Guía para el manejo de punteros en C

Qué es un puntero

Un puntero es un tipo de variable. Almacena la dirección de memoria de otra variable, es decir, el lugar donde está esa variable en la memoria.

Al declarar una variable, la computadora reserva un espacio en la memoria del tamaño suficiente para que la variable almacene datos en ese espacio.

Pero en la memoria hay muchos espacios reservados por otros programas y por el sistema operativo, por eso al correr el programa, el sistema operativo se encarga de buscar un espacio desocupado para que la variable lo ocupe, así que podemos decir que en realidad nuestra variable es ese espacio.

De aquí en adelante hay que entender variable = espacio reservado en la memoria.

Por ejemplo, al declarar una variable

```
int p;
```

Estamos diciendo "reservar un espacio en la memoria en el que quepa un número entero y que ese espacio se llame p".

Ese espacio podría estar en cualquier lugar de la memoria ya que prácticamente es escogido por el sistema operativo al azar (se podría decir).

Para saber en qué lugar de la memoria está la variable se usan los punteros.

El lugar de la memoria donde está la variable se llama dirección y se expresa en forma numérica, la dirección de una variable es un número.

Cuando un puntero tiene almacenada la dirección de una variable decimos que el puntero apunta a esa variable.

En una variable hay por tanto tres elementos independientes pero relacionados entre si:

- La variable en sí (el espacio reservado en memoria).
- El valor de la variable (lo que hay en ese espacio).
- La dirección de la variable (en qué lugar de la memoria se encuentra el espacio que reservamos).

Es importante notar que aún cuando no se haya almacenado nada en una variable, está ocupando espacio en la memoria, pues ese espacio ya se reservó para almacenar algo y ningún programa lo puede utilizar.

¿Qué necesita un puntero para funcionar?

Un puntero _siempre_ debe apuntar a una variable.

Un error muy común es utilizar punteros que no están apuntando a nada. En otras palabras, siempre los punteros deben contener alguna dirección.

Cuando un puntero no contiene ninguna dirección, es decir, no apunta a nada, es un puntero muerto.

Por eso, antes de utilizar los punteros _verifica_ que estén apuntando a algo.

Aparentemente los punteros no sirven para nada. Por ahora voy a explicar cómo se utilizan y más adelante para qué sirven.

Cómo se declaran los punteros

Para declarar un puntero se hace de la siguiente forma:

```
tipo_de_la_variable_a_la_que_queremos_apuntar *nombre_puntero;
```

Ejemplo:

```
float *puntero_a_una_variable_tipo_float;
```

El asterisco sólo es para _declarar_ el puntero, no es parte del nombre del puntero, así que después de declarar el puntero lo usaremos _sin_ *. Voy a explicar esto con más detalle.

Esta declaración la podemos dividir en dos partes:

```
float *
```

y

```
puntero_a_una_variable_tipo_float;
```

Con float * estamos diciendo que lo que estamos declarando es un puntero a un float (es decir, float * es un tipo de variable) y puntero_a_una_variable_tipo_float es el nombre del puntero.

También es válido

```
float * puntero_a_una_variable_float;
```

Pero personalmente, me gusta más declararlos de la otra forma, sin espacio entre el * y el nombre del puntero.

Esto quiere decir que ahora tenemos nuevos tipos de variables.

Antes teníamos:

```
int
float
char
struct
```

Ahora tenemos además los tipos:

```
int * (puntero a int)
float * (puntero a float)
char * (puntero a char)
struct algo * (puntero a struct algo)
```

Que quede claro que el * pertenece al tipo del puntero, no al nombre del puntero.

Una variable de tipo int * _sólo_ puede almacenar la dirección de una variable int, lo mismo para los demás.

Para almacenar la dirección de una variable en un puntero, se usa el operador & antes del nombre de la variable de la cuál queremos obtener la dirección. & significa "dirección de", cuando veas un &, leelo como "dirección de".

Ejemplo:

```
float variable_flotante;
float *puntero_a_una_variable_tipo_float; //aún no apunta a nada

puntero_a_una_variable_tipo_float  =  &variable_flotante;  //Ahora apunta a variable_flotante (contiene su dirección).
```

Ejemplo inválido:

```
float variable_flotante = 7;
float *puntero_a_una_variable_tipo_float;

puntero_a_una_variable_tipo_float = variable_flotante;
```

Eso está mal porque el puntero ahora contiene un 7, y 7 no es la dirección de la variable flotante sino su valor.

Nosotros queremos la dirección de esa variable, no su valor y por eso se debe usar el &.

De aquí se deduce que los punteros al ser variables, ocupan espacio en la memoria y también tienen las tres carácterísticas de las variables:

- Nombre
- Valor (la dirección de otra variable)
- Su propia Dirección. El puntero al ser una variable también tiene su propia dirección porque está en algún lugar de la memoria.

No confundir el valor de un puntero con su dirección.

El valor de un puntero es la dirección de otra variable. La dirección del puntero es la propia dirección del puntero, o sea, en qué lugar de la memoria está el puntero.

Como los punteros también tienen su propia dirección, podemos hacer punteros que apunten a otros punteros (punteros que contengan la dirección de otros punteros).

Un puntero que contiene la dirección de otro puntero es un puntero doble y también los hay triples (que contienen la dirección de un puntero doble), cuádruples, etc. Más adelante hay un ejemplo.

Operadores que se usan con los punteros

Los operadores son la fuente de muchísimas confusiones.

Hasta ahora hemos visto dos operadores que se usan con los punteros: * y &.

El operador *

Mencioné que * se utiliza para declarar punteros. Pero también tiene otro uso, * sirve para _desreferenciar_ un puntero.

Desreferenciar un puntero significa obtener el valor de la variable a la que apunta el puntero.

La desreferenciación se hace asi:

```
*nombre_del_puntero;
```

Y es ahí donde empieza la confusión.

Eso se parece mucho a la declaración de un puntero y además * se usa también para multiplicar.

Pero no hay nada qué temer, simplemente cuando _a la izquierda_ del * aparezca float, int, char, etc. es una _declaración_.

Si _a la izquierda_ del * NO aparece ningún tipo de dato y _a la derecha_ de él aparece el nombre de un puntero, es una desreferenciación.

Ejemplo:

```
int a;
int otra_variable = 2;
int *mi_puntero; //es una declaración de una variable de tipo int * que se
                    llama mi_puntero.

mi_puntero = &otra_variable;

a = 4 + *mi_puntero; //es una desreferenciación
```

El valor de a será 6, porque es 4 + el valor de la variable a la que apunta mi_puntero, como mi_puntero apunta a otra_variable y otra_variable vale 2, al desreferenciar al puntero usando *mi_puntero, obtenemos el 2.

Otro Ejemplo:

```
char a = 'x';
char *ap = &a; //declaramos una variable de tipo char *
               //que se llama ap y luego hacemos que
               //apunte a la variable a (le asginamos
               //la dirección de a) Este es un caso
               //especial, le estamos asignando una dirección
               //a un puntero usando * porque al mismo tiempo
               //lo estamos declarando, normalmente sería ap = &a

printf("%c\n", *ap); //aquí estamos desreferenciando ap
printf("%p\n", ap);
```

El primer printf imprimirá la letra x, que es el _valor de la variable a la que apunta_ ap, como ap apunta a y el valor de a es x, obtenemos x.

El segundo imprimirá un número extraño, ese número es la dirección de a, o sea, el valor de ap.

Nota: Al usar printf para imprimir una dirección usamos el formato %p.

El operador * cuando se usa como desreferenciación se puede leer como "dame el valor que esté en la variable a la que apunta este puntero".

Otro Ejemplo mas:

```
float a = 9.5;
float b;

float *ap_a = &a;
float *ap_b = &b;
float *ap;

ap = ap_b //es válido porque el valor de ap_b es una dirección(dirección de b)

ap = *ap_a //inválido porque le estamos asignando a un puntero
           //el valor de la variable a la que apunta ap_a, como
           //ap_a apunta a a y a vale 9.5, le estamos asignando a
           //ap 9.5 y 9.5 no es una dirección de memoria.
```

En realidad, al desreferenciar un puntero lo que sucede es que estamos accesando al espacio de memoria de esa variable.
Esto quiere decir que si por ejemplo ap_x = &x, *ap_x es sinónimo de x, por lo que podemos asignarle un valor a x a través de ap_x de la siguiente forma:

```
int x;
int *ap_x = &x;

*ap_x = 13; //Es válido porque al desreferenciar accesamos al espacio
            // e memoria de la variable a la que apunta el puntero.
            //*ap_x = 13 es lo mismo que x = 13 porque ap_x apunta a x.

printf("%d\n", x);
printf("%d\n", *ap_x);
```

Imprimirá el valor 13 en ambos printf.

Así cuando la desreferenciación de un puntero aparezca a la izquierda de un =, es para asignarle un valor a la variable a la que apunta, si no está a la izquierda de un =, es para obtener el valor de la variable a la que apunta.

En resumen, * se usa ademas de multiplicar para:

- Declarar punteros (como parte del tipo de dato).
- Desreferenciar punteros.

El operador &

Con este operador no hay mucho problema, simplemente con él obtenemos la dirección de una variable.

Ejemplo:

```
float a;
float *ap; //declaración de una variable tipo float * llamada ap

ap = &a;

printf("%p\n", ap);
printf("%p\n", &a);
```

Ambos printf imprimen lo mismo.

Otro ejemplo:

```
int x;
int *ap = &x;
int *ap2;

ap2 = &ap //Inválido
```

¿Por qué es inválido? porque aunque los punteros almacenan direcciones de otras variables, no pueden almacenar direcciones de otros punteros, para eso se ocupan punteros dobles.

```
int x;
int *ap = &x;
int **ap2; //puntero doble (de tipo int **)

ap2 = &ap; //OK, porque ap2 es un puntero doble
ap2 = &x;  //Mal, porque x no es un puntero y ap2 sólo puede almacenar
           //direcciones de punteros que sean int *
```

El operador ->

Existe un caso especial de los punteros: los punteros a estructuras.

Cuando declaramos un puntero a una estructura, usamos -> para accesar a los miembros de la estructura a la que apunta.
El -> es parecido al . sólo que se usa con punteros.

A la _izquierda_ de un -> siempre _debe_ estar el nombre de un _puntero a una estructura_ y de lado derecho un miembro de esa estructura.

Así que ahora tenemos un nuevo tipo de dato:

```
struct algo *
```

Ejemplo:

```
struct empleado
{
 char nombre[20];
 float sueldo;
};

struct empleado datos; //declaramos una estructura de tipo empleado
                       //llamada datos.


struct empleado *emp = &datos; //declaramos un puntero de tipo
                               //struct empleado * que se llama emp
                               //y lo inicializamos con la dirección
                               //de la variable datos.
```

Ahora si quisieramos imprimir el nombre y el sueldo usando al puntero emp, simplemente hacemos:

```
printf("Nombre: %s\n", emp->nombre);
printf("Sueldo: %f\n", emp->sueldo);
```

Así que -> también es un operador de desreferenciación porque nos da el valor de un miembro de la variable struct a la que apunta un puntero de tipo struct.

Ejemplo inválido:

```
struct datos
{
 char nombre[20];
 float sueldo;
};

struct empleado
{
 struct datos d;
};
```

```
struct empleado emple;
struct empleado *emp = &emple;

.
.
.

printf("Nombre: %s\n", emp->d->nombre); //Mal
```

Eso está mal porque dijimos que a la izquierda de -> debe estar el nombre de un puntero de tipo estructura.

emp->d está bien, porque emp es un struct empleado *, pero d-> está mal porque d NO es un puntero.

Lo correcto es:

```
printf("Nombre: %s\n", emp->d.nombre);
```

En cambio estaría bien si tuviéramos:

```
struct datos
{
 char nombre[20];
 float sueldo;
};

struct empleado
{
 struct datos dat;
 struct datos *d; //ahora sí, d es un puntero
};

emple->d = &(emple->dat); //y hacemos que apunte a dat.
struct empleado emple;
struct empleado *emp = &emple;

printf("Nombre: %s\n", emple->d->nombre); //ahora sí estaría bien
```

Otro error común es este:

```
struct empleado
{
 float sueldo;
 float *s;
```

```
};


struct empleado datos;
struct empleado *emp = &datos;

datos.sueldo = 1000;
datos.s = &datos.sueldo;

printf("Sueldo: %f\n", *emp->s);
```

Lo que se está tratando de hacer es obtener el sueldo a través del puntero s.
Pero el printf fallará miserablemente porque el * se está aplicando sólo a emp, no a emp->s.

Lo correcto es:

```
printf("Sueldo: %f\n", *(emp->s));
```

y así se está aplicando el * a emp->s.

Los arreglos son punteros

Otro caso especial de los punteros son los arreglos. Los arreglos son punteros. En serio.

si declaras un:

```
char car[10];
```

Estas diciendo, "reservar espacio para 10 variables de tipo char consecutivas, estando el primer elemento en la dirección a la que apunta car".

Por lo tanto, es posible declarar un

```
char *ap_char;
```

y hacer que apunte al primer elemento del arreglo de esta forma:

```
ap_char = car;
```

El _nombre del arreglo_ es entonces un puntero que apunta al primer elemento de ese arreglo. Esto último es importante.

Es importante destacar que ésto _únicamente_ sirve para el primer elemento.

Para hacer punteros que apunten a los demás elementos se debe utilizar la notación de índices.

Por ejemplo, si quiero un puntero que apunte al tercer elemento:

```
char *ap_3 = &car[2]; //recuerda que el primero es el 0
```

De esto se desprende que para apuntar al primer elemento también es válido:

```
ap_char = &car[0];
```

¡Atención! car[0] no es lo mismo que car. car[0] es la variable a la que apunta car. &car[0] sí es lo mismo que car, porque estamos obteniendo la dirección de la variable car[0];

Además de la notación de índices también podemos usar desreferenciación para accesar a los elementos individuales del arreglo.

De esta forma

car[0] es lo mismo que *car;

y para accesar a los demás elementos se le suma n a car, siendo n el elemento al que queremos accesar.

Para accesar a car[3] por medio de desreferenciación se hace de la siguiente forma:

```
*(car + 3)
```

Lo que significa "avanzar tres direcciones de memoria contando a partir de car y desreferenciar esa dirección (obtener lo que hay ahí o asignarle un valor si está a la izquierda de un =)".

El objetivo de esta sección es decir que si tienes una función

```
func(char *cadena)
```

y le quieres pasar un puntero al arreglo de chars car, únicamente hay que llamar a func de esta forma:

```
func(car);
```

ya que car es un puntero al primer elemento del arreglo, y sabiendo donde está el primer elemento automáticamente podemos accesar a los demás.

¿Para qué sirven los punteros?

Aparentemente el conocer la dirección de una variable no nos es de ninguna utilidad.

Los punteros se usan principalmente para dos cosas:

- Paso de parámetros entre funciones.
- Memoria dinámica.

¿Porqué complicarse pasándole un puntero a una función si es mas fácil pasarle una variable normal?.

Al pasar un puntero a una función ahorramos memoria y tiempo de ejecución.

Cada vez que pasamos una variable a una función esa función hace su propia copia de la variable. Es importante recordar siempre esto.

Lo voy a repetir:

Cada vez que pasamos una variable a una función, esa función crea su propia copia de la variable.

Si llamamos a la función bar pasándole la variable p

```
int p;

bar(p);
```

El resultado es que tenemos _dos_ variables p en la memoria, la original y la que la función bar crea dentro de sí misma.

En cambio si le pasamos a bar la dirección de la variable p sólo tenemos _una_ variable p en la memoria, la original, y bar no necesita crear su propia variable p, bar puede usar la variable original porque ya le dijimos dónde está la p original al pasarle la dirección. Así que además de la p original, en la memoria también está un puntero que contiene la dirección de p.

Supongamos, exagerando un poco para que se aprecie mejor, que creamos un arreglo llamado x de 1000000 elementos int, que ese arreglo se pasa a una función recursiva y que esa función se llama a sí misma 1000 veces.

Va a llegar un momento en que ese arreglo va estar copiado 1000 veces en la memoria porque cada vez que la función se llama a sí misma hace su propio arreglo x de 1000000 elementos.

Como cada elemento int mide 4 bytes tenemos que el arreglo x mide en total 4000000 bytes. Al estar en la memoria 1000 veces tenemos 1000 arreglos de 4000000 bytes cada uno en la memoria.

4000000 x 1000 = 400,000,000 bytes ocupados.

400,000,000 son aproximandamente ¡400 Mb de memoria desperdiciados!.

En cambio ahora supongamos que en vez de pasarle a la función el arreglo x le pasamos la dirección del arreglo. Esa dirección obviamente se va a almacenar en un puntero, el cual al ser una variable, ocupa también espacio en memoria.

Cada vez que la función sea llamada ésta creará su propio puntero que contendrá la dirección del arreglo.

Así tenemos en la memoria un arreglo x de 4000000 bytes y 1000 punteros (uno por cada función) que contienen la dirección del arreglo.

Un puntero ocupa 4 bytes de memoria.

Entonces tenemos 1000 x 4 = 4000 bytes de memoria son los que ocupan los 1000 punteros.

4000 bytes que ocupan los punteros + 4000000 del arreglo x = 4,004,000 bytes en total.

4004000 bytes son aproximadamente 4.4 Mb

¡4.4 Mb contra 400 Mb es una gran diferencia!.

Resumiendo, por lo general se consume menos memoria pasándole sólo la dirección de la variable a una función que al pasarle _toda_ la variable.

También aquí hay otro efecto indeseado y es que si hacemos:

```
int p;

bar(p);
```

y dentro de la función bar modificamos a la variable p, en realidad bar modifica _su_ copia de p y la p original queda inalterada.

Entonces si queremos que una función modifique la variable que le estamos pasando le tenemos que pasar la dirección de esa variable.

así:

```
int p;

bar(&p);
```

dentro de bar si modificamos a p, estaremos modificando a la p original.

bar para que reciba una dirección debería estar declarada así:

```
void bar(int *puntero);
```

Y dentro de bar para asignarle algo a p:

```
*puntero = algo;
```

Memoria dinámica

El otro uso que se le da a los punteros es la memoria dinámica.

Al usar memoria dinámica nosotros podemos indicarle al sistema operativo que reserve memoria para una variable y cuando no ocupemos mas esa variable podemos dejar el espacio libre.

Esto es útil en entornos multitarea donde están corriendo varios programas al mismo tiempo, todos ellos compartiendo los mismos recursos, así nosotros al dejar libre el espacio que ocupaba la variable, otro programa que necesite espacio lo puede utilizar.

La función más común en el manejo de memoria dinámica es malloc.

La definición de malloc es:

```
void *malloc(size_t size)
```

Lo que quiere decir que devuelve o regresa un puntero de tipo void y necesita que se le pase como argumento un tamaño.

La tarea de malloc es reservar espacio para una variable de tamaño size y devuelve un puntero a ella.

Hay algo muy interesante en relación a malloc y es que las variables que reserva son anónimas, ¡son variables sin nombre!.

Entonces si la variable que reservamos no tiene nombre ¿cómo accesamos a ella? pues por medio del puntero que malloc nos devuelve y que contiene la dirección de esa variable.

Ejemplo clásico de estructuras de datos:

```
struct nodo
{
 int numero;
 struct nodo *siguiente;
};
```

```
struct nodo *ap_nodo;

ap_nodo = (struct nodo *)malloc(sizeof(struct nodo));
```

En este ejemplo malloc reserva espacio en memoria del tamaño de una struct nodo. Ese espacio, esa variable, no tiene nombre, sólo le decimos que la reserve pero no le decimos como queremos que se llame. Pero no hay problema, porque nos regresa un puntero a esa variable.

Por eso siempre hay que almacenar el valor de retorno de malloc porque si sólo utilizarmos:

```
(struct nodo *)malloc(sizeof(struct nodo));
```

malloc reserva espacio para una estructuda nodo, pero como no almacenamos el valor de retorno, ese valor ya se perdió para siempre y nunca podremos accesar a la variable porque no tiene nombre y tampoco sabemos su dirección.

¿Por qué malloc tiene esa sintaxis tan extraña? ¿Qué significan los paréntesis antes de malloc?

en el ejemplo

```
ap_nodo = (struct nodo *)malloc(sizeof(struct nodo));
```

(struct nodo *) es lo que se conoce como type casting.

Un type casting consiste en convertir temporalmente un tipo de variable en otro tipo.

Un ejemplo típico es

```
int a = 1;
int b = 3;
float res;

res = (float)a / (float)b; //usando type casting

printf("%f\n", res);

res = a / b;  //sin type casting

printf("%f\n", res);
```

El primer printf imprimirá 0.3333 y el segundo 0.0000.
Esto es porque al usar el operador / el resultado es del tipo de las variables que se están usando con ese operador.

Como a y b son variables de tipo entero el resultado es un entero, el programa nos devuelve sólo la parte entera de ese resultado que es 0.

```
void *malloc(size_t size)
```

La definición nos dice que malloc devuelve un void *, pero anteriormente se mencionó que un un puntero sólo puede guardar direcciones de variables de un sólo tipo, si es un int * sólo puede guardar direcciones de variables int y si es un char * sólo de variables char.

Nuestro ejemplo dice:

```
ap_nodo = (struct nodo *)malloc(sizeof(struct nodo));
```

Con el type casting (struct nodo *) estamos convirtiendo el puntero void * que malloc devuelve en un puntero struct nodo * porque ap_nodo es un struct nodo * y por lo tanto no podemos almacenar en él un void *.

De lado derecho tenemos sizeof(struct nodo), sizeof devuelve el tamaño de una variable. En este caso se está usando para saber el tamaño de una struct nodo ya que malloc lo requiere para saber cuánto espacio reservar en memoria.

Esa línea traducida a palabras sería algo así:

"Reservar en memoria un espacio donde quepa una struct nodo, el puntero que me devuelva convertirlo en un puntero de tipo struct nodo * y almacenarlo en ap_nodo".

Webgrafía y Licencia

◆Textos tomados, corregidos y modificados de diferentes páginas de Internet, tutoriales y documentos.

◆Autor:
Matías E. García
Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar