Fun With Linked Lists

This handout was written by Julie Zelenski and Jerry Cain.

This material is introduced in Section 12.2 of the reader. Eventually we'll see that linked lists (and more generally, linked structures) are used to back the **`Queue`**, the **`Set`**, and the **`Map`**). But linked lists exist outside the container framework, so it's important we understand the mechanics involved in building, iterating over, and otherwise manipulating them.

Simple Linked Lists

First, here's our node definition:

```
struct entry {
   string name;
   string address;
   string phone;
   entry *next;
};
```

Here are the basic operations for creating and printing a single **`entry`**:

```
static entry *createEntry() {
   string name = getLine("Enter name (or return to quit): ");
   if (name.empty()) return NULL;
   entry *ent = new entry;
   ent->name = name;
   ent->address = getLine("Enter address: ");
   ent->phone = getLine("Enter phone: ");
   ent->next = NULL; // initialize field to show no one follows
   return ent;
}

static void printAddress(const entry *ent) {
   cout << ent->name << endl;
   cout << ent->address << endl;
   cout << ent->phone << endl;
}
```

Let's start with the unsorted version of the linked list of phone book entries. In creating the list, we prepend each new entry to the front of the list, since that's pretty easy to do.

```
static entry *createAddressBook() {
   entry *book = NULL;
   while (true) {
      entry *ent = createEntry();
      if (ent == NULL) return book;
      ent->next = book; // set rest of list to follow new entry
      book = ent;       // new entry becomes the head of list
   }
}
```

Note that in freeing the list we have to be careful to not access an entry after we've freed it.

```
static void freeAddressBook(entry *book) {
   entry *curr = book;
   while (curr != NULL) {
      entry *next = curr->next;  // save embedded next pointer
      delete curr;               // free entry surrounding it
      curr = next;               // advance curr to next entry
   }
}
```

We can use the idiomatic **for** loop, linked list traversal to print each address:

```
static void printAddressBook(const entry *book) {
   for (const entry *curr = book; curr != NULL; curr = curr->next) {
      printAddress(curr);
   }
}
```

A brute-force linear search can be used to look up an entry by name:

```
static entry *findEntry(entry *book, const string& name) {
   for (entry *curr = book; curr != NULL; curr = curr->next) {
      if (curr->name == name) {
         return curr;
      }
   }

   return NULL;
}
```

Now, let's reconsider the decision to maintain an unsorted list and instead work to keep the list of entries in alphabetical order. With this change, we can rewrite our **findEntry** function to be a bit smarter. We use **string::compare** to determine if we have an exact match between the name of interest and the name within the current entry. We can also determine if we've passed where the name of interest would need to be, and bail early if we know we're never going to find it:

```
static entry *findEntryInSorted(entry *book, const string& name) {
   for (entry *curr = book; curr != NULL; curr = curr->next) {
      int cmp = name.compare(curr->name);
      if (cmp == 0) return curr; // exact match right here
      if (cmp < 0) return NULL;  // passed position & didn't find it
   }

   return NULL;   // finished off list and didn't find it
}
```

Here's the tricky part. We need a function that'll build the list up while preserving alphabetical ordering. The simple prepend-to-front approach won't work here, as we need to splice the entry somewhere into the middle of the list. The loop we wrote for **findEntry** can find the appropriate position for us, but it'll go one beyond where we need to stop. ☹ After the loop, **curr** points to the entry that will follow **ent** in the list. Given the forward-

chaining nature of linked lists—at least as they're currently defined—we have no easy way to get to the entry behind to the one identified by **findEntry**.

How about this? We'll maintain two pointers while walking down the list, using the second pointer to track the previous entry. After the loop, **prev** will point to the entry that should precede **ent**, and **curr** will point to the one that should come after it. We need to splice **ent** right in between these two. This means attaching **curr** to follow the **ent** and setting **ent** to follow **prev**. It's certainly possible that **prev** is **NULL** (when **ent** is inserted at the head of the address book), and we need to handle that case specially. Since this will require changing the head of the entire list, we need to pass the head pointer by reference, necessitating the **entry *&**!

```
static void insert(entry *& book, entry *ent) {
   entry *curr;         // needs to live beyond for loop, so declare here
   entry *prev = NULL;  // first entry has no predecessor
   for (curr = book; curr != NULL; curr = curr->next) {
      if (ent->name < curr->name) break; // found place!
      prev = curr;
   }

   // now, "prev" points to one before ent, "next" is right after
   ent->next = curr; // works even if curr is NULL
   if (prev != NULL)
      prev->next = ent;
   else
      book = ent;     // note the special case!
}
```

Now's a good time to think through the special cases and make sure the code handles them correctly. What happens if the entry is inserted at the very end of the list? What about at the very beginning? What if the list is entirely empty?

Here's how we would call the insertion function to build a sorted address book:

```
static entry *buildSortedBook() {
   entry *book = NULL;
   while (true) {
      entry *ent = createEntry();
      if (ent == NULL) return book;
      insert(book, ent);     // note book is passed by reference!
   }
}
```

Deleting is also tricky. We need to find the entry to delete and then carefully splice it out of the list and free its memory. Again, we're going to carry two pointers down the list to find the entry to delete and the entry that precedes it. If we don't find the entry at all, we bail. Once we have the entry and its previous neighbor, we can wire up everything to circumvent the entry being killed. Again, we have the special case of deleting the first entry in the list.

```
static void deleteEntry(entry *& book, const string& name) {
   entry *curr;
   entry *prev = NULL;
   for (curr = book; curr != NULL; curr = curr->next) {
      int cmp = name.compare(curr->name);
      if (cmp == 0) break;       // found it
      if (cmp < 0) return;       // passed position, didn't find it
      prev = curr;
   }

   if (curr == NULL) return;     // we never found it
   if (prev != NULL)
      prev->next = curr->next;
   else
      book = curr->next;   // recall that list is a reference ☺
   delete curr;            // free all memory associated with this entry
}
```

You might note that find, inserting, deleting all start with a very similar loop, and a good instinct is to want to unify them into a helper function. This function could be given a list and a name and would find the relevant entry within the list. It could return the two surrounding entries by reference and use them to do linked list surgery.