*Capítulo Nº 7 : **Punteros en C***

Los punteros (o apuntadores) son una de las herramientas más poderosas de C, sin embargo en este caso, poder implica peligro. Es fácil cometer errores en el uso de punteros, y estos son los más difíciles de encontrar, una expresión típica usada en este caso por los programadores se refiere a la "perdida de un puntero", lo cual indica que ocurrió un problema en la asignación de algún puntero.

La ventaja de la utilización de punteros en C es que mejora enormemente la eficiencia de algunos procesos, además , permite la modificación de los parámetros pasados a las funciones (paso de parámetros por referencia) y son usados en los procesos de asignación dinámica de memoria.

Puntero es una representación simbólica de una dirección de memoria, es decir, contiene la dirección de un objeto o variable.

Operador &

Contiene la dirección o posición de memoria en la cual se ha almacenado una variable. El operador & es unuario, es decir, tiene un solo operando, y devuelve la dirección de memoria de dicho operando. Supongamos el siguiente ejemplo:

```
main()
{
   int auxiliar=5;
   printf("\nauxiliar=%d --->dirección=%p",auxiliar,&auxiliar);
}

por pantalla =>

      auxiliar=5 ---> dirección=289926.
```

En este ejemplo se utilizó el modificador de formato `%p` *que muestra la dirección según el formato del computador usado. Hasta ahora no había importado el lugar donde se almacenaban los datos.*

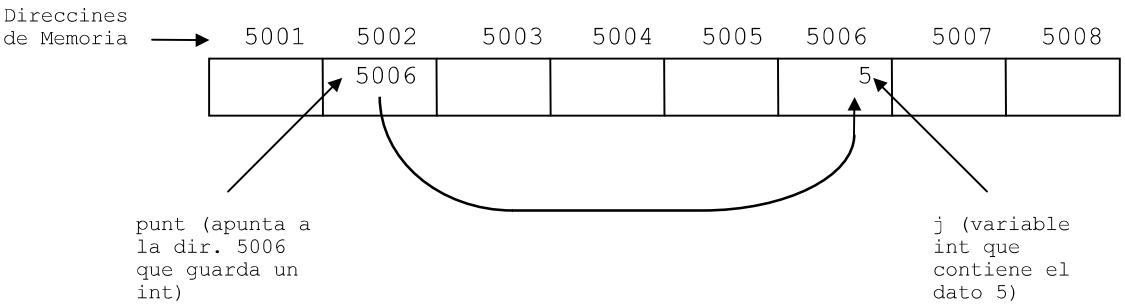
Los punteros se pueden declarar en los programas para después utilizarse y tomar las direcciones como valores. Por ejemplo:

```
int *punt;         /* un apuntador a int */
int j=5;
```

Se declaro un apuntador `punt` *a* `int`*, es decir, el contenido del puntero es una dirección que apunta a un objeto de tipo entero. Si se hiciese la siguiente asignación:*

```
punt=&j;
```

estaría correcta e implicaría que `punt` *apunta a la dirección de memoria de la variable* `j`*. Gráficamente sería:*



entonces, el valor de `punt` *y* `&j` *sería* `5006`.

Los punteros pueden ser inicializados, para ello existen varias formas:

```
punt=NULL;
punt=&j;
punt=(int *) 285395;     /* asignación de dirección */
                                 /* absoluta */
```

En la primera línea se le asigna un nulo, es como decir que apunta a ninguna parte. En la tercera línea se hace la asignación de una posición especifica de memoria.

Entonces, para declarar un puntero, primero se especifica el tipo de dato (básico o creado por el usuario), luego el nombre del puntero precedido por el caracter `*`.

Operador *

El operador `*` *es unuario y toma a su operando como una dirección de memoria, entonces el operador accesa al contenido de esa dirección. Por ejemplo, suponga las variables declaradas anteriormente y la asignación,*

```
int *punt;
int j=5;
punt=&j;
```

entonces, si imprimimos en pantalla:

```
printf("El contenido de punt es: %d",*punt);
```

imprimiría:

```
El contenido de punt es: 5
```

con esto se imprimió el contenido de la dirección a la que apunta `punt`. *Habría sido igual haber impreso la variable* `j`. *Si se declarara otra variable int, igualmente es valido hacer:*

```
int *punt;
int j=5,otro;
punt=&j;
:
:
otro=*punt;
```

que asignaría a `otro` *el dato* `5`*. Si se hace la siguiente asignación:*

```
*punt=200;
```

se modifica el contenido de la dirección a la que apunta `punt`*. Por lo tanto también se modificó la variable* `j`*, ya que la dirección de esta es a la que apunta* `punt`*.*

Hay ciertas restricciones en la asignación de punteros, por ejemplo:

```
int *p,*q,*r;      /* 3 punteros a entero*/
int array[20],var1;
register int j;

p=&j;
q=568200;
r=&array;
p=&(var1+5);
```

Todas las asignaciones anteriores están ***erróneas****. A un puntero no se le puede asignar la dirección de una variable register, de hecho, porque esta está almacenada en un registro de la CPU. A un puntero no se le puede asignar un valor cualquiera si no existe una conversión explícita de tipo. No puede recibir la dirección de un nombre de un arreglo dado que este último es un puntero al primer elemento del arreglo. La última asignación también es ilegal ya que se pretende asignar la posición de* `var1` *mas* `5` *posiciones de memoria.*

Las siguientes asignaciones están correctas:

```
int *p,*q;
int array[20],var1;

p=NULL;
q=&array[5];
p=q;
```

Primero a `p` *se le asigno el valor nulo, luego a* `q` *se le asigno la dirección del sexto elemento del arreglo y finalmente a* `p` *se le asigno la misma dirección que* `q`*, por lo tanto apuntan a la misma posición.*

Hasta ahora se ha ocupado implícitamente punteros en nuestros programas, ejemplo de ello es al ocupar scanf, donde pasamos como parámetro la dirección de la variable, también, en arreglos, como el caso de cuando es pasado a una función y sus valores resultan modificados.

Paso de parámetros por referencia

Sabemos que cuando se pasa como parámetro una variable a una función, esta no logra modificar el valor de la variable, eso es dado que fue pasada por valor. Para lograr que se modifique el contenido de una variable debe pasarse por referencia, para ello se pasa a la función la dirección de la variable y no el dato.

Ejercicio Nº 7.1: Supongamos que se desea hacer una función que haga el intercambio de datos entre dos variables (swap).

Solución: Primero hagamos el programa principal, de tal manera de saber como se hace la llamada por referencia. Este podría ser:

```
#include <stdio.h>
void swap();             /*funcion para el intercambio*/
main()
{
   int a=100,b=200;
   printf("a=%d   b=%d",a,b);
   swap(&a,&b);
   printf("a=%d   b=%d",a,b);
}
```

Nuestro programa resulta bastante simple, lo importante es notar que ahora los parámetros `a` *y* `b` *van precedidos de el operador* `&`*, es decir, no se pasan los datos* `100` *y* `200`*, sino, las direcciones de las variables.*

Ahora, como a la función se le pasaron direcciones de variables int, entonces en la cabecera de la función deben declararse parámetros como punteros a int. Esto sería:

```
void swap(int *x,int *y)
```

Dentro de nuestra función lo que tenemos que hacer es intercambiar los valores. Como los parámetros son punteros, entonces debemos trabajar con los contenidos de dichos punteros (o los contenidos de las direcciones a la que apuntan los punteros), para ello utilizamos el operador asterisco. La función completa es:

```
void swap(int *x,int *y)
{
   int aux;
   aux=*x;
   *x=*y;
   *y=aux;
}
```

Con esto logramos nuestro objetivo, la salida al programa es:

```
a=100   b=200
a=200   b=100
```

Ejercicio Nº 7.2: Cree un programa que lea dos números flotantes, luego una función que tenga dos parámetros y salga el mayor en el primer parámetro y el menor en el segundo.

Solución: El programa principal sería:

```
#include <stdio.h>
void maxymin(int *x,int *y);
main()
{
   int i,j;
   scanf("%d",&i);
   scanf("%d",&j);
```

```
   maxymin(&i,&j);
   printf("El Max es i=%d, El Min es j=%d",i,j);
   return 0;
}
```

La función no variaría mucho respecto al ejercicio anterior:

```
void maxymin(int *x,int *y)
{
   int aux;
   if (*x<*y)
   {
      aux=*x;
      *x=*y;
      *y=aux;
   }
   return;
}
```

En el `if` *se pregunto si "el contenido del puntero* `x` *es menor que el contenido del puntero* `y`*".*

En esta función podríamos utilizar la función anterior `swap` *para hacer el cambio. En ese caso, aparte de incluir la declaración de dicha función antes del* `main`*,* `maxymin` *queda:*

```
void maxymin(int *x,int *y)
{
   int aux;
   if (*x<*y)
      swap(&*x,&*y);
}
```

Hay que notar que al hacer `&*x` *hablamos de la dirección del contenido del puntero* `x`*. Esto se hace utilizando la misma lógica de pasar la dirección. Sin embargo, hablar de* `&*x` *es hablar de* `x`*, en otras palabras, si decimos que "el dato* `120` *(por ejemplo) esta almacenado en la dirección* `x`*" ,es lo mismo que "el dato* `*x` *esta almacenado en* `x` *o en* `&*x`*". Entonces la llamada a* `swap` *puede ser hecha como:*

```
   if (*x<*y)
      swap(x,y);
```

Lo que cumple con pasar las direcciones de las variables a las funciones, esto ya en este caso los parámetros de por si son direcciones (punteros).

Nota: Supongamos las siguientes sentencias:

```
   int a=10,*p;
   p=&a;
```

es posible imprimir:

```
   printf("%d",*&a);
   printf("%p",&*p);
   printf("%p",&p);
```

es decir, en la primera línea se imprime "el contenido de la dirección `&a` *" que es lo mismo que imprimir* `a`*, pero, no se puede imprimir* `&*a`*, esto provocaría un error. En la segunda línea es igual a imprimir* `p`*. En la tercera se imprime la dirección donde esta almacenado el puntero* `p`*.*

Aritmética de Punteros

Las operaciones matemáticas que se pueden usar con punteros es el decremento, incremento y resta entre dos punteros, es resto de las operaciones quedan prohibidas. La operatoria de punteros queda sujeta al tipo base de este, por ejemplo:

```
int *q,dato;
q=&dato;
q++;
```

En este caso si el puntero `q` *apunta a la dirección* `30000`*, el incremento apuntaría* `q` *a la dirección* `30002`*. Esto ya que un entero tiene 2 bytes de largo y el incremento hace que se apunte al siguiente entero. Lo mismo ocurriría con* `q--`*. Si a* `q` *se le incrementa en* `20` *haría que* `q` *apuntara al vigésimo entero que esté despues de la posición original del puntero. Suponga el siguiente ejemplo:*

```
int *p,arreglo[20],i;
:
:
p=&arreglo[0];
for(i=0;i<20;i++)
{
   printf("%d",*p);
   p++;
}
```

En este caso se imprime todo el arreglo `arreglo` *utilizando el puntero* `p`*. Esto es efectivo dado que al declarar el arreglo, sus elemento se ubican en posiciones contiguas de memoria. También sería correcto utilizar*

```
p=&arreglo[0];
printf("El decimo elemento es : %d",*(p+9));
```

para referirnos al décimo elemento del arreglo. Como hay que tener cierto cuidado con el manejo de punteros, veamos este ejemplo:

```
int *p,arreglo[20],i;
:
:
p=&arreglo[0];
for(i=0;i<20;i++)
{
   printf("%d",*p);
   p++;
}
printf("El decimo elemento es : %d",*(p+9));
```

Este es similar al anterior, sin embargo, no hace lo que supuestamente queremos, porque?, la razón es que el puntero fue incrementado 20 veces y despues del for ya no apunta al primer elemento del arreglo, por lo tanto puede imprimir cualquier valor (que sería aquel ubicado 9 variables enteras más allá).

También es posible hacer la resta entre dos punteros, esto con el objeto de saber cuantos datos del tipo base (de los punteros) se encuentran entre ambos.

Igualmente se puede hacer la comparación entre punteros:

```
if (p>q)
   printf("\np es mayor que q\n");
```

En este caso indicaría que `p` *esta ubicado en una posición de memoria superior a la de* `q`.

Algo más sobre punteros

Los punteros no solamente pueden apuntar a tipos conocidos, sino que también a tipos creados por el usuario, como es el caso de punteros a estructuras. Podemos mencionar además que un uso frecuente de estos son en estructuras de datos para trabajar con listas enlazadas, arboles, etc., sin embargo esto último no corresponde verlo en este libro.

Otro ejemplo de uso de punteros es cuando un arreglo es definido como un arreglo de punteros, por ejemplo, la siguiente declaración crea un arreglo de punteros de 5 elementos:

```
int *arrepunt[5];
```

donde cada una de las componentes del arreglo es un puntero a entero. Entonces, sentencias como las siguientes son correctas:

```
arrepunt[0]=&var1;
printf("\n%d",*arrepunt[0]);
```

entonces, al primer elemento del arreglo se le asignó la dirección de la variable entera `var1`*, posteriormente se imprimió el contenido de dicho puntero.*

malloc, free y sizeof

Muchas veces en nuestros programas necesitaremos ocupar memoria en el momento en que este se está ejecutando, para ello existen dos funciones malloc y free. La primera, pide memoria del conjunto de memoria disponible, free hace lo contrario, devuelve la memoria pedida para que pueda ser ocupada posteriormente. Estas funciones se encuentran en la librería llamada stdlib.h y son la base de la asignación dinámica de memoria en C.

Por ejemplo, si creamos un puntero a char y en algún momento necesitamos memoria para guardar algún dato, se puede hacer:

```
char *p1;
:
p1=malloc(20);
```

lo que nos permitirá crear 20 lugares de memoria para guardar datos, y `p1` *quedaría apuntando al primero de ellos. El parámetro pasado es un entero, pero en realidad el tipo*

definido en stdlib.h es ***size_t*** *que es parecido a un* ***unsigned int****. El prototipo de la función malloc es la siguiente:*

```
void *malloc(size_t cantidad_de_bytes)
```

El valor devuelto por malloc es un puntero (`void *`*) que se convierte a el tipo de puntero de la parte izquierda de la asignación. Sin embargo, para evitar problemas de portabilidad lo ideal es hacer la conversión explícita, de la forma:*

```
p1=(char *)malloc(20);
```

que convierte lo retornado por `malloc` *a un puntero a* `char`*.*

Para devolver la memoria solicitada, la instrucción sería:

```
free(p1);
```

En el ejemplo anterior un char ocupa un byte, no es problema pedir espacio para 20 char, sin embargo, si el puntero fuese entero, se tiene que multiplicar la cantidad pedida por el tamaño en bytes de un entero que es 2, entonces, si no sabemos el largo en bytes de un tipo o de una variable se nos complicaría pedir memoria, además que, dependiendo el computador los tipos podrían tener diferente largo.

Para resolver estos problemas y para hacer nuestro programa más portable existe el operador en tiempo de compilación llamado ***sizeof*** *(tamaño de) que retorna el tamaño en bytes del parámetro ingresado, que puede ser una variable o un identificador de tipo. Por ejemplo, si deseáramos imprimir el tamaño (en bytes) de un entero, lo correcto sería hacer:*

```
printf("%d",sizeof(int));
```

Suponga lo siguiente:

```
int a,matriz[20][10];
double var33;
:
printf("%d",sizeof(var33));
printf("%d",sizeof(matriz));
printf("%d\t\t%d",sizeof(a),sizeof(int));
:
```

Imprimiría el tamaño de las variables `var33`*,* `matriz` *(que sería 20*10*el largo de un entero) y el tamaño del tipo entero. En el ejemplo, sólo es obligatorio utilizar paréntesis cuando se pregunta el tamaño de un tipo.*

Con lo anterior podríamos crear memoria para un puntero a enteros usando

```
int *p2;
p2=(int *)malloc(10*sizeof(int));
```

lo que crearía espacio para 10 enteros (y `p2` *apuntaría al primero de ellos).*

Un ejercicio gráfico

El siguiente ejercicio pretende mostrar gráficamente que ocurre con las variables y punteros durante la creación y asignación.

```
int *p;
int z;
```

despues de la declaración anterior, es decir, crea una variable entera `z` *la cual tiene su espacio en memoria, aunque su valor es desconocido (basura). Además, una variable puntero* `p` *(a int) la cual aún no tiene espacio asignado (para contener valores) y no apunta a ningura parte.*

```
z=100;
```

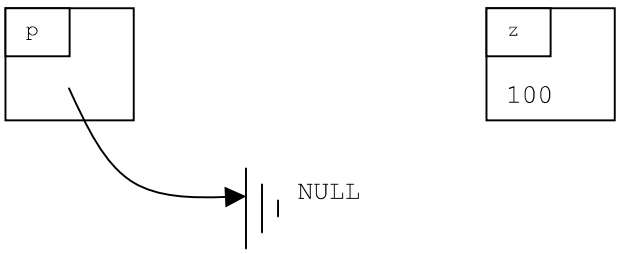

```
p=&z;
```

Despues de estas asignaciones, la variable `z` *contiene el valor* `100` *y la variable* `p` *apunta a la dirección donde está z. Entonces, recién ahora se puede hablar del contenido de* `p`*, es decir, si se ejecuta:*

```
printf("%d",*p);
```

Se imprime el valor `100`*. Si después,*

```
p=NULL;
```

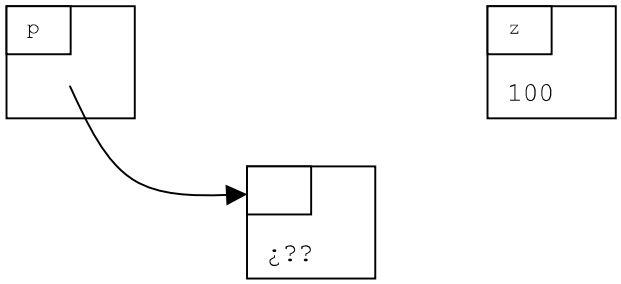


Donde `NULL` *es la forma de representar que el puntero apunta a* ***nada****. En la figura se muestra como barras paralelas decreciendo en tamaño.*

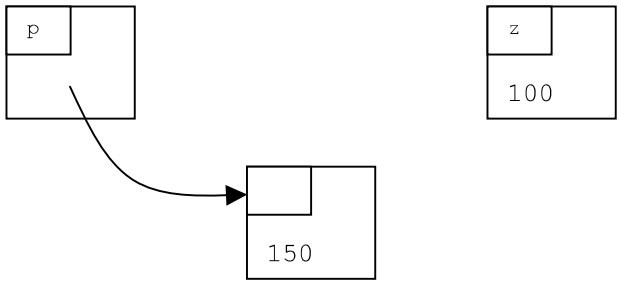
Si luego se hace:

```
p=(int *)malloc(sizeof(int));
```

se crea el espacio necesario en memoria para guarda un valor entero, a la que apuntará la variable `p`*, a este valor sólo se puede accesar a través de* `p`*. Gráficamente:*



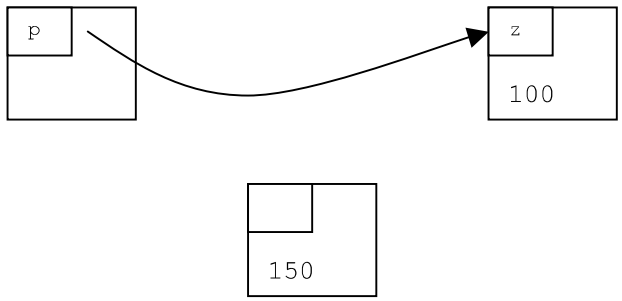

```
*p=150;
```



Después de esta asignación, el contenido de `p` *es igual a 150, y* `z` *aún mantiene el valor* `100`*.*

Si ahora hacemos nuevamente:

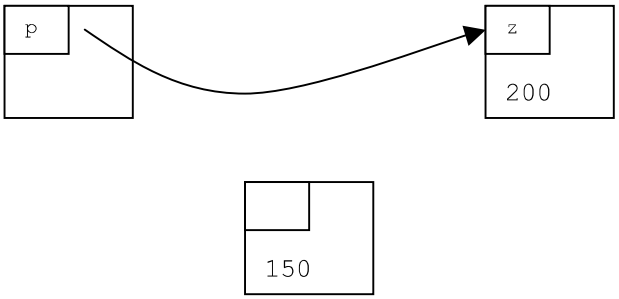
```
p=&z;
```



Con esto, en nuestra jerga diriamos que: "se nos perdio un puntero". En realidad, formalmente `p` *no está perdido, si no que ahora apunta a* `z`*, pero nos es imposible recuperar el valor que tenía antes. Como se muestra en la figura, no hay ninguna forma de acceder al espacio de memoria que contiene el valor* `150`*, siendo que ese espacio sigue ocupado. Si después de esto, se ejecuta la siguiente linea:*

```
*p=200;
```

ocurre lo siguiente:



Entonces, ahora, tanto `z` *como el contenido de* `p` *tienen el valor* `200`. *Finalmente, si:*

```
printf("%d %d",z,*p);
```

imprimiría en pantalla:

```
200 200
```

Un último comentario es recalcar que lo que almacena `p`*, es una dirección de memoria, es decir, la dirección de memoria a la que está apuntando, en los esquemas anteriores esta se representa a través de la flecha*

Problemas Propuestos

1.- *Para el siguiente extracto de programa:*

```
int *p,arreglo[6],i;
:
:                    /* datos del arreglo: 5 4 56 32 21 50 */
p=arreglo;
for(i=0;i<6;i++)
{
   printf("%d",*p);
   printf("%p",p);
   p++;
}
```

a) Indique que hace el programa.

b) Si el primer elemento del arreglo esta ubicado en la posición de memoria ***13500****, entonces, a que dirección de memoria queda apuntando* **`p`** *al finalizar el ciclo for?, escriba cada valor que va saliendo por pantalla.*

2.- *Escriba un programa que lea 100 números enteros y los almacene en un arreglo. Luego escriba una función que entregue a la función principal el valor máximo y mínimo, para ello ocupe parámetros pasados por referencia (a la función se le pasan tres parámetros: el arreglo y dos pasados por referencia que contendrán el máximo y mínimo).*

3.- *Escriba las funciones para calcular la potencia y factorial, perodonde los resultados sean devueltos en parametros pasados por referencia, es decir, la función no retorna nada (***`void`***).*

4.- *Para cada uno de los siguientes programas, imprima el resultado que aparece en pantalla. Adicionalmente, haga el esquema para la asignación de valores a las variables y punteros (como en clases).*

a)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int main()
{
   int i=0,a=15,b=35,*p,*q,*r;
   q=(int *)malloc(MAX*sizeof(int));
   p=q;
   for (i=0;i<MAX;i++)
   {
      *p=i*10;
      p++;
   }
   r=q;
   for (i=0;i<MAX;i++,r++)
      printf("%d, ",*r);
   printf("\n");
   r=&a;
   *r=b;
   p=q+2;
   printf("%d,%d,%d,%d",a,b,*r,*p);
   return 0;
}
```

b)

```
#include <stdio.h>
#include <stdlib.h>

void quehace(int **bla, int **ble, int bli);
int main()
{
   int a=15,b=35, c=45,*p,*q;
   p=&a;
   q=&b;
   quehace(&p,&q,c);
   printf("%d,%d,%d,%d,%d",a,b,c,*p,*q);
   return 0;
}

void quehace(int **bla, int **ble, int bli)
{
   int *aux;
   aux=*bla;
   *bla=*ble;
   *ble=aux;
   bli++;
   return;
}
```