Guía de C - avanzada

Arquitectura y Organización del Computador

Estructuras

Insanity consists of building major structures upon foundations which do not exist.

¿Qué es una estructura?

Una estructura es una **colección de variables** de distintos tipos, agrupadas bajo un mismo nombre.

Las estructuras son una forma de agrupar datos relacionados en un solo tipo de dato. Por ejemplo, si quisiéramos representar un **punto** en el espacio 3D, podríamos usar una estructura que contenga tres variables de tipo `float` (una para cada coordenada). Podríamos definir una estructura `Punto` de la siguiente manera:

Snippet 1:

```c
#include <stdio.h>

struct Punto {
    float x;
    float y;
    float z;
};
int main() {
    struct Punto p1;
    p1.x = 1.0;
    p1.y = 2.0;
    p1.z = 3.0;

    printf("Punto: (%f, %f, %f)\n", p1.x, p1.y, p1.z);
    return 0;
}
```

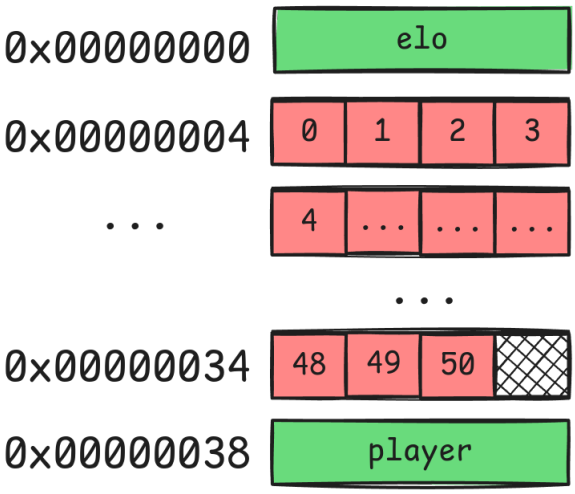
Podemos definir arrays dentro de las estructuras:

Snippet 2: struct

```c
#define NAME_LEN 50

struct {
  int elo;
  char name[NAME_LEN + 1];
  int ranking;
} player;
```

Esta estructura tiene tres campos: `elo` (un entero), `name` (una cadena de caracteres) y `ranking` (otro entero). Su layout en memoria sería algo así:



Struct en memoria

Podemos inicializar una estructura al momento de declararla, usando llaves:

Snippet 3: Inicialización de struct

```c
#define NAME_LEN 50

struct player{
  int elo;
  char name[NAME_LEN + 1];
  int ranking;
};

struct player player1 = { 2800, "Magnus Carlsen", 1 },
struct player player2 = { 2700, "Fabiano Caruana", 2 };
```

En este momento, es conveniente mencionar el keyword `typedef`, que nos permite definir un nuevo tipo de dato basado en un tipo existente. El nuevo tipo también se conoce por el nombre de **type alias**. Esto es útil para simplificar la declaración de estructuras y hacer el código más legible.

Snippet 4: typedef

```
typedef float real_t;
typedef int quantity_t;
typedef unsigned long int size_t; // en <stdint.h>
typedef uint32_t vaddr_t; // direccion virtual.
typedef uint32_t paddr_t; // direccion fisica.
```

Notar como primero viene el tipo original y luego el nuevo tipo.

En el siguiente fragmento definimos una estructura y le asignamos un alias `player_t` para poder usarlo más fácilmente en el resto del código. Además, notar el tipo de inicialización alternativa de `player3`:

Snippet 5: typedef de struct

```
#define NAME_LEN 50

typedef struct {
  int elo;
  char name[NAME_LEN + 1];
  int ranking;
} player_t;

player_t player1 = { 2800, "Magnus Carlsen", 1 },
player_t player2 = { 2700, "Fabiano Caruana", 2 };
player_t player3 = { .name = "Hikaru Nakamura",
                     .ranking = 3,
                     .elo = 2600}; //forma alternativa
```

Podemos acceder a los miembros de una estructura usando el operador `.` (punto) y tratar a esa variable como una variable normal. Por otro lado, la operación asignación `=` en `struct` produce una copia de la estructura completa.

Snippet 6: miembros

```
player_t magnus = { 2800, "Magnus Carlsen", 1 },
player_t faustino;

printf("Elo: %d\n", magnus.elo);
printf("Name: %s\n", magnus.name);
printf("Ranking: %d\n", magnus.ranking);

magnus.elo = 2700;
magnus.ranking--;

faustino = magnus; // copia de estructura
```

Las estructuras pueden pasarse a funciones como argumentos y también pueden ser devueltas como valores de retorno.

Como argumentos y valores de retorno

Snippet 7: struct como argumento y valor de retorno

```
player_t get_player(void) {
  player_t player = { 3000, "Bobby Fischer", 1 };
  return player;
}

void print_player(player_t player) {
  printf("Elo: %d\n", player.elo);
  printf("Name: %s\n", player.name);
  printf("Ranking: %d\n", player.ranking);
}
```

Notar que pasar estructuras grandes de esta manera puede ser ineficiente, ya que se copian todos los datos de la estructura. En su lugar, se suele usar punteros a estructuras, que son más eficientes. Lo veremos más adelante.

Nada impide que armemos un array de estructuras, como el siguiente:

Snippet 8: array de struct

```
typedef struct {
  char* pais;
  int code;
} dials_code_t;

dials_code_t country_codes[] = {
  {"Argentina", 54},
  {"Brasil", 55},
  {"Chile", 56},
  {"Uruguay", 598}
};

printf("Código para Argentina: %d\n", country_codes[0].code);
```

La inicialización puede ser parcial también, como en el siguiente ejemplo:

Snippet 9: Inicialización de array de struct

```
dials_code_t country_codes[100] = {
  [0] = {"Argentina", 54},
  [1] = {"Brasil", 55},
  [2] = {"Chile", 56},
  [3].pais = "Uruguay", [3].code = 598,
  // ... el resto se inicializa en 0
};
```

Ejercicio 1:

Definir una estructura `monstruo_t` que contenga los siguientes campos:

- `nombre` (string)
- `vida` (entero)
- `ataque` (entero)
- `defensa` (entero)

Luego, inicializar un array de monstruos y mostrar por pantalla el nombre y la vida de cada uno de ellos.

Ejercicio 2:

Definir una función `evolution` que reciba un `monstruo_t` y devuelva un nuevo monstruo con los mismos atributos, pero con el ataque y defensa aumentados en 10. Luego, usar esta función para evolucionar un monstruo y mostrar por pantalla sus atributos antes y después de la evolución.

Punteros

Paradox is a pointer telling you to look beyond it. If paradoxes bother you, that betrays your deep desire for absolutes. The relativist treats a paradox merely as interesting, perhaps amusing or even, dreadful thought, educational.

¿Qué es un puntero?

Un puntero es una **variable que almacena una dirección de memoria**.

En la guía anterior mencionamos al pasar que uno de los tipos nativos de C es el **puntero**, tipado `int*, char*, void*, etc`. Comencemos mirando el siguiente código:

Snippet 10:

```c
#include <stdio.h>

int main(){
    int x = 42;
    int *p = &x;

    printf("Direccion de x:  %p Valor: %d\n", (void*) &x, x);
    printf("Direccion de p: %p Valor: %p\n", (void*) &p, (void*) p);
    printf("Valor de lo que apunta p: %d\n", *p);
}
```

Ejercicio 3:

Sin correr el código, responder:

- ¿Cuál es la diferencia entre x y p? ¿Y entre `x` y `&x`? ¿Y entre `p` y `*p`?
- ¿Qué valores creen que se van a imprimir por terminal?

Ejercicio 4:

Compilar y ejecutar el código. ¿Qué valores se imprimieron? ¿Qué creen que significan?

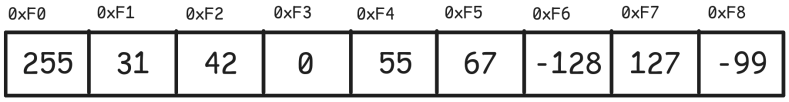
Para responder estas preguntas, es **clave** que entendamos como se organiza la memoria en C.

Al final del día, la memoria no es más que una tira de **bits** (1/0). Pero usualmente queremos trabajar con variables con más valores que solo 0 y 1[1], por lo que hoy en día usamos a la memoria como si fuera una tira de **bytes**, que son grupos de **8 bits**.

Entonces, desde ahora, pensemos a la memoria como una **tira de bytes contiguos**. Vamos a poder acceder a cada uno de estos bytes individualmente, porque cada uno tiene una **dirección única** que lo identifica.

[1] Tener memorias del tamaño que utilizamos hoy direccionables con granularidad de bit sería caro y dificultoso de implementar a nivel hardware. Este post de quora da una intuición al respecto https://www.quora.com/Why-do-memory-addresses-use-the-unit-of-byte-instead-of-bit.

Llamamos **puntero** a una variable que guarda la dirección de un valor en memoria. Por ejemplo, si la memoria de la dirección 0xF0 (hexadecimal) a 0xF8 tiene esta pinta:



Podríamos declarar las siguientes variables:

Snippet 11:

```c
#include <stdio.h>
#include <stdint.h>

int main(){
  uint8_t *x = (uint8_t*) 0xF0;
  int8_t *y = (int8_t*) 0xF6;

  printf("Dir de x: %p Valor: %d\n", (void*) x, *x);
  printf("Dir de y: %p Valor: %d\n", (void*) y, *y);

  //Devolverá:
  //  Dir de x: 0xF0 Valor: 255
  //  Dir de y: 0xF6 Valor: -128
}
```

Ejercicio 5:

¿Por qué x e y tienen distintos tipos? ¿Qué representan?

Si intentan ejecutar ese código, probablemente el programa falle con un error de tipo `Segmentation fault (core dumped)`. En nuestra computadora, la memoria esta siendo usada por varios componentes y es estadísticamente imposible que las direcciones 0xF0 a 0xF8 tengan los valores que usamos en este ejemplo. De hecho, las direcciones 0xF0 a 0xF8 de memoria probablemente no estén a nuestro alcance porque seguramente las esté usando otro proceso (por ende ocurre un `Segmentation Fault`).

Si queremos una tira contigua de bytes como la mostrada en la imagen, necesitamos declararla. Para eso, podemos usar arrays! Como vimos en la guía anterior.

Ejercicio 6:

Completar los ?? en el siguiente código:

Snippet 12:

```c
#include <stdio.h>
#include <stdint.h>

int main(){
    int8_t memoria[??] = ??;
    uint8_t *x = (uint8_t*) ??;
    int8_t *y = ??;

    printf("Dir de x: %p Valor: %d\n", (void*) x, *x);
    printf("Dir de y: %p Valor: %d\n", (void*) y, *y);
}
```

Pista: ¿Para qué sirve el operador `&`? Revisar los ejemplos de código anteriores o ver la sección Operando con punteros.

Operando con punteros

Habrán visto en los ejemplos hasta ahora que al trabajar con punteros contamos con dos operadores principales: `*` y `&`. Definamos propiamente:

- `*`: se usa en dos situaciones
  - Al **declarar** un puntero: indica que la variable siendo declarada es un puntero al tipo indicado. Por ejemplo, `int *num;` significa que `num` es un puntero a un número entero.
  - Cuando acompaña a una variable ya declarada: nos permite **desreferenciar** la variable que acompaña (que debería ser un puntero), permitiendonos acceder al valor que se encuentra en la dirección guardada en la variable.

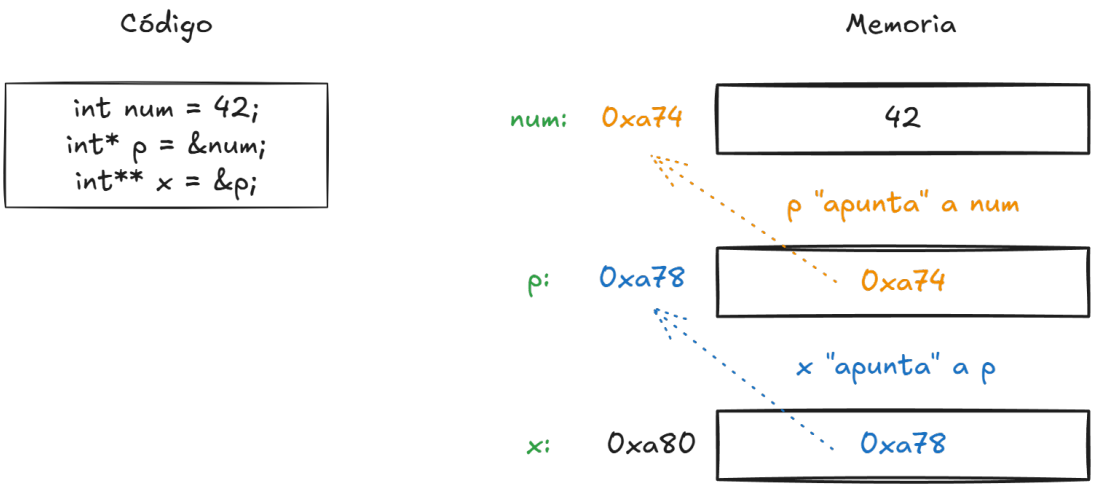
    Por ejemplo, en el Snippet 10, pueden ver que en el último printf, donde imprimimos el valor de lo que apunta p, para acceder a ese valor usamos `*p`. Efectivamente, ese printf resulta en que se imprima 42 a consola.
- `&`: se usa para obtener la dirección de una variable. En el Snippet 10, usamos `&x` para declarar p ya que queríamos que el puntero p guardara la dirección de la variable x (es decir, la dirección donde está guardado el valor 42).

Punteros a punteros

Hasta ahora solo usamos punteros a valores enteros, pero dado que un puntero no es más que la dirección de memoria de algún valor de algún tipo, podemos tener **punteros a cualquier cosa**. Incluso podemos tener punteros a punteros!

Algunos ejemplos:

- `char*`: puntero a una tira de `char`s. Si su último carácter es un `'\0'`, es equivalente a un string.
- `int**`: puntero (`int**`) a un puntero a un entero (`int**`)
- **void***: puntero a un tipo **desconocido**. No se puede desreferenciar sin antes especificar su tipo mediante un **casteo**.



Castear

Tipos de memoria

Tenemos varios tipos de memoria, y en cuál se encuentra nuestra variable dependerá de dónde y cómo declaramos nuestras variables.

Tipos de memoria	Scope	Lifetime
Global	El archivo completo	el tiempo de vida de la aplicación
Estática	La función (o bloque) donde está declarada	el tiempo de vida de la aplicación
Automática (local)	La función (o bloque) donde está declarada	Mientras la función (o bloque) esté en ejecución
Dinámica	Determinado por los punteros que referencian esta memoria	Hasta que la memoria sea liberada