Programación II

Gestion de la memoria

Lic. Federico Severino Guimpel
LCC - FCEIA - UNR

Introducción

En esta presentación intentaré contestar la mayoría de las preguntas que pueden estar presentándose a partir de las teorías anteriores. Estas tienen un elemento en común: la gestión y acceso a la memoria que realiza C.

Al ejecutar un programa, el mismo se carga en la memoria física de nuestra computadora. El sistema operativo nos asigna un fragmento de dicha memoria para que podamos usarlo en nuestro programa.

Operador &

Para comenzar recordemos que el `scanf`, a veces (casi siempre) requiere de un `&` delante de la variable. ¿Por qué es esto? Porque `scanf` es poco inteligente y se le debe indicar dónde escribir lo que se lee.

```c
int i;
printf("Ingrese un número: ");
scanf("%d", &i);
```

Entonces, esto significa que `&i` representa el lugar donde está la variable `i` en memoria.

Operador &

Hagamos una prueba. Imprimamos `i` y `&i` en diferentes momentos del programa. Para poder mostrar una dirección se tiene que usar el formato `%p`. Analicemos el siguiente código:

```c
int main() {

    int i;
    printf("el valor de i es %d y su dirección es %p\n", i, &i);

    printf("Ingrese un número: ");
    scanf("%d", &i);

    printf("el valor de i es %d y su dirección es %p\n", i, &i);
    return 0;
}
```

Operador &

Si lo ejecutamos, nuestra salida será algo como esto:

```
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 valor de i es 32767 y su dirección es 0x7fffd4013074
Ingrese un número: 17
el valor de i es 17 y su dirección es 0x7fffd4013074
```

Las direcciones de memoria siempre tienen esa forma. Es un número en hexadecimal (https://es.wikipedia.org/wiki/Sistema_hexadecimal).

Operador &

Tanto el valor basura de la variable `i` que mostramos en el primer `printf` como la dirección que nos aparece puede (muy posiblemente lo haga) variar en cada ejecución. Esto es porque la variable se crea en algún lugar de la memoria. No podemos controlar dónde. Eso es potestad del sistema operativo.

```
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 valor de i es 32767 y su dirección es 0x7fffd4013074
Ingrese un número: 17
el valor de i es 17 y su dirección es 0x7fffd4013074
```

Punteros

Analicemos esto: si se está imprimiendo una dirección de memoria entonces, es un valor que se puede manipular. ¿Existirá un tipo de dato para poder guardarlo en alguna variable?
Sí, existe un tipo de dato, en C, que permite guardar direcciones de memoria.
Una variable que almacena direcciones de memoria se la llama puntero.

Punteros

Podemos ver que las direcciones de memoria son sólo direcciones, es decir, no dependen del tipo de dato que se guarde ni brindan información sobre él. Por ejemplo:

```c
#include<stdio.h>

int main() {

    int i;
    double j;
    printf("la dirección de i es %p y la de j es %p\n", &i, &j);

}
```

Si imprimimos la dirección de memoria de una variable entera y una doble obtenemos:

```
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
la dirección de i es 0x7fffde5c2bbc y la de j es 0x7fffde5c2bc0
```

Punteros

Por este motivo, si se quisiera guardar la dirección de memoria en una variable habría que indicar qué tipo de dato hay en esa dirección. La forma de declarar un puntero es:

```
int* p;
```

```
double* p;
```

```
char* p;
```

Se antepone el tipo de dato y luego se usa el *. Este caracter indica que el identificador que continúa es un puntero.

Punteros

Una aclaración importante es que el `*` no se aplica a las definiciones de varias variables. Por ejemplo:

```
int* p, i, j;
```

En este caso `p` es un puntero pero `i` y `j` son variables enteras.

Punteros

Ahora se puede hacer esto:

```c
#include<stdio.h>

int main() {

    int i;
    int* p = &i;
}
```

Es decir, guardar la dirección de la variable `i` en en la variable (puntero) `p`. Al puntero también se lo menciona como referencia ya que apunta a algún lugar de la memoria.

Punteros

Imprimamos el valor y la dirección de cada variable:

```c
#include<stdio.h>

int main() {

    int i;
    int* p = &i;

    printf("el valor de i es %d y su dirección es %p\n", i, &i);

    printf("el valor de p es %p y su dirección es %p\n", p, &p);

}
```

El valor que guarda el puntero coincide con la dirección de la variable.

```
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 valor de i es 32745 y su dirección es 0x7fffc940502c
el valor de p es 0x7fffc940502c y su dirección es 0x7fffc9405030
```

Punteros

¿Se podrá desde un puntero acceder al valor que hay en la dirección de memoria que guarda? ¡Sí se puede! El mismo operador * que usamos para declarar que la variable es un puntero se usa para acceder al valor que referencia.

```c
#include<stdio.h>

int main() {
    int i;
    int* p = &i;

    printf("el valor de i es %d y su dirección es %p\n", i, &i);
    printf("el valor de p es %p, su dirección es %p y el valor que hay en la dirección es %d\n", p, &p, *p);
}
```

```
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 valor de i es 32576 y su dirección es 0x7ffff2cff34c
el valor de p es 0x7ffff2cff34c, su dirección es 0x7ffff2cff350 y el valor que hay en la dirección es 32576
```

Punteros

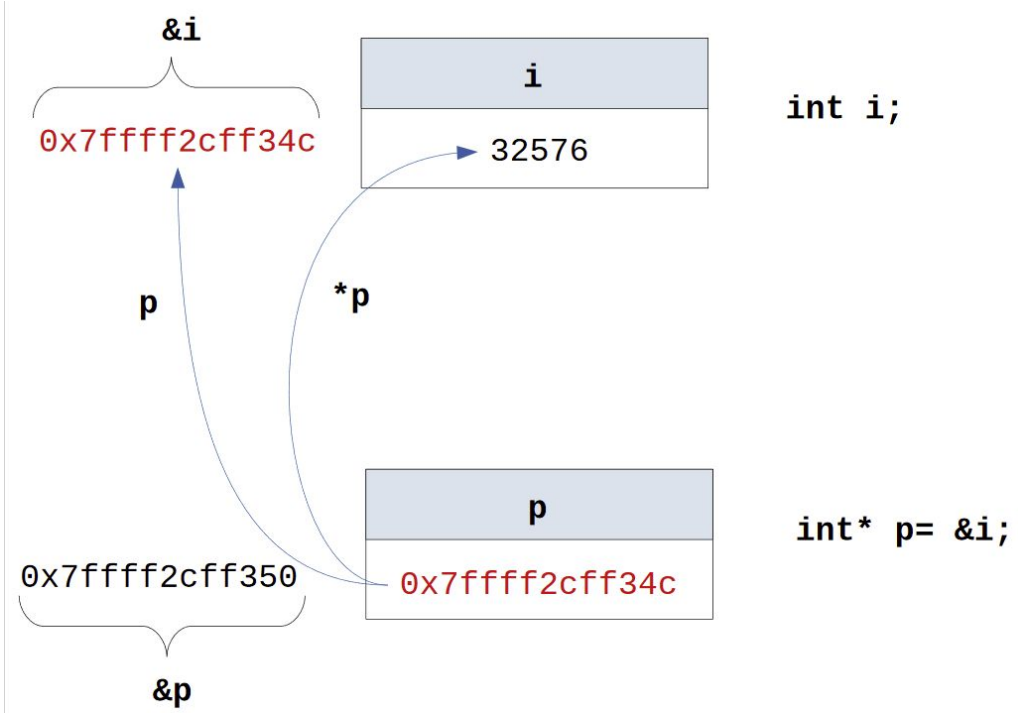
Al ejecutar confirmamos esto.

```
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 valor de i es 32576 y su dirección es 0x7ffff2cff34c
el valor de p es 0x7ffff2cff34c, su dirección es 0x7ffff2cff350 y el valor que hay en la dirección es 32576
```

Esta operación de anteponer el * al puntero se la denomina desreferencia.

Punteros

En el siguiente gráfico se puede ver el significado de esta asignación y qué representa cada operación.



Punteros

Entonces, siendo `p` un puntero a un dato de tipo `T` podemos decir que:

- `p` es de tipo `T*`
- `*p` es de tipo `T`

¿Cuál les parece que es el tipo de `&p`? La respuesta es `T**`. Ya que se está dando la dirección donde está el puntero en memoria.

Tipo `array`

Con esto que aprendimos analicemos por qué no se pone el operador `&` al leer un `array`.
El único motivo por el que no haría falta ponerlo sería que la variable ya sea una dirección. O sea, que el `array` fuera una especie de puntero. Vamos a verificar esto.

Tipo array

Veamos el siguiente código:

```c
#include<stdio.h>

int main() {
    char a[10];

    printf("El valor de a es %p y está en la dirección %p\n", a, &a);

}
```

Al ejecutarlo nos vamos a encontrar con una sorpresa.

Tipo array

El valor que guarda la variable coincide con su propia dirección:

```
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
El valor de a es 0x7fffff582a4e y está en la dirección 0x7fffff582a4e
```

Un `array` no es un puntero. C suele reemplazar al `array` por su dirección de memoria. Esto genera cierta confusión. Continuemos analizando esto.

Tipo array

Sabemos que a[0] es la primera posición del array y al anteponer el & estamos obteniendo su dirección en memoria. ¿Qué valor se piensan que tendría &a[0]?

```c
#include<stdio.h>

int main() {
    char a[10];

    printf("El valor de a es %p y está en la dirección %p\n\n", a, &a);

    printf("La dirección del primer elemento de a es: %p\n", &a[0]);
}
```

Se puede ver que los tres valores son iguales:

```
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
El valor de a es 0x7ffff8acbbde y está en la dirección 0x7ffff8acbbde

La dirección del primer elemento de a es: 0x7ffff8acbbde
```

Esto significa que el `array` está siendo reemplazado por su dirección y esta coincide con la dirección del primer elemento del `array`.

Tipo array

Entonces, podemos hacer lo siguiente:

```
char a[10];
char* p = a;
```

el puntero `p` guarda la dirección del `array`. Veamos un ejemplo para ilustrar qué significa esto.

Tipo array

Veamos el siguiente código:

```c
#include <stdio.h>

int main(){
    int a[10];
    int* p = a;

    *p = 5;
    p[1] = 10;

    printf("El primer elemento es %d y el segundo es %d \n", a[0], a[1]);

}
```

Al ejecutarlo obtenemos:

```
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
El primer elemento es 5 y el segundo 10
```

Operador []

La sentencia

```
*p = 5;
```

no hace otra cosa que ir al lugar al que apunta `p` y escribir un 5. Como `p` está apuntando a la primera posición del `array` está modificando el valor en `a[0]`.

Por otro lado, tenemos esta sentencia:

```
p[1] = 10;
```

¿Qué hace realmente el operador `[]`?

Operador []

En C, el comportamiento del operador [] es diferente del que teníamos en Python. Analicen el siguiente código. ¿Compilará? ¿Se romperá en ejecución? ¿Imprimirá algo?

```c
#include<stdio.h>

int main() {
    int a[5];
    for(int i =0; i<4; i++){
        a[i]=0;
    }
    printf("%d  %d\n", a[-1], a[5]);
}
```

Operador []

Sorprendentemente compila y muestra resultados. No serán necesariamente los mismos en cada ejecución.

```
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
4  32767
```

Pero, ¿qué estamos viendo? Claramente `a[-1]` no tiene un comportamiento como se vió en Python sobre listas (mostraba el último elemento).

Operador []

Lo que sucede es que en C el operador `[]` lo que hace es desplazarse en la memoria.

Es decir, `a[-1]` sería la posición anterior al inicio del `array a`. Por otro lado, `a[5]` sería desplazarse 5 lugares desde el inicio del `array a`. Siempre este desplazamiento es proporcional al tipo de dato asociado.

Con otras palabras, si se tiene un `array e` de enteros (int), `e[4]` implica un desplazamiento de 4 veces el tamaño que ocupa un entero en nuestra máquina. Si en cambio se tiene un `array c` de caracteres (char), `c[4]` implica un desplazamiento de 4 veces lo que ocupe un caracter en nuestra máquina.

Operador []

Veamos este programa:

```c
#include<stdio.h>

int main() {

    char a[] = "Hola";

    for(int i = -1; i< 10; i++){
        printf("Posición %p Caracter encontrado %c\n", &a[i], a[i]);
    }
}
```

Se define un `array` que se inicializa con un string de 4 caracteres.

Luego, se muestra desde la posición anterior hasta varias siguientes a la declaración.

Operador []

Al ejecutar se obtiene esto:

```
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
Posición 0x7fffff853152 Caracter encontrado
Posición 0x7fffff853153 Caracter encontrado H
Posición 0x7fffff853154 Caracter encontrado o
Posición 0x7fffff853155 Caracter encontrado l
Posición 0x7fffff853156 Caracter encontrado a
Posición 0x7fffff853157 Caracter encontrado
Posición 0x7fffff853158 Caracter encontrado
Posición 0x7fffff853159 Caracter encontrado
Posición 0x7fffff85315a Caracter encontrado
Posición 0x7fffff85315b Caracter encontrado X
Posición 0x7fffff85315c Caracter encontrado o
```

Al ser caracteres las posiciones varían en 1. Esto se debe a que cada tipo de dato tiene un tamaño asociado. Dicho tamaño puede variar en función de la arquitectura del sistema operativo.

Operador []

Ahora bien, también se puede notar que está la cadena y hay veces que imprime caracteres y a veces no. Esto es debido a que depende de lo que haya en la memoria.

```
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
Posición 0x7fffff853152 Caracter encontrado
Posición 0x7fffff853153 Caracter encontrado H
Posición 0x7fffff853154 Caracter encontrado o
Posición 0x7fffff853155 Caracter encontrado l
Posición 0x7fffff853156 Caracter encontrado a
Posición 0x7fffff853157 Caracter encontrado
Posición 0x7fffff853158 Caracter encontrado
Posición 0x7fffff853159 Caracter encontrado
Posición 0x7fffff85315a Caracter encontrado
Posición 0x7fffff85315b Caracter encontrado X
Posición 0x7fffff85315c Caracter encontrado o
```

Pero entonces, ¿se puede desplazar a cualquier lugar de la memoria así?

Manipulando direcciones

La respuesta es que no. Hay un límite. En los viejos sistemas operativos como Windows 2000 y anteriores sí se podía. Esto generaba que se pudiera leer, que no es tan riesgoso, y escribir en cualquier lugar de la memoria.
Veamos el siguiente ejemplo para ilustrar esto:

```c
#include<stdio.h>

int main() {

    char a[] = "Hola";

    for(int i = 0;  ; i++){
        printf("Posición %p Caracter encontrado %c\n", &a[i], a[i]);
    }
}
```

Manipulando direcciones

Al ejecutarlo el programa va a correr, más o menos tiempo dependiendo de cada ejecución, desplazándose carácter a carácter por la memoria. Eventualmente terminará así:

```
Posición 0x7fffe57d6ffa Caracter encontrado
Posición 0x7fffe57d6ffb Caracter encontrado
Posición 0x7fffe57d6ffc Caracter encontrado
Posición 0x7fffe57d6ffd Caracter encontrado
Posición 0x7fffe57d6ffe Caracter encontrado
Posición 0x7fffe57d6fff Caracter encontrado
Segmentation fault (core dumped)
```

Este error representa que nos fuimos del espacio de memoria asignado al programa. Cuando se quiera realizar una operación (lectura o escritura) sobre un lugar de la memoria que no está asignado al programa vamos a ver un error similar. En Windows no van a notar esto sino que se va a cerrar la ventana de ejecución.