Informática

Unidad 2H: Apuntadores

Ingeniería en Mecatrónica

Facultad de Ingeniería
Universidad Nacional de Cuyo

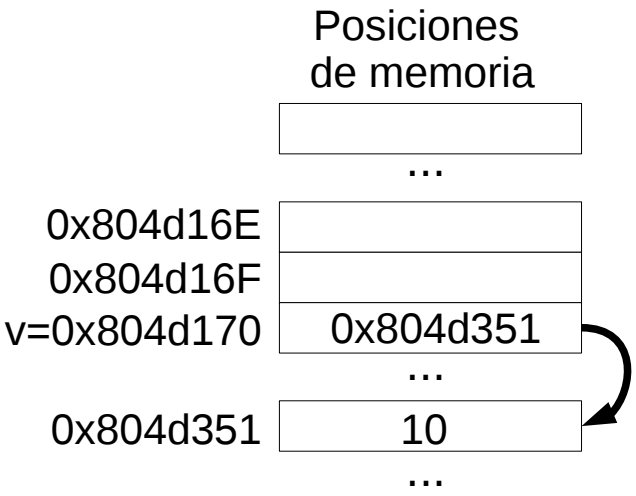
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERIA
en acción continua...

Apuntadores

- Un apuntador es una variable que contiene la dirección de memoria donde se almacena un valor

```
int *v;
...
*v=10;
```



- El tipo de un apuntador indica el tipo de dato del valor apuntado. Ej:

  `int *v;` → `v` es un apuntador a un entero

Apuntadores

- Declaración de una variable tipo apuntador

```
int *v;
```

- Obtención de la dirección de memoria de una variable: **operador de dirección &**

```
int *v;
int a;
v = &a;
```

- Obtención del valor apuntado por el apuntador: **operador de indirección ***

```
int *v;
int a=10;
v = &a;
*v = 11;
printf(“%d”, *v);
```
→ Imprime el valor 11;

Apuntadores

- Usos de los apuntadores
  - Pasaje de argumentos a funciones por referencia
  - Manipulación de arreglos
  - Implementación de estructuras de datos avanzadas, como ser listas enlazadas, pilas, colas, árboles, etc.
  - Creación de estructuras de datos dinámicas (estructuras de datos cuya existencia y tamaño se define *en tiempo de ejecución* en lugar de *en tiempo de compilación*)

Apuntadores

- Pasaje de argumentos a funciones por valor vs. por referencia
  - Cuando se pasa el argumento por **valor**, los cambios que la función/procedimiento realiza sobre el argumento no persisten. Ej:

```
void funcion(int valor){
    valor += 1;
}
int main(){
    int v=10;
    funcion(v);
    printf("%d", v); → v sigue teniendo el valor 10
}
```

Apuntadores

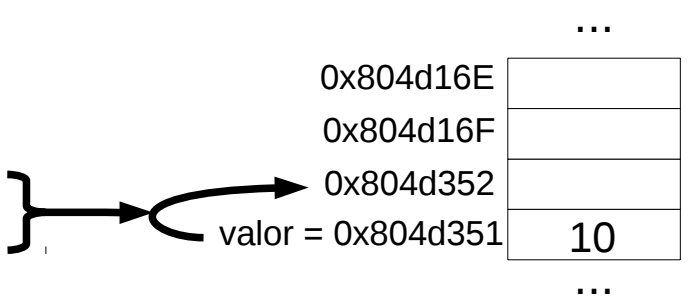
- Pasaje de argumentos a funciones por valor vs. por referencia
  - Cuando se pasa el argumento por **referencia**, los cambios que la función/procedimiento realiza sobre el argumento persisten. Ej:

```
void funcion(int *valor){
    *valor += 1;
}
int main(){
    int v=10;
    funcion(&v);
    printf("%d", v); → v ahora tiene el valor 11
}
```

Apuntadores

- Aritmética de apuntadores
  - Si se le suma un valor a un apuntador, no se modifica el valor apuntado, sino que se modifica la dirección de memoria almacenada en el apuntador
  - Ej:

```
int *valor;
*valor = 10; // guarda 10 en la posición
             // de memoria apuntada
valor += 1;  //hace que el apuntador apunte
             //a la siguiente dirección de memoria
```



  - La aritmética de apuntadores puede utilizarse para manipular arreglos (se verá más adelante)
  - Riesgos: violación de segmento
    - Ocurre cuando se modifica el valor de un apuntador haciendo que el mismo quede apuntando a una dirección fuera de los segmentos asignados al proceso

Apuntadores

- Apuntadores múltiples
  - Permiten apuntar a una dirección de memoria en la que no hay un valor, sino otra dirección de memoria
  - A esto se le llama ***referencia indirecta***

```
int a=1;
int *p;
int **p1;
p = &a;
p1 = &p;

printf(“%d %d %d”, a, *p, **p1);
```

→ los 3 valores son iguales

Apuntadores

- Asignación de memoria dinámica
  - Hasta ahora se han declarado variables para las cuales se ha reservado memoria al iniciar el programa (asignación *estática* de memoria)
  - Cuando no se sabe de antemano la cantidad de elementos necesarios, o su necesidad, es posible reservar memoria de manera *dinámica* (es decir, en tiempo de ejecución)
  - Existen varios casos: Asignación dinámica de memoria para elementos simples

```
int *v;  //Declaramos el apuntador
v = malloc(sizeof(int)); //Reservamos memoria
```

  - La función `malloc` recibe como argumento la cantidad de bytes a reservar, y devuelve un puntero `void *` que debe ser "moldeado" con un *cast* explícito al tipo de puntero declarado. En el ejemplo, el cast se hace con: `(int *)`

Apuntadores

- Asignación dinámica de memoria para elementos simples
  - El operador unario `sizeof` devuelve el tamaño en bytes del tipo de datos dado como argumento (pueden utilizarse tanto tipos simples como tipos complejos)

- Asignación dinámica de memoria para estructuras

```
struct VERTICE *v;
v = (struct VERTICE *)malloc(sizeof(struct VERTICE));
```

- Acceso a los elementos de una estructura asignada dinámicamente: Hay 2 formas equivalentes:

```
(*v).x = 10;  //Ambas formas son equivalentes
v->y = 10;    //(utilizando -> no es necesario el *)
```

Apuntadores

- Asignación dinámica de memoria para arreglos 1D: 2 formas
  - Declaramos simplemente el puntero

```
int *arreglo;
arreglo = calloc(10, sizeof(int));
```

  - Lo declaramos como arreglo:

```
int arreglo[];
//Reservamos espacio para un arreglo de 10 integers
arreglo = calloc(10, sizeof(int));
```

  - La función `calloc` es similar a malloc, sólo que toma un argumento más (el primero): la cantidad de elementos a reservar

Apuntadores

- Asignación dinámica de memoria para arreglos de N dimensiones
  - Un arreglo de 2 dimensiones es un “arreglo de arreglos”
  - En un arreglo de N dimensiones, cada elemento es un apuntador a un arreglo, cuyos elementos a su vez son apuntadores a otros arreglos, y así sucesivamente
  - Ejemplo para 2 dimensiones

Reservamos 10 elementos del tamaño de un puntero simple

```
int **arreglo2D;

int i;

arreglo2D = calloc(10, sizeof(int *));

for(i = 0; i < 10; i++){

    arreglo2D[i] = calloc(10, sizeof(int));

}
```

Apuntadores

- Liberando la memoria
  - El compilador no inserta código automáticamente para liberar la memoria, por lo que la misma debe liberarse manualmente una vez que ya no se utiliza
  - Para ello, se utiliza la función `free`
  - Liberando la memoria de elementos individuales

```
int *v;
v = malloc(sizeof(int));
…
free(v); //Liberamos la memoria
```

  - Liberando la memoria de arreglos 1D

```
int *arreglo;
arreglo = calloc(10, sizeof(int));
…
free(arreglo); //Liberamos la memoria
```

Apuntadores

- Liberando la memoria
  - Liberando la memoria de arreglos de 2 dimensiones

```
int **arreglo2D;
int i;
arreglo2D = calloc(10, sizeof(int *));
for(i = 0; i < 10; i++)
    arreglo2D[i] = calloc(10, sizeof(int));
…
for(i = 0; i < 10; i++)
    free(arreglo2D[i]);  //Liberamos cada “fila”

free(arreglo2D); //Liberamos el puntero doble
```

  - Lo mismo se aplica a arreglos de N dimensiones

Apuntadores

- Fugas de memoria
  - Cuando se usan apuntadores para mantener direcciones de memoria de estructuras de datos creadas dinámicamente, se debe liberar la memoria en forma explícita una vez que se termina de utilizar
  - Al finalizar la ejecución de una función, el compilador automáticamente libera la memoria de las variables locales de la función, **pero no libera la memoria de las variables dinámicas**
  - Cuando termina la ejecución de la función, los apuntadores locales de la función se liberan, pero si no se libera la memoria dinámica explícitamente, esa memoria queda marcada como “utilizada” (no puede ser asignada a otro proceso), y sin embargo no hay forma de accederla porque el apuntador se liberó
  - A esto se le llama ***fuga de memoria*** (***memory leak***)