**UNIDAD 5 – PUNTEROS**

**ÍNDICE**

1.. Concepto de puntero.

2.- Asignación de una dirección a un puntero.

3.- Aritmética de punteros.

4.- Punteros a punteros

5.- Los punteros y los argumentos a funciones

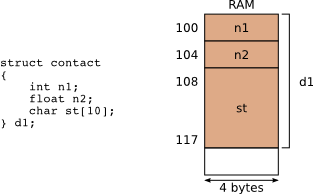
**1.. Concepto de puntero.**

Los punteros son uno de los aspectos más potentes de la programación en C, pero también uno de los más complejos de dominar. Los punteros permiten manipular la memoria del ordenador de forma eficiente. Dos conceptos son fundamentales para comprender el funcionamiento de los punteros:

* El tamaño de todas las variables y su posición en memoria.
* Todo dato está almacenado a partir de una dirección de memoria. Esta dirección puede ser obtenida y manipulada también como un dato más.

Los punteros son también una de las fuentes de errores más frecuente. Dado que se manipula la memoria directamente y el compilador apenas realiza comprobaciones de tipos, el diseño de programas con punteros requiere una meticulosidad muy elevada que debe ir acompañada de una dosis idéntica de paciencia. Programar eficientemente usando punteros se adquiere tras escribir muchas líneas de código, pero requiere una práctica sostenida.

Todos los datos se almacenan a partir de una dirección de memoria y utilizando tantos bytes como sea necesario. Por ejemplo, considera la siguiente definición de datos y su correspondiente representación en memoria a partir de la dirección 100.

Asumiendo que int y float ocupan 4 bytes cada uno, y el tipo char ocupa 1 byte, si d1 está almacenada a partir de la posición 100, entonces su campo n1 tiene esa misma dirección, el campo n2 está en la posición 104 y el campo st está almacenado a partir de la posición 10.

En C se obtiene la dirección de memoria de cualquier variable mediante el operando & seguido del nombre de la variable. En la figura anterior, la expresión &d1 devuelve el valor 100.

El valor que devuelve el operador & depende de la posición de memoria de su operando, y por tanto, no está bajo el control del programador. A la dirección de memoria almacenada como dato en la posición 200 se le denomina “puntero” pues su valor “apunta” a donde se encuentra la variable d1. Otra forma de enunciarlo: en la posición 200 hay un puntero a d1.

Si en la posición 200 está almacenado un puntero a la variable d1 se pueden acceder a los datos de esta estructura mediante una “indirección”. Se toma el dato almacenado en la posición 200 y su valor (el número 100) se interpreta ahora como una dirección. Se accede a esa dirección y de ahí se accede a los campos de d1. Acabamos de acceder a d1 de forma indirecta, o a través de una “indirección”.

La indirección se puede aplicar múltiples veces en un mismo acceso en lo que se conoce como “indirección múltiple”. Siguiendo con el ejemplo anterior, podemos almacenar ahora la dirección del puntero (esto es, el valor 200) en otra posición de memoria, por ejemplo, la posición 300. En esta posición 300 tenemos la dirección de la dirección de d1. O de forma análoga, en la posición 300 hay un puntero a un puntero a d1. Igualmente, se puede acceder a los datos de d1 pero esta vez mediante una doble indirección. Se pueden construir indirecciones múltiples con un número arbitrario de niveles.

El operador & seguido del nombre de una variable devuelve su dirección de memoria. El tipo de datos del resultado es “puntero a” seguido del tipo de la variable utilizada. La regla para obtener la sintaxis y el significado de estos tipos de datos es:

*Para todo tipo de datos T existe un tipo de datos que se denomina “Puntero a T” definido como “T \*”.*

Así, por ejemplo,

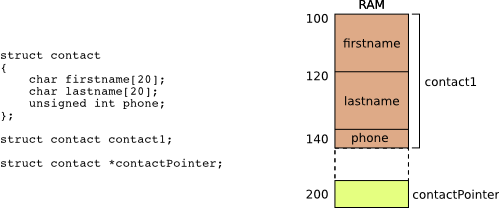
int \*a, \*b, \*c;

char \*s, \*t;

define varios punteros a enteros y char.

Además de los punteros a los tipos de datos creados, C permite declarar un puntero genérico de tipo void \*. Una variable de este tipo almacena un puntero a cualquier dato. Se recomienda utilizar este tipo de datos sólo cuando sea estrictamente necesario.

El tamaño de los punteros es siempre el mismo independientemente del dato al que apuntan porque todos ellos almacenan una dirección de memoria. Para las estructuras de datos, la regla se aplica de forma idéntica. El siguiente ejemplo muestra la definición de una estructura y la declaración de una variable de tipo estructura y otra de tipo puntero a esa estructura:



La variable contact1 es de tipo estructura y ocupa 44 bytes (20 + 20 + 4), mientras que contactPointer es de tipo puntero y ocupa 4 bytes.

**2.- Asignación de una dirección a un puntero.**

Dada una variable var de tipo t y una variable var\_ptr de tipo puntero a t (t \*) es posible asignar

var\_ptr = &var

Los punteros se comportan como el resto de variables, no tienen valor inicial al comienzo del programa. La dirección de una variable existe desde el comienzo de un programa, aunque no tenga valor asignado. Por eso es correcto definir un puntero a una variable, aunque ésta no tenga valor definido aún.

Un puntero siempre está asociado a objetos de un tipo: sólo puede apuntar a objetos (variables o vectores) de ese tipo.

int \*ip; /\* Sólo puede apuntar a variables enteras \*/

char \*c; /\* Sólo puede apuntar a variables carácter \*/

double \*dp, /\* dp sólo puede apuntar a variables reales \*/

float atof (char \*);

En el siguiente ejemplo mostramos el uso de los punteros

*int y=100;*

*int \*ip;*

*ip = &y;*

*printf ("\n y=%d, \*ip=%d, i= %d", y, \*ip, ip);*

El valor de y está definido en el código y es 100. Ip es un puntero a la dirección en la que se encuentra y y el resultado podría ser 2358852 (dirección de memoria) y, por último, \*ip es el contenido de esa dirección de memoria, es decir, 100 (valor inicial de y).

Si ahora hacemos

\*ip=200;

El valor de y sería 200 ya que lo que hemos hecho es guardar en la dirección 2358852 este valor. El puntero no cambia de valor sólo el contenido de la memoria

**3.- Aritmética de punteros**

Los punteros son variables de tipo int y, por lo tanto, se puede realizar operaciones con ellos.

Para aclarar estos conceptos veamos algunos ejemplos

#include <stdio.h>

int main()

{

int y=100;

int \*ip;

ip = &y;

\*ip = \*ip + 10;

printf (" y= %d, \*i= %d, i= %d", y,\*ip, ip);

return 0;}

Y es una variable de tipo entero y tiene asignado el valor 100. Ip es un puntero que apunta a la dirección de y. Por tanto, ip es, por ejemplo, 324534 (suponiendo que esta sea la dirección de memoria asignada a y) mientras que îp contiene el valor de la variable (que es 100)

Si ahora hacemos

y = \*ip + 10;

printf ("\n y=%d, \*ip=%d", y, \*ip);

El resultado será que, puesto que \*ip es el contenido de la dirección a la que apunta, es decir y, al sumarle 10 obtendremos 100+10=110 mientras que \*ip sigue siendo 100.

Además, si hacemos

\*ip += 1;

printf ("\n y=%d, \*ip=%d", y, \*ip);

estamos sumando 1 al contenido al que apunta ip, es decir, al valor de y. Por tanto, y=101 además de cumplirse que \*ip=101.

Es necesario utilizar paréntesis cuando aparezcan en la misma expresión que otros operadores unarios como ++ o --, ya que en ese caso se evaluarían de izquierda a derecha

Por ejemplo,

(\*ip)++;

printf ("\n y=%d, \*ip=%d", y, \*ip);

\*ip++;

printf ("\n y=%d, \*ip=%d", y, \*ip);

son instrucciones diferentes ya que la primera incrementa el contenido de la dirección de memoria a la que apunta ip (es decir, incrementa y) y por tanto el resultado sería 101. La segunda instrucción incrementa la dirección del puntero. Ya no apunta a a la dirección 2358852 sino que la incrementamos (los punteros son int de 4 bytes, por tanto, incrementar es sumar 4) y ahora vale 2358856.

Dado que los punteros son variables, también pueden usarse como asignaciones entre direcciones. Así:

int \*ip, \*iq;

iq = ip;

Indica que iq apunta a donde apunte el puntero ip.

**4.- Punteros a punteros**

Los punteros también pueden representar el concepto de indirección múltiple. La regla de derivación por la que de un tipo T se deriva el tipo “puntero a T” también se puede aplicar a este nuevo puntero para obtener el tipo “puntero a puntero a T” definido como “T \*\*”.

Estos punteros múltiples tienen el mismo tamaño que un puntero simple, la diferencia sólo está en el número de indirecciones para obtener el dato. El siguiente programa es un ejemplo en el que se manipulan punteros con varios niveles de indirección:

#include <stdio.h>

int main()

{

int y=100;

int \*ip;

int \*\*doble;

ip = &y;

doble=&ip;

printf ("\n y=%d, \*ip=%d, doble= %d", y, \*ip, \*doble);

return 0;

}

Analizando el código vemos que ip es un puntero a y, su contenido es la dirección en la que se encuentra y mientras que \*ip contiene el valor de dicha dirección. Por tanto, doble contiene la dirección en la que se encuentra el puntero ip y \*doble contiene el contenido de ip y, por último, \*\*doble contiene el contenido de la dirección a la que apunta ip, es decir, el valor de y (100).

**5.- Los punteros y los argumentos a funciones**

En C, por defecto, todos los parámetros a las funciones se pasan por valor (la función recibe una copia del parámetro, que no se ve modificado por los cambios que la copia sufra dentro de la función). Si llamamos a una función para que conmute los valores de dos variables no podemos pasarle los valores de las variables sino un puntero a esas variables.

En este primer ejercicio intercambiamos los valores de a y b pero sólo dentro de la función. Fuera de la función las variables siguen teniendo los mismo valores.

#include <stdio.h>

mal(int a, int b)

{ int tmp; tmp = a; a = b; b = tmp;}

int main()

{

int a,b;

a=4;b=7;

mal(a,b);

printf("a=%d, b=%d", a,b);

return 0;

}

Para que un parámetro de una función pueda ser modificado, ha de pasarse por referencia, y en C eso sólo es posible pasando la dirección de la variable en lugar de la propia variable. Si se pasa la dirección de una variable, la función puede modificar el contenido de esa posición (no así la propia dirección, que es una copia).

En este segundo ejemplo pasamos los valores de a y b por referencia (&a y &b) de forma que la función pueda procesarlos y modificarlos para que al volver al programa principal ya estén intercambiados.

#include <stdio.h>

bien(int \*a, int \*b)

{ int tmp; tmp = \*a; \*a = \*b; \*b = tmp;}

int main()

{

int a,b;

a=4;b=7;

bien(&a,&b);

printf("a=%d, b=%d", a,b);

return 0;}