08 Advanced Pointers

Pointer to Pointer for LinkedList functions

Full Code

```
void insert_v2(Person p, int pos, LinkedList* l) {
    *l = insert(p, pos, *l);
}

void reverse_v2(LinkedList* l) {
    *l = reverse(*l);
}
```

```c
int main() {
    Node third = {
        {"Alice", 22, Single},
        NULL
    };
    Node second = {
        {"Bob", 26, Married},
        &third
    };
    Node first = {
        {"Charlie", 20, Engaged},
        &second
    };
    Person D = {"Raj", 18, NotMentioned};
    Node l2 = { D, NULL};

    Person E = {"Eve", 28, Married};
    Node l3 = { E, NULL};

    LinkedList l = &first;
    printf("Size of the list is %d\n", size(l));
    print_list(l);
    //l = reverse(l);
    reverse_v2(&l);
    print_list(l);
    // l = insert(D,2,l);
    // Problem1: Write the insert function such that,
    // we dont need to reassign l to the return value
    // of the function to update it. That is the
    // line bellow is equivalent to line above
    insert_v2(D, 2, &l);
    insert_v2(E, 3, &l);
    print_list(l);

    return 0;
}
```

Function Pointers

A variable to store address of functions

```
bool check1(Person p) {
    return p.status == Single && p.age < 24;
}

bool check2(Person p) {
    return p.age <= 26 && p.age >= 16;
}

// check_person can store address of check or check_2
bool (*check_person)(Person p) = &check1;
```

Function which takes function as arguments (Higher Order Functions)

Suppose we want to filter person who is Single and age <= 24, how to implement it.

Copy paste solution

```
LinkedList filterby_age(LinkedList l, int lower, int upper) {
    LinkedList l2 = NULL;
    while(l != NULL) {
        if (l->data.age >= lower && l->data.age <= upper) {
            l2 = append(l->data, l2);
        }
        l = l->next;
    }
    return l2;
}

LinkedList filter_status_age(LinkedList l, RelStatus s, int age) {
    LinkedList l2 = NULL;
    while(l != NULL) {
        if (l->data.status == s && l->data.age <= age) {
            l2 = append(l->data, l2);
        }
        l = l->next;
    }
    return l2;
}
```

Problem: For every condition, we need to write new functions with similar logic.

Function pointer Solution

Full Code

```
LinkedList filter(LinkedList l, bool (*check)(Person)) {
    LinkedList l2 = NULL;
    while(l != NULL) {
        if ((*check)(l->data) == true) {
            l2 = append(l->data, l2);
        }
        l = l->next;
    }
    return l2;
}

int main() {
    Node third = {
        {"Alice", 22, Single},
        NULL
    };
    Node second = {
        {"Bob", 26, Married},
        &third
    };
    Node first = {
        {"Charlie", 20, Engaged},
        &second
    };
    Person D = {"Raj", 18, NotMentioned};
    Node l2 = { D, NULL};

    Person E = {"Eve", 28, Married};
    Node l3 = { E, NULL};

    LinkedList l = &first;
    print_list(l);

    bool check1(Person p) {
        return p.status == Single && p.age < 24;
    }

    bool check2(Person p) {
        return p.age <= 26 && p.age >= 16;
    }
    bool (*check_person)(Person p) = &check1;

    // Problem2: Filter the linked list of person
    // who are Single and less than 24 in age.
    l = filter(l, &check1);
    print_list(l);

    return 0;
}
```

Find Best Person

Full Code

```
Person best(LinkedList l, bool (*compare_fn)(Person, Person)) {
    Person* b = &(l->data);
    while(l!= NULL) {
        if ((*compare_fn)(l->data, *b)) {
            b = &(l->data);
            // printf("Best person uptill now is %s\n", b->name); // for debugging
        }
        l = l->next;
    }
    return *b;
}
```

Inside main

```
bool compare_status(Person p, Person q) {
    // printf("Compare %s %s\n", p.name, q.name); //for debuging
    return p.status >= q.status;
}

bool compare_status_age(Person p, Person q) {
    // printf("Compare %s %s\n", p.name, q.name); // for debugging
    return (p.status >= q.status) && (p.age >= q.age);
}

printf("Best person is\n");
print_person(best(l, &compare_status_age));
```

Homework 1

Suppose you want to print every person in the linkedlist and would like to customize how each person is printed.

Use a function pointer as a second argument to `print_list` function, which prints the person in a customized way. You should be able to pass the pointer to `print_person` function or any other similar function.

```
void print_person(Person p) {
    char status_string[][15] = {
        "Not Mentioned","Single",
        "Engaged", "Married"
    };
     printf("%s\t\t%d\t%s\n",p.name, p.age, status_string[p.status]);
}

void print_list(LinkedList l) {
    printf("-----------------------------------\n");
    while (l != NULL) {
        print_person(l->data);
        l = l->next;
    }
    printf("-----------------------------------\n");
}
```

Homework 2

Suppose you want to find some aggregate value of the elements of the linked list with integer data field. For example, sum of all elements, sum of squares of all elements, sum of absolute values of all elements etc.

Write an `aggregate` function which takes the linked list and a function pointer as arguments. The function pointer should be able to point to functions which implement any of the above functionalites. The `aggregate` function should return the aggregate value.

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef Node* LinkedList;

// Write functions to find sum of all elements,
// sum of squares of all elements, sum of absolute
// values of all elements

// Generalize above functions to get an agregate function
// which the takes the a function poiner where the
// aggregation method can be passed.
int aggregate(LinkedList l, /* function pointer here */) {
    // code for aggregae here
}
```

Full Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_NAME_LEN 100

typedef enum RelStatus {
    NotMentioned,
    Single,
    Engaged,
    Married
} RelStatus;

typedef struct Person {
    char name[MAX_NAME_LEN];
    int age;
    RelStatus status;
} Person;

typedef struct Node {
    Person data;
    struct Node* next;
} Node;

typedef Node* LinkedList;

void print_person(Person p) {
    char status_string[][15] = {
        "Not Mentioned","Single",
        "Engaged", "Married"
    };
```

```
     printf("%s\t\t%d\t%s\n",p.name, p.age, status_string[p.status]);
}

void print_list(LinkedList l) {
    printf("-----------------------------------\n");
    while (l != NULL) {
        print_person(l->data);
        l = l->next;
    }
    printf("-----------------------------------\n");
}

int size(LinkedList l) {
    int s = 0;
    while (l != NULL) {
        l = l->next;
        s ++;
    }
    return s;
}

Person* element_at(int pos, LinkedList l) {
    int s = 0;
    while (l != NULL) {
        if (s == pos) return &(l->data);
        l = l->next;
        s ++;
    }
    return NULL;
}

LinkedList append(Person p, LinkedList l) {
    if (l == NULL) {
        // Node D = {{"Raj", 18}, NULL};
        Node* D = (Node *) malloc(sizeof(Node));
        D->data = p;
        D->next = NULL;
        return D;
    } else {
        l->next = append(p, l->next);
    }
    return l;
}

LinkedList insert(Person p, int pos, LinkedList l) {
    if (pos == 0) {
        Node* D = (Node *) malloc(sizeof(Node));
        D->data = p;
        D->next = l;
        return D;
    } else {
        l->next = insert(p, pos-1, l->next);
        return l;
    }
}

void insert_v2(Person p, int pos, LinkedList* l) {
    *l = insert(p, pos, *l);
}

LinkedList filterby_age(LinkedList l, int lower, int upper) {
    LinkedList l2 = NULL;
    while(l != NULL) {
        if (l->data.age >= lower && l->data.age <= upper) {
            l2 = append(l->data, l2);
        }
        l = l->next;
    }
    return l2;
}

LinkedList filter(LinkedList l, bool (*check)(Person)) {
    LinkedList l2 = NULL;
    while(l != NULL) {
        if ((*check)(l->data) == true) {
            l2 = append(l->data, l2);
        }
        l = l->next;
    }
    return l2;
}

LinkedList reverse(LinkedList l) {
    int s = size(l);
    LinkedList l2 = NULL;
    for (int i = 0; i < s; i++) {
        l2 = insert(*element_at(s-i-1, l), i, l2);
    }
    return l2;
}

void reverse_v2(LinkedList* l) {
    *l = reverse(*l);
}

Person best(LinkedList l, bool (*compare_fn)(Person, Person)) {
    Person* b = &(l->data);
    while(l!= NULL) {
        if ((*compare_fn)(l->data, *b)) {
```

```c
            b = &(l->data);
            // printf("Best person uptill now is %s\n", b->name); // for debugging
        }
        l = l->next;
    }
    return *b;
}

int main() {
    Node third = {
        {"Alice", 22, Single},
        NULL
    };
    Node second = {
        {"Bob", 26, Married},
        &third
    };
    Node first = {
        {"Charlie", 20, Engaged},
        &second
    };
    Person D = {"Raj", 18, NotMentioned};
    Node l2 = { D, NULL};

    Person E = {"Eve", 28, Married};
    Node l3 = { E, NULL};

    LinkedList l = &first;
    printf("Size of the list is %d\n", size(l));
    print_list(l);
    l = reverse(l);
    reverse_v2(&l);
    print_list(l);
```

```
// l = insert(D,2,l);
// Problem1: Write the insert function such that,
// we dont need to reassign l to the return value
// of the function to update it. That is the
// line bellow is equivalent to line above
insert_v2(D, 2, &l);
insert_v2(E, 3, &l);
print_list(l);

bool compare_status(Person p, Person q) {
    // printf("Compare %s %s\n", p.name, q.name); //for debuging
    return p.status >= q.status;
}

bool compare_status_age(Person p, Person q) {
    // printf("Compare %s %s\n", p.name, q.name); // for debugging
    return (p.status >= q.status) && (p.age >= q.age);
}

printf("Best person is\n");
print_person(best(l, &compare_status_age));

bool check1(Person p) {
    return p.status == Single && p.age < 24;
}

bool check2(Person p) {
    return p.age <= 26 && p.age >= 16;
}

bool (*check_person)(Person p) = &check1;

// Problem2: Filter the linked list of person
```

```
    // who are Single and less than 24 in age.
    l = filter(l, &check2);
    print_list(l);



    return 0;
}
```