Punteros II.

Programación I – Laboratorio I.

Tecnicatura Superior en Programación.

UTN-FRA

Autor: *Lic. Mauricio Dávila. Basado en el apunte "El lenguaje de programación C" de la Universidad de Valencia*

Revisores: *Ing. Ernesto Gigliotti*

Versión : 1

Índice de contenido

1 Punteros

Los punteros son una de las poderosas herramientas que ofrece el lenguaje C a los programadores, sin embargo, son también una de las más peligrosas, el uso de punteros sin inicializar, etc., es una fuente frecuente de errores en los programas de C, y además, suele producir fallas muy difíciles de localizar y depurar.

1.1 Definición

Un puntero es una variable que contiene una dirección de memoria, normalmente esa dirección es una posición de memoria de otra variable, por lo cual se suele decir que el puntero "apunta" a la otra variable.

1.2 Declaración de un puntero

La sintaxis de la declaración de una variable puntero es:

```
tipo* nombre;
```

El tipo base de la declaración sirve para conocer el tipo de datos al que pertenece la variable a la cual apunta la variable de tipo puntero. Esto es fundamental para poder leer el valor que almacena la zona de memoria apuntada por la variable puntero y para poder realizar ciertas operaciones aritméticas sobre los mismos.

Algunos ejemplos de declaración de variables puntero son:

```
int* a;

char* p;

float* f;
```

1.3 Operadores

Existen dos operadores especiales de los punteros, el operador '&' encargado de retornar la dirección de memoria de una variable y el operador '*' encargado de retornar el valor de la variable apuntada por un puntero.

Si declaramos:

```
int* a;

int b;
```

Y hacemos:

```
a = &b;
```

La variable puntero '**a**' contendrá la dirección de memoria de la variable '**b**'.

Veámoslo otro ejemplo:

```
int* a;
int b,c;
```

Y hacemos:

```
b=15; // Asignamos el valor 15 a la variable 'b'
a=&b; // Obtenemos la posición de memoria de 'b' con el  operador '&'
c=*a; // Copiamos el contenido apuntado por el puntero 'a'
```

Entonces la variable '**c**' contendrá el valor 15, pues '***a**' devuelve el contenido (o valor) de la dirección a la que "apunta" la variable puntero, y con anterioridad hemos hecho que '**a**' contenga la dirección de memoria de la variable '**b**' usando para ello el operador '**&**'.

1.4 Asignación de punteros

Es posible asignar el valor de una variable de tipo puntero a otra variable de tipo puntero.

Por ejemplo:

```
int* a;
int* b;
int  c;
a=&c; // Obtenemos la posición de memoria de 'c' con el  operador '&'
b=a;  // Asignamos el valor del puntero 'a' al puntero 'b'
```

Entonces b contiene el valor de a, y por ello, b también "apunta" a la variable c.

1.5 Comparación de punteros

Sobre las variables de tipo puntero es posible realizar operaciones de comparación, veamos un ejemplo:

```
int* punteroA;
int* punteroB;
int  auxiliarC , auxiliarD;

punteroA = &auxiliarC; // Obtenemos la posición de memoria de 'auxiliarC'
punteroB = &auxiliarD; // Obtenemos la posición de memoria de 'auxiliarD'

if (punteroA<punteroB)
      printf("El punteroA  apunta a una dirección más baja que punteroB");
else if (punteroA>punteroB)
      printf("El punteroB  apunta a una dirección más baja que punteroA");
```

1.6 Aritmética de punteros

Sobre las variables de tipo puntero es posible utilizar los operadores +, -, ++ y --. Estos operadores incrementan o decrementan la posición de memoria a la que "apunta" la variable puntero. El incremento o decremento se realiza de acuerdo al tipo base de la variable de tipo puntero, de ahí la importancia del tipo del que se declara la variable puntero.

Veamos esto con la siguiente tabla:

	Posición Actual	a++	a--	a=a+2	a=a-3
`char *a`	0xA080	0xA081	0xA07F	0xA082	0xA07D
`int *a`	0xB080	0xB084	0xB07C	0xB088	0xB06E
`float *a`	0xC080	0xC084	0xA07C	0xA088	0xA06E

Como podemos observar en la tabla cada operación sobre el puntero se rige por su tipo, por lo tanto, si tenemos:

```
tipo *a;
```

Y hacemos:

```
a = a + num;
```

La posición a la que apunta a se incrementa en:

```
nueva dirección que contiene "a" = dirección que contiene "a" + (num * sizeof(tipo))
```

Para la resta se decrementa de igual forma en:

```
nueva dirección que contiene "a" = dirección que contiene "a" - (num * sizeof(tipo))
```

Los operadores ++ y -- son equivalentes a realizar num=1, y con ello quedan obviamente explicados.

1.7 Vectores y punteros

Existe una estrecha relación entre los punteros y los vectores. Consideremos el siguiente fragmento de código:

```
char cadena[80];
char *p;
p=&cadena[0]; // equivalente a: p=cadena
```

Este fragmento de código pone en la variable puntero '**p**' la dirección del primer elemento del array '**cadena**'.

Entonces, es posible acceder al valor de la quinta posición del array mediante: cadena[4] y *(p+4) (recuérdese que los índices de los arrays empiezan en 0).

Esta estrecha relación entre los arrays y los punteros queda más evidente si se tiene en cuenta que el **nombre del array sin índice es la dirección de comienzo del array**, y si además, se tiene en cuenta que un puntero puede indexarse como un array unidimensional, por lo cual, en el ejemplo anterior, podríamos referenciar ese elemento como p[4].

Es posible obtener la dirección de un elemento cualquiera del array de la siguiente forma:

```
int arrayInt[80];
int* p1;
int* p2;
p1 = &arrayInt[4];
p2 = &arrayInt;
```

Entonces, el puntero '**p1**' contiene la dirección del quinto elemento del arrayInt y el puntero '**p2**' contiene la dirección del primer elemento del arrayInt.

Hasta ahora hemos declarado variables puntero aisladas. Es posible, como con cualquier otro tipo de datos, definir un array de variables puntero. La declaración para un array de punteros int de tamaño 10 es:

```
int* a[10];
```

Para asignar una dirección de una variable entera, llamada '**var**', al tercer elemento del array de punteros, se escribe:

```
x[2] = &var;
```

Y para obtener el valor de var:

```
*x[2];
```

Dado además, que un puntero es también una variable, es posible definir un puntero a un puntero. Supongamos que tenemos lo siguiente:

```
int a;
int *punteroInt;
int **punteroPuntero;
punteroInt = &a; // Obtenemos la posición de memoria
punteroPuntero = &puntero; // Obtenemos la posición de memoria
```

Y entonces, ¿De qué formas podemos ahora acceder al valor de la variable '**a**'?

a (forma habitual)

*punteroInt (a través del puntero)

**punteroPuntero (a través del puntero a puntero)

Esto es debido a que '**punteroPuntero**' contiene la dirección de '**punteroInt**', que contiene la dirección de '**a**'.

Este concepto de puntero a puntero podría extenderse a puntero a puntero a puntero, etc., además, existe el concepto de puntero a una función, al cual nos referiremos más adelante.

2 Tipos de Punteros

Un puntero es una variable como cualquier otra destinada a guardar una posición de memoria, esta posición de memoria generalmente corresponde a otra variable.

Lo que se pretende decir es que si a un puntero se le carga cualquier valor , éste será interpretado como si fuera una dirección de memoria aunque el valor cargado no corresponda a una dirección de memoria.

2.1 Punteros a cadenas de caracteres

Un puntero a cadena de caracter (string) no es más que un puntero a caracter. Por lo tanto la forma de definir el puntero a cadena de caracteres es:

```
char  *p;
```

Por ejemplo la siguiente línea declara un puntero a caracter y un vector para guardar la cadena de caracteres.

```
char  cadena[LONGITUD] ;
char  *puntero;
```

Así se reserva espacio para una cadena de LONGITUD caracteres. En C, el nombre de las listas monodimensionales equivale a la dirección del primer elemento de las mismas. Una vez que se dispone de espacio reservado para la cadena, se puede construir un puntero de la cadena empleando las siguientes sentencias:

```
char  cadena[LONGITUD] ;
char  *puntero;
puntero = cadena; // es equivalente a escribir  puntero = &cadena[0]
```

Esto dará lugar a que en '**puntero**' se almacene la dirección de '**cadena**', se asigna el valor de la dirección de un carácer (el primero de la cadena) a una variable de tipo puntero de char.

2.1.1 Recorrido de una cadena mediante aritmética de punteros e índices.

El mecanismo de la aritmética de punteros en C tiene su expresión más sencilla en el caso de utilizar, precisamente, punteros de caracteres.

Consideremos las siguientes declaraciones:

```
tipo  lista[NUMERO_DE_ELEMENTOS];
tipo* puntero;
puntero = lista;
```

Se debe recordar que en general, cuando se suma 1 a un puntero, el valor numérico de la dirección no se incrementa en 1, sino en:

```
sizeof(tipo)
```

Entonces, al sumar "i" a un puntero se tiene que la dirección obtenida al calcular "puntero+i" tiene el valor:

```
direccion_actual_del_puntero + (i * sizeof(tipo))
```

Dando como resultado un nuevo puntero, que señala una posición de memoria que correspondería a un elemento de una lista situado "i" posiciones más allá del elemento señalado por "puntero". Se detalla a continuación un ejemplo en el cual aplicando aritmética de punteros recorreremos una cadena de caracteres :

```
void mostrarCadena(char* punteroCadena)
{
	while(*punteroCadena != '\0')
	{
		printf("%c",*punteroCadena);
		punteroCadena++;
	}
}
```

Como se puede observar, la condicion del bucle indica que mientras que sea el contenido de la posicion apuntada por 'punteroCadena' distinto a `'\0'` se continuará iterando, en cada iteración se imprime el contenido de la posición apuntada por 'punteroCadena' y se incrementa el valor del puntero mediante 'punteroCadena++' .

2.2 Punteros a estructura

Como una estructura es una variable, se la puede apuntar con un puntero. De acuerdo a la definición de puntero, deberá guardar la dirección de memoria de la estructura , si pensamos que una estructura puede estar formada por un grupo de variables distintas, el puntero tendrá la dirección de comienzo del grupo de variables. Veamos un ejemplo:

```
struct alumno {
	char nombre[20];
	int nota;
};
struct alumno auxiliarAlumno;
struct* punteroAlumno;
punteroAlumno = &auxiliarAlumno;
```

Si se piensa en un vector de estructuras y se almacena en un puntero (del tipo de esa estructura) la posición de memoria que corresponde al comienzo de dicho vector, al momento de recorrerlo, utilizando el puntero, en cada incremento pasará a apuntar a la próxima estructura almacenada en el array. Lo recién enunciado cumple con la aritmética de punteros explicada anteriormente. Veamos un ejemplo:

```
#define CANTIDAD 5
struct alumno
{
	char nombre[20];
	int nota;
};

void main(void)
{
	int i;
	struct alumno arrayAlumnos[CANTIDAD];
	struct alumno *punteroArrayAlumnos;
	punteroArrayAlumnos = arrayAlumnos;
	for(i=0; i<CANTIDAD; i++)
	{
		punteroArrayAlumnos++;
		//...
	}
}
```

2.2.1 Acceso a los campos – Operador flecha

Cuando se usan los punteros a estructura la forma de acceder a un campo es usando el operador FLECHA (->) el cuál se forma escribiendo un guión y a continuación un mayor. Veamos un ejemplo:

```
struct alumno {
	char nombre[20];
	int nota;
};

void main(void)
{
	struct alumno auxiliarAlumno;
	struct *punteroAlumno;
	punteroAlumno = &auxiliarAlumno;
	strcpy(punteroAlumno->nombre, "Ernesto");
	punteroAlumno->nota = 10;
}
```

Como se puede ver al momento de querer acceder al campo "nota" de la estructura "auxiliarAlumno" mediante el puntero "punteroAlumno" , se remplaza el operador punto (auxiliarAlumno.nota) por el operador flecha (punteroAlumno->nota).

2.2.2 Vector de punteros

Un vector de punteros es simplemente un vector que dentro de cada uno de sus elementos existe un puntero. La forma de definir un vector de punteros es:

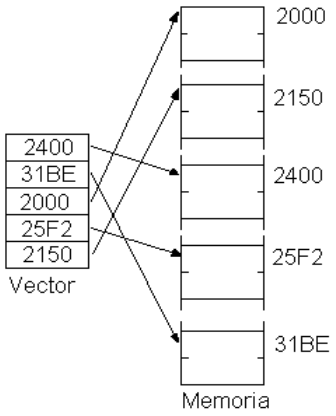
```
	tipo* nombre[MAX];
```

En el caso que se requiera definir un vector de punteros int.

```
	int* vec [MAX];
```

El vector tiene posiciones de memoria (punteros) de algunas variables, por ejemplo el elemento vec[i] contiene la dirección de memoria de una variable y por lo tanto el contenido de *vec[i] será un entero.

Al igual que se inicializaba un puntero, se debe inicializar el vector de punteros para que cada posición del vector apunte a algún lugar determinado. Veamos el siguiente dibujo que puede ilustrar mejor la situación.



A la izquierda del dibujo tenemos el vector "vec" que es un vector de punteros , por lo tanto cada elemento del vector es una dirección de memoria (puntero). Inicialmente como pasa en cualquier variable tiene un valor cualquiera al momento de arrancar el programa, por lo cuál los punteros están apuntando a cualquier posición.

Normalmente se asocia un vector de punteros a otro de estructuras, es decir que el vector de punteros tiene las direcciones de cada uno de los elementos del vector de estructuras. Veamos una utilidad del vector de punteros.

Supongamos que se debe realizar un programa de agenda que cargue los datos de 100 alumnos y luego tenga que ordenar los datos según su nota. En el proceso de ordenamiento lo que se hace es copiar el elemento '**i**' a un '**auxiliar**' , luego copiar el elemento '**j**' al elemento '**i**' y finalmente el auxiliar al elemento '**j**' , o sea está realizando 3 copias de estructuras lo cual implica un movimiento importante de datos.

```
for(i=0; i < CANTIDAD -1 ; i++)
{
	for(j=i+1; i < CANTIDAD; j++)
	{
		if(arrayAlumnos[i].nota > arrayAlumnos[j].nota)
		{
			auxiliarAlumno =  arrayAlumnos[i]
			arrayAlumnos[i] =  arrayAlumnos[j];
			arrayAlumnos[j] =  auxiliarAlumno;
		}
	}
}
```

Ahora si usted tiene un vector de punteros, el movimiento de datos se realiza en el vector de punteros, con lo cuál la copia se reduce notablemente (la cantidad de bytes a copiar se reduce a 2 bytes que es el tamaño de un puntero) y por otra parte no se modifica el vector de estructuras. Cuando se quiere mostrar los datos ordenados simplemente se recorre el vector de estructuras. Veamos un ejemplo.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define CANTIDAD 3
struct alumno
{
	char nombre[50];
	int nota;
};

void ordenar(struct alumno* arrayPunterosAlumnos[3], int longitudArray);

int main()
{
	int i,auxiliarNota;
	char auxiliarNombre[50];
	struct alumno arrayAlumnos[CANTIDAD];
	struct alumno* arrayPunterosAlumnos[CANTIDAD];

	for(i=0; i<CANTIDAD; i++)
	{
		// Copiamos la posición de memoria de cada elemento
		arrayPunterosAlumnos[i] = &arrayAlumnos[i];
			printf("\nIngrese el nombre: ");
		scanf("%s",auxiliarNombre);
		strcpy(arrayAlumnos[i].nombre, auxiliarNombre); //FALTARIA VALIDAR
		printf("\nIngrese la Nota: ");
		scanf("%i",&auxiliarNota);
		arrayAlumnos[i].nota = auxiliarNota; //FALTA VALIDAR

	}
```

```
        printf("/n-MOSTRAMOS EL SIN ORDENAR ARRAY-");
        for(i=0; i<CANTIDAD; i++)
        {
                printf("/n%s", arrayPunterosAlumnos[i]->nombre);
                printf("-%i",arrayPunterosAlumnos[i]->nota);
        }
        ordenar(arrayPunterosAlumnos , CANTIDAD);
        printf("/n-MOSTRAMOS EL ARRAY ORDENADO-");
        for(i=0; i<CANTIDAD; i++)
        {
                printf("/n%s", arrayPunterosAlumnos[i]->nombre);
                printf("-%i",arrayPunterosAlumnos[i]->nota);
        }
        return 0;
}
void ordenar(struct alumno* arrayPunterosAlumnos[], int longitudArray)
{
    struct alumno* punteroAuxiliarAlumno;
    int i,j;

    for(i = 0; i < longitudArray -1 ; i++)
    {
        for(j = i+1; j < longitudArray ; j++)
        {
            if(arrayPunterosAlumnos[i]->nota > arrayPunterosAlumnos[j]->nota)
            {
                punteroAuxiliarAlumno =  arrayPunterosAlumnos[i];
                arrayPunterosAlumnos[i] =  arrayPunterosAlumnos[j];
                arrayPunterosAlumnos[j] =  punteroAuxiliarAlumno;
            }
        }
    }
}
```

Observe que cuando se realiza el ordenamiento se ordena el vector de punteros, con lo cuál el vector de estructuras queda intacto. Además de no modificarse el vector de estructuras, en el proceso de ordenar al modificar el vector de punteros siempre se mueve la misma cantidad de datos, es decir que no interesa la cantidad de campos que posea la estructura. Entonces si una estructura contiene 20 campos y otra solo 2, en las asignaciones dentro del algoritmo que ordena solo se copiarán datos del tamaño del puntero.