Punteros en C

Abril Diaz Miguez

Última corrección: Febrero 2024

Apunte de la materia Algoritmos y Programación II cátedra Mendez-Pandolfo.
En éste, se presenta una introducción al tema punteros, de suma importancia en la materia, junto con ejemplos para mejor comprensión.
Se presupone que el alumno tiene conocimientos básicos de sintaxis de C.

Contents

1 Introducción **3**

2 ¿Para qué quiero punteros? **4**

3 Vamos de nuevo, ¿para qué los quiero? **5**

4 Ahora sí, ¿qué son los punteros? **6**

5 ¿Y cómo es la sintaxis de punteros en C? **9**
5.1 ¿Cómo declaro un puntero? 9
5.2 ¿Cómo desreferencio un puntero? 10

6 Ejemplo Iluminador 1: Swap **12**

7 Ejemplo Iluminador 2: Vectores **14**

8 Ejemplo Iluminador 3: Puntero a mi estructura **15**

9 Ejemplo Iluminador 4: Puntero a void **19**

10 Ejemplo iluminador 5: Puntero a archivo **20**

11 ¿Por qué todavía no hablaste del heap? **22**

12 Preguntas típicas de punteros **23**

1 Introducción

En este apunte, se presenta una introducción a los punteros, tanto conceptualmente como su uso y sintaxis en el lenguaje de programación C. Se presupone que el lector tiene conocimientos de manejo y organización de memoria; de todos modos, se recomienda leer el apunte de memoria de la cátedra antes de avanzar con éste.

2 ¿Para qué quiero punteros?

En el apunte de memoria, se repasó el concepto de variable, tanto local como global. Detallamos que a una variable se la puede pensar como una cajita que almacena información. También se explicó el concepto de *stackframe*, y cómo las variables locales solamente se pueden modificar en el stackframe donde viven.

Pero tal vez no quedó del todo claro el porqué. ¿Por qué una variable local solamente se puede modificar en el stackframe donde vive? ¿Qué me impide modificarla desde cualquier lugar de mi programa? Una parte de la respuesta es lo que se vio en el apunte de memoria: al llamarse a una nueva función, se genera un nuevo stackframe en el stack para esa función en particular. Como sólo se puede interactuar con el stack por el tope, si se quiere acceder a algo fuera de ese stackframe, y que por lo tanto fue declarado antes de todo lo necesario para la nueva función, se debería eliminar todo lo pertinente a la función que ahora se está ejecutando, lo cual no es el objetivo.

¿Y qué? Vos podrías preguntarte. ¿Qué importa que una variable no pueda ser accedida desde otro stackframe? Si yo llamé a una función auxiliar en el medio de mi función, es por algo, ¿no? ¿Para qué quiero modificar o acceder a cosas que yo declaré en otras funciones? Para eso, accedo o modifico en mi propia función, y listo.

Este planteo es válido... hasta que empezás a trabajar en proyectos más grandes. La idea de tener una única función gigante que modifique todo es increíblemente inconveniente en el mejor de los casos, e imposible en el peor.

Por un lado, se desea que el código sea legible y fácilmente mantenido, para tu sanidad mental y la del que revise tu código en el futuro (que bien podrías ser vos). Una de las prácticas que se promueve para alcanzar estos objetivos es la de modularizar y reutilizar código. Se busca que el código no esté en una gran función llena de cosas, sino que se divida el código en funciones y demás, que puedan ser llamadas y utilizadas múltiples veces a lo largo del programa. Esto implica que tu código es más corto, porque lo que sin reutilización tenés que programar varias veces, acá lo programás una; también hace que el código sea más fácilmente debuggueado y mantenido, detalle no menor, ya que si cierto comportamiento no funciona, hay una o muy pocas funciones donde puede encontrarse el error, en vez de estar todo desperdigado y mezclado. Finalmente, vas a ver en la materia y a lo largo de la carrera conceptos como TDA y clases, que implican por naturaleza la declaración de una variable en un stackframe, que luego será modificado en otros.

Así que suponete que insistimos en que necesitamos modificar una variable que creamos en un stack, por ejemplo, el número `int num = 5` en otra función. Por ahora, creeme que lo vamos a necesitar. Ya vas a ver ejemplos en este y otros apuntes, a su vez que a lo largo de la materia, donde es necesario. Si realmente quisieses acceder y/o modificar una variable de un stackframe en otro, alguien que trabajó previamente con Python podría plantear que se puede pasar la variable por parámetro, modificarla en esa nueva función, y listo.

Por ejemplo, esta persona podría plantear el siguiente código:

```
def sumador(x, y):
    x = x + y

def main():
    num = 5
    sumador(num, num)
    print("Nuevo num:", num)
```

Y si lo ejecuta en Python, esto funciona perfectamente. Podría entonces decirnos que hagamos lo mismo en C, algo así

```
#include <stdio.h>

int sumador(int x, int y) {
    x = x + y;
}

int main() {
    int num = 5;
    sumador(num, num);
    printf("Nuevo num: %d\n", num);
    return 0;
}
```

Si ejecutás este programa, te darás cuenta de que... no funciona. Pero, ¿por qué? Si el código es equivalente, o así lo parece a primera vista, ¿cuál es el problema? Estoy pasando la variable de mi función `main` a la función `sumador`, ahí la modifico, y vuelvo a la función `main`. ¿Qué está mal?

Hay una característica de C que no tomamos en cuenta, y es la que hace que esto no funciona. Y es la que nos lleva al concepto y el uso del tema principal de este apunte: punteros.

3 Vamos de nuevo, ¿para qué los quiero?

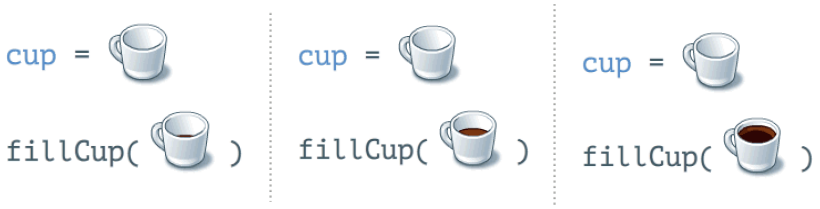
Hasta ahora, lo más probable es que hayas creado variables en una función y hayas modificado esa variable en ese mismo stackframe. En este caso, vos tenés acceso directo a la variable y siempre que quieras leerla o cambiarla, tenés completo acceso a ella. Por otro lado, sabés que si querés que otra función conozca este valor, podés mandar la variable por parámetro para que esta otra función tenga una copia del valor. Pero fijate que dije *copia*. En C sólo existe el pasaje por copia de las variables. Si yo tengo una variable en una función y la paso por parámetro a otra función, esta otra función crea en su stackframe otra variable, que nada tiene que ver con la original, con el mismo valor que se pasó por parámetro. Pero son diferentes variables, son independientes. Modificar una no implica modificar la otra.

Como todo es mejor cuando tomamos algo caliente, miremos si con una

buena taza de café entendemos mejor la diferencia. Acá tenemos nuestra taza y queremos llenarla... bueno, con café. Para eso, programamos una función que lo haga y pasamos por copia la taza.

Paso la taza original por copia a la función que llena la taza.

Pero vemos que, como pasamos una copia de la taza, cuando ésta se llena, la original no se modifica.



La taza copia se llena en la función, pero la taza original no se modifica.

Y por supuesto que no se llena, son dos tazas distintas. Son independientes entre sí, por lo que si modifico la copia que le pasé a la función, la original no cambia.

Bajando el ejemplo a tierra, al pasar una variable por copia, estamos pidiéndole a la máquina que cree otra variable en el stackframe de la función nueva, con el mismo valor que estamos pasándole nosotros. Si modificamos ese valor en la nueva función, el valor original no se verá afectado, porque son dos espacios distintos de memoria.

Aquí es donde entran los punteros.

4 Ahora sí, ¿qué son los punteros?

Los *punteros* son variables que contienen una dirección de memoria. Así como `num` contiene el valor `5`, una variable de tipo puntero contiene una dirección de memoria. Si recordamos la analogía de la calle con las casas, tendríamos una casa que contiene la dirección de otra casa, así

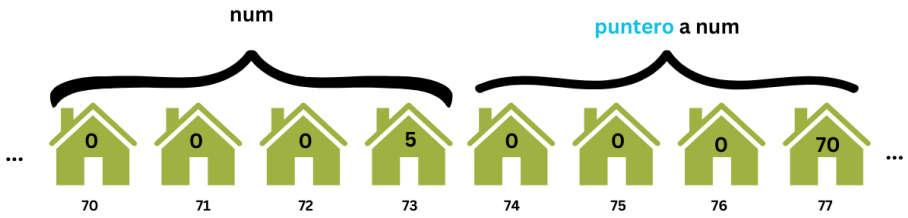


Figure 1: Variable vs puntero a variable en memoria

Donde los primeros cuatro bytes, del 70 al 73 inclusive, se utilizan para almacenar la variable `num`, que contienen el valor 5[1]. Luego se declara otra variable, esta vez un puntero, y se le da la dirección de memoria de la variable `num`. Por lo tanto, ¿qué se almacena en las casitas que le corresponden al puntero? La dirección de memoria de la variable que es apuntada por el puntero. En nuestro caso, la variable que es apuntada es `num`, por lo que se almacena el byte de menor dirección que le corresponde a esa variable, 70 en el ejemplo.

En el apunte de memoria, cuando presentamos esta analogía, mencionamos que con tener la dirección de memoria de una variable, podemos ir a buscar el dato que allí se encuentra. No hay dos casitas con la misma dirección. Y si queríamos ir a la data en sí que se encuentra en memoria, podíamos hacerlo utilizando el nombre de la variable: el compilador se encargaba de traducir ese nombre de variable en la dirección de memoria correspondiente. Ésta era la única forma de ir a buscar un dato en memoria : necesitabas estar en el stackframe correcto y usar el nombre de la variable correspondiente. Pero fijate en el gráfico, ahora hay otra variable que contiene la dirección de memoria. No sólo el compilador la tiene guardada en su tablita de conversiones, sino que la tenemos nosotros en una variable que nosotros declaramos. Estaría genial poder usarla como el compilador, ¿no? Y acceder a variables a través de estos punteros, en vez de la variable en sí...

Sí. Para eso son los punteros. Los punteros son variables que nos permiten acceder a otros espacios de memoria. ¿A cuál nos permiten acceder? A aquél que es apuntado por el puntero, claro. En el ejemplo, el puntero a `num` nos permite acceder a num. Si queremos acceder a otra cosa, debemos pisar el valor del puntero con otra dirección de memoria.

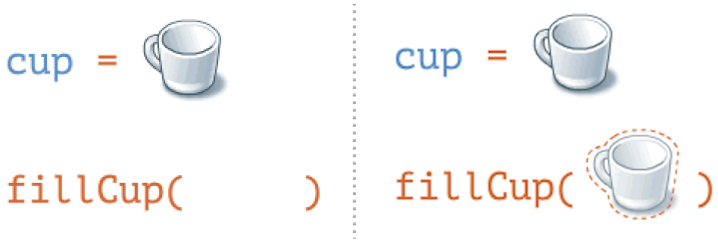
¿Y cómo accedemos al dato, sabiendo la dirección de memoria? Conceptualmente, conociendo la dirección de memoria, sabés a qué espacio de la memoria ir y revisar. Se dice que se "desreferencia" un puntero a la acción de "moverse a la dirección de memoria correcta". No te preocupes por la sintaxis, ya vamos a ver cómo se hace esto en C.

Pero todo esto, ¿qué tiene que ver con nuestro café duplicado? ¿Cómo puedo usar punteros para que se modifique mi taza original, no la copia que le mandé a la función?

[1]Por supuesto que en memoria en realidad está el valor en binario, pero para simplificar el ejemplo, dejé cada byte en decimal

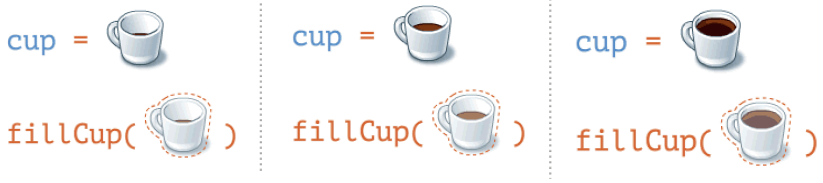
Se nos ocurre entonces una nueva idea: en vez de pasar por copia la variable en sí, la taza, podemos pasar la dirección de memoria de la misma. Sí, será una copia de la dirección de memoria actual, pero sigue siendo la misma dirección de memoria, copia u original.

Veámoslo gráficamente. Empezamos con la taza que debemos pasar por parámetro a la función, pero ahora no pasamos la taza en sí, sino que pasamos la dirección de memoria de la taza.



Paso un puntero a la taza original a la función.

Si ahora la función utiliza el puntero para llenar la taza original, vemos cómo la taza original se modifica.



La taza original se modifica.

Bajando el ejemplo a tierra nuevamente, lo que ocurre es que en la nueva función estoy accediendo a la misma porción de memoria que reservé en la función original. Como mi nueva función recibe una dirección de memoria a una taza, no la taza en sí, puede ir a buscar la taza original y modificarla directamente.

Esta comportamiento abre un mundo de posibilidades, e incluso si en lenguajes de más alto nivel no se encuentran explícitamente denominados, que sepas que por detrás de cámaras se están utilizando punteros constantemente.

5 ¿Y cómo es la sintaxis de punteros en C?

Hasta ahora venimos hablando de conceptos, con la intención de comprender la idea de qué es y para qué sirve un puntero, más allá de cómo declararlos y utilizarlos en la práctica. Pero ésta es una materia de programación, donde vas a tener que programar con punteros correctamente, por lo que me parece pertinente tener un apartado dedicado a la sintaxis y al correcto uso y manejo de punteros.

Debido a que la materia se dicta en el lenguaje de programación C, me centraré en resumir la sintaxis de punteros en C. Esta convención también aplica a otros lenguajes, como C++, pero que se sepa que la sintaxis es arbitraria y depende de cada lenguaje.

5.1 ¿Cómo declaro un puntero?

Un puntero se declara como cualquier otra variable, determinando su tipo y su nombre. El tipo de un puntero se le dice "puntero a X", que puede ser "puntero a `char`", "puntero a `int`", etc. Mencionar `char`, `int` o cualquier otro tipo, implica que el elemento que es apuntado por el puntero debe ser tratado como un char o un int respectivamente, por lo que se le pueden aplicar operaciones de chars e ints resp.

Pero tener una variable puntero con contenido basura no nos es útil. Queremos darle un valor que nos sea de interés. Para hacer que un puntero apunte a otra variable ya declarada, utilizamos el operador `&` en la variable que queremos. Este operador nos devuelve la dirección de memoria de la variable que agregamos después del operador.

Figure 2: Declaración de un puntero que apunta a otra variable

5.2 ¿Cómo desreferencio un puntero?

Desreferenciar un puntero es acceder a la dirección de memoria que el puntero tiene almacenado. Es partir de la dirección y llegar a la casita correspondiente, viendo lo que contiene. Por lo tanto, si se parte de este código

```
int id_leido = 1243;
int* id_cliente = &id_leido;
```

Imprimir esto

```
printf("El id_leido es: %d\n", id_leido);
```

e imprimir esto

```
printf("El *id_cliente es: %d\n", *id_cliente);
```

muestra el mismo número dos veces. Y si más adelante en el código la variable `id_leido` y se vuelven a imprimir ambas variables, se volverán a imprimir los mismos valores. Y si más adelante aún se modifica lo que hay en `*id_cliente`, también el cambio se verá reflejado en los dos prints. ¿Por qué? Porque estoy mostrando lo que hay en la misma dirección de memoria, en la misma casita, dos veces.

Desreferenciar un puntero, entonces, se consigue agregando el asterisco a la izquierda del nombre.

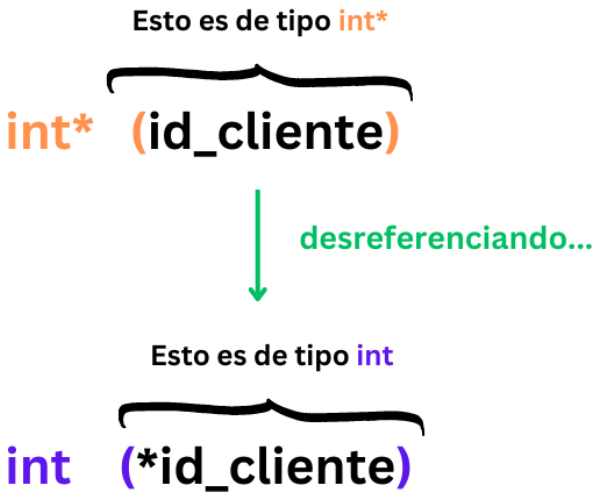


Figure 3: Desreferenciando un puntero

Pero ésa no es la única forma. También se puede hacer con el operador `[]`, que veremos en más detalle en el ejemplo de vectores. A su vez, existe el

operador `->` para desreferenciar un puntero, pero éste se utiliza en ciertos casos solamente, que se explicarán en el ejemplo de estructuras.

6 Ejemplo Iluminador 1: Swap

Tengo dos variables y quiero intercambiar su contenido.

```
int num_favorito = 19;
int segundo_num_favorito = 3;
```

Podría hacerlo dentro de la función donde declaré esos dos números y listo, ¿no?

```
int num_favorito = 19;
int segundo_num_favorito = 3;

int aux = num_favorito;
num_favorito = segundo_num_favorito;
segundo_num_favorito = aux;
```

Pero resulta que este código de swappeo es súper útil, y se usa un montón en tu código. Y como todo código que es reutilizado, debe ser modularizado. Por lo tanto, queremos meterlo en otra función, algo así

```
void swap(int num1, int num2) {
    int aux = num1;
    num1 = num2;
    num2 = aux;
}

int main() {
    int num_favorito = 19;
    int segundo_num_favorito = 3;

    swap(num_favorito, segundo_num_favorito);

}
```

Pero si ejecutás este código en tu máquina, te darás cuenta de que no funciona... y eso se debe a esa característica de C que ya se explicó anteriormente: en C sólo existe el pasaje por copia de las variables. Es decir, la variable `num1` que estoy modificando en `swap` no es la misma variable que pasé como primer parámetro (`num_favorito`) en `main`.

Lo que está pasando a nivel memoria es que se creó un nuevo stackframe y se copió en otra zona de memoria los valores de las dos variables originales. Pero si se imprimiese la dirección de memoria de `num1` y se la comparase con la de `num_favorito`, veríamos que no coinciden. Por lo tanto, si modifico una variable, la otra no se ve afectada: son dos zonas distintas de memoria.

¿Cómo puedo, entonces, modificar mis variables de `main` en la función `swap`, para que ésta sea realmente reutilizable y me sirva para swappear dos enteros cualesquiera de mi código?

Ya vimos que modificar las variables en la función no funciona. Siya trabajaste con Python o un lenguaje similar, se te ocurre devolver los dos valores nuevos y pisar los dos viejos. Pero este código, perfectamente factible en Python

```
def swap(a, b):
    return b, a

x = 5
y = 20
x, y = swap(x, y)
```

en C no está permitido. Éste es un caso particular de un problema más grande: el lenguaje C es no permite que se devuelvan múltiples variables de una misma función. Se puede devolver una o ninguna variable en el `return`. Por lo tanto, ¿cómo se implementa una función en C que efectivamente modifique más de una variable pasada por parámetro? Bueno, la pregunta está en un apunte de punteros, así que claramente la respuesta es: usando punteros.

Si en vez de pasar la variable en sí, paso un puntero a esa variable, puedo modificar el pedacito de memoria original. Estoy yendo a la casita correcta, a la que tiene el valor que quiero cambiar. El código sería el siguiente:

```
#include <stdio.h>

void swap(int* num1, int* num2) {
    int aux = *num1;
    *num1 = *num2;
    *num2 = aux;
}

int main() {
    int num_favorito = 19;
    int segundo_num_favorito = 3;

    printf("\\n-Primer entero: %d\n
            -Segundo entero: %d\n",
            num_favorito, segundo_num_favorito);

    swap(&num_favorito, &segundo_num_favorito);

    printf("Swapeo los enteros. Ahora, \n
            -Primer entero: %d\n
            -Segundo entero: %d\n",
            num_favorito, segundo_num_favorito);

    return 0;
}
```

¡Probalo localmente y comprobalo!

7 Ejemplo Iluminador 2: Vectores

En materias anteriores ya habrás trabajado con vectores de elementos. Sabés que en C un vector se declara así

```
int vector[3] = {31, 22, 63};
```

donde el vector tiene tres enteros, `31`, `22` y `63`. Supongamos que quiero cambiar los elementos de ese vector. Quiero pisar el vector con los números naturales.

Podríamos tener todo el código en la misma función main, así

```
int main() {
    int vector[3] = {31, 22, 63};

    for (int i = 0; i < 3; i++) {
        vector[i] = i;
    }

    return 0;
}
```

... pero eso no sería reutilizable. Así que lo vamos a abstraer en su propia función. Primera idea que se te ocurre es hacer esto

```
void pisar_vector_con_naturales(int v[3]) {
    for (int i = 0; i < 3; i++) {
        v[i] = i;
    }
}

int main() {
    int vector[3] = {31, 22, 63};
    pisar_vector_con_naturales(vector);

    return 0;
}
```

y cuando lo probás, resulta que... ¡funciona! Pero, ¿por qué? Si vimos que si pasamos solamente la variable estamos pasando una copia, ¿no? ¿Por qué con esta variable funciona y con las del ejemplo anterior no?

La clave está en que esta variable es un vector de enteros, mientras que las otras eran enteros. Al pasar la variable `vector`, en realidad no estás pasando una copia del vector: estás pasando una copia de la dirección de memoria del primer elemento del vector. Las variables `vector` y `v` en contienen la dirección de memoria del primer elemento del vector original, `vector`. Exactamente, los vectores en realidad son punteros. Si a esa variable se la desreferencia con el operador `[]`, obtenemos el valor que se encuentra en esa dirección de memoria, en esa casita.

Ya viste en la sección de sintaxis que es posible desreferenciar un vector con el operador `*`. También es posible hacerlo con el operador `[]`, y esta última es la que generalmente se utiliza con vectores. ¿Se puede usar el `*` para desreferenciar un vector? Sí, se puede, pero es más difícil de leer. Aquí te dejo algunas equivalencias para que veas la diferencia.

Si el vector se declaró como

```
int vector[3] = {1, 2, 3};
```

Podemos obtener el primer elemento del vector, con valor `1` en la posición `0` de las siguientes maneras:

```
printf("El primer elemento es: %d\n", vector[0]);
```

```
printf("El primer elemento es: %d\n", *(vector+0));
```

En este caso, el `+0` en realidad es opcional, pero lo explicito para que se vea más adelante el patrón.

Para obtener el segundo elemento, podríamos agregar cualquiera de las dos líneas siguientes:

```
printf("El segundo elemento es: %d\n", vector[1]);
```

```
printf("El segundo elemento es: %d\n", *(vector+1));
```

Similarmente, para obtener el tercer y último elemento del vector, haríamos

```
printf("El tercer elemento es: %d\n", vector[2]);
```

o

```
printf("El tercer elemento es: %d\n", *(vector+2));
```

8 Ejemplo Iluminador 3: Puntero a mi estructura

Hasta ahora en el apunte venimos hablando de estructuras nativas de C. Aquellas que ya vienen con el lenguaje, y que nosotros no necesitamos definir, el compilador ya las conoce. Pero ésas no son las únicas estructuras que podemos utilizar. De ser el caso, agrupar información que deba ir junta sería demasiado complicado.

Imaginate que estás implementando un proyecto que implica tener información almacenada de los empleados de la empresa. Un cliente tiene un nombre y apellido, un id, un puesto, un sueldo, dirección de correo electrónico, y mucha más data. El nombre, apellido, puesto y dirección de email se almacenarán en vectores de chars (en C no existe el dato string nativo), y el id y sueldo, en ints.

Pero si tuviese que declarar cada variable por separado, y siempre que quiera tratar algo del cliente, pasar todos los datos por parámetro, sería un lío, tendría demasiadas variables dando vuelta.

```
void mostrar_empleado(id,
                      nombre,
                      apellido,
                      sueldo,
                      correo_electronico,
                      puesto) {
    <...>
}


int main() {
    int id = 1;
    char nombre[] = "Juan";
    char apellido[] = "Perez";
    int sueldo = 500000;
    char correo_electronico[] = "juan_perez@empresa.com"
    char puesto = "Semi-senior developer";

    mostrar_empleado(id,
                     nombre,
                     apellido,
                     sueldo,
                     correo_electronico,
                     puesto);


    return 0;
}
```

Tener todos estos datos sueltos ya de por sí dificulta muchísimo la lectura, pero imaginate con cientos o miles de empleados. Se hace inmanegable. Y ni siquiera menciones el hecho de que si te llegás a confundir a la hora de enviar alguno de los datos, el programa no funcionaría y no tendrías idea a primera vista de porqué. En general, estos bugs son difíciles de encontrar, ¿a quién se le ocurriría que rompió todo porque mandaste mal una variable?

Es por esto que existen los `structs` en C[2]. Los structs permiten que declares nuevos tipos de variables, compuesto por tipos nativos a C (`char`, `int`, `float` y demás) y/o por tipos ya creados previamente por vos.

Entonces, si quisiésemos mejorar nuestro código, podríamos declarar un struct de tipo `empleado` que se viese así, y usarlo de la siguiente manera:

[2]En otros lenguajes hay estructuras similares pero diferentes, como las clases en C++ y en otros lenguajes orientados a objetos, como Python y Java. Pero eso lo verás en próximas materias

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

//ESTA ES LA DEFINICIÓN DEL ELEMENTO
typedef struct {
    int id;
    char nombre[50];
    char apellido[50];
    int sueldo;
    char correo_electronico[100];
    char puesto[50];
} empleado;

//PUEDO RECIBIR ESTE TIPO DE DATO POR PARAMETRO
void mostrar_empleado(empleado emp) {
    printf("ID: %d\n", emp.id);
    printf("Nombre: %s\n", emp.nombre);
    printf("Apellido: %s\n", emp.apellido);
    printf("Sueldo: %d\n", emp.sueldo);
    printf("Correo Electrónico: %s\n", emp.correo_electronico);
    printf("Puesto: %s\n", emp.puesto);
}


int main() {
    //DECLARO UN NUEVO EMPLEADO
    empleado emp;

    //CARGO CADA UNO DE SUS CAMPOS
    emp.id = 1;

    //PARA LOS STRINGS, UTILIZO FUNCIONES DE STRING
    strcpy(emp.nombre, "Juan");
    strcpy(emp.apellido, "Perez");
    emp.sueldo = 500000;
    strcpy(emp.correo_electronico, "juan_perez@empresa.com");
    strcpy(emp.puesto, "Semi-senior developer");


    mostrar_empleado(emp);

    return 0;
}
```

Vemos cómo el código para crear y pasar empleados es más limpio. Inicializar una de estas estructuras será laborioso siempre, no te queda otra que revisar cada atributo y llenarlo con el valor correcto. Pero en el futuro no le darás valores literales, sino que leerás esos datos de un archivo o de una base de datos, o incluso te la pasarán desde otra computadora. El primer caso se verá en el

curso, los otros dos los dejamos para futuras materias.

Tal vez te estás preguntando ¿qué tiene que ver todo esto con punteros? Hasta ahora sólo expliqué un nuevo concepto, que nada que ver con punteros tiene, y su sintaxis en C. ¿Por qué lo saco a colación?

Repasamos estos conceptos y su sintaxis para entender cuándo se usa la tercera forma de desreferenciar un puntero. El operador `->` se utiliza cuando se quiere acceder a un atributo de una estructura que fue pasada por puntero. En el ejemplo en código, en la función `mostrar_empleado` se utiliza el operador `.` para acceder a un atributo de un empleado. Pero en este caso, el empleado es pasado por copia, no por puntero. Si en cambio se hubiese pasado un puntero a ese empleado, el código anterior se debería modificar de la siguiente manera:

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

typedef struct {
    int id;
    char nombre[50];
    char apellido[50];
    int sueldo;
    char correo_electronico[100];
    char puesto[50];
} empleado;

//AHORA RECIBO UN PUNTERO A EMPLEADO
void mostrar_empleado(empleado* emp) {
    printf("ID: %d\n", emp->id);
    printf("Nombre: %s\n", emp->nombre);
    printf("Apellido: %s\n", emp->apellido);
    printf("Sueldo: %d\n", emp->sueldo);
    printf("Correo Electrónico: %s\n", emp->correo_electronico);
    printf("Puesto: %s\n", emp->puesto);
}


int main() {
    empleado emp;

    emp.id = 1;
    strcpy(emp.nombre, "Juan");
    strcpy(emp.apellido, "Perez");
    emp.sueldo = 500000;
    strcpy(emp.correo_electronico, "juan_perez@empresa.com");
    strcpy(emp.puesto, "Semi-senior developer");


    //AHORA ENVIO LA DIR DE MEMORIA DEL EMP
    mostrar_empleado(&emp);

    return 0;
}
```

9 Ejemplo Iluminador 4: Puntero a void

Uno de los temas más temidos de punteros es el atemorizante puntero a void (chan chan chan). Pero espero que cuando termines de leer esta sección, entiendas su propósito y veas lo útiles que son. O al menos, te sirva de introducción para cuando tengas que realmente usarlos.

Ya sabés que los punteros tienen tipo: si quiero una variable que apunte a un entero, el puntero tiene que ser de tipo puntero a entero. Lo mismo si quiero

una variable que apunte a un char, o a un float, o a una estructura mía que yo creé.

Pero habrá casos donde no sabremos qué tipo de elemento es apuntado por nuestro puntero. O más aún, quiero un puntero que sea versátil, y me deje apuntar a diferentes tipos de elementos sin tener que crearme uno nuevo cada vez. Para estos casos, existe el llamado "puntero a void". Un puntero a void es un puntero que no le indica al compilador qué tipo de dato es apuntado por el puntero; por lo tanto, el compilador no sabe el tamaño del dato al que estás apuntando. Esto nos permite reutilizar el mismo puntero para apuntar a diferentes tipos de datos, por ejemplo

```
int num = 45;
char letra = 'A';
char saludo[] = "Hola";

void *ptr_comodin = &saludo;
ptr_comodin = &letra;
ptr_comodin = saludo;
```

y también nos permite tener un puntero apuntando a un tipo de dato que nosotros no conocemos, lo cual es mucho más interesante para nosotros en esta materia. Este caso en particular lo vas a ver en profundidad cuando implementes estructuras de datos: listas, pilas, colas, árboles, hashes, etc., estructuras que sirven para contener elementos. Pero, ¿qué elementos? ¿De qué tipo? Del que el usuario quiera. Como un `void*` puede apuntar a cualquier tipo de dato, no necesitamos condicionar al usuario pidiéndole que sólo ingrese un tipo de dato en su lista; puede ingresar lo que quiera.

Eso sí, un detalle no menor es que el compilador no sabe el tamaño a lo que el puntero está referenciando. Esto implica que no se puede desreferenciar directamente un `void*` porque ¿cómo sabe el compilador cuántos bytes agarrar de memoria? Tal vez el puntero está apuntando a un char, por lo que se debería buscar un byte, pero si el puntero está apuntando a un int, se deberían buscar cuatro bytes. ¿Cómo puede saber eso el compilador?

La respuesta es: diciéndoselo nosotros. Y eso se hace casteando la variable. Le explicitamos al compilador que al `void*` lo trate como un `char*` o un `int*`, así sabrá cuántos bytes tomar de memoria para manipular esa variable.

10 Ejemplo iluminador 5: Puntero a archivo

Como último ejemplo, muestro que otra de las grandes librerías de C que utiliza punteros es la de archivos. Para manipular archivos en C se utilizan funciones como `fopen`, `fclose`, `fwrite` y `fread`, y todas reciben o devuelven un puntero a `FILE`. Es decir, un `FILE*`. El porqué es multifacético, como casi todo en programación.

Por un lado, la eficiencia. La data de un archivo puede ocupar desde KB hasta MB, muchísimo más que los honestos 4 u 8 bytes que ocupa un puntero.

Esto de por sí ya es una gran ventaja, ya que copiar un puntero de un stackframe a otro es mucho más performante que copiar KB o MB de data.

A su vez, y tal vez incluso más importante que la performance, es el hecho de que tener un puntero a una estructura le otorga un alto nivel de flexibilidad y abstracción al código, mucho más que pasar una copia de la estructura en sí. Al utilizar punteros, se le otorga al usuario una interfaz consistente, donde el programa no necesita saber los detalles de la implementación de la estructura base, con saber que en algún lugar está implementada es suficiente. De esta forma, los diferentes sistemas operativos pueden implementar la estructura `FILE` como más les convenga, e incluso pueden agregarle cosas a la implementación base de la estructura sin tener que recompilar o modificar de ninguna manera el código de las librerías estándar.

11 ¿Por qué todavía no hablaste del heap?

El heap es uno de los cuatro componentes de un programa. Es un espacio de memoria, como el stack, pero con una importante diferencia: es el programador el encargado de pedir memoria en el heap y de, más tarde, liberarla. Cuando el programador pide memoria en el stack, el propio sistema operativo se encarga de liberar toda la memoria pedida en él. No es el caso de la memoria pedida en el heap; ésta debe ser pedida y liberada por el programador. Para acceder a la memoria del heap, es necesario hacerlo a través de punteros, por eso ambos temas son muy asociados.

Si ya tenés conocimiento previos de punteros, memoria o cursaste ya una parte de la materia y querés refrescar el tema de punteros, probablemente te estés preguntando ¿por qué no mencionó el heap hasta ahora? ¿No es para eso que se crearon los punteros? ¿Para usar el heap?

Y como te habrás dado cuenta a lo largo del apunte, la respuesta es que no. No necesitás el heap para que los punteros sean útiles. Los punteros de por sí son una herramienta extremadamente útil, que resuelve éstos y muchos otros problemas que no mencioné en el apunte (principalmente para no extenderlo demasiado, pero un ejemplo es el de punteros a función, tema del cual ya hay un apunte provisto por la cátedra). Por supuesto que si mezclás punteros con el heap, dos herramientas poderosísimas de por sí, ganás mucho control y versatilidad en tu código, pero que quede explícito: se puede utilizar punteros, aún sin heap.

12 Preguntas típicas de punteros

Me gustaría terminar este apunte con una serie de preguntas y respuestas típicas de punteros.

- **¿Esto implica que ahora tengo que usar siempre punteros?** No. Hay casos donde es indispensable usar punteros, como en el caso de swap, que no se puede resolver sin ellos. También es cierto que pasar un puntero (de 4 u 8 bytes, dependiendo la arquitectura) es mucho más sencillo para la máquina que pasar una estructura más grande, por lo que a nivel performance también es recomendable. Pero como todo, tiene sus contras. Principalmente es el hecho de que estás modificando el valor original; una vez que modificás el espacio de memoria original, no es posible recuperar el valor anterior, se perdió para siempre. Si necesitás poder volver al valor original o mantenerlo en caso de error, debés tener mucho cuidado al trabajar con punteros.

- **¿Pero entonces, cuándo uso punteros y cuándo uso pasaje por copia?** Como todo en informática, depende. Si lo que necesitás es que una variable se modifique en un stackframe donde no se creó (pensá el caso de swap), no te queda otra que usar punteros. Si vos sólo querés consultar un valor, como un int, no es necesario que uses punteros, está perfecto pasar el valor por copia. Ahora, si la estructura comienza a ser más grande, o es una función que es llamada muchas veces (por ejemplo, en un loop), no sería mala idea considerar si pasar un puntero mejoraría la performance, aunque no el comportamiento. Como siempre, depende de la situación. Lo irás viendo con la práctica.

- **¿Si el puntero almacena una dirección de memoria, puedo inventarme una dirección yo y dársela a un puntero?** ¿Decís hacer algo así?

  ```
  int* ptr_a_algo = 0xDEADBEEF;
  ```

  Si ésa es la pregunta, la respuesta es depende del lenguaje. Lenguajes de más alto nivel pueden no permitir que asignes un puntero a una dirección arbitraria, pero C sí te deja. Eso sí, si intentás acceder a lo que se encuentra en esa dirección de memoria, lo más probable es que no tengas permisos y el sistema operativo te niegue el acceso, lanzando un Segmentation Fault y cortando el programa. No te lo recomiendo, pero poder, podés.

- **Si los punteros son necesarios solamente cuando quiero modificar una variable en una función donde no la creé, puedo sólo modificar mis variables en los stacks donde las creé y listo, no necesito punteros, ¿no?** Lamento decirte que eso es impracticable. Tener una única función que haga todo no sólo viola muchos principios de la programación, sino que aparte no siempre es siquiera aplicable. Hay problemas

que sólo se resuelven con punteros, como vas a ver cuando implementes estructuras de datos. A su vez, hay muchas prácticas y recomendaciones en la programación que se basan en la modularización y reutilización de código. El objetivo es que tu código sea simple, fácil de leer, debbuguear y de expandir en un futuro. Cuando trabajás con proyectos muy chicos, un archivo y un main te pueden ser suficientes. Pero cuando empieces a toparte con problemas más grandes, que requieren una solución más compleja, será imposible mantener todo en una sola función. Y ahí vas a usar punteros.

- **No entendí el ejemplo del vector, ¿un vector y un puntero en C son lo mismo?** No, no son lo mismo. Pero son tan parecidos que es fácil confundirlos. Sabés que un puntero en C es una variable con una dirección de memoria. Bueno, un vector es una variable que contiene la dirección de memoria del primer elemento del vector, y que a su vez reservó el espacio suficiente en el stack para almacenar todos los elementos que se pidió a la hora de declararlo. Entonces, al ejecutar la línea

```
int ids[3] = {142, 225, 34};
```

  En el stack se reservan (3 elementos * 4 bytes cada elemento=)12 bytes en total, y en la variable `ids` se almacena la dirección de memoria del primer elemento, el elemento en la posición 0.