Clase 11. Apuntadores a funciones, librerias estándar y algoritmos.

Sobre examen

Tarea 6

prog6.1 Programar la parte 1 del examen.

1. Definir la estructura ELEMENTO como un tipo (typedef).
2. Definir la estuctura MATRIZRALA como un tipo.
3. Escribir el programa (con la funcion que lee) requiriendo y devolviendo memoria dinámica para la estructura matríz, y para un arreglo unidimensional de estructuras tipo ELEMENTO.

prog6.1a Escribir un programa que haga lo mismo que el anterior, pero sin utilizar MATRIZRALA, solo utilizando un vector de memoria estática de ELEMENTO.

prog6.1b Escribir el mismo programa que el anterior utilizando una variable MATRIZRALA definida con memoria estática y un arreglo ELEMENTO dinámico.

prog6.1c Escribir el mismo programa con la memoria para MATRIZRALA dinámica, pero declarando en el tipo MATRIZRALA un arreglo estático de ELEMENTO.

prog6.1d Escribir el mismo programa con MATRIZRALA en memoria estática y declarando un arreglo de apuntadores a ELEMENTO estático y pidiendo memoria para cada ELEMENTO de manera dinámica.

Apuntadores a funciones

Apuntadores a funciones

Una función se almacena en una dirección de memoria, en donde se almacenan instrucciones en lugar de datos.


```
1  #include <stdio.h>
   #include <stdlib.h>
3  double sqfx(double x);
   double sqfx(double x){
5    return x*x;
   }
7  int main(int argc, char *argv[]){
     double x[100];
9    double *y=(double*)malloc(100*sizeof(double));
     char str[128];
11   int *i=(int*)malloc(200*sizeof(int));
     printf("Dir. funciones (code): %p %p\n",sqfx,main);
13   printf("Dir. datos estaticos(stack): %p %p\n",x,str);
     printf("Dir. datos dinamicoa(heap): %p %p\n",y,i);
15   free(y);  free(i);
   return 0;
17 }
```

Apuntadores a funciones, ejemplo


```c
#include <stdio.h>
#include <stdlib.h>
double sqfx(double x);
double sqfx(double x){
  return x*x;
}
int main(int argc, char *argv[]){
  double x[100];
  double *y=(double*)malloc(100*sizeof(double));
  char str[128];
  int *i=(int*)malloc(200*sizeof(int));
  printf("Dir. funciones (code): %p %p\n",sqfx,main);
  printf("Dir. datos estaticos(stack): %p %p\n",x,str);
  printf("Dir. datos dinamico(heap): %p %p\n",y,i);
  free(y);  free(i);
return 0;
}
```

```
Dir. funciones (code): 0x4005ec 0x400613
Dir. datos estaticos(stack): 0x7fff3f510520 0x7fff3f5104a0
Dir. datos dinamico(heap): 0x1e92010 0x1e92340
```

Sintáxis

La sintaxis es:

TIPO_DE_RETORNO
(*APUNTADOR_A_FUNCION)(TIPO_DE_ARGUMENTOS);

```
1  #include <stdio.h>
   #include <stdlib.h>
3  double sqfx(double x);
   double sqfx(double x){
5    return x*x;
   }
7  int main(int argc, char *argv[]){
   double (*funptr)(double);
9      funptr=sqfx;
       printf("Cuadrado de 5=%lf\n",funptr(5.0));
11 return 0;
   }
```

```
Cuadrado de 5=25.000000
```

Apuntadores a funciones como argumento de funciones


```
double sq(double x){
  return x*x;
}
double sum(double x){
  return x+x;
}
double oper(double x, double (*ptr)(double)){
  return ptr(x);
}

int main(int argc, char *argv[]){
    printf("Cuadrado de 5=%lf\n",oper(5.0,sq));
    printf("Suma de 5=%lf\n",oper(5.0,sum));
return 0;
}
```

```
Cuadrado de 5=25.000000
Suma de 5=10.000000
```

Funciones que devuelven apuntadores a funciones


```
1  double sq(double x){
     return x*x;
3  }
   double sum(double x){
5    return x+x;
   }
7  double (*oper(char *op))(double){
     if (strcmp(op,"sq")==0)
9      return sq;
     if (strcmp(op,"sum")==0)
11     return sum;
     return NULL;
13 }
   int main(int argc, char *argv[]){
15   double (*operador)(double);
     operador=oper("sq");
17   printf("Cuadrado de 5=%lf\n",operador(5.0));
     operador=oper("sum");
19   printf("Suma de 5=%lf\n",operador(5.0));
   return 0;
21 }
```

Simplificacion usando typedef


```c
double sq(double x){
  return x*x;
}
double sum(double x){
  return x+x;
}

typedef double (*FUNPTR)(double);
FUNPTR swap(FUNPTR fun){
  if (fun==sum) return sq;
  if (fun==sq)  return sum;
  return NULL;
}
int main(int argc, char *argv[]){
    double (*operador)(double);
    operador=swap(sum);
    printf("Cuadrado de 5=%lf\n",operador(5.0));
    operador=swap(sq);
    printf("Suma de 5=%lf\n",operador(5.0));
return 0;
}
```

Tarea 6

prog6.2 Escriba un programa que:

1. Defina un tipo de apuntador a función, la función (a la que apunta) recibe un vector de doubles y un entero (que es el tamaño del vector de dobles), y devuelve un doble que es el resultado de una operación sobre el vector.
2. Defina una estructura que en sus elementos tiene un arreglo estático de dos dobles, una variable doble **res**, y un apuntador a una función del tipo anterior.
3. Defina un arreglo de apuntadores a función utilizando el tipo definido en anteriormente.
4. Defina una funcion **double suma(double *x, int n)** que suma los elementos de x, una función **promedio**, que saca el promedio de los elementos de x, y una función **resta** que resta el segundo elemento al primero en x, una funcion **seno** que calcula $x_0 \sin(x_1)$. Todas las funciones anteriores tienen argumentos iguales a los de **suma** y devuelven lo mismo.
5. Defina un vector dinámico de tamaño **n** de estructuras (donde **n** es parametro de usuario), a estructura asignele de forma aleatoria un número real entre 0 y 1 y una función, aplique la función a estos números y escriba el resultado en **res** de la misma estructura. Imprima los resultados.
6. Devuelva la memoria y verifique que no tiene accesos indebidos a memoria.

Nota:Utilice el vector de apuntadores a funciones para asignar la función aleatoria: llene el vector de apuntadores con las direcciones de las funciones que programo, elija un indice de ese vector al azar y copie el apuntador a la estructura.

Librerias estándar de C

Librerias estándar

C fue creado en 1972 por Dennis M. Ritchie en los laboratorios Bell para desarrollar el sistema operativo Unix.
Las librerias estándar son un conjunto de funciones, definiciones de tipos de datos y macros. La interface para programar aplicaciones (Application Program Interface API) de las librerias estándar son un conjunto de archivos cabecera.

```
<assert.h>  <ctype.h>  <errno.h>  <float.h>
<limits.h>  <locale.h>  <math.h> <setjmp.h>
<signal.h> <stdarg.h> <stddef.h> <stdio.h>
<stdlib.h> <string.h> <time.h>
```

time.h, variables

http://www.tutorialspoint.com/c_standard_library/time_h.htm

Variable	Description
size_t	En general es un entero sin signo y puede almacenar el tamño del objeto mas grande posible que puede ser reservado.
clock_t	Debe de poder guardar el tiempo de reloj (del procesador) desde cierta referencia.
time_t	Guarda el tiempo del calendario. Usualmente es un entero que contiene el número de segundos transcurridos desde el 1ro de junio de 1970.
struct tm	Es una estructura que almacena el tiempo y fecha

```
1  struct tm {
     int tm_sec;  /* seconds,  range 0 to 59          */
3    int tm_min;  /* minutes, range 0 to 59           */
     int tm_hour; /* hours, range 0 to 23             */
5    int tm_mday; /* day of the month, range 1 to 31  */
     int tm_mon;  /* month, range 0 to 11             */
7    int tm_year; /* The number of years since 1900   */
     int tm_wday; /* day of the week, range 0 to 6    */
9    int tm_yday; /* day in the year, range 0 to 365  */
     int tm_isdst;/* daylight saving time             */
11 };
```

time.h, constantes (macros)

- NULL El valor de un apuntador constante que no apunta a un lugar valido de la RAM, usualmente (void *)0.
- CLOCKS_PER_SEC Valor del número de instrucciones procesadas por segundo o ciclos de reloj (mas o menos, en realidad es la frecuencia de oscilación del cristal que sirve para procesar las instrucciones).

time.h, funciones

- **char *asctime(const struct tm *timeptr)** Devuelve un apuntador a un string con el tiempo actual.
- **clock_t clock(void)** Regresa el tiempo de reloj del CPU.
- **time_t time(time_t *timer)** Regresa el tiempo de calendario.
- **struct tm *localtime(const time_t *timer)** Regresa el tiempo de calendario en la estructura tm.
- **char *asctime(const struct tm *timeptr)** Regresa el apuntador a un string con la fecha de calendario.
- **char *ctime(const time_t *timer)** Regresa un apuntador a un string con el tiempo local (puede ser igual al de arriba depende de los locales).
- **size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)** Regresa un apuntador al string en el formato especificado.
- **struct tm *gmtime(const time_t *timer)** Tiempo expresado en UTC.
- **time_t mktime(struct tm *timeptr)** cambia el contenido de una estuctura tm a un time_t.

time.h, ejemplos


```c
#include <stdio.h>
#include <time.h>
int main(int argc, char *argv[]){
 time_t t1,t2;
 clock_t cini,cend;
 struct tm *ti;
 char *date1,*date2, strtime[128];
 cini=clock(); t1=time(&t2); t2=time(NULL);
 printf("Seg. desde 1/ene/1970=%ld o %ld\n",t1, t2);
 printf("Dif. entre t2-t1=%lf\n",difftime(t2,t1));
 ti=localtime(&t2); date1=asctime(ti);date2=ctime(&t2);
 printf("Segs=%d mins=%d hrs=%d dia=%d...\n",ti->tm_sec
    ,ti->tm_min,ti->tm_hour,ti->tm_mday);
 strftime(strtime, 128, "%X", ti);
 printf("%s %s %s\n",date1,date2,strtime);
 //Solo para que se tarde algo
 for(int k=0; k<10000; k++)
   for(int m=0; m<10000; m++);
 cend=clock();
 printf("Milisegundos de ejecucion=%lf\n",1000.0*(
    double)(cend-cini)/(double)CLOCKS_PER_SEC);
return 0;
}
```

time.h, ejemplos


```
Seg. desde 1/ene/1970=1411406203 o 1411406203
Dif. entre t2-t1=0.000000
Segs=43 mins=16 hrs=12 dia=22...
Mon Sep 22 12:16:43 2014
 Mon Sep 22 12:16:43 2014
 12:16:43
Milisegundos de ejecucion=210.000000
```

Algoritmos de ordenamiento

Insertion sort

Data: n, A
Result: A ordenado

```
1 for j=1 to n-1 do
2     key=A[j];
3     i=j-1;
4     while i ≥ 0 & comp(A[i],key) do
5         A[i+1]=A[i];
6         i=i-1;
7     A[i+1]=key;
```

Algorithm 1: Insertion Sort

- Los elementos desde A[0..j-1] están ordenados en cada ciclo. A esta propiedad se le llama el **invariante del ciclo**.
- Los datos se cambian de orden si comp(A[i],key) es diferente de 0.

Invariante de ciclo

Un invariante de ciclo debe de tener 3 propiedades:

1. **Inicialización.** Antes de entrar al ciclo el invariante de ciclo debe ser verdadero. Ej. el arreglo A[0,...,j-1] en la inicialización del ciclo **for** debe de estar ordenado.

Invariante de ciclo

Un invariante de ciclo debe de tener 3 propiedades:

1. **Inicialización.** Antes de entrar al ciclo el invariante de ciclo debe ser verdadero. Ej. el arreglo A[0,...,j-1] en la inicialización del ciclo **for** debe de estar ordenado.
2. **Mantenimiento.** En cada iteración del ciclo se mantiene el invariante del ciclo.

Invariante de ciclo

Un invariante de ciclo debe de tener 3 propiedades:

1. **Inicialización.** Antes de entrar al ciclo el invariante de ciclo debe ser verdadero. Ej. el arreglo A[0,...,j-1] en la inicialización del ciclo **for** debe de estar ordenado.
2. **Mantenimiento.** En cada iteración del ciclo se mantiene el invariante del ciclo.
3. **Terminación** Cuando el ciclo termina la propiedad se mantiene. Ej. el ciclo termina para j=n, entonces el arreglo debe de estar ordenado para A[0,...,n-1].

Tiempo de cómputo. Insertion sort

	Algoritmo	Costo	Veces
	Data: n, A		
	Result: A ordenado		
1	**for** *j=1 to n-1* **do**	c_1	n
2	key=A[j];	c_2	$n-1$
3	i=j-1;	c_3	$n-1$
4	**while** $i \geq 0$ & *comp(A[i],key)* **do**	c_4	$\sum_{j=2}^{n} t_j$
5	A[i+1]=A[i];	c_5	$\sum_{j=2}^{n}(t_j-1)$
6	i=i-1;	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	A[i+1]=key;	c_7	$n-1$

Algorithm 2: Insertion Sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$$

Mejor caso: $t_j = 1$, peor caso $t_j = j$

$\sum_{j=2}^{n} j = n(n+1)/2 - 1 \qquad \sum_{j=2}^{n}(j-1) = n(n-1)/2$

Definición informal de orden de crecimiento (orden)

En el caso anterior se puede escribir el peor desempeño del algoritmo como:

$$T(n) = an^2 + bn + c$$

Podemos tomar el termino que crece más en función del número de datos de entrada que es an^2.
Y podemos decir que siempre existe una función de n^2 que acota el tiempo de cómputo por arriba, a esta función se le representa como: $O(n^2)$. Y normalmente se le denomina el orden del algoritmo.

Tarea 6

prog6.3 Escriba un programa con función de Insertion Sort para ordenar cadenas de caracteres, las cuales seran dadas como un arreglo de apuntadores a cadenas (una matriz de chars):

1. A la función que ordena se le pasara como argumento la funcion que compara dos cadenas.
2. Habrá 2 funciones que comparan: a)Una cadena es mayor que otra si es más larga, b) una cadena es mayor que otra si la suma de sus elementos (tomados como enteros) es mayor que la de otra. **comp(A,B)** regresa 1 si A es menor o igual que B y 0 si no lo es. Hacer las dos versiones enviando el apuntador a **comp**.
3. Escriba la funcion copiando una cadena en otra e intercambiando direcciones de memoria de la matriz que contiene las cadenas.

Merge Sort

Data: A,p,r
Result: A ordenado

```
1  if p < r then
2      q = ⌊(p + r)/2⌋
       Merge-Sort(A,p,q);
3      Merge-Sort(A,q+1,r);
4      Merge(A,p,q,r);
```

Algorithm 3: Merge-Sort

Data: A,p,q,r
Result: A con $r - p + 1$ elementos ordenados

```
1  n_1 = q − p + 1;
2  n_2 = r − q;
3  L = [A[p..n_1 + p − 1], ∞];
4  R = [A[q + 1..n_2 + q], ∞];
5  i = 0;
6  j = 0;
7  for k = p to r do
8      if comp(L[i], R[j]) then
9          A[k]=L[i];
10         i=i+1;
11     else
12         A[k]=R[j];
13         j=j+1;
```

Algorithm 4: Merge

Tarea 6

prog6.4 Escriba un programa con las mismas características que el 6.3 pero utilizando el algoritmo de **Merge-sort**.