CSc 352

Using C Structs

Russell Lewis (sub for Benjamin Dicken)

Announcements

- Ben is on paternity leave
  - Back Apr 4
- Changing the “extra” video:
  - Only one extra video, approx 50 minutes
    - Posted Friday
    - https://www.youtube.com/playlist?list=PL-F3IhGTDSSqe5cMDqrLdHkG0bleuA_xq
- My 352 Office Hours: 11am-noon, MWF
  - Live+online (G/S 837) as of today
    - Add yourself to the same Queue, for both online + live

Recap:

- C structs do not allow methods
- Instead, many libraries use the “first parameter” convention:

```
int some_func(Thing *obj, int param1, char *param2);
```

- If object is complex (so init is non-trivial), may have `create()` and `destroy()` methods
  - Akin to the constructor in Python, Java
  - No standard names

(More) Linked Lists in C

- Write `lln_add_tail(head, val)` in C
  - Starting list might be empty
  - Return updated list
  - Use `lln_create(val)` to create a new node

```
typedef struct ListNode {
    int              val;
    struct ListNode *next;
} ListNode;

ListNode *lln_create(int val);
```

```
ListNode *lln_add_tail(ListNode *head, int val)
{
    if (head == NULL)
        return lln_create(val);
    head->next = lln_add_tail(head->next, val);
    return head;
}
```

Java

```
ListNode lln_add_tail(ListNode head, int val)
{
    if (head == null)
        return new ListNode(val);
    head.next = lln_add(head.next, val);
    return head;
}
```

Linked Lists in C

- Write `lln_peek_head(head)` in C
  - Starting list guaranteed not empty
  - Return value of head node, don't change anything
- Write `lln_pop_head(head)` in C
  - Starting list guaranteed not empty
  - Free head node (`lln_destroy()`) and return **new head**
    - Might be NULL

```c
int lln_peek_head(ListNode *head) {
    return head->val;
}

ListNode *lln_pop_head(ListNode *head) {
    ListNode *old_head = head;
    head = head->next;
    lln_destroy(old_head);
    return head;
}
```

Discuss!

- Why did I write two different functions in C, `peek_head()` and `pop_head()` ?

- Could I have done both in a single function?

Discussion: `peek()` and `pop()`

- I needed to return two values:
  - Contents of the head
  - Updated head pointer

- In C, you can do this with “out” parameters, but it’s annoying

- Alternatively, we could have a “List” struct, and handle the nodes like private data

Out Parameters in the Wild

```
#include <stdlib.h>
unsigned long stroul(char *start, char **end, int base);
```

- Standard C function, converts a string to integer
  - Like `int()` in Python, or `Integer.parseInt()` in Java
- Returns the value read
- Also needs to return the *end* of the read (so you can parse more data from this string)

Activity

Out Pointers - Painful Yet Effective

- Write a `lln_get_and_pop_head()` function
  - First parameter (`head`) is an “inout” parameter
    - The user must fill it with a value going in
    - But you will also change it using an “out” parameter
  - Second parameter (`val`) is an “out” parameter
    - Is a pointer - that is, an “out” parameter
    - Value of the variable, before this function, is ignored
    - You will set it with the value of the old head

```
void lln_get_and_pop_head(ListNode **head_inout,
                          int       *val_out) {
    ListNode *old_head = *head_inout;
    ListNode *new_head =  old_head->next;

    *val_out = old_head->val;

    *head_inout = new_head;
    return;
}
```

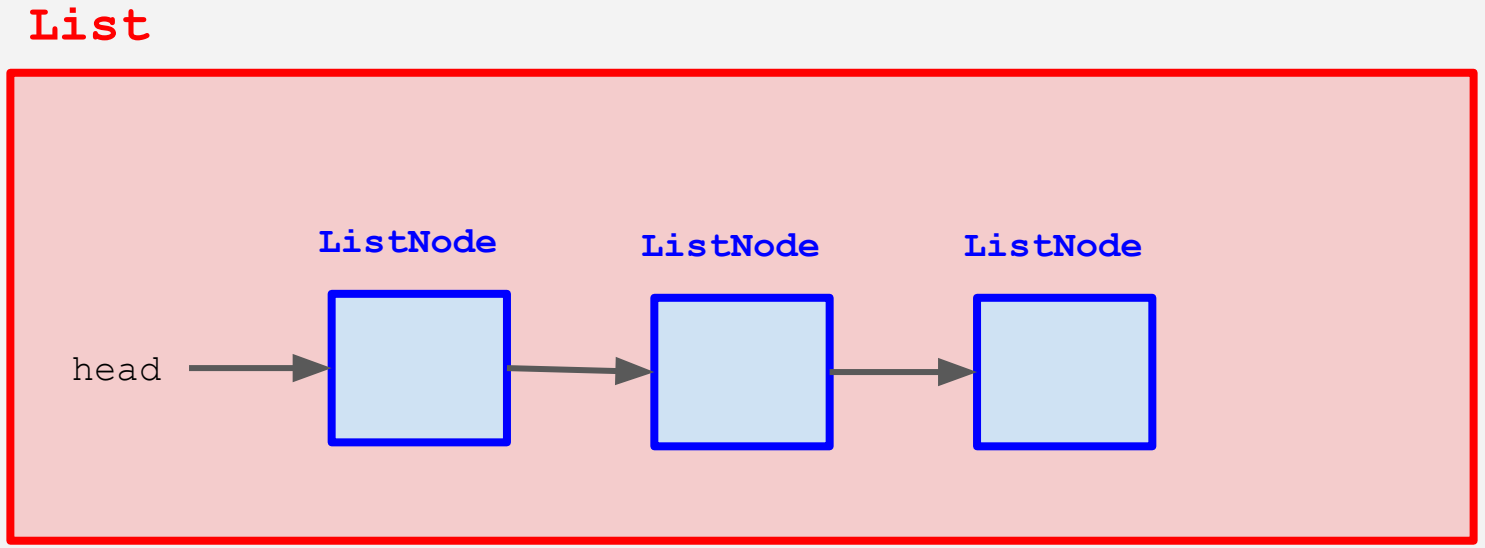
- Ugly!

```
void lln_get_and_pop_head(ListNode **head_inout,
                          int       *val_out) {
    ListNode *old_head = *head_inout;
    ListNode *new_head =  old_head->next;

    *val_out = old_head->val;

    *head_inout = new_head;
    return;
}
```

- Trick for making “out” params easier:
  - Read at beginning
  - Don’t update until the end

List Wrappers

- Some people prefer to have 2 classes for a linked list:
  - One class represents the nodes (private, users don't access it)
  - One class represents the entire list (public)



Activity

A Wrapper for a List

- Declare a simple `List` struct
  - Contains a head pointer, pointing to a `ListNode`
- Write new functions:
  - `list_create()` - creates struct with `head=NULL`
  - `list_print()`
  - `list_push_head()` - returns nothing, even though it changes the head
  - `list_pop_head()` - only returns the value, not the head (even though the head changes)

```
typedef struct List {
    ListNode *head;
} List;

List* list_create() {
    List *retval = malloc(sizeof(List));
    if (retval == NULL)
        return NULL;
    retval->head = NULL;
    return retval;
}
```

```
void list_print(List *list) {
    lln_print(list->head);     // no reason not to use!
}

int list_push_head(List *list, int val) {
    ListNode *new_head = lln_create(val);
    if (new_head == NULL)
        return 1;              // error
    new_head->next = list->head;
    list->head = new_head;
    return 0;
}
```

- Notice how `push_head()` uses the `create()` method for another type. Encapsulation!

```c
int list_pop_head(List *list) {
    ListNode *old_head = list->head;
    ListNode *new_head = old_head->next;

    int retval = old_head->val;

    lln_destroy(old_head);
    list->head = new_head;

    return retval;
}
```

C Struct Variables, Without Pointers

- Remember:
  - In Python, all variables are references (that is, pointers)
  - In Java, all object variables are references (although non-objects can be primitives)
- But in C, pointers are explicitly part of the type
- What happens if you declare a struct variable without a pointer?

```
MyType foo;       //  ???
```

C Struct Variables, Without Pointers

```
MyType foo;
```

- A struct variable without a * is **literally declaring a variable of that type,** not a pointer
- On the stack: can use just like any other variable
  - Goes out-of-scope when you return
- Inside another struct: just like any other field
  - `sizeof()` the outer struct includes space for the inner

```c
typedef struct Point {
    double x,y;
} Point;

typedef struct Triangle {
    Point a,b,c;
} Triangle;
```

- `Point` is the size of 2 `double`s.
- `Triangle` is the size of 6 `double`s.
- `malloc(sizeof(Triangle))` allocates everything, all at once.

Dot vs Arrow

- Python,Java (and other languages) use dot to access a field

```
obj.field
```

- C only allows dot when you have a **literal struct,** not a pointer
- Use arrow syntax if it's a pointer

```
MyType a;
MyType *ptr;
a.field    = ... ;
ptr->field = ... ;
```

- NOTE: arrow is just shorthand for:

```
(*ptr).field
```

Arrays of Structs

- Because structs can be simple variables, arrays are also possible!

```
MyType several_objects[100];
```

- Similarly, we can `malloc()` an array of structs easily:

```
int arrlen = 25;
MyType *arr = malloc(arrlen*sizeof(MyType));
```

Another Limitation of Structs

- C doesn't automatically provide copy syntax for structs
- Have to copy manually
  - Field by field OK, sometimes necessary
  - `memcpy()` better

```
Foo a,b;
a = b;          // ILLEGAL
a.x = b.x;      // OK
memcpy(&b, &a, sizeof(a));   // IDEAL
```

Another Limitation of Structs

- C doesn't automatically provide copy syntax for structs
- Have to copy manually
  - Field by field OK, sometimes necessary
  - `memcpy()` better

```
Foo a,b;
a = b;          // ILLEGAL
a.x = b.x;      // OK
memcpy(&b, &a, sizeof(a));   // IDEAL
```

- `sizeof(a)` and `sizeof(Foo)` are exactly the same, because `a` is of type `Foo`.

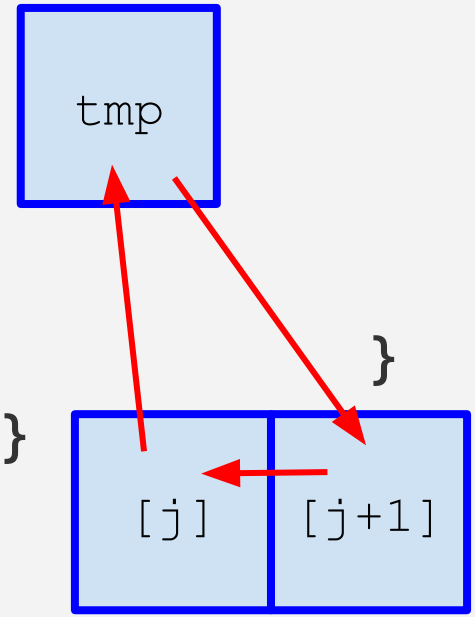
Arrays of Structs in Practice

- Assume that the struct `Foo` has been defined
  - But I won't tell you what it is!
- Write the function `bubble_sort_Foo()`
  - Takes an array of `Foo` as input. How long is it?
  - Sorts the array in-place (don't `malloc()` anything)
  - Use this function to compare two Foo objects:

```
int compare_Foo(Foo*,Foo*);
```

    - Returns (negative, zero, or positive) if first `Foo` is (less than, equal, greater than) the second `Foo`

```
void bubble_sort_Foo(Foo *arr, int count) {
    for (int i=0; i<count; i++)
        for (int j=0; j+1<count; j++)
            if (compare_Foo(&arr[j], &arr[j+1]) > 0)
            {
                Foo tmp;
                memcpy(&tmp    ,  arr+j  , sizeof(Foo));
                memcpy( arr+j  ,  arr+j+1, sizeof(Foo));
                memcpy( arr+j+1, &tmp    , sizeof(Foo));
            }
}
```



Isn’t It a Shame?

- Wouldn’t it be nice if we could write our sorting algorithms only once? But we need to call a different `compare()` function for each type.

- So it’s impossible!
- But wait, the standard library has done it, somehow…

```
#include <stdlib.h>
void qsort(...)
```

Function Pointers

- A **function pointer** is a variable that contains the **address of a function.**
- Declaration defines the parameter types & return type
- But you can point it at any compatible function

- Unfortunately, the syntax is really ugly:

```
RetType (*pointer_name)(int arg1, char arg2);
```

```
void bubble_sort_Foo_fptr(Foo *arr, int count,
                          int (*compare)(Foo*,Foo*)) {
    for (int i=0; i<count; i++)
        for (int j=0; j+1<count; j++)
            if ((*compare)(&arr[j], &arr[j+1]) > 0)
            {
                Foo tmp;
                memcpy(&tmp    ,  arr+j  , sizeof(Foo));
                memcpy( arr+j  ,  arr+j+1, sizeof(Foo));
                memcpy( arr+j+1, &tmp    , sizeof(Foo));
            }
}
```