Week 04 2

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

October 5, 2023

Announcements

- Lab 4 tomorrow
- End of this class
  - you should be comfortable
    - reading and making sense of A2

WORKSHEET (cont'd)

`strings.pdf`

C data-structures

- thus far
  - we have worked with the primitive types
- numeric (backed) types
  - int, char, float
- and pointers
  - to any type we know
    - including pointers

How do we make a linked list?

- Arbitrarily long list
  - with (at least) forward traversal
- given the primitives that you know…
  - an array won't work
    - at least, not super well
      - how do we insert in the middle?
      - we'd have to move *a lot* of data

C `struct`

- grouping primitives together
  - one *contiguous* space in memory
    - for all the grouped primitives
- that's all it really is
  - though it is **very** *reminiscent*
    - of objects from OO languages

Example

```
struct complex_number {
    float real;
    float imag;
};

// You use it like this:
// make a complex number (memory on the stack)
struct complex_number c1;
// assign values to the members
c1.real = 1.0;
c1.imag = 2.0;
```

How does this get us to a linked list?

- we can make a struct that has
  - a value
  - and **a pointer to the next node**
- then we can move from node to node
  - by following the pointers
  - and using a pointer to keep track
    - of where we are

Example

```
struct node {
    int value;
    struct node *next;
};

// You use it like this:
// make two nodes (memory on the stack)
struct node n1;
struct node n2;
struct node *root = &n1;
struct node *current = root;
// assign values to the members
n1.value = 1; n2.value = 2;
n1.next = &n2; n2.next = NULL;
```

Searching the linked list

- assume the same struct as before
- consider this function

```
struct node *find_value(struct node *root, int value) {
    struct node *current = root;
    while (current != NULL) {
        if (current->value == value) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```

Wait, what's `->`?

- `struct` variables access members with `.`
- but to make dereferencing pointers easier/cleaner
  - there is a special operator
    - `->`
  - only for struct pointers
- it means, assume the left side is a pointer
  - and dereference it
    - then access the member name on the right side

`typedef`

- often we alias something like `struct node`
  - to something shorter
    - like `node`
- we do this with `typedef`

```
typedef struct node {
    int value;
    struct node *next;
} node;
```

`typedef` (cont'd)

- now we can use `node` instead of `struct node`
- and we can use `node *` instead of `struct node *`
- which is a bit cleaner
  - and allows us to treat it as a primitive

        - for our particular application

- you can use `typedef` with any type
    - it follows the pattern:
        - `typedef <type> <new name>;`
    - `<type>` can be any type
        - including a struct (or struct of structs, etc.!)

WORKSHEET

`structs.pdf`