on a 30 point scale:

- (5 points) Array List Iterator
- (15 points) Linked List Implementation and Iterator
- (5 points) Testing
- (5 points) Documentation, README, and Coding Style

Additional deductions may occur for memory errors and other standard coding practices.

Extra Credit and Challenge Exercises

- **Extra Credit 1 Point**: Add scene jumping functionality to the player so that people can jump ahead by 5 seconds, 30 seconds, and 1 minute. You should require the user to pause first, but if you're interested, feel free to mess around with `khibit.h`, as long as you don't break anything else.
- **Extra Credit 1 Point**: Add a new option to the player that allows for cut scenes to be saved to a file. You must use the appropriate ascii-mation format so that your cut scene can be played using the player.
- **Extra Credit 3 Points**: Create a **new copy** of the lab assignment where all `string` based data structures are replaced with template data structures. Partial extra credit will be given, but only for working versions.

Provided Code

Below is the provided source and header files and short description of their purpose.

- `player.cpp` : The `main()` portion of the code base. You will need to make minimal edits to this file, just to switch from an Array List to a Linked List.
- `string_list.h` : The header file where all the List classes are defined. You will need to make edits to this file when defining an Linked List.
- `array_string_list.cpp` : The Array List source code, which is provided for you and presented in class. You will not need to edit this file, but you should review it.
- `linked_string_list.cpp` : The Linked List source file. You will need to make extensive edits to this file.
- `test_list.cpp` : A testing file. Some test are already provided, but you are required to add additional tests.
- `iterator.h` : The class definition for the iterators. You should review this file and make minor edits to the Linked List Iterator defined at the bottom.
- `iterator.cpp` : The source file for the iterators. You will need to make extensive edits to this file for both the Array List Iterator and the Linked List Iterator.
- `movie.h` : The header file where moview functionality is defined. You should review but not edit this file.
- `movie.cpp` : The source file for movie functionality. You will need to make edits to this file for key functions.
- `kbhit.h` : Don't touch this file. It's magic.
- `Makefile` : You may, but are not required, to edit the Makefile.
- `clips` : A directory storing clips to use in your movie player.

Compilation

Compilation will occur using `make`. To compile the player"

```
bash> make
```

If you add extra functionality contained in other files, you should edit the `Makefile`, but it is not necessary. The result of that compilation will be executable, `player`.

To compile the test code:

```
bash> make test
```

Which will compile an executable `test_list`. Again, you may adjust the `Makefile` to include other files to produce other testing executable.

Execution

To play a movie, you use the `player` executable as follows:

```
bash> player movie.ani
```

The player will not execute without a movie file. You can find movie files in the clips directory.