**PROGRAMACION**

**CHAR:**

Ocupa 1 Byte y normalmente se utiliza para almacenar variables que su valor es un carácter.

**VOID:**

En C, **void** se utiliza como una función de retorno de tipo. Indica que la función no devuelve un valor. Cuando void aparece en una declaración de puntero, especifica que el puntero es universal. Cuando es utilizado en una función con una lista de parámetros indica que la funciones no coge parámetros.

**FOR:**

Diagrama

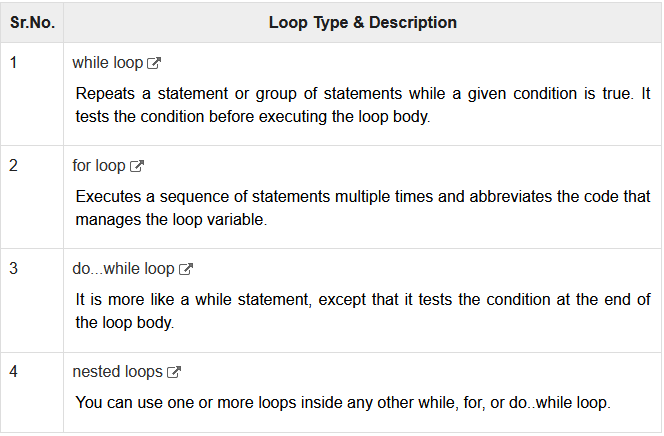
Descripción generada automáticamente El bucle **for** es una estructura controlada de repetición que permite escribir un bucle eficientemente que necesita ser ejecutado un cierto número de veces.

**Init** se ejecuta el primero, y solo una vez, esto te permite declarar e inicializar cualquier loop de variables.

Luego **condition** es evaluada, si es verdadera el contenido se ejecuta, en caso contrario ignora el bucle **for**.

Después de que el bucle **for** es ejecutado, vuelve para implementar la declaración **increment.** Esta declaración puede dejarse en blanco y ejecutar solo el contenido del bucle.

Después de esto, el bucle re repite hasta que la declaración se convierte en falsa.

**PUNTEROS:**

Dos conceptos son fundamentales para comprender el funcionamiento de los punteros:

1. El tamaño de todas las variables y su disposición en la memoria.

2. Todo dato esta almacenado a partir de una dirección de memoria. Esta dirección puede ser obtenida y manipulada como un dato más.

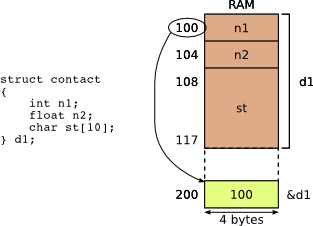
Todo dato tiene una dirección de memoria, utilizando tantos bytes como sea necesario

Un reloj con fondo negro y letras blancas

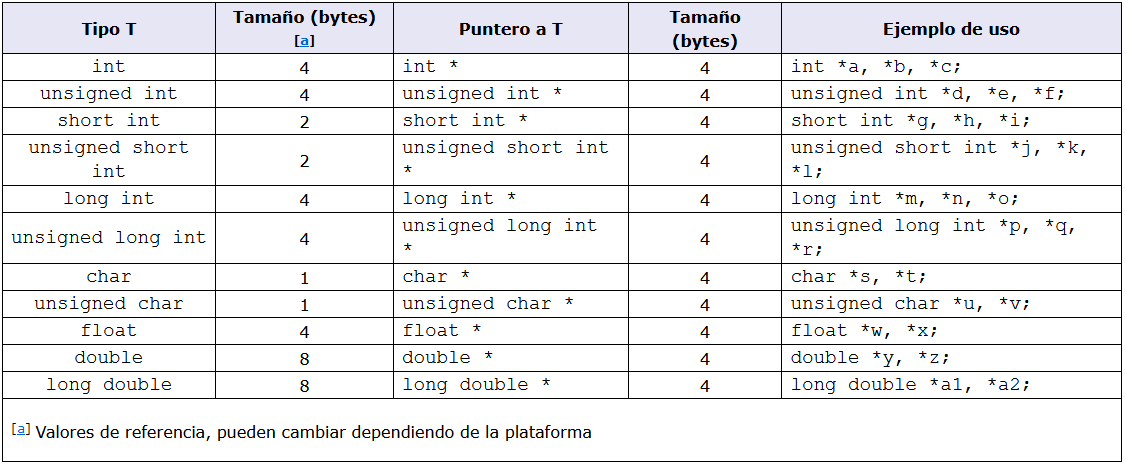
Descripción generada automáticamente con confianza mediaAsumiendo que **int** y **float** ocupan 4 bytes cada uno y un **char** ocupa 1 byte, si **d1** esta almacenado a partir de la posición 100, entonces su campo **n1** tiene que estar almacenado en la misma posición, el campo **n2** está en la posición 104 y el campo **st** esta almacenado a partir de la posición 108.

Supongamos que cada dirección de memoria ocupa 32 bits (4 bytes). Entonces podemos almacenar **d1** en otra dirección de memoria que ocupa 4 bytes, en este caso 200.

Una dirección de memoria se obtiene con el operando **&**, seguido del nombre de la variable.

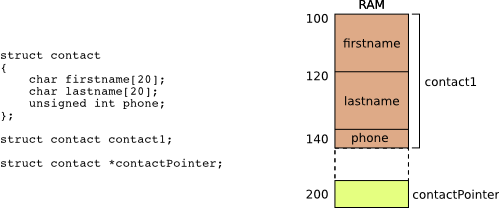
En este caso, en la dirección de memoria 200 hay un puntero a **d1.** En esta dirección de memoria se puede acceder a la variable **d1** de forma indirecta. Esta “indirección”, se puede almacenar múltiples veces a través de una “indirección múltiple”. De esta manera, ahora podemos almacenar la dirección de 200 en otra dirección, por ejemplo 300 y desde 300 acceder a **d1**.

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

En la siguiente tabla se muestran las consecuencias de aplicar esta regla a los tipos de datos básicos en C, así como ejemplos de declaración de variables.

Además de 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. Es recomendable usarlo solo cuando es estrictamente necesario.

El tamaño del puntero es siempre el mismo independientemente del dato que apunte por que 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.

Asignado un numero a un puntero que apunta a la dirección de memoria de una variable, cambiara el contenido de la variable. Si: **p1=num1** entonces si **p1=30,** la variable **num1** pasara a tener el valor **30**.

**FUNCIONES:**

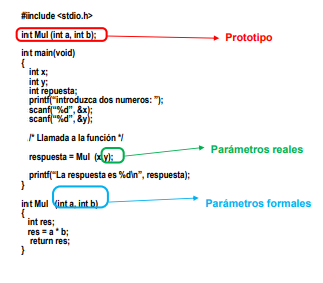
Una función en C es un segmento independiente de código fuente diseñado para hacer una tarea especifica y fácilmente reutilizable. Qué ventajas tiene:

* Ayuda a dividir un programa grande en componentes mas pequeños y de esta forma permite el desarrollo modular de los programas.
* Evita la necesidad de repetir las mismas instrucciones en diferentes partes del programa
* Permite transferir un conjunto de datos distinto cada vez que se accede a ellas

La función se puede dividir en varias partes, primero el tipo de dato (**int**, **void**…), después ira el nombre de la función con el que mas tarde la llamaremos. Al final del nombre se pone entre paréntesis una serie de declaraciones de variables que tomaran parte en la función. Abrimos corchetes **{}** y dentro de ellos irán las instrucciones de la función que se ejecutarán cuando la llamemos. Al final de las instrucciones habrá un **return**, dando el valor a devolver. En el caso de **void,** esta parte no hace falta. No es aconsejable que haya más de un **return**.

Al principio del programa y antes del **main** se pondrá el **prototipo** de la función, terminado en; , esta práctica es muy aconsejable ya que permite al compilador averiguar si hay algún error, en caso contrario no avisará.

Hay dos tipos de parámetros (variables) en una función: los parámetros formales (variables locales que aparecen en la cabecera de la función) y los parámetros reales (Aquellas variables que se utilizan para hacer referencia a la función). Ambos parámetros no tienen por qué llamarse igual, pero tienen que coincidir en tipo y orden.



**RETURN:**

Produce una salida inmediata de la función y vuelve al programa que le hizo la llamada. Devuelve un valor concreto y solo uno como resultado de la función. Como return hace que la expresión devuelva un valor es posible incluir la llamada a la función como operando de una expresión.

**RECURSIVIDAD:**

Es el proceso mediante el cual una funcion puede llamarse a si misma. En una funcion recursiva, una sentencia en el cuerpo de la funcion invoca a la propia funcion.

Estas funciones se suelen hacer en problemas repetitivos, en los que la iteración se determina mediante el resultado anterior.