***TP N° 8: Estructuras y Uniones***

***Ejercicio 1***

Se agregan comentarios a cada código

| char c = 'a';  char \*p = malloc(sizeof(char)); **// Reservamos memoria en el heap**  p = &c; **// apuntamos a c, la memoria reservada no se usa ni se libera** |
| --- |

1. No es eficiente usar calloc, ya que debe recorrer la zona reservada y llenarla con ceros. Sólo se debe usar si toda (o la mayor parte) de la zona debe inicializarse con cero, caso contrario usar malloc

| struct art {  int code;  double price;  }  struct art \* newArt(int code, double price) {  struct art \* p = calloc(1, sizeof(struct art));  p->code = code;  p->price = price;  return p;  } |
| --- |

| int \* p = malloc(n \* sizeof(int));  int \* q = p;  /\* llenamos y usamos el vector \*/  free(p); **// liberamos la zona en el heap**  free(q); **// q apunta a la misma zona que ya se liberó** |
| --- |

1. Es equivalente al punto anterior, mismo error. El segundo free dará error y aborta

| int \* p = malloc(n \* sizeof(int));  int \* q = p;  /\* llenamos y usamos el vector \*/  free(q);  free(p); |
| --- |

1. Fragmento de código para buscar secuencias de repetidos en un vector

| for (int i=0; i<dim ; i++){  if(vec[i] == vec[i+1]) { **// cuando i==dim-1 accede fuera del vector** |
| --- |

| struct art {  int code;  double price;  }  struct art \* newArt(int code, double price) {  struct art p;  p.code = code;  p.price = price;  return &p; **// retornamos una dirección en el stack, no el heap**  } |
| --- |

1. Algunos compiladores inicializan las variables en cero o NULL si son punteros, pero otros no. El puntero p tiene un valor "basura" y no NULL o una zona ya reservada en el heap, por lo que seguramente haga abortar el programa. Dos posibles soluciones

| int \* p = NULL;  p = realloc(p, cant \* sizeof(int)); |
| --- |

| int \* p;  p = malloc(cant \* sizeof(int)); |
| --- |

***Ejercicio 2***

1. Falso: una estructura puede contener diferentes tipos de datos.
2. Falso: las uniones no pueden ser comparadas.
3. Falso: la palabra reservada typedef se utiliza para definir nuevos nombres (sinónimos) para tipos de datos definidos previamente.
4. Verdadero: debido a los problemas de alineamiento.
5. Falso: No necesariamente, depende del tipo de los campos.
6. Falso: La palabra typedef sirve para definir nuevos nombres de tipos, no variables, por lo tanto no reserva memoria.

***Ejercicio 3***

1. El asterisco no corresponde. Al aplicarle a **cPtr** el operador **->,** ya se está accediendo al dato.
2. El subíndice del arreglo ha sido omitido. La expresión debería ser: **corazones[10].figura**
3. Se requiere de un punto y coma luego de la llave que cierra la estructura.
4. La palabra reservada **struct** fue omitida en la declaración de variable. Para utilizar la declaración sin modificaciones se debería definir la estructura de la siguiente forma:

**typedef struct**

**{**

**char apellido[15];**

**char nombre[15];**

**int edad;**

**} persona;**

e) El fragmento del cuadro de la izquierda no compila, porque trata de cambiar v, que es una constante, un rótulo, no tiene valor izquierdo.

***Ejercicio 4***

Este ejercicio corresponde a un juego de poker acotado. Requiere repartir 5 cartas de un mazo y analizarlas para detectar si se obtuvo alguna de las combinaciones ganadoras (par, pierna o poker).

La administración de un mazo de cartas puede ser genérica independientemente de las reglas del juego en el cual el mazo sea usado. Por lo tanto este problema se puede dividir en dos partes, una de ellas consiste en administrar un mazo de cartas en forma genérica y la otra corresponde al análisis de las reglas del poker específicamente.

**a)** En esta parte se pide tomar las 5 primeras cartas del mazo y analizar el juego obtenido. Por lo tanto diseñamos un módulo de cartas que permite crear el mazo y mezclar las cartas y un programa que utilizando dicho módulo toma las 5 primeras cartas del mazo y analiza cuál es el resultado obtenido:

El header del módulo de cartas (***cartas1.h***) es el siguiente:

#define CANT\_FIGURAS 13

#define CANT\_PALOS 4

#define CARTAS\_MAZO CANT\_FIGURAS \* CANT\_PALOS

#define CARTAS\_JUEGO 5

typedef struct carta

{

int numero;

int palo;

} tipoCarta;

*/\* Arma un arreglo que representa un mazo de cartas francesas \*/*

void llenarMazo(tipoCarta mazo[]);

*/\* Intercambia en forma aleatoria los elementos del arreglo mazo \*/*

void mezclarMazo(tipoCarta mazo[]);

La estructura **tipoCarta** contiene dos enteros que representan el número o figura de la carta y a su vez el mazo será un arreglo de cartas.

La implementación del módulo es la siguiente:

#include <stdlib.h>

#include <time.h>

#include "cartas1.h"

void

llenarMazo(tipoCarta mazo[])

{

int i;

for ( i = 0; i < CARTAS\_MAZO; i++)

{

mazo[i].numero = i%CANT\_FIGURAS;

mazo[i].palo = i/CANT\_FIGURAS;

}

return;

}

void

mezclarMazo(tipoCarta mazo[])

{

int i, posicion;

tipoCarta auxCarta;

srand(time(NULL));

for(i=0; i<CARTAS\_MAZO; i++)

{

auxCarta = mazo[i];

posicion = rand() % CARTAS\_MAZO;

mazo[i] = mazo[posicion];

mazo[posicion] = auxCarta;

}

return;

}

Luego escribimos un programa que utilice este módulo:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "cartas1.h"

typedef enum {NADA=1, PAR, PIERNA, POKER} tipoConfiguraciones;

*/\* Devuelve el valor de la configuración correspondiente al arreglo juego \*/*

void analizar(const tipoCarta juego[]);

int

main(void)

{

tipoCarta mazo[CARTAS\_MAZO];

llenarMazo(mazo);

mezclarMazo(mazo);

analizar(mazo);

return 0;

}

void

analizar(const tipoCarta juego[])

{

int i,j;

char \*cFiguras[] = {"Dos", "Tres", "Cuatro", "Cinco", "Seis",

"Siete", "Ocho", "Nueve", "Diez", "J", "Q", "K", "As"};

char \*cPalos[] = {"Diamante", "Trebol", "Corazon", "Pique"};

char \*cResultados[] = {"Nada", "Par", "Pierna", "Poker"};

tipoConfiguraciones total=NADA, parcial;

for (i=0; i<CARTAS\_JUEGO-1; i++)

{

parcial=NADA;

for(j=i+1; j<CARTAS\_JUEGO; j++)

if (juego[i].numero == juego[j].numero)

parcial++;

if (total < parcial)

total = parcial;

}

printf("\nSu juego es: \n");

for (i=0; i<CARTAS\_JUEGO; i++)

printf("%s %s\n", cFiguras[juego[i].numero], cPalos[juego[i].palo]);

printf("El resultado es: %s\n", cResultados[total-1]);

}

El programa que presentamos es aceptable pero no es la mejor alternativa. Como primer defecto podemos destacar que aunque existe un módulo de cartas que realiza parte del procesamiento (back-end), la función ***analizar*** también procesa la información y además interactúa con el usuario, por lo cual en esa función el front-end y el back-end están mezclados. Lo correcto sería que la función ***analizar*** retorne el resultado obtenido y otra función se encargue de realizar la impresión.

Además, esta función no es eficiente ya que revisa siempre las cinco cartas, aún cuando las tres primeras sean una PIERNA o las cuatro primeras sean POKER.

Si el jugador tiene dos pares la función selecciona el primero que encuentra en lugar de seleccionar el de mayor figura.

Por otra parte el módulo ***cartas1***, que también es aceptable, no es realmente independiente del programa por varios motivos:

* No existe una función dentro del módulo que “reparta” cartas, sino que el usuario, si es prolijo, toma del mazo las primeras y si no lo es puede tomar cualquiera ya que tiene acceso total al mismo.
* No existe un tipo de dato ***tipoMazo***, sino que existe el tipo ***tipoCarta*** y el usuario es el encargado de armar con dicho tipo un arreglo. Nuevamente si el usuario no es prolijo puede definir el arreglo con menos componentes que las necesarias y por lo tanto el módulo de cartas accede a zonas de memoria inválida.
* Si se necesita repartir cartas para más de un jugador, es el usuario del módulo quien debe recordar que las primeras cartas ya fueron entregadas a otro jugador y por lo tanto no debe repartir desde el comienzo del mazo.
* El header no tiene la verificación para múltiple inclusión (#ifndef / #define)

Por los motivos mencionados presentamos un nuevo diseño corregido en el módulo ***cartas2*** (cartas2.h y cartas2.c), aunque recalcamos que la primera versión es aceptable a los fines de esta práctica:

#ifndef \_\_CARTAS2\_H\_

#define \_\_CARTAS2\_H\_

#define CANT\_FIGURAS 13

#define CANT\_PALOS 4

#define CARTAS\_MAZO CANT\_FIGURAS \* CANT\_PALOS

*/\* Estructura que representa una carta \*/*

typedef struct carta

{

int numero;

int palo;

} tipoCarta;

*/\* La estructura tipoMazo encapsula el arreglo de cartas junto con un subíndice*

*\*\* (inicialmente en 0) que indica la primera carta disponible del mazo*

*\*/*

typedef struct

{

tipoCarta cartas[CARTAS\_MAZO];

int primeraCarta;

}tipoMazo;

*/\* Crea un mazo de cartas (asignando un valor a cada carta y colocando el*

*\*\* índice del mazo en cero) y lo retorna en su nombre*

*\*/*

tipoMazo armarMazo(void);

*/\* Intercambia en forma aleatoria las cartas del mazo \*/*

void mezclarMazo(tipoMazo \* pMazo);

*/\* Elimina del mazo las primeras "cant" cartas y las asigna en el arreglo juego*

*\*\* Retorna en su nombre la cantidad de cartas que pudo repartir realmente*

*\*/*

int repartir(tipoMazo \* pMazo, int cant, tipoCarta juego[]);

*/\* Retorna la cantidad de cartas disponibles en el mazo \*/*

int cantCartas(tipoMazo \* pMazo);

#endif

La implementación del mismo es la siguiente:

#include <stdlib.h>

#include <time.h>

#include "cartas2.h"

tipoMazo

armarMazo(void)

{

int i;

tipoMazo mazo;

mazo.primeraCarta = 0;

for ( i = 0; i < CARTAS\_MAZO; i++)

{

mazo.cartas[i].numero = i % CANT\_FIGURAS;

mazo.cartas[i].palo = i / CANT\_FIGURAS;

}

return mazo;

}

void

mezclarMazo(tipoMazo \* pMazo)

{

int i, posicion;

tipoCarta auxiliar;

srand(time(NULL));

*/\* Para realizar la mezcla se reemplaza cada carta por*

*\*\* otra elegida al azar del resto del mazo*

*\*/*

for(i=pMazo->primeraCarta; i<CARTAS\_MAZO; i++)

{

auxiliar = pMazo->cartas[i];

posicion = rand() % CARTAS\_MAZO;

pMazo->cartas[i] = pMazo->cartas[posicion];

pMazo->cartas[posicion] = auxiliar;

}

return;

}

int

repartir(tipoMazo \* pMazo, int cant, tipoCarta juego[])

{

int i;

*/\* Asigno las primeras cartas disponibles del mazo al jugador \*/*

for(i=0;i<cant && pMazo->primeraCarta < CARTAS\_MAZO; i++)

juego[i] = pMazo->cartas[pMazo->primeraCarta++];

return i;

}

int

cantCartas(tipoMazo \* pMazo)

{

return CARTAS\_MAZO - pMazo->primeraCarta;

}

El programa que resuelve el punto **a)** utilizando este módulo es el siguiente:

#include <stdio.h>

#include "cartas2.h"

#define CARTAS\_JUEGO 5

*/\* este tipo permite definir el juego obtenido por un jugador \*/*

typedef enum {NADA=1, PAR, PIERNA, POKER} tipoJugada;

typedef struct {

tipoJugada juego;

int figura ;

} tipoConfiguraciones ;

*/\* Devuelve el valor de la configuración encontrada en el arreglo juego \*/*

tipoConfiguraciones analizar(const tipoCarta juego[]);

*/\* Imprime el conjunto de cartas y la configuración obtenida \*/*

void imprimirJuego(const tipoCarta juego[], tipoConfiguraciones resultado);

int

main(void)

{

tipoMazo mazo;

tipoCarta juego[CARTAS\_JUEGO];

tipoConfiguraciones resultado;

mazo = **armarMazo**();

**mezclarMazo**(&mazo);

**repartir**(&mazo, CARTAS\_JUEGO, juego);

resultado = analizar(juego);

imprimirJuego(juego,resultado);

return 0;

}

tipoConfiguracionesanalizar(const tipoCarta juego[])

{

int i,j;

tipoConfiguraciones total ={NADA,0}, parcial;

for (i=0; i<CARTAS\_JUEGO-1 && total.juego <= PAR; i++)

{

parcial.juego=NADA;

parcial.figura=0;

for(j=i+1; j<CARTAS\_JUEGO; j++)

if (juego[i].numero == juego[j].numero)

{

parcial.juego++;

parcial.figura = juego[j].numero;

}

*/\* Si obtengo un juego de mayor valor que el que tengo o uno del mismo valor pero*

*\*\* con una figura más alta lo guardo*

*\*/*

if (total.juego < parcial.juego ||

(total.juego == parcial.juego && total.figura < parcial.figura))

total = parcial;

}

return total;

}

void

imprimirJuego(const tipoCarta juego[], tipoConfiguraciones resultado)

{

int i;

static char \*cFiguras[] = {"Dos", "Tres", "Cuatro", "Cinco", "Seis",

"Siete", "Ocho", "Nueve", "Diez", "J", "Q", "K", "As"};

static char \*cPalos[] = {"Diamante", "Trebol", "Corazon", "Pique"};

static char \*cResultados[] = {"Nada", "Par", "Pierna", "Poker"};

printf("\nSu juego es: \n");

for (i=0; i<CARTAS\_JUEGO; i++)

printf("%s %s\n", cFiguras[juego[i].numero], cPalos[juego[i].palo]);

printf("\nEl resultado es: %s", cResultados[resultado.juego-1]);

if (resultado.juego != NADA)

printf(" de %s", cFiguras[resultado.figura]);

putchar('\n');

}

Ventajas de esta implementación:

* El módulo de cartas administra el mazo en forma independiente y por ser genérico puede ser utilizado en otros juegos de cartas. Este módulo se ajusta a mazos de 52 cartas (francesas), para utilizar mazos de 48 cartas (españolas) se deben alterar las constantes y recompilar.
* Toda la interacción con el usuario la realiza la función ***imprimirJuego*** (front-end ) que es independiente del procesamiento.
* Se agrega a la función ***analizar*** la posibilidad de detectar la existencia de dos pares de cartas iguales, seleccionando la de mayor valor.
* Por lo dicho en el punto anterior se puede informar al usuario qué configuración obtuvo y con qué figura.
* Mejora la eficiencia de la función ***analizar*** cortando el ciclo cuando se detecta PIERNA o POKER.

**b)** En este punto se extiende el programa para dos jugadores (usuario y computadora), jugando varias manos y contabilizando los puntos correspondientes. Al analizar este nuevo problema surge que la diferencia entre uno y dos jugadores es simplemente la repetición de instrucciones (se reparte dos veces, se analizan dos juegos, se imprimen dos resultados).

Si no se realiza este análisis es probable que se caiga en la típica solución (incorrecta) de mantener todas las variables duplicadas. (Ej: jugador1, jugador2, puntaje1, puntaje2, etc). Esto es obviamente indeseable ya que de agregarse un nuevo jugador aparecería un conjunto extra de variables (jugador3, puntaje3, etc.). Es muy simple concluir que ***no hay diferencia entre dos jugadores y N jugadores***, por lo tanto el programa que presentamos a continuación fue diseñado e implementado para N jugadores (que obviamente incluye la versión de dos).

Cada jugador está representado por una estructura que contiene las cartas de la mano en curso, la configuración que se obtiene con dichas cartas y el puntaje que lleva acumulado hasta el momento.

Se destaca nuevamente la división entre el front-end y el back-end, observando que resultaría simple escribir un programa visual que utilice todas las funciones de back-end implementadas (nos referimos a las funciones del programa y a las funciones del módulo ***cartas2***).

Además es importante notar que **ninguna función** supera las 20 líneas de código.

#include <stdio.h>

#include "cartas2.h"

#include "getnum.h"

#define CANT\_JUGADORES 2

#define CARTAS\_JUEGO 5

*/\* Define la configuración obtenida por un jugador \*/*

typedef enum {NADA=1, PAR, PIERNA, POKER} tipoJugada;

typedef struct {

tipoJugada juego;

int figura ;

} tipoConfiguraciones ;

*/\* Toda la información referida a un jugador \*/*

typedef struct {

tipoCarta juego[CARTAS\_JUEGO];

tipoConfiguraciones resultado;

int puntaje;

} tipoJugador;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FUNCIONES DE BACK - END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*/\* Se encarga de crear el mazo, mezclarlo y colocar los puntajes de los jugadores en cero \*/*

void inicializarJuego( tipoMazo \* mazo, tipoJugador \* jugadores);

*/\* Reparte 5 cartas a cada jugar y analiza que configuración obtuvo cada uno \*/*

void jugarMano(tipoMazo \* mazo, tipoJugador \* jugadores);

*/\* Devuelve el valor de la configuración encontrada en un juego \*/*

tipoConfiguraciones analizar(const tipoCarta juego[]);

*/\* Retorna el índice del jugador con mayor config., si hay empate retorna el primero de ellos \*/*

int buscarGanador(const tipoJugador jugadores[]);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FUNCIONES DE FRONT - END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*/\* Imprime las cartas de todos los jugadores indicando que config. obtuvo cada uno \*/*

void imprimirMano(const tipoJugador jugadores[]);

/\* Imprime las cartas y la configuración obtenida por un jugador\*/

void imprimirJuego(const tipoCarta juego[], tipoConfiguraciones resultado);

*/\* Imprime el o los jugadores con mayor configuración en una mano \*/*

void imprimirGanadores(const tipoJugador jugadores[], int ganador);

*/\* Imprime el resultado final del juego \*/*

void imprimirFinal(const tipoJugador jugadores[]);

int

main(void)

{

tipoMazo mazo;

tipoJugador jugadores[CANT\_JUGADORES];

int ganador;

inicializarJuego(&mazo, jugadores);

do

{

jugarMano(&mazo, jugadores);

ganador = buscarGanador(jugadores);

imprimirMano(jugadores);

*/\* si el indicado como ganador tiene alguna configuración \*/*

if (jugadores[ganador].resultado.juego != NADA )

imprimirGanadores(jugadores, ganador);

}while( cantCartas(&mazo)>= CARTAS\_JUEGO \* CANT\_JUGADORES

&& yesNo("Desea continuar el juego ?") == 1);

imprimirFinal(jugadores);

return 0;

}

void

inicializarJuego( tipoMazo \* mazo, tipoJugador \* jugadores)

{

memset(jugadores, 0, sizeof(tipoJugador) \* CANT\_JUGADORES);

\*mazo = armarMazo();

mezclarMazo(mazo);

}

void

jugarMano(tipoMazo \* mazo, tipoJugador \* jugadores)

{

int i;

*/\* se reparten cartas y se analiza lo obtenido por cada jugador \*/*

for (i=0; i < CANT\_JUGADORES; i++)

{

repartir(mazo, CARTAS\_JUEGO, jugadores[i].juego);

jugadores[i].resultado = analizar(jugadores[i].juego);

}

}

int

buscarGanador(const tipoJugador jugadores[])

{

int i, ganador = 0; */\* Se postula como ganador al primero \*/*

for (i=1; i < CANT\_JUGADORES; i++)

*/\* Si el jugador actual tiene mayor cantidad de cartas iguales o tiene la misma cantidad*

*\*\* pero de un número mayor, se lo marca como ganador*

*\*/*

if (jugadores[i].resultado.juego > jugadores[ganador].resultado.juego ||

(jugadores[i].resultado.juego == jugadores[ganador].resultado.juego &&

jugadores[i].resultado.figura > jugadores[ganador].resultado.figura) )

ganador = i;

return ganador;

}

tipoConfiguraciones

analizar(const tipoCarta juego[])

{

int i,j;

tipoConfiguraciones total ={NADA,0}, parcial;

for (i=0; i<CARTAS\_JUEGO-1 && total.juego <= PAR; i++)

{

parcial.juego=NADA;

parcial.figura=0;

for(j=i+1; j<CARTAS\_JUEGO; j++)

if (juego[i].numero == juego[j].numero)

{

parcial.juego++;

parcial.figura = juego[j].numero;

}

*/\* Si se encuentra un juego de mayor valor al que tenía o uno del mismo valor pero*

*\*\* con una figura mas alta, se guarda*

*\*/*

if (total.juego < parcial.juego ||

(total.juego == parcial.juego && total.figura < parcial.figura))

total = parcial;

}

return total;

}

void

imprimirMano(const tipoJugador jugadores[])

{

int i;

for (i=0; i<CANT\_JUGADORES; i++)

{

printf ("\nJugador %d\n",i+1);

imprimirJuego(jugadores[i].juego,jugadores[i].resultado);

}

}

void

imprimirJuego(const tipoCarta juego[], tipoConfiguraciones resultado)

{

int i;

static char \*cFiguras[] = {"Dos", "Tres", "Cuatro", "Cinco", "Seis",

"Siete", "Ocho", "Nueve", "Diez", "J", "Q", "K", "As"};

static char \*cPalos[] = {"Diamante", "Trebol", "Corazon", "Pique"};

static char \*cResultados[] = {"Nada", "Par", "Pierna", "Poker"};

for (i=0; i<CARTAS\_JUEGO; i++)

printf("%s %s\n", cFiguras[juego[i].numero], cPalos[juego[i].palo]);

printf("\nEl resultado es: %s", cResultados[resultado.juego-1]);

if (resultado.juego != NADA)

printf(" de %s", cFiguras[resultado.figura]);

putchar('\n');

}

void

imprimirGanadores(const tipoJugador jugadores[], int ganador)

{

int i;

*/\* el siguiente ciclo se hace para el caso en que haya empate \*/*

printf ("\nGanadores\n");

for (i=0; i < CANT\_JUGADORES; i++)

if(jugadores[i].resultado.juego==jugadores[ganador].resultado.juego

&& jugadores[i].resultado.figura==jugadores[ganador].resultado.figura)

{

printf ("Jugador %d\n",i+1);

jugadores[i].puntaje++;

}

}

void

imprimirFinal(const tipoJugador jugadores[])

{

int i;

printf("\nPUNTAJE FINAL:\n");

for(i=0; i< CANT\_JUGADORES; i++)

printf("Jugador %d = %d puntos\n", i+1, jugadores[i].puntaje);

}

***Ejercicio 7***

En la versión tp8ej7\_1.c se realiza una función buscaPalabra que invoca 8 veces a buscaPalabraEnDir, que recibe desde dónde buscar y cómo moverse.

Una variante para que la función buscaPalabra no repita código es almacenar los movimientos en un vector, respetando el orden del enum. En vez de “matriz” de 2 columnas podría también usarse un vector de struct con dos campos.

int

**buscaPalabra**(int i, int j, char \* palabra, char sopa[][COLS], int direcciones[])

{

static int mov[][2]={{1,0}, {-1,0}, {0,-1}, {0,1}, {-1,1},{-1,-1},

{1,1}, {1,-1}};

int count = 0;

for(int m=0; m < sizeof(mov) / sizeof(mov[0]); m++)

if(buscaPalabraEnDir(i, j, mov[m][0], mov[m][1], palabra, sopa))

direcciones[count++] = m;

return count;

}

***Ejercicio 8***

1. por la forma en que está definido, words es un vector estático de 21 punteros a char, por lo que no sirve, ya que no debe haber un límite para la cantidad de palabras
2. No compila, ya que no se puede definir un vector estático sin especificar la dimensión
3. Esta opción sirve. A medida que se necesiten más palabras se crearán más punteros. Cada palabra no estará limitada en su longitud, y para copiarla se deberá crear espacio, en el siguiente ejemplo mostramos cómo copiar una sola palabra:

char \* s = "...";

struct dictionary a;

a.words = malloc(sizeof(char \*)); // o sizeof(a.words[0])

a.words[0] = malloc(strlen(s)+1);

strcpy(a.words[0], s);

a.dim = 1;

1. es válido
2. también es válido
3. no sirve, ya que en words sólo se puede guardar una cadena de caracteres, y no varias palabras. Alguien podría pretender guardar las palabras una detrás de otra (20 caracteres y un cero, seguidos de otros 20 caracteres y un cero, etc.), donde la primer palabra está en words, la segunda en words+21, etc. Si bien se podría, es muy poco práctico
4. En este caso también sirve, ya que words es un vector dinámico de strings de hasta 20 caracteres (más el cero), si quisiéramos hacer lo mismo que en el punto c, los pasos a seguir son:

char \* s = "...";

struct dictionary a;

a.words = malloc(sizeof(Tword)); // o sizeof(a.words[0])

strcnpy(a.words[0], s,20); // usamos strncpy para asegurarnos de no pisar memoria

a.words[20]=0; // por si copió exacto 20 lugares, en ese caso no puso el cero

a.dim = 1;

1. En este caso sobra un asterisco

***Ejercicio 8***

La idea es poder obtener la dirección relativa del campo recibido, para ello usamos un puntero que valga NULL (cero). Al obtener la dirección del campo esa dirección será relativa a cero, y por lo tanto nos da el desplazamiento en bytes. Si bien a la vista estamos desreferenciado NULL, el compilador se da cuenta que de ese campo que desreferenciamos sólo queremos su dirección, no su contenido, por eso no accede a zona de memoria inválida. En otro compilador menos inteligente podría abortar. En ese caso habría que definir una variable de tipo struct y restar la dirección del campo menos la dirección de la estructura (la variable y la cuenta tienen que estar separadas por el operador coma).

| #include <stdio.h>  #define offset(st, field) ((unsigned long) &((st \*)NULL)->field)  struct coso {  char a;  int b;  char c[5];  };  int main(void) {  printf("%d\n", offset(struct coso, a));  printf("%d\n", offset(struct coso, b));  printf("%d\n", offset(struct coso, c));  return 0;  } |
| --- |