Tema 1: Introducción a la Gestión Dinámica de Memoria en C

Basado en el material de J.L. Triviño Rodríguez — Versión 1.0 (2024)

Apuntes profesionales elaborados con estilo y claridad.

Índice

1. Estructura en Memoria de un Programa en C

Cuando se ejecuta un programa en C, la memoria se organiza en varias **zonas bien definidas**. Comprenderlas es esencial para evitar errores como **fugas de memoria** o **desbordamientos de pila**.

1.1 Zonas principales de memoria

- **Sección de Código (Text Segment):** contiene las instrucciones ejecutables. Es de solo lectura.
- **Sección de Datos Inicializados:** variables globales o estáticas con valores distintos de 0.
- **Sección BSS:** variables globales o estáticas sin inicializar o a 0.
- **Pila (Stack):** gestiona llamadas a funciones y variables locales. Crece hacia abajo.
- **Montículo (Heap):** memoria dinámica controlada por el programador mediante `malloc`, `calloc`, `realloc` y `free`.

1.2 Ejemplo

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

int global_var = 10; // Datos inicializados
int x;               // Datos no inicializados (BSS)

void funcionEjemplo() {
    static int static_var = 20; // Datos inicializados
    int local_var = 30;         // Pila
    int *dynamic_var = (int *) malloc(sizeof(int)); // Heap

    if (dynamic_var != NULL) {
        *dynamic_var = 40;
        printf("Variable dinámica: %d\n", *dynamic_var);
        free(dynamic_var);
    }
}
int main() {
    funcionEjemplo();
    return 0;
}
```

2. Direccionamiento de Memoria

El **direccionamiento** indica cómo se accede a las ubicaciones de memoria.

2.1 Direcciones de memoria

Cada variable tiene una dirección única, accesible con el operador `&`.

```c
int var = 20;
printf("La dirección de var es: %p\n", &var);
```

2.2 Tipos de direccionamiento

- **Directo:** se accede por nombre (`x`).
- **Indirecto:** se accede mediante un puntero (`*ptr`).

—

3. Punteros

Los **punteros** almacenan direcciones de memoria. Son fundamentales para la **gestión dinámica**.

3.1 Declaración y asignación

```c
int *ptr;       // Puntero a int
int variable = 10;
ptr = &variable; // ptr apunta a variable
```

3.2 Desreferenciación

```c
printf("Valor de variable: %d\n", *ptr);
```

3.3 Aritmética de punteros

Permite recorrer arrays.

```c
int arr[5] = {1,2,3,4,5};
int *p = arr;
p++; // Avanza al siguiente elemento
```

3.4 Buenas prácticas

- Siempre inicializa punteros a `NULL`.
- Verifica antes de desreferenciar.
- Usa `free()` para liberar memoria.

---

4. Paso de Parámetros

En C existen dos formas de pasar datos a funciones:

- **Paso por valor:** se pasa una copia del valor.
- **Paso por referencia:** se pasa la dirección (puntero).

4.1 Ejemplo paso por valor

```c
void modificarPorValor(int x){ x += 10; }

int main(){
    int n = 5;
    modificarPorValor(n);
    printf("%d", n); // sigue valiendo 5
}
```

4.2 Ejemplo paso por referencia

```c
void incrementar(int *num){ (*num)++; }

int main(){
    int x = 10;
    incrementar(&x);
    printf("%d", x); // ahora vale 11
}
```

---

5. Gestión Dinámica de Memoria

Permite reservar memoria durante la ejecución usando `malloc`, `calloc`, `realloc` y `free`.

5.1 malloc

```c
int *ptr = malloc(sizeof(int)*5);
if(ptr == NULL){
    printf("Fallo en asignación");
}
```

5.2 calloc

Inicializa la memoria a cero:

```c
int *ptr = calloc(5, sizeof(int));
```

5.3 realloc

Redimensiona la memoria existente:

```c
ptr = realloc(ptr, sizeof(int)*10);
```

5.4 free

Libera la memoria:

```c
free(ptr);
ptr = NULL;
```

5.5 Buenas prácticas

- Verifica siempre si la asignación devolvió `NULL`.
- Libera toda memoria antes de finalizar el programa.
- Nunca uses punteros después de `free()`.

---

6. Buffers y Cadenas

6.1 Buffers

Un **buffer** es una zona temporal para almacenar datos.

```c
#define BUFFER_SIZE 1024
char buffer[BUFFER_SIZE];
FILE *file = fopen("data.txt", "r");
fgets(buffer, BUFFER_SIZE, file);
fclose(file);
```

6.2 Cadenas de caracteres

```c
char saludo[] = "Hola Mundo";
printf("%s", saludo);
```

Funciones comunes de `<string.h>`:

- `strcpy()`, `strcat()`, `strcmp()`

---

7. Listas Enlazadas

Estructura dinámica donde cada nodo apunta al siguiente.

7.1 Definición

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

7.2 Inserción al principio

```c
void insertarInicio(Nodo **cabeza, int valor){
    Nodo *nuevo = malloc(sizeof(Nodo));
    nuevo->dato = valor;
    nuevo->sig = *cabeza;
    *cabeza = nuevo;
}
```

7.3 Recorrer lista

```c
void mostrarLista(Nodo *n){
    while(n != NULL){
```

```
        printf("%d -> ", n->dato);
        n = n->sig;
    }
    printf("NULL\n");
}
```

---

8. Conclusión

La **gestión dinámica de memoria** y el **uso de punteros** son pilares del lenguaje C. Dominar estos conceptos permite escribir programas eficientes, seguros y capaces de manipular estructuras como **listas enlazadas**, **buffers** y **cadenas de texto**. **Comprender la memoria es comprender el corazón de C.**