# LAB 5

# swap\_pointers.c

#include *<stdio.h>*

*// swap the values in two integers, given as pointers*

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

*// This is a simple main function which could be used*

*// to test your swap\_pointers function.*

*// It will not be marked.*

*// Only your swap\_pointers function will be marked.*

int main(void) {

int first = 1;

int second = 2;

swap\_pointers(&first, &second);

printf("%d, %d**\n**", first, second);

**return** 0;

}

# array\_sum\_prod.c

*// COMP1511 Array Sum Product*

*// Calculate the sum and the product of the elements in an array*

*// and write the results into variables passed into the function*

*// by reference.*

*// Modified by Marc Chee, March 2020*

#include *<stdio.h>*

void array\_sum\_prod(int length, int nums[length], int \*sum, int \*product);

*// This is a simple main function that you can use to test your array\_sum\_prod*

*// function.*

*// It will not be marked - only your array\_sum\_prod function will be marked.*

*//*

*// Note: the autotest does not call this main function!*

*// It calls your array\_sum\_prod function directly.*

*// Any changes that you make to this main function will not affect the autotests.*

int main(int argc, char \*argv[]){

int nums[] = {3,4,1,5,6,1};

int prod;

int sum;

*//Pass in the address of the sum and product variables*

array\_sum\_prod(6, nums, &sum, &prod);

printf("The sum is %d and prod is %d**\n**",sum,prod);

**return** 0;

}

*// Calculates the sum and product of the array nums.*

*// Actually modifies the variables that \*sum and \*product are pointing to*

void array\_sum\_prod(int length, int nums[length], int \*sum, int \*product) {

int i = 0;

\*sum = 0;

\*product = 1;

**while** (i < length) {

\*sum = \*sum + nums[i];

\*product = \*product \* nums[i];

i++;

}

}

# common\_elements.c

int common\_elements(int length, int source1[length], int source2[length], int destination[length]) {

int upto = 0;

int a1 = 0;

**while** (a1 < length) {

int found = 0;

int a2 = 0;

**while** (a2 < length && !found) {

**if** (source1[a1] == source2[a2]) {

found = 1;

}

a2 = a2 + 1;

}

**if** (found) {

destination[upto] = source1[a1];

upto = upto + 1;

}

a1 = a1 + 1;

}

**return** upto;

}

# boxes.c

*// Solution to week 4 boxes task*

*// By Curtis Millar*

*// Written on 2017-08-12*

#include *<stdio.h>*

#include *<stdlib.h>*

int main(int argc, char \*argv[]) {

printf("How many boxes: ");

int numBoxes;

**if** (scanf("%d", &numBoxes) != 1) {

printf("Unable to read number");

exit(0);

}

*// Each box has a gap in the middle*

*// so the width & height are*

int size = numBoxes \* 4 - 1;

int y = 0;

**while** (y < size) {

*// Vertical loop*

int x = 0;

**while** (x < size) {

int xInverse = (size - 1) - x;

int yInverse = (size - 1) - y;

**if** ((y == x) || (y == xInverse)) {

*// On diagonals*

**if** (y % 2 == 0) {

printf("1");

} **else** {

printf("0");

}

} **else** **if** ((y > x) && (y < xInverse) && (x % 2 == 0)) {

*// left side*

printf("1");

} **else** **if** ((y < x) && (y > xInverse) && (x % 2 == 0)) {

*// right side*

printf("1");

} **else** **if** ((x > y) && (x < yInverse) && (y % 2 == 0)) {

*// top side*

printf("1");

} **else** **if** ((x < y) && (x > yInverse) && (y % 2 == 0)) {

*// bottom side*

printf("1");

} **else** {

printf("0");

}

x = x + 1;

}

printf("**\n**");

y = y + 1;

}

**return** 0;

}

# LAB 7

# devowel.c

*// Written 3/3/2018 by Andrew Taylor (andrewt@unsw.edu.au)*

*// read characters from stdin and write to stdout*

*// except lower case vowels ('a', 'e','i', 'o', 'u') are not written*

#include *<stdio.h>*

int is\_vowel(int character);

int main(void) {

*// getchar returns an int which will contain either*

*// the ASCII code of the character read or EOF*

int character = getchar();

**while** (character != EOF) {

**if** (!is\_vowel(character)) {

putchar(character);

}

character = getchar();

}

**return** 0;

}

*// return 1 if character is a lower case vowel*

*// 0 otherwise*

int is\_vowel(int character) {

**return** character == 'a' ||

character == 'e' ||

character == 'i' ||

character == 'o' ||

character == 'u';

}

# swap\_case.c

*// Written 3/3/2018 by Andrew Taylor (andrewt@unsw.edu.au)*

*// Write stdin to stdout with upper case letters converted to lower case*

*// and lower case converted to upper case*

*//*

*// The shift will be supplied as a command-line argument*

#include *<stdio.h>*

#include *<stdlib.h>*

int swap\_case(int character);

int main(int argc, char \*argv[]) {

int character = getchar();

**while** (character != EOF) {

int swapped\_character = swap\_case(character);

putchar(swapped\_character);

character = getchar();

}

**return** 0;

}

int swap\_case(int character) {

**if** (character >= 'A' && character <= 'Z') {

**return** 'a' + character - 'A';

} **else** **if** (character >= 'a' && character <= 'z') {

**return** 'A' + character - 'a';

} **else** {

**return** character;

}

}

# caesar.c

*// Written 3/3/2018 by Andrew Taylor (andrewt@unsw.edu.au)*

*// Modified 20/10/2019 by Marc Chee (marc.chee@unsw.edu.au)*

*// Write stdin to stdout encrypted with a Caesar Cipher*

*// https://en.wikipedia.org/wiki/Caesar\_cipher*

*//*

*// The shift will be supplied as a command-line argument*

#include *<stdio.h>*

#define ALPHABET\_SIZE 26

int encrypt(int character, int shift);

int main(void) {

int shift = 0;

scanf("%d", &shift);

getchar();

*// negative shifts need to be converted to the equivalent positive shift*

**if** (shift < 0) {

shift = ALPHABET\_SIZE + (shift % ALPHABET\_SIZE);

}

int character = getchar();

**while** (character != EOF) {

int encrypted\_character = encrypt(character, shift);

putchar(encrypted\_character);

character = getchar();

}

**return** 0;

}

*// encrypt letters with a caesar cipher with the specified shift*

*// the specified characters is returned shifted the specified number of positions*

*// characters other than letters are returned unchanged*

int encrypt(int character, int shift) {

**if** (character >= 'A' && character <= 'Z') {

**return** 'A' + (character - 'A' + shift) % ALPHABET\_SIZE;

} **else** **if** (character >= 'a' && character <= 'z') {

**return** 'a' + (character - 'a' + shift) % ALPHABET\_SIZE;

} **else** {

**return** character;

}

}

# substitution.c

*// Written 3/3/2018 by Andrew Taylor (andrewt@unsw.edu.au)*

*// Modified 20/10/2019 by Marc Chee (marc.chee@unsw.edu.au)*

*// Write stdin to stdout encrypted with a Substitution cipher*

*// https://en.wikipedia.org/wiki/Substitution\_cipher*

*//*

*// The mapping will be supplied as a command-line argument containing 26 characters:*

*// These will be an an ordering of the letters 'a'..'z'.*

#include *<stdio.h>*

#include *<string.h>*

#define ALPHABET\_SIZE 26

int encrypt(int character, char mapping[ALPHABET\_SIZE + 1]);

int main(void) {

char cipher[ALPHABET\_SIZE + 1];

fgets(cipher, ALPHABET\_SIZE + 1, stdin);

*// this getchar takes the '\n' and discards it*

getchar();

**if** (strlen(cipher) != ALPHABET\_SIZE) {

printf("substitution: mapping must contain %d letters**\n**", ALPHABET\_SIZE);

**return** 1;

}

int character = getchar();

**while** (character != EOF) {

int encrypted\_character = encrypt(character, cipher);

putchar(encrypted\_character);

character = getchar();

}

**return** 0;

}

*// encrypt letters with a substitution cipher with the specified mapping*

int encrypt(int character, char mapping[ALPHABET\_SIZE + 1]) {

**if** (character >= 'A' && character <= 'Z') {

**return** mapping[character - 'A'] - 'a' + 'A';

} **else** **if** (character >= 'a' && character <= 'z') {

**return** mapping[character - 'a'];

} **else** {

**return** character;

}

}

# decode.c

*// Written 3/3/2018 by Andrew Taylor (andrewt@unsw.edu.au)*

*// Modified 20/10/2019 by Marc Chee (marc.chee@unsw.edu.au)// Write stdin to stdout decrypted with a Substitution cipher*

*// https://en.wikipedia.org/wiki/Substitution\_cipher*

*//*

*// The mapping used to encrypt the input*

*// will be supplied as a command-line argument containing 26 characters:*

*// These will be an an ordering of the letters 'a'..'z'.*

#include *<stdio.h>*

#include *<string.h>*

#include *<assert.h>*

#define ALPHABET\_SIZE 26

void compute\_inverse\_mapping(char mapping[ALPHABET\_SIZE + 1], char inverse\_mapping[ALPHABET\_SIZE + 1]);

int decrypt(int character, char inverse\_mapping[ALPHABET\_SIZE + 1]);

int main(int argc, char \*argv[]) {

char inverse\_mapping[ALPHABET\_SIZE] = {0};

char cipher[ALPHABET\_SIZE + 1];

fgets(cipher, ALPHABET\_SIZE + 1, stdin);

*// this getchar takes the '\n' and discards it*

getchar();

**if** (strlen(cipher) != ALPHABET\_SIZE) {

printf("decode: mapping must contain %d letters**\n**", ALPHABET\_SIZE);

**return** 1;

}

compute\_inverse\_mapping(cipher, inverse\_mapping);

int character = getchar();

**while** (character != EOF) {

int decrypted\_character = decrypt(character, inverse\_mapping);

putchar(decrypted\_character);

character = getchar();

}

**return** 0;

}

*// mapping must contain an ordering of letters 'a'..'z'*

*// the inverse\_mapping will be stored in inverse\_mapping*

void compute\_inverse\_mapping(char mapping[ALPHABET\_SIZE + 1], char inverse\_mapping[ALPHABET\_SIZE + 1]) {

int i = 0;

**while** (i < ALPHABET\_SIZE) {

int character = mapping[i];

assert(character >= 'a' && character <= 'z');

inverse\_mapping[character - 'a'] = 'a' + i;

i = i + 1;

}

}

*// decrypt letters with a substitution cipher with the specified inverse\_mapping*

int decrypt(int character, char inverse\_mapping[ALPHABET\_SIZE + 1]) {

**if** (character >= 'A' && character <= 'Z') {

**return** inverse\_mapping[character - 'A'] - 'a' + 'A';

} **else** **if** (character >= 'a' && character <= 'z') {

**return** inverse\_mapping[character - 'a'];

} **else** {

**return** character;

}

}

# LAB 8

# note\_compare.c

*//complete note\_compare()*

*//Completed by:*

*//On:*

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

*//There are 10 octaves (0 to 9) and 12 notes (0 to 11)*

**struct** note {

int octave;

int key;

**struct** note \*next;

};

int note\_compare(**struct** note \*a, **struct** note \*b);

**struct** note \*create\_note(int octave, int key);

int main(void) {

printf("this should brea**\n**");

int octave, key;

scanf("%d %d", &octave, &key);

**struct** note a = {octave, key};

scanf("%d %d", &octave, &key);

**struct** note b = {octave, key};

int compared = note\_compare(&a, &b);

**if** (compared == 1) {

printf("a is higher than b**\n**");

} **else** **if** (compared == -1) {

printf("b is higher than a**\n**");

} **else** {

printf("a and b are equal**\n**");

}

**return** 0;

}

*//Returns 1 if a is higher than b*

*// -1 if b is higher than a*

*// 0 if they are equal*

int note\_compare(**struct** note \*a, **struct** note \*b) {

assert(a != NULL && b != NULL);

**if** (a->octave < b->octave) {

**return** -1;

} **else** **if** (a->octave > b->octave){

**return** 1;

} **else** {

**if** (a->key < b ->key){

**return** -1;

} **else** **if** (a->key > b->key) {

**return** 1;

} **else** {

**return** 0;

}

}

}

**note\_subtract.c**

*//complete note\_subtract() and note\_print()*

*//Completed by:*

*//On:*

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

*//There are 10 octaves (0 to 9) and 12 notes (0 to 11)*

**struct** note {

int octave;

int key;

**struct** note \*next;

};

**struct** note \*note\_subtract(**struct** note \*higher, **struct** note \*lower);

**struct** note \*create\_note(int octave, int key);

void print\_note(**struct** note \*n);

int main(void) {

int octave, key;

scanf("%d %d", &octave, &key);

**struct** note \*a = create\_note(octave, key);

scanf("%d %d", &octave, &key);

**struct** note \*b = create\_note(octave, key);

**struct** note \*diff = note\_subtract(a, b);

print\_note(diff);

**return** 0;

}

*//Mallocs a note and creates it given an octave and a key*

**struct** note \*create\_note(int octave, int key) {

**struct** note \*new\_note = malloc(**sizeof**(**struct** note));

new\_note->octave = octave;

new\_note->key = key;

new\_note->next = NULL;

**return** new\_note;

}

*// For a note with octave 0, and note 9,*

*// `print\_note` should print:*

*// "0 09\n"*

void print\_note(**struct** note \*n) {

printf("%d %02d**\n**", n->octave, n->key);

}

*//Returns a pointer to a malloced struct containing the difference between a*

*//higher and a lower note*

**struct** note \*note\_subtract(**struct** note \*higher, **struct** note \*lower) {

int key = higher->key - lower->key;

int octave = higher->octave - lower->octave;

**if** (key < 0) {

octave = octave - 1;

key = 12 + key;

}

**return** create\_note(octave, key);

}

# list\_print.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

void print(**struct** node \*head);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

int main(int argc, char \*argv[]) {

*// create linked list from command line arguments*

**struct** node \*head = strings\_to\_list(argc - 1, &argv[1]);

print(head);

**return** 0;

}

*// print a linked list in this format:*

*// 17 -> 34 -> 51 -> 68 -> X*

void print(**struct** node \*head) {

**struct** node \*n = head;

**while** (n != NULL) {

printf("%d -> ", n->data);

n = n->next;

}

printf("X**\n**");

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

**for** (int i = len - 1; i >= 0; i = i - 1) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

}

**return** head;

}

Alternative solution for list\_print.c

#include *<stdio.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

*// print a linked list in this format:*

*// 17 -> 34 -> 51 -> 68 -> X*

void print(**struct** node \*head) {

**if** (head == NULL) {

printf("X**\n**");

}

printf("%d -> ", head->data);

print(head->next);

}

# list\_insert\_head.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

**struct** node \*insert\_head(int value, **struct** node \*head);

**struct** node \*create\_node(int data, **struct** node \*next);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

void print\_list(**struct** node \*head);

int main(int argc, char \*argv[]) {

int value;

scanf("%d", &value);

*// create linked list from command line arguments*

**struct** node \*head = NULL;

**if** (argc > 1) {

*// list has elements*

head = strings\_to\_list(argc - 1, &argv[1]);

}

**struct** node \*new\_head = insert\_head(value, head);

print\_list(new\_head);

**return** 0;

}

*// Insert a new node containing value at the start of the linked list.*

*// The head of the new list is returned.*

**struct** node \*insert\_head(int value, **struct** node \*head) {

**struct** node \*new\_node = malloc(**sizeof** (**struct** node));

**if** (new\_node == NULL) {

fprintf(stderr, "out of memory**\n**");

exit(1);

}

new\_node->data = value;

new\_node->next = head;

**return** new\_node;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

int i = len - 1;

**while** (i >= 0) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

i -= 1;

}

**return** head;

}

*// print linked list*

void print\_list(**struct** node \*head) {

printf("[");

**struct** node \*n = head;

**while** (n != NULL) {

*// If you're getting an error here,*

*// you have returned an invalid list*

printf("%d", n->data);

**if** (n->next != NULL) {

printf(", ");

}

n = n->next;

}

printf("]**\n**");

}

# list\_contains.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

int contains(int value, **struct** node \*head);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

int main(int argc, char \*argv[]) {

int value;

scanf("%d", &value);

*// create linked list from command line arguments*

**struct** node \*head = NULL;

**if** (argc > 1) {

*// list has elements*

head = strings\_to\_list(argc - 1, &argv[1]);

}

int result = contains(value, head);

printf("%d**\n**", result);

**return** 0;

}

*// Return 1 if value occurs in linked list, 0 otherwise*

int contains(int value, **struct** node \*head) {

**struct** node \*n = head;

**while** (n != NULL) {

**if** (n->data == value) {

**return** 1;

}

n = n->next;

}

**return** 0;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

int i = len - 1;

**while** (i >= 0) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

i -= 1;

}

**return** head;

}

Alternative solution for list\_contains.c

#include *<stdio.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

*// Return 1 if value occurs in linked list, 0 otherwise*

int contains(int value, **struct** node \*head) {

**if** (head == NULL) {

**return** 0;

}

**return** (head->data == value) || contains(value, head->next);

}

# list\_insert\_nth.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

**struct** node \*insert\_nth(int n, int value, **struct** node \*head);

**struct** node \*create\_node(int data, **struct** node \*next);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

void print\_list(**struct** node \*head);

int main(int argc, char \*argv[]) {

int n;

scanf("%d", &n);

int value;

scanf("%d", &value);

*// create linked list from command line arguments*

**struct** node \*head = NULL;

**if** (argc > 1) {

*// list has elements*

head = strings\_to\_list(argc - 1, &argv[1]);

}

**struct** node \*new\_head = insert\_nth(n, value, head);

print\_list(new\_head);

**return** 0;

}

*// Insert a new node containing value at position n of the linked list.*

*// if n == 0, node is inserted at start of list*

*// if n >= length of list, node is appended at end of list*

*// The head of the new list is returned.*

**struct** node \*insert\_nth(int n, int value, **struct** node \*head) {

**struct** node \*new\_node = malloc(**sizeof** (**struct** node));

**if** (new\_node == NULL) {

fprintf(stderr, "out of memory**\n**");

exit(1);

}

new\_node->data = value;

*// new node is head of list*

**if** (head == NULL || n == 0) {

new\_node->next = head;

**return** new\_node;

}

int i = n - 1;

**struct** node \*p = head;

**while** (p->next != NULL && i > 0) {

p = p->next;

i = i - 1;

}

new\_node->next = p->next;

p->next = new\_node;

**return** head;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

int i = len - 1;

**while** (i >= 0) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

i -= 1;

}

**return** head;

}

*// print linked list*

void print\_list(**struct** node \*head) {

printf("[");

**struct** node \*n = head;

**while** (n != NULL) {

*// If you're getting an error here,*

*// you have returned an invalid list*

printf("%d", n->data);

**if** (n->next != NULL) {

printf(", ");

}

n = n->next;

}

printf("]**\n**");

}

Alternative solution for list\_insert\_nth.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

*// Insert a new node containing value at position n of the linked list.*

*// if n == 0, node is inserted at start of list*

*// if n >= length of list, node is appended at end of list*

*// The head of the new list is returned.*

*// recursive version*

**struct** node \*insert\_nth(int n, int value, **struct** node \*head) {

**if** (n > 0 && head != NULL) {

head->next = insert\_nth(n - 1, value, head->next);

**return** head;

}

**struct** node \*new\_node = malloc(**sizeof** (**struct** node));

**if** (new\_node == NULL) {

fprintf(stderr, "out of memory**\n**");

exit(1);

}

new\_node->data = value;

new\_node->next = head;

**return** new\_node;

}

# list\_insert\_tail.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

**struct** node \*insert\_tail(int value, **struct** node \*head);

**struct** node \*last(**struct** node \*head);

**struct** node \*create\_node(int data, **struct** node \*next);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

void print\_list(**struct** node \*head);

int main(int argc, char \*argv[]) {

int value;

scanf("%d", &value);

*// create linked list from command line arguments*

**struct** node \*head = NULL;

**if** (argc > 1) {

*// list has elements*

head = strings\_to\_list(argc - 1, &argv[1]);

}

**struct** node \*new\_head = insert\_tail(value, head);

print\_list(new\_head);

**return** 0;

}

*// Add a new node containing value at the end of the linked list.*

*// The head of the new list is returned.*

**struct** node \*insert\_tail(int value, **struct** node \*head) {

**struct** node \*new\_node = malloc(**sizeof** (**struct** node));

**if** (new\_node == NULL) {

fprintf(stderr, "out of memory**\n**");

exit(1);

}

new\_node->data = value;

new\_node->next = NULL;

*// empty list is a special case*

*// new node is now the head of the now 1 element list*

**if** (head == NULL) {

**return** new\_node;

}

**struct** node \*l = last(head);

l->next = new\_node;

**return** head;

}

*// return pointer to last node in list*

*// NULL is returned if list is empty*

**struct** node \*last(**struct** node \*head) {

**if** (head == NULL) {

**return** NULL;

}

**struct** node \*n = head;

**while** (n->next != NULL) {

n = n->next;

}

**return** n;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

int i = len - 1;

**while** (i >= 0) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

i -= 1;

}

**return** head;

}

*// print linked list*

void print\_list(**struct** node \*head) {

printf("[");

**struct** node \*n = head;

**while** (n != NULL) {

*// If you're getting an error here,*

*// you have returned an invalid list*

printf("%d", n->data);

**if** (n->next != NULL) {

printf(", ");

}

n = n->next;

}

printf("]**\n**");

}

# LAB 9

# list\_increasing.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

int increasing(**struct** node \*head);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

int main(int argc, char \*argv[]) {

*// create linked list from command line arguments*

**struct** node \*head = strings\_to\_list(argc - 1, &argv[1]);

int result = increasing(head);

printf("%d**\n**", result);

**return** 0;

}

*// return 1 if values in a linked list are in increasing order,*

*// return 0, otherwise*

int increasing(**struct** node \*head) {

*// If the list is empty, it's considered increasing, so return 1.*

**if** (head == NULL) {

**return** 1;

}

*// Assume that it is increasing, and look for evidence*

*// that proves otherwise.*

int is\_increasing = 1;

**struct** node \*curr = head;

**while** (curr->next != NULL) {

*// If this one is not less than the next one,*

*// the list definitely isn't increasing*

*// (since these two nodes are out of order).*

**if** (curr->data >= curr->next->data) {

is\_increasing = 0;

}

curr = curr->next;

}

*// At this point, if is\_increasing is still 1, we didn't find*

*// any nodes that were out of order.*

*//*

*// However, if we did find any nodes that were out of order,*

*// we set it to 0 in the loop above.*

*//*

*// So, is\_increasing contains the answer to return.*

**return** is\_increasing;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

**for** (int i = len - 1; i >= 0; i = i - 1) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

}

**return** head;

}

Alternative solution for list\_increasing.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

int increasing(**struct** node \*head);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

int main(int argc, char \*argv[]) {

*// create linked list from command line arguments*

**struct** node \*head = strings\_to\_list(argc - 1, &argv[1]);

int result = increasing(head);

printf("%d**\n**", result);

**return** 0;

}

*// return 1 if values in a linked list are in increasing order,*

*// return 0, otherwise*

int increasing(**struct** node \*head) {

**if** (head == NULL) {

**return** 1;

}

**struct** node \*p = head;

**while** (p->next != NULL) {

**if** (p->data >= p->next->data) {

**return** 0;

}

p = p->next;

}

**return** 1;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

**for** (int i = len - 1; i >= 0; i = i - 1) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

}

**return** head;

}

Alternative solution for list\_increasing.c

#include *<stdio.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

*// return 1 if values in a linked list in increasing order*

*// recursive solution*

int increasing(**struct** node \*head) {

**if** (head == NULL || head->next == NULL) {

**return** 1;

} **else** **if** (head->data >= head->next->data) {

**return** 0;

} **else** {

**return** increasing(head->next);

}

}

# list\_delete\_first.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

**struct** node \*delete\_first(**struct** node \*head);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

void print\_list(**struct** node \*head);

int main(int argc, char \*argv[]) {

*// create linked list from command line arguments*

**struct** node \*head = strings\_to\_list(argc - 1, &argv[1]);

**struct** node \*new\_head = delete\_first(head);

print\_list(new\_head);

**return** 0;

}

*// Delete the first node in list.*

*// The deleted node is freed.*

*// The head of the list is returned.*

**struct** node \*delete\_first(**struct** node \*head) {

**if** (head == NULL) {

*// list is empty no node to delete*

**return** NULL;

}

**struct** node \*new\_head = head->next;

free(head);

**return** new\_head;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

**for** (int i = len - 1; i >= 0; i = i - 1) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

}

**return** head;

}

*// print linked list*

void print\_list(**struct** node \*head) {

printf("[");

**for** (**struct** node \*n = head; n != NULL; n = n->next) {

*// If you're getting an error here,*

*// you have returned an invalid list*

printf("%d", n->data);

**if** (n->next != NULL) {

printf(", ");

}

}

printf("]**\n**");

}

# list\_delete\_contains.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

**struct** node \*delete\_contains(int value, **struct** node \*head);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

void print\_list(**struct** node \*head);

int main(int argc, char \*argv[]) {

int value;

scanf("%d", &value);

*// create linked list from command line arguments*

**struct** node \*head = strings\_to\_list(argc - 1, &argv[1]);

**struct** node \*new\_head = delete\_contains(value, head);

print\_list(new\_head);

**return** 0;

}

*// Delete the first node in the list containing i*

*// The deleted node is freed.*

*// If no node contains i, the list is not changed*

*// The head of the list is returned.*

**struct** node \*delete\_contains(int value, **struct** node \*head) {

**if** (head == NULL) {

*// list is empty no node to delete*

**return** NULL;

} **else** **if** (head->data == value) {

*// deleting the first node*

**struct** node \*new\_head = head->next;

free(head);

**return** new\_head;

} **else** **if** (head->next == NULL) {

*// first node is the only node*

*// first node is definitely not value*

**return** head;

}

**struct** node \*n = head;

*// find node before first node containing i*

**while** (n->next->next != NULL && n->next->data != value) {

n = n->next;

}

**if** (n->next->data == value) {

**struct** node \*new\_next = n->next->next;

free(n->next);

n->next = new\_next;

}

**return** head;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

int i = len - 1;

**while** (i >= 0) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

i -= 1;

}

**return** head;

}

*// print linked list*

void print\_list(**struct** node \*head) {

printf("[");

**struct** node \*n = head;

**while** (n != NULL) {

*// If you're getting an error here,*

*// you have returned an invalid list*

printf("%d", n->data);

**if** (n->next != NULL) {

printf(", ");

}

n = n->next;

}

printf("]**\n**");

}

# list\_intersection\_size.c

#include *<stdio.h>*

#include *<string.h>*

#include *<stdlib.h>*

#include *<assert.h>*

**struct** node {

**struct** node \*next;

int data;

};

int intersection\_size(**struct** node \*head1, **struct** node \*head2);

int member(int i, **struct** node \*head);

**struct** node \*strings\_to\_list(int len, char \*strings[]);

int main(int argc, char \*argv[]) {

*// create two linked lists from command line arguments*

int dash\_arg = argc - 1;

**while** (dash\_arg > 0 && strcmp(argv[dash\_arg], "-") != 0) {

dash\_arg = dash\_arg - 1;

}

**struct** node \*head1 = strings\_to\_list(dash\_arg - 1, &argv[1]);

**struct** node \*head2 = strings\_to\_list(argc - dash\_arg - 1, &argv[dash\_arg + 1]);

int result = intersection\_size(head1, head2);

printf("%d**\n**", result);

**return** 0;

}

*// return the number of values which occur in both linked lists*

*// no value is repeated in either list*

int intersection\_size(**struct** node \*head1, **struct** node \*head2) {

int num\_both = 0;

**struct** node \*p = head1;

**while** (p != NULL) {

**if** (member(p->data, head2)) {

num\_both = num\_both + 1;

}

p = p->next;

}

**return** num\_both;

}

*// return 1 if i occurs in list, 0 otherwise*

int member(int i, **struct** node \*head) {

**struct** node \*p = head;

**while** (p != NULL) {

**if** (p->data == i) {

**return** 1;

}

p = p->next;

}

**return** 0;

}

*// create linked list from array of strings*

**struct** node \*strings\_to\_list(int len, char \*strings[]) {

**struct** node \*head = NULL;

**for** (int i = len - 1; i >= 0; i = i - 1) {

**struct** node \*n = malloc(**sizeof** (**struct** node));

assert(n != NULL);

n->next = head;

n->data = atoi(strings[i]);

head = n;

}

**return** head;

}

Alternative solution for list\_intersection\_size.c

#include *<stdio.h>*

**struct** node {

**struct** node \*next;

int data;

};

*// return 1 if i occurs in list, 0 otherwise*

int member(int i, **struct** node \*head) {

**if** (head == NULL) {

**return** 0;

} **if** (head->data == i) {

**return** 1;

} **else** {

**return** member(i, head->next);

}

}

*// return the number of values which occur in both linked lists*

*// no value is repeated in either list*

*// cute, recursive solution*

int intersection\_size(**struct** node \*head1, **struct** node \*head2) {

**if** (head1 == NULL) {

**return** 0;

} **else** {

**return** member(head1->data, head2) + intersection\_size(head1->next, head2);

}

}

# TEST 5

# array\_print\_pointer.c

*// COMP1511 Array Print Pointer*

*// Print out the contents of an array, starting*

*// from index 0 and ending by printing out*

*// a particular element that is also being*

*// pointed at by a given pointer*

*// Marc Chee, March 2020*

#include *<stdio.h>*

#define LENGTH 10

void array\_print\_pointer(int nums[LENGTH], int \*last);

*// This is a simple main function that you can use to test your array\_print\_pointer*

*// function.*

*// It will not be marked - only your array\_print\_pointer function will be marked.*

*//*

*// Note: the autotest does not call this main function!*

*// It calls your array\_print\_pointer function directly.*

*// Any changes that you make to this main function will not affect the autotests.*

int main(int argc, char \*argv[]){

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

int \*last = &nums[5];

*//Pass in the address of the sum and product variables*

array\_print\_pointer(nums, last);

**return** 0;

}

*// Print an array from the beginning until it reaches*

*// a pointer. Assumes that the pointer is aimed at one*

*// of the array elements.*

void array\_print\_pointer(int nums[LENGTH], int \*last) {

int i = 0;

int stop = 0;

**while** (stop == 0 && i < LENGTH) { *// printed i numbers*

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

**if** (&nums[i] == last) { *// last is aimed at nums[i]*

stop = 1;

}

i++;

}

}

# TEST 7

# reverse\_lines.c

*// Written 3/5/2018 by Andrew Taylor (andrewt@unsw.edu.au)*

*// Reads lines and writes them out with the characters of each line in reverse order.*

*// Stop when it reaches the end of input.*

#include *<stdio.h>*

#define MAX\_LINE\_CHARS 4096

int main(void) {

char line[MAX\_LINE\_CHARS];

*// get the lines*

**while** (fgets(line, MAX\_LINE\_CHARS, stdin) != NULL) {

*// find the size of this line*

int size = 0;

**while** ((line[size] != '\n') && (line[size] != '\0')) {

*// not the end yet*

size = size + 1;

}

*// now print in reverse*

int j = size - 1;

**while** (j >= 0) {

printf("%c", line[j]);

j = j - 1;

}

printf("**\n**");

}

**return** 0;

}

# remove\_repeated\_lines.c

*// Written 3/5/2018 by Andrew Taylor (andrewt@unsw.edu.au)*

*// Reads lines and writes them out to stdout if they haven't been previously seen*

*// Stops when it reaches the end of input.*

#include *<stdio.h>*

#include *<string.h>*

#define MAX\_LINE\_CHARS 256

#define MAX\_N\_LINES 256

int main(void) {

char lines[MAX\_N\_LINES][MAX\_LINE\_CHARS];

int line\_number = 0;

**while** (fgets(lines[line\_number], MAX\_LINE\_CHARS, stdin) != NULL && line\_number < MAX\_N\_LINES) {

int line\_seen\_previously = 0;

int i = 0;

**while** (i < line\_number && !line\_seen\_previously) {

**if** (strcmp(lines[line\_number], lines[i]) == 0) {

line\_seen\_previously = 1;

}

i = i + 1;

}

**if** (!line\_seen\_previously) {

printf("%s", lines[line\_number]);

line\_number = line\_number + 1;

}

}

**return** 0;

}

# TEST 8

# string\_to\_lower\_args.c

*// Read command line args and print them out*

*// converting all upper case letters to lower*

*// Marc Chee, April 2020*

#include *<stdio.h>*

#include *<ctype.h>*

void print\_lower\_word(char \*word);

int main(int argc, char \*argv[]) {

int i = 1;

**while** (i < argc) {

print\_lower\_word(argv[i]);

putchar(' ');

i++;

}

putchar('\n');

}

void print\_lower\_word(char \*word) {

int i = 0;

**while**(word[i] != '\0') {

putchar(tolower(word[i]));

i++;

}

}

# struct\_create.c

#include *<stdio.h>*

#include *<stdlib.h>*

*// Do not edit this struct. You may use it exactly as it is*

*// but you cannot make changes to it*

**struct** numbers {

int first;

int second;

};

*// create an instance of the struct and return a pointer to it*

**struct** numbers \*struct\_create(int a, int b) {

**struct** numbers \*n = malloc(**sizeof** (**struct** numbers));

n->first = a;

n->second = b;

**return** n;

}

*// This is a simple main function which could be used*

*// to test your struct\_create function.*

*// It will not be marked.*

*// Only your struct\_create function will be marked.*

int main(void) {

int first = 1;

int second = 2;

**struct** numbers\* n = struct\_create(first, second);

printf("%d, %d**\n**", n->first, n->second);

**return** 0;

}

# evolve\_pokemon.c

#include *<stdio.h>*

#include *<stdlib.h>*

#include *<string.h>*

#define MAX\_NAME\_SIZE 50

*// Do not edit this struct. You may use it exactly as it is*

*// but you cannot make changes to it*

**struct** pokemon {

char name[MAX\_NAME\_SIZE];

**struct** pokemon \*evolution;

};

*// Create a pokemon*

**struct** pokemon \*create\_pokemon(char name[MAX\_NAME\_SIZE]) {

**struct** pokemon \*p = malloc(**sizeof** (**struct** pokemon));

strcpy(p->name, name);

p->evolution = NULL;

**return** p;

}

*// Link a pokemon to another that it evolves into*

void evolve\_pokemon(**struct** pokemon \*base, **struct** pokemon \*evolution) {

base->evolution = evolution;

}

*// Print out the evolution of a pokemon*

void print\_evolution(**struct** pokemon \*p) {

**while**(p != NULL) {

printf("%s**\n**", p->name);

p = p->evolution;

}

}

*// This is a simple main function which could be used*

*// to test your pokemon functions.*

*// It will not be marked.*

*// Only your pokemon functions will be marked.*

int main(void) {

**struct** pokemon \*pikachu = create\_pokemon("Pikachu");

**struct** pokemon \*raichu = create\_pokemon("Raichu");

evolve\_pokemon(pikachu, raichu);

print\_evolution(pikachu);

**return** 0;

}