Structs y Listas Enlazadas

Ignacio Solar
ignacio.solar@usach.cl

20 de Abril, 2024

USACH

Estructura

de datos `struct`

```c
struct mi_struct
{
 int edad;
 char nombre;
 int alto;
};
```

- Tupla de variables
  - `mi_struct` almacena variables, los cuales son los **campos** del struct.
- No se pueden definir valores para los campos.
- Lo mas cercano a OOP o clases que hay en C.[1]

[1] OOP = *Object Oriented Programming*, es un paradigma que se basa en el concepto de objetos que contienen informacion y codigo. Esto se profundiza en el ramo de "Paradigmas de la programación".

El struct es la totalidad de mis variables

- Se usa como tipo de dato en una variable.
- Para asignar a alguna de ellas debo usar el operador `.`
    - Si es un puntero a la estructura, hay que usar el operador `->`
        - De igual forma se puede aplicar a cualquier tipo.
- Al acceder a una variable para asignar, se respeta el tipo de dato el cual es.

```
1     struct mi_struct *persona;
2     // variable persona ahora es del tipo de dato mi_struct
3     persona->edad = 25;
4     strcpy(persona->nombre, "Juan");
5     persona->alto = 180;
```

```
1     struct mi_struct persona = {.edad = 25, .alto = 180};
2     printf("%d %d", persona.edad,persona.alto); //25 180
```

Soy un perro 🐕🐕🐕

```c
struct Perro
{
    char nombre[50];
    char raza[50];
    int edad;
    float peso;
}
```

- Para ver las funciones de un perro, hay que recibirlo como argumento.

```c
void ladrar(struct Perro *perro);
void comer(struct Perro *perro, float cantidad);
void dormir(struct Perro *perro);
void presentarPerro(struct Perro *perro);
```

- Otra forma de asignar un struct, directamente como una lista. (no recomendable)

```c
struct Perro mi_perro = {"Max", "Labrador", 3, 25.5};
```

Creamos un perro

```
1  struct Perro* crearPerro(char *nombre,char *raza, int edad, float peso)
2  {
3      Perro *perro = (struct Perro*)malloc(sizeof(struct Perro));
4      //asignacion de memoria de tamaño perro
5      perro->nombre = nombre
6      perro->raza = raza
7      perro->edad = edad;
8      perro->peso = peso;
9      return perro;
10 }
11
12 Perro *mi_perro = crearPerro("Max", "Labrador", 3, 25.5);
```

```c
void ladrar(struct Perro *perro)
{
    printf("%s esta ladrando, woof wooof! 🐶", perro->nombre);
}
void comer(struct Perro *perro, float cantidad)
{
    perro->peso += cantidad;
    printf("%s esta comiendo y ahora pesa %.2f kg.\n",
     perro->nombre, perro->peso);
}
```

```c
void presentarPerro(struct Perro *perro)
{
    printf("Nombre: %s\n", perro->nombre);
    printf("Raza: %s\n", perro->raza);
    printf("Edad: %d años\n", perro->edad);
    printf("Peso: %.2f kg\n", perro->peso);
}
void dormir(struct Perro *perro)
{
    printf("%s esta durmiendo,💤💤💤", perro->nombre);
}
```

- La estructura de datos `Perro` encapsula la información relevante y nos permite tener todo ordenado.

Uso del struct

```
1 struct mi_struct persona;
```

- `struct mi_struct` es la declaración del tipo `persona`
    - Fácil perderse en el codigo .
    - Posibilmente se me olvide poner los 3 parametros.

Typedef al rescate

- Crea un alias para una estructura dada, puede ser declarado junto o después de la misma estructura.
- Sintaxis mas limpia 😎

```
typedef struct
{
  char nombre[50];
  char raza[50];
  int edad;
  float peso;
} Perro;
```

- struct sin nombre pero Alias `Perro` 🤔

```c
struct perro {
    char nombre[50];
    char raza[50];
    int edad;
    float peso;
};

typedef struct perro Perro;
```

- Ambas formas validas, ahora solo se usa `Perro` en vez de `struct Perro`

¿Struct dentro de otro struct?

- No problem 😎

```c
typedef struct {
  char nombre[50];
  int edad;
  struct Perro {
      char nombre[50];
      char raza[50];
      int edad;
      float peso;
  } mascota;
} Persona;
```

- Doble struct anónimo con alias definidos 🤯🤯🤯🤯

```c
  Persona mi_persona = {
      "John",
      30,
      {"Buddy", "Labrador", 3, 25.5}
  };
printf("Edad del perro: %d\n", mi_persona.mascota.edad); //💀💀
```

¿Struct recursivos?

- Nuevamente, No problem for C 😎😎

```c
typedef struct Nodo {
  int dato;
  Nodo* siguiente;
} Nodo;
```

¿Struct puntero dentro de otro Struct?

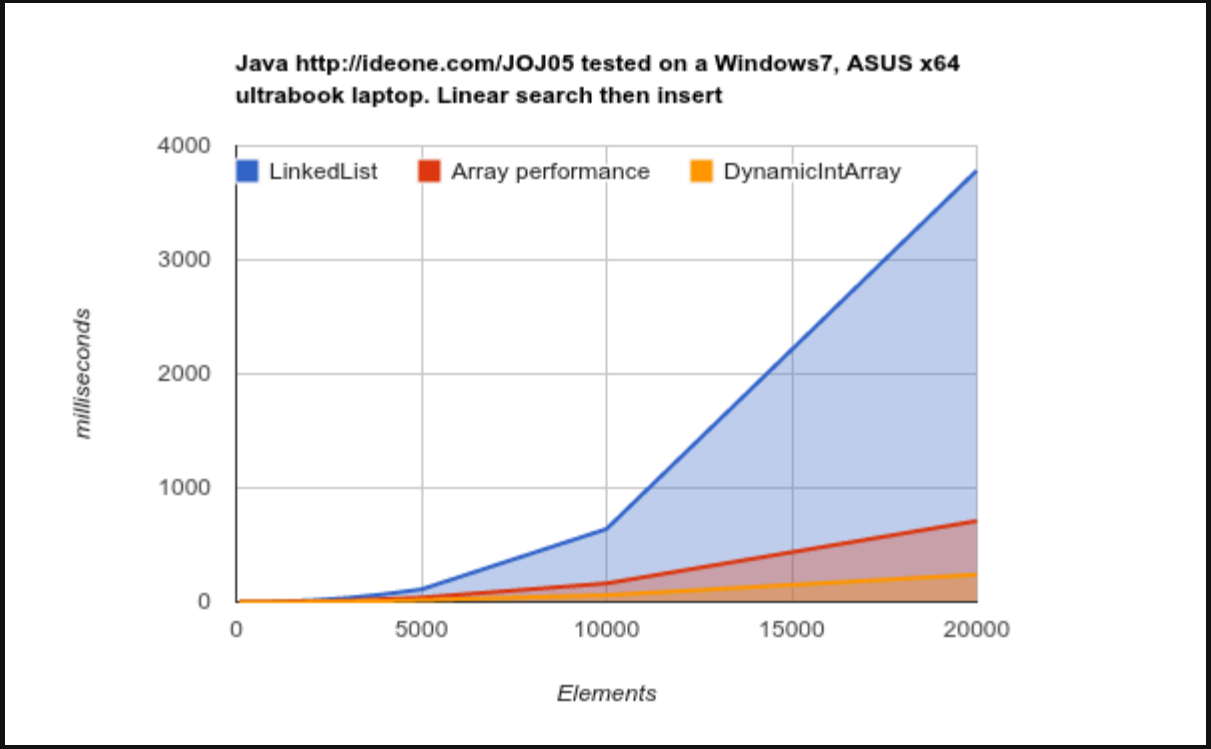
```c
typedef struct {
    char nombre[50];
    char raza[50];
    int edad;
    float peso;
} Perro;

typedef struct {
    char nombre[50];
    int edad;
    Perro* mascota;
} Persona;

int main() {
    Perro* mi_perro = (Perro*)malloc(sizeof(Perro));
    strcpy(mi_perro->nombre, "Buddy");
    strcpy(mi_perro->raza, "Labrador");
    mi_perro->edad = 3;
    mi_perro->peso = 25.5;
```

Struct puntero requiere asignación de memoria dinámica, no estática.

Listas Enlazadas

Orden lineal pero no via indices

- Se determina el orden en base a los punteros de cada objeto.
    - Representacion totalmente flexible, aunque no necesariamente eficiente.



Estructura

Valor y un puntero al siguiente Nodo.

```c
struct nodo {
  int info;
  struct nodo *Sig;
};
typedef struct nodo Nodo;

struct lista {
  Nodo *head;
};
typedef struct lista Lista;
```

`Lista` también es un nodo, es valido concadenar nodos en vez de tener `Lista`

```c
Lista *crearLista(){
    Lista *L=(Lista*)malloc(sizeof(Lista));
    L->head=NULL;
    return L;
}

Nodo *crear_nodo(int val){
    Nodo *nodo=(Nodo*)malloc(sizeof(Nodo));
    nodo->info=val;
    nodo->sig=NULL;
    return nodo;
}
```

Recorrer Listas Enlazadas

Ya que los elementos estan separados dentro de la memoria usar un iterador numerico **NO** es opcion.

- Si usamos la misma lista para recorrer, perderemos la lista.
- Hay que aprovechar que se guarda el siguiente nodo en nuestro nodo, creamos un auxiliar que recorre.

```
    Lista* cabeza = crearListaVacia();
    cabeza = insertarAlPrincipio(lista, 10);
    cabeza = insertarAlPrincipio(lista, 20);
    cabeza = insertarAlPrincipio(lista, 30);
```

```
void imprimirLista(Lista* cabeza) {
    Nodo* actual = cabeza; //auxiliar que recorre
    while (actual != NULL) {
        printf("Nodo: %p, Dato: %d, Siguiente: %p\n", actual, actual->dato, actual->siguiente);
        actual = actual->siguiente;// nos vamos al siguiente
    }
}
```

```
> ζ ./linkedlist.exe
Nodo: 000001765AEFF480, Dato: 30, Siguiente: 000001765AEFF460
Nodo: 000001765AEFF460, Dato: 20, Siguiente: 000001765AEFF520
Nodo: 000001765AEFF520, Dato: 10, Siguiente: 0000000000000000
```

Distintas formas validas de recorrer, siempre se requiere al menos 1 auxiliar.

```
nodo* aux = L->head
while(aux!= NULL){
    printf("%d ", aux->info)
    aux = aux->sig
    }
```

```
    Nodo *actual = L->head;
    do {
        printf("%d ", actual->info);
        actual = actual->sig;
    } while (actual != NULL);
```

```
    for (Nodo *actual = L->head; actual != NULL; actual = actual->sig) {
        printf("%d ", actual->info);
        }
```

Inserción

En la cabeza de la lista

1. Creo un nuevo nodo con el dato que quiero agregar
2. El siguiente del nodo nuevo sera el primero de la lista vieja.
3. Ahora el nuevo nodo es la cabeza de la Lista.

```c
void insertarInicio(Lista *L, int x) {
    Nodo *nuevoNodo = crear_nodo(x);
    nuevoNodo->sig = L->head;
    L->head = nuevoNodo;
}
```

En la cola de la lista

1. Como no puedo acceder al ultimo elemento, llego a el.
2. El siguiente del ultimo nodo sera mi nuevo nodo.

```c
void insertarFin(Lista *L, int x) {
    Nodo *nuevoNodo = crear_nodo(x);
    if (L->head == NULL) {
        L->head = nuevoNodo;
    } else {
        Nodo *actual = L->head;
        while (actual->sig != NULL) {
            actual = actual->sig;
        }
        actual->sig = nuevoNodo;
    }
}
```

¿Agregar después de cierto elemento/indice? 🤔🤔

```
1 void agregar_nodo(Lista *L, int valor, int posicion){....}
```

Eliminar elementos

El de la cabeza de la lista

```
1 void eliminarPrimero(Lista *L) {
2     if (L->head != NULL) {
3         Nodo *eliminar = L->head;
4         L->head = L->head->sig;
5         free(eliminar);
6     }
7 }
```

El ultimo de la lista

```
1  void eliminarUltimo(Lista *L) {
2      if (L->head != NULL) {
3          if (L->head->sig == NULL) {
4              free(L->head);
5              L->head = NULL;
6          } else {
7              Nodo *actual = L->head;
8              while (actual->sig->sig != NULL) {
9                  actual = actual->sig;
10             }
11             free(actual->sig);
12             actual->sig = NULL;
13         }
14     }
15 }
```

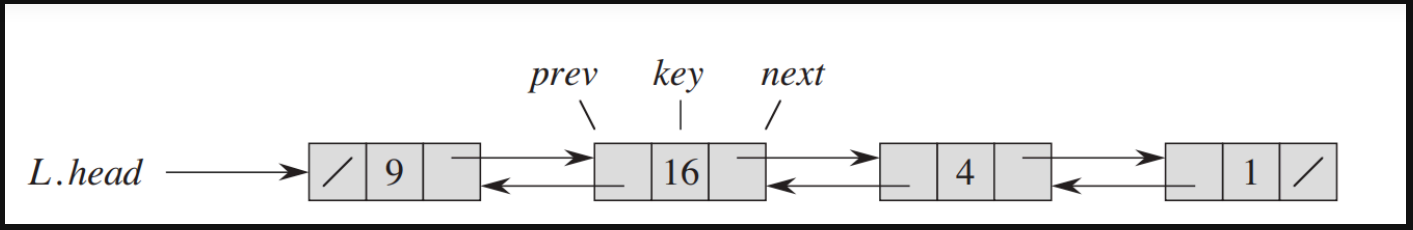
Ventajas

- Insertar o eliminar en cualquier posicion.
- **No requiere tamaño fijo de antemano**

Desventajas

- Se debe recorrer desde el inicio siempre.
- Mayor uso de memoria por uso de punteros `sizeof(int*) > sizeof (int)`
- No se puede acceder a elementos anteriores, tengo que partir de 0.
    - Listas doblemente enlazadas?

```c
typedef struct nodoDoble {
    int dato;
    struct nodoDoble *anterior;
    struct nodoDoble *siguiente;
} NodoDoble;
```



Referencias

- GCC Wiki
- CProgramming
- Introduction to Algorithms (2009).