**Lec 31 code**

#include <stdio.h>

// Variables declared outside any function are said to be in global scope

int p = 22;

// All functions can see global variables

void bar(int i){

printf("Inside bar, p = %d\n", p);

p = 25; // bar changes the global variable

}

// Unless those functions choose to shadow the global variable

// by defining a local variable of their own with the same name

// The new shadowing variable can be of any type, not necessarily

// The same type as the global variable

void foo(int i){

float p = 33.5;

i = 11;

printf("Inside foo p = %f\n", p);

}

// Static variables retain their value across function calls

// They are not destroyed when the function returns

// Moreover, the statement declaring and initializing them is

// only executed the first time the function is called

void count(){

static int calls = 1;

printf("The function count was called %d times\n", calls++);

}

int main(){

// Global variables are visible everywhere

printf("Global p = %d\n", p);

int i = 100;

foo(i); // foo shadows the global variable p

bar(i); // bar does not shadow the global variable p and changes it

printf("Global p is now changed = %d\n", p);

foo(p); // foo will not be affected by change of global variable

int a = 42;

printf("Outside loop, a = %d\n", a);

// Good practice to shadow the counter variable of loops

// This way we can prevent accidentally overwriting some useful

// value the loop counter variable may have stored earlier

for(int i = 0; i < 5; i++){

double a = -1.50; // a shadows the a defined on line 41

printf("Loop counter i = %d", i);

printf(" inside loop a = %lf\n", a);

}

printf("Outside loop, a = %d\n", a);

printf("Precious value of i is still intact = %d\n", i);

// Main is not a reserved keyword so we can define a variable

// named main as well

int main = 77;

printf("The value of the variable main = %d\n", main);

count();

count();

count();

int count = 90; // This will shadow the global function name count

// Now count is a variable and not a function so the following

// statement, if uncommented, will give a compilation error

// count();

return 0;

}

**lec 32 code**

#include <stdio.h>

#include <stdlib.h>

/\*\*\*\*\*\*\*\*\*\*

Structures are neat ways of collecting two or more variables into a nice

package so as to create a new datatype

\*\*\*\*\*\*\*\*\*\*/

// The structure Point has two feilds

// The first field is an integer called x

// The second field is another integer called y

struct Point{

int x;

int y;

};

// Structures can have any number of fields of any types

struct Misc{

int a;

float b;

char c;

double d;

};

// The structure Point3D has three feilds

// All three feilds are integers and are named x, y, and z

struct Point3D{

int x, y, z;

};

// Structure fields can be structures themselves

// The structure Rectangle has two fields. Both the fields are Point

// variables. The first Point variable is called UR, the next is called LL

struct Rectangle{

struct Point UR;

struct Point LL;

};

// The structure Cuboid has two Point3D fields with names Corner1, Corner2

struct Cuboid{

struct Point3D Corner1, Corner2;

};

// scanf cannot read user-defined structures directly

// We need to write a function to read structures ourselves

// scanPoint takes the address of a Point variable, reads data from

// input and writes it into the fields of that Point variable

void scanPoint(struct Point \*ptr){

scanf("(%d, %d)", &(ptr->x), &ptr->y);

// The above is a shortcut for

// scanf("(%d, %d)", &((\*ptr).x), &((\*ptr).y));

// In general, the -> operator is a neat shortcut to first

// dereference an address, then access a certain field i.e.

// ptr->x is a shortcut for (\*ptr).x

// WARNING: (\*ptr).x is not the same as \*ptr.x since the . operator

// has a higher precedence than the \* operator

// \*ptr.x would be interpreted as \*(ptr.x) which may cause a segfault

}

// Can write beautiful code to read a Rectangle variable from the input

// by reading two Point variables. Can reuse the scanPoint function here.

void scanRectangle(struct Rectangle \*ptr){

scanf("[");

scanPoint(&ptr->LL);

// Recall that the -> operator is a shortcut

// The above statement is exactly equivalent to

// scanPoint(&((\*ptr).LL));

scanf(":");

scanPoint(&ptr->UR);

scanf("]");

}

// Find the area of a Rectangle

int findArea(struct Rectangle r){

int area = (r.UR.x - r.LL.x) \* (r.UR.y - r.LL.y);

r.UR.x = 100;

return area;

}

int main(){

// Can declare structure variables just as simple int variables

int a, b;

struct Point p, q;

struct Rectangle r;

// We have decided that a Point variable must be formatted as

// (x, y)

// in the input. Try giving input (3, 4) and see what happens.

// Also try violating this format and give input as (3 4) and see.

scanPoint(&p);

printf("Point p = [%d %d]", p.x, p.y);

// We have decided that a Rectangle variable must be formatted as

// [(x1, y1):(x2, y2)]

// Give various input and see what happens

scanRectangle(&r);

printf("Rectangle [%d %d %d %d]\n", (r.LL).x, r.LL.y, r.UR.x, r.UR.y);

printf("Area is %d\n", findArea(r));

printf("Rectangle [%d %d %d %d]\n", (r.LL).x, r.LL.y, r.UR.x, r.UR.y);

// sizeof operator can work a bit funny with structures but it will

// always give num of bytes used to store a variable of type Cuboid

printf("size of rect is %d", sizeof(struct Cuboid));

// Can declare arrays of structures too

// Can declare static arrays

int n = 20;

int intArr[n];

struct Rectangle rectArrStatic[n];

// ... as well as dynamic arrays. The syntax is exactly the same

// as that used to declare a simple integer array

int \*intPtr = (int\*)malloc(n \* sizeof(int));

struct Rectangle \*rectPtr = (struct Rectangle\*)malloc(n \* sizeof(struct Rectangle));

// Can access ith element of an array of structures using []

struct Rectangle temp = rectPtr[2];

// Rule 5 of pointers applies here too - array name is just a

// pointer to first element. Thus, rectArrStatic points to the

// Rectangle variable rectArrStatic[0]

temp = \*(rectArrStatic + 5);

return 0;

}

**Lec 33 code**

#include <stdio.h>

// Use the typedef shortcut in case you wish to avoid writing

// struct Point again and again. Once you do this, you can just

// write Point as a bonafide datatype as you can write int

typedef struct Point{

char c;

int x, y;

}Point;

// Note that the name of the structure i.e. "Point" is repeated

// twice. This is not necessary and the above is just a very common

// idiomatic use of the typedef command. Look up books or websites

// in case you are interested in learning more.

// A line is made up of two points. We can use the typedef command

// to avoid writing struct Line again and again. After this, we just

// have to write Line and not struct Line everywhere.

typedef struct Line{

Point p1, p2;

}Line;

// The function takes in a Line variable (note we did not have to write

// struct Line l as the input to the function) and prints one of its

// subfields and then changes. However, this does nothing to the original

// Line variable that was passed to the function (Rule 4 of functions)

void fun(Line l){

printf("The char variable is %c\n", l.p2.c);

l.p2.c = 'x';

}

int main(){

// Can initialize structure variables at time of declaration

Point p = {'a',10,30};

// Can do so even if some fields are structures themselves

Line l1 = {{'a',1,2},{'b',4,5}};

// The sizeof operator can sometimes give a size of a structure

// variable that is larger than the sum of the sizes of the fields

// in that structure - this is due to padding done by the compiler

// to obey demands made by the microprocessors. However, if you use

// sizeof operator during malloc/calloc/realloc, you never have to

// worry about this weird feature.

printf("Size of Point is %d and that of Line is %d\n", sizeof(Point), sizeof(Line));

// The assignment operator, when used with structure variables,

// copies every field individually. If those fields happen to be

// structure variable themselves, their fields are also copied one

// by one - it is quite a thorough process :)

Line l2 = l1;

// I can change l1 but l2 will not get affected since I copied l1 to l2

l1.p2.c = 'p';

// Rule 4 of functions applies as is to structure variables

fun(l2);

printf("p2 looks like {%c %d %d}\n", l2.p2.c, l2.p2.x, l2.p2.y);

// Can declare an array of Line variables just as we could declare

// an array of int variables

Line lineArr[5];

// The [] and . operators are both left associative. Moreover, the two

// operators [] and . have the same precedence.

lineArr[2].p1.x = 90;

// The above expression is equivalent to

// ((lineArr[2]).p1).x = 90;

}

**Lec 34 code 1**

#include <stdio.h>

#include <stdlib.h>

/\*\*\*\*\*\*\* CODE FOR LINKED LIST \*\*\*\*\*\*\*\*/

typedef struct Node{

float x;

struct Node \*next; // The next node in the list

}Node;

// Get the address of a newly created structure variable

Node\* createNode(float x){

Node \*newNode = (Node\*)malloc(sizeof(Node));

newNode->x = x; // We know the value to be stored here

newNode->next = NULL; // Be helpful to the person calling this code

return newNode;

// Don't do what is commented below. Statically declared variables

// are deleted once the function returns

// Node newNode;

// return &newNode;

}

// Add a node at the end of the list and return the head of the list

// Head of the list can change if the list was empty to begin with

Node\* insertAtEnd(Node \*head, float x){

if(head == NULL)

head = createNode(x); // New head is the first node

else

head->next = insertAtEnd(head->next, x);

return head;

}

// Insert the given data (float x) at specified index of the linked list

// Return address of the head node of the linked list since the head

// node may have changed after if insertion took place at idx 0

Node\* insertAtPosition(Node \*head, float x, int idx){

if(idx == 0){ // Insert here

Node \*newHead = createNode(x);

newHead->next = head;

head = newHead; // This is the new head now

}else if(head == NULL){

// If idx != 0 and head == NULL, then something is wrong. This means

// the list has only k nodes and I am asking the k+2-th or k+3-th

// node to be inserted - this is wrong. The k+1-th node should

// be inserted before k+2-th node is inserted

printf("ERROR - cant insert at this position in the list\n");

}

else{ // All is well - recursively call this routine

head->next = insertAtPosition(head->next, x, idx-1);

}

// Head may not have changed but as promised, return the head

return head;

}

// Delete the node at specified index of the linked list and return the

// address of the new head node of the linked list since the head node may

// have changed if the head is the only node in the linked list

Node\* deleteAtPosition(Node \*head, int idx){

if(head == NULL)

printf("ERROR - nothing to delete at this position\n");

else{

if(idx == 0){ // Delete this node

Node \*newHead = head->next;

free(head); // Free the head

head = newHead; // Next becomes new head

}else{

head->next = deleteAtPosition(head->next, idx - 1);

}

}

// Head may or may not have changed but as promised, return head

return head;

}

// Delete the head and return the new head

Node\* deleteHead(Node \*head){

return deleteAtPosition(head, 0);

}

// Delete the entire list

Node\* dumpList(Node \*head){

while(head != NULL)

head = deleteHead(head);

return head;

}

// Get a pointer to the node in the linked list at index idx

// If idx is illegal, return a NULL pointer

Node\* getNodeAtPosition(Node \*curr, int idx){

Node \*answer = curr; // If idx == 0, answer is current node itself

if(curr == NULL)

printf("ERROR - no such node exists in the list\n");

else{

if(idx > 0)

answer = getNodeAtPosition(curr->next, idx-1);

}

return answer;

}

// Move numPos positions right from current position in the linked list

Node\* movePositionsRight(Node \*curr, int numPos){

return getNodeAtPosition(curr, numPos);

}

// Traverse and print the whole list

void traverse(Node \*head){

if(head == NULL){

printf("X\n");

}else{

printf("%0.2f => ", head->x);

traverse(head->next);

}

}

int main(){

Node \*head = NULL;

traverse(head);

head = insertAtEnd(head, 1.0);

traverse(head);

insertAtEnd(head, 2.0);

traverse(head);

insertAtEnd(head, 3.0);

traverse(head);

head = insertAtPosition(head, -1.0, 0);

traverse(head);

head = insertAtPosition(head, 10.0, 9);

traverse(head);

head = insertAtPosition(head, 1.5, 2);

traverse(head);

head = insertAtEnd(head, 4.0);

traverse(head);

Node \*ptr = getNodeAtPosition(head, 3);

printf("Node at index 3 is %0.2f\n", ptr->x);

// Move 2 locations to the right and print the node there

ptr = movePositionsRight(ptr,1);

if(ptr != NULL)

printf("Move 1 location to the right and find %0.2f\n", ptr->x);

ptr = movePositionsRight(ptr,1);

if(ptr != NULL)

printf("Move 1 more location to the right and find %0.2f\n", ptr->x);

ptr = movePositionsRight(ptr,1);

printf("Move 1 more location to the right and find nothing :)\n");

// Cannot move left in a linked list

// Need a doubly linked list for that

head = deleteAtPosition(head, 4);

traverse(head);

head = deleteHead(head);

traverse(head);

head = deleteAtPosition(head, 4);

traverse(head);

head = deleteAtPosition(head, 2);

traverse(head);

head = deleteAtPosition(head, 2);

traverse(head);

head = dumpList(head);

traverse(head);

return 0;

}

**Lec 34 code 2**

#include <stdio.h>

#include <stdlib.h>

/\*\*\*\*\*\*\* CODE FOR DOUBLY LINKED LIST \*\*\*\*\*\*\*\*/

typedef struct Node{

float x;

struct Node \*prev; // The previous node in the list

struct Node \*next; // The next node in the list

}Node;

// Get the address of a newly created structure variable

Node\* createNode(float x){

Node \*newNode = (Node\*)malloc(sizeof(Node));

newNode->x = x; // We know the value to be stored here

newNode->prev = NULL; // Be helpful to the person calling this code

newNode->next = NULL; // It is nice to be considerate

return newNode;

}

// Add a node at the end of the list and return the head of the list

// Head of the list can change if the list was empty to begin with

Node\* insertAtEnd(Node \*head, float x){

if(head == NULL) // This is the first node

head = createNode(x);

// Need to handle the case below separately in case of doubly linked

// lists since we need to establish back links as well

else if(head->next == NULL){ // head is the last node of the list

Node \*newNode = createNode(x);

head->next = newNode; // Forward link

newNode->prev = head; // Backward link

}else // There are more nodes to the right in the list

head->next = insertAtEnd(head->next, x);

// As promised, return the head even if it is still the same

return head;

}

// Insert the given data (float x) at specified index of the linked list

// Return address of the head node of the linked list since the head

// node may have changed after if insertion took place at idx 0

Node\* insertAtPosition(Node \*head, float x, int idx){

if(idx == 0){ // Insert this to become new head

// Need to handle this case specially here because of backlinks

Node \*newHead = createNode(x);

newHead->next = head; // Fix one forward link

if(head != NULL){ // If there is a head

// Fix the other forward link

if(head->prev != NULL)

(head->prev)->next = newHead;

// Fix the back links

// Warning: do not interchange the next two lines

// If you do so you will lose the pointer to the old prev

newHead->prev = head->prev;

head->prev = newHead;

}

head = newHead; // This is the new head now

}else{ // idx > 0

if(head == NULL){

// If idx > 0 and head == NULL, then something is wrong.

// This means that I am asking an element to be inserted into

// e.g. k+2-th position when the list has only k elements - this

// is wrong. The k+1-th node should be inserted before k+2-th

// node is inserted

printf("ERROR - cant insert at this position in the list\n");

}else{ // Okay, there is a head

if(idx == 1){ // Insert just after head

Node \*newNode = createNode(x);

newNode->prev = head; // Establish back link

if(head->next != NULL) // If there is a next node

(head->next)->prev = newNode; // The other back link

// Fix the forward links now

// Warning: do not interchange the next two lines

// If you do so you will lose the pointer to the old next

newNode->next = head->next; // Insert new node in the middle

head->next = newNode;

// head remains unchanged

}else{ // idx > 1 - recursively call this routine

head->next = insertAtPosition(head->next, x, idx-1);

}

}

}

// Head may not have changed but as promised, return the head

return head;

}

// Delete the node at specified index of the linked list and return address

// of the first node of the linked list since the first node may have

// changed after if the last node of the list gets deleted

Node\* deleteAtPosition(Node \*head, int idx){

if(head == NULL)

printf("ERROR - nothing to delete at this position\n");

else{

if(idx == 0){ // Delete the old head

Node \*newHead = head->next;

// Fix the broken back and forward links

if(head->prev != NULL)

(head->prev)->next = newHead;

if(newHead != NULL)

newHead->prev = head->prev;

head = newHead; // We have a new head

}else{ // idx > 0

head->next = deleteAtPosition(head->next, idx-1);

}

}

// Head may not have changed but as promised, return the head

return head;

}

// Delete the head and return the new head

Node\* deleteHead(Node \*head){

return deleteAtPosition(head, 0);

}

// Delete the entire list

Node\* dumpList(Node \*head){

while(head != NULL)

head = deleteHead(head);

return head;

}

// Returns the node idx locations from the current node.

// If idx < 0, returns nodes to the left of the current node

// If idx is illegal, return a NULL pointer

Node\* getNodeAtPosition(Node \*curr, int idx){

Node \*answer = curr; // If idx == 0, answer is current node itself

if(curr == NULL)

printf("ERROR - no such node exists in the list\n");

else{

if(idx > 0)

answer = getNodeAtPosition(curr->next, idx-1);

else if(idx < 0)

answer = getNodeAtPosition(curr->prev, idx+1);

}

return answer;

}

// Move numPos positions right from current position in the linked list

Node\* movePositionsRight(Node \*curr, int numPos){

return getNodeAtPosition(curr, numPos);

}

// Move numPos positions left from current position in the linked list

Node\* movePositionsLeft(Node \*curr, int numPos){

return getNodeAtPosition(curr, -numPos);

}

void traverse(Node \*head){

if(head == NULL){

printf("X\n");

}else{

printf("%0.2f <=> ", head->x); // <=> symbol since doubly linked

traverse(head->next);

}

}

int main(){

Node \*head = NULL;

traverse(head);

head = insertAtEnd(head, 1.0);

traverse(head);

insertAtEnd(head, 2.0);

traverse(head);

insertAtEnd(head, 3.0);

traverse(head);

head = insertAtPosition(head, -1.0, 0);

traverse(head);

head = insertAtPosition(head, 10.0, 9);

traverse(head);

head = insertAtPosition(head, 1.5, 2);

traverse(head);

head = insertAtEnd(head, 4.0);

traverse(head);

Node \*ptr = getNodeAtPosition(head, 3);

printf("Node at index 3 is %0.2f\n", ptr->x);

// Move 2 locations to the right and print the node there

ptr = movePositionsRight(ptr,1);

if(ptr != NULL)

printf("Move 1 location to the right and find %0.2f\n", ptr->x);

ptr = movePositionsRight(ptr,1);

if(ptr != NULL)

printf("Move 1 more location to the right and find %0.2f\n", ptr->x);

ptr = movePositionsLeft(ptr,3);

if(ptr != NULL)

printf("Move 3 locations to the left and find %0.2f\n", ptr->x);

ptr = movePositionsLeft(ptr,2);

if(ptr != NULL)

printf("Move 2 more locations to the left and find %0.2f\n", ptr->x);

ptr = movePositionsLeft(ptr,2);

printf("Move 2 more location to the left and find nothing :)\n");

head = deleteAtPosition(head, 4);

traverse(head);

head = deleteHead(head);

traverse(head);

head = deleteAtPosition(head, 4);

traverse(head);

head = deleteAtPosition(head, 2);

traverse(head);

head = deleteAtPosition(head, 2);

traverse(head);

head = dumpList(head);

traverse(head);

return 0;

}

**Lec 36 code 1**

#include <stdio.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is code for selection sort

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Find the index of the maximum value in an array

int findMax(int \*arr, int len){

int maxIdx = 0, maxVal = arr[0];

for(int i = 1; i < len; i++)

if(arr[i] > maxVal){

maxVal = arr[i];

maxIdx = i;

}

return maxIdx;

}

// Swap two integers, of which, addresses are known

void swap(int \*a, int \*b){

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Print an array neatly

// Also denote the active region using brackets

void print(int \*arr, int len, int left){

printf("[");

if(left == 0) printf("]");

for(int i = 0; i < len; i++){

printf("%d", arr[i]);

if(i == left-1) printf("]");

else if(i < len - 1) printf(" ");

}

printf("\n");

}

// len: length of the array

// left: size of the active region

void doSelSort(int \*arr, int len, int left){

if(left != 0){

int maxIdx = findMax(arr, left);

swap(arr + left - 1, arr + maxIdx);

printf("After iteration %d: ", len - left + 1);

print(arr, len, left - 1);

doSelSort(arr, len, left - 1);

}

}

int main(){

int arr[10] = {8,9,6,3,4,2,1,5,10,7};

printf("Initial: ");

print(arr, 10, 10);

doSelSort(arr, 10, 10);

return 0;

}

**Lec 36 code 2**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is code for merge sort

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Swap two integers, of which, addresses are known

void swap(int \*a, int \*b){

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Print an array neatly

void print(int \*arr, int len){

printf("[");

for(int i = 0; i < len; i++){

printf("%d", arr[i]);

if(i < len - 1) printf(" ");

}

printf("]");

}

// Merge two arrays

// c: index where second array begins

// len: total length of the two arrays

void doMerge(int \*arr, int c, int len){

int ptr1 = 0, ptr2 = c;

int \*cpy = (int\*)malloc(len \* sizeof(int));

for(int pos = 0; pos < len; pos++){

if(ptr1 < c && ptr2 < len){

if(arr[ptr1] < arr[ptr2])

cpy[pos] = arr[ptr1++];

else

cpy[pos] = arr[ptr2++];

}else if(ptr1 < c){ // Array 2 exhausted

cpy[pos] = arr[ptr1++];

}else if(ptr2 < len){ // Array 1 exhausted

cpy[pos] = arr[ptr2++];

}

}

for(int pos = 0; pos < len; pos++)

arr[pos] = cpy[pos];

free(cpy); // Done with this array - no memory leaks

}

// len: length of the array

void doMergeSort(int \*arr, int len){

if(len > 1){

int c = (int)ceil(len/2);

doMergeSort(arr, c);

doMergeSort(arr + c, len - c);

printf("Recursive Sort: ");

print(arr, c);

print(arr + c, len - c);

doMerge(arr, c, len);

printf("\nAfter Merge: ");

print(arr, len);

printf("\n");

}

}

int main(){

int arr[10] = {8,9,6,3,4,2,1,5,10,7};

printf("Initial: ");

print(arr, 10);

printf("\n");

doMergeSort(arr, 10);

return 0;

}

**Lec 36 code 3**

#include <stdio.h>

#include <stdlib.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is code for quick sort

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Get a random index in an array of length len

int randIdx(int len){

return rand()%len;

}

// Swap two integers, of which, addresses are known

void swap(int \*a, int \*b){

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Print these many tabs for nice formatting

void printTabs(int numTabs){

printf("\n");

for(int i = 0; i < numTabs; i++)

printf("\t");

}

// Print an array neatly indicating the pivot element in brackets

void print(int \*arr, int len, int pivot){

for(int i = 0; i < len; i++){

if(i == pivot) printf("[");

printf("%d", arr[i]);

if(i == pivot) printf("]");

if(i < len - 1) printf(" ");

}

}

// Partition an array

// len: length of array

// pivot: index of pivot

// return the new index of the pivot element

int partition(int \*arr, int len, int pivot){

int pVal = arr[pivot]; // Value about which to be partitioned

int \*cpy = (int\*)malloc(len \* sizeof(int));

int left = 0, right = len - 1;

for(int i = 0; i < len; i++){

if(arr[i] < pVal)

cpy[left++] = arr[i];

if(arr[i] > pVal)

cpy[right--] = arr[i];

}

for(int i = left; i <= right; i++)

cpy[i] = pVal;

for(int i = 0; i < len; i++)

arr[i] = cpy[i];

free(cpy);

return right;

}

// len: length of the array

// numTabs: nice formatting :)

void doQuickSort(int \*arr, int len, int numTabs){

if(len > 1){

int pivot = randIdx(len); // Choose a random index as pivot

printTabs(numTabs);

printf("Before partition: ");

print(arr, len, pivot);

pivot = partition(arr, len, pivot);

printTabs(numTabs);

printf("After partition: ");

print(arr, len, pivot);

doQuickSort(arr, pivot, numTabs + 1);

doQuickSort(arr + pivot + 1, len - pivot - 1, numTabs + 1);

printTabs(numTabs);

printf("After recursive sort: ");

print(arr, len, pivot);

}

}

int main(){

int arr[10] = {8,9,6,3,4,2,1,5,10,7};

printf("Initial: ");

print(arr, 10, -1); // No pivot yet

doQuickSort(arr, 10, 1);

return 0;

}