OUR LINKED LIST LIBRARY

LECTURE 05-2

JIM FIX, REED COLLEGE CS2-F20

ADMINISTRIVIA

Grading:

- I've decided to use "paper grading" to catch up on giving feedback.
- Homework 02 feedback will be posted today.
  - Look for a `hw02-yourname.pdf` inside a *feedback* branch of your repo.
- Homework 03 will be posted by Wednesday.

Lab tomorrow:

- You'll work in pairs on several linked list exercises.
- Take turns coding: one leads with their editor, the other watches and helps.

TODAY'S LECTURE: OUR LINKED LIST LIBRARY

I've organized Wednesday's lecture into a **`lec05-1`** repo on GitHub.

- Two files:
  - **`llist.hh`**: a "header file" describing the structs and the functions
  - **`llist.cc`**: the "implementation file" giving the code for each function
- Also, a **`Makefile`** and a testing file **`test_llist.cc`**.

To build:

```
make test_llist
```

To run: enter a command like

```
./test_llist 5 7 3
```

SEPARATE COMPILATION

- The Makefile describes the "build rules" for making the `test_llist` executable.
  - ➡It turns out you don't need to compile everything all at once.
  - ➡Compilation can be done incrementally, separately for different source files.
- Instead can build several "object files."
  - These are compiled, but *unlinked* machine code files.
  - These end in `.o`, e.g. `test_llist.o` and `llist.o`
- The get "*linked* together" to resolve inter-code function references by a line like:

```
g++ -o test_llist test_llist.o llist.o
```

HEADER FILES

- I've split the linked list code into two parts.
  1. A header file **`llist.hh`** containing the **`struct`** definitions; function *prototypes*.

```
namespace llist {
  struct node { ... };
  struct llist { ... };
  ...
  void insertAtFront(llist* list, int value);
  void insertAtEnd(llist* list, int value);
  void deleteFront(llist* list);
  std::string toString(llist* list);
  ...
}
```

NOTE: I've chosen to house these under a **`llist`** *namespace*.

IMPLEMENTATION FILES

- I've split the linked list code into two parts.
  2. A corresponding *implementation file* `llist.cc` with the function definitions:

```
#include "llist.hh"
namespace llist {
  ...
  void insertAtFront(llist* list, int value) { ... }
  void insertAtEnd(llist* list, int value) { ... }
  void deleteFront(llist* list) { ... }
  std::string toString(llist* list) { ... }
  ...
}
```

NOTE: the `#include` line essentially pastes in the contents of `llist.hh`

CLIENT CODE

- The files **`llist.hh`** and **`llist.cc`** serve as our linked list library code.
- *Client code* can **`#include`** the **`.hh`**, use our functions. **E.g. `test_llist.cc`**

```
#include <iostream>
#include "llist.hh"

...
int main(int argc, char** argv) {
  llist::llist* theList;
  theList = llist::build();
  ...
    int result = llist::contains(theList,value);
  ...
}
```

NOTE: because of the **`#include`** at the top **`g++`** will know the size of **`llist`** and the type signatures of **`build`** and **`contains`**.

CLIENT CODE

- The files **`llist.hh`** and **`llist.cc`** serve as our linked list library code.
- *Client code* can **`#include`** the **`.hh`**, use our functions. **E.g. `test_llist.cc`**

```
#include <iostream>
#include "llist.hh"

...
int main(int argc, char** argv) {
  llist::llist* theList;
  theList = llist::build();
  ...
    int result = llist::contains(theList,value);
  ...
}
```

NOTE: because of the **`#include`** at the top **`g++`** will know the size of **`llist`** and the type signatures of **`build`** and **`contains`**.

LINKED LIST CODE: BUILDING A LIST

```
// llist::build():
//
// Constructs a new empty `llist::llist` object returning a
// pointer to it from the heap.
//
llist* build(void) {
  llist* newList = new llist;
  newList->first = nullptr; // Initialize it as an empty list.
  return newList;
}
```

LINKED LIST CODE: BUILDING A LIST FROM AN ARRAY

```
// llist::buildWith(values, length):
//
// Constructs a new `llist::llist` object, filling its nodes with
// the integer sequence given by the array `values`, returning a
// pointer to it from the heap.
//
llist* buildWith(int* values, int length) {
  //
  // Build an empty list.
  llist* newList = build();
  //
  // Insert them in reverse order onto the front.
  for (int i=0; i<length; i++) {
    insertAtFront(newList, values[length-i-1]);
  }
  return newList;
}
```

LINKED LIST CODE: INSERT AT FRONT

```
// llist::insertAtFront(list, value):
//
// Adds a newly allocated node onto the front of the linked
// `list`, one housing the given `value`.
//
void insertAtFront(llist* list, int value) {
  node* newNode = new node;
  newNode->data = value;
  newNode->next = list->first; // Next node is the old front.
  list->first = newNode;       // It becomes the new front.
}
```

LINKED LIST CODE: DELETE AT FRONT

```
// llist::deleteFront(list):
//
// Modifies the given linked `list`, removing the node at the
// front. That node's storage is returned to the heap using
// `delete`.
//
void deleteFront(llist* list) {
  if (list->first != nullptr) {
    node* toDelete = list->first;
    list->first = list->first->next;
    delete toDelete;
  }
}
```

LINKED LIST CODE: COMPUTE THE LENGTH OF THE LIST

```
std::string length(llist* list,value) {
  // Returns how many items are stored in `list`.
```

```
}
```

LINKED LIST CODE: OUTPUT THE LAST

```
std::string outputLast(llist* list) {
  // Assuming the list isn't empty, output the int at the end.
```

```
}
```

LINKED LIST CODE: STRING FOR OUTPUT

```
std::string toString(llist* list) {
  if (list->first == nullptr) {
    return "[]";
  } else {
    std::string s = "[";
    node* current = list->first; // List traversal code.
    while (current->next != nullptr) {
      s += std::to_string(current->data) + ", ";
      current = current->next;
    }
    s += std::to_string(current->data) + "]";
    return s;
  }
}
```

LINKED LIST CODE: SEARCHING FOR AN ITEM

```
bool contains(llist* list, int value) {
  node* current = list->first; // List traversal code.
  while (current != nullptr) {
    if (current->data == value) {
      return true;
    }
    current = current->next;
  }
  return false;
}
```

LINKED LIST CODE: DELETE THE LAST

```
std::string deleteEnd(llist* list) {
  // Assuming the list isn't empty, remove the end node.
```

```
}
```

DEBUGGING: LLDB AND GDB

```
std::string deleteEnd(llist* list) {
  // Here, we forget to check if the list has only one item.
  node* follower = list->first;
  node* leader = list->first->next;
  while (leader->next != nullptr) {
    follower = leader;
    leader = leader->next;
  }
  node* toDelete = follower->next;
  follower->next = nullptr;
  delete toDelete;
}
```

- We compile it with the `-g` flag.
- We run it under the debugger, e.g. `lldb test_llist` ...

LINKED LIST CODE: REMOVE THE FIRST OCCURRENCE

```
std::string remove(llist* list,value) {
  // Find a node containing `value`, remove it from the list.
```

```
}
```

NEXT

- ***THIS WEEK:***
  I'll post a Homework 05 due next week.
- ***TOMORROW:***
  You'll write some linked list functions with a partner in the lab meeting.
- ***WEDNESDAY:***
  We'll start Part II. Circuits and Processors
- ***NEXT WEEK:***
  I'll introduce you to Project 1. Uses a bucket hash table as a word dictionary.
- ***TWO WEEKS:*** Midterm #1.