Indice

1 Introducción 4

1.1 Bibliografía 4

1.2 Variables , constantes y operadores 4

1.2.1 Nombres de identificadores 4

1.2.2 Tipo de Datos 4

1.2.3 Modificadores 5

1.2.4 Constantes 5

1.2.5 Operadores 5

1.3 Forma general de un programa 6

1.4 Vectores 7

1.4.1 Carga secuencial y aleatoria de un vector 8

1.4.2 Búsquedas y ordenamiento sobre vectores 13

2 Cadenas de Caracteres 17

2.1 Inicialización de cadenas de caracteres 17

2.2 Funciones usadas para trabajar con cadenas de caracteres 17

2.2.1 SCANF 18

2.2.2 GETS (GET String) 18

2.2.3 STRCMP (STRing CoMParison) 19

2.2.4 STRCPY (STRing CoPY) 20

2.2.5 STRLEN (STRing LENght) 21

2.3 Matrices con cadenas de caracteres 22

2.4 Comprobación de la longitud de un texto 23

3 Estructuras 26

3.1 Definición 26

3.2 Declaración de una estructura 26

3.3 Acceso a los campos – Operador punto 28

3.4 Vector de estructuras 29

3.5 Estructuras anidadas 29

3.6 Propiedades 31

3.6.1 Asignación 31

3.6.2 Comparación 32

4 Punteros 34

4.1 Definición 34

4.2 Declaración de un puntero 35

4.3 Operadores 36

4.4 Asignación de punteros 37

4.5 Punteros lejanos y cercanos 39

4.6 Comparación de punteros 40

4.7 Aritmética de punteros 41

4.7.1 Suma y resta de enteros 41

4.8 Vectores y punteros 42

4.9 Punteros a cadenas de caracteres 47

4.10 Punteros a estructura 48

4.10.1 Acceso a los campos – Operador flecha 49

4.10.2 Vector de punteros 49

4.11 Asignación Dinámica de memoria 52

5 Funciones 54

5.1 Declaración de una función 54

5.2 Variables locales y globales 57

5.3 Diagrama de flujo de un programa que usa funciones 58

5.4 Pasaje de parámetros por valor y por referencia 59

5.4.1 Pasaje por valor 59

5.4.2 Pasaje por referencia 60

5.5 Diseño de una función 63

5.6 Recursividad 63

5.7 Librería de funciones propias 64

5.8 Argumentos de main 66

6 Archivos 68

6.1 Concepto de archivos 68

6.1.1 Archivos de texto 68

6.1.2 Archivos binarios 69

6.2 Trabajo sobre archivos 69

6.2.1 La estructura FILE 70

6.2.2 Apertura y cierre de un archivo 71

6.2.3 Escritura de un archivo 75

6.2.4 Lectura de un archivo 76

6.2.5 Búsqueda y modificación 78

7 Gestión dinámica de memoria 84

7.1 Pilas 84

7.1.1 Insertar elementos en una pila 84

7.1.2 Esquema gráfico de inserción de elementos de una pila 86

7.1.3 Desapilar elementos de una pila 87

7.1.4 Esquema gráfico de la eliminación de los elementos de una pila 90

7.1.5 Esquema gráfico para recorrer los elementos de una pila 90

7.1.6 Funciones que permiten extraer, listar , eliminar y destruir elementos de una pila 90

7.2 Colas 93

7.3 Listas simplemente encadenadas 93

7.3.1 Primer elemento de la lista - Lista Vacia 94

7.3.2 Insertar al final de la lista 94

7.3.3 Insertar al comienzo de la lista 94

7.3.4 Insertar en una posición intermedia 95

7.3.5 Función para insertar un elemento en una lista 95

7.3.6 Función para eliminar un elemento de la ista 97

7.3.7 Función para mostrar todos los elementos de la lista 99

7.3.8 Función para eliminar la lista completa 100

7.3.9 Programa completo 100

8 Ejercicio Tipo 104

8.1 Desarrollo del ejercicio tipo Usando índices 104

8.2 Desarrollo del ejercicio tipo Usando punteros 106

8.3 Desarrollo del ejercicio tipo Usando funciones 109

8.4 Desarrollo del ejercicio usando estructuras 113

8.5 Desarrollo del ejercicio usando archivos 115

9 Anexo 1 119

9.1 Función de validación de cadenas de caracteres 119

Programación II

# Introducción

## Bibliografía

. Manual de Borland C++ [Chris H. Pappas, William H. Murray III] Editorial Mc. Graw-Hill

. Turbo C/C++ 3.1 Manual de Referencia [Herbert Schild] Editorial Mc. Graw-Hill

. El lenguaje de programación C [Brian W. Kernighan, Dennis M. Ritchie]

. Apunte de Programación en lenguaje C [ Lic. Marcelo Samia] UTN FRBA Cod: R2CT4

. Programación en C Parte I [Juan C. Maquieira, Gustavo R. Viard] Edición Alfafì

## Variables , constantes y operadores

### Nombres de identificadores

Se conocen como identificadores a los nombres de variables , funciones y etiquetas definidos por el usuario.

La longitud de un identificador en C puede tener entre 1 y 32 caracteres. El primer caracter debe ser una letra o un guión bajo ( \_ ) y los siguientes caracteres pueden ser letras , números o guion bajo.

Se debe tener en cuenta que los nombres de identificadores tienen en cuenta las mayúsculas y minúsculas. De esta forma si tenemos 3 variables como min , Min y MIN se las trata como variables distintas.

Por otro lado es importante destacar 2 puntos importantes para hacer mas fácil la interpretación de un programa.

Los nombres de las variables se escriben en minúsculas o con la primer letra en mayúsculas.

Los nombres de las constantes definidas por medio de la directiva del compilador #define se escriben en mayúsculas.

De esta forma cuando alguien lee el código fuente de un programa detecta fácilmente cuales son las constantes y cuales las variables.

### Tipo de Datos

En C existen 5 tipos de datos definidos. En la tabla siguiente

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo de dato** | **Bits** | **Rango (con signo)** | **Rango (sin signo)** |
| char | 8 | -128 a +127 | 0 a 255 |
| int | 16 | -32768 a +32767 | 0 a 65535 |
| float | 32 | -3.2x10+38 a +3.2x10+38 |  |
| double | 64 | -1.7x10+308 a +1.7x10+308 |  |
| void | 0 | sin valor | sin valor |

Una variable del tipo char se utiliza normalmente para almacenar valores definidos en la tabla ASCII o para almacenar cualquier número que se encuentre en el rango mostrado en la tabla. Por ejemplo para la edad de una persona puede tranquilamente utilizarse una variable del tipo char.

Las variables de tipo int se utilizan para guardar números enteros y los float y double se utilizan para números reales.

Ahora pensemos que uno delos datos a ingresar en nuestro programa sea el número de DNI de una persona , que tipo de dato utilizaria para la variable?

Un número de DNI podría ser 18.458.321 podemos ver que es un número entero por lo tanto en un primer momento podemos pensar en guardarlo en una variable del tipo int , pero el valor máximo de un entero sin signo es 65535 con lo cuál se ve claramente que el DNI no se puede guardar en una variable tipo int.

La edad de una persona siempre es positiva y normalmente menor a 150 años , entonces se elije un dato del tipo int donde solo se utiliza menos del 0.5% del rango de un entero.

Para solucionar este tipo de inconvenientes existen los modificadores del tipo de dato

### Modificadores

Existen 4 modificadores de tipo de dato. Los mismos se detallan abajo

**Signed**

Unsigned Aplicable a **char** e **int**

**Short**

Long Aplicable a char, int

Por defecto , es decir , mientras no se declare explicitamente las variables son signed y short.

Es decir que cuando se define una variable del tipo int , la misma será signed y short ( entero corto con signo ) y el rango será desde -32768 a +32767.

El modificador unsigned se utiliza cuando los valores que va a adoptar la variable no llevan signo. Por ejemplo una variable del tipo unsigned char tendrá un rango desde 0 a 255 , totalmente apto para guardar una edad.

El modificador long (normalmente utilizado con el tipo int) extiende el rango a 32 bits. Un tipo de dato long int es apto para guardar el número de DNI.

### Constantes

Las constantes se refieren a valores fijos que no pueden ser cambiados durante el transcurso del programa.

Normalmente en un programa , ya sea por practicidad o por una cuestión de claridad del código fuente , se suelen definir constantes por medio de la directiva **define.**

De acuerdo al tipo de variable los valores de las constantes se expresan de diferentes formas

Para el tipo char la constante se escribe con apostrofes por ejemplo ‘a’ , ‘\n’ ,’3’

Para el tipo int sera 123 , 5 , -2000

Para el tipo float y double será 125.3 , 3.14 , -0.05

Para las cadenas de caracteres el texto comienza y termina con comillas ( “ ) por ejemplo “hola” , “esto es una prueba”

### Operadores

Un operador es un símbolo que le dice al compilador que realice determinadas operaciones aritméticas o lógicas.

En C existen 3 tipos de operadores que son

1. Operadores aritméticos

|  |  |
| --- | --- |
| **Operador** | **Significado** |
| + | Suma |
| - | Resta |
| \* | Multiplicación |
| / | División |
| % | Resto de la división |
| ++ | Incremento |
| -- | Decremento |

1. Operadores relacionales y lógicos

Relacionales

|  |  |
| --- | --- |
| **Operador** | **Significado** |
| > | Mayor |
| >= | Mayor o igual |
| < | Menor |
| <= | Menor o igual |
| == | Igual |
| ¡= | Distinto |

Lógicos

|  |  |
| --- | --- |
| **Operador** | **Significado** |
| && | AND |
| | | | OR |
| ¡ | NOT |

1. Operadores a nivel de bit

|  |  |
| --- | --- |
| **Operador** | **Significado** |
| & | AND |
| | | OR |
| ^ | OR Exclusive |
| ~ | Complemento a 1 |
| >> | Desplazamiento a la derecha |
| << | Desplazamiento a la izquierda |

## Forma general de un programa

En un programa se pueden encontrar secciones determinadas y un órden para las mismas.

La forma que se detalla en la tabla es la mas general , esto no quiere decir que todos los programas deban contener cada una de las secciones detalladas , pero si las contiene ,debería estar en el lugar que corresponde.

Por ejemplo , la línea ***#include <stdio.h>*** usted la escribió en todos los programas por el simple hecho de que en algún momento utilizó la función ***printf*** o ***scanf.***

Tanto ***#include*** como ***#define*** son directivas del compilador que no hacen al lenguaje en si , pero que son necesarias y deben estar ubicadas en el lugar adecuado.

Si bien las directivas ***include*** pueden no estar al principio del programa y los ***define*** no hace falta que se encuentren inmediatamente después de los ***include*** , es conveniente , por varias razones , que se mantenga un órden en la escritura de todos los programas. El órden propuesto para cada una de las secciones del programa es el que se detalla a continuación.

|  |  |
| --- | --- |
| **- Archivos de cabecera** | **#include <stdio.h>**  **#include <conio.h>** |
| - Declararacion de constantes y macros | #define MAXNOM 20  #define MAXDIR 30  #define MAXPERS 100 |
| - Declaración de estructuras | struct gente{  char nombre[MAXNOM];  char dir[MAXDIR];  int edad;  }; |
| - Declaración de prototipos de funciónes | void CargaGente( struct gente \*);  int ValidaEdad(void); |
| - Declaración de variables globales | int flag; |
| - Desarrollo de la función principal  void main(void)  {  variables locales de main  - - - - - - -  - - - - - -  } | void main(void)  {  struct gente agenda[MAXPERS];  int i, cant;  - - - - - - - - - -  } |
| * Desarrollo de otras funciones |  |
| int f1 (char x)  {  Variables locales de f1  } | void CargaGente(char \* p)  {  int i , cant;  } |
| void f2 ( void)  {  Variables locales de f2  } | int ValidaEdad(void)  {  int edad;  } |

## Vectores

Un vector es un conjunto de elementos del mismo tipo (enteros, flotantes, caracteres) que se agrupan bajo un mismo nombre y se diferencian entre sí por un índice.

La forma genérica de declararlo en un programa es la siguiente

***Tipo* nombre[cantidad]**

Donde tipo es el tipo de dato, por ejemplo Int, float , char, etc.

Nombre es el nombre del vector y cantidad es la cantidad de elementos que contiene el vector.

Un ejemplo de esto puede ser

int v[5]; defino un vector que se llama v de 5 enteros

float vec[10]; defino un vector que se llama vec de 10 flotantes

char v1[5]; defino un vector llamado v1 de 20 caracteres

La representación gráfica del vector **v** puede ser la siguiente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12  Indice | 23 | 9 | 37 | 31 |
| 0 | 1 | 2 | 3 | 4 |

Como se puede ver el índice comienza siempre en *cero* y va hasta la *cantidad de elementos menos uno.*

Para acceder a un elemento cualquiera de vector, por ejemplo al tercer elemento se escribe

v[2]

Donde v es el nombre del vector y entre corchetes se le pone el índice.

De acuerdo a esto decimos que el vector tiene 5 enteros, el nombre del vector es **v** y cada uno de sus elementos serán

v[0] tiene el valor 12 y es el primer elemento del vector

v[1] tiene el valor 23 y es el segundo elemento del vector

v[2] tiene el valor 9 y es el tercer elemento del vector

v[3] tiene el valor 37 y es el cuarto elemento del vector

v[4] tiene el valor 31 y es el quinto elemento del vector

Observe que entre el número de elemento y el índice existe una diferencia de 1

**Note que v[5]no existe como elemento del vector.**

### Carga secuencial y aleatoria de un vector

Para llenar un vector con valores (cargar), existen 2 formas que son la secuencial y la aleatoria.

En la carga secuencial se carga el vector ordenadamente, es decir el primero después el segundo, el tercero y así hasta llegar al último.

En la carga aleatoria no existe un orden establecido, por lo tanto se debe informar el número a cargar y en que posición del vector se carga.

#### Carga secuencial

Como se comentó anteriormente la carga secuencial es ordenada y de acuerdo a esto deberemos llenar el vector respetando el siguiente orden

v[0], v[1], v[2], v[3], v[4]

Como se ve, el que cambia en forma creciente es el índice del vector que es el que me permite hacer la carga en forma secuencial.

Para llevar esto a código de programa debemos pensar en una estructura de control que me permita realizar un número prefijado de repeticiones y que me incremente una variable en una unidad. Dicha estructura de control es el ciclo **FOR.**

Un ejemplo de programa que me permite cargar un vector en forma secuencial es el siguiente

#include <stdio.h>

void main(void)

{

int v[5],i;

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

{

printf("Ingrese valor a cargar en el vector");

scanf("%d",&v[i]);

}

}

En el programa la variable i del ciclo for va a tomar valores desde 0 a 4 ordenadamente. Cuando comienza el for la variable i vale 0 (i=0), se cumple con la condición de que i<5 y entonces entramos al for. Se ejecuta el printf y luego el scanf lee desde el teclado y carga el valor en v[i] (v[0]). Se vuelve al for incrementando i en 1 (i++) por lo tanto el valor de i ahora es 1 y se cargará v[1], siguiendo i=2 y v[2], i=3 y v[3] para finalmente i=4 y v[4].

De esta forma se carga el vector en forma secuencial.

Note que de esta forma se cargan todos los elementos del vector.

***Ejemplo:*** De los 100 empleados de una fábrica se registra número de legajo (coincidente con el índice), edad y salario. Se pide ingresar los datos consecutivamente y calcular el sueldo promedio.

Comenzamos el problema analizando donde se guardarán los datos.

Sabemos que se ingresa legajo, edad y sueldo de 100 empleados y que el legajo coincide con el índice , por lo tanto debemos definir 2 vectores paralelos, uno será de enteros (para la edad) y el otro será de flotantes (para el sueldo). El legajo va de uno a 100 y nosotros deberemos establecer la relación que hay con el índice, que sabemos que va de cero a 99 en el vector. Aquí se nota la diferencia de uno entre el legajo e índice.

Un esquema de los vectores puede ser el siguiente

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Edad  Sueldo  Indice  Legajo |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 | 5 |

Sabiendo que se cargan los datos en forma consecutiva, sabemos que la carga deberá ser secuencial por legajo, por lo tanto usamos la estructura del for que se escribió arriba

for (i=0;i<100;i++)

{

printf("Ingrese edad para el legajo %d",i+1);

scanf("%d",&edad[i]);

printf("Ingrese sueldo para el legajo %d",i+1);

scanf("%f",&sueldo[i]);

}

Como se ve en el código, el legajo no se ingresa por que lleva relación directa con el índice que en este caso es la variable i.

Una vez cargados los 100 datos debemos calcular el sueldo promedio. Para ello se deben sumar todos los sueldos que estan en el vector y luego dividirlos por la cantidad de elementos que en este caso es 100.

Para realizar la suma utilizamos un for ya que conocemos la cantidad de iteraciones. Finalmente se muestra en pantalla el sueldo promedio.

El programa completo será el siguiente.

#include <stdio.h>

#define MAX 100

void main(void)

{

int edad[MAX],i;

float sueldo[MAX],suma=0,prom,aux;

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

{

printf("Ingrese la edad del legajo %d: ",i+1);

scanf("%d",&edad[i]);

printf("Ingrese el sueldo del legajo %d: ",i+1);

scanf("%f",&aux);

Esta modificación se debe hacer por un problema que tiene el scanf con vectores de flotantes.

sueldo[i]=aux;

}

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

En la variable suma que esta inicializada en CERO se acumulan los sueldos

suma=suma+sueldo[i];

prom=suma/MAX;

printf("El sueldo promedio es %.2f",prom);

}

En la carga cuando i=0 se esta pidiendo el ingreso de la edad del legajo 1, pero se guarda en edad[0]. Lo mismo ocurre con el sueldo y con el resto de los ingresos.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| I | Edad  ingresada | Sueldo  ingresado |  | Vector  Edad |  | Vector  Sueldo | Legajo |
| 0 | 21 | 500 |  | 21 |  | 500.00 | 1 |
| 1 | 18 | 470.20 |  | 18 |  | 470.20 | 2 |
| 2 | 30 | 890.75 |  | 30 |  | 890.75 | 3 |
| 3 | 29 | 756.20 |  | 29 |  | 756.20 | 4 |

Observe que el promedio se calcula recién cuando termina el *FOR*, es decir cuando tengo la suma completa. Si se pone dentro del *FOR*, en lugar de calcular el promedio 1 sola vez se calcularía 100 veces.

#### Carga aleatoria

De acuerdo a lo comentado con anterioridad para la carga aleatoria debo ingresar 2 valores que son la posición donde quiero guardar el número y luego el número que quiero cargar.

Puede ocurrir que en algún caso no se carguen todos los elementos del vector o que se pretenda cargar algún valor en un lugar ya cargado. Para analizar esto comenzaremos por el caso más simple y dejando que el usuario decida cuando terminar de cargar elementos en el vector. De acuerdo a esto como no se sabe cuantos elementos se van a cargar ni en que orden , debemos primero inicializar todos los elementos del vector en un valor conocido que normalmente es cero.

Seguidamente se debe usar un ciclo de control en el cuál la cantidad de iteraciones que se realicen quede fijada por alguna condición que acepte el usuario, en este caso el ciclo que más se adecua es el **while** o **do while**.

Un ejemplo de programa puede ser el siguiente

#include <stdio.h>

void main(void)

{

int v[5],pos,i;

char seguir;

Inicializo vector

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

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

v[i]=0;

do

{

printf("Ingrese posici¢n");

scanf("%d",&pos);

printf("Ingrese valor a cargar en el vector");

scanf("%d",&v[pos]);

printf("Desea ingresar otro dato S/N?");

scanf("%c",&seguir);

}while(seguir=='s');

}

Una vez inicializado el vector ingreso al ciclo **do while** y supongamos que se ingresa la posición 2 y como valor a cargar 15, con lo cual tendremos

Pos=2 y v[pos]=15 v[pos] será v[2] y el vector queda

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 15 | 0 | 0 |

Contestamos que si a ingresar otro dato y ahora pos=4 y el valor a cargar 43 , quedando el vector

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 15 | 0 | 43 |

Observe como se cargan los datos de acuerdo a la posición que el usuario elige y note que no es en forma ordenada. Solo será ordenada si se ingresan las posiciones en forma consecutiva.

Cuando se conteste que no se quieren ingresar mas datos, se finaliza la carga.

***Ejemplo:*** De los 100 empleados de una fábrica se registra número de legajo (coincidente con el índice), edad y salario. Se pide ingresar los datos y calcular el sueldo promedio.

El análisis del problema se plantea exactamente igual al caso de carga secuencial. La única diferencia es que ahora se debe ingresar el legajo quien va a ser el que me indique la posición en donde se va a cargar el sueldo y la edad.

#include <stdio.h>

#define MAX 100

void main(void)

{

int edad[MAX],i,leg;

float sueldo[MAX],suma=0,prom,aux;

char seguir;

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

{

edad[i]=0;

sueldo[i]=0;

}

do

{

printf("Ingrese Legajo");

scanf("%d",&leg);

printf("Ingrese Edad");

scanf("%d",&edad[leg-1]);

printf("Ingrese el sueldo");

scanf("%f",&aux);

sueldo[i]=aux;

printf("Desea ingresar otro dato S/N?");

scanf("%c",&seguir);

}while(seguir=='s');

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

En la variable suma que esta inicializada en CERO se acumulan los sueldos

suma=suma+sueldo[i];

prom=suma/MAX;

printf("El sueldo promedio es %.2f",prom);

}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Legajo  ingresado | Edad  ingresada | Sueldo  ingresado |  | Vector  Edad |  | Vector  Sueldo | Indice |
| 3 | 21 | 500 |  | 30 |  | 890.75 | 0 |
| 4 | 18 | 470.20 |  | 29 |  | 756.20 | 1 |
| 1 | 30 | 890.75 |  | 21 |  | 500.00 | 2 |
| 2 | 29 | 756.20 |  | 18 |  | 470.20 | 3 |

Recuerde que la relación entre el índice y el legajo es legajo=indice + 1.

El primer legajo que se cargó fue el 3 que corresponde al índice 2, por lo tanto los primeros datos que se cargaron fueron el 21 y el 500, luego el 18 y 470.20 y asi sucesivamente. La flecha al costado izquierdo de la tabla indica el órden en el que se cargaron los datos.

Observe que el legajo se carga pero no se guarda en el vector, sirve para posicionarse dentro de uno de los elementos del vector.

El resto del programa sigue igual que en el caso de carga secuencial. Solo es secuencial o aleatoria la carga, el resto de las operaciones con vectores normalmente se hace en forma secuencial usando un *for*.

### Búsquedas y ordenamiento sobre vectores

Vamos a desarrollar un programa de ejemplo con distintas tareas que se necesitan realizar sobre vectores.

Tenga en cuenta que en el programa figuran bien separados los módulos de programa que realizan cada tarea. Observe que dichos módulos realizan una tarea concisa y clara , por lo tanto cuando usted tenga que escribir un programa no debe inventar códigos nuevos, solo tiene que poner los módulos que necesite.

#include <stdio.h>

#define MAX 5

void main(void)

{

int vec[MAX],i,mayor,menor,num,flag;

**Carga el vector**

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

{

printf("Ingrese nro");

scanf("%d",&vec[i]);

}

**buscar mayor**

mayor=vec[0];

Aquí vienen todas aquellas asignaciones relacionadas con la busqueda, por ejemplo buscar el legajo de la persona de mayor edad. Se busca la edad mayor y dentro del bloque se agrega el legajo.

###### Mayor\_leg=legajo[i]

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

if(vec[i]>mayor)

{

mayor=vec[i];

/\* otras asignaciones \*/

}

**buscar menor**

menor=vec[0];

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

Aquí vienen todas aquellas asignaciones relacionadas con la búsqueda, por ejemplo buscar el legajo de la persona de menor edad. Se busca la edad menor y dentro del bloque se agrega el legajo.

###### Menor\_leg=legajo[i]

if(vec[i]<menor)

{

menor=vec[i];

/\* otras asignaciones \*/

}

**buscar un numero dentro del vector**

flag=0;

printf("Ingrese numero a buscar");

scanf("%d",&num);

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

{

if(num==vec[i])

{

/\* asignaciones y modificaciones \*/

printf("Se encontro el numero");

flag=1;

}

}

if(flag==0)

printf("El numero no se encontro");

**Ordenar el vector vector**

for(i=0;i<FIL-1;i++)

for(j=i+1;j<FIL;j++)

if(vec[i]>vec[j])

{

aux=vec[i];

vec[i]=vec[j];

vec[j]=aux;

}

}

**Búsqueda del mayor dentro de un vector**

Para encontrar el mayor elemento dentro de un vector lo primero que hacemos es tomar uno cualquiera de los elementos del mismo y suponer que es el mayor. Luego comparando cada elemento del vector contra el valor de la variable ***mayor*** , que es actualizado mientras se recorre el vector , se llega al final del vector con el mayor elemento guardado en la variable ***mayor***

Considere un vector con los siguientes datos cargados donde se pretende encontrar el elemento mayor y la posicion en la cual esta ubicado.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 1 | | 9 | 8 |
| [0] | | [1] | [2] | [3] |

mayor=vec[0];

posicion=0;

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

if(vec[i]>mayor)

{

mayor=vec[i];

posicion=i; /\* otras asignaciones \*/

}

en la primer línea ***mayor=vec[0];*** estamos asignando a la variable mayor el primer elemento del vector, es decir que estamos inicializando la variable mayor con un valor de 5 seguidamente seteamos la posicion inicial para dicha variable.

El ciclo ***for*** es el que me va a permitir recorrer todo el vector realizando MAX iteraciones. La decisión se hace con el ***if***  en donde tendremos

I=0 mayor=5 5>5 Falso

I=1 mayor=5 1>5 Falso

I=2 mayor=5 9>5 Verdadero -> mayor=9 -> posicion=2;

I=3 mayor=9 8>9 Falso

Terminado el for nos queda la variable mayor con el valor 9, que es el mayor de los números dentro del vector.

**Búsqueda del menor dentro de un vector**

El razonamiento es el mismo que para la búsqueda del mayor. Solo cambia el signo de la desigualdad ***if (vec[i]< menor)***

**Búsqueda de un número dentro de un vector**

Para buscar un número dentro de un vector, lo debemos recorrer y preguntar si el elemento del vector es igual al número ingresado.

De acuerdo a esto surge que para recorrer el vector debo colocar un ***for*** y dentro del mismo un ***if*** para preguntar por la igualdad.

El código mas simple para realizar la búsqueda es el siguiente

printf("Ingrese numero a buscar");

scanf("%d",&num);

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

if(num==vec[i])

{

printf("Se encontro el numero");

break;

}

Las 2 primeras líneas de código solicitan el ingreso del número a buscar. Analicemos el funcionamiento con un ejemplo del vector cargado

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 1 | | 9 | 8 |
| [0] | | [1] | [2] | [3] |

Ahora supongamos que el número que se quiere buscar es el 9, con lo cual la variable ***num*** tendrá el valor 9. Si seguimos el código tendremos

I=0 vec[i]=5 if(num==vec[i]) Falso

I=1 vec[i]=1 if(num==vec[i]) Falso

I=2 vec[i]=9 if(num==vec[i]) Verdadero

Al entrar al ***if*** se muestra el mensaje y el ***break*** hace que se termine el ***for*** .

Observe que cuando el número se encuentra en el vector se termina la búsqueda.

Nos queda analizar una pequeña modificación al algoritmo para que me permita informar si el número no existe en el vector.

El problema a resolver es como se da cuenta el programa que el número no esta en el vector. La respuesta es EL PROGRAMA NO TIENE FORMA DE SABER QUE EL NÚMERO NO ESTA EN EL VECTOR, pero si sabe cuando lo encuentra.

De acuerdo a esto puedo decir que si el programa nunca entro al if (despues de recorrer todo el vector) significa que no lo encontro y entonces la solución es colocar una bandera dentro del if. El código es el siguiente

Supongo que el número no lo encuentro

flag=0;

printf("Ingrese numero a buscar");

scanf("%d",&num);

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

{

if(num==vec[i])

{

Si el programa entro al if el número se encontró y pongo la bandera en 1.

printf("Se encontro el numero");

flag=1;

}

}

if(flag==0)

printf("El numero no se encontro");

Estas son las formas básicas para realizar búsquedas sobre vectores, mas adelante se analizará el tema tratándolo con cadenas de caracteres, punteros y estructuras.

**Ordenamiento de un vector**

Una de las formas mas usadas para ordenar un vector se llama Método de burbujeo.

Se van comparando los elementos del vector ,que en forma genérica será el elemento i ( vec[i] ) con el elemento j ( vec[j] ).

En la tabla que sigue se hace un esquema de cómo va evolucionando el vector a medida que avanzan los 2 *for*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i   |  |  |  |  | | --- | --- | --- | --- | | 10 | 60 | 50 | 30 |  |  |  |  |  | | --- | --- | --- | --- | | 10 | 50 | 60 | 30 | | 10 | 30 | 60 | 50 | | 10 | 30 | 50 | 60 | | j | Vec[i] | Vec[j] | If(vec[i]>vec[j]) |
| 0 | 1 | 10 | 50 | Falso |
| 0 | 2 | 10 | 60 | Falso |
| 0 | 3 | 10 | 90 | Falso |
| 1 | 2 | 60 | 50 | Verdadero |
| 1 | 3 | 50 | 30 | Verdadero |
| 2 | 3 | 60 | 50 | Verdadero |

Cuando se da la condición verdadera del if se cambian los elementos , para lo cuál se necesita un auxiliar. Si el auxiliar no se utiliza uno de los 2 valores se perdería.

# Cadenas de Caracteres

Hasta el momento no teniamos la posibilidad de ingresar texto , ya sea nombres , apellidos , direcciones , comentarios o cualquier otro dato que contenga mas de una letra.

Una cadena de caracteres es una palabra o frase (un texto cualquiera). Todos los lenguajes de programación necesariamente deben poder trabajar con este tipo de datos. Puesto que en **C** no existe un tipo de datos **“string”** o **“cadena”** que nos permita almacenar mas de una letra **,** lo que se hace es guardar la frase o palabra en un vector de caracteres.

De acuerdo a esto cada vez que necesite trabajar con datos que sean cadenas de caracteres usted deberá definir un vector de caracteres.

En el lenguaje C las cadenas se representan como una secuencia de letras terminadas en un CERO (0 o ASCII ‘\0’), por lo tanto si ingresamos la palabra hola en el vector se cargará lo siguiente

Este cero se agrega automáticamente al cargar una cadena e indica que el texto termina ahí.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| H | O | L | A | 0 |

Es importante recordar que para trabajar con cadenas debe definir un vector y que el tamaño del vector debe ser uno mas que la máxima cantidad de letras a guardar . Si necesita por ejemplo guardar 10 nombres usted debería definir una matriz de 10 filas por **N** columnas, con lo cuál usted podra guardar 10 nombres con **N-1** letras como máximo cada nombre.

Todas las tareas que usted realice con cadenas seran hechas por medio de funciones realizadas a tal fin.

A continuación se detallarán las funciones básicas mas comunes para trabajar con cadenas de caracteres.

**Nota**: En las librerías que proporciona C existen muchas funciones que trabajan con cadenas, en este apunte solamente trataremos aquellas que se utilizan en el curso de programación II.

## Inicialización de cadenas de caracteres

Al igual que podemos inicializar en algun valor una variable del tipo entero , se puede inicializar una cadena de caracteres.

Alguna de las formas de hacerlo son:

char saludo[10]=”hola”;

char saludo[10]={‘h’,’o’,’l’,’a’,’\0’};

char saludo[ ]=”hola”;

char menu [ ][3]={“Archivo”,”Nuevo”,”Abrir”};

## Funciones usadas para trabajar con cadenas de caracteres

Se describirán solo las funciones usadas durante el curso.Es de hacer notar que todas las funciones que se explicarán en este capítulo llevan como parámetros la dirección de comienzo de la cadena , que por el momento lo interpretaremos de la siguiente forma:

Si la cadena está guardada en un vector se coloca solamente el nombre del vector (no se tiene en cuenta el índice).

Si estamos tratando con una matriz , para indicar la fila , se coloca el nombre de la matriz y el índice que corresponde a la fila.

### SCANF

La función ***scanf*** permite realizar el ingreso de datos según el formato que se le indique en uno de sus parámetros.

Para el uso con cadenas de caracteres se deberá usar **%s**

**Scanf ( “ %s “ , vec )**

Indica que lo que se ingrese sea interpretado como cadena de caracteres

Dirección de comienzo del vector donde se va a guardar la palabra

***Scanf*** lee los caracteres que se ingresan por teclado hasta que se presiona un espacio o enter. Lo que se leyó lo guarda a partir de la dirección indicada por vec y coloca el terminador nulo en lugar del enter.

Tenga en cuenta que con esta función no se pueden ingresar palabras separadas por espacio ya que corta el ingreso de datos cuando detecta un espacio en blanco.

Para que el ingreso termine cuando se presiona el enter se le debe colocar en la cadena de formato lo siguiente

**scanf ( “%[^\n] “ , vec )**

***Ejemplo***: Cargar 5 nombres y mostrarlos en el órden que ingresaron

#include <stdio.h>

#include <conio.h>

#define FIL 5

#define COL 20

void main(void)

{

char mat[FIL][COL];

int i;

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

{

printf("\nIngrese nombre: ");

scanf(“%s”, mat[i]);

}

/\* Muestro los nombres ingresados \*/

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

printf("\nNombre: %s ",mat[i]);

getch();

}

### GETS (GET String)

La función gets permite leer desde el teclado cadenas de caracteres.

**gets ( vec )**

Dirección de comienzo del vector donde se va a guardar la palabra

***gets*** lee los caracteres que se ingresan por teclado hasta que se presiona un enter. Lo que se leyó lo guarda a partir de la dirección indicada por vec y coloca el terminador nulo en lugar del enter.

***Ejemplo***: Cargar 5 nombres y mostrarlos en el órden que ingresaron

#include <stdio.h>

#include <conio.h>

#define FIL 5

#define COL 20

void main(void)

{

char mat[FIL][COL];

int i;

/\* Cargo los 5 nombres en la matriz \*/

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

{

printf("\nIngrese nombre: ");

gets(mat[i]);

}

/\* Muestro los nombres ingresados \*/

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

printf("\nNombre: %s ",mat[i]);

getch();

}

### STRCMP (STRing CoMParison)

La función strcmp sirve para comparar cadenas de caracteres.

**int strcmp ( cad1 , cad2 )**

Dirección de comienzo del vector donde esta una de las palabras a comparar

Dirección de comienzo del vector donde esta la otra palabra a comparar

Como resultado de la comparación la función devuelve un valor entero que se interpreta de la siguiente manera

Si devuelve menor a cero significa que la palabra en cad1 es menor que que la palabra en cad2

Si devuelve cero significa que que la palabra en cad1 es la misma que que la palabra en cad2

Si devuelve mayor a cero significa que la palabra en cad1 es mayor que que la palabra en cad2

Se debe tener en cuenta que esta función diferencia las mayúsculas de las minúsculas, por lo tanto encontrará como distintas las palabras HOLA y hola. Si se quiere comparar sin importar mayúscula o minúscula se debe usar **stricmp.**

***Ejemplo:***Cargar 10 nombres en una matriz, luego ingresar un nombre y buscar si se encuentra en la lista.

#include <stdio.h>

#include <string.h>

#define FIL 10

#define COL 20

void main(void)

{

char mat[FIL][COL],vec[COL];

int i,x;

Cargo los 10 nombres en la matriz

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

{

printf("\nIngrese nombre: ");

gets(mat[i]);

}

Cargo el nombre a buscar en un vector

printf("Ingrese el nombre a buscar");

El nombre a buscar se carga en un vector

gets(vec);

Busco dentro de la matriz

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

Se compara cada nombre de la matriz con el nombre guardado en el vector

{

x=strcmp(mat[i],vec);

if(x==0)

printf("Se encontro el nombre");

}

}

### STRCPY (STRing CoPY)

La función *strcpy* sirve para copiar una cadena de caracteres desde un lugar a otro , es decir desde un origen a un destino.

Dirección de comienzo del vector donde se van a guardar los datos

**strcpy ( destino , origen )**

Dirección de comienzo del vector que tiene los datos a copiar

La función copia la cadena de caracteres que se encuentra guardada en un vector (cuya dirección de comienzo es *origen* ) hacia otro vector (cuya dirección de comienzo es *destino*).

***Ejemplo:*** Cargar un apellido en un vector y luego llenar una matriz de 10 filas con el mismo apellido.

#include <stdio.h>

#include <string.h>

#define FIL 10

#define COL 20

void main(void)

{

char mat[FIL][COL],vec[COL];

int i,x;

Cargo el vector con un apellido

Printf("\nIngrese nombre: ");

Gets(vec);

Copio el vector a la matriz

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

strcpy(mat[i],vec);

}

### STRLEN (STRing LENght)

La función strlen sirve para obtener la cantidad de letras que tiene una cadena de caracteres. Esta función cuenta los caracteres hasta que llega al 0(terminador de cadena), es decir , si ingresamos la siguiente cadena “Hola como estas” , la función devolvera 15 que es la cantidad de caracteres que tiene la cadena (los espacios tambien son contados).

**int strlen ( vec )**

Valor devuelto por la función.

Dirección de comienzo del vector que tiene los datos

***Ejemplo:*** Cargar un vector con una palabra y mostrar la cantidad de caracteres que tiene.

#include <stdio.h>

#include <string.h>

#define MAX 10

void main(void)

{

char vec[MAX];

int i,x;

printf("\ningrese nombre: ");

gets(vec);

x=strlen (vec );

printf("\nla cantidad de letras es %d: ",x);

}

## Matrices con cadenas de caracteres

Sabemos que para guardar varios nombres necesitamos varios vectores o sea una matriz. La cantidad de filas que tenga la matriz va a depender de la cantidad de nombres que se quieran ingresar y la cantidad de columnas esta asociada directamente con el número máximo de letras del texto a guardar.

Tenga en cuenta que para cargar una matriz con cadenas de caracteres debo posicionarme al comienzo de cada fila, ya que es a partir de alli donde se carga la cadena.

Por lo tanto cuando este cargando la matriz debo moverme de fila en fila y la forma de posicionarme al comienzo de cada fila es escribir

mat[i] o sea la dirección de comienzode la fila i

Si estoy trabajando con una matriz de caracteres definida como

char mat [ FIL][COL]

Al escribir mat[0] me indica al dirección de comienzo de la primer fila , mat[1] me indica la dirección de comienzo de la segunda fila y en forma genérica mat [i] es la dirección de comienzo de la fila i.

Significa que cuando usted deba cargar una matriz usara un ciclo ***for*** y cargará fila por fila de la siguiente forma

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

{

printf("\nIngrese nombre: ");

gets(mat[i]);

}

***Ejemplo tipo:*** Cargar 20 nombres, mostrarlos ordenados alfabéticamente y luego mostrar el nombre que mas letras tiene.

#include <stdio.h>

#include <string.h>

#define FIL 20

#define COL 25

void main(void)

{

int i,j,x,lmax=0,imax;

char mat[FIL][COL],aux[COL];

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

{

printf("Ingrese nombre: ");

gets(mat[i]);

}

Ordeno alfabeticamente la matriz

for(i=0;i<FIL-1;i++)

for(j=i+1;j<FIL;j++)

{

if((strcmp(mat[i],mat[j]))>0)

{

strcpy(aux,mat[i]);

strcpy(mat[i],mat[j]);

strcpy(mat[j],aux);

}

}

Busqueda del nombre mas largo

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

{

x=strlen(mat[i]);

if(x>lmax)

{

lmax=x;

imax=i;

}

}

printf("El nombre mas largo es %s",mat[imax]);

}

Ordenamiento de una matriz de cadenas

El algoritmo para ordenar una matriz que contiene cadenas de caracteres tiene el mismo formato que el ordenamiento de un vector.

La razón es simple, una matriz que contiene cadenas de caracteres no es mas que un vector donde cada elemento (una fila de la matriz ) es una palabra.

Entonces en resumen debemos considerar que ordenamos un vector de palabras y lo tratamos como tal.

Como se ve en el ejemplo la forma del ordenamiento es exactamente la misma que se usa para ordenar un vector, solo cambia como se tratan los distintos tipos de datos.

Por ejemplo cuando en el ordenamiento de números enteros se llegaba a la comparación se analizaba si el elemento i era mayor o menor que el j. Con cadenas de caracteres quien hace la comparación es la función ***strcmp*** a la cual le pasamos la fila i y la fila j.

En el ordenamiento de números enteros se usaba el auxiliar y las asignaciones para realizar el cambio de datos. Con cadenas de caracteres el tratamiento es el mismo, la diferencia es que para realizar asignaciones se debe usar la función ***strcpy*** y que el auxiliar debe ser un vector.

## Comprobación de la longitud de un texto

Hasta ahora cuando ingresamos un texto (nombre , apellido, etc) usamos las funciones scanf o gets y le indicámos cual es la dirección de memoria a partir de la cual se tiene que cargar el texto. Esto es correcto siempre y cuando se ingrese un texto de N letras y el vector tenga como mínimo N+1 lugares. El problema se presenta si se ingresa un texto que tiene mas letras que cantidad de elementos que dispone el vector.

Para entender este concepto vamos a dar un ejemplo que no tiene que ver con la programación.

Imagine que usted tiene ganas de ir al cine a ver una pelicula que se estrena y reserva entradas para 3 personas (usted y 2 amigos). Antes de salir de su casa , lo van a visitar otros 2 amigos que tambien van al cine a ver la misma pelicula pero ellos no reservaron entradas. Entonces se dirijen todos al cine entran primeros y se sientan en la misma fila , uno al lado del otro. De los 5 asientos ocupados solo 3 estan reservados , los otros dos corresponden a otras personas que los van a reclamar cuando lleguen.

Lo que ocurrió en este caso es que se ocuparon lugares que no estababan reservados , si usted hubiera reservado mas lugares de los que iba a ocupar esto no hubiera ocurrido. Evidentemente nadie hace esto por que debe pagar por los lugares , los ocupe o no.

El mismo efecto puede ocurrir cuando trabajamos con cadenas de caracteres. Veamos como es esto con un ejemplo muy simple

#include <stdio.h>

#include <conio.h>

#define MAX 5

void main(void)

{

char vec[MAX];

/\* Cargo el vector \*/

printf("\nIngrese nombre: ");

gets( vec );

}

Cuando yo defino el vector lo que estoy haciendo es reservar MAX lugares de memoria para que sean ocupados por la palabra que ingreso. En nuestro ejemplo estoy reservando 5 lugares.

Supongamos que ingreso el nombre Gerardo

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| G | e | r | a | r | d | o | ‘\0’ |

Como vemos en el dibujo de arriba el vector se ocupa por completo y parte de las letras quedan fuera de el, en el lugar que esta reservado para otras variables.

Esto ocurre por que a gets se le pasa la direccion de comienzo y es a partir de alli donde se comienzan a colocar las letras ingresadas y se colocan sin importar si parte de el texto queda fuera del vector.

Si en lugar de haber reservado 5 lugares para el vector , se hubieran reservado 10 , nada de esto hubiera ocurrido.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| G | e | r | a | r | d | o | ‘\0’ |  |  |

Observe que en este caso se ocupa parte de los lugares reservados , pero lo mas importante es que no se ocupa el lugar de otras variables.

Entoces podemos pensar en una primera solución, que sería hacer un vector gigante que sea capaz de contener el texto.

Pero la pregunta sería...Cual sería el tamaño de ese vector gigante?....100 , 1000, 5000 caracteres?

Entonces piense que para evitar este problema usted esta reservando mucha memoria que no va a usar , por ejemplo si elige usar un vector de 1000 elementos estaria usando 1Kbyte de memoria para guardar un solo nombre. Y si tiene que gusrdar 100 nombres estaría usando 100Kbytes , con lo cual su programa estaría usando 100K solo para parte de los datos. Esta solución no es la correcta.

Una segunda alternativa sería usar un vector auxiliar suficientemente grande donde se cargan los datos, se miden y en caso de aceptación se cargan en el destino, caso contrario se pide el reingreso. Este vector auxiliar puede pensarse como un cuaderno borrador en el cual se realizan pruebas antes de hacer la version definitiva. El tamaño del vector auxiliar puede considerarse suficientemente grande tomando10 veces mas que la matriz o vector definitivos.

Esta alternativa es aceptable aunque tiene algunas desventajas como ser la utilización del vector auxiliar y que puede ocurrir sobre el vector auxiliar el mismo problema que estamos tratando.

La tercer alternativa sería contar la cantidad de letras que se ingresan mientras estas se estan ingresando y actuar en consecuencia.Esta última opción nos impide el uso de ***scanf*** o ***gets***.

Tenga en cuenta que si usted carga un nombre en el vector definitivo y luego procede a la validación , esta empleando una forma incorrecta de trabajar, ya que si se exedió en el tamaño del texto, los datos estan pisados y no hay forma de volverlos a su valor original.

# Estructuras

Imaginemos que alguien nos pide desarrollar un programa que nos permita administrar los datos de personas , por ejemplo Nombre , apellido , dirección , provincia , localidad , telefono y fecha de nacimiento.

Con los conocimientos adquiridos hasta el momento lo primero que usted plantea es en que tipo de variables va a guardar los datos que se ingresen. Puede pensar en principio en 5 matrices , una para nombre , otra para apellido , direccion provincia y localidad. Luego puede dudar un poco en donde guarda los telefonos y para la fecha de nacimiento utiliza 3 vectores de enteros.

Las cosas asi planteadas , permiten resolver el problema , pero debido a la gran cantidad de datos a ingresar se torna denso el manejo del código.

Ahora piense en lo siguiente , cuando usted se compa una agenda con índice telefónico y desea llenar los datos de sus conocidos (nombre, apellido, dirección etc...) no tiene una hoja para los nombres otra para los apellidos otra para las direcciones, etc. Lo que tiene es una hoja en la cual guarda todos los datos de una persona , por lo tanto necesitará tantas hojas como personas desee ingresar.

En definitiva la cantidad y tipo de datos son los mismos solo cambia la forma en la cuál se los agrupa. Hasta el momento solo sabe agrupar datos que sean del mismo tipo , la idea es que por una cuestión de practicidad y para facilitar el manejo de gran cantidad de información pueda agrupar de alguna forma datos que sean de diferentes tipos.

Este forma de agrupar datos es conocida en el lenguaje C con el nombre de estructuras de datos.

## Definición

Una estructura de datos es un conjunto de variables de distinto tipo a la cuál se hace referencia bajo un mismo nombre.

Siguiendo con el ejemplo mencionado en la introducción , puedo definir una estructura que contenga variables (vectores) para guardar el nombre, apellido , dirección , localidad , telefono y 3 enteros para guardar el día mes y año de nacimiento. A todo este conjunto de variables lo llamo agenda y éste será el nombre de referencia con el cuál se conocerá la estructura de datos.

Tenga en cuenta que la estructura es un dato definido por el usuario y por lo tanto puede tener la cantidad de variables que se crean convenientes.

A las variables dentro de la estructura se las suele llamar “**miembro de la estructura**” o “**campo**”.

## Declaración de una estructura

Dado que una estructura es una variable definida por el programador , se hace necesario que el compilador conozca el formato que va a tener la o las estructuras que componen el programa. Para esto se describe fuera de todas las funciones y al comienzo del programa cual es el formato de la estructura. La forma genérica es:

***struct*** nombre {

tipo variable\_1;

tipo variable\_2;

............

.............

tipo variable\_n;

};

Donde ***struct*** es la palabra reservada que le indica al compilador que se esta declarando una estructura.

nombre es optativo y se lo llama nombre de referencia o etiqueta de la estructura.

Entre llaves se colocan todas las variables que se requieran en la estructura, terminadas por ;

Tenga en cuenta que lo que se hizo hasta el momento es solamente definir el formato de la estructura **, no se declaro ninguna variable estructura.**

Cuando usted necesite definir una variable estructura para utilizarla dentro del programa , escribirá lo siguiente

struct nombre var\_1,var\_2,.............,var\_n;

Donde struct nombre se refiere al formato de estructura y var\_1, var\_2 son las variables que se pueden usar dentro del programa.

Analicemos un ejemplo mas concreto.

Se necesitan guardar los datos de los medicamentos de una farmacia como ser nombre del laboratorio, nombre del medicamento y precio. De acuerdo a esto podemos definir una estructura con el siguiente formato

struct remedio {

char laboratorio[30];

char nombre[20];

float precio;

};

cuando tenga que declarar una variable solo tiene que escribir

struct remedio medi1,medi2;

Variable

Tipo de dato

Para aclarar en que parte del programa escribe la declaración del formato de la estructura y la variable se verá un ejemplo.

#include <stdio.h>

**struct remedio {**

El formato de la estructura se define fuera de todas las funciones.

**char laboratorio[30];**

**char nombre[20];**

**float precio;**

**};**

void funcion1(int a,float b);

void main(void)

{

int x,y,z;

Se declaran para su uso las variables medic1 y medic2 que son variables del tipo struct remedio y son locales de main.

**struct remedio medic1,medic2;**

................

................

................

................

................

}

Se declara la variable x del tipo struct remedio y es local de la funcion1.

Vec es un vector de estructuras

void funcion1(int a,float b)

{

**struct remedio x,vec[10];**

int suma;

................

................

}

A pesar de trabajar con un nuevo tipo de variable (estructura de datos) se siguen manteniendo y aplicando los mismos conceptos que para variables comunes , es decir , se pueden tener variables de tipo estructura que sean locales y globales.

En el ejemplo se agrego como variable un vector que es del tipo struct remedio , es decir , se tiene un vector de 10 estructuras.

A dicho vector se lo puede imaginar graficamente de acuerdo al siguiente diagrama

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Laboratorio | | | | | |  |  |  |  | Laboratorio | | | | | |  |  |  |  | Laboratorio | | | | | |  |  |  |  |  |  |  | Laboratorio | | | | | |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Nombre | | | |  |  |  |  |  |  | Nombre | | | |  |  |  |  |  |  | Nombre | | | |  |  |  |  |  |  |  |  |  | Nombre | | | |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Precio | | | |  |  |  |  |  |  | Precio | | | |  |  |  |  |  |  | Precio | | | |  |  |  |  |  |  |  |  |  | Precio | | | |  |  |  |  |  |
| vec[0] | | |  |  |  |  |  |  |  | vec[1] | | |  |  |  |  |  |  |  | vec[2] | | |  |  |  |  |  |  |  |  |  |  | vec[MAX-1] | | | | | |  |  |  |  |

## Acceso a los campos – Operador punto

Con la introducción del concepto de estructura lo que se está haciendo es empaquetar datos , pero en definitiva necesitamos cargar las variables que se encuentran dentro de la estructura ,para ello se utiliza el operador punto . La forma genérica de usar este operador es la siguiente:

*Nombre\_de\_variable\_estructura . Nombre\_del\_campo*

si tomamos el ejemplo que se dio anteriormente donde esta la estructura remedio y la variable medic1 y queremos cargar el precio, se debería escribir de la siguiente forma

medic1.precio=10.25;

strcpy ( medic1.laboratorio , ”Bayer” );

strcpy ( medic1.nombre , ”Aspirina” );

Aqui cargamos la estructura completa asignando a cada campo un valor. En el caso en que se deba ingresar el dato desde el teclado (normalmente se da esta situación) se escribirá lo siguiente

scanf ( “%f” , &medic1.precio );

gets (medic1.laboratorio);

scanf ( “%s” , medic1.nombre);

En definitiva , el uso del operador punto nos permite acceder a un campo de la estructura. Observe que aunque se este trabajando con una estructura , en definitiva llegamos al campo por medio del operador punto y se esta trabajando con variables conocidas y anteriormente usadas. Lo que se pretende remarcar con esto es que la forma de trabajar sigue siendo la misma . Por ejemplo cuando usted debia cargar una variable del tipo ***float*** como precio y usaba ***scanf*** tambien usaba el ***&*** precediendo al nombre de la variable , por lo tanto cuando utilice estructuras con uno de los campos que sea ***float*** tambien deberá usar el ***&*** precediendo a la variable de la misma forma que se muestra en el ejemplo de arriba.

Cuando cargaba una cadena de caracteres en un vector , no usaba ***&*** , por lo tanto cuando tenga que cargar un vector que se encuentra dentro de una estructura tampoco debe usar el ***&***.

## Vector de estructuras

En la mayoría de los programas que se realicen se va a necesitar ingresar una determinada cantidad de datos. Al igual que ocurria en los programas realizados en programación I cuando se ingresaban datos para luego usarlos se los guardaba en un vector. Con las estructuras ocurre lo mismo , es decir si se necesita ingresar por ejemplo nombre y nota de 20 alumnos se utiliza un modelo de estructura que contenga un campo nombre y un campo nota y luego como variable se utiliza un vector de 20 estructuras para guardar los datos.

Para cargar dicho vector hay que situarse en cada uno de los elementos del vector y recien ahi cargar los campos de la estructura.

El código sería:

void main (void)

{

struct alumno alu[20];

for(i=0;i<20;i++)

{

printf("Ingrese nombre: ");

gets(alu[i].nombre);

printf("Ingrese nota: ");

scanf("%d",&alu[i].nota);

fflush(stdin);

}

…………….

………………………

}

La forma de cargar el vector es la misma que cuando se carga un vector de enteros , la diferencia es que por cada elemento del vector debo cargar todos los campos de la estructura.

## Estructuras anidadas

Una estructura anidada es una estructura que se encuentra dentro de otra.

De acuerdo al problema que se necesite resolver se van a agrupar los datos de forma de usar estructuras. Ahora imagine que tiene que ingresar Nombre, apellido , dirección y fecha de nacimiento , la estructura que se puede definir sería:

struct gente {

char nom[20],ape[20],dir[30];

int dia,mes,anio;

};

Este formato de estructura funciona perfectamente y puede resolver nuestro problema. Pero piense que normalmente día , mes y año se tratan siampre juntas , es como si hablamos de cada uno por separado no tiene demasiado sentido. Lo mismo ocurre si se desea trabajar con horas minutos y segundos.

Por lo tanto estos 2 ejemplos ( fecha y hora) son adecuados para trabajarlos con estructuras.

Entonces podemos redefinir la estructura inicial para poner una estructura anidada que tenga que ver con la fecha de nacimiento , la nueva estructura quedaría

struct fecha { int dia,mes,anio;};

struct gente {

char nom[20],ape[20],dir[30];

struct fecha fecha\_nac;

};

Con la nueva definición de la estructura se necesita definir las 2 estructuras a usar (la principal y la anidada) , y la que se encuentra anidada debe definirse primero.

Cuando se define una estructura anidada es necesario que coloque un nombre de variable dentro de la estructura principal , es decir , dentro de la estructura gente tengo una variable llamada ***fecha\_nac*** del tipo ***struct fecha.***

Si quiere imaginarse gráficamente una estructura puede imaginarse una caja con cosas adentro , o sea , la caja es la variable estructura y las cosas que tiene adentro son las variables.

Ahora , una estructura anidada se puede ver como una caja (estructura principal) que tiene cosas adentro (variables), dentro de esas cosas hay una caja (estructura anidada)que tiene cosas adentro (variables de la estructura anidada). En definitiva las cosas que sirven son las variables , las estructuras simplemente mantienen ordenadas las variables.

Para acceder a las variables que se encuentran dentro de la estructura anidada se procede usando el operador punto.

#include <stdio.h>

struct fecha { int dia,mes,anio;};

struct gente {

char nom[20],ape[20],dir[30];

struct fecha fecha\_nac;

};

void main(void)

{

struct gente persona;

strcpy(persona.nom,"Gustavo");

strcpy(persona.ape,"Fernandez");

persona.fecha\_nac.dia=20;

persona.fecha\_nac.mes=3;

persona.fecha\_nac.anio=1963;

...........

...........

}

Observe que para cargar dia , mes y año se comienza desde la variable estructura exterior , con el punto se baja a la anidada y nuevamente con punto se accede al campo. Tenga en cuenta que siempre lo que se carga son los campos , por lo tanto para llegar a los campos mas interiores se utilizan tantos operadores punto como estructuras anidadas existan.

## Propiedades

### Asignación

Se puede realizar la asignación de una estructura a otra siempre y cuando las 2 estructuras tengan el mismo formato.

**Ejemplo 1**

Cargar una variable estructura para luego copiar los datos a una segunda estructura

#include <stdio.h>

struct alumno {

char nombre[20];

int nota;

};

void main(void)

{

struct alumno alu1 , alu2;

printf("Ingrese nombre");

gets(alu1.nombre);

printf("Ingrese nota");

scanf("%d",&alu1.nota);

alu2=alu1;

}

Como se ve en el ejemplo se define una estructura y 2 variables de ese tipo que son alu1 y alu2.

Se carga la variable alu1 y luego se pretende asignar lo que esta cargado en alu1 a alu2. De acuerdo a la propiedad de asignación de estructuras se puede hacer dado que las 2 variables son del mismo tipo de estructura. Lo que realmente pasa es que se copian los datos desde la estructura alu1 hacia alu2 sin necesidad de tener que copiar campo por campo los datos.

Si no existiera esta propiedad se debería hacer lo siguiente

strcpy (alu2.nombre , alu1.nombre);

alu2.nota=alu1.nota;

Imagine que el caso que estamos tratando no es complicado ya que solo se cuenta con 2 campos , pero si la estructura tuviese mas campos , se debería asignar uno por uno y la tarea sería tediosa.

**Ejemplo 2**

Cargar los datos en una estructura para luego copiar los datos a otra.

#include <stdio.h>

#include <string.h>

#include <conio.h>

struct fecha { int dia,mes,anio;};

struct gente {

char nombre[20];

struct fecha f\_nacimiento;

};

void main(void)

{

struct gente pers;

struct fecha fn;

printf("Ingrese nombre");

gets(pers.nombre);

printf("Ingrese dia de nacimiento");

scanf("%d",&fn.dia);

printf("Ingrese mes de nacimiento");

scanf("%d",&fn.mes);

printf("Ingrese a¤o de nacimiento");

scanf("%d",&fn.anio);

pers.f\_nacimiento=fn;

}

En este ejemplo se trata de ver lo mismo que en el anterior pero usando estructuras anidadas. Observe que se sigue cumpliendo la propiedad , es posible asignar siempre y cuando las estructuras tienen la misma forma y efectivamente pers.f\_nacimiento no es mas ni menos que la estructura fecha.

### Comparación

No es posible realizar comparación entre 2 estructuras. Lo que se hace es comparar los campos del mismo tipo entre 2 estructuras. Imagine el ejemplo 1 dado en el punto 1.6.1 como hace para comparar las 2 variables como sabe cual es mayor , menor o igual. Solo se puede comparar entre 2 nombres o 2 notas.

Por lo tanto tenga en cuenta que siempre que necesite realizar comparaciones entre 2 variables estructura será necesario establecer el criterio de comparación.

Todos los operadores relacionales (usados para comparar) que fueron utilizados hasta el momento son aplicables a la comparación entre los campos de la estructura. En definitiva lo que se está haciendo es comparar 2 variables simples , ya que con el operador punto se llega a la variable.

**Ejemplos**

if ( alu1.nota == alu2.nota)

if (alu2.nota < alu1.nota )

if ( ¡ strcmp ( alu1.nombre , alu2.nombre))

**Ejemplo**

Se desarrollará el algoritmo para ordenar alfabéticamente un vector de estructuras de MAX elementos en el cuál se van a aplicar las 2 propiedades vistas.

#include <stdio.h>

#include <string.h>

#include <conio.h>

#define MAX 10

struct alumno {

char nombre[20];

int nota;

};

void main(void)

{

struct alumno pers[MAX],aux;

..............

..............

for(i=0;i<MAX-1;i++)

for(j=i+1;j<MAX;j++)

if((strcmp(pers[i].nombre,pers[j].nombre))<0)

{

aux=pers[i];

pers[i]=pers[j];

pers[j]=aux;

}

..............

..............

}

Notas:

* Los campos de una estructura pueden ser cualquiera de los tipos de variables conocidas (char , int , float , double) y tambien pueden ser vectores , matrices o punteros.
* No se puede tener como estructura anidada a la misma estructura que se esta definiendo , por ejemplo

struct alumno {

char

# Punteros

Cuando se escribe un programa una de las primeras cosas que se hacen es definir las variables, es decir ,debemos explicitar el nombre de la variable y que tipo de dato guarda. A partir de ese momento cada vez que se desee cargar un valor, compararlo o mostrarlo , simplemente invocamos el nombre de la variable que contiene el dato y realizamos la operación correspondiente.

No concebimos la idea de realizar un programa donde pueda existir una variable que no tenga un nombre ya que de existir, el problema sería como hacer para referirme a ella.

Ahora veamos lo siguiente, imagine que usted está escribiendo un programa en C pero no tiene ninguna de las funciones que le permiten mostrar los datos por pantalla , y como es lógico necesita comunicar los resultados de una determinada operación por pantalla. En este caso como haría para solucionar el problema? ,o lo que es lo mismo , como hacen las funciones como printf , puts , putch para mostrar datos en pantalla?.

Una de las soluciónes para este problema es escribir el dato en la posición de memoria a partir de la cuál comienza la memoria de video , pero ese lugar no tiene asociado un nombre, solo se sabe en que lugar está (o sea la dirección de memoria).

Veamos este planteo desde otro punto de vista. Piense que usted desea mandarle una carta a un amigo o familiar , entonces escribe la hoja , la guarda dentro de un sobre y le pone el destinatario (nombre de su amigo o familiar) para enviarla. Al momento de llevarla al correo le van a informar que falta un dato , ese dato es la dirección donde se debe dejar la correspondencia , es decir que en algún momento , al margen de para quien va la carta , usted debe informar la dirección donde la deben dejar.

Algo similar ocurre con los programas. Usted mientras escribe el programa (programa fuente) conoce los nombres de sus variables, pero existen ocasiones en las cuales el nombre de la variable es desconocido y el único dato del que se dispone es el lugar donde se encuentra esa variable , o sea su dirección de memoria.

Recuerde que todos los programas corren en memoria RAM , están cargados en ella , y el código que se está ejecutando son instrucciones del microprocesador , por lo tanto las órdenes que se manejan son básicas , por ejemplo leer desde la posición de memoria 1000 y comparar ese dato contra lo que tiene guardado el procesador en uno de sus registros.

## Definición

Un puntero es una variable destinada a guardar una posición de memoria. Esta posición de memoria generalmente corresponde a otra variable.

De acuerdo a la definición , un puntero es una variable como cualquier otra . La diferencia es el significado de lo que guarda la variable puntero. Es importante que entienda lo siguiente:

**Un puntero siempre guarda una dirección de memoria**

Lo que se pretende decir es que si a un puntero se le carga cualquier valor , éste será interpretado como si fuera una dirección de memoria aunque el valor cargado no corresponda a una dirección de memoria.

Veamos una representación gráfica de la memoria

|  |  |  |
| --- | --- | --- |
| 2000h |  |  |
| 2001h |  |  |
| 2002h | **56** | Contador |
| 2003h | **0** | i |
| 2004h |  |  |
| 2005h |  |  |
| 2006h |  |  |
| Dirección de memoria | **Memoria** |  |

En el diagrama se puede observar del lado izquiero las direcciones de memoria , se debe tener en cuenta que dichas direcciones estan expresadas en sistema hexadecimal (por eso la h).

En el centro figura el valor que tiene cada posición de memoria .En nuestro ejemplo la dirección 2002 tiene guardado el número 56 que corresponde a una variable llamada contador y en la posición 2003 hay guardado un 0 (cero) correspondiente a la variable i.

Habiendo entendido este diagrama , modifiquemos para que un puntero llamado p apunte a la dirección de la variable Contador

|  |  |  |
| --- | --- | --- |
| 2000h |  |  |
| 2001h |  |  |
| 2002h | **56** | Contador |
| 2003h | **0** | i |
| 2004h |  |  |
| 2005h | **2002** | p |
| 2006h |  |  |
| Dirección de memoria | **Memoria** |  |

Como el puntero p es una variable , al igual que cualquier variable , se va a almacenar en memoria , en nuestro caso en particular se guarda en la posición 2005. Cuando se pone el puntero p apuntando a la variable contador , lo que se está haciendo es asignar al puntero la dirección de memoria de la variable contador que es 2002.

La utilización de punteros le da una gran potencia al lenguaje C , pero también trae inconvenientes en la depuración de los programas ya que es posible alcanzar cualquier dirección de memoria.

Algunas de las aplicaciones de los punteros son:

1. Permiten el acceso a cualquier posición de memoria para ser leida o escrita . Por ejemplo permiten el acceso a la memoria de video.
2. Permiten el pasaje de argumentos a las funciones por referencia.
3. Permiten realizar las funciones que manejan asignación dinámica de memoria. La asignación dinámica de memoria se utiliza cuando se necesita mas memoria de la solicitada en forma estática por medio de los vectores o matrices.
4. Se utilizan con estructuras de datos para manejar listas simple y doblemente enlazadas ,colas , pilas y árboles.

## Declaración de un puntero

La forma genérica de declarar un puntero es la siguiente:

**Tipo** \* nombre\_variable

Nombre de la variable puntero

Tipo de dato de la variable

Int, char, float...

Es quien indica que la variable es un puntero

Por ejemplo

Int x , \*p , vec[20];

Char nombre[15] , rta , \*punt ;

En el ejemplo tenemos 2 variables puntero , p es un puntero a enteros y punt es un puntero a carácter. Observe que de acuerdo a lo explicado en la declaración , la forma de reconocer que una variable es un puntero , es cuando tiene el \* delante del nombre de variable.

Debido a que el \* aparece repetidas veces durante el transcurso de un programa con interpretaciones distintas , se recomienda prestar atención al significado que toma en cada caso.

Cuando aparece el la línea de declaración de las variables significa que se está declarando un puntero.

## Operadores

Existen 2 operadores que se utilizan directamente con los punteros , tales operadores son **\* y &**

**Operador &**

Se utiliza para obtener la dirección de memoria de una variable.

Sea la siguiente definición de variables

int x , \*p;

p=&x;

En la primera línea se definen 2 variables enteras , la primera de ellas es x y la segunda un puntero llamado p.

En la segunda línea se asigna la dirección de la variable x al puntero p.

Esto es correcto ya que por definición un puntero guarda la dirección de memoria de una variable. El operador & se puede leer como “LA DIRECCIÓN DE MEMORIA DE” , por lo tanto al escribir **&x** estamos diciendo “La dirección de memoria de x”.

**Operador \***

Se utiliza para obtener el contenido de una posición de memoria. Este operador solo es aplicable a punteros.

Sean las siguientes líneas de programa

int x , y , \*p;

p=&x;

x=2;

y=\* p;

Las 2 primeras líneas vienen del ejemplo del operador & , la última línea lo que hace es asignar a y el contenido de lo apuntado por p. El resultado va a ser que x e y tengan el mismo valor.

Al operador \* lo podemos leer como “EL CONTENIDO DE LO APUNTADO POR”.

Veamos gráficamente el ejemplo dado. Cuando el programa arranca en la primera línea las variables tienen cualquier valor (lo simbolizamos con ¿?).

|  |  |  |
| --- | --- | --- |
| 2000h | **¿?** | x |
| 2001h |  |  |
| 2002h | **¿?** | y |
| 2003h |  |  |
| 2004h | **¿?** | p |
| 2005h |  |  |
| 2006h |  |  |

En la segunda línea se inicializa el puntero , es decir se le asigna la dirección de memoria de x. Como la variable está ubicada en la dirección 2000 , al puntero se le asignará ese valor.

|  |  |  |
| --- | --- | --- |
| 2000h | **¿?** | x |
| 2001h |  |  |
| 2002h | **¿?** | y |
| 2003h |  |  |
| 2004h | **2000** | p |
| 2005h |  |  |
| 2006h |  |  |

La tercera línea asigna un valor a la variable x quedando el mapa de la memoria de la siguiente forma

|  |  |  |
| --- | --- | --- |
| 2000h | **2** | x |
| 2001h |  |  |
| 2002h | **¿?** | y |
| 2003h |  |  |
| 2004h | **2000** | p |
| 2005h |  |  |

Finalmente se toma el contenido de lo apuntado por p y se le asigna a y .El puntero p apunta a la dirección 2000 y el contenido de esa dirección es el número 2.

|  |  |  |
| --- | --- | --- |
| 2000h | **2** | x |
| 2001h |  |  |
| 2002h | **2** | y |
| 2003h |  |  |
| 2004h | **2000** | p |
| 2005h |  |  |
| 2006h |  |  |

## Asignación de punteros

Existen 3 formas básicas para la asignación de punteros

1. Por medio del operador &: Al puntero se le asigna la dirección de memoria de una variable a través del operador & tal cual se vió en el punto 2.3

**Ejemplo:**

char c , \* punt;

punt =&c;

1. Por medio de otro puntero: Al puntero se le asigna el valor de otro puntero con lo cuál podemos decir que ambos punteros apuntan a la misma variable

**Ejemplo:**

int x , \*p , \*q;

p=&x;

q=p;

1. Con la dirección de memoria: Al puntero se le asigna directamente la dirección de memoria a la cuál va a apuntar. Este caso se utiliza cuando se desea asignar una dirección de memoria conocida como es el caso de la memoria de video o la parte baja de la memoria en donde se guardan datos referidos al sistema operativo.

**Ejemplo:**

int \*pvar;

pvar=0x24A3;

Como se hizo notar al comienzo del capítulo las direcciones de memoria normalmente se manejan en hexadecimal , por lo tanto para expresar un número en hexadecimal se debe anteponer un 0x.

Es importante tener en cuenta que los punteros SIEMPRE DEBEN SER INICIALIZADOS. En el caso que no hayan sido inicializados quedarán apuntando a una dirección de memoria desconocida en donde pueden existir datos que tal vez no sean los que nos interesan y en el caso de sobreescribirlos nos podemos encontrar con resultados inesperados.

A un puntero se le puede asignar cualquier valor sin que aparezcan errores en la compilación o ejecución de un programa , pero recuerde que si ese valor no es una dirección de memoria de todas formas será interpretado como si lo fuera.

Veamos gráficamente que pasa al no inicializar un puntero

int x , \*punt;

\*punt=3;

|  |  |  |
| --- | --- | --- |
| 2000h | **¿?** | x |
| 2001h |  |  |
| 2002h | **3025** | p |
| 2003h |  |  |
| 2004h |  |  |
| 2005h |  |  |
|  |  |  |
| 3025h | **18** | p |
| 3026h | **40** |  |

Al no inicializar el puntero , de acuerdo al dibujo , tenemos el puntero apuntando a la posición 3025 que puede o no corresponder a nuestro programa. Al momento de asignarle el valor 3 estamos sobreescribiendo el valor original que existía en esa posición de memoria.

|  |  |  |
| --- | --- | --- |
| 2000h | **¿?** | x |
| 2001h |  |  |
| 2002h | **3025** | p |
| 2003h |  |  |
| 2004h |  |  |
| 2005h |  |  |
|  |  |  |
| 3025h | **3** | p |
| 3026h | **40** |  |

## Punteros lejanos y cercanos

Para entender este concepto se debe comentar algo mas acerca de la memoria de las computadoras.

Las computadoras basadas en los procesadores Intel manejan el concepto de memoria segmentada , es decir que a la memoria la dividen en segmentos , cada uno de tamaño 64K.

Entonces cada posición de memoria esta constituida por un segmento y un desplazamiento cuya notación es

Segmento : Desplazamiento

por ejemplo una dirección de memoria podría ser B800:0000 donde el segmento es B800 y el desplazamiento es 0.

Esta dirección de memoria es donde se encuentra la memoria de video delas placas VGA (Video Graphic Array).

La totalidad de los programas que se manejan en este curso pueden correr tranquilamente en un segmento es decir dentro de los 64K de memoria de datos.

Por lo tanto si corren dentro de los 64K solo cambia el desplazamiento siendo el segmento el mismo , entonces un puntero va a guardar solo la porción de memoria correspondiente al desplazamiento ya que el segmento no cambia.

Este tipo de punteros son los punteros cercanos y es el valor por default que toma el compilador al momento de generar el puntero.

En cambio si tenemos datos que ocupan mas de 64K , nos estaremos moviendo a través de 2 segmentos , por lo tanto se deberán guardar las 2 componentes de la dirección de memoria. En este caso el puntero va a ser lejano ya que necesita guardar segmento y desplazamiento.

Un puntero lejano se declara anteponiendo el modificador far , por ejemplo:

int **far** \*punt;

Observe que punt es un puntero lejano.

Si por ejemplo se desea trabajar con la memoria de video para escribir algo en la pantalla , deberemos definir un puntero far.

Por defecto se asume que un puntero es near, salvo que se le agregue el modificador far. Sea la siguiente definición de variables

char \*pcercano , far \*plejano , c;

Tenemos 3 variables del tipo char , pcercano es un puntero a char que al no tener modificador se asume como cercano (near) , plejano es un puntero lejano a carácter ya que tiene el modificador far y finalmente c es una variable de tipo caracter.

Hagamos un programa simple para escribir una letra en la pantalla pero sin usar ninguna de las funciones conocidas. Para esto usaremos un puntero lejano apuntando a la dirección de memoria del video.

void main (void)

{

int far \*pvideo;

/\* Inicializo el puntero a la dirección B800:0000 que corresponde a la memoria de video para VGA \*/

pvideo=0xB8000000;

/\* Cargo una letra en el angulo superior izquiero de la pantalla \*/

\*pvideo=0x7025;

}

Existe otra forma de asignar el puntero lejano y es a traves de la función MK\_FP ( segmento , desplazamiento ) .Para mas información acerca de esta función consulte la ayuda del compilador.

## Comparación de punteros

Los punteros al igual que las variables conocidas se pueden comparar. Los operadores usados en la comparación (operadores relacionales) son los ya conocidos o sea = = ¡= < > <= >=

La comparación de punteros es válida siempre y cuando respete una lógica. Se le ocurriría comparar el peso de una persona con la altura de la otra?...probablemente le sea lógico comparar alturas de personas o peso de personas. Lo mismo ocurre con los punteros.

Un puntero contiene una dirección de memoria por lo tanto puede ser comparado contra otra posición de memoria , de acuerdo a esto existen 3 formas para comparar punteros

- Un puntero contra otro puntero

- Un puntero contra la dirección de memoria de una variable expresada por medio del operador &

- Un puntero contra una dirección de memoria dada directamente.

**Ejemplo**

void main (void)

{

int \*pa , \*pb , x;

pa=&x;

pb==pa;

if ( pa==&x ) //comparación de un puntero contra direccion de memoria expresada por &

printf(“El puntero esta apuntando a la variable x”);

if ( pb ¡=pa) //Comparación de un puntero contra otro

printf(“Los 2 punteros apuntan a lugares distintos”);

if ( pa > 0x0400) //Comparación contra una dirección fija

printf (“El puntero esta direccionado mas alla de la direccion 0x400”);

}

Los ejemplos dados corresponden a cada forma de comparación enunciada anteriormente.

## Aritmética de punteros

Este punto trata acerca de las operaciones aritméticas que pueden realizarse con los punteros.

### Suma y resta de enteros

A todo puntero se le puede sumar o restar un número entero. El efecto de sumar o restar un entero a un puntero es desplazar a partir de la dirección apuntada por el puntero tantos elementos del tipo de dato como valor tenga el entero que se suma o resta.

Para aclarar esto vamos a poner algunos ejemplos.

**Ejemplo**

int \*p , x;

x=5;

p=&x;

|  |  |  |
| --- | --- | --- |
| 2000h |  | p-1 |
| 2001h |  |  |
| 2002h | **5** | p |
| 2003h |  |  |
| 2004h |  | p+1 |
| 2005h |  |  |
| 2006h |  | p+2 |
| 2007h |  |  |
| 2008h |  |  |

En el ejemplo se definen 2 variables una es un puntero (p) que se inicializa para apuntar a la variable x.

Como la variable es entera el puntero debe ser entero por lo tanto cada vez que se sume o reste un número al puntero se va a estar desplazando la misma cantidad de enteros , es decir p apunta a la dirección 2002 y p+1 a la 2004 , p+2 a la 2006.

Observe que la suma o resta del entero determina el desplazamiento a partir del puntero pero no se modifica el valor del puntero.

Para modificar el valor del puntero , es decir mover el puntero se debe realizar la asignación , es decir:

p=p+1;

p=p+2;

en ambos casos la operatoria es la misma , a la dirección del puntero se le suma un número y luego se le asigna al puntero.

Otra forma de escribir esto mismo es p++ o p-- es decir , se utilizan los operadores de incremento o decremento.

Tenga en cuenta que si bien se accede a la misma posición de memoria en un caso el puntero esta fijo y se le suma un desplazamiento y en el otro el puntero se desplaza.

Si definimos un puntero a caracter el esquema de la memoria es el siguiente

|  |  |  |
| --- | --- | --- |
| 2000h |  |  |
| 2001h |  | p-1 |
| 2002h | **5** | p |
| 2003h |  | p+1 |
| 2004h |  | p+2 |
| 2005h |  |  |

En este caso cuando sumamos un entero no se desplaza a las mismas posiciones que en el caso anterior , eso es por que el puntero esta definido para otro tipo de dato. En realidad no nos importa tanto sobre que dirección queda , solamente debemos tener en claro que al sumar N a un puntero nos estaremos desplazando N tipos de dato para atras o adelante a partir de la posición del puntero.

Sea pc un puntero a caracter

pc+1 se apunta 1 caracter despues de la posición actual del puntero

pc+2 se apunta 2 caracteres despues de la posición actual del puntero

pc+3 se apunta 3 caracteres despues de la posición actual del puntero

Sea pi un puntero a entero

pi+1 se apunta 1 entero despues de la posición actual del puntero

pi+2 se apunta 2 enteros despues de la posición actual del puntero

pi+3 se apunta 3 enteros despues de la posición actual del puntero

**Notas sobre aritmética de punteros**

* No pueden sumarse números reales a un puntero
* Si bien pueden sumarse 2 punteros el resultado no tiene sentido
* Tampoco tiene sentido el resultado de multiplicar o dividir 2 punteros

## Vectores y punteros

Cuando en un programa se define un vector se esta reservando una cantidad de memoria igual a la cantidad de elementos del vector. Si definimos por ejemplo int vec[4] la memoria se verá de la siguiente manera

|  |  |  |
| --- | --- | --- |
| 2000h |  |  |
| 2001h |  |  |
| 2002h |  | vec[0] |
| 2003h |  |  |
| 2004h |  | vec[1] |
| 2005h |  |  |
| 2006h |  | vec[2] |
| 2007h |  |  |
| 2008h |  | vec[3] |
| 2009h |  |  |
| 200Ah |  |  |

Donde la región pintada es la que ocupa el vector en memoria , del lado derecho tenemos la denominación de cada elemento del vector y del lado izquierdo estan las direcciones de memoria.

Supongamos que se cargan los números 15,16,17 y 18 en cada elemento del vector , con lo cuál el diagrama de la memoria queda:

|  |  |  |
| --- | --- | --- |
| 2000h |  |  |
| 2001h |  |  |
| 2002h | **15** | vec[0] |
| 2003h |  |  |
| 2004h | **16** | vec[1] |
| 2005h |  |  |
| 2006h | **17** | vec[2] |
| 2007h |  |  |
| 2008h | **18** | vec[3] |
| 2009h |  |  |
| 200Ah |  |  |

Si ahora ponemos un puntero apuntando al comienzo del vector se deberá escribir

p= & vec[0]

Primer elemento del vector

y se lee “ al puntero p se le asigna la dirección de memoria del primer elemento del vector”. Si a partir del puntero vamos sumando enteros ( 0, 1, 2, 3) obtenemos direcciones desplazadas a partir del comienzo del vector. De esta forma el diagrama de la memoria queda así

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2000h |  |  |
|  | 2001h |  |  |
| p+0 | 2002h | **15** | vec[0] |
|  | 2003h |  |  |
| p+1 | 2004h | **16** | vec[1] |
|  | 2005h |  |  |
| p+2 | 2006h | **17** | vec[2] |
|  | 2007h |  |  |
| p+3 | 2008h | **18** | vec[3] |
|  | 2009h |  |  |
|  | 200Ah |  |  |

Si se observa detenidamente el gráfico podemos sacar la siguiente conclusión

La dirección de memoria del primer elemento del vector coincide con p+0 p+0 &vec[0]

La dirección de memoria del segundo elemento del vector coincide con p+1 p+1 &vec[1]

La dirección de memoria del tercer elemento del vector coincide con p+2 p+2 &vec[2]

La dirección de memoria del cuarto elemento del vector coincide con p+3 p+3 &vec[3]

En forma genérica se puede decir

**p+i &vec [i]**

Análogamente se puede decir que el valor de un elemento del vector es igual al contenido de lo apuntado por el puntero realizando el desplazamiento correspondiente. Así se puede expresar en forma genérica la siguiente relación

\*(p+i) vec [i]

Estas dos relaciones son importantes ya que usted puede realizar el pasaje de cualquier programa que use índices a uno que use punteros.

Es importante tener en cuenta que por definición **el nombre de un vector es un puntero al comienzo del mismo.** De acuerdo a este concepto se puede decir que

**p=&vec[0] es lo mismo que p=vec**

Veamos un ejemplo simple del uso de punteros sobre un vector , primero usando índices y luego haciendo el reemplazo para usar punteros.

**Ejemplo:** Cargar un vector de 20 enteros y luego mostrarlos por pantalla.

**Usando índices**

#include <stdio.h>

#define MAX 20

void main(void)

{

int vec[MAX],i;

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

{

printf("Ingrese un numero: ");

scanf("%d",&vec[i]);

}

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

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

}

**Usando punteros PRIMER FORMA**

#include <stdio.h>

#define MAX 20

void main(void)

{

int vec[MAX],i.\*p;

Se inicializa el puntero al comienzo del vector

p=&vec[0];

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

{

printf("Ingrese un numero: ");

scanf("%d",p+i);

}

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

printf("%d ",\*(p+i));

}

En el ejercicio que utiliza punteros se hace el reemplazo directo de las relaciones que se dieron anteriormente.

Otra forma de realizar el mismo ejercicio usando punteros es la siguiente

**Usando punteros SEGUNDA FORMA**

#include <stdio.h>

#define MAX 20

void main(void)

{

int vec[MAX],i.\*p;

for(p=vec;p<vec+MAX;p++)

{

printf("Ingrese un numero: ");

scanf("%d",p);

}

for(p=vec;p<vec+MAX;p++)

printf("%d ",\*p);

}

El efecto tanto en la primera forma como en la segunda es exactamente el mismo , la única diferencia es que en la primera forma el puntero se inicializa al comienzo y se le suman enteros para alcanzar las direcciones de memoria y en la segunda el puntero se va desplazando por todo el vector quedando al final del mismo cuando el for finaliza.

En realidad el verdadero uso de los punteros para este tipo de aplicaciones se da cuando se utiliza asignación dinámica de memoria que se verá un poco mas adelante en este mismo capítulo.

Veamos un poco que es lo que pasa con las matrices. Para ello es necesario conocer perfectamente como se ve la matriz en la memoria. Consideremos una matriz de enteros que tenga 3 filas y 4 columnas cuya representación será

int mat [ FIL ][ COL ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| 2 |  |  |  |  |

Los números que figuran en el dibujo corresponden a los índices de filas y columnas.

La representación en memoria de la matriz se verá así

|  |  |
| --- | --- |
|  | Elemento |
|  |  |
|  | mat[0][0] |
| 1° Fila | mat[0][1] |
|  | mat[0][2] |
|  | mat[0][3] |
| 2° Fila | mat[1][0] |
|  | mat[1][1] |
|  | mat[1][2] |
|  | mat[1][3] |
| 3° Fila | mat[2][0] |
|  | mat[2][1] |
|  | mat[2][2] |
|  | mat[2][3] |
|  |  |

En el diagrama de la memoria se ve que cada fila esta a continuación de la anterior , es como un vector formado por las filas de la matriz , nuestra matriz tiene 3 filas por 4 columnas (12 elementos) y en la memoria tendremos 12 elementos consecutivos.

Necesitamos determinar ahora la relación entre filas y columnas respecto a las posiciones de memoria determinada por un puntero.

Veamos la relación entre el primer elemento de la matriz y el primer elemento de cada fila analizando cuantos elementos hay que desplazarse para llegar desde un lugar al otro

Primer elemento de la primera fila desplazamiento 0

Primer elemento de la segunda fila desplazamiento 4

Primer elemento de la tercer fila desplazamiento 8

Observe que el desplazamiento respecto del primer elemento es un múltiplo de 4 , que casualmente es la cantidad de columnas de la matriz. Entonces si colocamos un puntero al comienzo de la matriz y le sumamos un entero a modo de desplazamiento podemos escribir lo siguiente para obtener la dirección de memoria del primer elemento de cada fila

p + índice de la fila \* cantidad de columnas

Como normalmente en un programa el que indica el número de fila es la variable i podemos escribir

p + i \* COL

Siendo COL una constante cuyo valor es la cantidad de columnas. Pero por ahora solo nos direccionamos al comienzo de cada fila y lo que se pretende es acceder a cualquier posición. Entonces podemos decir que si a partir del comienzo de una fila nos desplazamos la cantidad de columnas deseadas llegamos al lugar buscado , siendo la expresión que lo resume la siguiente

p + i \* COL + índice de la columna

Siendo normalmente la variable j la encargada de dar el índice de la columna nos queda la expresión

p + i \* COL + j

y las relaciones entre los elementos de la matriz y sus direcciones la siguiente

\*(p+i\*COL +j) mat[i][j]

p+i \*COL +j &mat [i][j]

## Punteros a cadenas de caracteres

Un puntero a cadena de caracter (string) no es mas ni menos que un puntero a caracter. Por lo tanto la forma de definir el puntero a cadena de caracteres es la que se vió en el estudio de punteros. Recuerde que debe realizar la inicialización del puntero al comienzo de la cadena. Por ejemplo la siguiente línea declara un puntero a caracter y un vector para guardar la cadena de caracteres

char vec[MAX] , \*p;

Para inicializar el puntero al comienzo de la cadena solo se escribe p=vec.

Tenga en cuenta que cuando se trabaja con cadenas de caracteres no interesan las columnas de la matriz o los elementos del vector , solo tiene importancia el comienzo de cada fila en la matriz ó el comienzo del vector .

De acuerdo a esto cuando se trabaja con una matriz y con un puntero para cargarla , se deberá tener en consideración que el puntero debe quedar siempre apuntando al comienzo de cada fila. Para poder hacer eso una de las formas es colocar el puntero al comienzo de la matriz y sumarle un desplazamiento que sea la cantidad de columnas de la matriz por la fila en la que se debe guardar la cadena. La expresión que resume lo que se comentó es la siguiente

p+ i \* COL

donde

p es el puntero al comienzo de la matriz

i es el índice que corresponde a l afila en la que se desea cargar la cadena

COL es la cantidad de columnas que tiene la matriz

Esta operación es necesaria ya que el puntero p es un puntero a caracter por lo tanto si lo muevo una posición va a apuntar al próximo caracter (por la aritmética de punteros) y eso no es lo que se necesita. Es por tal motivo que al puntero hay que desplazarlo COL posiciones de memoria.

Un ejemplo simple de la carga de una matriz utilizando punteros es el siguiente.

#include <stdio.h>

#include <string.h>

#include <conio.h>

#define FIL 2

#define COL 20

void main(void)

{

char nom[FIL][COL],\*p;

int i;

p=&nom[0][0];

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

{

printf("Ingrese nombre");

gets(p+i\*COL);

}

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

printf("\n%s",p+i\*COL);

}

**Nota**: Todas las funciones que trabajan con cadenas de caracteres reciben punteros como parámetros

## Punteros a estructura

Como una estructura es una variable se la puede apuntar con un puntero. De acuerdo a la definición de puntero deberá guardar la dirección de memoria de la estructura , si pensamos que la estructura es un grupo de variables distintas , el puntero tendrá la dirección de comienzo del grupo de variables.

Si se piensa en un vector de estructuras al cuál se le pone un puntero al comienzo del mismo , en el momento de recorrerlo con el puntero al incrementar el puntero se está apuntando a la próxima estructura. Lo recien dicho cumple con la aritmética de punteros explicada en el capítulo correspondiente.

Todas las propiedades de los punteros se cumplen al utilizar estructuras.

**Ejemplo**

#include <stdio.h>

struct alumno {

char nombre[20];

int nota;

};

void main(void)

{

struct alumno alu1 , \*pstr;

pstr=&alu2;

printf("Ingrese nombre");

gets(alu1.nombre);

printf("Ingrese nota");

scanf("%d",&alu1.nota);

\*pstr=alu1;

}

### Acceso a los campos – Operador flecha

Cuando se usan los punteros a estructura la forma de acceder a un campo es usando el operador FLECHA (->) el cuál se forma escribiendo un menos y a continuación un >.

**Ejemplo**

Cargar una estructura usando punteros.

#include <stdio.h>

#include <string.h>

struct alumno {

char nombre[20];

int nota;

};

void main(void)

{

struct alumno pers,\*pstr;

pstr=&pers;

printf("Ingrese nombre");

gets(pstr->nombre);

printf("Ingrese nota");

scanf("%d",&pstr->nota);

}

Observe que si usa un puntero a estructura , el uso del operador flecha da acceso al campo con lo cual en el útimo scanf a pesar de usar un puntero necesita usar el & ya que lo que cuenta es la variable nota.

### Vector de punteros

Un vector de punteros es simplemente un vector que dentro de cada uno de sus elementos existe un puntero.

La forma de definir un vector de punteros es:

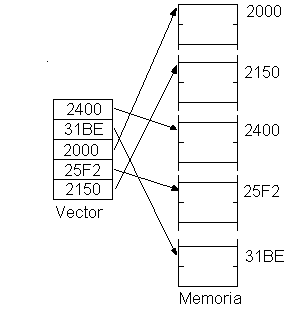
tipo \* nombre[MAX]

int \*vec [MAX]

El vector tiene posiciones de memoria (punteros) de algunas variables , por ejemplo el elemento vec[ i ] contiene la dirección de memoria de una variable y \*vec [ i ] será un entero.

Al igual que se inicializaba un puntero , se debe inicializar el vector de punteros para que cada posición del vector apunte a algún lugar determinado.

Veamos el siguiente dibujo que puede ilustrar mejor la situación.



A la derecha del dibujo tenemos el vector vec que es un vector de punteros , por lo tanto cada elemento del vector es una dirección de memoria (puntero). Inicialmente como pasa en cualquier variable tiene un valor cualquiera cuando arranca el programa , en el caso del vector de punteros pasa lo mismo , por lo cuál los punteros están apuntando a cualquier posición.

Normalmente se asocia un vector de punteros a otro de estructuras , es decir que el vector de punteros tiene las direcciones de cada uno de los elementos del vector de estructuras.

Veamos una utilidad del vector de punteros.

Suponga que debe realizar un programa de agenda que cargue los datos de 100 personas y luego tenga que ordenar los datos alfabéticamente. En el proceso de ordenamiento lo que se hace es copiar una estructura entera a un auxiliar , luego copiar el elemento j al alemento I (una estructura) y finalmente el auxiliar al elemento j , o sea está realizando 3 copias de estructuras lo cual implica un movimiento importante de datos.

Ahora si usted tiene un vector de punteros , el movimiento de datos se realiza en el vector de punteros , con lo cuál la copia se reduce notablemente (la cantidad de bytes a copiar se reduce a 2 bytes que es el tamaño de un puntero) y por otra parte no se modifica el vector de estructuras.

Cuando se quiere mostrar los datos ordenados simplemente se recorre el vector de estructuras. Veamos un ejemplo.

**Ejemplo**

Cargar nombre , legajo y sueldo de 20 personas. Mostrar los datos ordenados por nombre.

#include <stdio.h>

#include <string.h>

#define MAX 20

struct gente { char nom[20];

int leg;

float sueldo;

};

void main(void)

{

struct gente persona[MAX],\*vec[MAX],\*aux;

int i,j;

float x;

**Carga de el vector de estructuras**

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

{

printf("\nIngrese nombre ");

gets(persona[i].nom);

printf("\nIngrese Legajo ");

scanf("%d",&persona[i].leg);

fflush(stdin);

printf("\nIngrese Sueldo ");

scanf("%f",&x);

fflush(stdin);

persona[i].sueldo=x;

}

**Inicialización del vector de punteros**

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

vec[i]=&persona[i];

for(i=0;i<MAX-1;i++)

for(j=i+1;j<MAX;j++)

if((strcmp(vec[i]->nom,vec[j]->nom))<0)

{

aux=vec[i];

vec[i]=vec[j];

vec[j]=aux;

}

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

{

printf("\nNombre: %s",vec[i]->nom);

printf("\tLegajo: %d",vec[i]->leg);

printf("\tSueldo: %.2f",vec[i]->sueldo);

}

}

En el código escrito primero se carga el vector de estructuras y luego se inicializa el vector de punteros con las direcciones de cada uno de los elementos del vector de estructuras. La inicialización del vector de punteros se puede hacer en el mismo for en el que se cargan los datos.

Observe que cuando se realiza el ordenamiento se ordena el vector de punteros , con lo cuál el vector de estructuras queda intacto. Aparte de el hecho de no modificar el vector de estructuras se puede decir que en el proceso de ordenar al modificar el vector de punteros siempre se mueve la misma cantidad de datos , es decir que no interesa la cantidad de campos que posea la estructura. Entonces si una estructura contiene 20 campos y otra solo 2 , en las asignaciones dentro del algoritmo que ordena solo se copiaran datos de 2 o 4 bytes dependiendo de que los punteros sean **near** o **far** respectivamente.

## Asignación Dinámica de memoria

Cada vez que se declara una variable simple, un vector o una matriz estamos resevando memoria de la computadora para el momento de la ejecución. El hecho de reservar memoria nos sirve para solucionar determinados problemas pero existen situaciones en las que no se conoce la cantidad de memoria a utilizar por lo tanto no se puede reservar.

Para esos casos se solicita la memoria en forma dinámica , es decir , se pide memoria de acuerdo a la necesitad pidiendo al sistema operativo la cantidad de bytes necesarios y devolviendola cuando se deja de usar.

En C existen 2 funciones creadas para tal fin, dichas funciones son ***malloc*** y ***free*** . En C++ existen los operadores ***new*** y ***delete*** que realizan operaciones análogas a sus correspondientes de C***.***

Por una cuestión de conveniencia y para compatibilidad con Programación avanzada usaremos los operadores de C++.

El operador ***new*** reserva memoria dinámicamente , la forma genérica es la siguiente

variable\_puntero= ***new***  tipo\_de\_dato [cantidad ]

El operador new reserva cantidad de elementos de tipo\_de\_dato y devuelve el puntero al comienzo de la zona de memoria , en caso de encontrarse con algún problema el operador new devuelve NULL. Si deseamos usar un vector dinámico de 10 elementos el código será

void main ( void )

{

int \*pint;

pint=new int [10];

……………

}

pint apunta al comienzo del vector de 10 enteros. Antes de terminar el programa o cuando ya no se use mas ese vector se debe liberar la memoria reservada en forma dinámica y para ello se usará el operador delete.

void main ( void )

{

int \*pint;

pint=new int [10];

……………

delete pint;

}

Un ejemplo de un programa que genere un vector dinámico lo cargue y lo muestre puede ser el siguiente

#include <stdio.h>

void main(void)

{

int i,\*p , cant;

printf(“Ingrese la cantidad de elementos a cargar”);

scanf(“%d”,&cant);

p= new int[cant];

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

{

printf("Ingrese un numero: ");

scanf("%d",p+i);

}

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

printf("%d ",\*(p+i));

delete p;

}

# Funciones

Todos los programas desarrollados hasta el momento estaban formados por un cuerpo principal , llamado main, en el que se desarrollaba la totalidad del código. Dado que los programas eran relativamente sencillos no existía inconveniente en desarrollarlos de esa forma.

A medida que los programas crecen en extensión y complejidad la resolución se torna mas complicada y su depuración y modificaciones futuras resultan casi imposibles.

Para resolver este tipo de problemas lo que se hace es dividir el programa en módulos mas pequeños que cumplan una tarea simple y bien acotada. Con esto se logra:

1. El programa es mas simple de comprender ya que cada módulo se dedica a realizar una tarea en particular.
2. La depuración queda acotada a cada módulo.
3. Las modificaciones al programa se reducen a modificar determinados módulos.
4. Cuando cada módulo esta bien probado se lo puede usar las veces que sea necesario sin volver a revisarlo.
5. Se obtiene una independencia del código en cada módulo.

Lo que llamamos en forma genérica como “modulo” en el lenguaje C son las funciones.

Una función es un programa que realiza una tarea determinada y bien acotada a la cual le pasamos datos y nos devuelve datos. Este programa se ejecuta cuando se lo llama (llamada a la función). Como ejemplo de funciones conocidas son:

getche

scanf

gets

printf

strcpy……etc.

Estas funciones usted las usa continuamente cuando escribe su nombre ,le pasa datos y la función retorna un resultado.

Por ejemplo la función getche. A esta función usted no le pasa nada y ella le devuelve el caracter que leyó del teclado.

Cuando la llama se escribe

rta=getche( )

El código que se ejecuta cuando se llama a la función getche está dentro de las librerías que vienen con el compilador , usted no conoce ese código (independencia del código) pero sabe que es lo que hace y que lo hace bien.

Lo que se trata en este capítulo es como trabajan las funciones y cual es el procedimiento a seguir para escribir una función.

## Declaración de una función

La forma general de declarar una función es la siguiente

***Tipo\_devuelto*** Nombre\_de\_funcion ( ***tipo*** variable\_1, ***tipo*** variable\_2 , ...... , ***tipo*** variable\_N)

Tipo\_devuelto es el tipo de dato que devuelve la función luego de terminada su ejecución

Nombre\_de\_función es el nombre de la función

tipo variable\_1 es el tipo y nombre de la variable que se le pasa a la función.

Observe que según la declaración de la función se devuelve solamente un valor.

Cuando trabajamos con funciones tenemos que tener en cuenta que la forma en la que escribimos un programa se modifica. Los programas desarrollados hasta el momento contaban solamente con la función main y tenían el siguiente formato

# include ........ Lugar donde se incluyen todos los archivos de cabecera necesarios

........................

# define........... Lugar donde se definen las constantes y macros

........................

void main (void) comienza el programa principal

{

................... desarrollo del programa

...................

}

Al incluir el uso de las funciones , la forma general de escribir un programa es la siguiente

# include ........ Lugar donde se incluyen todos los archivos de cabecera necesarios

........................

# define........... Lugar donde se definen las constantes y macros

........................

**declaración de las funciones aquí se escriben todos los prototipos de las funciones a desarrollar**

**........................**

void main (void) comienza el programa principal

{

................... desarrollo del programa

**Llamada a la función** desde aquí se puede realizar la llamada a las funciones

...................

}

**desarrollo de las funciones** terminado el programa principal comienza el desarrollo de las funciones

Observe las 3 instancias donde se hace mención a las funciones

1. **declaración de las funciones**: En este lugar se declaran todas aquellas funciones propias que se van a utilizar durante el programa. La forma de declararlas es como se explicó al comienzo , teniendo en cuenta que cada declaración de función termina con punto y coma (;) .A las declaraciones que se realizan en este punto se las llama prototipo de la función. Los prototipos son necesarios para que el compilador verifique que sean coherentes los tipos de datos declarados en el prototipo contra los que realmente se usan en la llamada a la función.
2. **Llamada a la función**: Cuando se llama a la función es para usarla, es decir para que realice su trabajo. Desde cualquier parte de main o desde otra función se puede hacer la llamada. En el momento en que se produce la llamada a la función se produce un salto en el flujo del programa. En ese momento se pasa a ejecutar el código de la función que va a terminar de ejecutarse al encontrar la sentencia ***return*** o llegar a la llave que cierra la función. Cuando la función finaliza se sigue ejecutando el código de la función que produjo la llamada.
3. **Desarrollo de las funciones**: Esta es la parte en la cuál se escribe el código de la función (tal cual lo hacía con main).

Un ejemplo sencillo para ver cada una de las partes puede ser el siguiente:

Ingresar 2 números enteros y por medio de una función calcular la suma de los mismos

#include <stdio.h>

Declaración de la función.

Observe que el prototipo de la función termina con ;

**int suma(int a, int b);**

void main(void)

{

int x,y,z;

printf("ingrese numero a sumar: ");

scanf("%d",&x);

printf("ingrese numero a sumar: ");

scanf("%d",&y);

Llamada a la función

**z=suma(x,y);**

printf("La suma es %d",z);

}

**int suma(int a, int b)**

**{**

Desarrollo de la función

**int total;**

**total=a+b;**

**return total;**

**}**

En el ejemplo de arriba y en negrita aparecen los lugares donde interviene la función.

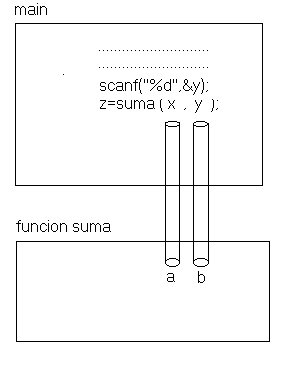
Observe que el prototipo termina con ;. La función se llama suma, se le pasan 2 variables enteras como parámetros y devuelve un entero. A las variables que se le pasan a la función se la llaman ***parámetros formales.***

Cuando se llama a la función se le pasan las variables con las que tiene que trabajar , en este momento a las variables se las llama ***parámetros actuales.***

En el desarrollo de la función se vuelve a escribir el prototipo pero ahora no se coloca el ; al final de la línea. En este lugar se escribe el código de la función tal cual se escribe en el main. Observe que la última línea contiene una instrucción ***return*** que lo que hace es terminar la función y devolver el valor que se encuentra a continuación, en este caso devuelve el valor que tiene la variable total.

**NOTA: Las variables definidas como parámetros formales de la función son variables locales de la misma.**

Gráficamente podemos imaginarnos a la función trabajando con el main de acuerdo al siguiente esquema



## Variables locales y globales

Dependiendo en el lugar donde se define una variable serán las características de visibilidad y ámbito de validez que ellas tienen.

Todas las variables que se encuentren definidas dentro de las llaves de una función (recuerde que main también es una función) tienen validez dentro de dicha función y se llaman ***variables locales***.

Veamos el ejemplo anterior donde usamos la función suma. Las variables x , y , z que están definidas dentro de main , son locales de main y valen solamente mientras se esta dentro de main, es decir que cuando se esta ejecutando la función suma las variables mencionadas no existen y no es posible utilizarlas ya que en la función suma no existen.

Dentro de la función suma las variables que existen son a , b , total que son variables locales de suma y existen solamente mientras se esta ejecutando la función suma.

Ahora si una variable puede ser usada desde cualquier función y durante el transcurso de todo el programa, esa es una variable global. Las variables globales de definen fuera de cualquier función , normalmente debajo de los ***include*** que se colocan al comienzo del programa.

#include <stdio.h>

**int var;**

void carga(void);

void main(void)

{

int x,y,z;

var=5;

carga();

printf(“%d”,var);

}

void carga(void)

{

var=3;

}

La variable var es una variable global. Esta variable puede ser usada desde el main o desde la función carga.

Cuando el programa comienza a correr se le asigna a la variable var el valor 5, luego se llama a la función y esta le asigna el valor 3. Finalmente cuando se muestra el valor de la variable var aparecerá un 3 que es el último valor asignado. Tenga en cuenta que al tener una variable definida en forma global se puede acceder a ella desde cualquier parte del programa ya sea para asignarle un nuevo valor o para leerla.

Veamos la siguiente modificación al programa anterior.

#include <stdio.h>

**int var;**

void carga(void);

void main(void)

{

int x,y,z;

var=5;

carga();

printf(“%d”,var);

}

void carga(void)

{

int var;

var=3;

}

Cuál supone que será el resultado mostrado en pantalla?

El resultado es 5.

Veamos que es lo que pasa. Cuando comienza el programa se le asigna a la variable var el valor 5 , se llama a la función y esta le asigna un 3 a var para finalmente mostrar el valor 5. Si bien parece que hablamos siempre de la misma variable no es asi ya que a pesar que tienen el mismo nombre son distintas.

Dentro de main cuando se ejecuta var=5 se esta cargando la variable global ya que el compilador se fija primero si esta definida como local y después busca la global. Al no encontrar la local se la asigna a la global.

Dentro de la función carga esta definida una variable local con el nombre var, por lo tanto se hace la asignación de dicha variable que cuando termina la función y por ser una variable local ya no tiene mas validez.

Entonces de acuerdo a esto podemos establecer algunas reglas

1. La variable local tiene validez solo dentro de la función en la que fue definida, fuera de ella no existe.
2. Si existe una variable global y otra local con el mismo nombre tiene validez la variable local dentro de esa función
3. Si en la función a tenemos la variable nombre y en la función b existe la variable nombre no existe problema de ningún tipo ya que ambas son variables locales de funciones distintas (aunque tengan el mismo nombre)
4. Es conveniente usar variables con distinto nombre para evitar confusiones.

## Diagrama de flujo de un programa que usa funciones

Vamos a analizar como es la ejecución de un programa que utiliza funciones como por ejemplo el de la función suma.

La variable x se copia en a. La variable y se copia en b

función main

……………

……………

……………

z=suma(x,y);

……………

……………

……………

total=a+b;

return total

Se retorna el valor de total y se le asigna a z

función suma

Un programa siempre comienza por ***main*** , se generan las variables y se empiezan a ejecutar las líneas del programa.

Cuando se llega al llamado de la función se produce un salto en el flujo del programa para ejecutar el código de la función. Antes de comenzar a ejecutarse el código se deben inicializar las variables de la función suma , es decir se copia el valor de la variable x en la variable a y el de la variable y en b (tenga en cuenta que este proceso se realiza automáticamente, usted no debe agregar nada en el programa para hacerlo). Hecha la inicialización se ejecuta el programa y al aparecer la instrucción ***return*** se devuelve el valor de total que se asigna a la variable z de main. A partir de alli se sigue la ejecución normal de main.

## Pasaje de parámetros por valor y por referencia

Se llama pasaje por valor cuando a la función se le pasa como parámetro actual el valor de la variable.

Se llama pasaje por referencia cuando a la función se le pasa como parámetro actual la dirección de memoria de una variable. Tenga en cuenta que en este caso la variable que recibe deber ser un puntero , ya qu elo que esta pasando es una dirección de memoria.

Veamos con un ejemplo la diferencia entre ambos casos.

### Pasaje por valor

#include <stdio.h>

void muestra(int x,int y);

void main(void)

{

int x,y;

printf("Ingrese un numero entero");

scanf("%d",&x);

printf("Ingrese un numero entero");

scanf("%d",&y);

muestra(x,y);

printf("\n-----valores dentro de main----");

printf("\nx vale %d \ny vale %d",x,y);

}

void muestra(int x,int y)

{

x=y;

printf("\n-----valores dentro de la funcion----");

printf("\nx vale %d \ny vale %d",x,y);

}

Cuando se ejecuta la función muestra , debido a la asignación que se encuentra en la primera línea , las variables x e y tienen el mismo valor. Por lo tanto cuando se ejecute el printf se muestra el valor de las 2 variables (que son variables locales de la función). Ahora cuando se ejecute el printf del main aparecen los valores que se cargaron en el primermomento sin ninguna modificación.

Esto ocurre por que aunque las variables tengan el mismo nombre , son distintas ya que son locales de cada función y como sabemos tienen validez solamente dentro de la función en la cual estan definidas.

### Pasaje por referencia

#include <stdio.h>

void muestra(int \*x,int \*y);

void main(void)

{

int x,y;

printf("Ingrese un numero entero");

scanf("%d",&x);

printf("Ingrese un numero entero");

scanf("%d",&y);

muestra(&x,&y);

printf("\n-----valores dentro de main----");

printf("\nx vale %d \ny vale %d",x,y);

}

void muestra(int \*x,int \*y)

{

\*x=\*y;

printf("\n-----valores dentro de la funcion----");

printf("\nx vale %d \ny vale %d",\*x,\*y);

}

Tal cual se dijo en la definición de pasaje por referencia , a la función se le pasa la dirección de una variable y por lo tanto está recibiendo un puntero. De acuerdo a esto modificamos la función del ejemplo anterior para que los argumentos concuerden.

Cuando se llama a la función se le pasan las direcciones de memoria de las variables , dentro de la función se igualan los contenidos de los punteros ( con esta línea lo que en realidad estamos haciendo es igualar las variables x e y de main).

Desde main cuando se ejecuta el printf se ve que las 2 variables tienen el mismo valor a pesar de que se hayan cargado originalmente valores distintos.

Tenga en cuenta que la modificación de las variables fue hecho desde la función por medio de punteros.

Como conclusión importante podemos decir que **por medio de un puntero se puede modificar el valor de una variable sin necesidad de conocer su nombre.**

Teniendo en claro este concepto vamos a solucionar el problema del pasaje de vectores a una función. Es necesario que sepa que en C no existe forma de pasarle los valores de un vector a una función.

Trabajemos con el siguiente problema

**Problema**

Realizar un programa que cargue un vector de 5 enteros y por medio de una función muestre los valores que se cargaron.

#### Primer planteo

#include <stdio.h>

#define MAX 5

void mostrar(void);

void main(void)

{

int i,vec[MAX];

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

{

printf("Ingrese un numero entero");

scanf("%d",&vec[i]);

}

mostrar();

}

void mostrar(void)

{

int i,vec[MAX];

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

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

}

Este planteo es incorrecto ya que tiene un error grave de concepto. Ejecute el programa y observe los valores del vector dentro de main y de la función mostrar. Son iguales?

#### Primer solución

El problema a resolver es el siguiente: Como se hace para acceder a una misma variable desde 2 o mas funciones distintas (en nuestro caso desde main y mostrar). Desde una de las funciones (main) se carga la variable y desde la otra (mostrar) se deben mostrar los datos que se cargaron.

Evidentemente y de acuerdo a la conclusión sacada del primer planteo , la variable debe ser la misma , entonces recordando las definiciones de variable local y global , concluimos que nuestro problema se resuelve colocando una variable global . Dicha variable global en nuestro caso será el vector. Recuerde que para que esto sea válido deben desaparecer los vectores locales definidos en el primer planteo dentro de cada una de las funciones.

#include <stdio.h>

#define MAX 5

int vec[MAX];

void mostrar(void);

void main(void)

{

int i;

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

{

printf("Ingrese un numero entero");

scanf("%d",&vec[i]);

}

mostrar();

}

void mostrar(void)

{

int i;

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

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

}

Cuando ejecutamos el programa veremos que los datos que se muestran en pantalla son correctos.

Ejecute paso a paso el programa e inspeccione la variable vec para comprobar que lo que está haciendo es correcto y que el vector cumple con la definición de variable global

#### Segunda solución

La segunda forma de resolver el problema es la mas usada en C y es utilizando punteros.

Sabemos que desde dentro de la función no existe forma de acceder al vector por su nombre , pero tranquilamente podemos conocer la dirección en la cual comienza el vector (puntero al comienzo del mismo) y a travez de ella recorrerlo ya sea para leerlo o modificar sus datos.

Es por este motivo que le pasamos un puntero a la función. En lugar de llamar al vector por su nombre lo ubicamos por su posición , **por lo tanto cada vez que necesite acceder a una variable vectorial desde una función voy a pasar como uno de los argumentos un puntero al elemento en cuestión (vector o matriz).**

#include <stdio.h>

#define MAX 5

void mostrar(int \*p);

void main(void)

{

int i,vec[MAX];

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

{

printf("Ingrese un numero entero");

scanf("%d",&vec[i]);

}

mostrar(vec);

}

void mostrar(int \*p)

{

int i;

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

printf("%d ",\*(p+i));

}

## Diseño de una función

El diseño de la función es una tarea que debe llevar un tiempo importante para su análisis ya que de acuerdo a la forma en la que se piensa nos puede servir para un solo programa o para muchas aplicaciones. Cada una de estas tres etapas se comentan cuando se desarrollan las funciones del ejemplo tipo que se encuentra en el apéndice.

Para el diseño de la función debemos tener en cuenta 3 puntos

* **Determinación de la tarea a realizar**: No tener en claro este punto significa no saber que hacer , hasta donde llegar y menos aún que codigo escribir.Se necesita tener muy en claro **que es lo que va a hacer la función.** Una vez que esta clara la tarea se debe acotar hasta donde se quiere llegar con los “chiches” que se le agregen , si no establecemos dicha cota resulta en que nunca se termina de escribir la función por que siempre se le quiere agregar algo mas lindo.

Por lo tanto es importante definir en este punto cual es la tarea BASICA de la función mas los “chiches” que se le deseen agregar.

* **Determinación de los parámetros formales**: Cuando ya tenemos en claro que es lo que hay que hacer se debe determinar como obtengo los datos para realizar la tarea que debo hacer. Por ejemplo si mi función lee algo del teclado no es necesario pasarle datos con lo cual el parámetro formal será **VOID.** En cambio si mi función se dedica a realizar alguna búsqueda podemos pensar 2 alternativas

1. El dato a buscar se lo paso a la función
2. El dato a buscar lo lee la función

De acuerdo al origen de los datos los parámetros formales cambian en cantidad y tipo. La respuesta a cuál de las opciones a elegir la da el sentido común y el análisis de cual de las 2 formas es mas genérica.

Entonces en este momento se decide de acuerdo al origen de los datos cuales van a ser los parámetros formales de la función.

* **Determinación del valor a retornar:** Finalmente resta definir cual será el tipo de dato que retorna la función. Al igual que en el punto anterior el valor a retornar va intimamente relacionado con la tarea que realice la función.

## Recursividad

Recursividad es el proceso de definir algo en términos de si mismo.

Básicamente un problema podrá ser resuelto en forma recursiva si la solución se puede expresar en términos de si misma , es decir , para obtener la solución deberá resolverse el mismo problema sobre un conjunto de datos de entrada menor.

Se dice que una función es recursiva cuando se llama a si misma. El lenguaje C soporta funciones recursivas.

El caso mas común para explicar recursividad es el cálculo del factorial. Si bien encaramos el cálculo del factorial utilizando funciones recursivas , es necesario hacer notar que se puede hacer el mismo programa sin usar recursividad.

Si recordamos la definición de factorial

1 si n=0

n!=

n \* (n-1)! si n>0

Si queremos calcular el factorial de 4 tendremos

4!= 4 . 3 . 2 . 1 = 24

Realicemos el programa que calcula el factorial

#include <stdio.h>

#include <conio.h>

int factorial(int );

void main(void)

{

int valor,result;

printf("\nIngrese numero:");

scanf("%d",&valor);

result=factorial(valor);

printf("\nEl factorial de %d es %d",valor,result);

}

int factorial(int n)

{

int resp;

if(n==1)

return 1;

resp=n\* factorial(n-1);

return (resp);

}

En la función se ve claramente que se vuelve a llamar a si misma.

Hay que tener muy presente que cada vez que la función se llama a si misma , se está utilizando memoria en la pila del programa , que en algún momento se debe liberar. Es decir que en algún momento la función debe comenzar a retornar , en nuestro ejemplo se comienza a retornar cuando el valor de n es 1.

Veamos gráficamente el ejemplo del factorial para entender el funcionamiento.

1

2

3

factorial

factorial

factorial

6

2

1

main

## Suponemos que se desea calcular el factorial de 3. En el dibujo se representa el desarrollo del programa. Junto con cada una de las flechas se encuentra una estrella que en su interior contiene el valor que se le pasa a la función o el valor de retorno.

La función factorial es llamada 3 veces con los valores 3 , 2 y 1 respectivamente. En la última llamada debido a la línea if (n==1) se produce el retorno de la función. Todas las funciones recursivas deben tener una condición que permita el comienzo del retorno , si esto no ocurre se puede producir un desbordamiento de la pila.

## Librería de funciones propias

Todas aquellas funciones que en algún momento se desarrollaron y creemos que estan lo suficientemente depuradas y son aptas para usarlas en distintos programas las podemos incluir en una librería propia.

La idea de las funciones , recordando la introducción a este tema , es poder usarlas en muchos programas independizándonos del código que ejecutan. De acuerdo a esto cada vez que se use una función no se debe volver a escribir el código , solamente se la llama.

Para poder aplicar en la práctica esta idea lo que se hace es generar un nuevo archivo en el compilador de C en el cuál solo se escribirán (copiar) todas aquellas funciones desarrolladas en programas anteriores , en este archivo no figura main.

Un ejemplo podria ser el siguiente

#include <stdio.h>

#include <conio.h>

#include <ctype.h>

#define MAX 20

#define HOTLETTER RED

#define COLORTEXTO BLACK

struct text\_info ti;

void barra(int pos,int x);

unsigned int verifica(void)

/\*----------------------------------------------------------------------\*/

/\* Funcion usada para verificar el ingreso de caracteres.Chequea para que

solo sean admitidas las letras S o N.

Se ingresa void y devuelve un entero sin signo

0 si se ingreso la N

1 si se ingreso la S

\*/

unsigned int verifica(void)

{

char letra;

printf("\nIngrese opcion...S/N?");

letra=toupper(getche());

while(!((letra=='S')||(letra=='N')))

{

printf("\nHa ingresado opci¢n no v lida...Reintente el ingreso(S/N)... ");

letra=toupper(getche());

}

return (letra=='S');

}

void barra(int pos,int x)

{

int i,attr,color=COLORBOX,dx;

union REGS keyb;

struct text\_info ti;

gettextinfo(&ti);

dx=ti.winright-ti.winleft;

if(x)

color=COLORBAR;

for(i=1;i<=dx;i++)

{

keyb.h.ah=0x08;

keyb.h.bh=0;

gotoxy(i,pos);

int86(0x10,&keyb,&keyb);

attr=keyb.h.ah;

keyb.h.ah=0x09;

keyb.x.cx=1;

keyb.h.bl=(attr & 0X0F)|(color<<4);

int86(0x10,&keyb,&keyb);

}

}

Observe que no aparece la función main. Es simplemente escribir un programa que no tiene main y en el cual se escriben todas las funciones que el programador ha desarrollado. Este archivo solamente se compila y se verifica que no contenga errores de compilación, se supone que no los debe tener ya que se han copiado las funciones de programas en los cuales estaban funcionando.

Para poder incluir estas funciones en un programa que usted desarrolle solo necesita agregar en el programa que usted esta escribiendo un define como el siguiente

#include <stdio.h>

#include “c:\prog\funciones.c”

.........................

.........................

Desarrollo del programa

.........................

.........................

Donde funciones.c es el archivo que contiene el código y prototipos de las funciones a usar.

## Argumentos de main

Si bien todos los programas escritos hasta el momento nos pueden llevar a pensar que la función ***main*** solo puede llevar como argumento ***void*** , es incorrecto. La función ***main*** está preparada para recibir 3 argumentos que luego veremos.

Cuando se ejecuta un programa desde la línea de comandos (en windows desde la ventana ejecutar y en DOS desde el prompt del sistema) normalmente se escribe el nombre del ejecutable y algunos casos modificadores permitidos por el programa, por ejemplo:

ping 10.1.15.27 ping es el nombre del programa y 10.1.15.27 es un parámetro que se le pasa al programa ping

dir /p dir es el programa y /p un parámetro

setup /quiet /standard.lst se ejecuta una instalación en el modo "silencioso" y no se pide intervención al usuario

Todos estos programas deben tener alguna forma de comunicación con el código para recibir los parámetros que se le pasan a través de la línea de comandos. En C se reciben por los argumentos del ***main*** .

Como se mencionó antes son 3 los argumentos que recibe main

void main ( int argc , char \*\*argv , char \*\*env )

A modo de ejemplo trabajaremos con un programa para haceer backup de un archivo, las opciones del programa backup.exe son:

backup.exe Archivo\_Origen Archivo\_Destino /Modificador

donde

Archivo\_Origen: Es el nombre y ruta del archivo origen

Archivo\_Destino: Es el nombre del archivo y la ruta en donde se guarda la copia

Modificador: /v Una vez realizada la copia se debe hacer la verificación.

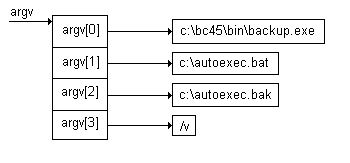
A continuación describimos los 3 argumentos de main

**argc**: es un entero que indica la cantidad de argumentos que se escribieeron en la línea de comandos incluido el nombre del programa ejecutable. En nuestro ejemplo tendremos

c:\ backup.exe c:\autoexec.bat c:\autoexec.bak argc=3

c:\ backup.exe c:\autoexec.bat c:\autoexec.bak /v argc=4

**argv**: Es un doble puntero a caracter. Apunta al comienzo de un vector de punteros donde cada posición del vector contiene la dirección de comienzo de cada una de las cadenas que se escribieron en la línea de comandos. Gráficamente sería



Para ver como se utilizan los argumentos del main en un programa , veamos un ejemplo.

**Ejemplo**: Realizar un programa para hacer backup de archivos. El programa deberá permitir manejar por la línea de comandos las siguientes opciones

/V Se debe verificar la copia

/? Muestra una ayuda acerca de los parámetros que se pueden utilizar.

La sintaxis de la llamada es

Back.exe Archivo\_Origen Archivo\_destino Opciones

#include <stdio.h>

#define VERDADERO 1

#define FALSO 0

void main(int argc, char \*\* argv)

{

FILE \*Origen, \*Copia;

int verificar=FALSO;

/\* Verifica que existan por lo menos 2 argumentos \*/

if(argc<3)

{

ayuda();

exit(1);

}

if((Origen=fopen(argv[1],"rb"))==NULL)

{

printf("No se puede abrir el archivo");

exit(2);

}

if((Copia=fopen(argv[2],"wb"))==NULL)

{

printf("No se puede abrir el archivo");

exit(2);

}

if(!strcmp(argv[3],"/V"))

verificar=VERDADERO;

if(!strcmp(argv[3],"/?"))

{

ayuda();

exit(0);

}

backup(Origen, Copia);

if(verificar)

VerifBack(Origen, Copia);

}

# Archivos

## Concepto de archivos

Todos los programas que se ejecutaron hasta el momento requerían de ingreso de datos para realizar determinadas tares y luego mostrar resultados. Tales programas tienen un gran defecto y es que los datos que se cargan desaparecen cuando el programa termina , por lo tanto cada vez que se corra el programa hay que cargar los datos nuevamente.

Esto ocurre debido a que los datos ingresados quedan guardados en memoria y al terminar el programa todos los espacios de memoria reservados son devueltos al sistema operativo perdiendo todos los datos que se habían cargado.

Para solucionar el problema debería existir una forma de almacenar los datos en forma permanente , por ejemplo en un medio magnético como el disco rígido o disquete. Los datos que se ingresan se deberán guardar en un archivo que se almacena en el disco o disquete de forma de disponer de la información en cualquier momento.

De tal forma si por ejemplo trabajamos con un programa de agenda en la cual se cargan los datos de personas , cada vez que se ejecute el programa se ira a leer el archivo que contiene los datos de las personas , en lugar de tener que ingresarlos cada vez .

En D.O.S los nombres de archivo tienen el siguiente formato

Nombre.extension

Donde :

Nombre es el nombre del archivo que puede tener un máximo de 8 caracteres

Extensión define un tipo de archivo en particular y puede tener un máximo de 3 caracteres

Algunas de las extensiones usadas son las siguientes

EXE programa ejecutable

COM programa ejecutable (el tamaño no exede 64KB)

BAT archivo de procesamiento por lotes

TXT archivo de texto

DOC archivo para ser abierto por el microsoft word

BIN archivo binario

DAT archivo de datos

INI archivo usado por windows para las condiciones iniciales de los programas

MP3 archivos de música

WAV archivos de música

Básicamente existen 2 tipos de archivos que son los de texto y binarios. Esta clasificación hace referencia a la forma en la cuál se interpretan los datos, se debe tener en claro que la única forma de guardar datos en el disco es en formato binario. Todos los archivos tienen al final una marca que indica el final del archivo. Dicha marca se conoce como EOF(End Of File).

A continuación se hace un breve comentario de los archivos binarios y de texto.

En particular nosotros trabajaremos con los archivos binarios.

### Archivos de texto

Un archivo de texto contiene toda su información guardada en binario pero se interpreta como texto. Absolutamente todo lo que contiene debe der interpretado como texto , ya que cuando se escribe el archivo , los datos son enviados como caracteres.

Cuando se dice que la información esta guardada en formato texto se esta haciendo referencia a como es la forma en la que hay que entender el dato , ya que sobre el disco los datos son guardados en forma de secuencia de “UNOS” y “CEROS” es decir según el sistema binario.

Supongamos que se desea guardar en un archivo de texto la siguiente cadena “Ana 12”. Lo que se guarda en el archivo son caracteres es decir la A , la n , la a , el espacio , el caracter 1 y el caracter 2 , entonces si analizamos el equivalente ASCII de los caracteres , en el archivo queda

Caracter A n a 1 2

ASCII 33 110 97 20 1 2

Binario 00100001 01011110 01100001 00100000 00110001 00110010

Lo que figura en la fila que dice binario es lo que queda en el disco rígido , observe que de acuerdo a como se tomen los datos se esta cambiando la información.

Si por ejemplo se quiere guardar en el archivo de texto el número 4567 se va a guardar como caracteres , es decir que en el disco queda:

Caracter 4 5 6 7

ASCII 52 53 54 55

Binario 00110100 00110101 00110110 00110111

Observe que para guardar un número como texto la cantidad de bytes a usar igual a la cantidad de dígitos , que en este caso en particular estamos usando 4 bytes . En el caso de usar un tipo de variable int solo se necesitan 2 bytes.

### Archivos binarios

En un archivo binario se guardan datos con distinto formato , es decir se pueden guardar caracteres mezclados con enteros y flotantes. Si bien todos los datos terminan escritos en el disco en sistema binario , la interpretación de los datos guardados cambia.

Si tomamos el último ejemplo dado para archivos de texto y queremos escribir en el archivo el número 4567 , en el programa va a estar definido como un entero y por lo tanto se escribirán 2 bytes al archivo. El número quedaría de la siguiente forma

Número 4567

Binario 00010001 11010111

Normalmente el hecho de trabajar con archivos binarios esta asociado con el uso de estructuras , dado que la forma mas simple de trabajar es cargar una estructura para luego escribirla en el archivo y de esa forma guardar los datos.

Si bien se puede trabajar con archivos binarios y no usar estructuras , el hecho de utilizarlas permite tener un programa mejor armado y mas consistente.

## Trabajo sobre archivos

En C , todas las operaciones que se realizan sobre archivos son hechas a travez de funciones.

Básicamente existen 2 categorías de funciones para trabajar con archivos y son las que usan “buffer” y las que acceden directamente al archivo.

Durante el transcurso del apunte se usarán solamente la primer categoría , o sea aquellas que usan “buffer”. El hecho de utilizar un buffer significa que no se tiene acceso directo al archivo y que cualquier operación que se desee realizar (lectura o escritura) va a ser hecha sobre el buffer. Cuando el buffer se llena o se vacía se actualizan los datos desde y hacia el archivo.

Algunas de las funciones usadas para trabajar con archivos son:

fopen

fclose

fcloseall

fread

fwrite

feof

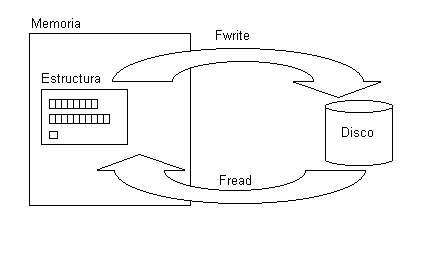
fseek

ftell

rewind

Se debe tener en cuenta que al usar las funciones sobre archivos estamos trabajando con un intermediario que accede al archivo.

Por lo tanto todos los datos que van al archivo (escritura ) y que vienen (lectura) se encuentran en memoria y fread y fwrite toman o dejan esos datos en el archivo. Gráficamente podemos verlo de la siguiente forma.



### La estructura FILE

Para trabajar con archivos en C , las funciones utilizan un puntero a la estructura FILE. Dicha estructura se encuentra definida en el archivo stdio.h y se detalla a continuación.

typedef struct {

int level; /\* fill/empty level of buffer \*/

unsigned flags; /\* File status flags \*/

char fd; /\* File descriptor \*/

unsigned char hold; /\* Ungetc char if no buffer \*/

int bsize; /\* Buffer size \*/

unsigned char \_FAR \*buffer; /\* Data transfer buffer \*/

unsigned char \_FAR \*curp; /\* Current active pointer \*/

unsigned istemp; /\* Temporary file indicator \*/

short token; /\* Used for validity checking \*/

} FILE;

Entonces para el manejo de archivos es indispensable definir un puntero a la estructura FILE , como por ejemplo:

FILE \*parch;

Donde parch es el puntero a la estructura FILE.

### Apertura y cierre de un archivo

**Apertura del archivo**

Cada vez que se necesite trabajar con un archivo , lo primero que se debe hacer es abrirlo. Si no se realiza esto no se puede leer ni escribir en el mismo. La función que me permite la apertura del archivo es ***fopen.***

El formato de la función ***fopen*** es el siguiente

*FILE* \* ***fopen*** (const char \**Nombre\_de\_archivo* , const char \**Modo*)

Donde:

*Nombre\_de\_archivo* Es una cadena de caracteres que representa el archivo , es decir se pone la ruta y el nombre del archivo

*Modo* Es una cadena de caracteres que determina el modo en el que será abierto el archivo.

Los modos en los que se puede abrir un archivo estan detallados en la siguiente tabla

|  |  |
| --- | --- |
| **Modo** | **Detalle** |
| r | Abre un archivo de texto para operaciones de lectura. |
| w | Abre un archivo de texto para operaciones de escritura |
| a | Abre un archivo de texto para añadir datos. |
| rb | Abre un archivo binario para operaciones de lectura. |
| wb | Abre un archivo binario para operaciones de escritura. |
| ab | Abre un archivo binario para añadir datos. |
| r+b | Abre un archivo binario para operaciones de lectura escritura. |
| w+b | Abre un archivo binario para operaciones de lectura escritura. |
| a+b | Abre un archivo binario para operaciones de lectura escritura. |
| r+ | Abre un archivo de texto para operaciones de lectura escritura. |
| w+ | Abre un archivo de texto para operaciones de lectura escritura |

**Notas sobre los modos de apertura del archivo**

Modo escritura (**w**)

Si se abre un archivo para operaciones de escritura ( w , wb , w+b , w+ ) y el archivo no existe se creará , pero si existe todos los datos del archivo serán borrados.

Si el archivo que se desea abrir tiene atributo de solo lectura , el disco esta lleno , el disco tiene un problema en la superficie , etc la función ***fopen*** devuelve error.

Cada vez que se abra un archivo en este modo , el indicador de posición se encuentra al comienzo del archivo.

Si se abre un archivo en el modo w o wb solamente se puede escribir. Si se intenta leer datos del archivo no va a aparecer ningún error , simplemente lo que aparezca como dato leido no va a reflejar la realidad.

Modo lectura ( **r** )

Si se abre un archivo para operaciones de lectura ( r , rb , r+b , r+ ) , si el archivo no existe la función ***fopen*** devuelve error.

Cada vez que se abra un archivo en este modo , el indicador de posición se encuentra al comienzo del archivo.

Si se abre un archivo en el modo r o rb solamente se pueden realizar lecturas. No tiene ningún efecto realizar operaciones de escritura sobre al archivo , es decir , por mas que se intente escribir sobre el archivo no se va a poder.

Modo append ( **a** )

Si se abre un archivo para agregar datos ( a , ab , a+b , a+ ) y el archivo no existe se creará , caso contrario el indicador de posición del archivo queda posicionado al final del mismo de forma de poder agregar datos.

Cada vez que se agregan datos se hace al final del archivo.

Con el modo append no tienen efecto las operaciones de desplazamiento a traves del archivo.

**Valor retornado:**

Si el archivo es abierto exitosamente , la función devuelve un puntero a la estructura FILE asociada al archivo. En caso de detectarse un error devuelve NULL

**Notas:**

* Se pueden abrir varios archivos al mismo tiempo siempre y cuando exista por lo menos un puntero a FILE para cada uno.
* La cantidad de archivos que se pueden abrir al mismo tiempo depende del sistema operativo y de la variable de sistema FILES que se encuentra en el archivo config.sys. Tenga en cuenta que cuando se ejecuta el programa hay abiertos 5 archivos que son propios de C.
* No se debe modificar el valor del puntero devuelto por ***fopen***.

**Ejemplo**

#include <stdio.h>

void main (void)

{

FILE \*parch;

if((parch=fopen(“c:\\banco.dat”,”rb”))==NULL)

{

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

fclose(parch);

}

En el ejemplo se utiliza un if para detectar la correcta apertura del archivo. Recuerde que cuando la función ***fopen*** retorna NULL significa que se ha fallado en abrir el archivo en el modo solicitado.

Para crear un archivo por primera vez se debe usar el modo w , pero primero nos debemos asegurar que el archivo no exista , ya que en ese caso el contenido del archivo se borra. Para contemplar esta situación se modifica levemente el código del ejemplo.

void main (void)

{

FILE \*parch;

if((parch=fopen(“c:\\banco.dat”,”rb”))==NULL) //Se abre en modo lectura

if((parch=fopen(“c:\\banco.dat”,”wb”))==NULL) //Si el modo anterior dio error el archivo

{ //no existe , por lo tanto se crea

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

fclose(parch);

}

La idea es abrir un archivo para leer , en el caso de que exista se trabaja normalmente , pero si no existe lo abre el segundo fopen.

De esta forma nos evitamos borrar un archivo que existe y tiene datos.

En el caso de querer ingresar el nombre del archivo por teclado , el programa se modifica de la siguiente manera

void main (void)

{

FILE \*parch;

char nombre[20];

printf(“\nIngrese el nombre y ruta del archivo que desea abrir: “);

gets(nombre);

if((parch=fopen( nombre ,”rb”))==NULL)

if((parch=fopen( nombre ,”wb”))==NULL)

{

printf(“\nEl archivo %s no puede ser abierto”,nombre);

exit (1);

}

fclose(parch);

}

**Cierre del archivo**

Todo archivo que se abre debe ser cerrado antes de terminar el programa. El terminar el programa sin cerrar el o los archivos puede causar pérdida de datos.

La función ***fclose*** es la que se encarga de cerrar un archivo. El formato de la función es el siguiente

*int* ***fclose*** ( *FILE* \*parch )

Donde parch es el puntero a la estructura FILE asociada con el archivo que se desea cerrar.

**Valor retornado**

Si el archivo es cerrado exitosamente se retorna un 0 , en caso contrario se devuelve –1;

**Ejemplo**

void main (void)

{

FILE \*parch;

if((parch=fopen(“c:\\banco.dat”,”rb”))==NULL) //Se abre en modo lectura

{

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

if((fclose(parch))= = -1) //Se cierra el archivo

printf(“\nNo se pudo cerrar el archivo”);

else

printf(“\nEl archivo se cerro exitosamente”);

}

En el ejemplo se chequea que se haya cerrado correctamente el archivo.

En el caso de abrir mas de un archivo , antes de terminar el programa se deben cerrar.

void main (void)

{

FILE \*parch1, \*parch2;

if((parch1=fopen(“c:\\banco.dat”,”rb”))==NULL) //Se abre banco.dat en modo lectura

{

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

if((parch2=fopen(“c:\\result.dat”,”wb”))==NULL) //Se abre result.dat en modo escritura

{

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

fclose(parch1); //Se cierra el archivo banco.dat

fclose(parch2); //Se cierra el archivo result.dat

}

Existe una función llamada ***fcloseall*** que cierra todos los archivos que se encuentran abiertos. El formato de la función es

*int* ***fcloseall*** ( void )

Si la operación es exitosa retorna la cantidad de archivos que se cerraron , en caso contrario devuelve –1.

Si modificamos el ejemplo anterior

void main (void)

{

FILE \*parch1, \*parch2;

if((parch1=fopen(“c:\\banco.dat”,”rb”))==NULL) //Se abre banco.dat en modo lectura

{

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

if((parch2=fopen(“c:\\result.dat”,”wb”))==NULL) //Se abre result.dat en modo escritura

{

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

fcloseall( ); //Cierra los 2 archivos

}

### Escritura de un archivo

Una vez que el archivo se encuentra abierto se puede empezar a trabajar para leer o escribir.

La función utilizada para realizar la escritura es ***fwrite*** .Esta función sirve para escribir archivos de texto o binarios.

El prototipo de la función es el siguiente:

int ***fwrite*** ( void \* *origen* , size\_t *tamaño* , size\_t *cantidad* , FILE \**arch*)

**Donde:**

*origen* Es un puntero al lugar desde donde se obtienen los datos para escribir en el archivo

*tamaño*  Es el tamaño en bytes del dato que se va a escribir

*cantidad*  Es la cantidad de datos de longitud *tamaño* que se van a escribir

*arch* Es el puntero a FILE asociado al archivo

Nota: size\_t es un **unsigned int** definido en stdio.h

**Valor retornado**: Devuelve el número de datos escritos (*cantidad*). Si el valor retornado es menor al que se indicó por medio de la variable *cantidad* , significa que hubo un error en la escritura.

La función ***fwrite*** toma *cantidad* de datos de longitud *tamaño* desde la dirección *origen* y los escribe en el archivo asociado al puntero *arch* comenzando desde la posición actual del indicador de posición del archivo. Una vez que se completó la operación de escritura el indicador de posición es actualizado (queda apuntando al lugar donde se puede escribir el próximo dato).

Ejemplo:

Si queremos escribir en un archivo de texto

void main (void)

{

FILE \*parch;

char texto[ ]=”Prueba de escritura”;

int cant , longi;

if((parch=fopen(“c:\\prueba.txt”,”w”))==NULL) //Se abre en modo escritura

{

printf(“\nEl archivo no puede ser abierto”);

exit (1);

}

longi=strlen (texto );

cant=fwrite ( texto , sizeof ( char ) , longi , parch ); //Se escribe al archivo

if (cant<longi)

printf(“\nError al escribir el archivo”);

else

printf(“\nSe escribieron %d caracteres”, cant);

fclose(parch);

}

Si se trata de un archivo binario el programa será el siguiente

#include <stdio.h>

#include <conio.h>

#define ARCH "c:\\bin.dat"

#define ESC 27

struct a{

char nombre[10];

int edad;

};

void main (void)

{

FILE \*bin;

struct a pers;

clrscr();

printf("\nSe va a generar por primera vez el archivo\n");

if ((bin=fopen(ARCH,"wb"))==NULL)

{

printf("El archivo no puede ser abierto");

exit(1);

}

do

{

printf("\nIngrese el nombre: ");

gets(pers.nombre);

printf("Ingrese la edad: ");

scanf("%d",&pers.edad);

fflush(stdin);

fwrite(&pers,sizeof(pers),1,bin);

printf("\nPresione ESC para terminar");

}while((getche())!=ESC);

fclose(bin);

}

Antes de escribir el archivo se debe cargar el dato que se desea guardar , en nuestro caso debemos cargar la esctructura. Una vez que se cargaron todos los campos se llama a fwrite y se le pasa la dirección de comienzo de la estructura (&pers) , el tamaño en bytes de la estructura ( se puede escribir directamente o usar sizeof) , la cantidad de estructuras que se van a escribir (se cargó solo una por lo tanto se escribe una) y finalmente el puntero que hace referencia al archivo.En el caso que no se desee escribir la estructura entera se deberán hacer 2 ***fwrite*** , uno para la edad y el otro para el nombre.

El operador sizeof me permite obtener la cantidad de bytes de un determinado dato. Se puede usar con nombres de variables o con tipos de dato (char , int , float , etc). Por ejemplo

int i;

char c;

sizeof (char ) sizeof ( c )

sizeof ( int ) sizeof ( i )

### Lectura de un archivo

Para realizar la lectura de un archivo se utiliza la función ***fread*** que tiene el siguiente prototipo

int ***fread*** ( void \* *destino* , size\_t *tamaño* , size\_t *cantidad* , FILE \**arch* )

Donde:

*destino* Es un puntero al lugar donde se va a dejar el dato leido con fread

*tamaño* Es el tamaño en bytes del dato a leer

*cantidad*  Es la cantidad de elementos de longitud *tamaño* que se van a leer

*arch* Es el puntero a la estructura FILE asociada al archivo desde el que se va a leer.

**Valor retornado:** : Devuelve el número de datos leidos (*cantidad*). Si el valor retornado es menor al que se indicó por medio de la variable *cantidad* , significa que hubo un error en la lectura o que se llegó al final de archivo.

La función ***fread*** lee desde el archivo referenciado por *arch* a partir de la posición actual del indicador de posición , *cantidad* de elementos de longitud *tamaño* y deja los elementos leidos en la dirección de memoria indicada por *destino*.

Una vez que se completó la operación de lectura se actualiza automáticamente el indicador de posición del archivo.

A diferencia de lo que ocurre en la escritura , se debe verificar que se realice la lectura mientras no se haya llegado al final del archivo. Esta operación se realiza por medio de la función ***feof.***

Ejemplo:

#include <stdio.h>

#include <conio.h>

#define ARCH "c:\\bin.dat"

struct a{

char nombre[10];

int edad;

};

void main (void)

{

FILE \*bin;

struct a pers;

int cant;

clrscr();

if ((bin=fopen(ARCH,"rb"))==NULL)

{

printf("No se pudo abrir el archivo");

exit(1);

}

while(!feof(bin))

{

cant=fread(&pers,sizeof(pers),1,bin);

if(cant!=1)

{

if(feof(bin))

break;

else

{

error("No leyo el ultimo registro");

break;

}

}

printf("\n%s\t%d",pers.nombre,pers.edad);

}

fclose(bin);

getch();

}

Despues de hacer la lectura se debe verificar que se haya leido la cantidad de datos que se indicó. Ocurre que cuando no se lee la cantidad de datos indicada puede haberse alcanzado el final de archivo o se pudo haber producido un error.

Es por esto que cuando se entra al if que verifica la cantidad , debemos preguntar si se llego al final del archivo.

Nos queda explicar el uso de la función *feof.*

La función *feof* determina si se ha llegado al final de el archivo , el prototipo es

int *feof* ( FILE \* *arch* )

donde arch es el puntero a la estructura FILE asociada con el archivo. El valor retornado por la función

0 si no se llegó al final del archivo

distinto de cero si se llegó al final del archivo.

### Búsqueda y modificación

En la lectura y escritura de archivos el indicador de posición se actualiza automáticamente , pero existen casos , por ejemplo en las búsquedas y modificaciones sobre archivos , en los cuales se necesita mover el indicador de posición a algún lugar en particular. Para ello se cuenta con 2 funciones que permiten realizar tal operación , ellas son ***fseek*** y ***rewind***. Por otra parte se cuenta con una función que permite obtener el lugar donde se encuentra el indicador de posición del archivo , esa función es ***ftell.*** A continuación se detallan las 3 funciones.

#### rewind

Esta función permite llevar el indicador de posición al comienzo del archivo. El prototipo es el siguiente:

void ***rewind*** ( FILE \**arch* )

La función rewind ubica el indicador de posición del archivo referenciado por el puntero arch al principio y limpia los indicadores de fin de archivo y error que se encuentran el la estructura FILE.

Ejemplo:

#include <stdio.h>

#include <conio.h>

#define ARCH "c:\\bin.dat"

struct a{

char nombre[10];

int edad;

};

void main (void)

{

FILE \*bin;

struct a pers;

int cant;

if ((bin=fopen(ARCH,"rb+"))==NULL)

if ((bin=fopen(ARCH,"wb+"))==NULL)

{

printf("No se pudo abrir el archivo");

exit(1);

}

do

{

printf("\nIngrese el nombre: ");

gets(pers.nombre);

printf("Ingrese la edad: ");

scanf("%d",&pers.edad);

fflush(stdin);

fwrite(&pers,sizeof(pers),1,bin);

printf("\nPresione ESC para terminar");

}while((getche())!=ESC);

rewind (bin); //se lleva el indicador de posición al principio para

while(!feof(bin)) //comenzar a leer

{

cant=fread(&pers,sizeof(pers),1,bin);

if(cant!=1)

{

if(feof(bin))

break;

else

{

error("No leyo el ultimo registro");

break;

}

}

printf("\n%s\t%d",pers.nombre,pers.edad);

}

fclose(bin);

getch();

}

#### fseek

Esta función permite desplazar el indicador de posición del archivo a la posición que se le indique. El prototipo es:

int ***fseek*** ( FILE \**arch* , long *desplazamiento* , int *origen*)

**Donde:**

*arch* Puntero a la estructura FILE asociada con el archivo

*desplazamiento* es la cantidad de bytes que se desplazará el indicador de posición a partir de *origen*

*origen* es una constante que determina el punto de referencia a partir del cuál se realiza el desplazamiento.

Los valores que se le pueden dar a origen figuran en la siguiente tabla. Dichos valores se encuentran definidos en stdio.h

|  |  |
| --- | --- |
| SEEK\_SET | A partir del comienzo del archivo |
| SEEK\_CUR | A partir de la posición actual del archivo |
| SEEK\_END | A partir de el final del archivo |

**Valor retornado**: Si la operación es exitosa devuelve cero , caso contrario retorna un valor distinto de cero.

La función fseek mueve el indicador de posición del archivo *desplazamiento* bytes a partir de la posición indicada por *origen.*

**Ejemplos:**

fseek ( ptr , 0L , SEEK\_SET )

Mueve el indicador de posición al comienzo del archivo. El origen es SEEK\_SET que indica el comienzo del archivo y se desplaza 0 bytes , por lo tanto queda al principio. Es aconsejable que cuando se desee llevar el indicador de posición al comienzo del archivo se utilice ***rewind*** ya que ***fseek*** no limpia el indicador de error ni el de fin de archivo , por lo tanto cuando en determinada situación se use ***fseek*** no va a dar los resultados esperados.

fseek ( ptr , 0L , SEEK\_END )

Mueve el indicador de posición al final del archivo. El origen es SEEK\_END que indica el final del archivo y a partir de alli se desplaza 0 bytes , por lo tanto esta al final del archivo. Si se desea en algún momento agregar datos , simplemente se debe usar esta función para enviar el indicador de posición al final del archivo.

fseek ( ptr , 20L , SEEK\_SET )

Mueve el indicador de posición 20 bytes a partir del comienzo del archivo.

fseek ( ptr , (long) (-1)\*sizeof (struct x) , SEEK\_CUR )

Mueve el indicador de posición una estructura para atras a partir de la posición actual. Normalmente esta forma se utiliza cuando se estan editando datos del archivo. Al realizar una búsqueda se va leyendo cada uno de los datos del archivo por medio de fread, pero cuando encontramos el dato el indicador de posición del archivo esta en el dato siguiente al que queremos modificar , con lo cual al hacer fwrite para escribirlo se modificará otro dato. Por lo tanto antes de escribir se debe mover el indicador de posición una estructura para atrás.

#### ftell

La función ***ftell*** me permite obtener la posición actual del indicador de posición. El prototipo es el siguiente:

long ***ftell*** (FILE \* *arch* )

Donde *arch* es el puntero a la estructura FILE asociada al archivo.

**Valor retornado**: Si la operación es exitosa devuelve la cantidad de bytes que hay desde el commienzo del archivo hasta el lugar en que se encuentra el indicador de posición del archivo , en caso contrario devuelve –1L (-1 como tipo long).

**Ejemplo**: Obtener el tamaño de un archivo en bytes

#include <stdio.h>

#include <conio.h>

#define ARCH "c:\\bin.dat"

void main (void)

{

FILE \*bin;

long int cant;

if ((bin=fopen(ARCH,"rb"))==NULL)

{

printf("No se pudo abrir el archivo");

exit(1);

}

fseek (bin , 0L , SEEK\_END ); //Se envía el indicador de posición al final del archivo

cant=ftell (bin);

printf(“\nEl archivo tiene %ld bytes”,cant);

fclose(bin);

getch();

}

Terminada la descripción de todas las funciones que se utilizarán sobre archivos , podemos escribir 2 ejemplos simples de búsqueda y modificación .

**Búsqueda:** En el archivo person.dat que ya contiene datos cargados se desea buscar las edades de varias personas indicando para la búsqueda el nombre.

#include <stdio.h>

#include <conio.h>

struct a{

char nombre[10];

int edad;

};

void main (void)

{

FILE \*bin;

struct a pers;

int cant;

char auxnom[10],rta=’s’;

clrscr();

if ((bin=fopen(“c:\\person.dat”,"rb"))==NULL)

{

printf("No se pudo abrir el archivo");

exit(1);

}

while(rta= = ‘s’)

{

printf(“\nIngrese nombre a buscar”);

gets(auxnom);

while(!feof(bin))

{

cant=fread(&pers,sizeof(pers),1,bin);

if(cant!=1)

{

if(feof(bin))

break;

else

{

error("No leyo el ultimo registro");

break;

}

}

if (¡strcmp(pers.nombre,auxnom))

printf("\n%s\t%d",pers.nombre,pers.edad);

}

printf(“\nDesea continuar?”);

rta=getche();

}

fclose(bin);

getch();

}

La forma de buscar en esencia es la misma que cuando se busca en un vector , la diferencia principal es que no podemos usar un for ni definir un vector para guardar los datos del archivo debido a que en el momento de la definición del vector no se sabe la cantidad de elementos (estructuras) del archivo.

Al igual que en cualquier búsqueda se debe ingresar el dato a buscar , que en nuestro caso es el nombre. Conocido el nombre se va leyendo una a una las estructuras que se encuentran en el archivo ( por medio de fread ) y se va comparando el nombre ingresado con el que estaba guardado en el archivo. Este proceso se hace hasta encontrar el final de archivo.

**Modificación:**

Básicamente tiene la misma forma que en el ejemplo anterior , con el agregado que cuando se encuentra el dato se modifica en memoria , se cambia el indicador de posición del archivo para luego escribirlo.

#include <stdio.h>

#include <conio.h>

struct a{

char nombre[10];

int edad;

};

void main (void)

{

FILE \*bin;

struct a pers;

int cant;

char auxnom[10],rta=’s’;

clrscr();

if ((bin=fopen(“c:\\person.dat”,"rb"))==NULL)

{

printf("No se pudo abrir el archivo");

exit(1);

}

while(rta= = ‘s’)

{

printf(“\nIngrese nombre a buscar”);

gets(auxnom);

while(!feof(bin))

{

cant=fread(&pers,sizeof(pers),1,bin);

if(cant!=1)

{

if(feof(bin))

break;

else

{

error("No leyo el ultimo registro");

break;

}

}

if (¡strcmp(pers.nombre,auxnom))

{

printf("\ningrese nueva edad”);

scanf(“%d”,&pers.edad);

fseek(bin , (long) (-1)\*sizeof(struct a) , SEEK\_CUR);

fwrite(&pers,sizeof(pers),1,bin);

break;

}

}

printf(“\nDesea continuar?”);

rta=getche();

}

fclose(bin);

getch();

}

# Gestión dinámica de memoria

**Estructuras dinámicas de datos.**     En función de la forma en que se relacionan existen varios tipos de estructuras de datos. Este tipo de estructuras son autorreferenciadas, es decir, contienen entre sus campos un puntero de su mismo tipo. Las más utilizadas son:

**- pilas**

**- colas**

**- listas**

## Pilas

Este tipo de estructuras se caracteriza porque todas las operaciones se realizan en el mismo lado. Es de tipo **LIFO** ( **L**ast **I**n **F**irst **O**ut ), el último elemento en entrar es el primero en salir.  
     La estructura tipo que utilizaremos será ésta:

struct pila

{

tipo variables;

struct pila \*ant;

};

donde tipo variables serán las diferentes variables que guardaremos en la estructura, struct pila \*ant es un puntero que apunta al elemento de tipo pila introducido anteriormente. Además declararemos 2 variables de puntero del tipo struct pila. \*ultimo será donde guardaremos el último elemento insertado en la pila y \*actual nos servirá para guardar elementos temporalmente.

### Insertar elementos en una pila

Veamos primero el código completo

#include <stdio.h>

#include <conio.h>

struct pila{

char nombre[6];

struct pila \*ant;

};

void main()

{

char opc;

struct pila \*ultimo=NULL,\*actual=NULL;

do

{

actual= new struct pila;

printf("Nombre: ");

gets(actual->nombre);

actual->ant=ultimo;

ultimo=actual;

printf ("Tecla para seguir insertando elementos...ESC para salir");

opc=getche();

}while (opc!=27);

}

Veamos ahora el código con comentarios

#include <stdio.h>

#include <conio.h>

struct pila{

Definición de la estructura autorreferenciada conteniendo un puntero a una estructura del mismo tipo declarado

char nombre[6];

struct pila \*ant;

};

void main(){

Declara 2 variables de puntero del tipo struct pila y se las inicializa en NULL para que no apunten a ningún lugar de la memoria

char opc;

struct pila \*ultimo=NULL,\*actual=NULL;

Al inspeccionar las variables su contenido mostrará algo similar a lo siguiente

nombre char[6]

ant struct pila\*

Inspecting: **actual**

Struct pila\* 3A7F:007A >> **NULL**

"\x0\x0\x0\x0\x0\x0"

**NULL**

nombre char[6]

ant struct pila\*

Inspecting: **ultimo**

Struct pila\* 3A7F:0076 >> **NULL**

"\x0\x0\x0\x0\x0\x0"

**NULL**

do{

Reserva memoria en forma dinámica para la variable actual que será utilizada como auxiliar. Con esta sentencia se crea en memoria un elemento del tipo struct pila

actual= new struct pila;

nombre char[6]

ant struct pila\*

Inspecting: **actual**

Struct pila\* 3A7F:007A >> **39FF:00EE**

"**7:\x0ú\x0\x0**"

018F:0000

Lee el campo nombre de la variable actual. Supongamos que ingrese PEPE

printf("Nombre: ");

gets(actual->nombre);

nombre char[6]

ant struct pila\*

Inspecting: **actual**

Struct pila\* 3A7F:007A >> 39FF:00EE

"**PEPE\x0\x0**"

018F:0000

Se asigna al campo ant de la variable actual la dirección a la que apunta ultimo, que al haberse inicializado en NULL apunta a NULL

actual->ant=ultimo;

nombre char[6]

ant struct pila\*

Inspecting: **actual**

Struct pila\* 3A7F:007A >> 39FF:00EE

"PEPE\x0\x0"

**NULL**

Entonces el campo ant de la variable actual contendrá NULL

Se asigna ultimo la dirección a la que apunta actual que es 39FF:00EE

ultimo=actual;

nombre char[6]

ant struct pila\*

Inspecting: **ultimo**

Struct pila\* 3A7F:0076 >> **39FF:00EE**

"PEPE\x0\x0"

NULL

Entonces ultimo pasa a apuntar a esa dirección ( 39FF:00EE )

printf ("Tecla para seguir insertando elementos...ESC para salir");

opc=getche();

}while (opc!=27);

}

La segunda vez que se Inserta un elemento, si inspeccionamos las variables veríamos la siguiente información:

Luego de la sentencia **actual= new struct pila**;

nombre char[6]

ant struct pila\*

Inspecting: actual

Struct pila\* 3A7F:007A >> 39FF:00DE

"\x2\x14\x0\x0\ x0\x0"

0100:0100

Se crea un nuevo elemento actual que apunta a la dirección 39FF:00DE y cuyos campos contienen basura

Luego de la sentencia **gets(actual->nombre);** suponiendo que se ingresa LOPEZ

Se completa el campo nombre con el valor ingresado

nombre char[6]

ant struct pila\*

Inspecting: actual

Struct pila\* 3A7F:007A >> **39FF:00DE**

"**LOPEZ**x0"

0100:0100

Luego de la sentencia **actual->ant=ultimo;**

El campo ant de la variable actual pasa a contener la dirección de memoria a la que apunta la variable ultimo, en este caso 39FF:00EE, es decir apunta al elemento ingresado antes o dicho de otro modo al anterior

nombre char[6]

ant struct pila\*

Inspecting: **actual**

Struct pila\* 3A7F:007A >> 39FF:00DE

"LOPEZ\x0"

**39FF:00EE**

Luego de la sentencia **ultimo=actual;**

nombre char[6]

ant struct pila\*

Inspecting: **ultimo**

Struct pila\* 3A7F:0076 >> **39FF:00DE**

" LOPEZ\x0"

**39FF:00EE**

Se asigna a la variable ultimo la dirección de memoria a la que apunta actual, en este caso 39FF:00DE, por lo tanto sus campos van a tener los mismos valores que actual, incluyendo el campo ant que contendrá la dirección de memoria del elemento anterior

### Esquema gráfico de inserción de elementos de una pila

ESQUEMA CORRESPONDIENTE AL PRIMER ELEMENTO.

Esquematizando todo lo visto podríamos representarlo del siguiente modo:

|  |  |  |
| --- | --- | --- |
|  | SENTENCIAS | REPRESENTACIÓN GRÁFICA |
|  | struct pila \*ultimo=NULL,\*actual=NULL; | ultimo NULL  actual NULL |
| 1  **¿?** | **actual= new struct pila;** | actual |
| 2  PEPE  **¿?** | **gets(actual->nombre);** | actual |
| 3  PEPE  **NULL** | **actual->ant=ultimo;** | actual  ultimo NULL |
| 4  PEPE  **NULL** | **ultimo=actual;** | actual  ultimo |

**ESQUEMA PARA EL SEGUNDO Y TERCER ELEMENTO**

|  |  |  |
| --- | --- | --- |
|  | 2º ELEMENTO | 3º ELEMENTO |
| 1  **¿?**  actual  **¿?**  actual |  |  |
| 2  PEREZ  **¿?**  actual  LOPEZ  **¿?**  actual |  |  |
| 3  LOPEZ  PEPE  **NULL**  PEREZ  actual  ultimo  LOPEZ  actual  PEPE  **NULL**  ultimo |  |  |
| 4  LOPEZ  actual  ultimo  PEPE  **NULL**  PEREZ  actual  ultimo  LOPEZ  PEPE  **NULL** |  |  |

### Desapilar elementos de una pila

Veamos primero el código completo incorporando las sentencias para desapilar elementos y listar la pila

#include <stdio.h>

#include <conio.h>

struct pila{

char nombre[6];

struct pila \*ant;

};

void main(){ /\* Insertar, extraer y visualizar \*/

char opc;

struct pila \*ultimo=NULL,\*actual=NULL;

do{

clrscr();

printf("1.- Insertar");

printf("\n2.- Extraer");

printf("\n3.- Visualizar la pila");

printf("\n4.- Salir");

opc=getch( );

switch(opc){

case '1':

clrscr();

actual= new struct pila;

printf("Nombre: ");

gets(actual->nombre);

actual->ant=ultimo;

ultimo=actual;

break;

case '2':

if (ultimo==NULL)

{

printf("\nNo hay elementos para desapilar");

getch();

break;

}

actual=ultimo;

ultimo=ultimo->ant;

delete actual;

printf("\nElemento desapilado");

getch();

break;

case '3':

clrscr();

if (ultimo==NULL){

printf("No hay elementos para listar");

getch();

break;

}

actual=ultimo;

while (actual!=NULL){

printf("Nombre: %s\n",actual->nombre);

actual=actual->ant;

}

getch( );

}

}while (opc!='4');

}

Ahora veamos el código comentado

#include <stdio.h>

#include <conio.h>

struct pila{

char nombre[6];

struct pila \*ant;

};

void main(){

char opc;

struct pila \*ultimo=NULL,\*actual=NULL;

do{

clrscr();

printf("1.- Insertar");

printf("\n2.- Extraer");

printf("\n3.- Visualizar la pila");

printf("\n4.- Salir");

opc=getch( );

switch(opc){

case '1':

clrscr();

actual= new struct pila;

printf("Nombre: ");

gets(actual->nombre);

actual->ant=ultimo;

ultimo=actual;

break;

case '2':

if (ultimo==NULL){

printf("\nNo hay elementos para desapilar");

getch();

break;

}

Asigna a la variable actual la dirección a la que apunta ultimo, que coincide con ser el último ingresado

actual=ultimo;

A la variable ultimo le asigna la dirección a la que apunta su campo ant que es la dirección donde se encuentra el que se ingresó antes o dicho de otro modo el anterior

ultimo=ultimo->ant;

Se elimina actual, liberando la memoria que ocupaba

delete actual;

printf("\nElemento desapilado");

getch();

break;

case '3':

clrscr();

if (ultimo==NULL){

printf("No hay elementos para listar");

getch();

break;

}

Asigna a la variable actual la dirección a la que apunta ultimo, que coincide con ser el último ingresado

actual=ultimo;

Itera mientras actual no apunte a NULL. En realidad el único que quedó en la pila apuntando a NULL fue el primero que se cargó. Entonces iterará hasta que llegue a contener la dirección del primero

while (actual!=NULL){

Muestra el campo nombre de la dirección de memoria a la que apunta

printf("Nombre: %s\n",actual->nombre);

actual=actual->ant;

A actual le asigna la dirección de memoria a la que apunta su campo ant. Dicha dirección con en la que se encuentra el elemento que se ingresó antes. Cuando este campo sea igual a NULL, el while deja de iterar

}

getch( );

}

}while (opc!='4');

}

### Esquema gráfico de la eliminación de los elementos de una pila

|  |  |  |
| --- | --- | --- |
|  | SENTENCIAS | REPRESENTACIÓN GRÁFICA |
| 1  PEREZ  actual  ultimo  LOPEZ  PEPE  **NULL** | actual=ultimo; |  |
| 2  PEREZ  actual  ultimo  LOPEZ  PEPE  **NULL** | ultimo=ultimo->ant; |  |
| 3  PEREZ  actual  ultimo  LOPEZ  PEPE  **NULL** | delete actual; |  |

### Esquema gráfico para recorrer los elementos de una pila

|  |  |
| --- | --- |
| SENTENCIAS | REPRESENTACIÓN GRÁFICA |
| actual=ultimo;  PEREZ  actual  ultimo  LOPEZ  PEPE  **NULL** |  |
| while (actual!=NULL) | FALSO. Actual apunta al elemento que contiene PEREZ |
| printf("%s", actual->nombre); | Muestra el contenido del campo nombre del elemento al que apunta, en este caso PEREZ |
| actual=actual->ant;  PEREZ  ultimo  actual  LOPEZ  PEPE  **NULL** | Ahora actual apunta al elemento que contiene LOPEZ. La condición del while será Verdadera entonces mostrará el nombre y así sucesivamente hasta que luego de mostrar PEPE esta sentencia provocará que actual apunte a NULL y la condición sea Falsa y salga del bucle |

### Funciones que permiten extraer, listar , eliminar y destruir elementos de una pila

Al utilizar funciones para realizar tareas en una pila, es importante no perder nunca la dirección del ultimo elemento. Por eso tanto para insertar como extraer elementos debemos generar funciones que retornen la dirección de este luego de haberse agregado o quitado algún elemento.

/\* Ejemplo de una pila utilizando funciones para apilar, desapilar y mostrar \*/

#include <stdio.h>

#include <conio.h>

struct pila{

char nombre[6];

struct pila \*ant;

};

struct pila \* insertar(struct pila \*);

struct pila \* extraer(struct pila \*);

void visualizar(struct pila \*);

void destruir(struct pila \*);

void main(){ /\* Insertar, extraer y visualizar \*/

char opc;

struct pila \*ultimo=NULL;

do{

clrscr();

printf("1.- Insertar");

printf("\n2.- Extraer");

printf("\n3.- Visualizar la pila");

printf("\n4.- Salir");

opc=getch( );

switch(opc){

case '1':

ultimo=insertar(ultimo);

break;

case '2':

ultimo=extraer(ultimo);

break;

case '3':

visualizar(ultimo);

}

}while (opc!='4');

destruir(ultimo);

}

struct pila \* insertar(struct pila \*ultimo){

struct pila \*actual;

clrscr();

actual= new struct pila;

printf("Nombre: ");

gets(actual->nombre);

actual->ant=ultimo;

ultimo=actual;

return(ultimo);

}

struct pila \* extraer(struct pila \*ultimo){

struct pila \*actual;

if (ultimo==NULL){

printf("\nNo hay elementos para desapilar");

getch();

}

else{

actual=ultimo;

ultimo=ultimo->ant;

delete actual;

printf("\nElemento desapilado");

getch();

}

return(ultimo);

}

void visualizar(struct pila \*ultimo){

struct pila \*actual;

clrscr();

if (ultimo==NULL){

printf("No hay elementos para listar");

getch();

}

else{

actual=ultimo;

while (actual!=NULL){

printf("Nombre: %s\n",actual->nombre);

actual=actual->ant;

}

}

getch( );

}

void destruir(struct pila \*ultimo){

struct pila \*actual;

clrscr();

if (ultimo==NULL){

printf("No hay elementos en la lista");

getch();

}

else{

actual=ultimo;

while (actual!=NULL){

ultimo=actual->ant;

delete actual;

actual=actual->ant;

}

}

}

## Colas

En una cola el primer elemento en ingresar es el primero en salir. El manejo de la cola es básicamente el mismo que el de la pila con la diferencia del órden en que ingresan y salen los elementos. Las operaciones básicas en una cola son

1. Insertar elementos
2. Eliminar elementos
3. Destruir
4. Recorrer

## Listas simplemente encadenadas

La diferencia entre una lista y una cola o pila es que los elementos de la lista pueden insertarse en cualquier lugar , logrando de esta forma que la lista quede ordenada.

Solamente trabajaremos con listas simplemente encadenadas que son aquellas que tienen una referencia al elemento siguiente.

Cada elemento de la lista será una estructura autoreferenciada , la cuál esta formada por los datos y un puntero a la estructura del mismo tipo.

La estructura que utilizaremos es la siguiente:

typedef struct lista {

int dato;

struct lista \*prox;

}Nodo;

como se puede ver la estructura contiene datos (este ejemplo solo trata un entero) y un puntero a *struct lista.*



Cada elemento de la lista lo representaremos por medio del diagrama de la Figura 1

Donde aux es el puntero al comienzo de la estructura , el 8 representa el dato cargado y la flecha que sale es el punero que va a apuntar al próximo elemento de la lista.



La lista comleta quedaría representada por la Figura 2

La única forma de poder recorrer una lista simplemente encadenada es accediendo por el primer elemento (que nunca debe perderse) . El final de la lista se encuentra por que el puntero al próximo elemento apunta a NULL.

Básicamente existen 3 casos distintos para insertar un elemento en una lista. Para el análisis trataremos cada uno por separado para luego generar una función que permita insertar un elemento en la lista.

Los 3 casos básicos son:

Insertar un elemento al comienzo de la lista

Insertar un elemento al final de la lista

Insertar un elemento en algun lugar intermedio.

Antes de empezar a analizar cada caso en particular conviene ponernos de acuerdo en cuanto al significado y nomenclatura de los punteros a usar.

Siempre , en algún lugar del programa , normalmente en *main* se debe llevar actualizado el puntero al comienzo de la lista.

Vamos a utilizar 4 punteros a saber

Inicio Contiene la dirección de comienzo de la lista. Cuando la lista esta vacía vale NULL.

aux Contiene la dirección de comienzo del nuevo elemento

p,w Son 2 punteros que sirven para recorrer la lista.

Para agregar un elemento a la lista se deben seguir los pasos que a continuación se enumeran:

Realizar la solicitud de memoria dinámica por medio del operador ***new***

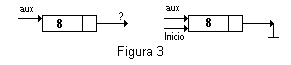
Cargar los datos en la estructura.

Enlazar el nuevo elemento con los que existan en la lista

Actualizar el puntero al primer elemento.

### Primer elemento de la lista - Lista Vacia

Inicialmente no existen elementos en la lista. Para identificar fácilmente esta condición asignamos a NULL el puntero al primer elemento (Inicio=NULL)



El primer ingreso se representa en la Figura 3

p=Inicio como es el primer elemento p apunta a NULL

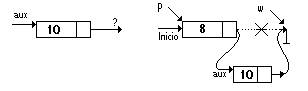
SI ( p==NULL) Si es verdadero entonces es el único elemento de la lista

{ Inicio=aux El inicio de la lista es la dirección del nuevo elemento

aux->prox=NULL

}

### Insertar al final de la lista



Para saber donde se va a insertar el nuevo elemento se debe comparar el valor del dato del nuevo elemento contra cada uno de los elementos de la lista.

Inicialmente se apunta p al comienzo de la lista y w al proximo elemento.

p=Inicio

w=p->prox

---------------

---------------

SI (w==NULL)

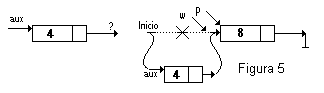
{ p->prox=aux

aux->prox=w

}

### Insertar al comienzo de la lista

La representación gráfica de este caso se observa en la Figura 5



p=Inicio

w=Inicio

SI (aux->dato < w->dato )

{ Inicio=aux

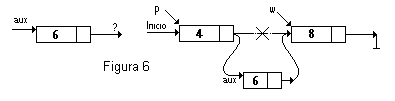
aux->prox=w

}

### Insertar en una posición intermedia

Para saber en que posición de la lista se debe insertar el elemento se debe recorrer la lista y detenerse cuando el valor del dato nuevo sea menor al valor del dato del próximo elemento.

La figura 6 representa esta condición.



p=Inicio

w=Inicio

MIENTRAS (w !=NULL )

{

SI (aux->dato < w->dato )

{

p->prox=aux

aux->prox=w

BREAK

}

p=w

w=p->prox

}

### Función para insertar un elemento en una lista

La función para insertar un elemento necesita como parámetro formal el puntero al primer elemento de la lista y devolverá la nueva dirección de comienzo para que se pueda actualizar desde la función que realizó el llamado.

Dentro de la función se pueden ver identificados cada uno de los casos analizados anteriormente.

Nodo \* Insertar (Nodo \*Inicio)

{

Nodo \*p,\*w,\*aux;

aux=new Nodo;

if(!aux)

{

puts("\nNo hay memoria disponible");

return Inicio;

}

printf("\nIngrese Dato: ");

scanf("%d",&aux->dato);

//Si Lista vacia,Primer elemento que se carga en la lista

if ( !Inicio) //Si Inicio es NULL

{

Inicio=aux;

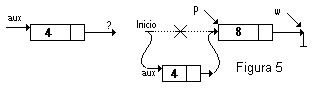
aux->prox=NULL;

return Inicio;

}

//Verifico si el elemento a insertar es el primero de la lista

p=Inicio;



w=Inicio;

if(aux->dato<p->dato)

{

aux->prox=Inicio;

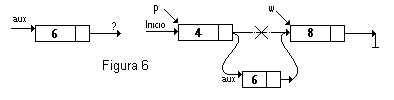
Inicio=aux;

return Inicio;

}

while(w)

{



if(aux->dato<w->dato)

{

aux->prox=w;

p->prox=aux;

break;

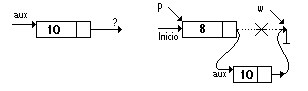
}

p=w;

w=p->prox;

}

if(w==NULL)



{

p->prox=aux;

aux->prox=NULL;

}

return Inicio;

}

Si prestamos atención a los casos citados hay varios puntos comunes que permiten compactar un poco el código de la función que inserta.

Podemos entonces reescribirla de la siguiente forma

Nodo \* Insertar(Nodo \*Inicio)

{

Nodo \*p,\*w,\*aux;

p=w=Inicio;

aux=new Nodo;

if(!aux)

{

puts("\nNo hay memoria disponible");

return Inicio;

}

printf("\nIngrese Dato: ");

scanf("%d",&aux->dato);

while(w)

{

if(aux->dato < w->dato)

break;

p=w;

w=p->prox;

}

if(w==Inicio)

{

aux->prox=w;

Inicio=aux;

}

else

{

aux->prox=w;

p->prox=aux;

}

return Inicio;

}

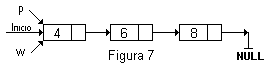
Inicialmente se colocan los punteros auxiliares al comienzo de la lista (para poder detectar si se inserta al comienzo) y se comienza a recorrer la lista mientras no se encuentre el final de la lista (puntero **NULL**). Cuando se encuentra el lugar en el que se debe insertar se sale del ***While.*** En este momento se debe verificar si el puntero w esta al comienzo o no, en el caso de encontrarse al comienzo significa que el nuevo dato se inserta al comienzo, caso contrario se insertará en algún lugar de la lista.

### Función para eliminar un elemento de la ista

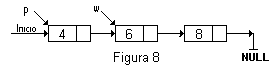
La función para remover un elemento deberá recibir el puntero al comienzo de la lista y el elemento que se desea eliminar.

En el caso que el elemento a eliminar sea el primero de la lista se debería actualizar el puntero al comienzo , por lo tanto la función debería devolver el puntero al nuevo comienzo de la lista.

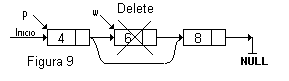
Para poder eliminar el elemento deseado se debe buscar a partir del comienzo por medio de los 2 punteros auxiliares. Supongamos tener una lista como la de la figura 7 donde el número a remover es el 6. Inicialmente los punteros p y w se encuentran al inicio de la lista



Se realizará la comparación del número a buscar con el valor del dato apuntado por w , por lo tanto en este primer caso la comparación es FALSO y se deberá mover el puntero al próximo elemento (Figura 8)



Cuando ahora se realiza la comparación el resultado es VERDADERO y debemos realizar la modificación de los punteros al próximo elemento para luego eliminar el nodo apuntado por w de acuerdo a la figura 9.



Nodo \* Remover(Nodo \*Inicio,int nro)

{

Nodo \*p,\*w;

if (Inicio==NULL)

{

printf("No hay elementos en la lista");

return Inicio;

}

p=Inicio;

w=p->prox;

if(p->dato==nro)

{

Inicio=p->prox;

delete p;

return Inicio;

}

while(w)

{

if (w->dato==nro)

{

p->prox=w->prox;

delete w;

break;

}

p=w;

w=p->prox;

}

if(!w)

puts("\nNo se encontro el numero en la lista");

return Inicio;

}

Otra forma de realizar la misma función es la siguiente

Nodo \* Remover(Nodo \*Inicio,int nro)

{

Nodo \*p,\*w;

p=Inicio;

w=Inicio;

while(w)

{

if (w->dato==nro)

break;

p=w;

w=p->prox;

}

if(!w)

puts("\nNo se encontro el numero en la lista");

else

{

if(w==Inicio)

Inicio=w->prox;

else

p->prox=w->prox;

delete w;

}

return Inicio;

}

### Función para mostrar todos los elementos de la lista

Para mostrar la lista completa solo necesitamos un puntero auxiliar , que es el que irá desplazándose a lo largo de la lista.

Inicialmente apuntamos al comienzo de la lista y, mientras no se llegue al final , se muestra el dato cargado y se cambia el valor del puntero al comienzo del próximo elemento de la lista ( p= p->prox)

Si se presta atención al código , no haría falta usar el puntero auxiliar para recorrer la lista , sinó que se puede utilizar el puntero que se recibe en la función. Recuerde que el puntero Inicio de la función es una variable local de la misma y por lo tanto la modificación del valor no afecta sobre variables con el mismo nombre en otras funciones.

void Listar(Nodo \*Inicio)

{

Nodo \*p;

p=Inicio;

while(p)

{

printf("|%4d|-->",p->dato);

p=p->prox;

}

}

### Función para eliminar la lista completa

Antes de terminar el programa se tiene que devolver toda la memoria solicitada por el operador ***new*** en cada ingreso de datos.

Por lo tanto se debe recorrer la lista eliminando por medio del operador delete todos los nodos.

void Eliminar(Nodo \*Inicio)

{

Nodo \*p,\*w;

p=Inicio;

while(p)

{

w=p->prox;

delete p;

p=w;

}

}

### Programa completo

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <ctype.h>

typedef struct lista{

int dato;

struct lista \*prox;

}Nodo;

Nodo \* Insertar(Nodo \*Inicio);

Nodo \* Remover(Nodo \*Inicio,int nro);

void Listar(Nodo \*Inicio);

void Eliminar(Nodo \*Inicio);

void main(void)

{

Nodo \*pr=NULL;

char opc;

int nro;

while(1)

{

clrscr();

puts("\n-Agregar Dato");

puts("\n-Eliminar Dato");

puts("\n-Listar");

puts("\n-Salir");

puts("\Ingrese opcion: ");

opc=toupper(getche());

switch(opc)

{

case 'A': pr=Insertar(pr);

break;

case 'E': printf("Ingrese nro a eliminar ");

scanf("%d",&nro);

pr=Remover(pr,nro);

break;

case 'L': Listar(pr);

getch();

break;

case 'S': exit(0);

}

}

}

Nodo \* Insertar(Nodo \*Inicio)

{

Nodo \*p,\*w,\*aux;

p=w=Inicio;

aux=new Nodo;

if(!aux)

{

puts("\nNo hay memoria disponible");

return Inicio;

}

printf("\nIngrese Dato: ");

scanf("%d",&aux->dato);

while(w)

{

if(aux->dato < w->dato)

break;

p=w;

w=p->prox;

}

if(w==Inicio)

{

aux->prox=w;

Inicio=aux;

}

else

{

aux->prox=w;

p->prox=aux;

}

return Inicio;

}

Nodo \* Remover(Nodo \*Inicio,int nro)

{

Nodo \*p,\*w;

p=Inicio;

w=Inicio;

while(w)

{

if (w->dato==nro)

break;

p=w;

w=p->prox;

}

if(!w)

puts("\nNo se encontro el numero en la lista");

else

{

if(p==Inicio)

{

Inicio=p->prox;

delete p;

}

else

{

p->prox=w->prox;

delete w;

}

}

return Inicio;

}

void Listar(Nodo \*Inicio)

{

Nodo \*p;

p=Inicio;

while(p)

{

printf("|%4d|-->",p->dato);

p=p->prox;

}

}

void Eliminar(Nodo \*Inicio)

{

Nodo \*p,\*w;

p=Inicio;

while(p)

{

w=p->prox;

delete p;

p=w;

}

}

# Ejercicio Tipo

Desarrollaremos un ejercicio tipo de distintas formas , cubriendo cada una de los temas vistos durante el curso.

La idea de realizar este ejercicio de distintas maneras es que analice los cambios en la forma de tratar los datos en cada uno de los problemas y que observe como se mantienen sin variación los algoritmos para realizar cada parte del programa.

El ejercicio es el siguiente:

Cargar nombre y nota de 20 alumnos. Terminada la carga permitir cambiar la nota de los alumnos ingresando el nombre y la nueva nota. Finalizados los cambios mostrar los siguientes resultados:

El nombre del alumno que tiene la nota mas baja.

El nombre del alumno que tiene la nota mas alta.

Nombre y nota de los alumnos ordenado alfabéticamente.

## Desarrollo del ejercicio tipo Usando índices

Para resolver el problema primero planteamos donde se van a guardar los datos. Para ello usamos una matriz de caracteres donde guardaremos los nombres y un vector de enteros para las notas.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | M | u | s | u | m | e | c | i | \0 |  |  |  |  |  |  |  |  | 0 | 9 |
| 1 | I | d | i | a | r | t | \0 |  |  |  |  |  |  |  |  |  |  | 1 | 7 |
| 2 | A | v | e | r | s | a | \0 |  |  |  |  |  |  |  |  |  |  | 2 | 8 |
| 3  Vector de notas | C | a | p | o | b | i | a | n | c | o | \0 |  |  |  |  |  |  | 3 | 5 |
| 4 | F | e | r | n | a | n | d | e | z | \0 |  |  |  |  |  |  |  | 4 | 2 |
| 5 | M | o | r | e | s | c | o | \0 |  |  |  |  |  |  |  |  |  | 5 | 4 |
| 6 | P | e | r | e | z | \0 |  |  |  |  |  |  |  |  |  |  |  | 6 | 6 |
| 7 | P | i | ñ | e | i | r | o | \0 |  |  |  |  |  |  |  |  |  | 7 | 10 |

Matriz de nombres

Una vez conocida la disposición de los datos , se procede a dividir el programa en partes mas chicas.

Observe que en el desarrollo del programa que sigue y en el resto de los programas escritos se resalta con comentarios cada uno de los bloques que componen el programa. Tenga en cuenta que cualquier problema complejo se puede dividir en varios problemas simples.

#include <stdio.h>

#include <string.h>

#define FIL 20

#define COL 25

void main(void)

{

int nota[FIL],i,j,max\_nota,min\_nota,imax=0,imin=0,nnota,flag=0;

char nom[FIL][COL],aux[COL],rta;

clrscr();

**Carga de datos**

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

{

printf("Ingrese nombre: ");

gets(nom[i]);

printf("Ingrese nota: ");

scanf("%d",&nota[i]);

fflush(stdin);

}

**Modificacion de la nota**

printf("\nDesea cambiar alguna nota... S/N?");

rta=toupper(getche());

while(rta=='S')

{

printf("\nIngrese el nombre del alumno: ");

gets(aux);

printf("\nIngrese la nueva nota: ");

scanf("%d",&nnota);

fflush(stdin);

for(i=0;i<FIL-1;i++)

if((strcmp(aux,nom[i]))==0)

{

flag=1;

printf("\nLa nota del alumno %s es %d...la modifica S/N?",nom[i],nota[i]);

rta=toupper(getche());

if(rta=='S')

{

nota[i]=nnota;

printf("\nSe realizo el cambio");

}

else

printf("\nNo se acepto el cambio");

break;

}

if(!flag) /\* es lo mismo que if(flag==0) \*/

printf("\No se encontro el nombre %s",aux);

flag=0;

printf("\nDesea cambiar otra nota... S/N?");

rta=toupper(getche());

}

**Buscar el nombre de la nota mas alta**

max\_nota=nota[0];

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

if(max\_nota<nota[i])

{

max\_nota=nota[i];

imax=i;

}

**Buscar el nombre de la nota mas baja**

min\_nota=nota[0];

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

if(min\_nota>nota[i])

{

min\_nota=nota[i];

imin=i;

}

printf("\nLa nota mas baja corresponde a %s",nom[imin]);

printf("\nLa nota mas alta corresponde a %s",nom[imax]);

**Ordenar los datos por nombre**

for(i=0;i<FIL-1;i++)

for(j=i+1;j<FIL;j++)

{

if((strcmp(nom[i],nom[j]))<0)

{

strcpy(aux,nom[i]);

strcpy(nom[i],nom[j]);

strcpy(nom[j],aux);

nnota=nota[i];

nota[i]=nota[j];

nota[j]=nnota;

}

}

**Mostrar los datos ordenados**

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

printf("\n%-10s%d",nom[i],nota[i]);

}

## Desarrollo del ejercicio tipo Usando punteros

Para realizar el ejercicio usando punteros lo primero que debemos hacer es colocar punteros apuntando al comienzo de la matriz y el vector. Nuestro esquema de datos quedaría

q

p

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | M | u | s | u | m | e | c | i | \0 |  |  |  |  |  |  |  |  | 0 | 9 |
|  | I | d | i | a | r | t | \0 |  |  |  |  |  |  |  |  |  |  | 1 | 7 |
|  | A | v | e | r | s | a | \0 |  |  |  |  |  |  |  |  |  |  | 2 | 8 |
|  | C | a | p | o | b | i | a | n | c | o | \0 |  |  |  |  |  |  | 3 | 5 |
|  | F | e | r | n | a | n | d | e | z | \0 |  |  |  |  |  |  |  | 4 | 2 |
|  | M | o | r | e | s | c | o | \0 |  |  |  |  |  |  |  |  |  | 5 | 4 |
|  | P | e | r | e | z | \0 |  |  |  |  |  |  |  |  |  |  |  | 6 | 6 |
|  | P | i | ñ | e | i | r | o | \0 |  |  |  |  |  |  |  |  |  | 7 | 10 |

Para realizar el ejercicio usando punteros tenemos que conocer las equivalencias entre vectores y punteros, las cuales pasamos a detallar. Estas equivalencias son válidas siempre y cuando se haya inicializado el puntero correctamente al comienzo.

#### En un vector

vec [ i ] \* ( p + i)

& vec [ i ] p + i

#### En una matriz

mat [ i ][ j ] \* ( p + i \* COL + j)

& mat [ i ][ j ] p + i \* COL + j

Para el caso particular de las matrices que guardan cadenas de caracteres, j vale cero y reemplazamos **mat [i]** por **p+i\*COL**

Para mayor detalle acerca de los punteros a cadenas de caracteres, referirse al tema ***Cadena de Caracteres***

A continuación se desarrolla el ejercicio utilizando punteros

#include <stdio.h>

#include <string.h>

#include <conio.h>

#define FIL 2

#define COL 25

void main(void)

{

int nota[FIL],i,j,max\_nota,min\_nota,imax=0,imin=0,nnota,flag=0,\*q;

char nom[FIL][COL],aux[COL],rta,\*p;

clrscr();

p=&nom[0][0];

q=&nota[0];

**Carga de datos**

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

{

printf("Ingrese nombre: ");

gets(p+i\*COL);

printf("Ingrese nota: ");

scanf("%d",q+i);

fflush(stdin);

}

**Modificacion de la nota**

printf("\nDesea cambiar alguna nota... S/N?");

rta=toupper(getche());

while(rta=='S')

{

printf("\nIngrese el nombre del alumno: ");

gets(aux);

printf("\nIngrese la nueva nota: ");

scanf("%d",&nnota);

fflush(stdin);

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

if((strcmp(aux,p+i\*COL))==0)

{

flag=1;

printf("\nLa nota del alumno %s es %d...la modifica S/N?",p+i\*COL,\*(q+i));

rta=toupper(getche());

if(rta=='S')

{

\*(q+i)=nnota;

printf("\nSe realizo el cambio");

}

else

printf("\nNo se acepto el cambio");

break;

}

if(!flag) /\* es lo mismo que if(flag==0) \*/

printf("\No se encontro el nombre %s",aux);

flag=0;

printf("\nDesea cambiar otra nota... S/N?");

rta=toupper(getche());

}

**Buscar nota mayor**

max\_nota=\*q;

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

if(max\_nota<\*(q+i))

{

max\_nota=\*(q+i);

imax=i;

}

**Buscar nota menor**

min\_nota=\*q;

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

if(min\_nota>\*(q+i))

{

min\_nota=\*(q+i);

imin=i;

}

printf("\nLa nota mas baja corresponde a %s",\*(q+imin));

printf("\nLa nota mas alta corresponde a %s",\*(q+imax));

**Ordenar datos por apellido**

for(i=0;i<FIL-1;i++)

for(j=i+1;j<FIL;j++)

{

if((strcmp(p+i\*COL,p+j\*COL))<0)

{

strcpy(aux,p+i\*COL);

strcpy(p+i\*COL,p+j\*COL);

strcpy(p+j\*COL,aux);

nnota=\*(q+i);

\*(q+i)=\*(q+j);

\*(q+i)=nnota;

}

}

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

printf("\n%-10s%d",p+i\*COL,\*(q+i));

}

## Desarrollo del ejercicio tipo Usando funciones

Vamos a realizar el ejercicio usando funciones. En principio vamos a usar 4 funciones

1. Función para cargar los datos ......... Cargar
2. Función para buscar un elemento......Buscar
3. Función para ordenar........................Ordenar
4. Función para buscar el menor...........Busca\_menor

Ya sabemos cuantas funciones se van a usar y cual es la tarea que ellas van a realizar, por lo tanto queda analizar cuales son los valores que debemos pasarles y que valores van a retornar. **Tenga en cuenta que antes de empezar a escribir el programa tengo que saber como es el prototipo de las funciones y para ello se debe hacer un breve análisis de la función**. Comencemos por la función Cargar

#### Función cargar

La tarea de la función es cargar el vector y la matriz. Para que ella pueda cargar los datos necesita saber *DONDE CARGARLOS*, este es el dato que hay que proporcionarle a la función.

Como no hay variables globales la única forma de que la función tenga acceso a los datos es a través de un puntero, por lo tanto como debo cargar el vector y la matriz , necesito 2 punteros al comienzo de los elementos a cargar.

Cuando la función pide el ingreso de datos los carga directamente en el vector y la matriz por medio de los punteros, por lo tanto la función no devuelve nada. De acuerdo a esto el prototipo de la función será

**void**  *cargar* ( **int** \*q , **char** \**p* )

#### Función Buscar

El trabajo de la función es buscar un apellido para luego cambiarle la nota, por lo tanto se necesita conocer el nombre a buscar , la nueva nota y donde están guardadas las notas y nombres. En principio a la función le debo pasar donde estan guardados los datos para lo cual le voy a pasar 2 punteros, uno al vector y otro a la matriz.

El nombre a buscar y la nota la puedo leer desde dentro de la función o leerla desde main y pasarla como parámetro.

Como le pasamos punteros ,la modificación se hace directamente desde la función , por lo tanto no es necesario devolver nada.

El prototipo de la función será

**void**  *Buscar* ( **int** \**q* , **char** \*p )

#### Función Ordenar

El trabajo de la función Ordenar es justamente ordenar alfabéticamente los datos guardados en la matriz, pero como estos datos están vinculados con la nota en el vector , deberán ser cambiados también.

Entonces la función debe recibir como parámetros la dirección de comienzo de la matriz y el vector y no devuelve ningún dato.

**void**  *Ordenar* ( **int** \**q* , **char** \**p* )

#### Función Busca\_menor

La tarea de esta función es buscar la nota mas baja dentro del vector de notas. Para ello debe conocer donde esta el vector , por lo tanto hay que pasarle un puntero al comienzo del vector de notas. Cuando la función termina de recorrer el vector ya sabe cuál es la nota mas baja y en que lugar del vector la encontró, por lo tanto devuelve el índice del vector es decir el lugar en que encontró la nota mas baja.

**int**  *Busca\_menor* ( **int** \**q*)

Recién ahora estamos en condiciones de poder escribir el programa ya que conocemos que hace cada función y que se le debe pasar para que realice su trabajo.

El programa entonces es el siguiente.

#include <stdio.h>

#include <string.h>

#include <conio.h>

#include <ctype.h>

#define FIL 20

#define COL 25

void cargar(int \*, char \*);

void buscar(int \*, char \*);

void ordenar(int \*, char \*);

int busca\_menor(int \*);

void main(void)

{

int nota[FIL],i,j,min\_nota,imax=0,imin,max\_nota;

char nom[FIL][COL],rta;

clrscr();

cargar(nota,nom[0]);

printf("\nDesea cambiar alguna nota... S/N?");

rta=toupper(getche());

while(rta=='S')

{

buscar(nota,nom[0]);

printf("\nDesea cambiar otra nota... S/N?");

rta=toupper(getche());

}

max\_nota=nota[0];

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

if(max\_nota<nota[i])

{

max\_nota=nota[i];

imax=i;

}

imin=busca\_menor(nota);

printf("\nLa nota mas baja corresponde a %s",nom[imin]);

printf("\nLa nota mas alta corresponde a %s",nom[imax]);

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

printf("\n%-10s%d",nom[i],nota[i]);

}

void cargar(int \*q, char \*p)

{

int i;

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

{

printf("Ingrese nombre: ");

gets(p+i\*COL);

printf("Ingrese nota: ");

scanf("%d",q+i);

fflush(stdin);

}

}

void buscar(int \*q, char \*p)

{

char aux[COL],rta;

int i,nnota,flag=0;

printf("\nIngrese el nombre del alumno: ");

gets(aux);

printf("\nIngrese la nueva nota: ");

scanf("%d",&nnota);

fflush(stdin);

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

if((strcmp(aux,p+i\*COL))==0)

{

flag=1;

printf("\nLa nota del alumno %s es %d...la modifica S/N?",p+i\*COL,\*(q+i));

rta=toupper(getche());

if(rta=='S')

{

\*(q+i)=nnota;

printf("\nSe realizo el cambio");

}

else

printf("\nNo se acepto el cambio");

break;

}

if(!flag) /\* es lo mismo que if(flag==0) \*/

printf("\No se encontro el nombre %s",aux);

flag=0;

}

void ordenar(int \*q, char \*p)

{

char aux[COL];

int i,j,auxint;

for(i=0;i<FIL-1;i++)

for(j=i+1;j<FIL;j++)

{

if((strcmp(p+i\*COL,p+j\*COL))<0)

{

strcpy(aux,p+i\*COL);

strcpy(p+i\*COL,p+j\*COL);

strcpy(p+j\*COL,aux);

auxint=\*(q+i);

\*(q+i)=\*(q+j);

\*(q+i)=auxint;

}

}

}

int busca\_menor(int \*q)

{

int min\_nota,i,imin=0;

min\_nota=\*q;

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

if(min\_nota>\*(q+i))

{

min\_nota=\*(q+i);

imin=i;

}

return imin;

}

## Desarrollo del ejercicio usando estructuras

Dado que estamos trabajando con nombre y nota de varias personas , podemos realizar un agrupamiento distinto de los datos.

En lugar de guardar las notas en un vector y los nombres en una matriz , vamos a generar un tipo distinto de dato ,al que llamaremos ***estructura*** , que agrupará un nombre y una nota.

De acuerdo a esta nueva forma de agupar datos necesitamos definir un vector que contenga 20 estructuras.

Una estructura tendrá la siguiente forma

Estructura

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C | a | r | l | o | s | \0 |  |  |  |

Vector para un nombre

|  |
| --- |
| 5 |

Nota

Como ya sabemos una estructura es una variable que contiene datos de distinto tipo y que se referencian bajo un mismo nombre.

Imagine un Maletin, dentro de el encontramos elementos como lápices , cuadernos , carpetas , agenda, etc. ( equivalente a una estructura).

Son todos de distinto tipo pero estan dentro del maletin , cuando quiero sacar algo de adentro primero debo abrir el maletín.

Puedo tener muchos maletines con el mismo tipo de cosas adentro , pero de distintas personas , por lo tanto el contenido de las carpetas , agendas, etc , será distinto (vector de estructuras). El formato de la estructura con la que trabajaremos en este ejercicio sera la siguiente:

struct alumno { char nom [20 ];

int nota;

};

Una vez establecido el formato se debe definir la o las variables que utilicen dicho formato. Por ejemplo en main

struct alumno x;

Donde x es la variable del tipo ***struct alumno.*** Recuerde que siempre se debe acceder al campo de la estructura y que para ello se utiliza el operador punto.

x.nom accedo al vector del nombre

x.nota accedo a la variable nota.

El ejercicio desarrollado con estructuras queda :

#include <stdio.h>

#include <string.h>

#include <conio.h>

#define MAX 20

struct alumno {

char nombre[20];

int nota;

};

void main(void)

{

int i,j,max\_nota,min\_nota,imax=0,imin=0,nnota,flag=0;

char aux[20],rta;

struct alumno alu[MAX],auxstr;

clrscr();

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

{

printf("Ingrese nombre: ");

gets(alu[i].nombre);

printf("Ingrese nota: ");

scanf("%d",&alu[i].nota);

fflush(stdin);

}

printf("\nDesea cambiar alguna nota... S/N?");

rta=toupper(getche());

while(rta=='S')

{

printf("\nIngrese el nombre del alumno: ");

gets(aux);

printf("\nIngrese la nueva nota: ");

scanf("%d",&nnota);

fflush(stdin);

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

if((strcmp(aux,alu[i].nombre))==0)

{

flag=1;

printf("\nLa nota del alumno %s es %d...la modifica S/N?",alu[i].nombre,alu[i].nota);

rta=toupper(getche());

if(rta=='S')

{

alu[i].nota=nnota;

printf("\nSe realizo el cambio");

}

else

printf("\nNo se acepto el cambio");

break;

}

if(!flag) /\* es lo mismo que if(flag==0) \*/

printf("\No se encontro el nombre %s",aux);

flag=0;

printf("\nDesea cambiar otra nota... S/N?");

rta=toupper(getche());

}

max\_nota=alu[0].nota;

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

if(max\_nota<alu[i].nota)

{

max\_nota=alu[i].nota;

imax=i;

}

min\_nota=alu[0].nota;

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

if(min\_nota>alu[i].nota)

{

min\_nota=alu[i].nota;

imin=i;

}

printf("\nLa nota mas baja corresponde a %s",alu[imin].nombre);

printf("\nLa nota mas alta corresponde a %s",alu[imax].nombre);

for(i=0;i<MAX-1;i++)

for(j=i+1;j<MAX;j++)

{

if((strcmp(alu[i].nombre,alu[j].nombre))<0)

{

auxstr=alu[i];

alu[i]=alu[j];

alu[j]=auxstr;

}

}

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

printf("\n%-10s%d",alu[i].nombre,alu[i].nota);

}

## Desarrollo del ejercicio usando archivos

Para desarrollar el programa haremos unas modificaciones al enunciado. La cantidad de alumnos a ingresar es desconocida y no se hace el ordenamiento.

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <conio.h>

#define MAX 20

struct alumno{ char nom[MAX];

int nota;

};

void validastr(char \*, int);

void listar(FILE \*parch);

void main(void)

{

struct alumno aux;

int cant,flag;

char rta='s',nombre[MAX];

FILE \*arch;

if((arch=fopen("c:\\alumno.dat","rb+"))==NULL)

if((arch=fopen("c:\\alumno.dat","wb+"))==NULL)

{

printf("\nNo se puede abrir el archivo");

exit(1);

}

while(rta=='s')

{

printf("\nIngrese nombre: ");

validastr(&aux.nom,20);

printf("\nIngrese nota: ");

scanf("%d",&aux.nota);

fflush(stdin);

fwrite(&aux,sizeof(aux),1,arch);

printf("\nIngresa otro alumno?");

rta=tolower(getche());

}

rewind(arch);

listar(arch);

printf("\nDesea modificar alguna nota? s/n");

rta=tolower(getche());

while(rta=='s')

{

printf("\nIngrese nombre del alumno: ");

validastr(nombre,20);

rewind(arch);

flag=0;

while(!feof(arch))

{

cant=fread(&aux,sizeof(aux),1,arch);

if(cant!=1)

if(feof(arch))

break;

else

{

printf("\nNo se pudo leer el ultimo dato");

break;

}

if(!strcmp(aux.nom,nombre))

{

flag=1;

printf("\nLos datos del alumno son los siguientes");

printf("\nNombre: %s",aux.nom);

printf("\nEdad : %d\n",aux.nota);

printf("\nDesea cambiar la nota?");

if((tolower(getche()))=='s')

{

printf("\nIngrese la nota nueva: ");

scanf("%d",&aux.nota);

fseek(arch,(long)(-1)\*sizeof(aux),SEEK\_CUR);

fwrite(&aux,sizeof(aux),1,arch);

}

else

printf("\nLa nota no se ha cambiado");

}

}

if(!flag)

printf("\nEl nombre no se encontro");

printf("\nDesea modificar otra nota? s/n");

rta=tolower(getche());

}

rewind(arch);

listar(arch);

}

void validastr(char \*str, int colum)

{

char auxvec[50];

gets(auxvec);

while((strlen(auxvec))>colum-1)

{

printf("\nError , reingrese...maximo %d letras",colum-1);

gets(auxvec);

}

strcpy(str,auxvec);

}

void listar(FILE \*parch)

{

struct alumno aux;

int cant;

while(!feof(parch))

{

cant=fread(&aux,sizeof(aux),1,parch);

if(cant!=1)

if(feof(parch))

break;

else

{

printf("\nNo se pudo leer el ultimo dato");

break;

}

printf("\n----------------------------------------");

printf("\nNombre: %s",aux.nom);

printf("\nEdad : %d\n",aux.nota);

printf("\n----------------------------------------");

}

}

# 

# Anexo 1

## Función de validación de cadenas de caracteres

Cargar 20 nombres en una matriz para luego mostrarlos por pantalla. Utilizar una función para validar el texto que se ingresa.

Para resolver el problema comenzaremos escribiendo el código para realizar la validación pero sin usar funciones. El código es el siguiente

#include <stdio.h>

#include <string.h>

#define FIL 20

#define COL 10

void main(void)

{

char nombre[FIL][COL],aux[COL\*5];

int i,x;

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

{

printf("\nIngrese nombre: ");

gets(aux);

x=strlen(aux);

while(x>COL-1)

{

printf("\nError...Reingrese");

gets(aux);

x=strlen(aux);

}

strcpy(nombre[i],aux);

}

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

printf("\nNombre: %s ",nombre[i]);

getch();

}

Lo que se hace para realizar la validación es utilizar un vector suficientemente grande en el cuál se realiza la carga y se mide la cantidad de caracteres ingresados. Una vez obtenida la cantidad de letras se compara contra la máxima cantidad de letras que puede tener la matriz (cantidad de columnas menos una que la ocupará el terminador), mientras la cantidad de letras ingresadas supere a las columnas de la matriz se pide el reingreso sobre el vector auxiliar. Al terminar el while significa que el texto que se encuentra en el vector auxiliar cabe en la matriz , por lo tanto solo resta copiarlo en la fila que corresponda.

Ahora , en base al código escrito vamos a realizar la función.

En principio podemos pensar en llevar a la función todo el for de la carga , pero esta alternativa solo nos permitiría usar la función de validación para cargar F nombres. Lo lógico sería que esta función pueda servir para validar y cargar cualquier cadena de caracteres , ya sea un solo nombre (en un vector) o muchos (en una matriz). En el caso de la matriz simplemente se llamaría a la función dentro de un for para cada una de las filas de la matriz ( piense que gets carga solo una cadena y en una matriz se carga fila por fila).

Entonces pensemos que la función lea el nombre , realice el while para la validación y finalmente haga la copia en la fila de la matriz.

Para poder hacer esto , necesito conocer en que fila de la matriz debo realizar la carga y aparte donde comienza el vector auxiliar , por eso es que se deben pasar como parámetros a la función dos punteros a caracter , uno que apunte a la fila de la matriz y el otro al vector auxiliar. Entonces el código de nuestra función será:

#include <stdio.h>

#include <string.h>

#define FIL 4

#define COL 5

void valida\_str(char \*, char \*);

void main(void)

{

char nombre[FIL][COL],aux[COL\*5];

int i,x;

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

{

printf("\nIngrese nombre: ");

valida\_str(nombre[i],aux);

}

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

printf("\nNombre: %s ",nombre[i]);

getch();

}

void valida\_str(char \*pdestino,char \*paux)

{

int x;

gets(paux);

x=strlen(paux);

while(x>COL-1)

{

printf("\nError...Reingrese");

gets(paux);

x=strlen(paux);

}

strcpy(pdestino,paux);

}

Si analizamos la función vemos que se carga a través del puntero el vector auxiliar que se encuentra en main , y terminada la validación el texto correcto está cargado en el vector auxiliar , y por medio de strcpy se carga en la fila de la matriz indicada por el puntero pdestino. Mirando bien la función podemos decir que uno de los parámetros que se le pasa , está sobrando ya que la copia se podría hacer directamente desde main.

De acuerdo a esto se puede hacer una modificación al programa ( se pasa un solo puntero y la copia se hace en main).

#include <stdio.h>

#include <string.h>

#define FIL 4

#define COL 5

void valida\_str(char \*);

void main(void)

{

char nombre[FIL][COL],aux[COL\*5];

int i,x;

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

{

printf("\nIngrese nombre: ");

valida\_str(aux);

strcpy(nombre[i],aux);

}

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

printf("\nNombre: %s ",nombre[i]);

getch();

}

void valida\_str(char \*paux)

{

gets(paux);

while(COL-1<strlen(paux))

{

printf("\nError...Reingrese");

gets(paux);

}

}

En principio podríamos decir que la función escrita arriba es la definitiva , pero haciendo un poco mas de análisis se puede mejorar.

La idea de hacer funciones , entre otras cosas , es que sean de aplicación genérica y que hagan una tarea particular , con lo cual el código de la función no debería necesitar de demasiados agregados en la llamada para su funcionamiento. Si nos fijamos en el programa necesitamos agregar 2 cosas para poder usar nuestra función. La primera es el agregado del vector auxiliar y la segunda es la utilización de strcpy.

Si hacemos una pequeña modificación a la función podemos usarla sin necesidad de agregar líneas de código que complementen la tarea de la función.

Pensemos lo siguiente: la función realiza la validación haciendo uso de un vector auxiliar , por lo tanto sería lógico que ese vector lo tenga la función y no el main.

Cuando usamos la función gets le pasamos como parámetro la dirección donde debe guardar el texto que lee , de esta misma forma podemos pensar que a la función de validación le pasamos como parámetro la dirección donde debe guardar el texto validado.

Con estas 2 modificaciones llegamos a la función final , que como veremos es un reemplazo directo de gets con el adicional que aparte de cargar hace la validación.

El programa final queda de la siguiente forma

#include <stdio.h>

#include <string.h>

#define FIL 4

#define COL 5

void valida\_str(char \*);

void main(void)

{

char nombre[FIL][COL];

int i,x;

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

{

printf("\nIngrese nombre: ");

valida\_str(nombre[i]);

}

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

printf("\nNombre: %s ",nombre[i]);

getch();

}

void valida\_str(char \*pdest)

{

char aux[COL\*5];

gets(aux);

while(COL-1<strlen(aux))

{

printf("\nError...Reingrese");

gets(aux);

}

strcpy(pdest,aux);

}

Observe que cuando se llama a la función se le pasa la dirección donde debe guardar el texto.

Este nuevo modelo de la función no necesita que se le agregue nada en la llamada.