PUNTEROS EN C

El tipo de datos más característico del C son los punteros. Un puntero contiene un valor que es la dirección en memoria de un dato de cierto tipo. Cuando se declara una variable, se reserva un espacio en la memoria para almacenar el valor de la variable. Ese espacio en memoria tiene una dirección. Un puntero es una dirección dentro de la memoria, o sea, un apuntador a donde se encuentra una variable. Los punteros son una parte fundamental de C. Si usted no puede usar los punteros apropiadamente entonces esta perdiendo la potencia y la flexibilidad que C ofrece básicamente. El secreto para C esta en el uso de punteros. C usa los punteros en forma extensiva. ¿Por qué?

- Es la única forma de expresar algunos cálculos.
- Se genera código compacto y eficiente.
- Es una herramienta muy poderosa.

Los punteros se emplean en C para muchas cosas, por ejemplo recorrer vectores, manipular estructuras creadas dinámicamente, pasar parámetros por referencia a funciones, etc.

Definición de un puntero

Un puntero es una variable que contiene la dirección en memoria de otra variable. Se pueden tener punteros a cualquier tipo de variable. El operador unario o monádico **&** devuelve la dirección de memoria de una variable.

El operador de **indirección** o **desreferencia** * (asterisco) devuelve el “contenido o valor de una variable apuntada por un puntero”.

Para **declarar** un puntero para a variable entera hacer:

```
int *puntero;
```

Se debe asociar a cada puntero un tipo particular. Por ejemplo, no se puede asignar la dirección de un short int a un long int.

Varios punteros pueden apuntar a la misma variable:

```
int* puntero1;
int* puntero2;
int var;
puntero1 = &var;
puntero2 = &var;
*puntero1 = 50;          /* mismo efecto que var=50  */
var = *puntero2 + 13;    /* var=50+13 */
```

Para tener una mejor idea, considerar el siguiente código:

```
main()
{
    int x = 1, y = 2;
    int *ap;

    ap = &x;
    y = *ap;
    x = ap;
    *ap = 3;
}
```

Cuando se compile el código se mostrará el siguiente mensaje:

warning: assignment makes integer from pointer without a cast.

Con el objetivo de entender el comportamiento del código supongamos que la variable x esta en la localidad de la memoria 100, y en 200 y ap en 1000.

Nota: un puntero es una variable, por lo tanto, sus valores necesitan ser guardados en algún lado.

int x = 1, y = 2; int *ap;

ap = &x;

Dirección	100	200	1000
Contenido	x←1	y←2	ap←100

Las variables x e y son declaradas e inicializadas con 1 y 2 respectivamente, ap es declarado como un puntero a entero y se le asigna la dirección de x (&x). Por lo que ap se carga con el valor 100.

y = *ap;

Dirección	100	200	1000
Contenido	x←1	y←1	ap←100

Después y obtiene el contenido de la variable apuntada por ap. En el ejemplo ap apunta a la localidad de memoria 100 (la localidad de x). Por lo tanto, y obtiene el valor de x (el cual es 1).

x = ap;

Dirección	100	200	1000
Contenido	x←100	y←1	ap←100

Como se ha visto C no es muy estricto en la asignación de valores de diferente tipo (aqui un puntero se asigna a un entero). Así que es perfectamente legal (aunque el compilador genera un aviso de cuidado) asignar el valor actual de ap a la variable x. El valor de ap en ese momento es 100.

*ap = 3;

Dirección	100	200	1000
Contenido	x←3	y←1	ap←100

Finalmente se asigna un valor al contenido de la variable apuntada por un puntero (*ap).

Importante: Cuando un puntero es declarado, apunta a cualquier dirección. Se debe inicializar el puntero antes de usarlo. Por lo que:

```
main()
{
   int *ap;
   *ap = 100;
}
```

puede generar un error en tiempo de ejecución o presentar un comportamiento errático.

El uso correcto será:

```
main()
{
   int *ap;
   int x;

   ap = &x;
   *ap = 100;
}
```

NOTA: Un puntero a cualquier tipo de variables es una dirección en memoria (la cual es una dirección entera), pero un puntero NO es un entero. La razón por la cual se asocia un puntero a un tipo de dato, es porque se debe conocer en cuantos bytes esta guardado el dato. De tal forma que, cuando se incrementa un puntero, se incrementa el puntero por un "bloque" de memoria, en donde el bloque esta en función del tamaño del dato. Por lo tanto para un puntero a un char, se agrega un byte a la dirección y para un puntero a entero se agregan 4 bytes. De esta forma si a un puntero a entero se le suman 2, el puntero entonces se mueve dos posiciones int que equivalen a 8 bytes.

Operaciones

Como un **puntero desreferenciado** *p es una variable, acepta todas las operaciones que corresponden a su tipo. Ej: int *p

Las siguientes son expresiones equivalentes:

*p = *p + 1

*p += 1

++ *p

(*p)++ aqui los paréntesis son necesarios porque los operadores unarios se asocian de derecha a izquierda.

Las operaciones validas con **punteros** (sin desreferenciar) son:

- asignación de punteros de igual tipo,
- suma y resta de un puntero y un entero,
- resta o comparación de dos punteros a miembros del mismo arreglo,
- asignación o comparación con cero.

Ejemplo de aritmética de punteros:

```
int vector [100];      /* un vector de enteros */
int *ptr;              /* un puntero a enteros */
...
ptr = &vector[0];      /* ptr apunta al principio del vector */
*ptr = 33;             /* igual que vector[0] = 33 */
*(ptr+1) = 44;         /* igual que vector[1] = 44 */
*(ptr+2) = 90;         /* igual que vector[2] = 90 */
```

La expresión que se suma al puntero ha de ser entera y no tiene por qué ser constante. Obsérvese que ptr es lo mismo que ptr+0.

La expresión sumada NO es el número de bytes que se suman a la dirección, es el número de elementos del tipo al que apunta el puntero:

```
/* Supongamos que un "char" ocupa 1 byte */
/* y que un "double" ocupa 8 bytes */
char* ptrchar;
double* ptrdouble;
...
*(ptrchar+3) = 33;        /* la dirección es ptrchar + 3 bytes */
*(ptrdouble+3) = 33.0;    /* la dirección es ptrdouble + 24 bytes, ya que cada double ocupa 8 bytes */
```

El compilador "sabe" cómo calcular la dirección según el tipo.

A un puntero se le puede aplicar cualquier clase de operación de suma o resta (incluyendo los incrementos y decrementos).

```
/* Rellenar de unos los elementos del 10 al 20 */
int* ptr;                 /* el puntero */
int vector [100];         /* el vector */
int i;                    /* variable contadora */

ptr = &vector[0];         /* ptr apunta al origen del vector */
ptr+=10;                  /* ptr apunta a vector[10] */
for ( i=0; i<=10; i++ )
{
    *ptr = 1;             /*asigna 1 a la posición de memoria apuntada por "ptr" */
    ptr++;                /*ptr pasa al siguiente elemento */
}
```

Precauciones con los punteros

a) Punteros no inicializados

Si se altera el valor al que apunta un puntero no inicializado, se estará modificando cualquier posición de la memoria.

```
main()
{   int* puntero;
    *puntero = 1200;      /* Se sobreescribe una zona  cualquiera de la memoria */
}
```

b) Confusión de tipos

Un puntero a un tipo determinado puede apuntar a una variable de cualquier otro tipo. Aunque el compilador lo puede advertir, no es un error.

(Afortunadamente, esto no ocurre en C++).

```
main()
{
        int p;
        double numero;
        int* puntero;

        p = &numero;        /* incorrecto,
  pero el compilador no aborta */
        *p = 33;            /* Un desastre */
}
```

c) Punteros a variables locales fuera de ámbito

Si un puntero apunta a una variable local, cuando la variable desaparezca el puntero apuntará a una zona de memoria que se estará usando para otros fines. Si se desreferencia el puntero, los resultados son imprevisibles y a menudo catastróficos.

```
main()
{       int* puntero;
        while (...)
        {
          int local;
          puntero = &local;
 /* 'puntero' apunta a una variable local */
          ...
          *puntero = 33;            /* correcto */
        }
        ...
/* ahora 'puntero' apunta a una zona de memoria
  inválida */
        puntero = 50; /* catástrofe */
}
```

Punteros y Funciones

Cuando C pasa argumentos a funciones, los pasa por valor, es decir, si el parámetro es modificado dentro de la función, una vez que termina la función el valor pasado de la variable permanece inalterado. Hay muchos casos en los que se quiere alterar el argumento pasado a la función y recibir el nuevo valor una vez que la función ha terminado. Para hacer lo anterior se debe usar una llamada **por referencia**, **en C se puede simular pasando un puntero al argumento**. Con esto se provoca que la computadora pase la dirección del argumento a la función.

NOTA: en C++ las llamadas por referencia no necesitan punteros y se realizan anteponiendo & al identificador del parámetro.

Para entender mejor lo anterior consideremos la función swap() que intercambia el valor de dos argumentos enteros:

```
void swap(int *px, int *py);
main()
{
   int x, y;
   x = 10;
   y = 20;
   printf("x=%d\ty=%d\n",x,y);
   swap(&x, &y);
   printf("x=%d\ty=%d\n",x,y);
}
void swap(int *px, int *py)
{
   int temp;
   temp = *px;   /* guarda el valor de la direccion x */
   *px = *py;    /* pone y en x */
   *py = temp;   /* pone x en y */
}
```

Punteros y arreglos

Existe una relación estrecha entre los punteros y los arreglos. En C, un nombre de un arreglo es un puntero constante a la dirección de comienzo del arreglo. En esencia, el **nombre de un arreglo es un puntero al arreglo**. Considerar lo siguiente:

```
int a[10];
int x;
```

```
int *ap;
ap = &a[0];    /* ap apunta a la dirección de a[0] */

x = *ap;       / A x se le asigna el contenido de ap (a[0] en este caso) */

*(ap + 1) = 100;  / Se asigna al segundo elemento de 'a' el valor 100 usando ap*/
```

Como se puede observar en el ejemplo la sentencia a[t] es idéntica a *(ap+t). Se debe tener cuidado ya que **C no hace una revisión de los límites del arreglo**, por lo que se puede ir fácilmente más allá del arreglo en memoria y sobrescribir otras cosas.

C sin embargo es mucho más sutil en su relación entre arreglos y punteros.

Por ejemplo se puede teclear solamente:

ap = a; en vez de ap = &a[0];

y también

***(a + i) en vez de a[i],**

esto es,

&a[i] es equivalente con a+i.

Sin embargo los punteros y los arreglos son diferentes:

- **Un puntero es una variable.** Se puede hacer ap = a y ap++.
- **El nombre de un arreglo NO ES una variable (es una constante).** Hacer a = ap y a++ ES ILEGAL.

Esta parte es muy importante, asegúrese haberla entendido. Con lo comentado se puede entender como los arreglos son pasados a las funciones. Cuando un arreglo es pasado a una función lo que en realidad se le esta pasando es la dirección de su elemento inicial en memoria. Por eso decimos que **los arreglos son siempre parámetros pasados por referencia** sin que tengamos que hacer nada para especificarlo. (ver → Pasaje por Referencia constante).

Por lo tanto:

strlen(s) es equivalente a strlen(&s[0])

Esta es la razón por la cual se declara la función como:

int strlen(char s[]); y una declaración equivalente es int strlen(char *s); ya que:

char s[] es igual que char *s.

Un string se representa como sabemos como una array de char. El identificador de un array es un puntero (al primer elemento del array), por lo que un puntero a un char puede verse como un array de char.

O sea: un puntero a un char es equivalente a un array de char (puede ser un string).

La función strlen() es una función de la biblioteca estándar que regresa la longitud de una cadena. Se muestra a continuación la versión de esta función que podría escribirse:

```
int strlen(char *s)
{
```

```
    char *p = s;
    while ( *p != '\0' )
    p++; return p - s;
}
```

Se muestra enseguida una función para copiar una cadena en otra. Al igual que en el ejercicio anterior existe en la biblioteca estándar una función que hace lo mismo.

```
void strcpy(char *s, char *t)
{
while ( (*s++ = *t++) != '\0' );
}
```

Nota: Se emplea el caracter nulo con la sentencia while para encontrar el fin de la cadena.

Otro ejemplo:

```
/*      Rellena de ceros los "n_elem"  primeros elementos de "vector"  */
void rellena_de_ceros ( int n_elem, int* vector )
{
  int i;
  for ( i=0; i<n_elem; i++ )
        *(vector++) = 0;          /* operador de post-incremento */
}

main()
{
  int ejemplo [300];
  int otro_vector [200];

  /* pasa la dirección del vector "ejemplo" */
  rellena_de_ceros ( 300, ejemplo );

  /* rellena los elems. del 150 al 199 */
  rellena_de_ceros ( 50, otro_vector+150 );
}
```

Punteros y strings

Sutil diferencia:

- char *p = “cadena”

  aquí **p** es un puntero al primer carácter de “cadena” que es un **array de char constante**. Por lo tanto se puede modificar p para que apunte a otro string, pero no se puede modificar el string ya que es constante.

- Char a[ ] = “cadena”

  Aquí **a** es un **puntero constante a un array de char**. Por lo tanto se puede modificar el contenido del array, pero no puede modificarse el puntero **a** para que apunte a otra dirección, porque es un puntero constante.

Ejemplo de uso de la notación puntero+desplazamiento en lugar de array[indice]:

```
#include <stdio.h>
#define N 5          // constante N
void main()
{
        // declaracion de variables
        int arre1[N], arre2[N];
        int i, j, max, aux;

        // instrucciones del programa
        for (i=0; i < N; i=i+1)
        {
                printf(" Ingresa el valor de arreglo[%d]: ", i);
                scanf("%d", arre1+i);  // arre1++ equivale a &arre1[i]
        };
      printf("\n Arreglo ingresado:");
        for (i=0; i < N; i=i+1)
        {
                        printf("\n arreglo[%d]= %d", i, *(arre1+i));     // *(arre1+1) es arre1[i]
        };
        // busqueda del mayor
        max= *arre1;
        for (i=1; i < N; i++)
        {
                if (*(arre1+i) > max)
```

```
                    max = *(arre1+i);
        };
        printf("\n \n el mayor es %d", max);
        // creamos el arreglo2 invertido
        for (i=0, j=N-1; i < N; i++, j--)
        {
                *(arre2+j)=*(arre1+i);
        };
        printf("\n \n Arreglo invertido:");
        for (i=0; i < N; i=i+1)
        {
                printf("\n arreglo[%d]= %d", i, *(arre2+i));     // imprimimos el arreglo2
        };
        // arreglo1 invertido
        for (i=0; i < N/2; i++)
        {
                aux=*(arre1+i);
                *(arre1+i)=*(arre1+N-i-1);
                *(arre1+N-i-1)=aux;
        };
        printf("\n \n Arreglo invertido:");
        for (i=0; i < N; i=i+1)
        {
                    printf("\n arreglo[%d]= %d", i, *(arre1+i));     // imprimimos el arreglo1
        };

}
```

Punteros a estructuras

También se pueden usar punteros con estructuras. Vamos a ver como funcionan. Primero hay que definir la estructura de igual forma que hacíamos antes. La diferencia está en que al declara la variable de tipo estructura debemos ponerle el operador '*' para indicarle que es un puntero.

Es importante recordar que un puntero no debe apuntar a un lugar cualquiera, debemos darle una dirección válida donde apuntar. No podemos por ejemplo crear un puntero a estructura y meter los datos directamente mediante ese puntero, no sabemos dónde apunta el puntero y los datos se almacenarían en un lugar cualquiera.

Y para comprender cómo funcionan nada mejor que un ejemplo. Este programa utiliza un puntero para acceder a la información de la estructura:

```
#include <stdio.h>
struct estructura_amigo {
      char nombre[30];
      char apellido[40];
      char telefono[10];
      int edad;
      };
struct estructura_amigo amigo = {
      "Juanjo",
      "Lopez",
      "592-0483",
      30
      };
struct estructura_amigo *p_amigo;

int main()
    {
    p_amigo = &amigo;
    printf( "%s tiene ", p_amigo->apellido );
    printf( "%i años ", p_amigo->edad );
    printf( "y su teléfono es el %s.\n" , p_amigo->telefono );
    }
```

Con la definición del puntero p_amigo vemos que todo era igual que antes. p_amigo es un puntero a la estructura estructura_amigo. Dado que es un puntero tenemos que indicarle dónde debe apuntar, en este caso vamos a hacer que apunte a la variable amigo:

p_amigo = &amigo;

No debemos olvidar el operador & que significa 'dame la dirección donde está almacenado...'.

Ahora queremos acceder a cada campo de la estructura. Antes lo hacíamos usando el operador '.', pero, como muestra el ejemplo, si se trabaja con punteros se debe usar el operador '->'. Este operador viene a significar algo así como: "dame acceso al miembro ... del puntero ...".

Ya sólo nos queda saber cómo podemos utilizar los punteros para introducir datos en las estructuras. Lo vamos a ver con un ejemplo:

```
#include <stdio.h>
struct estructura_amigo {
      char nombre[30];
      char apellido[40];
      int edad;
      };
struct estructura_amigo amigo, *p_amigo;
int main()
    {
    p_amigo = &amigo;
    /* Introducimos los datos mediante punteros */
    printf("Nombre: ");fflush(stdout);
    gets(p_amigo->nombre);
    printf("Apellido: ");fflush(stdout);
    gets(p_amigo->apellido);
    printf("Edad: ");fflush(stdout);
    scanf( "%i", &p_amigo->edad );

     /* Mostramos los datos */
     printf( "El amigo %s ", p_amigo->nombre );
     printf( "%s tiene ", p_amigo->apellido );
     printf( "%i años.\n", p_amigo->edad );
     }
```

NOTA: p_amigo es un puntero que apunta a la estructura amigo. Sin embargo
p_amigo->edad es una variable de tipo int.
Por eso al usar el scanf tenemos que poner el &.

Punteros a arrays de estructuras

Por supuesto también podemos usar punteros con arrays de estructuras. La forma de trabajar es la misma, lo único que tenemos que hacer es asegurarnos que el puntero inicialmente apunte al primer elemento, luego saltar al siguiente hasta llegar al último.

```
#include <stdio.h>
#define ELEMENTOS    3
struct estructura_amigo {
      char nombre[30];
      char apellido[40];
      char telefono[10];
      int edad;
      };
struct estructura_amigo amigo[] =
      {
      "Juanjo", "Lopez", "504-4342", 30,
      "Marcos", "Gamindez", "405-4823", 42,
      "Ana", "Martinez", "533-5694", 20
      };
struct estructura_amigo *p_amigo;
int main()
    {
    int num_amigo;
    p_amigo = amigo;  /* apuntamos al primer elemento del array */
    /* Ahora imprimimos sus datos */
    for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
        printf( "El amigo %s ", p_amigo->nombre );
        printf( "%s tiene ", p_amigo->apellido );
        printf( "%i años ", p_amigo->edad );
        printf( "y su teléfono es el %s.\n" , p_amigo->telefono );
        /* y ahora saltamos al siguiente elemento */
```

```
        p_amigo++;
        }
    }
```

En vez de p_amigo = amigo; se podía usar la forma p_amigo = &amigo[0];, es decir que apunte al primer elemento (el elemento 0) del array. La primera forma creo que es más usada pero la segunda quizás indica más claramente al lector principiante lo que se pretende.

Ahora veamos el ejemplo anterior de cómo introducir datos en un array de estructuras mediante punteros:

```
#include <stdio.h>
#define ELEMENTOS       3
struct estructura_amigo {
      char nombre[30];
      char apellido[40];
      int edad;
      };

struct estructura_amigo amigo[ELEMENTOS], *p_amigo;
int main()
    {
    int num_amigo;
    /* apuntamos al primer elemento */
    p_amigo = amigo;
    /* Introducimos los datos mediante punteros */
    for ( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
       {
       printf("Datos amigo %i\n",num_amigo);
       printf("Nombre: ");fflush(stdout);
       gets(p_amigo->nombre);
       printf("Apellido: ");fflush(stdout);
       gets(p_amigo->apellido);
       printf("Edad: ");fflush(stdout);
       scanf( "%i", &p_amigo->edad );
       /* vaciamos el buffer de entrada */
       while(getchar()!='\n');
```

```
        /* saltamos al siguiente elemento */
        p_amigo++;
        }
    /* Ahora imprimimos sus datos */
    p_amigo = amigo;
    for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
        printf( "El amigo %s ", p_amigo->nombre );
        printf( "%s tiene ", p_amigo->apellido );
        printf( "%i años.\n", p_amigo->edad );
        p_amigo++;
        }
    }
```

Es importante no olvidar que al terminar el primer bucle for el puntero p_amigo apunta al último elemento del array de estructuras. Para mostrar los datos tenemos que hacer que vuelva a apuntar al primer elemento y por eso usamos de nuevo p_amigo=amigo; (en negrita).