

# OUR LINKED LIST LIBRARY

---

*starting soon!*

**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.

Monday 5-6:30  
Zoom link ← slack Friday

## TODAY'S LECTURE: OUR LINKED LIST LIBRARY

stdio.h

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

↑  
 <stdlib>  
 <string>

▶ Two files:

lec05-1-code

• **l1ist.hh**: a "header file" describing the structs and the functions

• **l1ist.cc** the "implementation file" giving the code for each function

▶ Also, a Makefile and a testing file test\_l1ist.cc.

**To build:**

**make test\_l1ist**

#include "l1ist.hh"

**To run:** enter a command like

**./test\_l1ist 5 7 3**

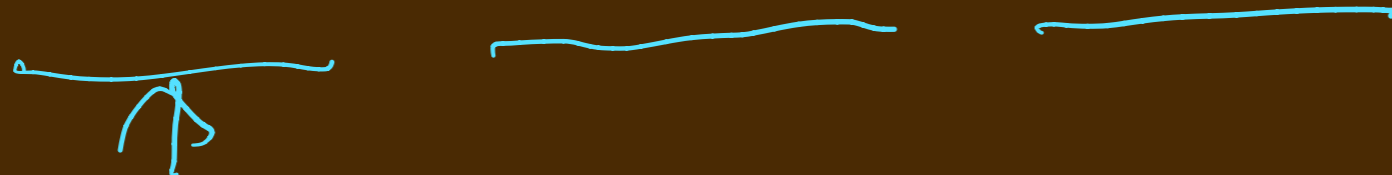
command

command line arguments

## SEPARATE COMPILATION

- ▶ The Makefile describes the "build rules" for making the `test_llist` executable. make
  - 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`
- ▶ They get "*linked* together" to resolve inter-code function references by a line like:

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



```

n = sqrt(x);
int a[n];

```

## HEADER FILES

► I've split the linked list code into two parts.

1. A header file `l1ist.hh` containing the **struct** definitions; function *prototypes*.

```
namespace l1ist {
```

```
    struct node { .. };
```

← single ll. node

```
    struct l1ist { .. };
```

← "cover object"

```
    void insertAtFront(l1ist* list, int value);
```

```
    void insertAtEnd(l1ist* list, int value);
```

```
    void deleteFront(l1ist* list);
```

```
    std::string toString(l1ist* list);
```

function  
← "prototype"

test\_l1ist.cc

l1ist::l1ist } ①

```
}
```

**NOTE:** I've chosen to house these under a `l1ist` namespace.

## IMPLEMENTATION FILES

► I've split the linked list code into two parts.

**2.** A corresponding *implementation file* `l1ist.cc` with the function definitions:

```
#include "l1ist.hh"
namespace l1ist {
    ...
    void insertAtFront(l1ist* list, int value) { ... }
    void insertAtEnd(l1ist* list, int value) { ... }
    void deleteFront(l1ist* list) { ... }
    std::string toString(l1ist* list) { ... }
    ...
}
```

actual code.

**NOTE:** the `#include` line essentially pastes in the contents of `l1ist.hh`

## CLIENT CODE

- ▶ The files `l1ist.hh` and `l1ist.cc` serve as our linked list library code.
- ▶ *Client code* can `#include` the `.hh`, use our functions. **E.g.** `test_l1ist.cc`

```
#include <iostream>
#include "l1ist.hh"
...
int main(int argc, char** argv) {
    l1ist::l1ist* theList;
    theList = l1ist::build();
    ...
    int result = l1ist::contains(theList, value);
    ...
}
```

**NOTE:** because of the `#include` at the top `g++` will know the size of `l1ist` and the type signatures of `build` and `contains`.

# CLIENT CODE

- ▶ The files `l1ist.hh` and `l1ist.cc` serve as our linked list library code.
- ▶ *Client code* can `#include` the `.hh`, use our functions. **E.g.** `test_l1ist.cc`

```
#include <iostream>
#include "l1ist.hh"
...
int main(int argc, char** argv) {
    l1ist::l1ist* theList;
    theList = l1ist::build();
    ...
    int result = l1ist::contains(theList, value);
    ...
}
```

**NOTE:** because of the `#include` at the top `g++` will know the size of `l1ist` 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;  
}
```

```

struct llist {
    node* first;
};

```

```

struct node {
    int data;
    struct node* next;
};

```

```

llist* LL = llist::build(-9, 3);
llist::insertAtFront(LL, 18);

```

```

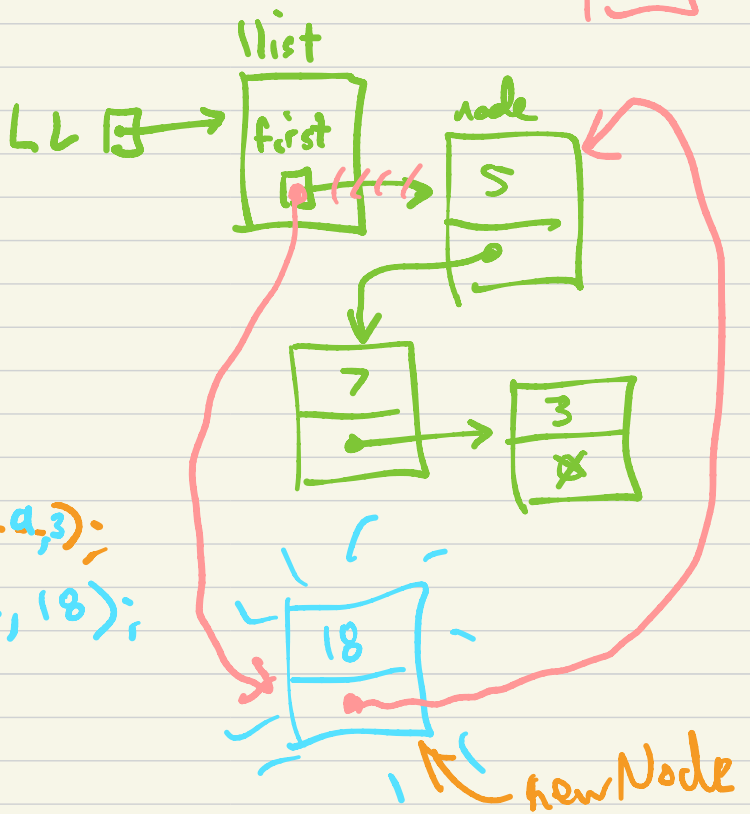
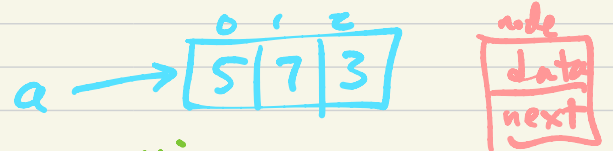
namespace llist {

```

```

void insertAtFront(llist* list, int value) {
    llist* newNode = new node;
    newNode->data = value;
    newNode->next = list->first;
    list->first = newNode;
}

```



```

struct llist {
    node* first;
};

```

```

struct node {
    int data;
    struct node* next;
};

```

```

llist* LL;
LL = llist::build(-9, 3);
llist::insertAtFront(LL, 18);

```

```

unanswar llist {

```

```

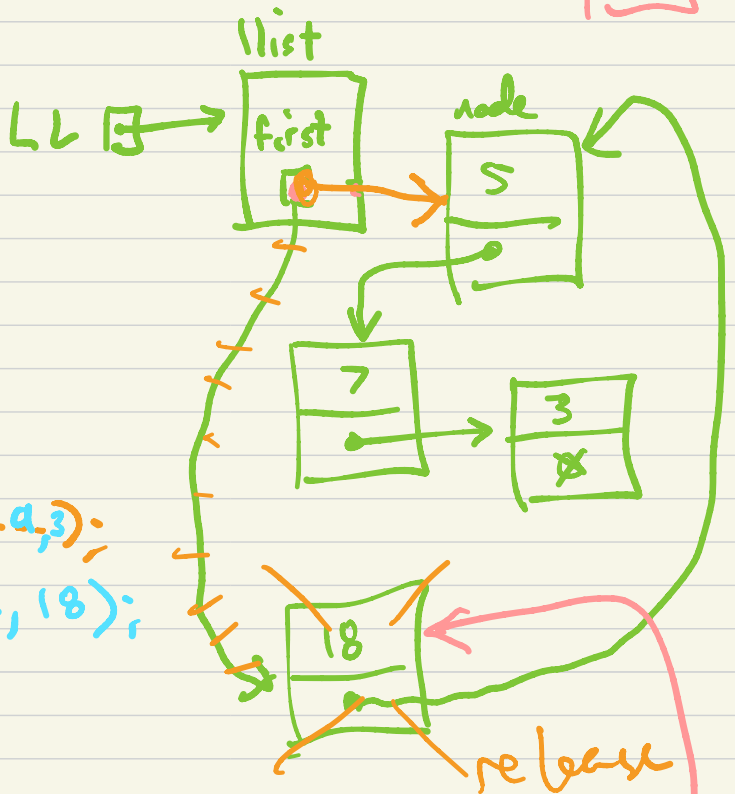
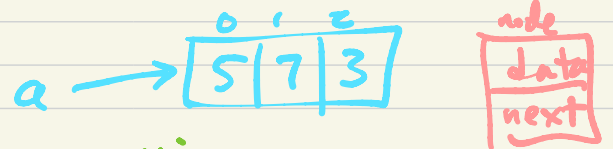
-> void deleteFront(llist* list) {
    node* toDelete = list->first;
    list->first = list->first->next;
    delete toDelete;
}

```

```

if (list->first == nullptr)
    return;

```



```

delete list->first;
list->first = list->first->next;

```

## 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: 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;
}
```



# 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 ...`



# 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.