.UBAfiuba

FACULTAD DE INGENIERÍA

Apunte de punteros a función

[7541/9515] Algoritmos y Programación II

Abril Diaz Miguez

¿Qué son?

Antes de hablar de punteros a funciones, repasemos el concepto de punteros.

¿Qué era un puntero?

Un puntero es una variable que almacena una dirección de memoria. Vamos de nuevo, un puntero es una variable, que como dato tiene dónde está localizado algo. Ese algo puede ser cualquier cosa, una variable, parte de un archivo mapeado en memoria, parte del mapeo de un input en memoria (como un joystick, un teclado o una placa de video, dispositivos con los que se puede interactuar a través de cierto espacio que reservan de memoria), y sí, una función.

En C, a diferencia de otros lenguajes, se permite asignar un puntero a una dirección arbitraria, por lo que podemos tener una variable con cualquier dirección de memoria que nosotros querramos. ¿Incluso podemos tener un puntero a una dirección a memoria que no le pertenece a nuestro programa? Sí, así que hay que manejarlos con cuidado.

Con eso dicho, **se diferencian dos tipos de punteros, que son los punteros a datos y los punteros a funciones**. Los diferenciamos porque actúan distintos, y no podemos pasar de un puntero a otro. Es decir, no podemos castear un puntero a dato para que sea un puntero a función, ni viceversa.

Los que venimos viendo hasta ahora son los punteros a datos. Ahora, pasamos a ver los punteros a funciones.

Ahora sí, ¿qué son los punteros a funciones?

Un puntero a función es una variable que almacena la dirección de memoria de una función. Al igual que un puntero a dato es una variable que almacena la dirección de memoria de un dato que está en memoria.

Así de sencillo. Ahora, no sólo tenemos acceso a las funciones a través de su nombre, como las invocamos normalmente, sino que también podemos acceder a una función a través de un puntero a ella. De manera análoga, podemos acceder a un dato a través de la variable que lo contiene o a través de un puntero que contiene la dirección de memoria de dicha variable.

¿Para qué los quiero?

Aunque parezcan bichos feos y complicados, los punteros a funciones en realidad son muy útiles cuando buscás que tu código sea lo más genérico y modularizado posible.

Pensémoslo así. Tengo un algoritmo. La lógica general, principal, es siempre la misma, pero una parte varía, a veces es A y a veces es B.

```
void funcionA(){                    void funcionB(){

        <código común>                  <código común>

        A                               B

        <código común>                  <código común>
}                                       }
```

Vemos que hay un montón de código repetido, así que nuestra primera idea es hacer lo que venimos aprendiendo de Algo1 y ahora en Algo2, que es *abstraer lo que cambia y parametrizarlo*. Es decir, lo que vemos que es diferente, intentar recibirlo por parámetro.

El problema es que acá no tenemos una simple variable que difiere, A y B son líneas y líneas de código. Entonces, ¿cómo abstraigo lo que es diferente y hago que mi función sea genérica?

La respuesta es con punteros a función. La idea es abstraer A y B en dos funciones distintas y recibir esa función por parámetro, así

```
void funcion(void ( *funcion_extra)() ) {
    <código común>
    funcion_extra();
    <código común>
}
```

La función extra pasada por parámetro puede ser A, B o cualquiera que tenga la misma firma (exceptuando el nombre) de A y B. Pero ya nos vamos a preocupar por la sintaxis en un ratito; por ahora, comprendamos el concepto.

Con el uso de punteros a función, logro aislar lo que cambia. Tengo una función genérica, que recibe por parámetro lo que antes me hubiese dividido la función en dos funciones distintas. De esta manera, tengo una función que puede hacer lo mismo que la funcionA y la funcionB, pero sin duplicar código.

Recordame la sintaxis...

Veamos lo necesario para utilizar punteros a función. Si se desea expandir en este tema, la cátedra recomienda el libro "Understanding and Using C Pointers", de Richard Reese.

Puntero a función

Empecemos viendo la sintaxis de un puntero a función.

Cómo declarar un puntero a función

Si la función es

```
char obtener_letra (int numero, bool chequeo);
```

Un puntero a esta función es

```
char (*puntero_obtener_letra)(int, bool);
puntero_obtener_letra = obtener_letra;       //asignación
```

O, también

```
char (*puntero_obtener_letra)(int, bool) = obtener_letra;
//inicialización
```

Cómo recibir un puntero a función por parámetro

A su vez, esa función se recibiría por parámetro de esta manera

```
void mostrar_letra (int repeticiones, char (*puntero_obtener_letra)(int,
bool) );
```

Cómo invocar una función si tengo el puntero de la misma

Finalmente, para invocar una función cuando tengo el puntero a la misma, ejecuto el siguiente código

```
char mi_letra = puntero_obtener_letra(180, true);
```

Entonces, para uno tener un puntero a función, debemos conocer qué tipo de dato devuelve, y cuáles parámetros recibe. El nombre de la función, la cantidad de líneas, los llamados a otras funciones que ella haga, nada de esto nos importa.

Vector de punteros a función

Pasemos a ver cómo podemos tener un vector de punteros a función (sí, leíste bien, ahora tenemos vectores).

Si la firma de mis funciones son

```
int* suma(int* primer_numero, int* segundo_numero);
int* resta(int* primer_numero, int* segundo_numero);
```

Cómo declarar un vector de punteros a función

Un vector de funciones que las contenga se declara de esta manera

```
int* (funciones[2])(int*, int*) = {suma, resta};
```

Cómo invocar una función de un vector de punteros a función

A su vez, una función de un vector de punteros a función se invoca de cualquiera de estas dos maneras

Primer manera

```
tipo_de_dato mi_variable_resultado = vector_de_funciones[posicion]
(variable1, variable2);
```

Aplicado al ejemplo, suponiendo que valor1 y valor2 ya fueron declaradas correctamente, tenemos

```
int* suma_total = funciones[i](valor1, valor2);
```

Segunda manera

```
tipo_de_dato (*funcion_actual)(tipo_de_variable1, tipo_de_variable2) =
vector_de_funciones[posicion];
tipo_de_dato nombre_variable = funcion_actual(variable1, variable2);
```

Aplicado al ejemplo, suponiendo que valor1 y valor2 ya fueron declaradas correctamente, tenemos

```
int* (*funcion_actual)(int*, int*) = funciones[i];
int* suma_total = funcion_actual(valor1, valor2);
```

Otra manera de definir punteros a función

En esta sección, vamos a introducir una nueva sintaxis de punteros a función. Decidimos presentarla ya que, a pesar de que al principio puede resultar confusa, nos permite utilizar los punteros de una manera más sencilla y clara.

Pero no confundir, la forma mostrada previamente de definir un puntero a función es tan válida como la que explicaremos a continuación.

Tengo la función a almacenar, que es `void caminar(int* x, int* y);`.

Pero en vez de definirla de esa manera, la definimos así `typedef void (*moverse)(int*, int*);`.

Entonces, por ejemplo, quiero almacenarla en el struct de personaje

```
typedef struct personaje{
    size_t edad;
    char* nombre;
    int x;
    int y;
    //la función
} personaje_t;
```

Lo cual se hace de esta manera

```
typedef struct personaje{
    size_t edad;
    char* nombre;
    int x;
    int y;
    moverse caminar;
} personaje_t;
```

La función `caminar` ahora es "de tipo" `moverse`. A su vez, `moverse` es de tipo puntero a una función que recibe dos `int*` y devuelve `void`.

De esta manera, si mi personaje mañana en vez de moverse caminando, se mueve en patineta, esa función (que recibiría los mismos parámetros y también devolvería `void`), podría ser fácilmente reemplazada. Lo cual es una gran ventaja.

Otro beneficio es a la hora de escribir la función a recibir. En vez de tener que escribir `void (*caminar)(int*, int*)` ahora puedo escribir `moverse` directamente. Me ahorro escribir todo el tipo cada vez.

Aclaración de sintaxis

Habrás notado la falta de `&` para asignar punteros. Eso es porque la sintaxis de punteros a funciones es un poco diferente a la sintaxis de punteros a datos.

Cuando hablamos de punteros a funciones, no necesitamos utilizar el operador `&` para acceder a la dirección de memoria de la función. Se puede utilizar, el compilador lo permite, pero no es necesario como

en la sintaxis de punteros a datos. De esta manera, tenemos que este código

```
char (*puntero_obtener_letra)(int, bool) = obtener_letra;
```

Es equivalente a este código

```
char (*puntero_obtener_letra)(int, bool) = &obtener_letra;
```

Por otro lado, no existen los punteros doble a funciones, o triples. Entonces, este código

```
puntero_obtener_letra = obtener_letra
```

Es equivalente a este código

```
*puntero_obtener_letra = obtener_letra
```

Que, a su vez, es equivalente a este código

```
*******puntero_obtener_letra = obtener_letra
```

Ejemplos salvadores

Dejamos tres ejemplos de cómo se pueden aprovechar los punteros a funciones y sus diferentes aplicaciones.

Bubble sort genérico

Seguramente ya implementaste algún método de ordenamiento. Recibías un vector de enteros, el tope del mismo, y tal vez hasta un booleano que especificase cómo ordenar tu vector. Con esa función lista, ya podés ordenar cualquier vector de enteros.

Eso sí, ese método no sirve para ordenar floats, o size_t, o pokemones... ¿Debería implementar un sort específico para cada tipo de dato?

Si lo hiciese, yo tendría que implementar un bubble_sort, por ejemplo, que ordenase enteros. Otro bubble_sort que ordenase floats, otro que ordenase pokemones, etc. ¡Cuánta repetición de código! Pero notamos que la lógica del bubble_sort es siempre la misma, aunque necesitemos varios tipos de bubble_sort dependiendo del tipo de dato que quiero ordenar.

En Algo1 como en Algo2 venimos aprendiendo que tenemos que aislar lo que se modifica o cambia y, de ser posible, encapsularlo en una función. Ya sabemos que lo que es diferente entre el bubble_sort que ordena

enteros y el bubble_sort que ordena pokemones es el tipo de dato que ordena. ¿Podemos abstraer eso en otra función? De esta manera, tendríamos un bubble_sort genérico, ¿no?

La respuesta es sí, y se logra con, adivinaste, punteros a funciones. Si logramos aislar todo lo relacionado al tipo de dato, obtenemos un bubble_sort genérico.

Pensemos cómo podríamos obtenerlo. Necesito desligarme del tipo de dato que poseo en el vector. Sabemos de la clase de punteros de Algo2 que existe el llamado void*, que es el puntero comodín de los punteros a datos. Es decir, no sé a qué apunta mi puntero, puede apuntar a cualquier cosa. Si ahora tengo un vector de ese tipo de punteros, logro un vector de elementos genérico. Por otro lado, necesito saber cuántos elementos tiene mi vector, así que debería recibir el tope.

Finalmente, si no conozco los elementos que se almacenan en el vector, no sé cómo compararlos para ordenarlos. Y acá es donde entran nuestros nuevos amigos, los punteros a funciones. Yo podría recibir un puntero a una función que compara los elementos del vector. Esta función sería programada por el usuario de la función bubble_sort, que conoce qué elementos hay en el vector y cómo compararlos.

Con toda esta información, llegamos a que la firma de nuestra función bubble_sort debería ser ésta

```
void bubble_sort(void** vector, size_t cant_elementos, int (*comparador)
(void*, void*);
```

Y podríamos programar un bubble_sort genérico, capaz de ordenar cualquier vector de punteros.

Invasores

Estamos programando una versión del juego Space Invaders. Llegó el momento de programar los diferentes tipos de invasores. Tenemos tres tipos de invasores, y cada uno ataca de una manera particular.

A primera vista, podríamos pensar en tener el struct de invasor que posee sus coordenadas en el tablero y un carácter que nos dice qué tipo de ataque realiza. Cuando necesitamos que cada invasor ataque, verificamos el carácter del ataque, y dependiendo de su valor, llamamos a la función correspondiente. Hasta podríamos pensarlo con un switch.

El problema que tenemos con esta implementación es que no es versátil. Si mañana se crean 20 tipos de invasores nuevos, tenemos que modificar ese switch. El código se extiende demasiado; es difícil de seguir y de leer; es fácil confundirse y generar bugs, ya que tenemos que tocar código que funcionaba y modificarlo; en general, es una mala idea programar un switch tan extenso.

Este problema se puede arreglar, de nuevo, con punteros a funciones. Si pensamos en que cada invasor tiene su estilo de ataque, podemos modificar nuestra estructura para que el invasor ahora posea sus coordenadas y un puntero a su función de ataque.

```
typedef void (*funcion_de_ataque)(void*);

typedef struct invasor {
    int x;
    int y;
```

```
    funcion_de_ataque atacar;
} invasor_t;
```

De esta manera, cuando creamos un nuevo tipo de invasor, simplemente le agregamos el puntero a la función correspondiente. En este código, la función de ataque recibe un void* y no devuelve nada. Si se recibe al invasor por puntero, cuando llega la hora de atacar, es tan simple como esta línea

```
invasor->atacar(extra);
```

Ahora, cada invasor conoce su forma de atacar. Nosotros como programadores podemos abstraernos del comportamiento específico y ahorrarnos un switch gigante.

Peleas de pokemones

Último ejemplo. Estamos programando un torneo de pokemones, donde se inscriben enternadores y hacen competir a sus pokemones con diferentes funciones de pelea. Tenemos varias funciones que hacen pelear a dos pokemones, éstas son algunas:

```
pokemont_t   el_mas_fuerte(pokemon_t poke1, pokemon_t poke2);
pokemont_t   el_mas_inteligente(pokemon_t poke1, pokemon_t poke2);
pokemont_t   el_mas_rapido(pokemon_t poke1, pokemon_t poke2);
pokemont_t   el_mas_lindo(pokemon_t poke1, pokemon_t poke2);
pokemont_t   el_mas_carismatico(pokemon_t poke1, pokemon_t poke2);
```

Como podrás notar, tienen una estructura muy similar. Todas reciben dos pokemones y devuelven un pokemón. Cierto, por dentro son distintas, cada una tiene su lógica implementada. Pero desde afuera, salvando el nombre, son bastante parecidas.

Conocemos la firma de todas estas funciones, y también conocemos su implementación (las programamos nosotros, después de todo). Entonces, siempre que querramos invocar alguna, lo hacemos directamente con su nombre. Podemos invocar cualquier función (que hayamos codeado en nuestro código) directamente. Por lo tanto, en el torneo invocamos el tipo de pelea que queremos usar en ese momento.

Y es más, si agregamos la firma de todas esas funciones en mi .h, el usuario que quiera usar nuestro programa también puede invocar esas funciones sin problema. Eso sí, el usuario solamente puede usar las que nosotros programamos. Si quiere más tipos de pelea entre los pokemones, tiene que llamarnos a nosotros para que las agreguemos al .c y al .h... mmmm...

¿Hay alguna forma de desligarnos de la situación? ¿Hay alguna forma de que el usuario pueda programar sus propias peleas y que los pokemones puedan pelear con ellas, sin tener que involucrarnos nosotros?

Qué conveniente que hagamos esa pregunta en un apunte de punteros a función, ¡porque justamente podemos solucionar este problema con ellos!

Resulta que para uno tener un puntero a función, debemos conocer qué tipo de dato devuelve, y cuáles parámetros recibe. Y como mencionamos antes, todas las funciones reciben dos pokemones y devuelven

un pokemon, así que ya empezamos bien.

Lo increíble de punteros a función es que cuando recibimos ese puntero, en realidad no sabemos a qué función apunta. Sabemos que apunta a una función que recibe (en nuestro caso) dos pokemones y devuelve otro. Entonces, siempre y cuando el usuario programe funciones de pelea que reciban dos pokemones y devuelva otro, las podemos recibir en una función e invocarlas.

Es decir, algo similar a esto

```
pokemon_t pelea_entre_pokemones(pokemon_t poke1, pokemon_t poke2,
pokemon_t (*funcion)(pokemon_t, pokemon_t)) {
    if (!poke1 || !poke2) return NULL; //no podemos hacer competir a dos
pokemones cuando al menos uno de ellos en NULL

    if (!funcion) return NULL; //no puedo invocar una función nula

    pokemont_t poke_ganador = funcion(poke1, poke2);
    <…>
}
```

De esta manera, recibo la función de pelea por parámetro y la invoco con los pokemones. Recibo el pokemon ganador y efectúo con éste cualquier operación necesaria. Esta función que se recibe por parámetro puede ser una de las que yo programé y agregué a mi .c, o podría ser una programada por el usuario que yo no conozco, pero que el usuario necesita. Y si mañana se necesitan más tipos de pelea, no se tiene que hacer más que programar esas funciones y mandar un puntero a las mismas; no necesitamos modificar el código original en lo absoluto.