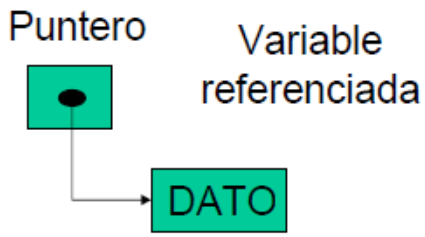
TUTORIAL PUNTEROS EN C

Definición de Punteros:

Por definición un puntero es una variable que puede almacenar la dirección en memoria de otra variable. Un puntero es una entidad, que puede ser variable o constante, que siempre contiene una dirección de memoria (valida, inválida o nula). Para definir una variable puntero en c se utiliza la siguiente forma:

```
<TipoVariableApuntada> *NombreVariablePuntero;
```



Una variable de tipo puntero debe ser de igual tipo que la variable cuya dirección en memoria contiene, o dicho de otra forma, la variable a la que apunta.

Por ejemplo, una variable puntero llamada **p** que pueda almacenar referencias a posiciones de memoria donde se almacenen objetos de tipo **int** se declara así:

```
int * p;
```

El lenguaje C contiene dos operadores de punteros especiales: * y &. El operador **&** devuelve la dirección de la variable a la que precede. El operador * devuelve el valor almacenado en la dirección. Por ejemplo examine este breve programa:

Ejemplo 1

```
#include <stdio.h>

main()
{

  int *p,q;
  q=100; /*Asignar 100 a q*/
  p=&q; /* Asignar a p la direcciòn de q */
  printf("valor de q : %d \n", q);
  printf("valor de la direccion de q: %p \n", p);
  printf("valor de q por medio del puntero p: %d \n", *p);

}
```

La salida en pantalla es la siguiente:

```
valor de q : 100
valor de la direccion de q: 0xbfad4dc8
valor de q por medio del puntero p: 100
```

Ejemplo 2

El siguiente programa asigna indirectamente un valor a q utilizando un puntero p.

```
#include <stdio.h>

main()
{

  int *p,q;
  q=100; /*Asignar 100 a q*/

  p=&q; /* Asignar a p la direcciòn de q */

  *p=205; /* Asignar 205 a q*/

  printf("valor de q : %d \n", q);

}
```

La salida en pantalla es la siguiente:

```
valor de q : 205
```

Arreglo de punteros

Los punteros pueden estructurarse como arreglos igual que cualquier otro tipo de datos. Por ejemplo, un arreglo de nombre **pu** que puede almacenar punteros a variables de tipo int se declara así:

```
int *  pu[];
```

Para asignar una dirección de memoria a uno de los punteros del arreglo, se realiza como se indica en el ejemplo:

```
pu[9] = &mivar;
```

La dirección de la variable entera llamada **mivar** se asigna al décimo elemento del array.

Se debe aclarar que el nombre de un array es en sí mismo un puntero a la primera (0-ésima) posición del vector.

Ejemplo 3:

```
int v[10];
int i;
int * p;

for (i=0; i < 10; i++)
{
   v[i] = i; /* Asignamos valores iniciales*/
}

for (i=0; i < 10; i++)
{
    printf ("\n%d", v[i]); /*Mostramos los valores del vector*/
}

p = v; /* p apunta a la primera posición del vector v */

for (i=0; i < 10; i++)
{
     printf ("\n%d", *p++);
    /* Tras cada p++ el puntero señala a la siguiente posición en v */
}
```

Void

Hay un tipo especial de puntero que es capaz de almacenar referencias a objetos de cualquier tipo. Éstos punteros se declaran indicando **void** como **<tipo>**. El valor **void** indica que el puntero apunta a ningún tipo específico de objeto. Es decir, no se da información sobre el tipo apuntado. Por ejemplo:

```
void * punteroACualquierCosa;
```

Un puntero no tiene porqué almacenar referencias a objetos de algún tipo en concreto. Por ejemplo, pueden existir punteros **int** * que en realidad apunten a objetos **char**, o punteros **void** * que no almacenen información sobre el tipo de objeto al que debería considerarse que apuntan, o punteros que apunte a direcciones donde no existan objetos, etc.

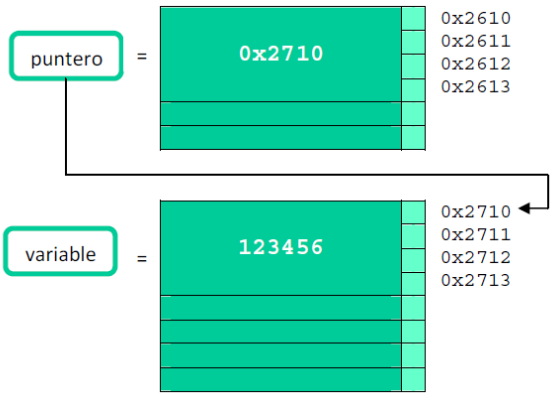
Asignación de Punteros

Cuando el compilador encuentra la declaración de una variable, le asigna una dirección en memoria en la que se almacenarán los valores que sucesivamente se le asignen a esa variable. El operador **&** antepuesto al nombre de una variable retorna el valor de la dirección inicial de memoria donde el compilador almacenó o almacenará el valor de la variable. Para acceder al valor que se encuentra en la dirección apuntada por un puntero se debe quitar la referencia del puntero colocando *

delante del mismo. Por estos motivos se suele denominar a **&** el operador de referencia y a ***** el operador de indirección.

Ejemplo 4

```
int *puntero; // Declara un puntero a una variable tipo int
int variable; /* Declara una variable tipo int. El compilador identifica y
asigna una dirección (supongamos 0x2710, en Hexadecimal) al identificador
<variable> */
variable = 123456; /* Almacena en las direcciones 0x2710, 0x2711, 0x2712 y
0x2713 el valor <123456>, tomando en cuenta que una variable de tipo int
ocupa 4 bytes. Verifíquenlo en su compilador con sizeof(int) */
puntero = &variable; // Almacena 0x2710 en <puntero>, apunta a la primera
dirección
```



El operador * antepuesto a una variable puntero devuelve el valor que ésta almacena, partiendo de su dirección inicial, en un número de celdas de memoria, que viene definido por el tipo de la variable apuntada. Entonces, siguiendo con el ejemplo anterior:

```
int otra_variable; // Declara una variable tipo int
otra_variable = *puntero; /* Almacena en <otra_variable> el valor <123456>
el valor que contiene <variable> que es apuntada por <puntero> */
```

Punteros a Punteros:

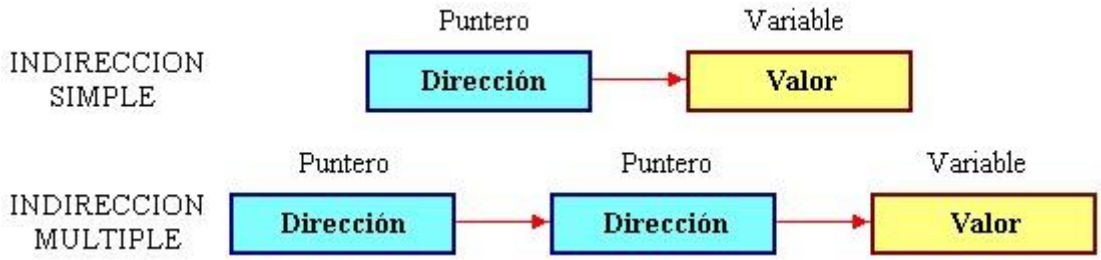
Variable cuyos posibles valores son direcciones de otras variables puntero, como se muestra en el siguiente ejemplo de declaración de un puntero a punteros de tipo **int** llamando punteroApunteros:

int ** punteroApunteros;

Ejemplo 5

Declaración de un puntero que apunte a un puntero a entero.

```
int **a;
```



Para guardar en **a** el entero 5 (usando todas las variables auxiliares necesarias):

```
int b = 5, *c;
    c = &b;
    a = &c;
```

Para mostrar el valor de b:

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

Utilización de punteros como parámetros

En el lenguaje C, por defecto, todos los parámetros a las funciones se pasan por valor (la función recibe una copia del parámetro, que no se ve modificada por los cambios que la copia sufra dentro de la función). Para pasar un parámetro por referencia es posible pasando la dirección de la variable, en lugar de la propia variable.

Ejemplo 6

{VERSION ERRONEA}

```
intercambia (int a, int b) {
int tmp;

tmp = a;
a = b;
b = tmp
}
```

{VERSION CORRECTA}

```
intercambia (int *a, int *b) {
int tmp;

tmp = *a;
*a = *b;
*b = tmp
}
```

Punteros a Estructuras:

Podemos especificar una variable puntero que apunta a una estructura. El uso de estas variables se emplea frecuentemente cuando es necesario pasar la estructura por referencia a una función o cuando se quiere hacer una reserva dinámica de memoria para una estructura en un determinado momento de la ejecución del programa.

En general la sintaxis para declaración de punteros a estructuras sigue el siguiente formato:

```
struct <nombre_estructura> *<variable_puntero>;
```

Ejemplo 7

```
//Declaración de la estructura
struct  Punto{
      int x;
      int y;
}typedef Punto;

//En este caso, nombre_estructura es "Punto", con lo que la declaración sería:
struct Punto *puntero;

//Opcionalmente puede incluirse un valor inicial así:
struct Punto registro;
struct Punto *puntero = &registro;
```

Lectura / Escritura Básica de Punteros a estructuras:

```
(*puntero).x = 3;
(*puntero).y = 4;
```

Los paréntesis son necesarios, ya que el operador de acceso (.) tiene mayor precedencia que el de indirección. Por esta razón, ***puntero.x*** es interpretada como ***(puntero.x)***, lo que es erróneo, ya que ***puntero.x*** no es un puntero (en este caso es un entero).

Lectura/Escritura con el Operador ->:

Los punteros son muy importantes para el manejo de matrices y estructuras. El operador que se emplea para acceder a los miembros de una estructura a través de un puntero es **"->".**

puntero->campo

Donde **campo** es uno de los campos de la estructura.

Lo anterior es equivalente a escribir: **(*puntero).campo**

Ejemplo 8

```
struct Dato
{
  int campo1, campo2;
  char campo3 [30];
} typedef Dato;

struct Dato x;
struct Dato *ptr;

ptr = &x;
```

```
//Esto es equivalente a: (*ptr).campo1 = 33;
ptr->campo1 = 33;

//Esto es equivalente a: strcpy ((*ptr).campo3,"hola");
strcpy ( ptr->campo3, "hola" );
```

Llamado de funciones mediante punteros:

Una función tiene una dirección física en memoria que puede asignarse a un puntero, aunque la función no es una variable. La dirección de la función es el punto de entrada de la función; por lo tanto el puntero a función puede utilizarse para llamar dicha función.

Un puntero a función es una dirección, usualmente un segmento de código, donde el código ejecutable de la función está almacenado. La forma general para declarar un puntero a una función es la siguiente:

```
tipo (*nombre_puntero_funcion)();
```

Donde **tipo** es el tipo retornado por la función.

Ejemplo 1: Formas de declaración de punteros a función:

```
void (*func)();     /* No recibe argumentos y no retorna nada */
void (*func)(int); /* No retorna nada pero toma un argumento entero*/
```

- La dirección de la función se obtiene utilizando el nombre de la función sin paréntesis ni argumentos (similar a como se obtiene la dirección de un array).
- Para llamar a una función apuntada por un puntero, se usa el nombre del puntero como si fuera una función.

Ejemplo 2: Utilizando un puntero a función.

```
int suma (int a, int b)
{
  return a+b;
}

int resta (int a, int b)
{
  return a-b;
}
// Declaramos un puntero a funciones con dos parámetros enteros que
devuelven un entero
int (*funcion) (int,int);
{
funcion = suma;          // 'funcion' apunta a 'suma'
 x = funcion(4,3);       // x=suma(4,3)
 funcion = resta;        // 'funcion' apunta a 'resta'
 x = funcion(4,3);       // x=resta(4,3)
}
```

Asignación Dinámica de Memoria:

La asignación dinámica es la forma en la que un programa puede obtener memoria mientras se está ejecutando. A las variables globales se les asigna memoria en tiempo de compilación y las locales usan la pila. Sin embargo, durante la ejecución no se pueden añadir variables globales o locales, pero existen ocasiones en las que un programa necesita usar cantidades de memoria variables.

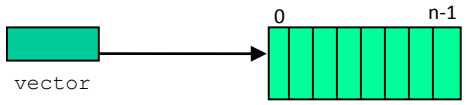
El centro del sistema de asignación dinámica está compuesto por las funciones (existentes en la biblioteca ***stdlib.h***) ***malloc()***, que asigna memoria; y ***free()*** que la devuelve.

El prototipo de la función *malloc()* es:

```
void *malloc(size_t numero de bytes);
```

Por ejemplo:

```
vector= (int *)malloc(n*sizeof (int));
```



Tras una llamada ***malloc()*** devuelve un puntero, el primer byte de memoria dispuesta. Si no hay suficiente memoria libre para satisfacer la petición de ***malloc(),*** se da un fallo de asignación y devuelve un nulo.

Ejemplo 1: El fragmento de código que sigue asigna 1000 bytes de memoria.

```
char *p;
p = (char*)malloc (1000);
```

Después de la asignación, ***p*** apunta al primero de los 1000 bytes de la memoria libre.

Ejemplo 2: El siguiente ejemplo dispone espacio para 50 enteros. Se observa el uso de *sizeof* para asegurar la portabilidad:

```
int *p;
p=(int*)malloc(50*sizeof(int));
```

La función ***free()*** es la opuesta de *malloc()* porque devuelve al sistema la memoria previamente asignada. Una vez que la memoria ha sido liberada, puede ser reutilizada en una posterior llamada a *malloc().* El prototipo de la función *free()* es:

```
void free (void *p)
```

Ejemplo 3:

```
free (p);
```