Programación en C moderno

Álvaro Neira Ayuso <alvaro@soleta.eu>

c) Ejemplo 2: clase "gestor de tareas"

- Creación de objeto con malloc.
- Liberación de objetos con free
- Impresión por pantalla.
- Acceso a campos de las estructuras.
- Ejercicios.

Creación de objeto con malloc.

- Punteros y direcciones

  - Un puntero es una variable que contiene la dirección de un dato.

  - Existen dos tipos de operadores relacionados con memoria:

    - Operador valor * : A partir de una variable tipo puntero nos proporciona el dato apuntado.
    - Operador dirección &: A partir de una variable nos da la dirección de memoria donde se almacena dicha valor.

Como se declara un puntero

- tipo_de_dato *nombre_variable

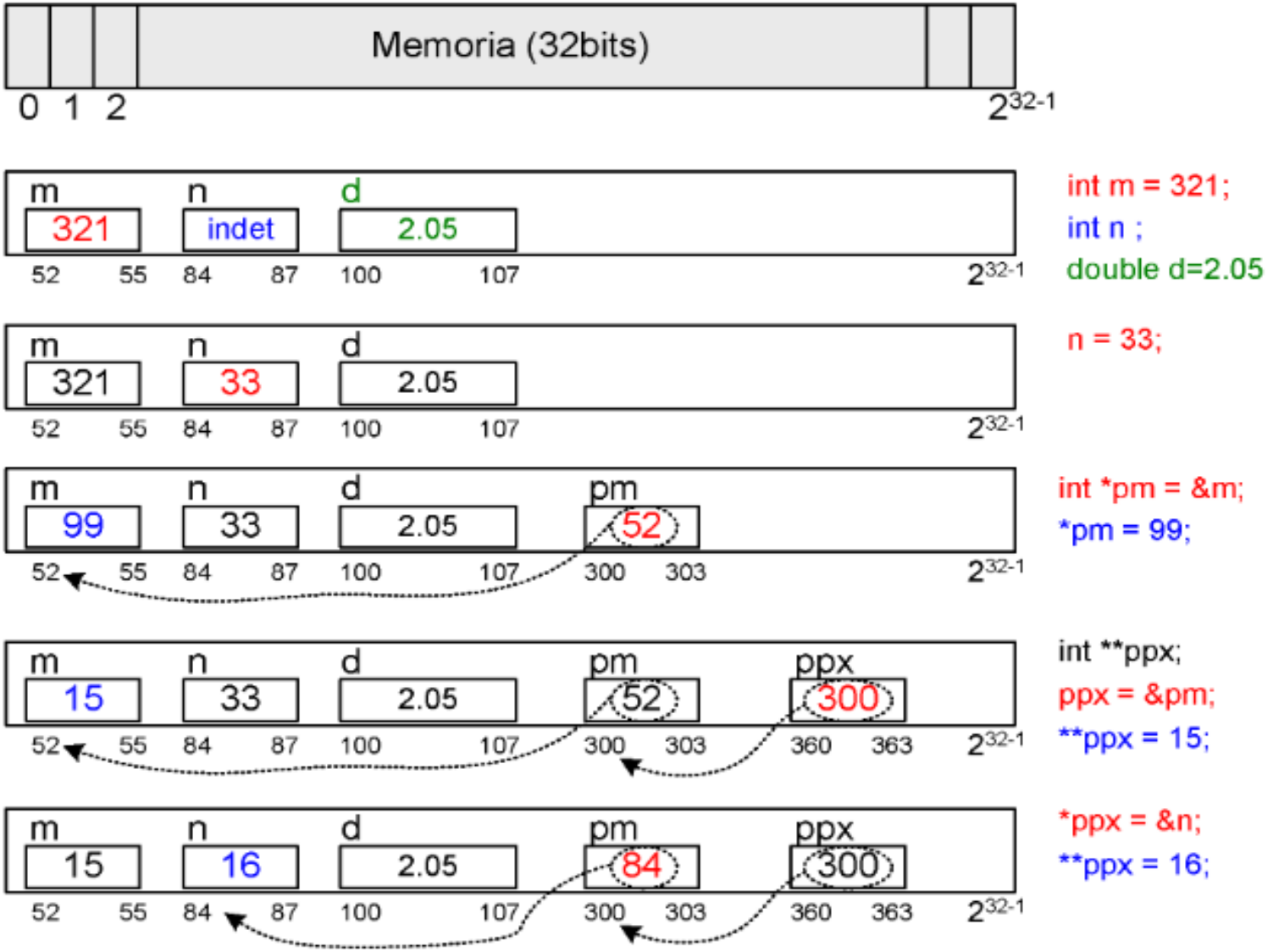
- int *valor;

  declara que *valor es un puntero, por tanto valor es un puntero a entero.

Operaciones con punteros

- C permite estas operaciones:
  - puntero + entero
  - puntero - entero
  - puntero = puntero Asignación entre punteros
  - puntero = NULL
  - puntero == NULL Comparación con NULL
  - puntero != NULL

Juguemos con los punteros



Reserva de memorias

- En el ejercicio anterior, ¿hemos reservado dicha memoria?.
- La respuesta es no. La variable de inicio se alberga en una zona de memoria temporal.
- Heredamos un problema que puede dar la cara en ejecución.

Creación de objeto con malloc

- Para que no nos surja el problema anterior, debemos reservar memoria.

- Para reservar memoria, libc nos proporciona una función llamada malloc.

Creación de objeto con malloc

```
void *malloc(size_t tamaño);
```

- Adjudica espacio para un objeto, cuyo tamaño es especificado por tamaño y cuyo valor es indeterminado.
- La función malloc retorna un puntero nulo o un puntero al espacio adjudicado.

Ejemplo

```
int *p;
p = (int *)malloc(sizeof(int));
```

Liberación de objetos con free

- Siempre que reservamos un espacio en memoria, debemos liberarla.
- Dicha memoria no liberada recibe el nombre de leak o fuga.
- Para ello debemos utilizar la función free

Liberación de objetos con free

- void free(void *ptr);
- La función libera el objeto reservado en la posición correspondiente a ptr.
- free no devuelve nada.

Estructuras

- Una estructura es una colección de una o mas variables, no necesariamente del mismo tipo, agrupadas bajo un solo nombre

- struct nombre_estructura { lista_variables } nombre_variable;

Declaración de estructuras

- Existen tres formas para declarar una estructura parecida:

a)
```
struct par_numeros {
        int n;
        int m;
    } par1, par2;
```

Declaración de estructuras

b) struct par_numeros {

```
    int n;
    int m;
};
```

```
struct par_numeros p1, p2;
```

Declaración de estructuras

```
c) struct {
        int n;
        int n;
} par1 , par2;
```

Operaciones

- copiar estructuras
- acceso a sus miembros
- obtener su dirección con &
- pasarlas como argumentos a funciones
- ser devueltas por funciones
- las estructuras no se pueden comparar en sí pero si sus campos.

Acceder a los miembros de una estructura

- Para acceder a los miembros de una estructura usamos el operador .
  - p1.x = 3
  - p1.y = 9;
  - p2.x = p2.y = 0
- Una estructura puede tener miembros que sean a su vez estructuras

Punteros a estructuras

- También permite punteros a estructuras.
- La declaración de un puntero a una estructura es como la de un puntero a cualquier otra variable.
- Existe dos formas para declararlos:
  - Asignándole la dirección de una variable.
  - Reservándole memoria con alguna función.

Asignando la dirección de una variable

```
struct par_numeros {
        int n;
        int m;
};

struct par_numeros p1;
struct par_numeros *p1 = &p1;
```

Reservando memoria

```
struct par_numeros {
        int n;
        int m;
};

struct par_numeros *p1 =
            (struct par_numeros *)malloc(sizeof(struct nft_table));
```

Acceder a los miembros de un puntero a estructura

- Para acceder a los miembros de una estructura usamos el operador ->

```
p1->n=3
p1->m=9;
```

Buenos métodos para diseñar buena Api

- Estructura privada o pública.
- Método/función alloc
- Método/función free
- Uso de flags (is_set)
- Método/función get
- Método/función unset
- Método/función set
- Función de impresión

Estructura pública o privada.

- Si publicamos una estructura, que expuesta todos los atributos en nuestro programa
- Por tanto, una modificación de ella será mas complicada porque nuestros usuarios pueden estar utilizándola
- Por ello la mejor opción es mantenerla privada (definida en el .c)

Método/función alloc

- Dicha función nos facilita la reserva en memoria de objetos específicos
- También puede realizar reserva de memoria de campos internos.

Método/función free

- Dicha función realiza la liberación de la estructura en cuestión.
- También sirve para liberar campos internos los cuales necesiten un tratamiento especial

Uso de flags (is_set)

- Los flags es un indicador que añadimos a nuestras estructuras para saber que atributos de ella misma están activos o no.
- Esto facilita al desarrollador la prevención de crash por acceso a atributos no inicializados

Método/función get

- Función que devuelve los atributos de una estructura
- En las estructuras existen campos de diversos tipos, para ello se juega con los punteros de tipos no definidos y funciones que realizan casting de ellos.
- Debemos comprobar si los campos de nuestras estructuras están activos para no tener errores al acceder a campos no inicializados.

Método/función unset

- Función que nos permite liberar de forma controlada los atributos.
- Es usada por la función set antes de asignar un valor nuevo.

Método/función set

- Función que nos ayuda a asignar valores a los atributos de una estructura.
- Nos permite la liberación controlada de atributos y la asignación de nuevos.

Método/función impresión

- Esta función nos ayuda a depurar fallos en nuestras estructuras
- Se pueden realizar en diversos formatos como json, xml, formatos personalizados...

Ejercicios

Ejemplo 4: arrays.

- Declaración
- Acceso y usos

Declaración de arrays unidimensionales

Tipo_dato nombre_array[TAMAÑO]

ó

Tipo_dato nombre_array[]

```
char *nombres[] = {"el", "curso", "de", "C", "mola"};
char *nombres[20];
```

Declaración de arrays unidimensionales

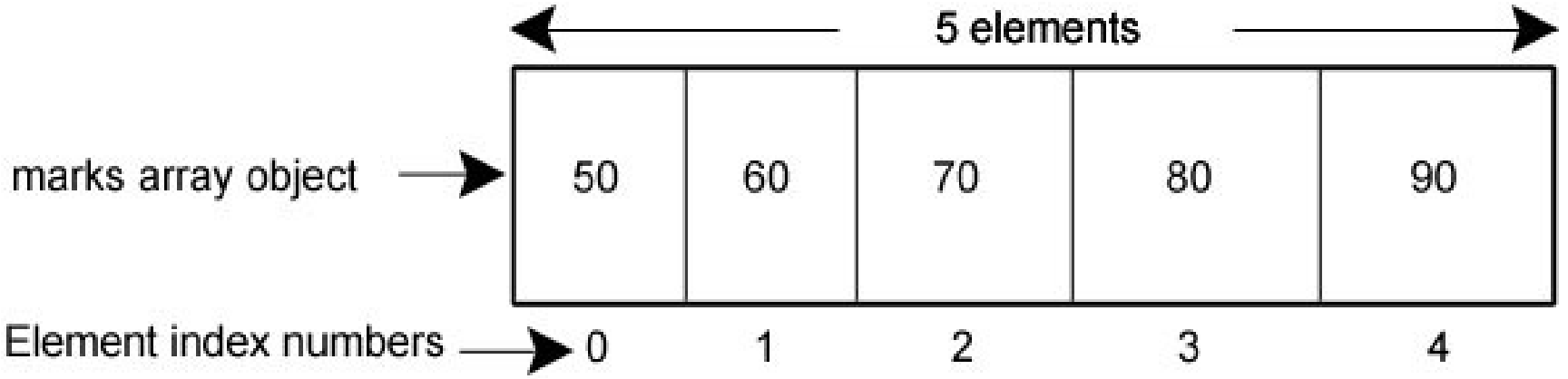


Figure : marks array with 5 elements
(after assigning values to elements)

Declaración de arrays unidimensionales

- Podemos crear arrays de estructuras de datos

  struct coche concesionario[50];

- O incluso de punteros a estructuras de datos

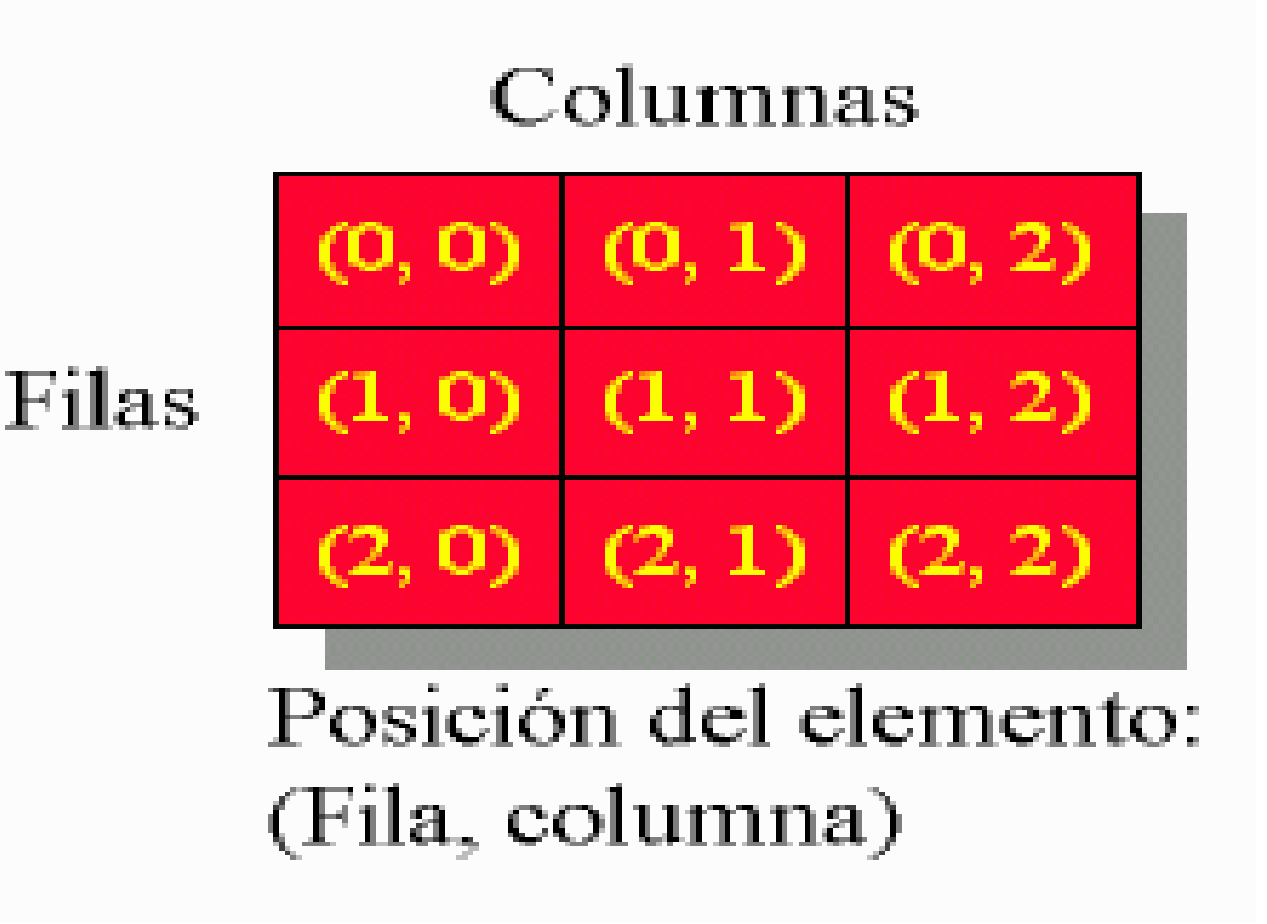
  struct coche *concesionario[50];

Declaración de arrays bidimensionales

```
int valores[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

- El uso de los arrays bidimensionales se tiene que llevar con cautela, por su coste computacional.

Declaración de arrays bidimensionales



Declaración de arrays usando la memoria

```
int *v;
 v = (int *)malloc(10 * sizeof(int));
```

- Equivalente a

```
int *v[10];
```

Acceso a arrays unidimensionales

- El acceso a los valores se realiza con un índice, ej:

  concesionario[0];

  Esto nos devolverá el struct coche comprendido en la posición 0 de nuestro array.

Acceso a arrays bidimensionales

- El acceso a los valores se hace a partir de dobles índices, ej:

  valores[1][1]

  Usando la declaración del array anterior, esa sentencia nos devolvería el número 1.

¿Para que usar los arrays?

- Los arrays son buenos contenedores para información.
- Nos permite guardar y acceder a ellos con gran facilidad
- El mayor inconveniente de los arrays es que su tamaño no puede ser modificado

Ejercicios

Ejemplo 3: errores clásicos y depurando con Valgrind.

- Segfaults.
- Acceso a punteros inválidos
- Fugas de memoria
- Corrupciones de memoria.
- Valgrind: En qué puede ayudarnos.

errores clásicos y depurando con Valgrind

- Es común cuando se realiza un programa que tengamos errores típicos.
- En algunos casos estos errores, nos pueden llevar solucionarlo incluso días
- Los errores de memoria son los mas frecuentes y los mas complicados de solucionar
- Es difícil desenmascararlos e incluso acotarlos.

Segfaults o Fallo de segmentación

- Se denomina fallo de segmentación al intento fallido de acceso a información o a programas a los que no se tiene autorización para ver o modificar.
- Son los mas típicos, el S.O. por lo general devuelve la cadena "Violación de segmento" para notificárnoslos
- Estos fallos rompen la ejecución de nuestros programas.

Acceso a punteros inválidos

- La causa de este error, viene por intentar acceder a direcciones de tipo invalidas o nulas.
- La principal causa de estos errores es el acceso a un puntero el cual ha sido ya liberado.

Fugas de memoria (memory leaks)

- La causa de este error es la no liberación de memoria de forma correcta, de memoria que hayamos reservado usando malloc.
- Una fuga de memoria, es de los mas impredecibles ya que nuestro programa funciona de forma correcta.
- Con la ejecución el tiempo puede llegar a bloquear o reiniciar maquinas por overflow

Corrupciones de memoria

- Ocurre en un programa cuando los contenido de una dirección de memoria involuntariamente se modifican debido a errores de programación; esto se conoce como violación de la seguridad de memoria.
- Hay que tener mucho cuidado con mantener los punteros intactos...

Valgrind

- Valgrind es una herramienta que permite detectar fallos en la gestión de memoria
- Errores como los nombrados anteriormente (leaks, segfaults, corrupciones de memoria...)
- Es de las herramientas mas potentes para un buen desarrollador en C

Como usar Valgrind

- valgrind --leak-check=full --show-reachable=yes nombre_programa

  --leak-check=full : Hace que la herramienta nos muestre las fugas de memoria que tiene nuestro programa

  --show-reachable=yes : Nos muestra un informe detenido

Ejemplo con Valgrind

- Prog.c

```
int main()
{
int i;
int *v;
v = (int *)malloc(10 * sizeof(int));
for (i = 0; i < 20; i++)
    v[i] = 10;

// free(v);
}
```

Ejemplo con Valgrind

- ==8775== Memcheck, a memory error detector
- ==8775== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
- ==8775== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
- ==8775== Command: ./prog
- ==8775==
- ==8775== Invalid write of size 4
- ==8775== at 0x40053F: main (in /home/alvaro/Escritorio/soleta/cursosextension/cursoC/2Clase/presentacion/prog)
- ==8775== Address 0x51bb068 is 0 bytes after a block of size 40 alloc'd
- ==8775== at 0x4C28BED: malloc (vg_replace_malloc.c:263)
- ==8775== by 0x40051D: main (in /home/alvaro/Escritorio/soleta/cursosextension/cursoC/2Clase/presentacion/prog)
- ==8775==
- ==8775==
- ==8775== HEAP SUMMARY:
- ==8775== in use at exit: 40 bytes in 1 blocks
- ==8775== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
- ==8775==
- ==8775== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
- ==8775== at 0x4C28BED: malloc (vg_replace_malloc.c:263)
- ==8775== by 0x40051D: main (in /home/alvaro/Escritorio/soleta/cursosextension/cursoC/2Clase/presentacion/prog)
- ==8775==
- ==8775== LEAK SUMMARY:
- ==8775== definitely lost: 40 bytes in 1 blocks
- ==8775== indirectly lost: 0 bytes in 0 blocks
- ==8775== possibly lost: 0 bytes in 0 blocks
- ==8775== still reachable: 0 bytes in 0 blocks
- ==8775== suppressed: 0 bytes in 0 blocks
- ==8775==
- ==8775== For counts of detected and suppressed errors, rerun with: -v
- ==8775== ERROR SUMMARY: 11 errors from 2 contexts (suppressed: 4 from 4)
-

Ejercicios