Programación II

Gestión de la memoria (cont.)

Lic. Federico Severino Guimpel
LCC - FCEIA - UNR

Pasaje de argumentos a funciones

Continuaremos viendo qué sucede en las funciones cuando se pasan argumentos. Analicemos este código:

```c
#include<stdio.h>

void suma(int op1, int op2, int r) {
    r = op1 + op2;
}

int main() {

    int a = 3, b= 4;
    int resultado;
    suma(a, b, resultado);

    printf("El resultado de la suma es: %d\n", resultado);
}
```

Pasaje de argumentos a funciones

Tenemos una función que toma tres argumentos; y guarda en el tercero el resultado de la suma de los dos primeros. Luego en el main llamamos a la función y mostramos el resultado. Cuando lo ejecutamos vemos que la salida es:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
El resultado de la suma es: 0
```

La variable que pasamos no guarda el valor asignado dentro de la función, de hecho muestra 0. ¿Por qué será esto?

Pasaje de argumentos a funciones

Para mostrar lo que sucede vamos a modificar el programa agregando salidas por pantalla en la función y en el main.

```c
#include<stdio.h>

void suma(int op1, int op2, int r) {
    printf("Las direcciones de los argumentos son: %p %p %p\n\n", &op1, &op2, &r);
    r = op1 + op2;
}

int main() {

    int a = 3, b= 4;
    int resultado;

    printf("Las direcciones de las variables son: %p %p %p\n\n", &a, &b, &resultado);

    suma(a, b, resultado);

    printf("El resultado de la suma es: %d\n", resultado);
}
```

Pasaje de argumentos a funciones

Al ejecutar el programa obtenemos una salida como esta:

```
Fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
Fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Las direcciones de las variables son: 0x7ffff0b23dbc 0x7ffff0b23dc0 0x7ffff0b23dc4

Las direcciones de los argumentos son: 0x7ffff0b23d9c 0x7ffff0b23d98 0x7ffff0b23d94

El resultado de la suma es: 32767
```

Todas las direcciones que vemos son distintas. Es decir, los argumentos con los que trabaja la función no son las variables que pasamos. Están en otro lugar de la memoria.

Pasaje de argumentos a funciones

En C todo argumento es pasado por valor. ¿Qué significa esto? Se hace una copia de los mismos en otro lugar de la memoria. Se manipula ese espacio dentro de la función y, al finalizar se libera. Es por eso que los cambios realizados en una función a sus argumentos no se ven al volver al lugar de la llamada.

Pasaje de argumentos a funciones

Para terminar de entender esto veamos este ejemplo donde se retorna la dirección de una variable creada dentro de la función.

```c
#include<stdio.h>

int* creaVariable() {

    int i = 1;
    return &i;
}

int main() {

    return 1;
}
```

Al compilar nos salta este error:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function ‘creaVariable’:
ejemplo1.c:7:9: warning: function returns address of local variable [-Wreturn-local-addr]
  return &i;
         ^~
```

Pasaje de argumentos a funciones

Modificando un poco el ejemplo:

```c
#include<stdio.h>

int* creaVariable(int j) {

    return &j;
}

int main() {

    return 1;
}
```

El resultado, al compilar, es el mismo:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function ‘creaVariable’:
ejemplo1.c:6:9: warning: function returns address of local variable [-Wreturn-local-addr]
  return &j;
         ^~
```

Pasaje de argumentos a funciones

Los argumentos están en un mismo espacio junto a las variables locales (propias de la función). Cuando la función finaliza ese espacio de memoria es liberado automáticamente. Este es el motivo por el que no se puede retornar direcciones de variables locales o argumentos de una función.

Pasaje de argumentos a funciones

Sin embargo, esto sí funciona:

```c
#include<stdio.h>

void cambiaCaracter(char c[]) {
    c[0] = 'S';
}

int main() {

    char a[] = "Hola";
    cambiaCaracter(a);
    printf("%s\n", a);
}
```

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Sola
```

¿Por qué les parece que aquí sí se modifica la cadena pasada?

Pasaje de argumentos a funciones

Veamos qué está pasando. Imprimamos las direcciones de memoria y el valor del argumento y la variable.

```c
#include<stdio.h>

void cambiaCaracter(char c[]) {
    printf("El argumento está en %p y tiene el valor %p\n\n", &c, c);
    c[0] = 'S';
}

int main() {

    char a[] = "Hola";
    printf("La variable está en %p y tiene el valor %p\n\n", &a, a);
    cambiaCaracter(a);
    printf("%s\n", a);
}
```

Pasaje de argumentos a funciones

Al correr obtenemos:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
La variable está en 0x7fffd5acddf3 y tiene el valor 0x7fffd5acddf3

El argumento está en 0x7fffd5acddd8 y tiene el valor 0x7fffd5acddf3

Sola
```

En el main, como ya habíamos visto, el `array` tiene el mismo valor que su dirección de memoria pero en el argumento no es así. El argumento está en un lugar distinto que el valor que guarda. Este valor es la dirección del `array` definido en el main.

Pasaje de argumentos a funciones

En este caso, C convierte al `array` pasado como argumento en un puntero que referencia al lugar donde está realmente el `array`.

La sentencia

```
c[0] = 'S';
```

está yendo al lugar donde está el array originalmente y lo modifica.

Pasaje de argumentos a funciones

Es decir, en caso de tener un puntero se utiliza el operador referencia al lugar donde está realmente el `array`.

La sentencia

```
c[0] = 'S';
```

está yendo al lugar donde está el array originalmente y lo modifica.

Pasaje de argumentos a funciones

Lo que no se podría hacer es modificar el argumento dentro de la función y que los cambios se vean en el main:

```c
#include<stdio.h>

void cambiaCaracter(char c[]) {
    c = "Sola";
}

int main() {

    char a[] = "Hola";
    cambiaCaracter(a);
    printf("%s\n", a);
}
```

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Hola
```

Pasaje de argumentos a funciones

Ya que esta sentencia:

```
c = "Sola";
```

hace que la variable cambie el lugar al que hace referencia.

```
#include<stdio.h>

void cambiaCaracter(char c[]) {
    printf("%p\n", c);
    c = "Sola";
    printf("%p\n", c);
}

int main() {

    char a[] = "Hola";
    cambiaCaracter(a);
    printf("%s\n", a);
}
```

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
0x7fffce01e553
0x7f324cc00828
Hola
```

Tipos de memoria

Como vimos la memoria asignada a la ejecución de una función es automática. Es decir, se asigna y se libera automáticamente cuando la misma finaliza. Sin embargo, no toda la memoria es así.

Hay memoria estática y dinámica.

Memoria estática

El concepto de memoria estática hace referencia a la memoria que no va a ser liberada hasta que finalice el programa. Como vimos en la presentación anterior, la memoria estática que nos asigna el sistema operativo no es infinita.

Cuando un programa se ejecuta las funciones, variables y definiciones globales se alojan en esta memoria.
Por ese motivo cuanta más memoria estática ocupemos menos memoria va a quedar disponible para la ejecución del programa.

Memoria dinámica

La memoria dinámica se denomina así porque se asigna en tiempo de ejecución y, se va liberando cuando ya no se precisa. En C esta memoria es administrada por el programador o programadora.

Para manejar la memoria dinámica se deben usar algunas funciones que se encuentran en la librería stdlib.h

Memoria dinámica

Para solicitar un espacio de memoria dinámica se debe usar la función `malloc`. Esta función suele usarse en conjunto con la función `sizeof`.

La función `malloc` (memory allocation) pide un bloque de memoria al sistema operativo. Se le indica el tamaño del bloque y retorna la dirección de memoria donde inicia ese bloque.

La función `sizeof` se usa para saber el tamaño de un tipo de dato.

La función `free` se usa para liberar la memoria. Se le pasa el puntero que hace referencia al bloque de memoria que se desea liberar.

Un ejemplo de una llamada sería:

```
int* ptr;
ptr = malloc(sizeof(int)*5);
```

En este caso se define un puntero y se guarda en él la dirección que se obtiene de pedir un bloque de memoria suficiente para almacenar 5 enteros.

¿Qué sucede si no hay memoria suficiente? En ese caso `malloc` retornará un valor especial: NULL. Este valor, que puede ser asignado a un puntero, indica que el mismo no hace referencia a una dirección de memoria.

Ahora que tenemos acceso a un bloque de memoria. ¿Cómo accedemos a él? ¿Cómo escribimos o recuperamos un dato de ese bloque?

Tenemos dos opciones: el operador `[]` o lo que se conoce como aritmética de punteros.

Memoria dinámica

El operador [] ya vimos que se desplaza por las direcciones de memoria pero, ahora veremos explícitamente qué representa.

Observemos el siguiente código:

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

int main() {
    int* ptr;
    ptr = malloc(sizeof(int)*5);

    for(int i=0; i<5; i++){
        ptr[i] = i;
    }

    for(int i=0; i<5; i++){
        printf("posición %d dirección %p valor %d\n", i, &ptr[i], ptr[i]);
    }

    free(ptr);
    return 1;
}
```

Al ejecutarlo tenemos:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Wall ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
posición 0 dirección 0x7fffc9f0d260 valor 0
posición 1 dirección 0x7fffc9f0d264 valor 1
posición 2 dirección 0x7fffc9f0d268 valor 2
posición 3 dirección 0x7fffc9f0d26c valor 3
posición 4 dirección 0x7fffc9f0d270 valor 4
```

Aritmética de punteros

¿Entonces ptr[i] qué hace?

En nuestro código ptr tiene la dirección donde inicia el bloque de memoria que el sistema operativo nos dio.
Escribir ptr[i] es lo mismo que escribir *(ptr+i). Por otro lado, ptr+i representa la dirección de memoria que está i lugares desplazado del valor que tiene ptr.Al poner el * va a esa dirección

A las expresiones del tipo ptr+i se las llama aritmética de punteros. A continuación veremos un ejemplo para ilustrar esto.

Aritmética de punteros

El ejemplo es este:

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

int main() {
    int* ptr;
    ptr = malloc(sizeof(int)*5);

    for(int i=0; i<5; i++){
        ptr[i] = i;
    }

    printf("Usando el operador []\n");
    for(int i=0; i<5; i++){
        printf("posición %d dirección %p valor %d\n", i, &ptr[i], ptr[i]);
    }

    printf("\n\nUsando aritmética de punteros\n");
    for(int i=0; i<5; i++){
        printf("posición %d dirección %p valor %d\n", i, ptr+i, *(ptr+i));
    }

    free(ptr);
    return 1;
}
```

Aritmética de punteros

Al ejecutarlo validamos que:

- ptr[i] es lo mismo que *(ptr+i)
- &ptr[i] es lo mismo que ptr+i

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Wall ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Usando el operador []
posición 0 dirección 0x7fffde366260 valor 0
posición 1 dirección 0x7fffde366264 valor 1
posición 2 dirección 0x7fffde366268 valor 2
posición 3 dirección 0x7fffde36626c valor 3
posición 4 dirección 0x7fffde366270 valor 4


Usando aritmética de punteros
posición 0 dirección 0x7fffde366260 valor 0
posición 1 dirección 0x7fffde366264 valor 1
posición 2 dirección 0x7fffde366268 valor 2
posición 3 dirección 0x7fffde36626c valor 3
posición 4 dirección 0x7fffde366270 valor 4
```

Aritmética de punteros

Es importante ver que la memoria dinámica, al no ser liberada hasta que se invoque explícitamente a la función `free`, se puede retornar desde una función.

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

int* crea(int tamanio) {
    int* nuevo = malloc(sizeof(int)* tamanio);
    return nuevo;
}

int main() {
    int t = 10;
    int* ptr = crea(t);

    free(ptr);
    return 1;
}
```

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Wall ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
```

Estructuras

Una Estructura permite definir un nuevo tipo de dato. Además, permite agrupar bajo un único nombre diferentes datos. Un ejemplo sería:

```
struct Persona {
    char* nombre, *direccion, *nacionalidad;
    int edad;
};
```

En este caso se está definiendo a una estructura que tiene cuatro atributos. Un dato entero y tres son cadenas.

Estructuras

Para poder definir una variable del tipo de la estructura simplemente hacemos:

```
struct Persona p1;
```

y, para acceder a sus atributos se usa el operador . con la siguiente sintaxis

```
p1.nombre = "Federico";

p1.direccion = malloc(sizeof(char)*20);
printf("Ingrese una dirección: ");
scanf("%s", p1.direccion);

p1.nacionalidad = malloc(sizeof(char)*20);
strcpy(p1.nacionalidad, "argentino");
```

Además de mostrar la forma de acceder a los atributos se puede ver diferentes formas de inicializar un puntero a un string.

La primera asignación, en particular, es diferente de las otras. No tiene un `malloc` sino que se le asigna, directamente, un string. Esta asignación tiene una limitación.

```
p1.nombre = "Federico";
```

Estructuras

Veamos el siguiente ejemplo:

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

struct Persona{
    char* nombre, *direccion, *nacionalidad;
    int edad;
};


int main() {
    struct Persona p1;
    p1.nombre = "Federico";
    p1.nombre[0] = 'S';

    printf("%s", p1.nombre);
    return 1;
}
```

No hay nada extraño en este código. Vamos a ejecutarlo.

Al hacerlo nos encontramos con esto:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Wall ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Segmentation fault (core dumped)
```

¿Por qué? Sucede que esta asignación:

```
p1.nombre = "Federico";
```

está guardando en el puntero nombre la dirección de memoria de una constante, es decir, de una cadena que NO se puede modificar. Al tratar de modificar el primer caracter de esa cadena es cuando se genera el error.

Volviendo a las otras asignaciones

```
p1.direccion = malloc(sizeof(char)*20);
printf("Ingrese una dirección: ");
scanf("%s", p1.direccion);
```

```
p1.nacionalidad = malloc(sizeof(char)*20);
strcpy(p1.nacionalidad, "argentino");
```

Ellas tienen algo en común. Se está pidiendo un bloque de memoria dinámica y se escribe allí. En un caso se escribe la cadena ingresada por teclado y, por otro se copia un string constante permitiendo que el mismo esté en un área de memoria que se pueda manipular.

En ambos casos estos bloques de memoria deben ser liberados:

"Por cada `malloc` debe haber un `free`"

```
p1.direccion = malloc(sizeof(char)*20);
printf("Ingrese una dirección: ");
scanf("%s", p1.direccion);
```

```
p1.nacionalidad = malloc(sizeof(char)*20);
strcpy(p1.nacionalidad, "argentino");
```

```
free(p1.direccion);
free(p1.nacionalidad);
```

Ahora bien, retomando nuestro ejemplo. ¿Cuál sería la diferencia entre estas dos definiciones? ¿Se les ocurre alguna?

```
struct Persona {
    char* nombre, *direccion, *nacionalidad;
    int edad;
};
```

```
struct Persona1{
    char nombre[30], direccion[30], nacionalidad[30];
    int edad;
};
```

En este caso, la estructura Persona tiene tres punteros mientras que Persona1 tiene tres `arrays` de tamaño fijo. Esto hace que cada vez que se cree una Persona1 la misma ya tiene su espacio completamente definido, sin importar cuánto ocupe la información que se vaya a guardar.

```
struct Persona{
    char* nombre, *direccion, *nacionalidad;
    int edad;
}
```

```
struct Persona1{
    char nombre[30], direccion[30], nacionalidad[30];
    int edad;
};
```

Para sacar provecho de tener punteros en lugar de `arrays` podría cambiar el código anterior por este:

```
Persona p1;
p1.nombre = malloc(sizeof(char)* (strlen("Federico")+1));
strcpy(p1.nombre, "Federico");

char temp[30];
printf("Ingrese una dirección: ");
scanf("%s", temp);
p1.direccion = malloc(sizeof(char)*(strlen(temp)+1));
strcpy(p1.direccion, temp);

p1.nacionalidad = malloc(sizeof(char)* (strlen("argentino")+1));
strcpy(p1.nacionalidad, "argentino");
```

En cada una de estos casos se aprovecha el tener punteros para pedir la memoria exacta que se requiere para almacenar la información.

En nombre y nacionalidad se va a copiar una cadena entonces se calcula la longitud de la misma y se pide memoria para guardarla. El +1 que aparece es para poder tener lugar para el caracter '\0'.

```
p1.nombre = malloc(sizeof(char)* (strlen("Federico")+1));
strcpy(p1.nombre, "Federico");
```

```
p1.nacionalidad = malloc(sizeof(char)* (strlen("argentino")+1));
strcpy(p1.nacionalidad, "argentino");
```

En direccion se va a almacenar el resultado de la entrada por teclado. En este caso se usa una variable temporal para leer y, se copia la cadena leída en un espacio de memoria del tamaño exacto (también con el +1 por el caracter '\0').

```
char temp[30];
printf("Ingrese una dirección: ");
scanf("%s", temp);
p1.direccion = malloc(sizeof(char)*(strlen(temp)+1));
strcpy(p1.direccion, temp);
```

Ahora bien, ¿qué pasaría si usamos un puntero para referenciar una estructura? Supongamos que tenemos esta definición:

```
struct Persona* p1;
p1 = malloc(sizeof(struct Persona));
```

Para acceder a los campos de la estructura tendríamos que hacer algo como: `(*p1).nombre`. Esta expresión nos permite ir al lugar al que hace referencia el puntero y acceder al campo de la estructura.

C nos provee una sintaxis especial para evitar escribir tantos símbolos: en vez de escribir `(*p1).nombre`, escribimos `p1->nombre`.

Estructuras

El código anterior quedaría así:

```c
int main() {
    struct Persona* p1;
    p1 = malloc(sizeof(struct Persona));

    p1->nombre = malloc(sizeof(char)* (strlen("Federico")+1));
    strcpy(p1->nombre, "Federico");

    char temp[30];
    printf("Ingrese una dirección: ");
    scanf("%s", temp);
    p1->direccion = malloc(sizeof(char)*(strlen(temp)+1));
    strcpy(p1->direccion, temp);

    p1->nacionalidad = malloc(sizeof(char)* (strlen("argentino")+1));
    strcpy(p1->nacionalidad, "argentino");

    printf("%s %s %s\n", p1->nombre, p1->direccion, p1->nacionalidad);

    free(p1->nombre);
    free(p1->direccion);
    free(p1->nacionalidad);
    free(p1);

    return 1;
}
```

O sea, este código:

```
p1->nombre = malloc(sizeof(char)* (strlen("Federico")+1));
strcpy(p1->nombre, "Federico");
```

se podría escribir así:

```
(*p1).nombre = malloc(sizeof(char)* (strlen("Federico")+1));
strcpy((*p1).nombre, "Federico");
```

Typedef

La palabra clave typedef permite declarar un tipo de dato.
La sintaxis sería, por ejemplo:

```
typedef int* pInt;
```

En este caso, pInt se reemplazaría por int*. Estas dos declaraciones serían iguales:

```
pInt a;
```

```
int* a;
```

Typedef

Se puede usar typedef para evitar tener que repetir continuamente la palabra struct.
Analicemos la siguiente definición:

```
typedef struct {
    char* nombre, *direccion, *nacionalidad;
    int edad;
} Persona;
```

En la misma sentencia se está declarando el struct y se lo está llamando Persona.
El uso sería muy simple:

```
Persona p1;
```