#### *T.P. N° 11: Punteros a función y T.A.D.*

***Ejercicio 2***

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "getnum.h"

#define BORRA\_BUFFER while (getchar() != '\n');

#define CANT\_FUNC 3

#define PASOS 100000

#define EPSILON 0.000001

#define MEM\_INC 10

typedef struct {

double a,b ;

} tipoIntervalo;



typedef struct {

int cant;

int dim;

tipoIntervalo \* lista;

} tipoRaices;

*/\* Estructuras necesarias para el manejo del menú \*/*

typedef struct{

char \* titulo;

double (\*funcion)(double);

}tipoItemMenu;

typedef struct{

int cantItems;

tipoItemMenu \* items;

} tipoMenu;

*/\* Dado un intervalo y un puntero a una función busca subintervalos con raíces \*/*

tipoRaices buscar\_raices(tipoIntervalo, double (\*funcion)(double));

*/\* Almacena un intervalo en la estructura tipoRaices, de no hay espacio ejecuta realloc() \*/*

int almacenarRaiz(tipoRaices \* resp, tipoIntervalo inter);

*/\* Libera la respuesta cuando deja de ser necesaria \*/*

void freeRaices(tipoRaices resp);

*/\* Arma el menú y devuelve la opción elegida. Nota: asume menos de 10 opciones \*/*

int armarMenu(tipoMenu menu);

*/\* función propia para evaluación \*/*

double mifuncion(double);

*/\* Realiza la lectura de los extremos del intervalo \*/*

void leerExtremos(tipoIntervalo \*extremos);

*/\* Función que imprime los resultados en salida estándar \*/*

void verResultados(tipoRaices respuesta);

int

main(void)

{

tipoItemMenu tabla[CANT\_FUNC] = {{"Seno",sin},{"Coseno",cos},

{"x2 - 4",mifuncion}};

tipoMenu menu = {CANT\_FUNC, tabla};

int opcion;

tipoRaices respuesta;

tipoIntervalo extremos;

while (opcion = armarMenu(menu))

{

leerExtremos(&extremos);

respuesta = buscar\_raices(extremos,tabla[opcion-1].funcion);

verResultados(respuesta);

if (respuesta.cant >=0)

reeRaices(respuesta);

}

return 0;

}

void

verResultados(tipoRaices respuesta)

{

int i;

if (respuesta.dim == -1) */\* Corresponde a error de malloc() \*/*

printf("Error en el almacenamiento de raíces\n");

else if (respuesta.cant > 0)

{

for (i=0; i<respuesta.cant; i++)

printf("\nRaíz %d en [%f,%f]\n",

i+1,respuesta.lista[i].a, respuesta.lista[i].b);

}

else

printf("\nNo hay raíces en ese intervalo\n");

}

double

mifuncion(double x)

{

return (x\*x - 4.0);

}

int

armarMenu(tipoMenu menu)

{

int i;

*/\* Imprime las opciones \*/*

for (i=0; i < menu.cantItems; i++)

printf("%d - %s\n", i+1, menu.items[i].titulo);

printf("0 - Salir\n");

*/\* Espera respuesta del usuario \*/*

do

i = getchar()-'0';

while (i < 0 || i > menu.cantItems);

BORRA\_BUFFER

return i;

}

tipoRaices

buscar\_raices(tipoIntervalo extr, double (\*funcion)(double))

{

double x,fx,fxant, incremento;

tipoRaices respuesta;

int esRaiz = 0, error = 0;

tipoIntervalo intActual;

incremento = (extr.b - extr.a) / PASOS ;



if (incremento == 0 )

incremento = 0.00001;

respuesta.cant = -1;

respuesta.dim = 0;

*/\* Se recorre el intervalo, evaluando la función en cada punto \*/*

x = extr.a;

fxant = fx = (\*funcion)(x);

while ( x <= extr.b && !error )

{

if ( fx\*fxant < 0 || fabs(fx) < EPSILON)

{

*/\* Con el flag esRaiz se controla que no se indique más de una vez la misma raíz.*

*\*\* Si el punto anterior ya era raíz, entonces extender el intervalo*

*\*/*

if ( ! esRaiz )

{

respuesta.cant++;

intActual.a = x - incremento;

}

*/\* Si se detectó un cambio de signo es porque hay una raíz entre el x*

*\*\* anterior y el actual por lo tanto el fin del intervalo es x.*

*\*\* En cambio si x es raíz el extremo derecho del intervalo será el próximo x*

*\*/*

if ( fx \* fxant < 0 )

intActual.b = x;

else

intActual.b = x + incremento;

*/\* Se almacena el nuevo intervalo, o actualiza uno existente \*/*

if ( !almacenarRaiz(&respuesta, intActual) )

error = 1;

esRaiz = 1;

}

else

esRaiz = 0;

*/\* Paso al próximo punto \*/*

x += incremento;

fxant = fx;



fx = (\*funcion)(x);

}

respuesta.cant++; */\* Se incrementa cant porque internamente representa la posición actual,*

*\*\* y al terminar debe indicar cantidad de raíces \*/*

return respuesta;

}

int

almacenarRaiz(tipoRaices \* resp, tipoIntervalo inter)

{

tipoIntervalo \* aux;

if (resp->dim == 0) */\* Si es la primera vez que se invoca \*/*

{

resp->lista = malloc(sizeof(tipoIntervalo) \* MEM\_INC);

if (!resp->lista)

{

resp->dim = -1; */\* Si falla el pedido dim queda en –1 \*/*

return 0;

}

resp->dim = MEM\_INC;

}

else if (resp->cant == resp->dim) */\* Si es necesario pedir más memoria \*/*

{

resp->dim += MEM\_INC;

aux = realloc(resp->lista, resp->dim \* sizeof(tipoIntervalo));

if (aux == NULL)

{

free(resp->lista);

resp->dim = -1;

return 0;

}

resp->lista = aux;

}

*/\* Se almacena/actualiza el intervalo \*/*

resp->lista[resp->cant] = inter;

return 1;

}

void

freeRaices(tipoRaices resp)

{

free(resp.lista);

}

void

leerExtremos(tipoIntervalo \*extremos)

{

float a, b;

*/\* leer extremos del intervalo \*/*

a = getfloat("Extremo 1:");

b = getfloat("Extremo 2:");

if (a<b)

{

extremos->a = a; extremos->b = b;

}

else

{

extremos->b = a; extremos->a = b;

}

}

**Para pensar:**

**¿Cómo se podría reutilizar esta función para obtener resultados más precisos?**

***Ejercicio 3***

1. p es un puntero a una función que acepta como argumento un puntero a un arreglo de enteros y devuelve un puntero a entero
2. p es un arreglo de 10 punteros a enteros
3. p es un puntero a un arreglo de 10 enteros
4. p es una función que devuelve un puntero a entero
5. p es una función que acepta como argumento un puntero a carácter y devuelve un entero
6. p es una función que acepta como argumento un puntero a carácter y devuelve un puntero a entero
7. p es un puntero a una función que acepta como argumento un puntero a carácter y devuelve un entero
8. p es una función que acepta como argumento un puntero a carácter y devuelve un puntero a un arreglo de 10 enteros
9. p es una función que acepta como argumento un puntero a un arreglo de caracteres y devuelve un entero
10. p es una función que acepta como argumento un arreglo de punteros a caracteres. devuelve un entero
11. p es una función que acepta como argumento un arreglo de caracteres y devuelve un puntero a entero
12. p es una función que acepta como argumento un puntero a un arreglo de caracteres y devuelve un puntero a entero
13. p es una función que acepta como argumento un arreglo de punteros a caracteres y devuelve un puntero a entero
14. p es un puntero a una función que acepta como argumento un puntero a un arreglo de caracteres y devuelve un entero
15. p es un puntero a una función que acepta como argumento un arreglo de punteros a caracteres, devuelve un puntero a entero
16. p es un arreglo de 10 punteros a función, donde cada función devuelve un entero
17. p es un arreglo de 10 punteros a función, donde cada función acepta como argumento un caracter y devuelve un entero
18. p es un arreglo de 10 punteros a función, donde cada función acepta como argumento un caracter y devuelve un puntero a entero
19. p es un arreglo de 10 punteros a función, donde cada función acepta como argumento un puntero a carácter y devuelve un puntero a entero
20. p es un puntero a una función que acepta como argumento un arreglo de punteros a caracteres, devuelve un entero

### *Ejercicio 4*

1. double \* (\* p[] ) (void)
2. int (\* p(char \*a)) [3][5]
3. int \* (\* (\*p) [N]) (int a)

### ***Ejercicio 6***

**Notar que al usar listElementT en el .h nos impide tener simultáneamente dos listas de distinto tipo.**

listElementT

get(listADT list, unsigned int idx)

{

int i;

nodeP auxi;

if ( idx >= list->size )

return NULL;

auxi = list->first;

for(i=0; i<idx; i++)

auxi = auxi->tail;

return auxi->head;

}

### *Ejercicio 7*

void

map(listADT list, listElementT (\*fn)(listElementT))

{

nodeP aux = list->first;

while ( aux != NULL) {

aux->head = fn(aux->head);

aux = aux->tail;

}

}

### *Ejercicio 8*

Al basarnos en el TAD de listas debemos tener cuidado de no usar funciones con el mismo nombre, como size, elementBelongs, etc.

***Ejercicio 9***

a) El usuario puede castear el puntero y modificar el dato que desee. Por ejemplo, en el caso de la lista, donde la estructura es

typedef struct listCDT

{

struct node \* first;

int size;

struct node \* next;

};

el usuario –si sabe cómo está definida la estructura- puede escribir el siguiente código para pisar un campo:

struct soyUnHacker

{

int \* a; // cualquier tipo de puntero

int b;

int \* c;

};

int main(void)

{

listADT s = newList();

struct soyUnHacker \* p = (struct soyUnHacker \*) s;

p->b = 10; // modifico el size

b) Para que no pueda modificarlo, no se debe entregar un puntero. Una solución es mantener un vector interno con los punteros a las estructuras y retornarle al usuario un entero como “id” del TAD, donde ese entero represente el índice dentro del vector.

Ejemplo de implementación parcial:

#include <stdlib.h>

#include "setADT.h"

#define MAX 50

static listADT lists[MAX] = {NULL};

struct listCDT

{

nodeP first;

unsigned int size;

nodeP next;

};

int

newList()

{

int i;

listADT list;

for (i=0; i < MAX && lists[i] != NULL; i++)

;

if ( i==MAX ) *// No hay espacio para nuevas listas*

return -1;

list = calloc(1, sizeof(struct listCDT));

if ( list == NULL )

return -1;

lists[i] = list;

return i;

}

void

freeList(unsigned int list)

{

listADT aux = lists[list]; *// Asumimos un índice válido*

nodeP curr=list->first, aux;

while (curr != NULL) {

aux = curr->tail;

free(curr);

curr = aux;

}

free(list);

lists[i] = NULL;

}

int

isEmpty(unsigned int list)

{

return listIsEmpty(lists[list]); *// Asumimos un índice válido*

}

Obviamente se puede cambiar el vector para que en vez de ser estático sea dinámico o una lista. También puede agregarse en cada método que valide que el índice recibido esté dentro de los límites del vector y no sea NULL el valor en esa posición.

***Ejercicio 10***

Básicamente es una lista pero donde cada nodo además del elemento lleva la cuenta de cuantos hay.

También se podría haber hecho con un vector, donde cada elemento es una estructura que tiene elemType y cantidad