Linked Structures, Project 1: Linked List

Bryce Boe

2013/07/11

CS24, Summer 2013 C

Outline

- Separate Compilation Review
- “Things” from Lab 3
- Linked Structures
- Project 1 Linked List Walk Through

SEPARATE COMPILATION REVIEW

Questions

- Why should you never #include a ".c" file?
  - Doing so doesn't allow for *separate compilation*
- What is the purpose of the "#ifndef … #define … #endif" guard around the content of ".h" files?
  - Avoids structures and functions from being declared more than once

Another Question

- What is the primary purpose of separate compilation?
  - To reduce subsequent compilation time by reusing *object* files

“THINGS” FROM LAB 3

Code reduction tip

- How can we improve the following?

```
if (size == 0)
        return 1;
else
        return 0;
```

```
return size == 0;
```

What’s the potential problem?

```
struct List *list;
if((list = malloc(sizeof(struct List))) == NULL)
        return NULL;
if((list->_items = malloc(2*sizeof(char *))) == NULL)
        return NULL;
list->_allocated = 2;
list->_size = 0;
return list;
```

Memory leak of the memory assigned to list

What’s the potential problem?

```
struct List *list;
if((list = malloc(sizeof(struct List))) == NULL)
        return NULL;
if((list->_items = malloc(2*sizeof(char *))) == NULL) {
        free(list);
        return NULL;
}
list->_allocated = 2;
list->_size = 0;
return list;
```

Memory leak of the memory assigned to list

String Memory Question

```
char msg[] = “hello world”
list_push_back(msg);
```

Should list_push_back make a copy of the string to store in the List?

or

Should the “user” be responsible for making a copy before calling list_push_back when necessary?

```
        list_push_back(strdup(msg));
```

sizeof(some_pointer)

- Using sizeof works for static arrays:
  - int nums[] = {1, 2, 3, 4, 5}
  - sizeof(nums) results in 20 (5 ints * 4 bytes)
- Using sizeof does not work for pointers (even if they are static arrays in a different scope)
  - int *nums = malloc(20);
  - sizeof(nums) results in 4 as the size of a pointer is 4 bytes (32 bit architecture)

LINKED STRUCTURES

Let’s talk about complexity

- When evaluating data structures and algorithms we often want to consider
- Time complexity
    - How long might an operation take as a function of the input size in the
        - worst case, average case, best case
- Storage complexity
    - How much memory is required to complete an operation

big-O Notation

- We use O(?) to represent the complexity of an algorithm
- O(1) means the operation requires a constant time or space requirement (this is the best)
  - Accessing a random element in an array
- O(n) means the time (or space) required is linear with respect to the input size
  - Copying an array

Common Ordered Complexities

- $O(1)$ – constant time
- $O(\log(n))$ – logarithmic time
- $O(n)$ – linear time
- $O(n\log(n))$ – linearithmic time
- $O(n^2)$ – quadratic time
- $O(2^n)$ – exponential time
- $O(n!)$ – factorial time

What’s wrong with using arrays to store data?

- Arrays require continuous chunks of memory
- Unless the array is full, there is *wasted* space
- Expanding the array is typically done by doubling the size
  - Worst case time: Have to copy all the existing items: *BIG-O* O(n)
  - Hint: realloc does this for you (think about how realloc is implemented)

How long does it take?

- Appending an item to a non-full array?
- Appending an item to a full-array?
- Removing an item from the end of the array?
- Removing an item from the beginning of the array?
- Accessing an element in the middle of the

Single-link **Node** structure

```
struct Node {
        int _data;
        struct Node *_next;
}
```

Node allocation walkthrough

- Add an initial node
- Add another node at the beginning
- Add another node at the end
- Remove a node at the beginning
- Remove a node at the end

PROJECT 1 LINKED WALKTHROUGH

Linked-implementation walk through

- struct List* list_construct()
- void list_destruct(struct List *list)
- int list_size(struct List *list)
- int list_is_empty(struct List *list)
- char *list_at(struct List *list, int position)
- int *list_push_back(struct List *list, char *ite)
- char *list_remove_at(struct List *list, int pos)