PROGRAMANDO CON LENGUAJE C

RICARDO W. SÁNCHEZ SCHULZ

A CLARY Y MIS HIJOS RICARDO Y PAULA

ACERCA DEL AUTOR

Ricardo W. Sánchez Schulz es actualmente profesor titular del Departamento de Ingeniería Eléctrica de la Universidad de Concepción en CHILE.
Obtuvo el grado de Doctor en la Universidad del Estado de Ohio, U.S.A. y posteriormente realizó un postdoctorado en el Imperial College de Londres.
Su ámbito de trabajo está relacionado con los sistemas digitales, visualización y computación gráfica.
Dentro de una vasta trayectoria en el desarrollo de productos computacionales, está la construcción de un simulador computacional para el entrenamiento de submarinistas de la Fuerza de Submarinos de la Armada de Chile. Trabajó también, en el desarrollo de un simulador gráfico tridimensional, basado en elementos hápticos, para operaciones quirúrgicas a la rodilla.
En otro proyecto, usó elementos hápticos para la enseñanza de las matemáticas y ciencias para colegios básicos e intermedios.
Actualmente enseña programación, computación gráfica y arquitectura de computadores en el departamento de Ingeniería Eléctrica de la Universidad de Concepción.
En su tiempo libre disfruta de la lectura y pintura.
Cualquier consulta con el autor, enviar un email a **ricardo@onewayar.org**

Gobierno de CHILE: Departamento de Derechos Intelectuales
Inscripción Nº 294.392 del libro Programando con Lenguaje C
2018

ÍNDICE GENERAL

Capítulo 1: INTRODUCCIÓN

Capítulo 2: NÚMEROS, DATOS Y TIPOS

Capítulo 3: ARREGLOS Y ASIGNACIÓN DINÁMICA DE MEMORIA

Capítulo 4: EXPRESIONES, BUCLES, SENTENCIAS DE SELECCIÓN VARIABLES ESTÁTICAS y scanf

Capítulo 5: FUNCIONES, typedef, PUNTEROS A FUNCIONES, STACKS Y RECURSIÓN

PREFACIO

¿Por qué escribir un nuevo libro sobre programación usando lenguaje C? Mi experiencia, basada en la enseñanza del uso de este lenguaje para aprender a programar, durante más de 15 años a alumnos que se inician en la programación, me ha llevado a tomar esta decisión por varios factores:

1. El lenguaje C, siendo un lenguaje compacto con un limitado número de características, presenta dificultades a los alumnos por su extensivo uso de punteros (direcciones de objetos) y cercanía con el hardware. En la enseñanza de este lenguaje, era estándar incorporar el uso de punteros pasado la mitad de la materia. Como consecuencia de esta metodología, los alumnos al terminar el curso no comprendían a cabalidad el uso de punteros y su diagnóstico era que el lenguaje C es difícil de aprender. Hace unos 5 años, modifiqué esta metodología y comencé a enseñar lenguaje C incorporando punteros a partir de la primera clase. Para hacer posible esto, hago uso de analogías que permitan a los alumnos asociar punteros (direcciones) con objetos, que para ellos es de comprensión inmediata.
2. Mis años de experiencia, enseñando este lenguaje, me han permitido localizar los puntos donde los alumnos tienen dificultades para comprender. Para corregir esto, en el libro hago uso extensivo de diagramas que permiten visualizar mejor las dificultades y, por ende, mejorar notablemente su comprensión.
3. En el libro, entrego muchos problemas resueltos, que enfatizan los tópicos que presentan mayores dificultades, como creación de tipos complejos, funciones, el uso de punteros a funciones, arreglos dinámicos etc.

El resultado de estos cinco años, usando esta nueva metodología, ha sido realmente satisfactorio y en mi opinión, han generado un cambio en los alumnos en su predisposición hacia el aprendizaje del lenguaje C. He podido comprobar, que ya en el último tercio de un curso, los alumnos manejan con naturalidad el uso de punteros. Considero que todo ingeniero, debe saber razonablemente bien como programar usando lenguaje C y espero que este nuevo libro contribuya en esa dirección.

LeanPub.com me permite publicar este libro en sus primeros cinco capítulos y primera edición, lo cual agradezco y ciertamente me motiva a terminar los próximos capítulos y mantener, en la medida de lo posible, actualizado este libro.

1 INTRODUCCIÓN

Si un profesional trabaja desarrollando software en investigación genética, será mucho más efectivo si comprende razonablemente bien biología molecular.
Bjarne Stroustrup

1.0 Origen y versiones del Lenguaje C

En 1970, Ken Thompson de los laboratorios Bell escribió UNIX para el computador DEC PDP-7. Este sistema operativo se escribió en lenguaje assembly, y pronto Ken Thompson decidió desarrollar otro lenguaje llamado B, que le permitiera mejorar el UNIX del PDP-7. Thompson se basó en un lenguaje de la época llamado BCPL, una especie de derivado del lenguaje ALGOL de los años 60, para escribir B. Posteriormente, Dennis Ritchie se unió al proyecto UNIX y comenzó a programar en lenguaje B. Ritchie comenzó a elaborar una extensión del lenguaje B que fuera mas adecuado para la programación de UNIX, que llamó NB para finalmente derivar en el nombre C. Cuando el lenguaje C alcanzó cierta madurez, en 1973, UNIX fue totalmente reescrito en lenguaje C. Con este nuevo lenguaje C, UNIX logró una excelente portabilidad, que se mantiene en la actualidad.
En 1978, aparece el primer libro “El Lenguaje de Programación C”, que pasó a ser la biblia de los programadores de la época. Los autores de este libro fueron Brian Kernighan y Dennis Ritchie. En diciembre de 1989, apareció el primer estándar del lenguaje C, llamado ANSI estándar X3.159-1989. En 1990 aparecen las versiones C89 o C90, aprobadas por la Organización Internacional para Estándares ISO. En 1999 se publica otro estándar denominado C99 y posteriormente en el año 2011 le sigue la versión C11. C17(2018) es una corrección de errores de C11, pero sin nuevas características. En este libro se indica cuándo una característica del lenguaje C es válida sólo en versión C99 o posteriores.

1.1 Características del Lenguaje C

Lenguaje C es un lenguaje para programación de propósitos generales, y por su origen, permite estar cerca del hardware por su poderoso sistema de direcciones o punteros, que permiten el acceso a cualquier espacio (bytes o bits) de una memoria o registros (hardware). Se considera un lenguaje de bajo y mediano nivel, aún cuando permite crear abstracciones complejas.
Es un lenguaje con un limitado número de opciones si se compara con otros lenguajes de alto nivel, lo que permite que sus programas puedan ser instalados en memorias de pequeño tamaño como sistemas embebidos, pero a su vez, posee un gran número de bibliotecas con funciones que lo convierten en un lenguaje para grandes proyectos. También, lenguaje C tiene un gran número de Tipos de datos, operadores y creación de nuevos Tipos, lo que lo hace un lenguaje de gran poder.
Es rápido en su ejecución y se usa como un referente, para probar la rapidez de otros lenguajes.
Permite que el programador pueda hacer muchas cosas, que otros lenguajes no lo permiten. No necesariamente, el lenguaje C puede detectar errores, ya sea en tiempo de compilación o durante la ejecución. Es por esto, quién programe usando el lenguaje C, debe comprender muy bien su funcionamiento.

Hoy día, prácticamente toda arquitectura de hardware computacional puede ejecutar programas escritos en lenguaje C, lo que lo hace un lenguaje sumamente portable. Muchos lenguajes de alto nivel están basados en lenguaje C por su rapidez y portabilidad. Algunos ejemplos son: C++ que además es totalmente compatible con código C. Objective-C de Apple, Swift de Apple mayormente basado en C++ y por lo tanto hereda muchas características del lenguaje C. Incluso el run-time de Swift está escrito en Objective-C que es un derivado directo del lenguaje C. Java está basado en C++. C# un lenguaje que deriva de C++ y Java. Perl que contiene muchas características de C. Python está escrito en C y su versión fundamental se llama CPython. Son muchos los lenguajes basados en las características del lenguaje C, lo que nos ayuda a comprender por qué es tan necesario aprender a programar usando lenguaje C.

1.2 Elementos esenciales del Lenguaje C

El lenguaje C hace uso de cinco conceptos primarios para la manipulación de información o datos. Estos son **Tipos**, **Objetos**, **Variables**, **Valores** y **Direcciones**.

1.2.1 Tipos

Cuando un profesional usa un sensor para medir la temperatura ambiente, obtiene un valor numérico que, al guardarlo en su computador, éste valor se almacena en la memoria del computador como una secuencia de ceros y unos (0 y 1). Un **Tipo** permite interpretar, al ser leída la memoria que contiene estos ceros y unos, como un valor decimal, Ejemplo: 25.4

Un **Tipo**, en lenguaje C, permite definir cuanto espacio se requiere en memoria y de qué forma se interpreta la información guardada en la memoria. También establece el rango en el cual la información pertenece a dicho Tipo y cuáles son los operadores que pueden manipular dicha información.

Ejemplo: el **Tipo int** en lenguaje C, establece que la información que pertenece a este **Tipo** son valores numéricos enteros que están en el rango – 65535 a + 65536 y los operadores que pueden actuar sobre estos tipos son + (suma), - (resta), * (multiplicación), / (división) etc.

Lenguaje C, posee Tipos directos (“built-in”) sólo para números enteros y decimales, como también para caracteres. Tipos más complejos para arreglos, estructuras u otros no posee, pero permite construirlos.

1.2.2 Objetos

Un **Objeto** es un sector de la memoria física del computador, que permite almacenar una información de cierto Tipo.

1.2.3 Variables

Una **Variable** es el nombre que se da a un objeto y representa a la información que se guarda en el objeto.

1.2.4 Valores

Valores son los ceros y unos que se guardan en el objeto, pero que se interpretan de acuerdo con el tipo del objeto.

1.2.5 Direcciones

Una **Dirección** en lenguaje C, es la dirección del comienzo del sector de la memoria u **Objeto**, donde se guarda un valor. La Dirección de un Objeto es un número entero sin signo, que conlleva la información del Tipo del Objeto. En este sentido, no se puede considerar sólo como un número entero.

Lenguaje C hace uso o de la variable o de la dirección del objeto, para leer o escribir un valor en el objeto. Cuando usa la dirección del objeto, está trabajando con punteros. Ambas formas, variables y punteros, se usan extensivamente en los próximos capítulos.

1.3 Esquema general de un programa en Lenguaje C

El esquema básico de un programa en lenguaje C, se muestra a continuación:

```
#include        <stdio.h>


int     main(void)
{
        // programa escrito por el usuario

        return 0;
}
```

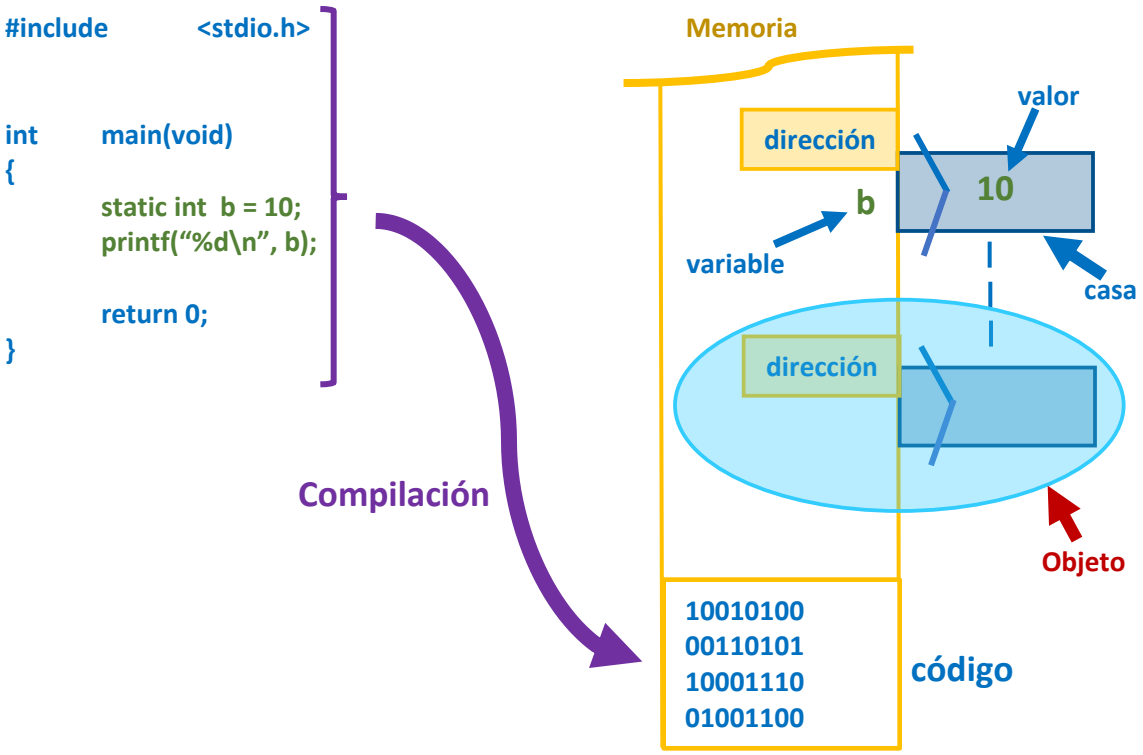
La primera sentencia **#include**, se utiliza para incluir bibliotecas que proporcionan funciones necesarias para el programa. Lenguaje C, posee una extensa cantidad de distintas bibliotecas que enriquece y empodera su programación. Las funciones más típicas son aquellas que permiten imprimir en la consola o leer caracteres desde teclado.
La segunda parte del programa es una función llamada **main**, que es propia del lenguaje C, y es la parte dónde se inicia la ejecución de un programa. En lenguaje C, una función, en general, retorna un Tipo que en este caso es un número 0, y que significa que el programa se ejecutó exitosamente.
Un programa o proyecto en lenguaje C puede estar compuesto de múltiples archivos, que contienen encabezamientos, funciones de bibliotecas o escritas por el usuario, macros etc., pero sólo un archivo puede contener **una** función **main**, que es dónde comienza a ejecutarse el programa. El nombre **main** y varios otros son parte del lenguaje C y el usuario no puede ocuparlos para otros propósitos.

1.4 Compilación

Para escribir un programa en lenguaje C, en general, se utiliza una IDE (Entorno de Desarrollo Integrado) y un compilador. Una IDE permite escribir texto, que son sentencias del lenguaje C, para ser luego compiladas y ejecutadas. Algunas IDE permiten elegir el compilador y otras son un paquete que integran también el compilador. En este libro, todos los problemas que se incluyen resueltos fueron realizados con IDE's opensource.
El proceso de compilación comprende varias etapas (preprocesamiento, compilación, enlazamiento), que se detallan en el capítulo 7, pero que comprende dos acciones principales:

1. Traduce el programa de texto a lenguaje de máquina, que puede ejecutar el computador.
2. Reserva memoria para todos los objetos que es posible hacerlo, antes de la ejecución (“run-time”) del programa. Otros objetos son creados en tiempo de ejecución.

En este libro, se usa una analogía para la memoria del computador que reúne todas las características de una memoria real, pero fácil de comprender por el lector que se inicia en la programación. Se asume que la memoria del computador es una larga calle con casas de distinto tamaño, dónde cada casa es un objeto con una dirección y que puede guardar un valor. La variable, se asume que vive en una casa y representa al valor guardado en dicha casa.
En el siguiente esquema se muestra el proceso básico de una compilación:



En el esquema anterior se muestra un sector de la memoria, que se denomina segmento de texto o segmento de código o simplemente código, dónde se deposita todo el código de máquina traducido por el compilador. El resto de la memoria se muestra como la analogía asumida.

NÚMEROS, DATOS y TIPOS.

Qué clase de hombre sería, si no tratara de mejorar el lugar en el que vivo.
Autor desconocido (edad media)

2.1 Números

El lenguaje C utiliza como valores numéricos los números enteros y decimales. Los números enteros los clasifica de acuerdo con su tamaño y si son positivos, negativos o ambos. Los números decimales comprenden los números racionales e irracionales y se clasifican conforme a su tamaño, y precisión por el número de dígitos decimales que ocupan.

Los rangos de los números enteros positivos van desde [0, 255] hasta [0, +18446744073709551615] y los rangos de los números enteros con signo van desde [-127, +127] hasta [-9223372036854775807, +9223372036854775807] en el estándar C99 del lenguaje C.
El lenguaje C ocupa rangos intermedios para los números enteros, entre estos mínimos y máximos.

Los números decimales se dividen en tres rangos, [1.2E-38, 3.4E+38] con 6 dígitos decimales de precisión, [2.3E-308, 1.7E+308] con 15 dígitos decimales de precisión y [3.4E-4932, 1.1E+4932] con 19 dígitos decimales de precisión. Un número decimal se representa por una parte entera y una parte decimal separada por un punto. Ejemplo: 23.4567777 con 7 dígitos decimales de precisión.

La clasificación en rangos es importante, porque estos definen el tamaño de memoria a utilizar por los valores numéricos.

2.2 Datos

¿Qué es un dato en programación? Revisemos la actividad que realiza un médico al visitar sus pacientes en un hospital. El médico se dirige a una cama y toma la bitácora con anotaciones. Lee:

Nombre	**Jorge Haft**
Edad	55
Peso	78 Kg
Temperatura	37.8 ºC
Glicemia	110
Colesterol	230

Jorge Haft son caracteres que representan el nombre del paciente. 55 es un número entero que representa la edad del paciente. 37.8 es un número decimal que representa la temperatura actual del paciente. Todos estos valores (caracteres o números) entregan un significado o información al médico, de tal forma que él puede usar e interpretar estos datos. Entonces, entenderemos que un dato es un valor que proporciona información. Por ahora, asumiremos que **valor** es un carácter, un número entero o

decimal. Ciertamente, los textos Edad, Peso, Temperatura, etc. son los que permiten obtener información de los valores dados.

2.3 Tipos.

El lenguaje C proporciona el concepto de **Tipo** que es un conjunto de valores posibles y un conjunto de operaciones para objetos de este tipo.
Por **Objeto** se entiende una parte o sector de la memoria que guarda un valor de un Tipo dado. Un Tipo define el espacio de memoria para un objeto.

El lenguaje C proporciona varios Tipos para valores numéricos y caracteres, de acuerdo con su rango y espacio de memoria a ocupar. En general, se acostumbra a hablar de Tipos de Datos, puesto que en programación los valores conllevan un significado o información.
Los tipos en lenguaje C se pueden clasificar en Tipos Básicos, Tipos de Enumeración, Tipo Vacío y Tipos Derivados. **En lenguaje C, los nombres de un Tipo cualquiera, se escriben sólo con minúsculas.**

Tipos Básicos

2.3.1 Tipos Enteros

La siguiente tabla proporciona los tipos para valores enteros.

Tipo	Tamaño de memoria	Rango de valores
char	1 byte	[0, 255] o [-128, 127]
unsigned char	1 byte	[0,255]
signed char	1 byte	[-128,127]
int	2 o 4 bytes	[-32768,32767] o [-2147483648,2147483647]
unsigned int	2 o 4 bytes	[0,65535] o [4294967295]
short int	2 bytes	[-32768, 32767]
unsigned short int	2 bytes	[0, 65535]
long int	4 bytes	[-2147483648, 2147483647]
unsigned long int	4 bytes	[0, 4294967295]
long long int	8 bytes	[-9223372036854775807, 9223372036854775807]
unsigned long long int	8 bytes	[0, 18446744073709551615]

El color azul de int indica que es optativo usarlo en esos casos. Byte es el almacenamiento o espacio mínimo en memoria que se ve en otro capítulo.

2.3.2 Tipos Flotantes o Decimales

La siguiente tabla proporciona los tipos para valores flotantes o decimales.

Tipo	Tamaño de memoria	Rango de Valores	Precisión
float	4 bytes	[1.2E-38, 3.4E+38]	6 dígitos decimales
double	8 bytes	[2.3E-308, 1.7E+308]	15 dígitos decimales
long double	10 bytes	[3.4E-4932, 1.1E+4932]	19 dígitos decimales

2.3.3 Tipo char

El tipo char ocupa 1 byte de espacio de memoria y guarda siempre un número entero. El número entero del objeto del tipo char representa un código para un determinado carácter. Los códigos más utilizados en programación son los códigos ASCII y EBCDIC. En este libro usaremos siempre el código ASCII que se muestra a continuación:

Símbolos ASCII

código ascii	0	**NULL**	(Carácter Nulo)
código ascii	1	**SOH**	(Comienzo de Encabezamiento)
código ascii	2	**STX**	(Comienzo de Texto)
código ascii	3	**ETX**	(Fin de Texto)
código ascii	4	**EOT**	(Fin de Transmisión)
código ascii	5	**ENQ**	(Averiguar)
código ascii	6	**ACK**	(Reconocimiento)
código ascii	7	**BEL**	(Sonido Timbre)
código ascii	8	**BS**	(Espacio hacia atrás)
código ascii	9	**HT**	(Tab Horizontal)
código ascii	10	**LF**	(Alimentación de Línea)
código ascii	11	**VT**	(Tab Vertical)
código ascii	12	**FF**	(Alimentación de Forma)
código ascii	13	**CR**	(Retorno de Carro)
código ascii	14	**SO**	(Desplazamiento hacia afuera)
código ascii	15	**SI**	(Desplazamiento hacia adentro)
código ascii	16	**DLE**	(Enlace de escape de datos)
código ascii	17	**DC1**	(Control de dispositivo 1)
código ascii	18	**DC2**	(Control de dispositivo 2)
código ascii	19	**DC3**	(Control de dispositivo 3)
código ascii	20	**DC4**	(Control de dispositivo 4)
código ascii	21	**NAK**	(Reconocimiento negativo)
código ascii	22	**SYN**	(Espera sincrónica)
código ascii	23	**ETB**	(Fin de transmisión de bloque)
código ascii	24	**CAN**	(Cancelar)
código ascii	25	**EM**	(Fin de medio)
código ascii	26	**SUB**	(Sustituto)
código ascii	27	**ESC**	(Escape)
código ascii	28	**FS**	(Separador de archivo)
código ascii	29	**GS**	(Separador de grupo)
código ascii	30	**RS**	(Separador de grabación)
código ascii	31	**US**	(Separador de Unidad)
código ascii	32		(espacio)
código ascii	33	**!**	(marca de exclamación)
código ascii	34	**"**	(marca de mención)
código ascii	35	#	(signo de número)
código ascii	36	$	(signo peso)
código ascii	37	%	(signo porcentaje)
código ascii	38	&	(símbolo)
código ascii	39	'	(apóstrofe)
código ascii	40	**(**	(paréntesis redondo)
código ascii	41	**)**	(paréntesis redondo)
código ascii	42	*****	(asterisco)
código ascii	43	**+**	(signo más)

código ascii	44	**,**	(coma)
código ascii	45	**-**	(guion)
código ascii	46	**.**	(punto)
código ascii	47	**/**	(barra oblicua)
código ascii	48	**0**	(cero)
código ascii	49	**1**	(uno)
código ascii	50	**2**	(dos)
código ascii	51	**3**	(tres)
código ascii	52	**4**	(cuatro)
código ascii	53	**5**	(cinco)
código ascii	54	**6**	(seis)
código ascii	55	**7**	(siete)
código ascii	56	**8**	(ocho)
código ascii	57	**9**	(nueve)
código ascii	58	**:**	(dos puntos)
código ascii	59	**;**	(punto y coma)
código ascii	60	**<**	(signo menor que)
código ascii	61	**=**	(signo igual)
código ascii	62	**>**	(signo mayor que)
código ascii	63	**?**	(signo de interrogación)
código ascii	64	@	(arroba)
código ascii	65	**A**	(mayúscula A)
código ascii	66	**B**	(mayúscula B)
código ascii	67	**C**	(mayúscula C)
código ascii	68	**D**	(mayúscula D)
código ascii	69	**E**	(mayúscula E)
código ascii	70	**F**	(mayúscula F)
código ascii	71	**G**	(mayúscula G)
código ascii	72	**H**	(mayúscula H)
código ascii	73	**I**	(mayúscula I)
código ascii	74	**J**	(mayúscula J)
código ascii	75	**K**	(mayúscula K)
código ascii	76	**L**	(mayúscula L)
código ascii	77	**M**	(mayúscula M)
código ascii	78	**N**	(mayúscula N)
código ascii	79	**O**	(mayúscula O)
código ascii	80	**P**	(mayúscula P)
código ascii	81	**Q**	(mayúscula Q)
código ascii	82	**R**	(mayúscula R)
código ascii	83	**S**	(mayúscula S)
código ascii	84	**T**	(mayúscula T)
código ascii	85	**U**	(mayúscula U)
código ascii	86	**V**	(mayúscula V)
código ascii	87	**W**	(mayúscula W)
código ascii	88	**X**	(mayúscula X)
código ascii	89	**Y**	(mayúscula Y)
código ascii	90	**Z**	(mayúscula Z)
código ascii	91	**[**	(paréntesis cuadrado)
código ascii	92	****	(barra inversa)
código ascii	93	**]**	(paréntesis cuadrado)
código ascii	94	**^**	(intercalación)
código ascii	95	**_**	(barra baja)
código ascii	96	**`**	(acento grave)
código ascii	97	**a**	(minúscula a)
código ascii	98	**b**	(minúscula b)
código ascii	99	**c**	(minúscula c)

código ascii	100	**d**	(minúscula d)
código ascii	101	**e**	(minúscula e)
código ascii	102	**f**	(minúscula f)
código ascii	103	**g**	(minúscula g)
código ascii	104	**h**	(minúscula h)
código ascii	105	**i**	(minúscula i)
código ascii	106	**j**	(minúscula j)
código ascii	107	**k**	(minúscula k)
código ascii	108	**l**	(minúscula l)
código ascii	109	**m**	(minúscula m)
código ascii	110	**n**	(minúscula n)
código ascii	111	**o**	(minúscula o)
código ascii	112	**p**	(minúscula p)
código ascii	113	**q**	(minúscula q)
código ascii	114	**r**	(minúscula r)
código ascii	115	**s**	(minúscula s)
código ascii	116	**t**	(minúscula t)
código ascii	117	**u**	(minúscula u)
código ascii	118	**v**	(minúscula v)
código ascii	119	**w**	(minúscula w)
código ascii	120	**x**	(minúscula x)
código ascii	121	**y**	(minúscula y)
código ascii	122	**z**	(minúscula z)
código ascii	123	**{**	(paréntesis curvo)
código ascii	124	**|**	(barra vertical)
código ascii	125	**}**	(paréntesis curvo)
código ascii	126	**~**	(tilde curva)
código ascii	127	**DEL**	(borrar)

código ascii	128	**Ç**	(mayúscula C-cedilla)
código ascii	129	**ü**	(letra "u" con diéresis)
código ascii	130	**é**	(letra "e" con acento agudo)
código ascii	131	**â**	(letra "a" con acento circunflejo)
código ascii	132	**ä**	(letra "a" con diéresis)
código ascii	133	**à**	(letra "a" con acento grave)
código ascii	134	**å**	(letra "a" con un anillo)
código ascii	135	**ç**	(minúscula c-cedilla)
código ascii	136	**ê**	(letra "e" con acento circunflejo)
código ascii	137	**ë**	(letra "e" con diéresis)
código ascii	138	**è**	(letra "e" con acento grave)
código ascii	139	**ï**	(letra "i" con diéresis)
código ascii	140	**î**	(letra "i" con acento circunflejo)
código ascii	141	**ì**	(letra "i" con acento grave)
código ascii	142	**Ä**	(letra "A" con diéresis)
código ascii	143	**Å**	(letra "A" con un anillo)
código ascii	144	**É**	(mayúscula con acento agudo)
código ascii	145	**æ**	(latín "ae")
código ascii	146	**Æ**	(latín "AE")
código ascii	147	**ô**	(letra "o" con acento circunflejo)
código ascii	148	**ö**	(letra "o" con diéresis)
código ascii	149	**ò**	(letra "o" con acento grave)
código ascii	150	**û**	(letra "u" con acento circunflejo)
código ascii	151	**ù**	(letra "u" con acento grave)
código ascii	152	**ÿ**	(letra "y" con diéresis)
código ascii	153	**Ö**	(letra "O" con diéresis)
código ascii	154	**Ü**	(letra "U" con diéresis)

código ascii	155	ø	(cero con barra o conjunto vacío)
código ascii	156	£	(signo de libra)
código ascii	157	Ø	(cero con barra o conjunto vacío)
código ascii	158	×	(signo de multiplicación)
código ascii	159	ƒ	(signo de función)
código ascii	160	á	(letra "a" con acento agudo)
código ascii	161	í	(letra "i" con acento agudo)
código ascii	162	ó	(letra "o" con acento agudo)
código ascii	163	ú	(letra "u" acento agudo)
código ascii	164	ñ	(letra "n" con tilde)
código ascii	165	Ñ	(letra "N" con tilde)
código ascii	166	ª	(indicador ordinal femenino)
código ascii	167	º	(indicador ordinal masculino)
código ascii	168	¿	(signo pregunta)
código ascii	169	®	(marca registrada)
código ascii	170	¬	(signo negación lógica)
código ascii	171	½	(mitad)
código ascii	172	¼	(un cuarto)
código ascii	173	¡	(exclamación invertida)
código ascii	174	«	(cremillas en ángulo)
código ascii	175	»	(cremillas en ángulo)
código ascii	176	▒	
código ascii	177	▒	
código ascii	178	▒	
código ascii	179	│	(carácter de dibujo)
código ascii	180	┤	(carácter de dibujo)
código ascii	181	Á	(mayúscula con acento agudo)
código ascii	182	Â	(letra con acento circunflejo)
código ascii	183	À	(letra con acento grave)
código ascii	184	©	(signo de propiedad)
código ascii	185	╣	(carácter de dibujo)
código ascii	186	║	(carácter de dibujo)
código ascii	187	╗	(carácter de dibujo)
código ascii	188	╝	(carácter de dibujo)
código ascii	189	¢	(símbolo cent)
código ascii	190	¥	(signo yen)
código ascii	191	┐	(carácter de dibujo)
código ascii	192	└	(carácter de dibujo)
código ascii	193	┴	(carácter de dibujo)
código ascii	194	┬	(carácter de dibujo)
código ascii	195	├	(carácter de dibujo)
código ascii	196	─	(carácter de dibujo)
código ascii	197	┼	(carácter de dibujo)
código ascii	198	ã	(letra "a" con tilde)
código ascii	199	Ã	(letra "A" con tilde)
código ascii	200	╚	(carácter de dibujo)
código ascii	201	╔	(carácter de dibujo)
código ascii	202	╩	(carácter de dibujo)
código ascii	203	╦	(carácter de dibujo)
código ascii	204	╠	(carácter de dibujo)
código ascii	205	═	(carácter de dibujo)
código ascii	206	╬	(carácter de dibujo)
código ascii	207	¤	(signo monetario corriente)
código ascii	208	ð	(minúscula "eth")
código ascii	209	Ð	(mayúscula letra "Eth")
código ascii	210	Ê	(letra "E" con acento circunflejo)

código ascii	211	Ë	(letra "E" con diéresis)
código ascii	212	È	(letra "E" con acento grave)
código ascii	213	ı	(minúscula i)
código ascii	214	Í	(mayúscula letra "I" con acento agudo)
código ascii	215	Î	(letra "I" con acento circunflejo)
código ascii	216	Ï	(letra "I" con diéresis)
código ascii	217	┘	(Carácter de dibujo)
código ascii	218	┌	(Carácter de dibujo)
código ascii	219		(Bloque)
código ascii	220		
código ascii	221	¦	(barra vertical quebrada)
código ascii	222	Ì	(letra "I" con acento grave)
código ascii	223		
código ascii	224	Ó	(letra mayúscula “O" con acento agudo)
código ascii	225	ß	(letra "aguda S")
código ascii	226	Ô	(letra "O" con acento circunflejo)
código ascii	227	Ò	(letra "O" con acento grave)
código ascii	228	õ	(letra "o" con tilde)
código ascii	229	Õ	(letra "O" con tilde)
código ascii	230	µ	(letra minúscula “mu”)
código ascii	231	þ	(mayúscula “espinoso")
código ascii	232	Þ	(minúscula "espinoso")
código ascii	233	Ú	(mayúscula “U" con acento agudo)
código ascii	234	Û	(letra "U" con acento circunflejo)
código ascii	235	Ù	(letra "U" con acento grave)
código ascii	236	ý	(letra "y" con acento agudo)
código ascii	237	Ý	(Mayúscula “Y" con acento agudo)
código ascii	238	¯	(signo macron)
código ascii	239	´	(acento agudo)
código ascii	240	¬	(Hipen)
código ascii	241	±	(signo más-menos)
código ascii	242	‗	(barra baja)
código ascii	243	¾	(tres cuartos)
código ascii	244	¶	(parágrafo)
código ascii	245	§	(signo sección)
código ascii	246	÷	(signo división)
código ascii	247	¸	(cedilla)
código ascii	248	°	(signo grado)
código ascii	249	¨	(diéresis)
código ascii	250	•	(inter-punto)
código ascii	251	1	(superíndice)
código ascii	252	3	(cubo)
código ascii	253	2	(cuadrado)
código ascii	254	■	(bloque cuadrado)
código ascii	255	nbsp	(espacio de no ruptura)

Ejemplo: La letra A se representa por el número entero 65, que ocupa 1 byte de memoria y corresponde al número binario 0100 0001.
Los tipos Enumeración, Vacío y Derivados se ven en los próximos capítulos.

2.4 Variables

Variable es un nombre o símbolo que se asigna a un objeto. Entonces, variable es un nombre que se da a una parte o sector de la memoria que guarda un valor de cierto tipo. Para crear una variable se debe declarar y/o inicializar, lo cual significa asignarle un valor.

Sólo un **Tipo** nos permite declarar(crear) una variable, puesto que el objeto que represente la variable debe ser de un cierto tipo. En adelante usaremos la palabra **sentencia** para indicar una acción que se realiza en tiempo de ejecución.

Ejemplo:
La siguiente sentencia declara una variable del tipo entero:

```
int     dat;    //esto es un comentario. El punto y coma es imperativo en lenguaje C para terminar
                //la sentencia.
```

Esta sentencia asigna el nombre dat a una parte de la memoria, aunque en esa parte de la memoria no se guarde ningún valor específico. En este sentido, la simple declaración de una variable otorga un nombre y reserva memoria para ese nombre. Para guardar un valor en esa parte de la memoria se escribe la siguiente sentencia:

```
dat = 55;
```

El valor entero 55 se guarda en la parte de la memoria con el nombre dat. Esto se llama **asignación** de valor a una variable.
También una variable se puede declarar y asignar un valor en una única sentencia, ejemplo:

```
int dato = 23;
```

En este caso, la variable dato es **declarada e inicializada**. La palabra **inicialización** se usa cuando se da un valor a una variable al momento de crearla.
El uso del tipo int obliga a que el valor que se guarde en la parte de la memoria con el nombre dato sea un valor entero, en el rango que tiene un tipo int. Cualquier otro valor que no sea del tipo int, el compilador lo reconoce como un error en tiempo de compilación.
El ejemplo anterior es válido para cualquier tipo de los indicados en las tablas en 2.3.1 y 2.3.2.

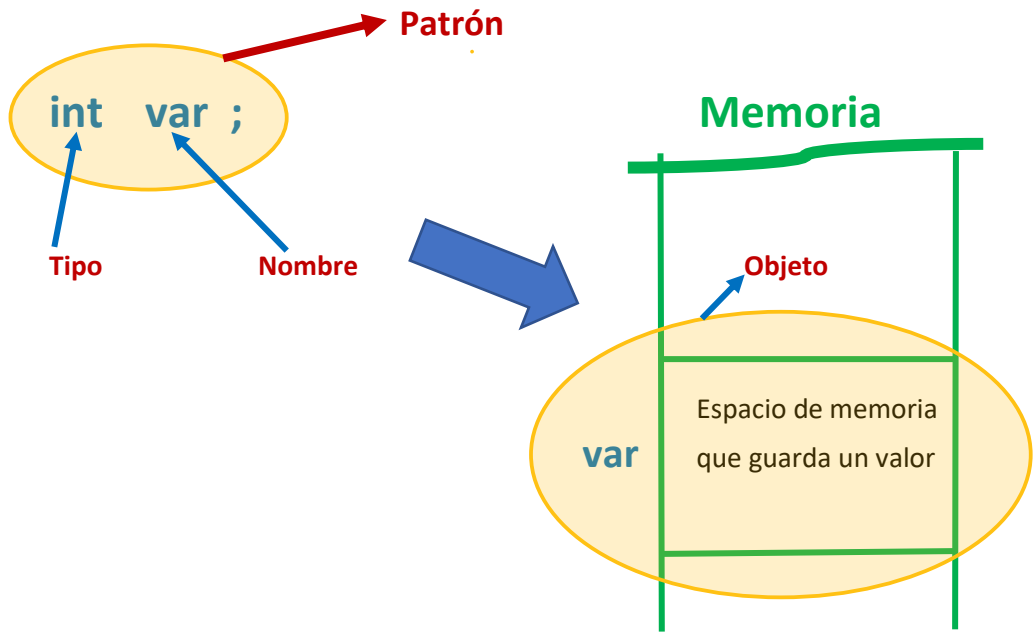
¿Qué sucede con el valor que tiene una variable declarada, pero no inicializada? La parte de la memoria que se reserva para el nombre declarado puede tener cualquier valor que existía antes de usar ese lugar de la memoria. Hay declaraciones de variables como las variables estáticas, las cuales, al no ser inicializadas, colocan un cero en la memoria. Cualquier parte de una memoria solo contiene ceros y unos, por lo cual el valor dependerá del tipo con se lea esa porción de memoria.
Una excepción a la reserva de memoria al declarar una variable lo entrega la siguiente sentencia:

```
extern int temp;
```

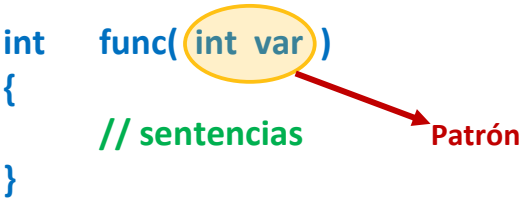
Esta sentencia declara una variable externa temp del tipo entero. Solo reserva el nombre temp para tipo entero, pero no reserva memoria. La memoria se reserva en otra parte del programa cuando se define la variable temp, es decir, cuando se asigna un valor a ésta.

2.4.1 Patrones en C

Cada vez que nos encontremos con un **Tipo y un nombre** a su derecha, lo podemos considerar un patrón cuyo significado es la creación de una variable de un objeto con ese nombre, para guardar un valor de ese tipo. Es importante ver dónde está el patrón, porque de ello dependerá el tiempo de existencia de la variable. Un **Tipo y un nombre en el programa main()** crea una variable en memoria que existe por todo el programa. Un **Tipo y un nombre en el cuerpo de una función** crea una variable en memoria que existe sólo el tiempo de ejecución de la función.
Ejemplos:



En este caso, el patrón int var reserva un espacio de memoria con el nombre var para guardar un valor del tipo entero. El tiempo de existencia del espacio de memoria, dependerá de dónde está ubicado el patrón int var;



Aquí, el patrón int var está ubicado dentro del paréntesis de la función func como parámetro de dicha función. Se creará un espacio de memoria con el nombre var para guardar un valor entero. Sin embargo, este espacio de memoria u objeto existirá solo por el tiempo que emplee la función en su ejecución. Una vez terminada la función, el espacio de memoria con el nombre var desaparece. Es decir, el sistema operativo puede hacer uso de este espacio para otra tarea.

2.5 Operadores

El conjunto de operadores aritméticos para enteros y decimales que pueden operar en objetos de estos tipos son los siguientes:

+ - * / % ++ --

El primero es el símbolo de adición, el segundo el símbolo de substracción, el tercero multiplicación, el cuarto división y quinto es el símbolo de módulo. Estos operadores pueden ser usados con variables de los tipos enteros o flotantes.
El operador ++ suma un 1 a la variable y el operador – resta un 1 a la variable. Ambos operadores se pueden usar con números enteros o flotantes, pero en general, se usan siempre con números enteros, lo cual se considera un buen estilo de programación.
Ejemplos:

```
int dat = 34;        // declara e inicializa la variable dat con un valor 34

int alfa = 40;       // declara e inicializa la variable alfa con un valor 40

int fat = dat + alfa;
```

La última sentencia declara la variable fat y la inicializa con la suma de las variables dat y alfa, esto es con el valor numérico entero 74.

El símbolo % es el módulo que entrega el resto de la división de dos números. Ejemplo: 5%3 es 2; 7%2 es 1, etc.

```
++dat   //cambia el valor de dat a 35
--dat   //cambia el valor de dat a 33
```

2.6 Función de la biblioteca estándar printf()

printf() lo utilizaremos muy a menudo para imprimir datos u objetos en la pantalla del computador. Es una función proporcionada por la biblioteca estándar del lenguaje C que se incluye en el programa al agregar el encabezamiento #include <stdio.h> antes del main().

Ejemplo:

```
int dat = 23;
printf("%d\n", dat);    // imprime el número 23 en la pantalla
```

"%d\n" es el formato con que se imprime la variable dat.
d es un formato para el tipo int.

\n implica un salto de línea al término de la impresión.
dat es el nombre de la variable a imprimir.

Hay varios formatos para los diferentes tipos básicos que se muestran en la siguiente tabla:

Formato	Acción del formato
c	Imprime un carácter
d, i	Imprime un número entero
f	Imprime un número flotante o doble
p	Imprime un puntero
e	Imprime un número flotante en notación científica
s	Imprime una secuencia de caracteres o string

Ejemplos:

```
int var = 45;
float temp = 34.54567;
double jak = 5.067857467;
char dato = 'H';   // Se debe encerrar la letra H en los símbolos ' ' para que se guarde el código ASCII en la variable dato.

printf("%f\n", temp);      // imprime la variable temp
printf("%c\n", dato);      // imprime la letra H
printf("%c%.2f%f%d\n",dato, jak,temp,var);  // imprime en una línea   H  5.06  34.54567  45
```

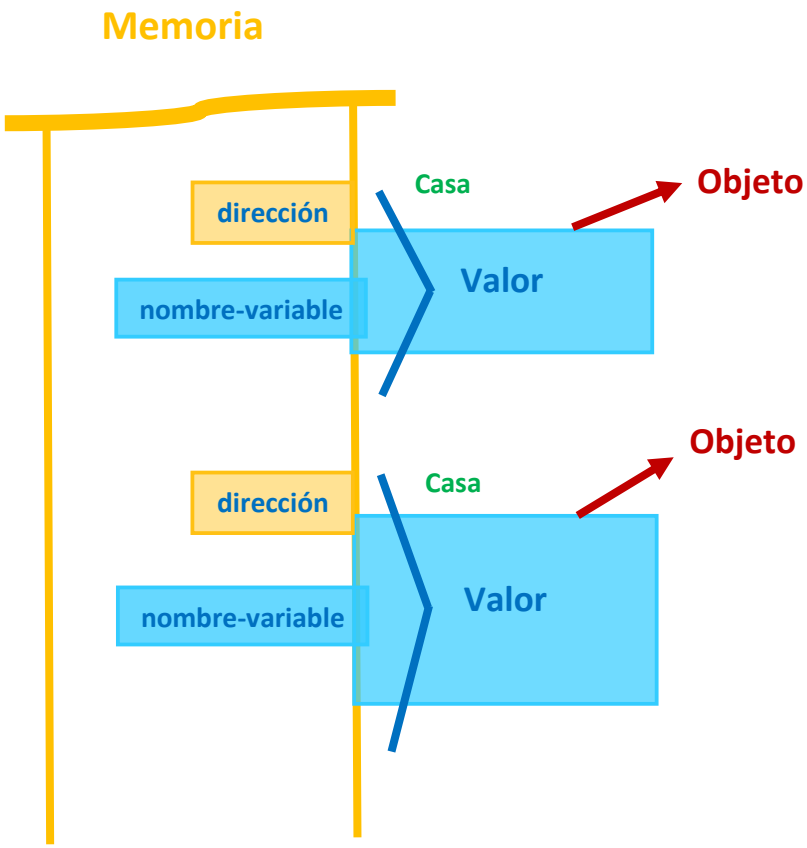
imprime jak sólo con dos decimales porque al formato f se antepone un .2, que fuerza a considerar 2 dígitos decimales para la impresión.

Usar un .digito antes de un formato f, permite imprimir con un número determinado de dígitos decimales.

2.7 Memoria

En la memoria del computador se almacenan tanto el código como los datos de un programa. En este capítulo usaremos una analogía para la memoria que consiste en considerar la memoria como una calle extensamente larga, con casas a un lado de la calle, y que tiene casas con tamaño de 1 byte, 4 bytes y 8 bytes. Como en toda calle, cada casa tiene una **única** dirección. Las casas de 1 byte pueden almacenar o guardar datos del tipo char; las casas de 4 bytes de tamaño pueden almacenar tipos int, float y todo tipo que necesite 4 bytes y finalmente las casas de tamaño 8 bytes pueden almacenar datos del tipo double.

Al declarar una variable de un tipo, el nombre de la variable se asigna a una casa desocupada. Esto en realidad lo hace el sistema operativo en tiempo de ejecución. Una casa desocupada no implica que no tenga un dato que haya quedado ahí de antes, sino que simplemente no tiene variable(arrendatario) que la ocupe y el sistema operativo la tiene como disponible. **Cada casa es un objeto,** que puede guardar un valor de cierto Tipo.
En el siguiente esquema se muestra la analogía de la memoria que se usa en este libro:



Cuando se inicializa o posteriormente se asigna un valor a una variable, este valor se guarda en la casa donde vive la variable específica. En adelante, el valor guardado en la casa se obtiene con el nombre de la variable. Como toda casa tiene una dirección, también resulta posible usar esta dirección para obtener el valor guardado en la casa. Así, **un valor guardado en una casa se puede obtener de dos formas: usando el nombre de la variable o la dirección donde vive dicha variable.**
¿Como se conoce la dirección dónde vive una variable? Muy simple, el lenguaje C proporciona una forma para hacer esto colocando el símbolo **&** a la izquierda del nombre de la variable y así se obtiene la dirección de la casa donde vive la variable.

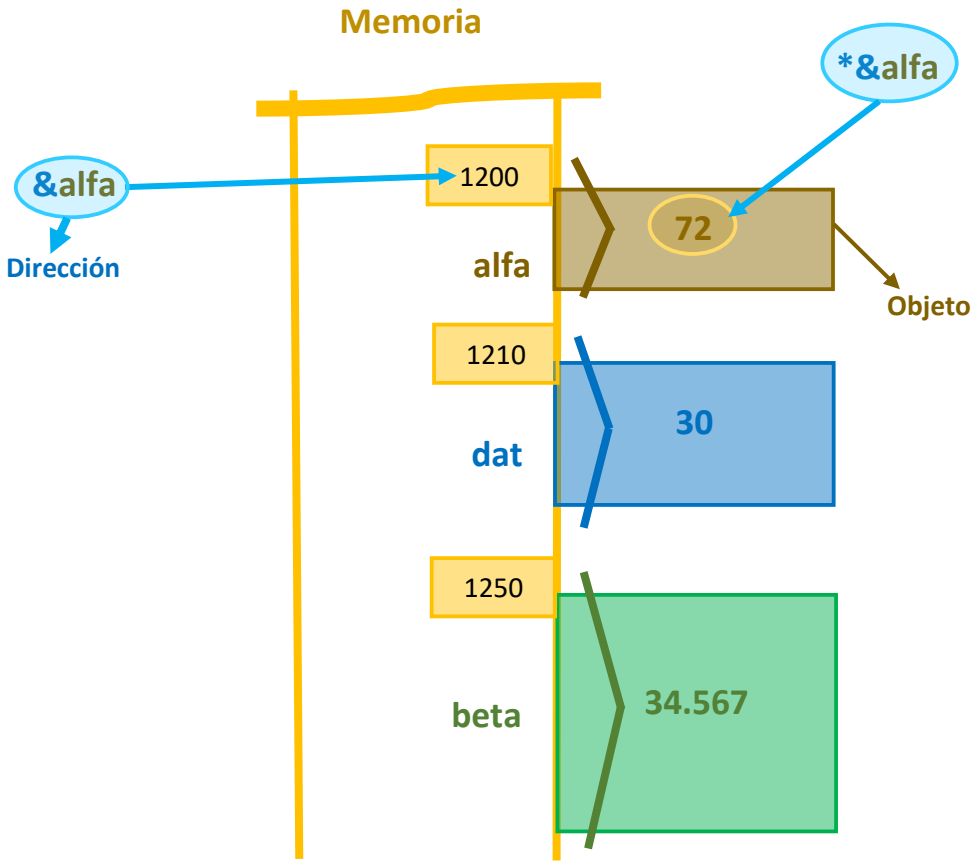
Ejemplo:
int dat = 30;
La **dirección** de dat es &dat.

Si se conoce la dirección de una casa dónde se guarda un valor de cierto tipo, ¿Es posible obtener el valor guardado usando la dirección? Sí, efectivamente el lenguaje C proporciona una forma de hacer esto y es colocando a la izquierda de la dirección el símbolo *, esto es *&dat es equivalente al valor 30 y por lo tanto *&dat es también una **variable u otro nombre del objeto** que representa al valor 30.

En la figura siguiente se muestra una calle con casas y su analogía con la memoria de un computador desde un punto de vista funcional.

int **dat** = 30;

```
char alfa = 'H';  // Código ASCII de H es 72
double beta = 34.567;
```



&alfa es igual al valor 1200.

***&alfa** es igual al valor 72.

alfa es igual a 72.
dat es igual a 30.
beta es igual a 34.567.

***&alfa** y **alfa** representan al mismo valor 72 y entonces, ambas son **variables** para el mismo objeto.
Objeto es la casa que guarda un valor, esto es una parte física de la memoria con un valor binario.

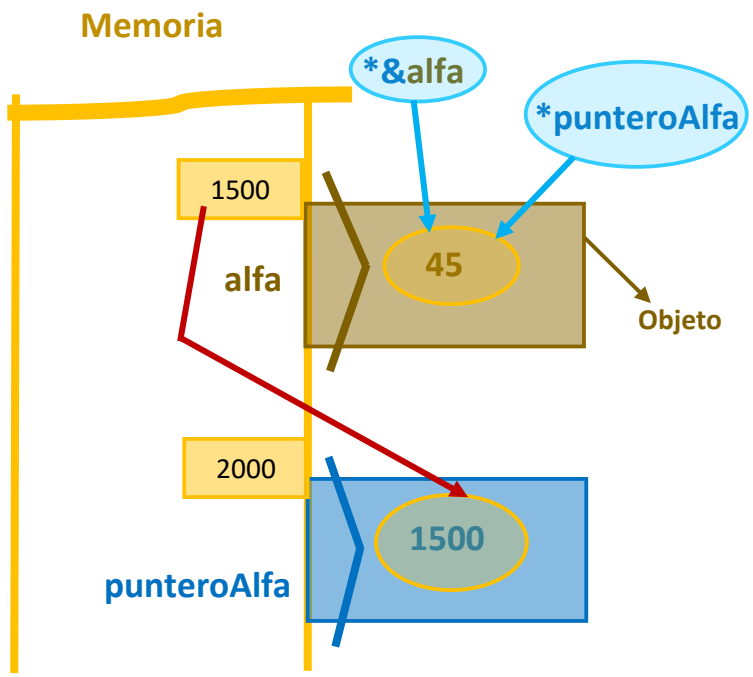
El valor 72 que se guarda en alfa es el código ASCII de la letra H. Al usar printf("%c\n", alfa), el formato c convierte el valor entero 72 en la letra H, de tal forma que se imprime en pantalla la letra H y no el valor 72. Para imprimir el valor 72 debe usarse en el printf() el formato **d** en vez de **c**.

Las casas de color café tienen el tamaño de 1 byte, las casas de color azul tienen el tamaño de 4 bytes y las casas de color verde tienen el tamaño de 8 bytes.

2.8 Tipo Puntero

El lenguaje C proporciona un tipo para variables que guarden en memoria una dirección. El tamaño de este tipo es de 4 bytes. Este tipo se denomina **Tipo Puntero** y permite declarar variables puntero para cada tipo básico, tanto de enteros como de decimales. Un tipo puntero se forma agregando un * a la derecha del tipo básico, ejemplo**: int*** es un tipo puntero para declarar variables puntero a un entero.
Ejemplo de uso del tipo puntero:

```
int alfa = 45;

int* punteroAlfa = &alfa;  // asigna la dirección de alfa a la variable punteroAlfa
```



alfa, *&alfa y *punteroAlfa son **variables** para
el mismo **objeto**.
La variable punteroAlfa guarda el valor 1500, que es la dirección de alfa. La variable punteroAlfa vive en otro lugar de la memoria en la dirección 2000 en una casa con tamaño 4 bytes. Como punteroAlfa es equivalente a la dirección de alfa, si se coloca un * a la izquierda de punteroAlfa se obtiene el valor entero 45, que se guarda en la dirección 1500.
Las variables punteros o simplemente punteros se ocupan para manipular direcciones de memoria y así evitar tener que usar el símbolo &. Las operaciones permitidas en los tipos punteros son la suma (+) y la substracción (-) solo con números enteros.
Ejemplos de punteros para distintos tipos básicos:

```
float*    tipo puntero para declarar variables puntero a flotante
char*     tipo puntero para declarar variables puntero a carácter
double*   tipo puntero para declarar variables puntero a doble, etc.
```

No existe un puntero universal, si se considera que una dirección siempre tiene el tamaño de 4 bytes. Cada puntero que guarda una dirección está estrictamente relacionado con el tipo de dato cuya dirección se guarda en el puntero. Esto es necesario que sea así, para realizar aritmética con punteros, que se ve en otro capítulo.
Un puntero además de guardar una dirección posee la información del tipo de dato al cual pertenece esa dirección, es decir conoce el tamaño del objeto al cual apunta.

Tres formas de imprimir el valor de una variable.
Ejemplo:

```
double gama = 2.8978;        //declara e inicializa la variable gama.
double* point = &gama;       //declara  la variable puntero point a double  y la inicializa con la dirección de
                             //gama.
```

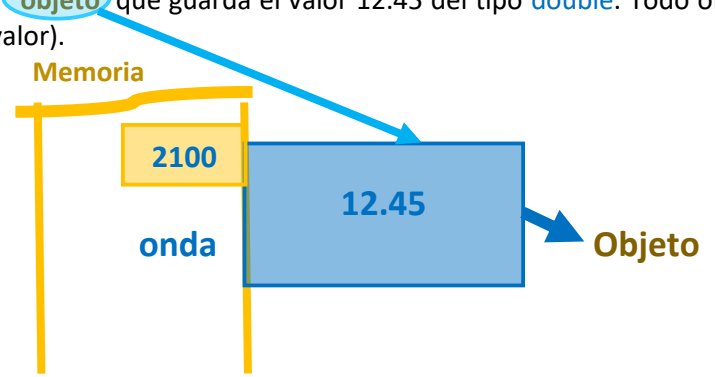
1º Forma: `printf("%lf\n", gama);` // imprime usando el nombre de la variable.
2º Forma: `printf("%lf\n", *&gama);` // imprime usando la dirección de la variable.
3º Forma: `printf("%lf\n", *point);` // imprime usando el puntero point que contiene la dirección //de la variable gama.

Tres formas de cambiar el valor de una variable.
Ejemplo:

```
double onda = 12.45;              //declara e inicializa la variable gama.
double* ondaPoint = &onda;        //declara  la variable puntero ondaPoint a double  y la inicializa con la
                                  //dirección de  gama.
```

1º Forma: `onda = 45.0;` // Cambia el valor de onda de 12.45 a 45.0..
2º Forma: `*&onda = 45.0;` //Cambia el valor de onda usando la dirección de la variable.
3º Forma: `*ondaPoint =45.0;` //Cambia el valor de onda usando el puntero ondaPoint.

Importante: *&onda y *ondaPoint se pueden usar tanto al lado izquierdo del signo **=** como a su lado derecho, porque ambos representan al objeto que guarda el valor 12.45 del tipo double. Todo objeto puede ser leído o escrito (modificar su valor).



El nombre del objeto, **onda,** representa al valor 12.45 que contiene el **objeto**.

onda, *&onda y *ondaPoint son **variables** que representan al mismo **objeto**.

2.9 Acciones de signo de asignación.

El signo de asignación = implica acciones tanto para lo que está a su lado derecho como para lo que está a su lado izquierdo. Se explica a continuación:

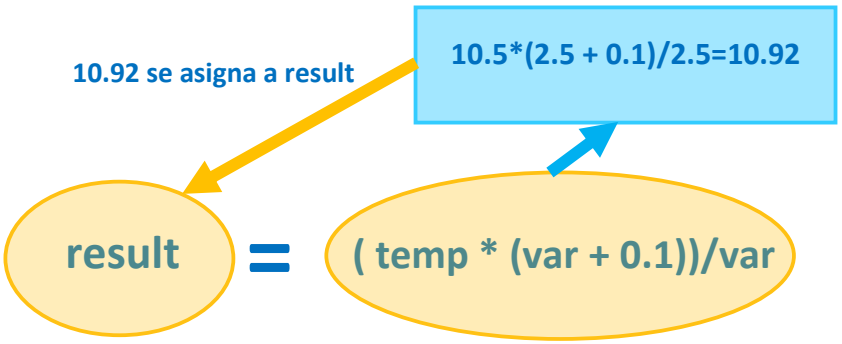
En este lado del signo = hay una sola variable que recibe y modifica su valor actual por el nuevo que se le asigna desde el lado derecho.
Lo que ocurre aquí es siempre una ESCRITURA.
Una escritura cambia el valor actual de la variable.

=

Lo que está a este lado del signo = puede ser una expresión muy compleja que involucre variables de distinto tipo, valores constantes, etc.; pero finalmente debe resolverse y convertirse en un solo objeto del mismo tipo de la variable que está al lado izquierdo del signo =.
El valor de este objeto se asigna a la variable del lado izquierdo.
En este lado las variables se LEEN y por lo tanto sus valores originales no son modificados.

Ejemplo:

```
double temp = 10.5, var = 2.5;      // En una línea se declaran e inicializan dos variables dobles
double result = 0.0;
```



Luego:

```
printf("%lf\n",  result);    // imprime el valor 10.92
```

Una variable puede estar al lado izquierdo o derecho del signo = , porque representa al valor de un objeto que reside en una parte física de la memoria.

2.10 Identificadores.

Un identificador, en lenguaje C, hace referencia a los nombres que se dan a variables, funciones u otros objetos. Un identificador debe comenzar con una letra minúscula o mayúscula o el signo menos bajo (“*underscore”*) y puede tener dígitos entre las letras, después de la primera.

Ejemplos de identificadores:

tempVariación23
_unicoValorFinanciero
Battle45AndTwo etc.

Al crear un identificador se debe hacer un esfuerzo porque éste tenga un significado mnemónico en referencia al objeto que se asigna.
Ejemplo:
Se desea crear un identificador para la temperatura del paciente Juan:

```
temp          // es una opción

temperaturaJuan     // es una mejor opción porque identifica al paciente y el parámetro que se mide
```

2.11 Palabras Claves (*KeyWords*) en lenguaje C.

Lenguaje C tiene 32 palabras que tienen un significado preciso para el compilador y que no pueden ser usadas como identificadores en un programa de usuario.

Estas son:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Cada una de ellas irá teniendo significado en los siguientes capítulos de este libro.

2.12 Operadores en C.

Un operador es un símbolo que indica al compilador realizar una operación matemática o lógica específica. Lenguaje C proporciona una gran cantidad de operadores propios (“*built-in”*).
Estos operadores se clasifican en:

- **Operadores aritméticos**
- **Operadores relacionales**
- **Operadores lógicos**
- **Operadores a nivel de bit**
- **Operadores de asignación**

2.12.1 Operadores Aritméticos.

Están descritos en 2.5

2.12.2 Operadores Relacionales.

La siguiente tabla muestra los operadores relacionales y su función:

Operador	Descripción
==	Comprueba si los operandos a ambos lados del operador son iguales o no. Si son iguales se considera la condición como verdadera. Esto es, **operador == operador** es verdadero y asume el valor entero 1. Caso contrario se considera falsa y asume el valor 0.
!=	Comprueba si los operandos a ambos lados del operador son distintos. Si lo son se considera la condición como verdadera.
>	Comprueba que el operando de la izquierda es mayor que el operando de la derecha. Si es así la condición es verdadera.
<	Comprueba que el operando de la izquierda es menor que el operando de la derecha. Si es así la condición es verdadera.
>=	Comprueba que el operando de la izquierda es mayor o igual al operando de la derecha. Si es así la condición es verdadera.
<=	Comprueba que el operando de la izquierda es menor o igual al operando de la derecha. Si es así la condición es verdadera.

Operandos son variables o expresiones que se utilizan a ambos lados de un símbolo operador.
En lenguaje C, la condición verdadera se representa por un valor distinto de cero, pero en los casos de la tabla anterior que involucra un operador y operandos la condición verdadera se representa por un 1. La condición falsa por un cero.

Ejemplo:

```
int temp = 20, dat = 30;

printf(“%d\n”, temp < dat);   // imprime un 1 porque temp es menor que dat y por lo tanto la condición es verdadera. La expresión temp < dat asume el valor 1.
```

2.12.3 Operadores Lógicos.

La siguiente tabla muestra los operadores lógicos y su función:

Operador	Descripción
&&	Realiza una operación lógica AND entre el operador de la izquierda y el de la derecha. Si ambos operandos son distintos de cero, la condición es verdadera.
||	Realiza una operación lógica OR entre el operador de la izquierda y el de la derecha. Si un operando es distinto de cero, la condición es verdadera.
!	Llamado operador lógico NOT. Cambia una condición a su opuesto. Verdadera a falsa y viceversa.

Ejemplo:

```
int temp = 20, dat = 30;

printf("%d\n", temp && dat);    // imprime un 1.  La expresión  temp && dat  es verdadera

printf("%d\n", !(temp && dat));    // imprime un 0. Cambia de verdadero a falso
```

2.12.4 Operadores a nivel de bits (*BitWise*).

La siguiente tabla muestra los operadores a nivel de bits y su función:

Operador	Descripción
&	Operador binario AND
|	Operador binario OR
^	Operador binario XOR
~	Operador binario complemento uno
<<	Operador desplazamiento a la izquierda
>>	Operador desplazamiento a la derecha

Un número entero X en base 10 se representa como un número binario de acuerdo a la siguiente fórmula:

$$X_{10} = \sum_{i=0}^{n} a_i * 2^i$$

a_i sólo puede tomar los valores 0 ó 1.

Ejemplo:

$$8_{10} = 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$$

Luego, en binario el número entero 8 se representa por 1000. Los coeficientes a_i solo pueden tomar los valores 0 o 1. En el número binario 1000, la posición del dígito indica el peso 2^i qué éste tiene en el número. La posición 0 corresponde al dígito más a la derecha.

Un **byte** contiene 8 bits y el máximo número que puede representar cuando todos los bits son un 1 es 255.

Los operadores binarios &, | y ^ operan bit a bit de acuerdo a la siguiente tabla de verdad:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Ejemplos:

int p = 8, q = 6;

p&q ⟶

```
  1 0 0 0   8
  0 1 1 0   6
  ---------
  0 0 0 0   0
```

Usando la tabla de verdad bit a bit para p&q se obtiene un 0.

p|q ⟶

```
  1 0 0 0   8
  0 1 1 0   6
  ---------
  1 1 1 0   13
```

Usando la tabla de verdad bit a bit para p|q se obtiene un 13.

Realice p^q.

El operador ~ invierte los dígitos.
Ejemplo: p en binario es 1000 ~p es 0111 que corresponde al número 7.

El operador << desplaza los bits a la izquierda tantas veces como se indique.
Ejemplo:
q << 1 q se desplaza un bit a la izquierda y queda convertido en el número binario 1100 que corresponde al entero 12. El desplazamiento a la izquierda es equivalente a una multiplicación, en este caso por 2. Desplazamiento de dos bits es equivalente a una multiplicación por cuatro, etc. El número a la derecha

del signo << indica cuantas veces ocurre el desplazamiento de bits. Debe ser un entero positivo, aunque **el compilador no detecta un error si se usa un entero negativo**, y en este caso, el resultado del desplazamiento es impredecible y depende de la implementación del compilador.

El operador >> desplaza los bits a la derecha tantas veces como se indique.
Ejemplo:
q >> 1 q se desplaza un bit a la derecha y queda convertido en el número binario 0011 que corresponde al entero 3. El desplazamiento a la derecha es equivalente a una división, en este caso por 2.

2.12.5 Operadores de Asignación.

La siguiente tabla muestra los operadores de asignación y su función:

Operador	Descripción
=	Asigna el operando de la derecha al operando o variable de la izquierda
+=	C += B es equivalente a C = C+B
-=	C -= B es equivalente a C = C-B
*=	C *= B es equivalente a C = C*B
/=	C /= B es equivalente a C = C/B
%=	C %= B es equivalente a C = C%B
<<=	C <<= 2 es equivalente a C = C << 2
>>=	C >>= 2 es equivalente a C = C >> 2
&=	C &= B es equivalente a C = C & B
^=	C ^= B es equivalente a C = C ^ B
|=	C |= B es equivalente a C = C | B

2.12.6 Precedencia de los operadores en lenguaje C.

La evaluación de los operadores en una expresión tiene un orden de precedencia. Por ejemplo, en la siguiente expresión:
A*b + c*f primero se evalúan los productos y después la suma. La multiplicación tiene precedencia sobre la suma.

A continuación, se muestra una tabla con las principales precedencias en lenguaje C:

Categoría	Operador	Asociatividad
Sufijo	() [] -> . ++ - -	Izquierda a derecha
Unitario	+ - ! ~ ++ - - (type)* & sizeof	Derecha a izquierda
Multiplicativo	* / %	Izquierda a derecha
Aditivo	+ -	Izquierda a derecha
Shift Desplazamiento	<< >>	Izquierda a derecha

Relacional	< <= > >=	Izquierda a derecha
Igualdad	== !=	Izquierda a derecha
AND a nivel de bit	&	Izquierda a derecha
XOR a nivel de bit	^	Izquierda a derecha
OR a nivel de bit	|	Izquierda a derecha
AND Lógico	&&	Izquierda a derecha
OR Lógico	||	Izquierda a derecha
Condicional	?:	Derecha a izquierda
Asignación	= += -= *= /= %=>>= <<= &= ^= |=	Derecha a izquierda
Coma	,	Izquierda a derecha

En C, la precedencia de operadores aritméticos (*, %, /, +, -) es mayor que operadores relacionales (==, !=, >, <, >=, <=) y la precedencia de operadores relacionales es mayor que operadores lógicos(&&, || and !).

En la tabla anterior, precedencia está en el orden de las filas de la tabla (1º fila mayor precedencia y así sucesivamente) y para una misma precedencia rige entonces la asociatividad de izquierda a derecha o derecha a izquierda, según corresponda.

Ejemplo:
(3 > 4 + 7 && 9)
Esta expresión es equivalente a:
((3 > (4+7)) && 9) esto es, (4+7) se ejecuta primero con resultado 11, entonces se ejecuta la primera parte de la expresión (3 > 11) lo que resulta falso, esto es 0.
Luego, se ejecuta 0 && 9 lo que es falso y por lo tanto 0.
Una sentencia printf("%d\n", 3 > 4 + 7 && 9); imprime un 0.

2.12.6.1 Asociatividad de Operadores.

Si dos operadores tienen la misma precedencia en una expresión, entonces la asociatividad prevalece para ordenar la ejecución.

Ejemplo:
4 == 6 != 5
Los operadores == y != tienen la misma precedencia, pero la asociatividad, entre ellos, rige de izquierda a derecha, luego:
((4 == 6) ! = 5) Se ejecuta primero 4 == 6 lo que resulta en 0 y luego (0 != 5) lo que resulta verdadero y por lo tanto en un 1.

2.12.7 Conversiones y Cast.

¿Qué sucede cuando en una expresión hay variables u operandos de distinto tipo? La respuesta a esta pregunta es que en general las variables u operandos menores se convierten al mayor existente en la expresión.
Ejemplo:

```
double gama = 34.5;
float aa = 10.5;
int val = 45;
```

Sea la expresión: `gama = gama*val + aa*2.0;`

En esta expresión la variable **val** se convierte a double, **aa** se convierte a double; todo esto ocurre previamente a ejecutarse ningún cálculo en dicha expresión. Las conversiones se realizan tomando en cuenta el operando mayor, que en este caso es **gama** y es del tipo double.
En C, estas conversiones se denominan **implícitas** y sus reglas son las siguientes:

Primero, en toda expresión que involucre operandos char y short, estos se convierten a int.
Luego,

- Si cualquier operando en la expresión es long double, entonces todos los otros se convertirán a long double y el resultado de la expresión será del tipo long double.
- De lo contrario, si cualquier operando es double, entonces todos los otros se convertirán a double y el resultado será del tipo double.
- De lo contrario, si cualquier operando es float, entonces todos los otros se convertirán a float y el resultado será del tipo float.
- De lo contrario, si cualquier operando es unsigned long int, entonces todos los otros se convertirán a unsigned long int y el resultado será del tipo unsigned long int.
- De lo contrario, si cualquier operando es long int y otro operando cualquiera es del tipo unsigned int entonces se consideran dos casos: si unsigned int puede convertirse a long int, entonces se convertirá a long int y el resultado será del tipo long int y si no, ambos se convierten a unsigned long int y el resultado es del tipo unsigned long int.

La siguiente tabla muestra la jerarquía en la conversión:

long double
double
float
unsigned long int
long int
unsigned int
int
short char

La jerarquía es ascendente. Las conversiones las realiza C de manera automática.
La conversión en orden descendente debe ser de manera explícita y forzada, porque implica pérdida de precisión.

Las conversiones explícitas son forzadas y pueden hacerse de manera ascendente o descendente ocupando lo que se llama **cast**.
Ejemplo:

```
float bb = 12.3;
int aa = 23;
```

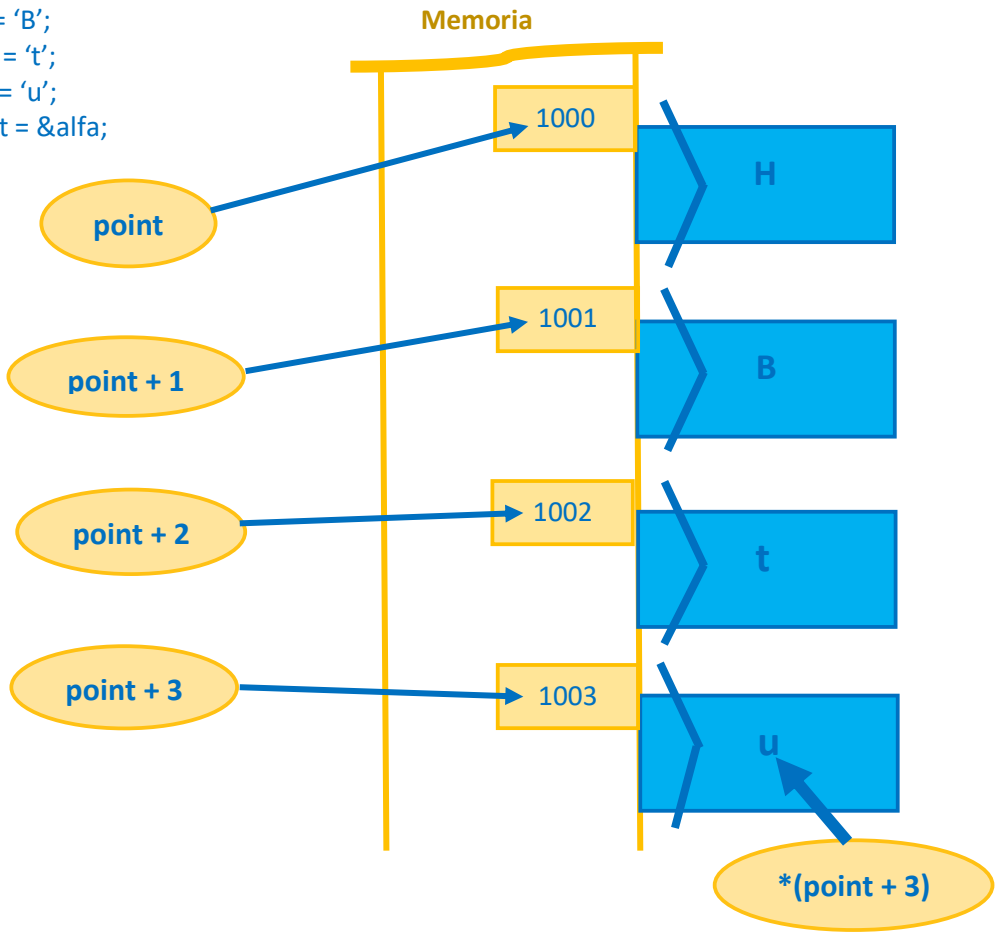
En la expresión (int)bb + aa la variable bb se convierte explícitamente al tipo int.
Lo indicado en paréntesis se denomina **cast**.
En general:

(tipo) variable o expresión convierte al tipo en paréntesis deseado.

2.13 Aritmética de punteros

Una vez inicializado un puntero, éste puede apuntar a distintos lugares en la memoria usando operaciones de suma o resta con enteros. Un puntero es una variable que guarda la dirección de un objeto o espacio de memoria. Además, y muy importante, el puntero sabe a qué tipo de objeto apunta. La información del tipo de objeto (espacio de memoria) al cual apunta se usa para calcular la verdadera dirección resultante cuando se suma o resta un entero a un puntero. En nuestra analogía de la memoria, es equivalente a decir que un puntero conoce el tamaño de la casa a la cual apunta.
Ejemplo 1:

```
char alfa = 'H';
char beta = 'B';
char gama = 't';
char delta = 'u';
char* point = &alfa;
```



En el ejemplo1 la letra H está guardada en una casa que tiene la dirección 1000. Cada casa de color azul tiene el tamaño de un byte. Una casa, en nuestra analogía, es un objeto (espacio de memoria) con una dirección y un nombre (variable). El objeto o casa que guarda la letra H tiene el nombre alfa y la dirección 1000.
Al sumar un entero 1 a un puntero, la dirección que se obtiene se calcula de la siguiente manera:

point + 1 es equivalente a:

point + 1* tamaño de la casa en bytes

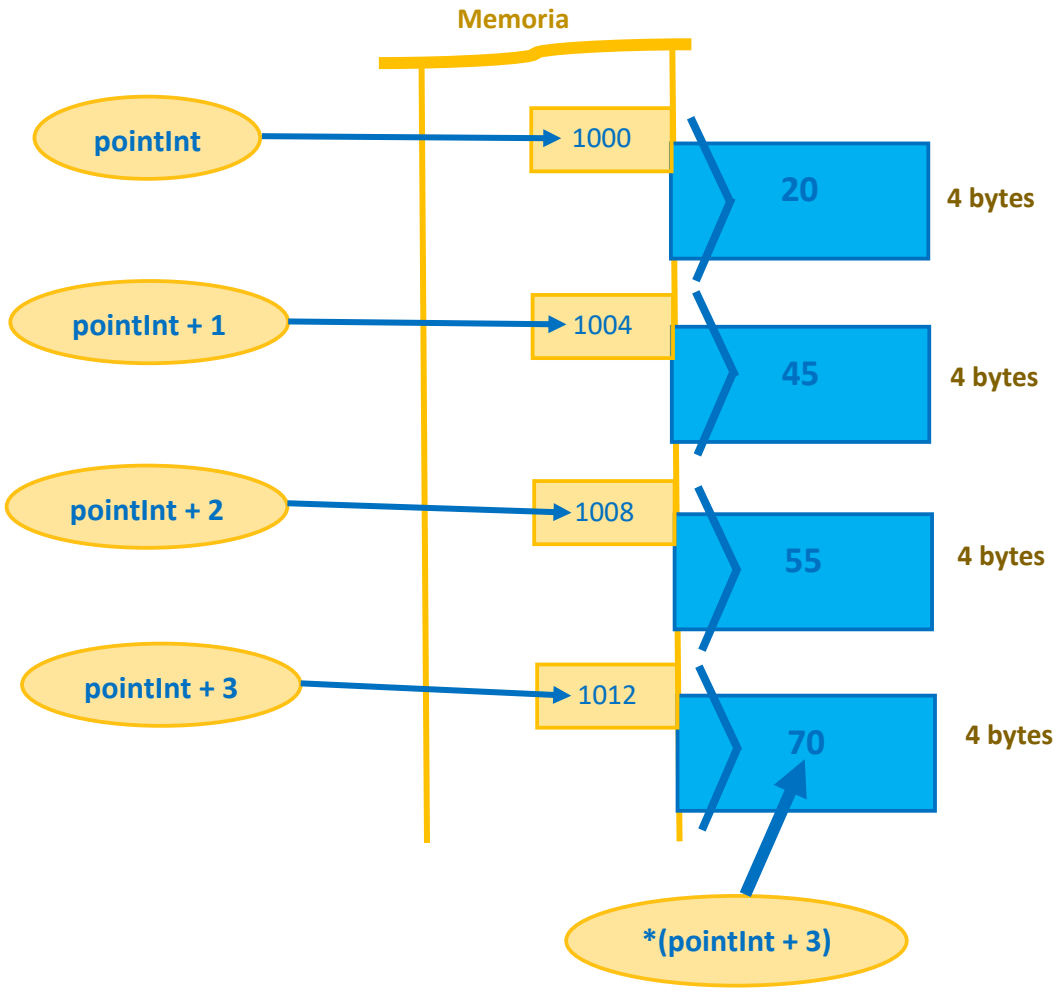
Esto es: **point + 1 * 1 byte = 1001**

Luego, point + 1 es equivalente a la dirección 1001, dónde se guarda la letra **B**. Es decir, al sumar uno al puntero point se pasa a apuntar a la variable beta.
point + 3 apunta a la variable delta, por lo cual al agregar un asterisco a la izquierda de esta dirección se obtiene la letra u. Así, *(point + 3) es una nueva variable que representa a la letra u.
El valor 1 que se suma a la variable point se conoce como índice y la posición donde se inicializa un puntero tiene un índice 0.
Ejemplo: point es lo mismo que point + 0
Estos índices pueden sumarse o restarse a un puntero. Al sumar se avanza en el sentido que crecen las direcciones y la resta avanza en el sentido que decrecen las direcciones.

Ejemplo 2:
¿Qué sucede cuando se tiene un conjunto de casas de 4 bytes cada una? Usando la analogía de la memoria:

```
int alfa = 20;
int beta = 45;
int gama = 55;
int delta = 70;
int* pointInt = &alfa;
```

El Tipo entero necesita 4 bytes para guardar un valor y asumiremos en nuestra analogía que las casas tienen un tamaño de 4 bytes para guardar un valor entero. Veamos, cómo se comporta la aritmética de punteros en este caso.



Al sumar a pointInt un 1 no se obtiene 1001, lo que se obtiene es lo siguiente:

pointInt + 1 es equivalente a:

pointInt + 1* tamaño de la casa en bytes

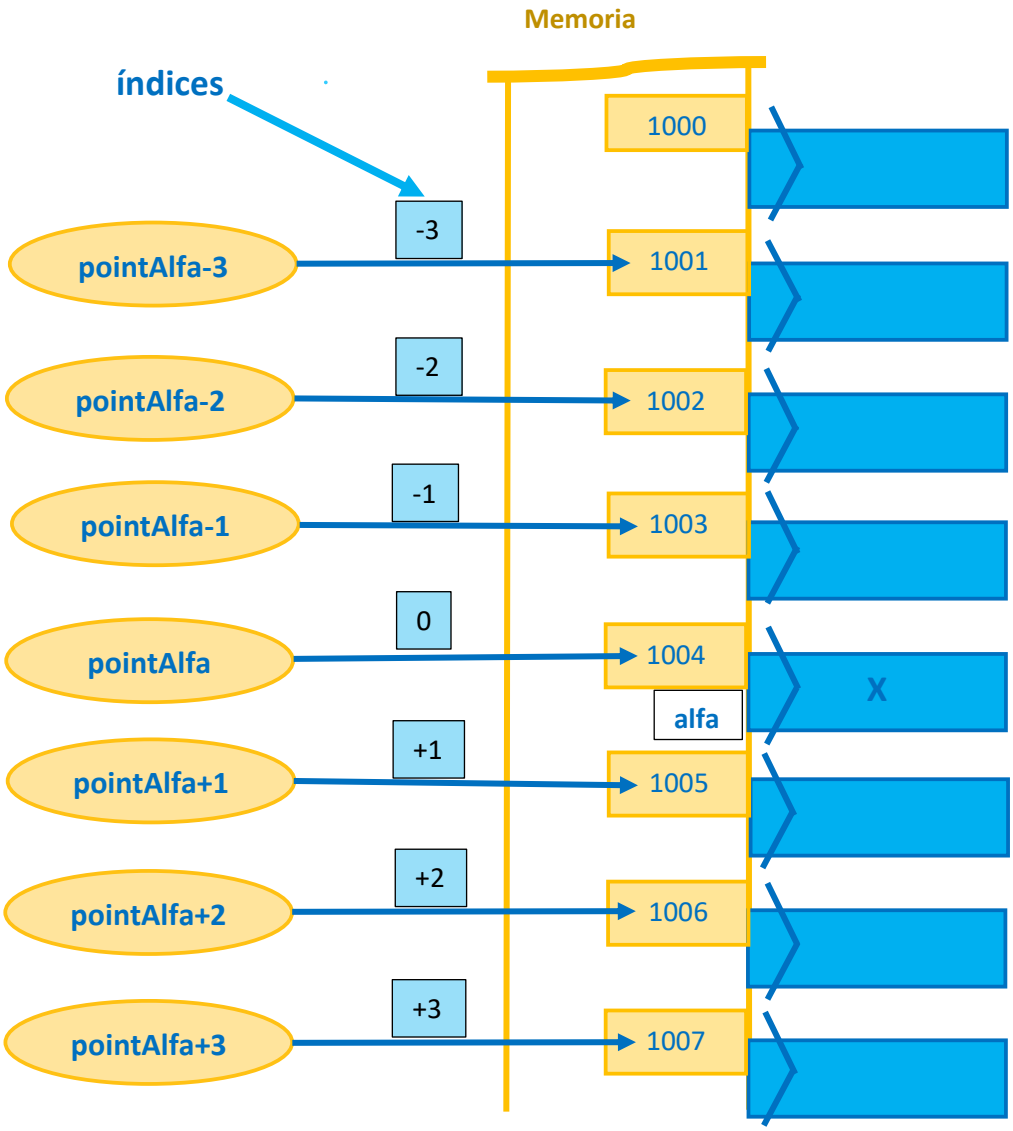
Esto es: **pointInt + 1 * 4 bytes = 1004**

El tamaño de la casa hace que la dirección cambie de 1000 a 1004, para el puntero pointInt. De aquí la importancia que el puntero además de ser un número entero sin signo posea la información a que tamaño de objeto o casa está apuntando. **Sin la información del tamaño, no es un puntero y es sólo un número.**

El caso del ejemplo1 y ejemplo2 corresponde al Tipo arreglo, que veremos en el próximo capítulo. En un arreglo, las casas u objetos que guardan un valor son siempre del mismo tamaño.

Ejemplo 3.
¿Qué sucede con el índice cuando el puntero se inicializa con una dirección no del comienzo, sino que con una dirección de una casa entre medio?

```
char alfa = 'X';
char* pointAlfa = &alfa;      // INICIALIZACION DEL PUNTERO
```

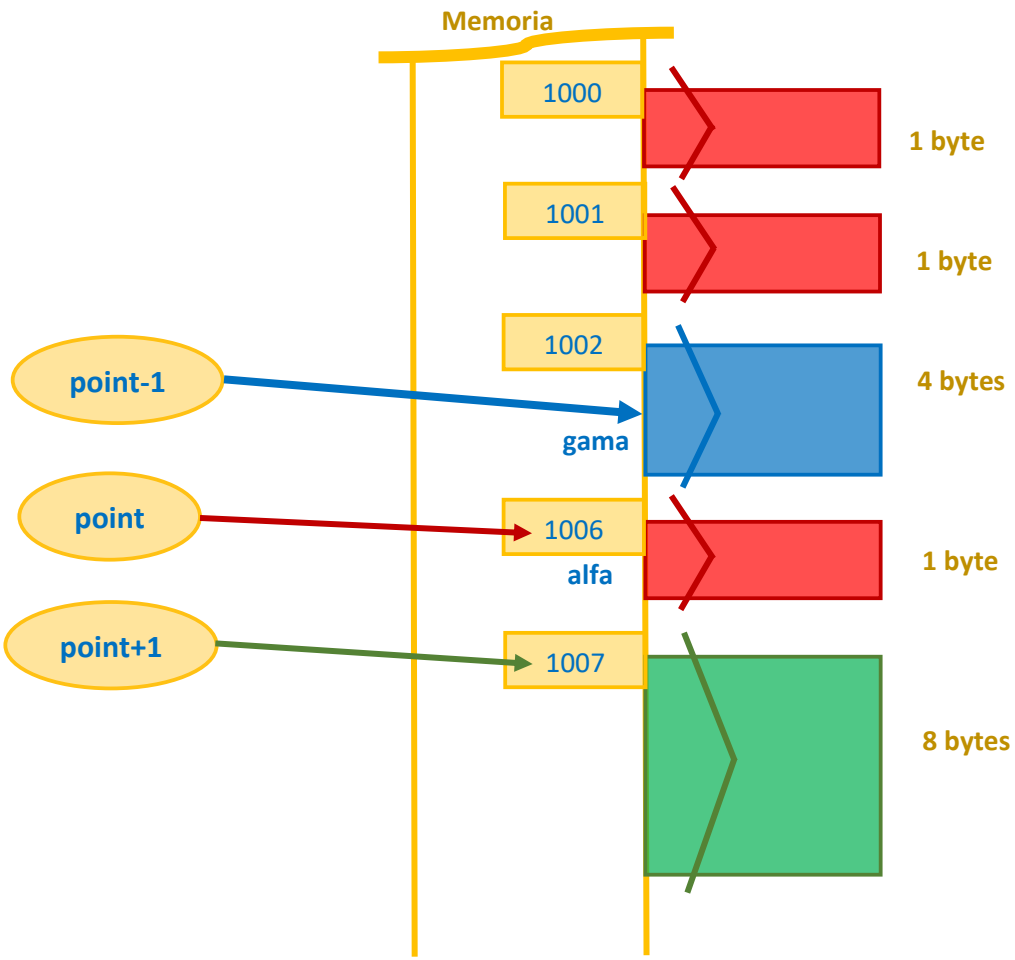


El puntero pointAlfa se inicializa con la dirección donde vive la variable alfa. Luego, para esta casa automáticamente se asigna el índice 0. Para apuntar a una casa antes de alfa, los índices son negativos,

esto es hacia las direcciones que decrecen de valor. Para apuntar a una casa delante de alfa, los índices son positivos, esto es se suman. El gráfico anterior muestra esto para ambos casos.
Lo importante es tener presente que en el lugar donde se inicializa un puntero, ese lugar es el índice 0.
Todo es válido si todas las casas son del mismo tamaño. ¿Qué sucede cuando las casas u objetos de memoria son contiguos, pero de distinto tamaño?

Entonces, no es posible recorrer toda la calle usando un solo puntero, porque el puntero guarda la información únicamente de su propio tamaño al cual apunta inicialmente y esto, ciertamente llevaría a un error en el cálculo de la dirección.
Esto ocurre en el caso del Tipo estructura, y por esto, los elementos que conforman una estructura no pueden ser accesados mediante un único puntero, como en un Tipo arreglo.

Ejemplo 4.
El siguiente gráfico muestra objetos de memoria o casas contiguas, pero de diferente tamaño.



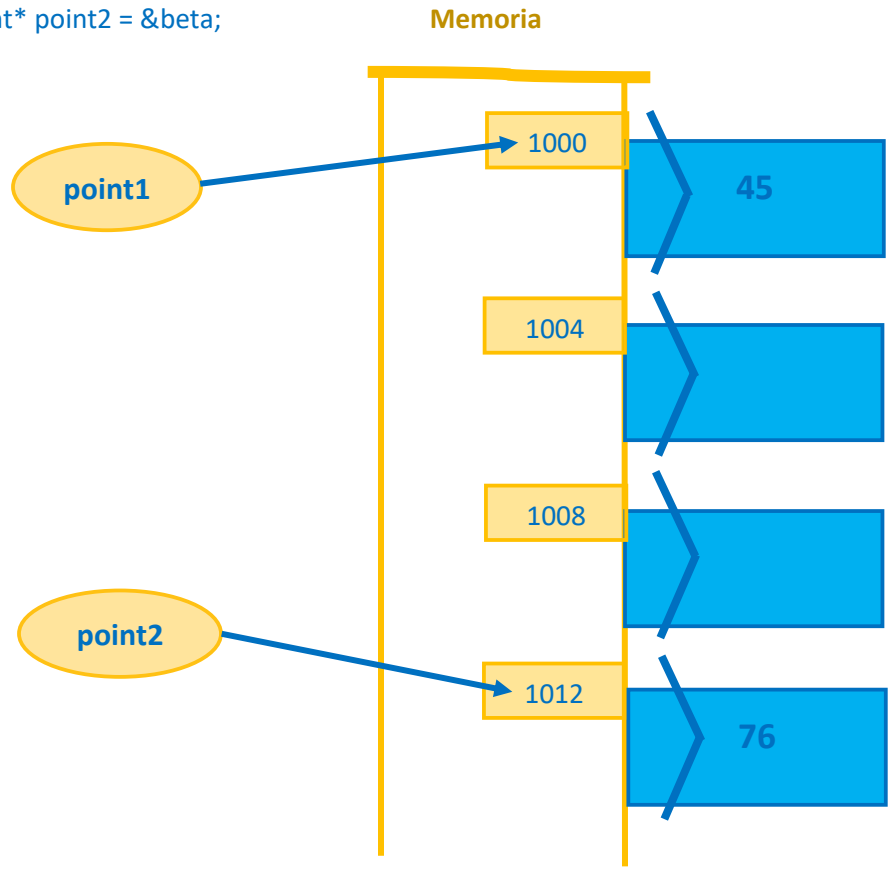
Si se elige un puntero point del tipo char* que apunte a la casa donde está la variable alfa, este puntero sabe que su tamaño es un byte. Si se resta un 1 al puntero point, éste apuntará a un lugar interno de la casa donde vive la variable gama (por su tamaño avanza un byte), y por lo tanto es un error. Este error no es detectado por el programa en ejecución y muy probablemente hará que el programa falle.
También, si se aplica un asterisco a point+1, éste tomará un byte y no los 8 que debería tomar (casa color verde). Este error tampoco es detectado por el programa.
Esto ocurriría en las estructuras, si se ocupara un puntero para recorrer todos los miembros u objetos de la estructura. El recorrido de los miembros u objetos de distinto tamaño en una estructura se hace de distinta forma.

Al sumar o restar un entero a un puntero, recorremos casa a casa u objeto a objeto si estos son del mismo tamaño. ¿Qué sucede si restamos un puntero de otro?

Esta pregunta se contesta en el siguiente ejemplo, donde las variables alfa y beta se asumen en la memoria como se muestra en el gráfico:

Ejemplo 5.

```
int alfa = 45;
int beta = 76;
int* point1 = &alfa;
int* point2 = &beta;
```



Si ejecutamos la sentencia printf("%d\n", point2 – point1) el resultado de la impresión es 3.

¿Porqué? La resta 1012 – 1000 = 12, pero debe dividirse por el tamaño del objeto o casa que es 4, lo que da como resultado 3. Esto nos dice que entre ambas direcciones hay un espacio de memoria equivalente a tres casas.
La diferencia entre punteros es posible realizarla, pero es necesario saber interpretar el resultado.

¿Es posible comparar un puntero con otro? Sí, efectivamente se pueden usar los siguientes operadores para comparar punteros: ==, !=, <, <=, >, >= los cuales son en orden: igual, distinto, menor que, menor o igual, mayor que, mayor o igual.
Ejemplos: point2 == point1 entrega como resultado 0.
point2 > point1 entrega como resultado 1. ¿y los otros?
Los resultados que se obtienen es 1 si la comparación lógica es verdadera o 0 si la comparación es falsa.

2.14 Propiedades importantes de punteros.

Los punteros tienen dos propiedades importantes que hay que tener siempre presente:

1.- Todo puntero se convierte en una variable o nuevo nombre para el objeto al cual apunta al agregarle un asterisco a la izquierda de éste.

Ejemplo: Usando los datos del ejemplo 5 anterior, point1 apunta a la variable alfa y *point1 es una nueva variable o nombre para el objeto que guarda el valor 45.
La variable alfa representa al valor 45 y ahora *point1 también representa al valor 45. Este asterisco, se conoce como **dereferenciación**. **La variable alfa y *point1 son funcionalmente equivalentes**.
Así, como *point1 es una nueva variable, ésta se puede usar tanto al lado izquierdo como al lado derecho del signo de asignación **=** .
Ejemplos:

```
*point1 = 23;      // cambia el valor 45 de alfa por 23.
beta = *point1;  // cambia el valor de beta por el valor que tenga alfa en ese momento.
```

2.- Todo puntero adquiere implícitamente corchetes cuadrados **[ ]** , que lo convierten en una variable al igual que lo hace el asterisco de dereferenciación. Es obligatorio usar índices con los corchetes **[ ]**.

Al igual que en ejemplo del caso 1, el puntero point1 se convierte en una variable al aplicarle los corchetes con índice 0, esto es, point[0] es una nueva variable que representa al valor 45.
La variable **alfa** y **point[0]** son funcionalmente equivalentes. Luego, point[0] se puede usar al lado derecho o izquierdo del símbolo de asignación.
Ejemplos:

```
point1[0] = 23;       // cambia el valor 45 de alfa por 23.
beta = point1[0];   // cambia el valor de beta por el valor que tenga alfa en ese momento.
```

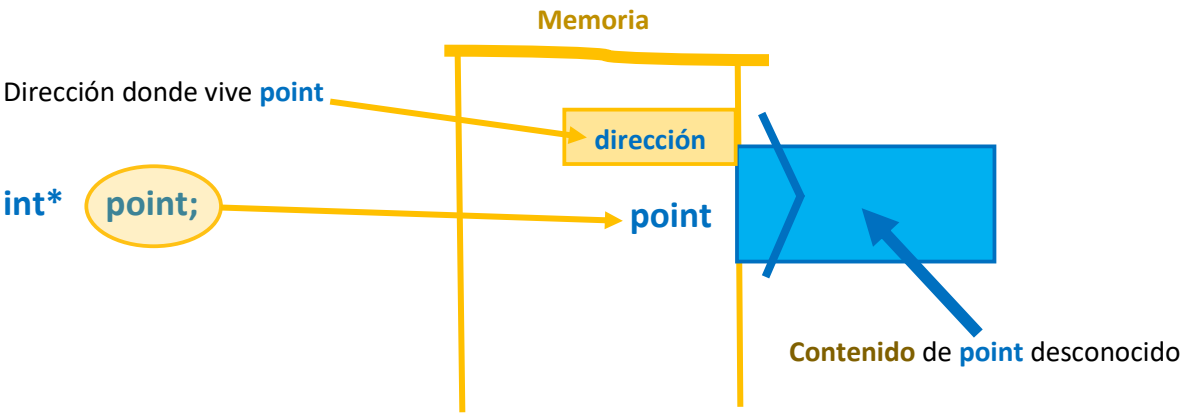
¿Entonces, se usa * o se usa [ ]?
Cuando se trata de una sola variable u objeto y su puntero, es muy recomendable usar *. Cuando se está en un arreglo, donde se recorren varios objetos es preferible usar [ ] con sus correspondientes índices.

¿Por qué en este ejemplo se usó el índice 0? Para un puntero a una sola variable, el índice 0 nos indica la propia posición de la variable. Usar 1 o -1 u otro índice, nos llevaría a otro objeto en otra parte de la memoria y sería un error.

2.15 Puntero NULL

Un puntero apunta a una parte de la memoria física real que llamamos objeto donde se guarda un valor. ¿Cómo se crea un puntero de algún tipo, pero que no apunte a nada? Declarar un puntero sin inicializarlo no significa que no apunte a algo. Este puntero puede apuntar a cualquier lugar, dependiendo del valor que tenía el objeto elegido por el sistema operativo al momento de declarar el puntero. En otras palabras, al momento de declarar el puntero sin inicializar, igual el sistema operativo le elige una casa donde vivir. El contenido de esa casa podría ser cualquier cosa y el puntero quedar apuntando a cualquier parte. De aquí, que se recomienda nunca crear un puntero sin inicializar.

Ejemplo:



point es una variable que guarda un **contenido** que es una dirección. Si no se inicializa point, entonces se desconoce qué valor guarda point y puede apuntar a cualquier parte, dependiendo del valor que guarda la casa donde vive point.

A modo de prueba ejecute el siguiente programa y vea que imprime:

```
int* point;
printf("%d\n", point); // imprime el contenido de point.
```

El lenguaje C entrega una Macro que permite declarar un puntero sin que apunte a ningún lugar de manera segura. Esta Macro es la palabra **NULL**.

Luego, al declarar un puntero como:

```
int* point = NULL;   // point no apunta a ningún lugar de manera segura.
```

Si imprime ahora point con el formato d, va a obtener un cero. Pero distintos compiladores pueden interpretar NULL de distinta forma, por lo cual es mejor no tratar de saber que es. Es suficiente entender que NULL crea un puntero que no apunta a ningún lugar, de manera segura.
En cualquier momento después de declarar point como nulo, se puede asignar a él una dirección del tipo correspondiente.

2.16 Puntero a puntero

Un puntero a puntero es una situación en la cual deben utilizarse dos direcciones para llegar al objeto que contiene el valor o dato deseado. En forma genérica, se puede analizar la siguiente sentencia, de derecha a izquierda:

****beta**

beta debe ser una dirección para que tenga sentido el * aplicado a su izquierda. ***beta** representa el contenido guardado en la dirección beta.

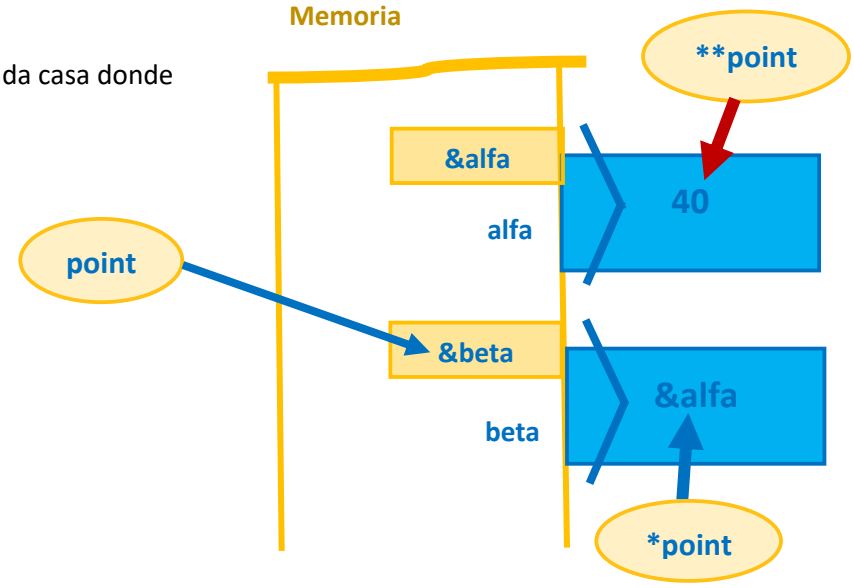
***beta** o el contenido que representa, también debe ser una dirección para que tenga sentido el * aplicado a ***beta**. Luego, ****beta** representa un valor guardado en la dirección ***beta**. Se han utilizado entonces, dos direcciones **beta** y ***beta**.

La manera de crear un puntero a puntero es la siguiente:

```
int**   point;     // se crea un puntero a puntero.
```

La variable **point** apunta a una dirección que contiene otra dirección que apunta al lugar donde está el valor o dato que se requiere. Esto se puede ver de forma gráfica de la siguiente manera:

point apunta a la segunda casa donde se guarda **&alfa**.



El gráfico anterior se programa como sigue:

```
int alfa = 40;

int* beta = &alfa;

int** point = &beta;

printf(“%d\n”, **point); //imprime el valor 40
```

Es importante comprender que, si point es un puntero a puntero, entonces solo se puede asignar a point una dirección que, a su vez, sea una variable que contenga otra dirección.
Como ejemplo, a **point no es posible** asignarle &alfa porque en esa dirección se guarda un valor numérico y no otra dirección.

Un puntero a puntero también adquiere paréntesis cuadrados de manera implícita. Para el ejemplo anterior, **point[0][0]** es equivalente a ****point**. Ambos representan al valor 40.
La variable **point[0]** es equivalente a ***point** y representan a la dirección de **alfa**, esto es **&alfa**, que es el contenido del puntero **beta**.

2.17 Puntero a void

El uso de la palabra especial del lenguaje C **void**, permite crear un puntero sin especificar el tipo de dato al cual apunta en el momento de declarar el puntero.
Posteriormente en el programa, se puede asignar la dirección de una variable de cualquier tipo a dicho puntero a void. Para usar este puntero para obtener el valor del objeto al cual apunta, debe usarse un casting con el tipo de la dirección utilizada.

Ejemplo:

```
void* point = NULL;                 //se declara puntero point a void.
float alfa = 34.0;
point = &alfa;                      //se asigna dirección de variable alfa tipo float.
printf("%f\n", *((float*)point));  //se aplica casting puntero a float a puntero point.

int delta = 45;
point = &delta;                     //ahora se asigna otra dirección a point del tipo entero.
printf("%d\n", *((int*)point));   //se aplica casting puntero a entero a puntero point.
```

El puntero a void permite usarlo con cualquier tipo de variable, usando de manera apropiada un casting.

2.18 Uso de const

La palabra clave **const**, permite asegurar que el valor de un objeto no pueda ser modificado o que a un puntero no se permita asignar una nueva dirección.
Ejemplos de estos casos:

```
const    int       var = 20;
var = 34;   // ERROR var está declarada como constante.

const    int*      point = &var;
*point = 34;   // ERROR point apunta a un valor constante
int*     const     p1 = &var;
```

3 ARREGLOS y ASIGNACIÓN DINÁMICA DE MEMORIA

Vamos a la luna, no porque sea fácil,
sino porque es difícil.
John Fitzgerald Kennedy.

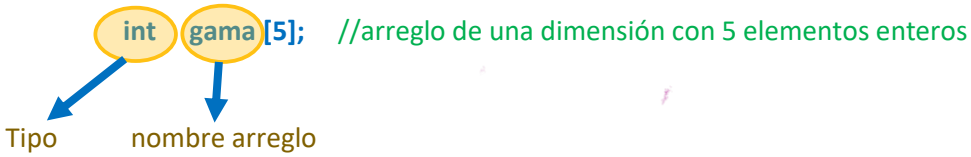
3.1 Arreglos

Un arreglo en C es un Tipo que agrupa de manera consecutiva en memoria, elementos de un mismo tipo. Los elementos del arreglo pueden ser de cualquier tipo permitido por el lenguaje C. Los tipos más comunes usados en un arreglo son int, float, double, char, pero también se usan a menudo estructuras, punteros y punteros a funciones. Tipos más complejos creados con typedef pueden ser incluidos en un arreglo. Un arreglo puede tener más de una dimensión, siendo las dimensiones 1, 2 y 3 las más utilizadas. Un arreglo de una dimensión se puede asociar a un vector, uno de dos dimensiones a una imagen y uno de tres dimensiones a un cubo.

3.2 Declaración de arreglos.

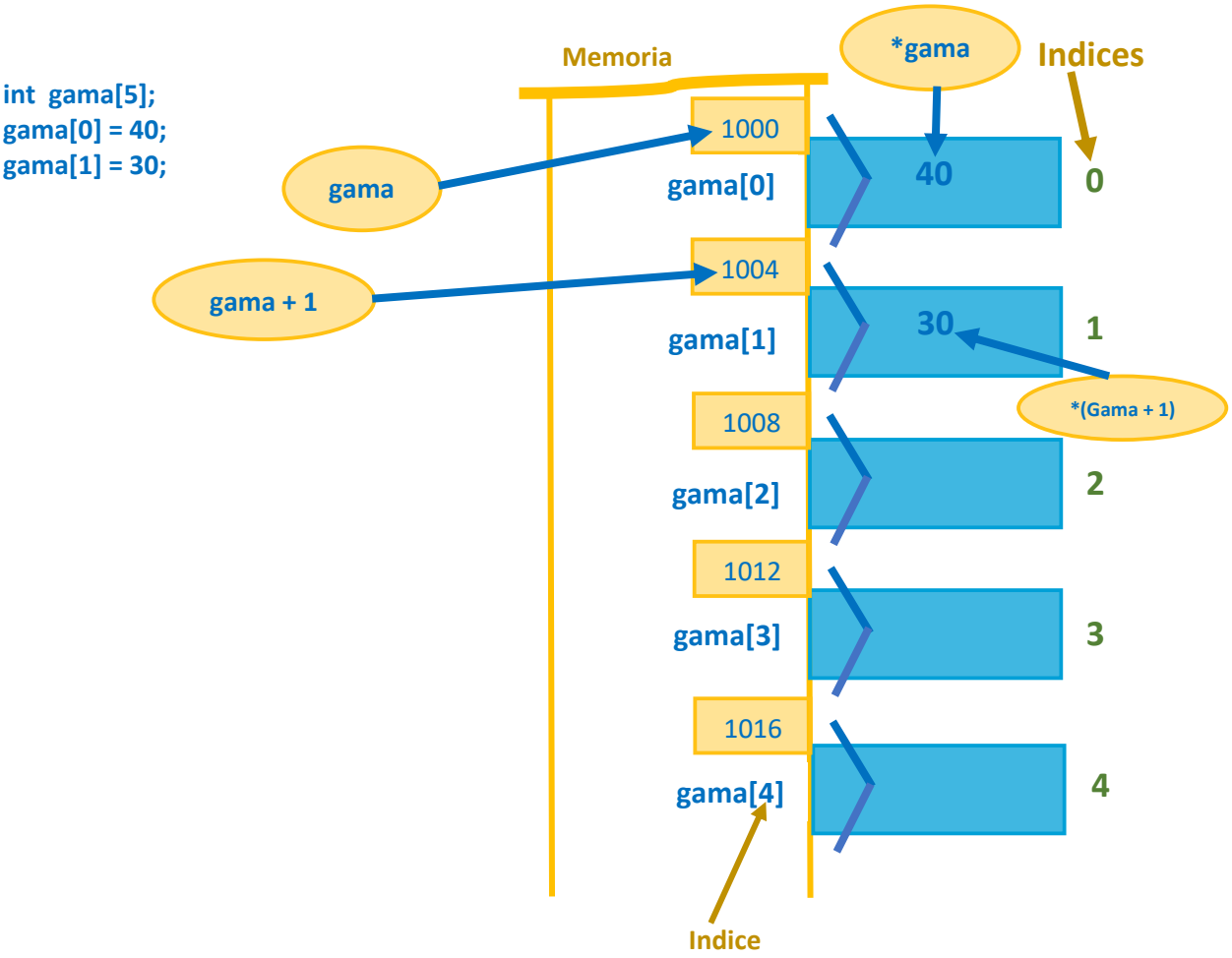
La declaración de un arreglo debe incluir el tipo de los elementos que lo conforman, las dimensiones y el número total de elementos por dimensión. Cada arreglo tiene un nombre dado por el programador, y este nombre siempre es implícitamente la dirección del primer elemento del arreglo, para el caso de una dimensión.

Ejemplos:



En este ejemplo, int es el tipo para los 5 elementos del arreglo y gama es el nombre del arreglo. Los corchetes [ ] colocados a la derecha del nombre, encierran un número entero que indica cuántos elementos tiene el arreglo, y la cantidad de corchetes [ ] muestran cuántas dimensiones posee el arreglo. El número dentro de un corchete o dimensión, debe ser un valor constante. No puede ser una variable, excepto en las versiones C99 y C11 del lenguaje C.

La declaración del arreglo **gama** crea lo siguiente en la memoria (nuestra analogía):



La declaración **int gama[5];** busca en la memoria cinco casas contiguas del mismo tamaño y vacías.
La dirección 1000 de la primera casa es igual al nombre del arreglo gama. El nombre del arreglo, por ser una dirección, adquiere de manera implícita índices en paréntesis cuadrados, de tal forma que cada elemento o valor del arreglo se representa por el nombre y su correspondiente índice.
gama[0] representa al valor guardado en la primera casa.
gama[1] representa al valor guardado en la segunda casa y así sucesivamente hasta el último índice que es el 4. En lenguaje C los índices siempre comienzan en cero.

Luego, gama[0] = 40; es una sentencia que asigna el valor entero 40 a la variable gama[0] que vive en la primera casa. Efectivamente, gama[0], gama[1] …etc son las variables del arreglo.
Como gama es la dirección de la primera casa, entonces también se puede usar para obtener el valor guardado en esa casa anteponiendo un asterisco al nombre, esto es ***gama**.

Para guardar un valor en la segunda casa, se usa el índice 1 y luego la sentencia gama[1] = 30; asigna el valor 30 a la variable gama[1] que representa al valor guardado en la segunda casa.

También usando el índice 1, sumado a la dirección gama se pasa a la segunda casa y por lo tanto se puede usar la dirección gama + 1 para obtener el valor guardado, esto es, *(gama + 1).
Así, en un arreglo de una dimensión, cada valor guardado en una casa se puede obtener por:
gama[i] o *(gama + i) , dónde i es el índice, que en este ejemplo puede ser 0,1,2,3,4.

Importante: gama[i] o *(gama + i) son las variables que representan los valores guardados en memoria y por lo tanto son los nombres de los objetos existentes en la memoria del computador y entonces pueden estar al lado izquierdo o al lado derecho del signo **=** (símbolo de asignación).

Ejemplos:

```
*(gama + 2) = 45;    //guarda el valor 45 en la tercera casa del arreglo
gama[2] = 23;     // guarda el valor 23 en la tercera casa. Se borra el 45 escrito por la sentencia anterior

gama[3] = *(gama +2);  //guarda el valor 23 en la cuarta casa, porque *(gama + 2) es la variable que representa al valor 23, de la tercera casa.
```

3.2.1 Inicialización de un arreglo

La inicialización de un arreglo se hace al momento de su declaración. Para ello se utilizan los paréntesis curvos { }.
Ejemplo:

```
int beta[6] = {2,45,7,89,12,56};
```

El arreglo beta tiene una dimensión con seis valores del tipo int. Los paréntesis { } se usan para encerrar los valores con los cuales se inicializa cada objeto del arreglo, separados por coma.
Luego la variable beta[0] representa al valor 2, la variable beta[3] representa al valor 89 etc.

3.2.2 Inicialización designada de un arreglo

Cuando se tiene un arreglo en el cual unos pocos elementos tienen valor y el resto es cero, se puede utilizar una inicialización designada, que consiste en lo siguiente:
Ejemplo:

```
int delta[20] = {[1] = 34, [10] = 4, [15] = 23};
```

Sólo se inicializan aquellos elementos del arreglo que tienen un valor distinto de cero y todos los demás elementos se asumen cero. En el ejemplo, al segundo elemento se asigna el valor 34, al onceavo elemento el valor 4 y al elemento en la posición 16 el valor 23. Para ello se usan corchetes con índice o designador, como se muestra en el ejemplo. Esta forma de inicialización está disponible en el lenguaje C a partir de la versión C99. En esta forma de inicializar un arreglo, no es necesario conservar el orden en que se asignan los valores a los elementos del arreglo.

3.3 Arreglos de dos y tres dimensiones

Las dimensiones de un arreglo permiten asociar los valores o datos del arreglo a estructuras conocidas.
Por ejemplo, un arreglo de dos dimensiones puede asociarse a una matriz, a una imagen y a cualquier estructura que requiera de un ordenamiento de datos en dos niveles. Un arreglo de tres dimensiones puede asociarse a un paquete de imágenes, como por ejemplo aquellas obtenidas por un resonador magnético.

3.3.1 Arreglo de dos dimensiones

La declaración e inicialización de un arreglo de dos dimensiones es de la siguiente manera, ejemplo:

```
float   dataMatriz[2][3] = {{2.0,3.1,4.5},{4.9,6.7,3.0}};
```

La dirección &dataMatriz[0][0] apunta al primer valor del arreglo, esto es al número decimal 2.0. La variable que representa este valor es dataMatriz[0][0]. También se puede representar por *&dataMatriz[0][0].
Cualquier valor del arreglo se representa por las variables dataMatriz[i][j] ó *(&dataMatriz[0][0] + k).
El valor de i puede ser 0 o 1 y el valor de j puede ser 0, 1, 2.
El índice k puede variar entre 0 y 5, para este ejemplo.

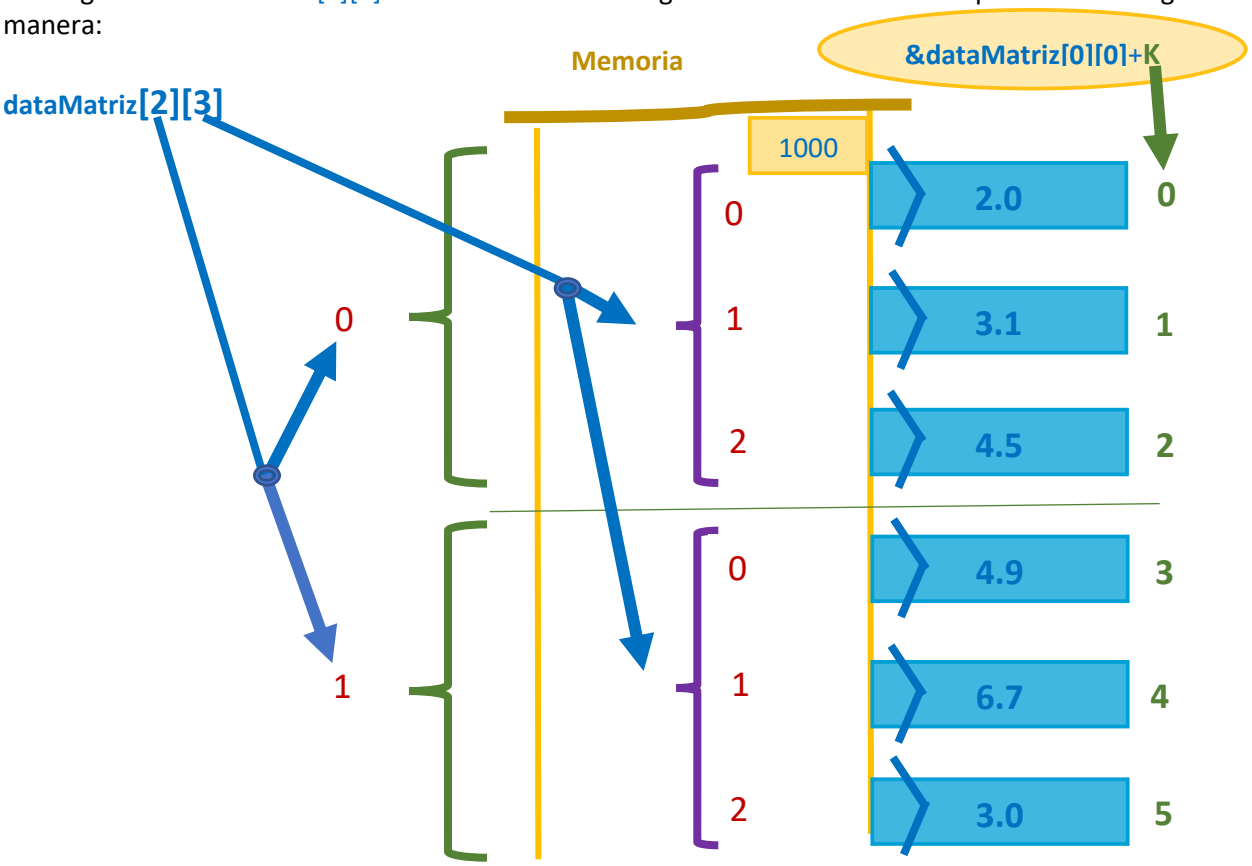
El primer [ ] representa las filas y el segundo [ ] representa las columnas.

El valor 6.7 del arreglo se representa por las variables dataMatriz[1][1] ó *(&dataMatriz[0][0] + 4).

Ejemplo: Cambiar el valor 4.5 del arreglo a 60.0.
Se puede hacer de dos formas:

```
dataMatriz[0][2] = 60.0;   // usando paréntesis [ ]

*(&dataMatriz[0][0] + 2) = 60.0   //usando asterisco aplicado a la dirección
```

El arreglo float dataMatriz[2][3] se visualiza en la analogía de la memoria del computador de la siguiente manera:



&dataMatriz[0][0] es la dirección 1000 que corresponde a la dirección del primer valor del arreglo. Ciertamente esta dirección puede ser cualquiera asignada por el sistema operativo.
Los índices en lenguaje C siempre parten de cero.

Pregunta: ¿Cómo se obtiene la dirección de cualquier valor del arreglo?
De dos formas: Obtener la dirección del valor 4.9 del arreglo.

1º Forma: **&dataMatriz[1][0]** //anteponiendo el símbolo & a la variable dataMatriz[1[[0]

2º Forma: **&dataMatriz[0][0]** + **3** //sumando 3 a la dirección &dataMatriz[0][0]

Pregunta: ¿Cómo se imprime cualquier valor del arreglo?
Para imprimir un valor del arreglo, debe usarse la variable que represente al valor que se quiere imprimir con el debido formato.

Ejemplo: Imprimir el valor 3.0 del arreglo.

```
printf("%f\n", dataMatriz[1][2]);
```

también puede usarse:

```
printf("%f\n", *(&dataMatriz[0][0] + 5))
```

El formato utilizado para la impresión es f , porque los valores del arreglo son del tipo float.

3.3.2 Arreglo de tres dimensiones

Un arreglo de tres dimensiones se declara e inicializa de la siguiente manera:

```
double   dat[2][3][2] = {{{2.0, 4.5},{1.4, 6.7},{2.1, 3.9}},{{7.0, 6.0 },{5.1, 4.8},{6.5,9.0}}};
```

El nombre del arreglo **dat** es la dirección de la dirección de la dirección del primer valor o elemento del arreglo, esto es ***dat** es 2.0.
La variable que representa al valor 2.0 es dat[0][0][0].

La dirección de cualquier valor se obtiene aplicando el símbolo & a la izquierda de la variable que represente dicho valor.
Ejemplo: Obtenga la dirección del valor 4.8.

```
&dat[1][1][1]  //dirección del valor 4.8
```

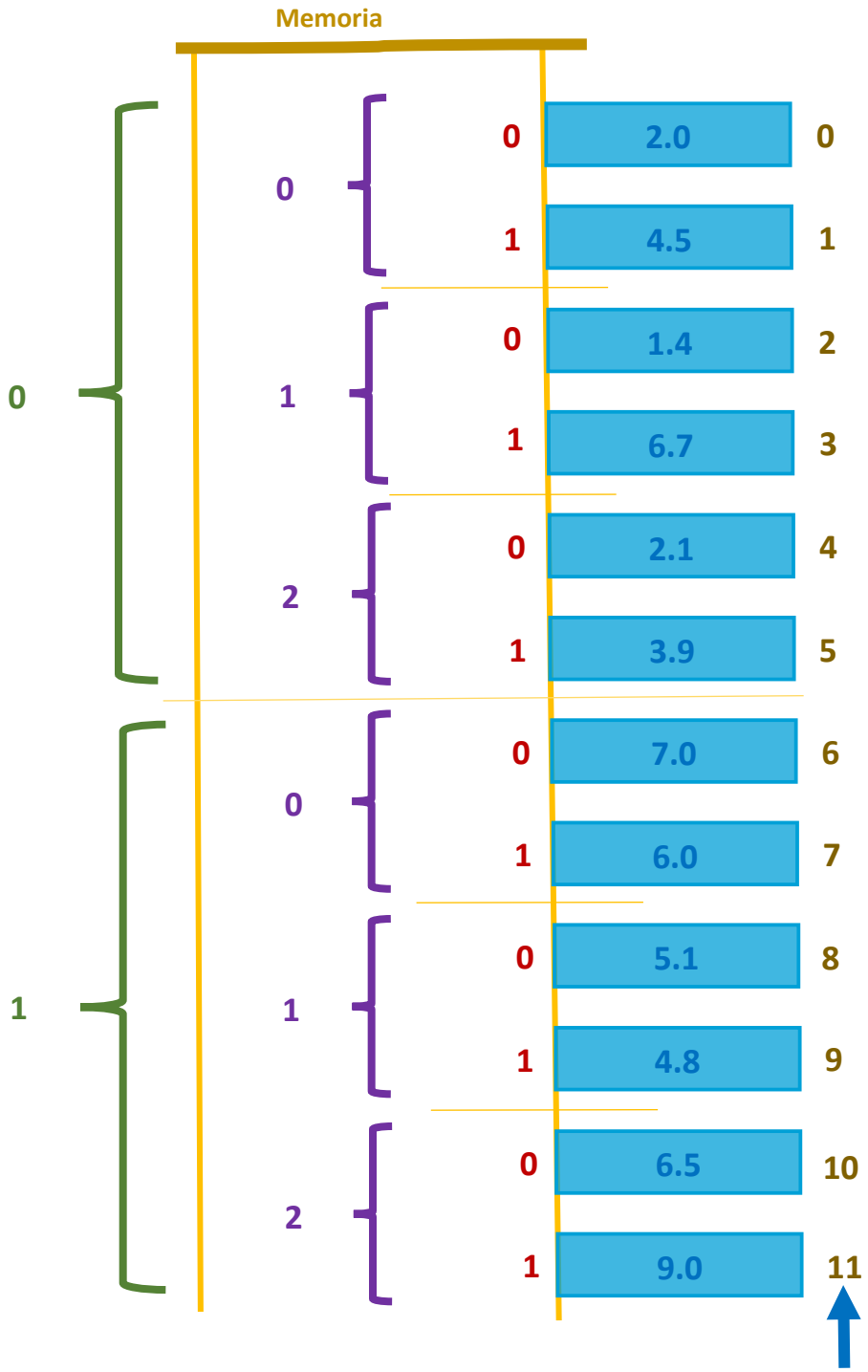
Ejemplo: Sumar los valores 6.7 con 9.0 y reemplazar el valor 4.5 por este resultado.

Se puede hacer de dos formas:

1º Forma: `dat[0][0][1] = dat[0][1][1] + dat[1][2][1] //usando paréntesis`

2º Forma: `*(&dat[0][0][0] + 1) = *(&dat[0][0][0] + 3) + *(&dat[0][0][0] + 11); //usando asterisco`

Visto en memoria (analogía) este arreglo se visualiza de la siguiente manera:



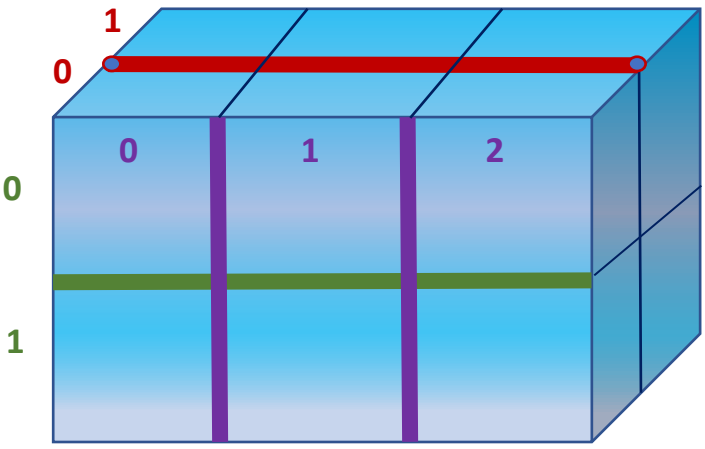
Variables:

dat [2] [3] [2]

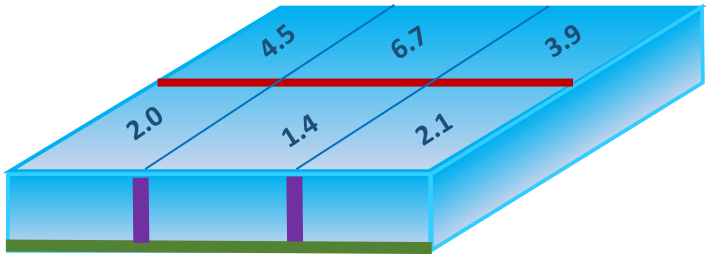
Direcciones: **&dat [0][0][0] + K**

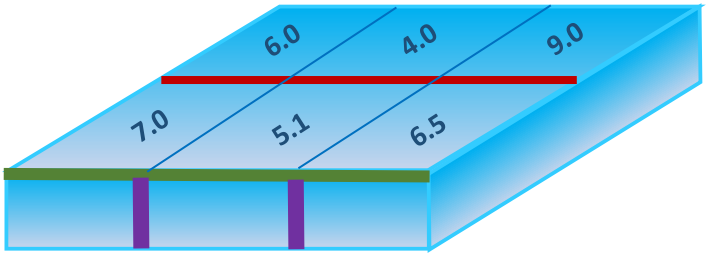
La dirección de cualquier valor se puede obtener por **&dat[i][j][n]** o por **&dat[0][0][0] + k**.
El valor en cualquiera de las direcciones, se obtiene por las variables **dat[i][j][n]** ó ***(&dat[0][0][0] + k)**.
El índice i está en el rango [0 ,1], el índice j en [0,1,2] y el índice n en [0,1].
El índice k varía entre 0 y 11. En total son 2*3*2 =12 elementos o valores en el arreglo, en este ejemplo.

Otra forma interesante de visualizar el arreglo de datos de tres dimensiones es en un cubo:



Color verde indica las filas, color morado las columnas y el rojo la partición del fondo del cubo.
¿Dónde están ubicados los valores del arreglo? Para esto, es mejor mostrar el cubo cortado por la línea verde de la siguiente forma:





3.4 Punteros en arreglos.

En arreglos de dos o más dimensiones, los punteros ayudan a simplificar la nomenclatura y facilitar la manipulación de datos o valores del arreglo.
Por ejemplo:
Las direcciones en el ejemplo anterior del arreglo de tres dimensiones, están dadas por &dat[0][0][0] + k
Declarando un puntero:

```
double*  point = &dat[0][0][0];
```

Las direcciones ahora son point + k, lo cual es ciertamente más simple de manejar.
Así, cualquier variable que represente a un valor del arreglo está dado por *(point + k). Esto también es más simple que usar dat[i][j][n] para representar cualquier valor del arreglo.

En un arreglo de una dimensión, el nombre del arreglo es la dirección del primer valor o dato del arreglo.
En un arreglo de dos o más dimensiones, el significado del nombre del arreglo es algo más complejo.
Para determinar cuál es el significado del nombre de un arreglo de dos o más dimensiones, es necesario revisar lo siguiente:

En lenguaje C, toda dirección de un objeto, adquiere implícitamente paréntesis cuadrados con índice, de tal forma que la dirección con paréntesis [ ] a su derecha es otra variable que representa el valor o dato guardado en dicha dirección.

Ejemplo: `float  var = 34.5;`

El valor 34.5 se puede imprimir usando la variable var, pero también se puede imprimir con (&var)[0], porque &var es la dirección del valor 34.5 y agregando [0] a la derecha de &var, esta dirección se convierte en una variable que también representa al valor 34.5. Se ocupa el índice 0, porque es una sola variable.
Así,

```
printf("%f\n", var)     //imprime el valor 34.5

printf("%f\n", (&var)[0])    //imprime el valor 34.5
```

El paréntesis que encierra a &var es necesario, porque la expresión (&var)[0] se evalúa de derecha a izquierda, y el paréntesis [ ] sólo se puede aplicar a una dirección y no a una variable. El paréntesis ( ) asegura que el símbolo [ ] encuentre una dirección, al ser aplicado.
Aplicando este análisis a un arreglo de dos dimensiones, se determina lo siguiente:

Sea el siguiente arreglo, `float  sun[2][3] = {{2.0,3.4,5.6},{1.2,7.8,9.0}};`

sun[0][0] es la variable que representa al primer valor 2.0 del arreglo.
Por el análisis del ejemplo anterior, sun[0] debe ser la dirección del primer valor 2.0 del arreglo, puesto que si se agrega a su derecha otro [0], se obtiene sun[0][0], que sabemos que es la variable que representa al valor 2.0.
Por lo tanto, si sun[0] es una dirección, entonces podemos aplicar un * a su izquierda e imprimir el valor que representa.

Así, `printf("%f\n", *sun[0])   // efectivamente imprime el valor 2.0`

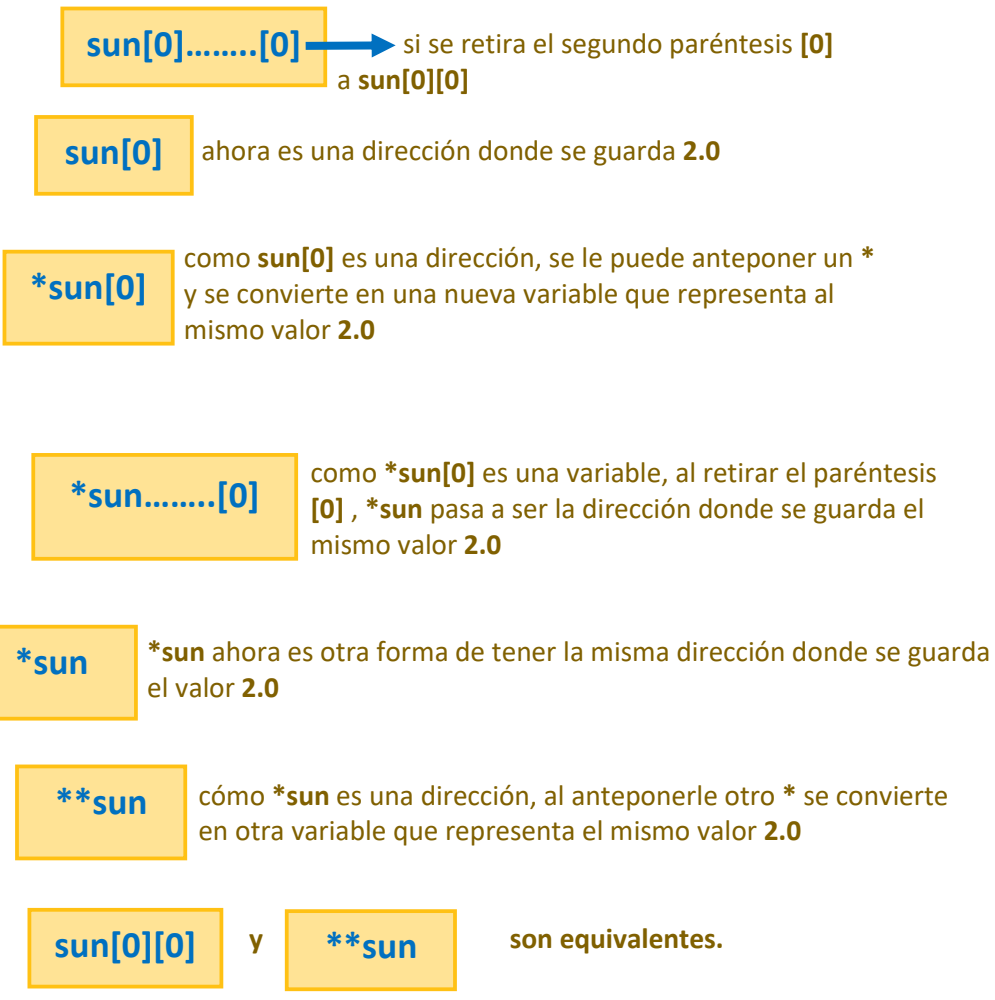
Como la expresión se evalúa de derecha a izquierda, en este caso, no se necesita usar paréntesis redondo.

Repitiendo el mismo análisis, si *sun[0] es una variable que posee un paréntesis [ ] a su derecha, entonces *sun debe ser también una dirección.
Luego, aplicando un * a la izquierda de *sun podemos imprimir el valor que se encuentra en la dirección *sun.

Así, printf("%f\n", **sun); // imprime el valor 2.0

Este mismo ejemplo lo podemos analizar gráficamente de la siguiente manera:

La variable sun[0][0] representa el primer valor del arreglo 2.0:



De esta forma, podemos concluir que el **nombre de un arreglo de dos dimensiones con asterisco a la izquierda es la dirección del primer valor del arreglo**. El nombre **sun** del arreglo, es la dirección de la dirección del primer valor del arreglo.

Para comprender mejor que es la dirección de la dirección de una variable, podemos ver gráficamente el siguiente ejemplo:

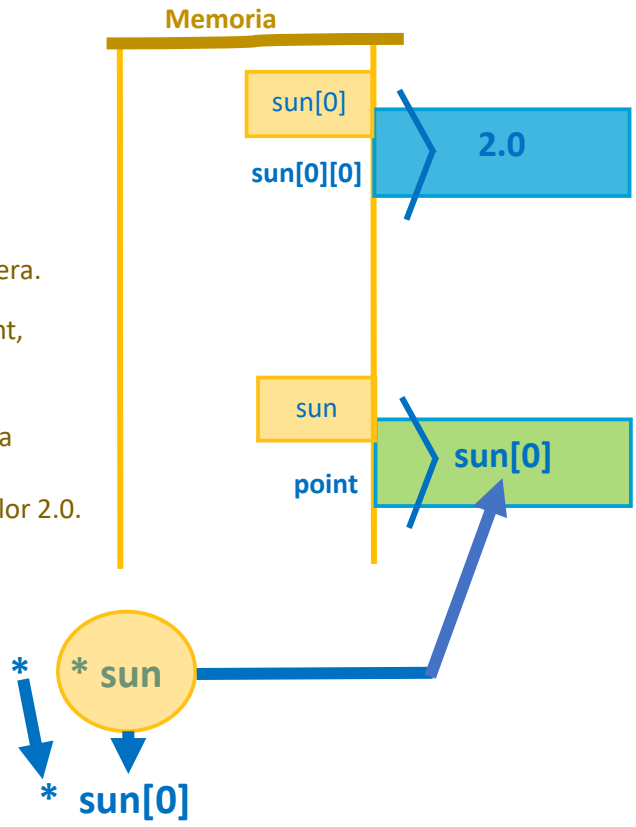
sun[0][0] es la variable que representa el primer valor 2.0 del arreglo.

sun[0] es la dirección del valor 2.0

point es una variable puntero que guarda la dirección **sun[0]**. **point** es un nombre cualquiera.

sun es la dirección donde vive el puntero point, esto es, donde se guarda la dirección **sun[0]**.

Si se aplica * a **sun** se obtiene sun[0], que es la dirección donde se guarda el valor 2.0.
Luego, si se aplica * a **sun[0]**, se obtiene el valor 2.0.



sun es la dirección donde se guarda **sun[0]** , que es la dirección donde se guarda el valor **2.0**.

Para el caso de un arreglo de tres dimensiones el análisis es similar, y rápidamente se llega a la conclusión que el nombre del arreglo representa la dirección de la dirección de la dirección del primer valor del arreglo. Es decir, ***(nombre arreglo) es una variable que representa el primer valor del arreglo.

Pregunta: ¿Es necesario, en la inicialización de un arreglo de dos o más dimensiones usar paréntesis curvos internos?

Ejemplo: a) `int hat[2][2][2] = {{{2,4},{9,7}},{{10,34},{45,89}}};`

En efecto, también es válido: b) `int hat[2][2][2] = {2,4,9,7,10,34,45,89};`

La diferencia de a) con b) sin paréntesis { } internos, es que es más difícil ver; por ejemplo, cuál es el valor que representa la variable hat[0][1][1]. En la forma a) se ve rápidamente que el valor es 7. Ambas formas son funcionalmente iguales.

3.5 Tamaño de un arreglo.

El tamaño de un arreglo se puede determinar usando la función **sizeof()** de la biblioteca estándar del lenguaje C.
La función sizeof() entrega el número total de bytes, por lo cual, para tener el número de datos o valores en un arreglo, el resultado de sizeof() debe dividirse por el tamaño del Tipo al que pertenecen los valores o datos del arreglo.

Ejemplo: `float sun[2][3];`

```
int numeroElementosArreglo = sizeof(sun)/sizeof(float);  // se asigna 6 que es el tamaño del arreglo
```

3.6 Declaración de arreglos con paréntesis vacíos.

Todo arreglo puede dejar el primer paréntesis [ ] vacío, siempre y cuando se inicialice correctamente.

Ejemplos:

a) `double gama[ ] = {2.3,5.6,9.0,2.5}; //arreglo de 5 valores`

En a), el tamaño del arreglo lo define el número de valores en la inicialización.

b) `int beta[ ][3] = {{2,4,6},{6,8,1}}; // arreglo de 6 valores`

En b), la dimensión del primer paréntesis es deducida automáticamente igual a 2. No importa si en la inicialización se usan o no paréntesis { } internos.

c) `float delta[ ][2][3] = {2.0,4.5,3.4,1.0,5.6,3.0,7.0,3.4,2.3,2.3,8.8,5.0}; // 12 valores`

En c), la dimensión del primer paréntesis [ ] es deducida automáticamente igual a 2, considerando el número total de valores en la inicialización y la información de la segunda y tercera dimensión.

3.7 Arreglos de caracteres.

En lenguaje C, un mensaje se describe por caracteres que conforman una o más palabras, encerradas en cremillas.
Ejemplo: “Planetas Habitables” que se puede asignar tanto a un arreglo como a un puntero a caracteres. Las siguientes sentencias permiten guardar este mensaje en memoria del computador.

```
char  mensaje[ ] = {“Planetas Habitables”};
```

Aquí el nombre del arreglo apunta al primer carácter del arreglo, esto es a la letra P. El último carácter de este arreglo no es la letra s , si no un carácter que se escribe ‘\0’ y que representa un dígito cero. Este cero se usa para determinar el fin del mensaje. Se adiciona automáticamente al mensaje al encerrarlo en “ “.
También, se puede usar un puntero a carácter para guardar este mensaje en memoria.

```
char*   pMensaje = “Planetas Habitables”;
```

Sin embargo, en este caso ningún carácter de **pMensaje** se puede modificar. El sistema operativo coloca **pMensaje** en una parte de la memoria que sólo se puede leer.

Para imprimir el mensaje se usa el formato s que requiere se entregue la dirección de primer carácter del mensaje. Termina de imprimir cuando encuentra el carácter '\0'.

Ejemplo: printf("%s\n", pMensaje);

printf(%s\n", mensaje);

En ambos casos, pMensaje y mensaje son direcciones que apuntan al primer carácter del mensaje, como lo requiere el formato s del printf().

Cada carácter del arreglo mensaje se puede obtener usando paréntesis []. El primer carácter se representa por la variable mensaje[0], el segundo por mensaje[1] y así sucesivamente.

Ejemplo: printf("%c\n", mensaje[3]); // imprime la letra n del mensaje Planetas Habitables
Se usa el formato c para imprimir un carácter o letra.
Ciertamente, se pueden usar punteros para imprimir cualquier letra del mensaje.

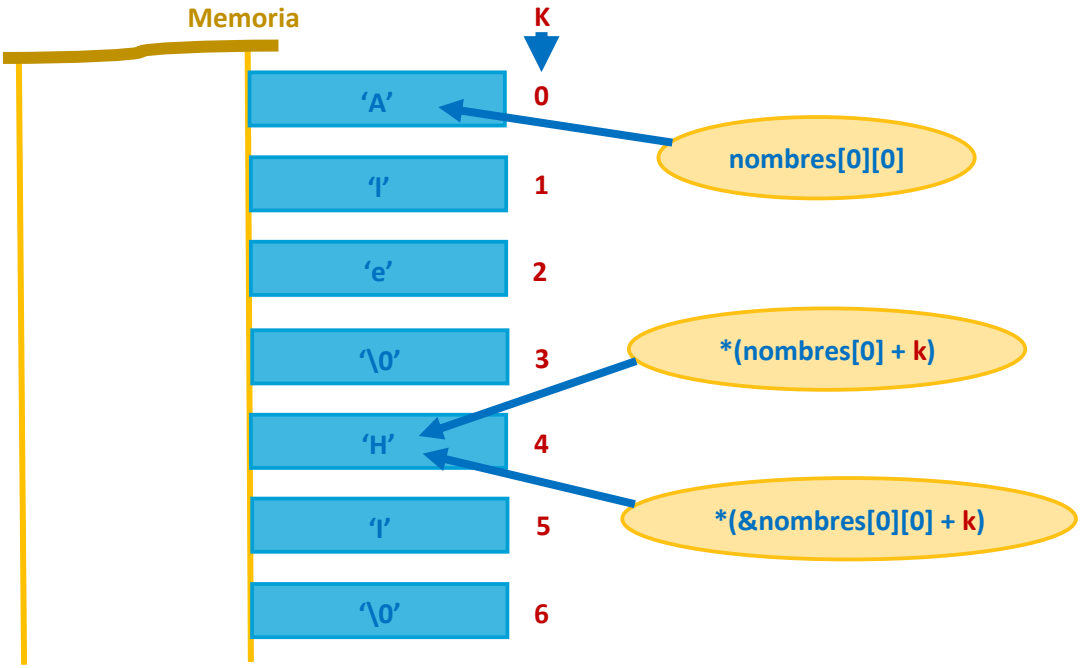
Ejemplo: printf("%c\n", *(mensaje + 3)); // imprime la letra n del mismo mensaje

Las variables mensaje[3] y *(mensaje + 3) representan al mismo carácter guardado en memoria.

Un arreglo de dos dimensiones puede guardar varios mensajes en memoria. Cada mensaje debe tener un máximo de caracteres considerando el carácter de término '\0'. Este máximo esta dado por la dimensión del segundo paréntesis [ ] en el arreglo. El primer paréntesis puede tener o no un valor constante. En caso de no tenerlo se deduce automáticamente.

Ejemplo: char nombres[2][4] = {{"Ale"},{"Hi"}};

¿Cómo se guardan en memoria estos nombres?
En nuestra analogía de memoria podemos visualizar lo siguiente:



&nombres[0][0] es la dirección del primer carácter del arreglo

&nombres[0][0] + k es la dirección de cualquier carácter del arreglo, dependiendo del índice **k**

***(&nombres[0][0] + k)** representa a cualquier carácter del arreglo según sea el índice k.

También, **nombres[0]** es la dirección del primer carácter, esto es la letra **A** y **nombres[1]** es la dirección del primer carácter del segundo mensaje **"Hi",** esto es la letra **H**.

K varía entre 0 y 7. Un total de 8 caracteres en memoria incluyendo los caracteres '\0'. El carácter cuyo k es 7 no se muestra en la figura, pero es '\0'.

3.7.1 Arreglos de punteros a caracteres.

Un arreglo puede tener como datos o elementos a cualquier Tipo permitido por el lenguaje C. Los punteros son variables que sólo pueden guardar una dirección y ciertamente pueden ser datos de un arreglo.
¿Qué sucede si en el ejemplo anterior los nombres son muy disímiles entre sí en cuanto a su largo de caracteres?. Entonces, el arreglo con un valor fijo para la segunda dimensión parece inadecuado, puesto que varias posiciones de memorias no se utilizarán.

Una manera de hacer más eficiente el almacenamiento de nombres es usar un arreglo de punteros a caracteres.

Ejemplo:

```
char nombre1[10] = {"Alejandro"};
char nombre2[9] = {"Fernanda"};
char nombre3[5] = {"Juan"};
char nombre4[12] = {"Maximiliano"};

char*   Nombres[4] = {nombre1,nombre2,nombre3,nombre4};
```

Ahora, Nombres es un arreglo de 4 punteros que son nombre1, nombre2, nombre3 y nombre4. Cada uno de ellos es una dirección que apunta al primer carácter de su arreglo. Las dimensiones de los arreglos que guardan los nombres son distintas y exactas para el tamaño del nombre que representan, permitiendo un uso más eficiente de la memoria.

¿Cómo se imprime el nombre Maximiliano usando el arreglo Nombres?

```
printf("%s\n", Nombres[3]); // imprime Maximiliano
```

¿Cómo se imprime la letra a del nombre Juan? Puede hacerse de dos formas:

```
printf("%c\n", Nombres[2][2]);     // imprime la letra a de Juan usando []

printf("%c\n", *(Nombres[2] + 2));    //imprime la letra a de Juan usando *
```

Nombres[2] es la dirección **nombre3** y al adicionarle el **[2],** se convierte en la variable que representa el carácter o letra **a** de **Juan**.

Como **Nombres[2]** es la dirección que apunta a **Juan**, se puede sumar la posición de la letra **a** de **Juan** que es **2**, y aplicando ***** a la izquierda se crea la variable ***(Nombres[2] + 2)** que también representa la letra a de **Juan**.

3.7.2 Arreglos de dimensión variable.

Lenguaje C en sus versiones C99 y C11, permiten que un arreglo tenga un tamaño dado por una variable del tipo entero.

Ejemplo:

```
int n = 4;

int  alfa[n]; // permitido en C99 y C11
```

El espacio en memoria para el arreglo se calcula en tiempo de ejecución y no en tiempo de compilación, como ocurre cuando el tamaño es un valor constante. De esta forma, es posible ingresar por teclado el valor del tamaño del arreglo. Sin embargo, cuando se usa una variable, no es posible inicializar el arreglo.

3.8 Asignación dinámica de memoria.

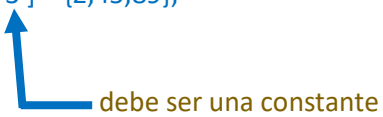
Cuando se declara una variable en lenguaje C, el tamaño de la memoria u objeto de dicha variable queda establecido durante la compilación del programa fuente. El tamaño del objeto lo decide el Tipo que se utiliza para declarar la variable.
Cuando se declara un arreglo con el debido número de elementos por dimensión, el tamaño de la memoria necesario para almacenar todos los elementos del arreglo queda también definido en tiempo de compilación. El tamaño de la memoria se decide por el Tipo de un elemento y el número total de elementos en el arreglo. Esto queda definido por toda la existencia del programa.

¿Cómo entonces, se puede definir el tamaño de un arreglo, no en tiempo de compilación, si no en tiempo de ejecución del programa ("run-time")?

Un arreglo obliga a usar un valor constante para el número de elementos de una dimensión. Esto no permite modificar en tiempo de ejecución el tamaño del arreglo. Sin embargo, a partir de la versión C99 y posteriores del lenguaje C, se permite usar una variable para determinar el número de elementos en la dimensión de un arreglo. El valor de la variable se puede determinar en tiempo de ejecución y en consecuencia, el tamaño del arreglo. Pero una vez asignado en tiempo de ejecución el valor a la variable de la dimensión del arreglo, éste fija de manera indefinida (por toda la duración del programa) y no modificable su tamaño.
Ejemplos:

1.- Para versiones anteriores a C99 del lenguaje C.

```
int   arr[ 3 ] = {2,45,89};
```



2.- Para la versión C99 o posteriores.
El ejemplo anterior es válido, pero ahora es posible lo que se muestra en el siguiente ejemplo:

```
int alfa ;    // se ingresa por teclado el valor de alfa en tiempo de ejecución
int arr[ alfa];
```

ahora puede ser una variable

Como el valor de alfa se determina en tiempo de ejecución; en este caso se ingresa por teclado. El arreglo no puede ser inicializado al momento de declararlo. Hay que esperar al tiempo de ejecución para asignar valores a cada elemento del arreglo, una vez determinado alfa.
La primera vez que se determina el valor de alfa en tiempo de ejecución, se define por toda la duración del programa el tamaño del arreglo y éste no puede volver a ser modificado.

Entonces surge la siguiente pregunta: ¿Cómo se puede asignar un tamaño dinámico de memoria para un arreglo en tiempo de ejecución? Lo que se desea, es tener tamaños de memoria modificables en tiempo de ejecución que puedan ser utilizados como un arreglo.

Para poder hacer lo anterior, el lenguaje C proporciona dos funciones de la biblioteca estándar <sdtlib.h>, las cuales son **malloc()** y **realloc()**.

3.8.1 malloc().

Aunque aún no se ha visto funciones, es suficiente comprender lo que hace malloc() desde un punto de vista funcional. La sintaxis de malloc() es la siguiente:

malloc( )

Aquí se coloca el tamaño de la memoria en número de bytes que se quieren reservar

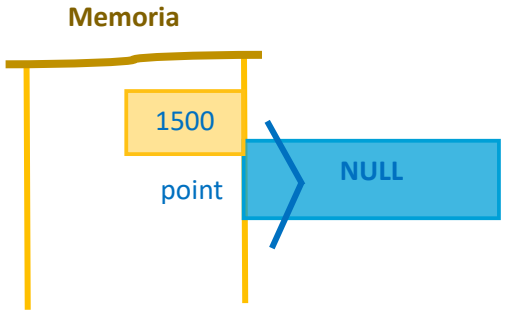
Para calcular el número de bytes es necesario tener el **Tipo** de los datos a guardar en la memoria. Por ejemplo, si los datos a almacenar son enteros, entonces son del tipo **int** y cada entero necesita 4 bytes. Es mejor usar la función **sizeof()** para calcular el número de bytes de cualquier **Tipo**. Para este caso, **sizeof(int)** nos da un valor de 4. Si se requieren almacenar 35 enteros en tiempo de ejecución, se usa entonces **35* sizeof(int)** para indicar el tamaño de la memoria deseada en **malloc()**.

Ejemplo: Se desea tener un arreglo de 5 enteros en tiempo de ejecución de un programa.

a) Se declara un puntero a entero, inicializado con NULL antes de la ejecución.
b) Se usa malloc() para buscar en memoria 5 casas azules o más contiguas, que estén desocupadas y con el tamaño adecuado para almacenar 5 enteros.
c) malloc() entrega la dirección de la primera casa al puntero antes creado.

d) El puntero se usa para tener acceso a cualquier valor del arreglo (memoria reservada) usando las reglas ya conocidas para usar punteros en un arreglo.

```
int*  point = NULL;
```



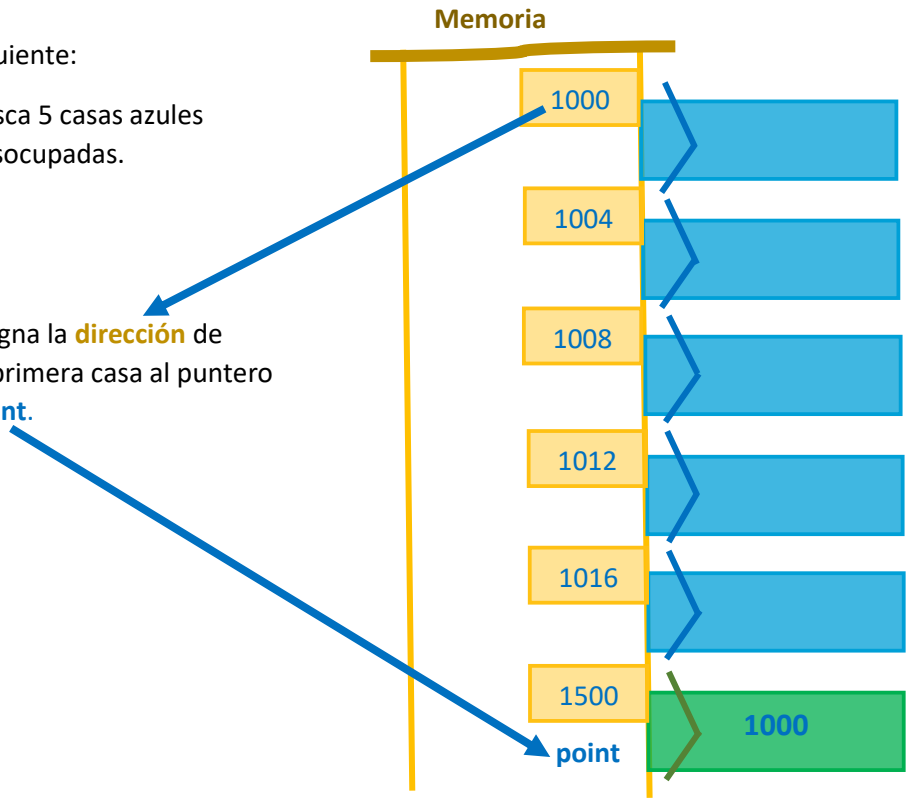
Al ejecutarse la sentencia anterior, se crea un puntero con el nombre **point** en la casa que posee la dirección 1500 en la memoria. El puntero **point** inicialmente representa a un valor NULL.

```
point = malloc( sizeof(int)*5);
```

Hace lo siguiente:

a) Busca 5 casas azules desocupadas.

b) Asigna la **dirección** de la primera casa al puntero **point**.



Las casas azules en nuestra analogía con la memoria física del computador tienen el tamaño de 4 bytes, necesarios para contener un número entero.
La función malloc() una vez que encuentra 5 casas contiguas disponibles para ser ocupadas, toma la dirección de la primera casa y donde vive la variable **point** reemplaza el valor NULL por **1000**. Ahora el puntero point queda apuntando a la primera casa ubicada en la dirección 1000 de la memoria.

Con el puntero point se puede tener acceso al valor de cualquiera de las 5 casas que constituyen un nuevo arreglo creado dinámicamente en tiempo de ejecución del programa.

Se puede usar * o [] para asignar valores a cualquier elemento (casa) del arreglo.
Ejemplos:

```
point[0] = 45; //asigna el valor 45 al primer elemento del arreglo.

*(point + 3) = 23; //asigna el valor 23 al cuarto elemento del arreglo.
```

point[0] es el nombre del objeto o primera casa, en adelante la variable que representa el valor 45 del primer elemento del arreglo. Esta variable también puede tomar la forma *point, la cual es equivalente a point[0].

*(point + 3) es el nombre del objeto o cuarta casa, en adelante la variable que representa el valor 23 del cuarto elemento del arreglo. También esta variable puede tomar la forma point[3] que es equivalente a *(point + 3).

Al usar malloc() para reservar memoria, las casas nunca se inicializan con cero, si no que tienen cualquier valor que tenían antes de ser asignadas al puntero como un bloque de memoria.

Importante.
La función malloc() cuando busca espacio desocupado en la memoria, busca un bloque completo del tamaño total solicitado y devuelve la dirección del comienzo del bloque encontrado. En el caso del ejemplo, busca un bloque de tamaño 5*sizeof(int) que es igual a 5*4 = 20 bytes. malloc() no reconoce memoria para enteros, dobles (en nuestra analogía casas azules, verdes) etc. La dirección que retorna, la dirección del comienzo del bloque de memoria es genérica y se puede asignar a cualquier Tipo del lenguaje C.
En nuestro ejemplo, se asigna a un puntero point declarado del **Tipo entero**, y es el **Tipo** del puntero el que reconoce el bloque total de memoria como sectores contiguos que pueden almacenar números enteros.
Sin embargo, el lenguaje C++ obliga a hacer un casting a la dirección genérica que retorna malloc(), para convertir dicha dirección en el Tipo del puntero al cual se asigna.
Esto no es necesario en lenguaje C.
Nuestra analogía de casas de distinto tamaño para los Tipos del lenguaje C, es funcionalmente correcta y nos ayuda a comprender la asignación dinámica de memoria sin entrar en los detalles antes explicados.

3.8.2 realloc()

La función realloc() permite, una vez usada malloc(), reducir o ampliar el bloque de memoria asignado a un puntero. En el ejemplo anterior, usando el puntero point y realloc(), es posible disminuir o aumentar el número de casas azules. Al usar realloc() pueden ocurrir los siguientes casos:

Caso a):

```
point = realloc(point, 3*sizeof(int); //  se disminuye el número de casas a 3
```

La función realloc() ocupa dentro de su paréntesis, primero una dirección de un bloque de memoria previamente encontrado por malloc() (esto es obligatorio) y segundo, separado por una coma, el nuevo tamaño que se desea. El resultado de realloc() es indefinido si como dirección dentro del paréntesis, se usa una dirección no asignada por la función malloc().
En este ejemplo, el tamaño cambia de 5 casas a 3 casas. realloc() mantiene el bloque de memoria donde está actualmente y sólo elimina las dos últimas casas. Entonces el puntero point recibe desde realloc() la misma dirección que tenía.
La función realloc(), al igual que malloc(), retorna la dirección del comienzo del nuevo bloque de memoria y la asigna al puntero que se quiere usar para manejar el bloque. En el caso del ejemplo, el puntero es point, previamente asignado por malloc() para las 5 casas azules.

Caso b):

```
point = realloc(point, 500*sizeof(int); //  se aumenta el número de casas a 500
```

En este caso, realloc() primero indaga si hay suficiente espacio, a continuación de las 5 casas, para agregar 495 casas y así completar un bloque de 500 casas. Si esto es posible, entonces retorna la misma dirección de comienzo de bloque que tenía, al puntero point.
Si no es posible, realloc() busca en otra parte de la memoria la disponibilidad de tener 500 casas azules desocupadas. Si las encuentra, copia los datos que guardan actualmente las 5 casas, a las nuevas 5 primeras casas del nuevo bloque de 500 casas azules. Por estar este nuevo bloque, en otra parte de la memoria, realloc() asigna una nueva dirección, que corresponde a la dirección de la nueva primera casa, al puntero point. Ahora, point cambia su contenido.
Se debe tener el cuidado que, si otros punteros estaban apuntando al bloque de las 5 casas antiguas, estos deben actualizarse con la nueva dirección.

Casos especiales:

```
point = realloc(NULL, cualquier tamaño); // se comporta como malloc()

point = realloc(cualquier puntero, 0); // libera la memoria apuntada por el puntero
```

La función realloc() nunca inicializa en cero las casas añadidas.

3.8.3 calloc()

La función calloc() es similar a malloc(), con la diferencia que inicializa todas las casas encontradas con cero. Su sintaxis es distinta a malloc() y es la siguiente:

```
puntero = calloc(n, tamaño de cada n en bytes);
```

donde: n es el número de valores o datos y después se coloca el tamaño de cada dato.
Ejemplo:

```
point = calloc( 5, sizeof(int)); // reserva memoria para 5 enteros
  //inicializados en cero. point apunta al primer entero
```

3.8.4 Declaración e inicialización de un puntero con malloc()

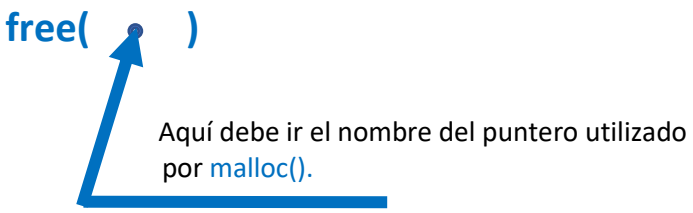
Un puntero se puede declarar e inicializar en una sola sentencia con malloc(). Es decir, se crea el puntero y se reserva de inmediato memoria para él en tiempo de ejecución.
Ejemplo:

```
int alfa = 26;

int* point = malloc(alfa*sizeof(int));
```

En este ejemplo se crea el puntero point que apunta al primer entero de un bloque de memoria que contiene espacio para 26 enteros.

3.8.5 Liberación de la memoria reservada por malloc()

Cada vez que se reserva memoria usando la función malloc() y se asigna la dirección de comienzo del bloque de memoria a un puntero, ésta debe ser liberada cuando ya no se usa más en un programa. El no hacerlo, dejará el bloque de memoria sin poder ocuparlo para otros efectos, aun cuando el programa hubiese terminado.
Para liberar un bloque de memoria reservada por malloc(), se usa la función **free().**
La sintaxis de free es la siguiente:



Es un error, con consecuencias impredecibles, volver a usar free() con un puntero con el cual ya se ha utilizado la función free().

Un puntero, al cual se le ha aplicado free(), se le debe asignar NULL. Esto no es necesario, si free() se aplicó justo antes de terminar el programa.

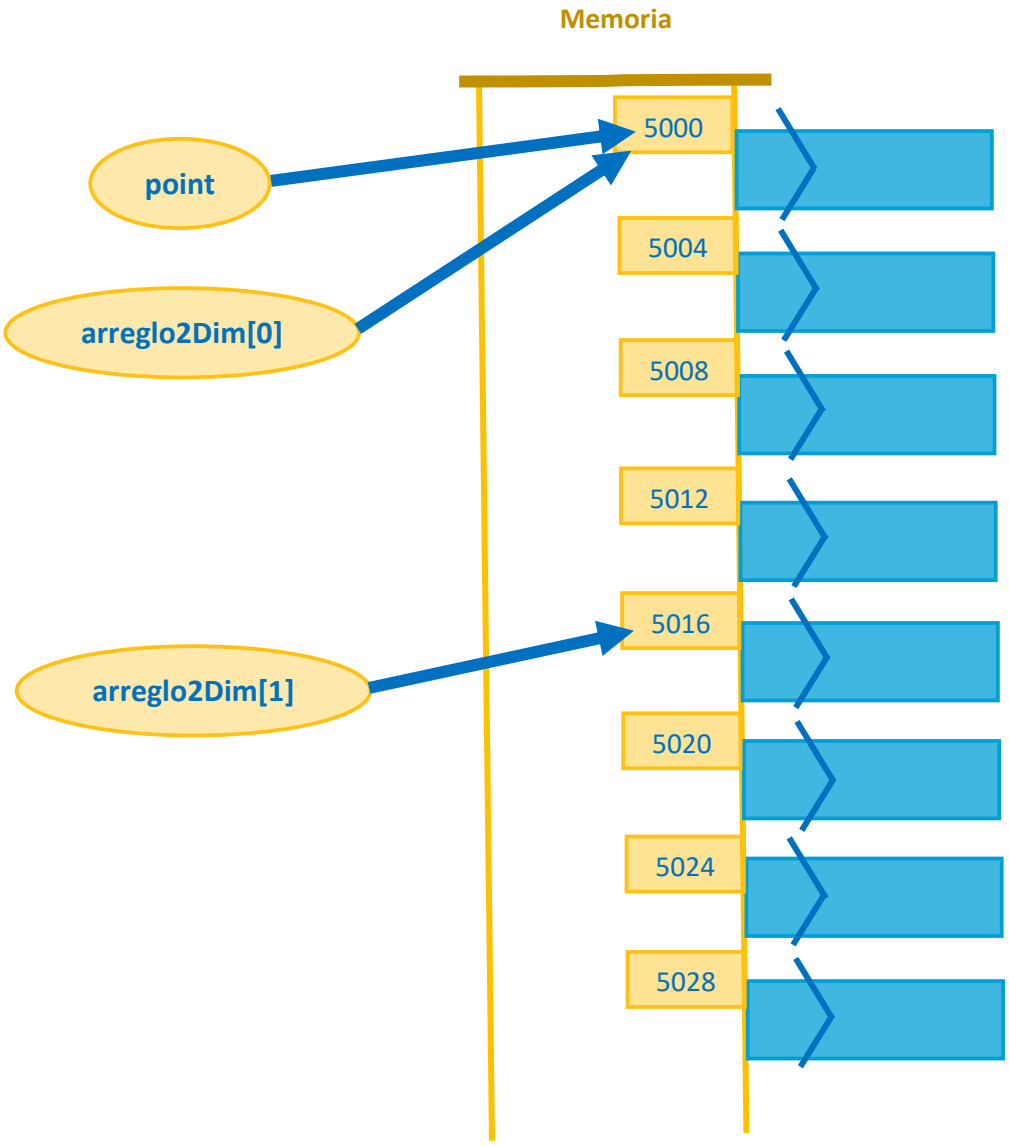
3.8.6 Creación dinámica de arreglos de dos o más dimensiones.

Cuando se reserva memoria con la función malloc(), se reserva un bloque de memoria de acuerdo al tamaño solicitado por malloc() y por lo tanto, este bloque puede adaptarse para ser usado como un arreglo de una, dos , tres etc. dimensiones.
En el siguiente ejemplo se describe como usar un bloque de memoria reservada por malloc() para un arreglo de dos dimensiones. Es muy conveniente revisar el tema 2.16 del capítulo 2.

Ejemplo 1: Crear dinámicamente, en tiempo de ejecución, un arreglo con dimensiones **[2][4]**, para enteros.

El tamaño en bytes para reservar memoria con malloc() para este caso, es de 2*4*sizeof(int) , lo que es igual a 32 bytes u ocho casas azules, que tienen el tamaño para contener números enteros.

```
int* point = malloc(8*sizeof(int)); // entrega a point la primera dirección del bloque
```



Lo que se desea es tener un solo puntero, que al aplicar dos paréntesis cuadrados se tenga acceso a cualquier valor del bloque de memoria, en la manera que se hace con un arreglo de dos dimensiones.
Las cuatro primeras casas forman el bloque para **[0][ ]** en el arreglo , y las cuatro siguientes forman el bloque **[1][ ]** para el mismo arreglo. Por el hecho de tener dos paréntesis cuadrados, se necesita crear un puntero a puntero.
Sea el nombre del puntero a puntero **arreglo2Dim**.
La dirección de comienzo del primer bloque de casas debe estar dada por **arreglo2Dim [0]** y la dirección de comienzo del segundo bloque de casas por **arreglo2Dim [1]** , siendo **arreglo2Dim** el nombre cualquiera de un puntero a puntero a crear.
Para crear un puntero a puntero se necesitan dos direcciones válidas. La primera la entrega el puntero **point**. La segunda se obtiene con cualquier puntero a entero y se asigna **point** a este puntero.

```
int* pointD = point; // pointD es un nombre cualquiera

int** arreglo2Dim = &pointD; // &pointD es la segunda dirección donde se guarda
                             // la primera dirección point, que es donde está el
                             // valor entero del arreglo de esa posición.
arreglo2Dim[0] = point;
arreglo2Dim[1] = point + 4;
```

Luego:

arreglo2Dim[0][ ] puede tener acceso a cualquier valor de las cuatro primeras casas
arreglo2Dim[1][ ] puede tener acceso a cualquier valor del segundo bloque de casas

Usando **arreglo2Dim[ ][ ]** , se puede tener acceso a cualquier valor del arreglo de dos dimensiones.

Ahora se tiene un arreglo de dos dimensiones de 2x4 con el nombre **arreglo2Dim**, que se puede trabajar con la misma nomenclatura de un arreglo creado en tiempo de compilación.

Se debe tener cuidado con el uso de los índices puesto que, el lenguaje C no comprueba en tiempo de ejecución si estos se están usando fuera de su rango. El uso de índices fuera del rango de validez puede traer serios problemas al funcionamiento del programa.

Sin embargo, la forma recién descrita para crear un arreglo dinámico de dos dimensiones sólo funciona si la primera dimensión tiene un tamaño no mayor a 2.

Para crear de manera dinámica un arreglo de dos dimensiones con cualquier tamaño de ambas dimensiones, se debe hacer lo siguiente:

```
int mem;  // guarda tamaño de arreglo
// ingresar ahora tamaño de memoria para el arreglo por teclado o programa.
int*       beta = malloc(sizeof(int)*mem);    // reserva memoria para arreglo.
int n1=1, n2=1;  // n1 guarda el tamaño de la primera dimensión
                 // n2 guarda el tamaño de la segunda dimensión
// Ingresar ahora valores para n1 y n2, mediante teclado o por programa.

int* arr2[n1];
        for(int i = 0; i < n1; i++)
        {
                arr2[i] = beta + i*n2;
        }
```

Ahora se puede usar **arr2[][]** para tener acceso a cualquier valor del arreglo.
Ver problema resuelto 5.15.9 en el capítulo 5.

No olvidar usar antes del return 0 en el main(), la sentencia **free(point)** para el primer caso y **free(beta)** para el segundo caso.

Ejemplo 2. Crear un arreglo de manera dinámica con **tres** dimensiones.

La metodología es igual al caso del ejemplo 1, con la excepción que ahora se necesita un puntero a puntero a puntero, esto es **int*** ** . Veremos a continuación como se crea este puntero.
El arreglo que se desea tener en tiempo de ejecución ("run-time") tiene las dimensiones [2][2][3].
Luego, necesitamos malloc() de la siguiente forma:

```
int* point = malloc(2*2*3*sizeof(int)); // bloque de memoria de 48 bytes
```

Se necesitan dos direcciones adicionales que se crean con dos punteros de la siguiente manera:

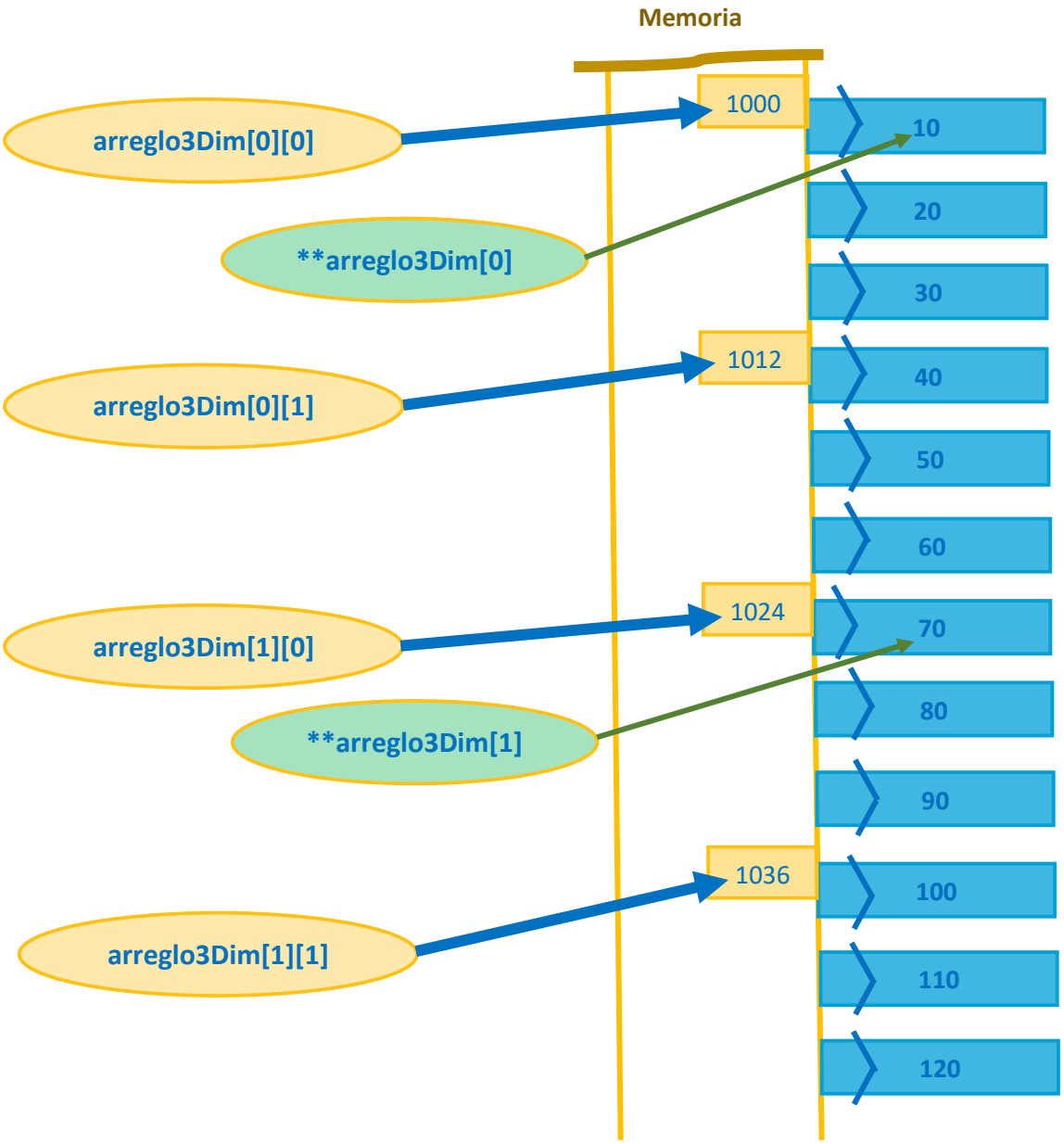
```
int* dir1 = point;
int** dir2 = &dir1;
```

y el puntero a puntero a puntero:

```
int*** arreglo3Dim = &dir2;

arreglo3Dim[0] = &point; //dirección donde vive point
arreglo3Dim[1] = arreglo3Dim[0] + 6; //al aplicar [ ] a esto, resulta en la
                                     //dirección de comienzo del segundo bloque de datos
arreglo3Dim[0][0] = point;
arreglo3Dim[0][1] = point + 3;
arreglo3Dim[1][0] = point + 6;
arreglo3Dim[1][1] = point + 9;
```

Ahora, se puede utilizar la nomenclatura normal de un arreglo de tres dimensiones. Para tener acceso a cualquier valor del arreglo se usa arreglo3Dim[i][j][k], donde i es 0 o 1, j es 0 o 1, k es 0,1,2 (de acuerdo a las dimensiones) y **arreglo3Dim** pasa a ser el nombre del arreglo.

En el siguiente gráfico se muestran las ubicaciones de todas las direcciones anteriores:



arreglo3Dim[0][0] es la dirección del valor 10 del arreglo.
arreglo3Dim[0][1] es la dirección del valor 40 del arreglo.
arreglo3Dim[1][0] es la dirección del valor 70 del arreglo.
arreglo3Dim[1][1] es la dirección del valor 100 del arreglo.

arreglo3Dim[0] es una variable que representa al valor 1 del arreglo. **¿Porqué?
arreglo3Dim[1] es una variable que representa al valor 7 del arreglo. **¿Porqué?

Al igual que en caso de dos dimensiones, la forma recién descrita para crear un arreglo dinámico de tres dimensiones sólo funciona si la primera dimensión tiene un tamaño no mayor a 2.

Para crear de manera dinámica un arreglo de tres dimensiones, con cualquier tamaño para cada dimensión, se debe hacer lo siguiente:

```
int mem;  // guarda tamaño de arreglo
// ingresar ahora tamaño de memoria para el arreglo por teclado o programa.
int*       beta = malloc(sizeof(int)*mem);     // reserva memoria para arreglo.
int n1=1, n2=1, n3 =1;    // n1 guarda el tamaño de la primera dimensión
                          // n2 guarda el tamaño de la segunda dimensión
                          // n3 guarda el tamaño de la tercera dimensión
// Ingresar ahora valores para n1, n2 y n3, mediante teclado o por programa.

int**     arr3D[n1];      // crea un doble puntero necesario para tener 3 dimensiones
          int m = 0;
          for(int i = 0; i < n1; i++)  // los for anidados extienden el caso anterior para cualquier tamaño
          {                                        //de las tres dimensiones ¿por qué?.
                    arr3D[i] = &beta + i*n2*n3;
                    for(int k = 0; k < n2; k++)
                    {
                              arr3D[i][k] = beta + m;
                              m = m + n3;
                    }
          }
```

Ahora se puede usar **arr3D[][][]** para tener acceso a cualquier valor del arreglo.
Ver problema resuelto 5.15.10 en el capítulo 5.

4 EXPRESIONES, BUCLES, SENTENCIAS DE SELECCIÓN, VARIABLES ESTÁTICAS y scanf.

Un matemático brillante es aquel que es capaz de ver analogías entre teoremas.
Un matemático genio, es aquel que es capaz de ver analogías entre analogías.
Stanislaw Ulam (1909-1984)

4.1 Expresiones

En lenguaje C, una expresión consiste en operandos y operadores. Un operando es un objeto que puede ser manipulado y ser algo tan simple como una variable o un valor constante. Una expresión básica puede ser una sola variable o una constante. Los operadores son símbolos que ejecutan una acción sobre los operandos. Una fórmula es una expresión que consta de variables, constantes y operadores que actúan sobre estas. Ejemplo: x + 2.0 es una expresión donde x es una variable y 2.0 un valor constante. El símbolo + es un operador aritmético que suma la variable x al valor constante.
Lenguaje C es generoso, tanto en la cantidad como en la función de los operadores que proporciona. Una clasificación y listado de operadores se proporciona en 2.12 del capítulo 2.

4.1.1 Expresiones con Operadores Relacionales.

Los operadores relacionales se indican en 2.12.2
Ejemplos:

```
int op1=10, op2 = 34, op3 = 4;
float op5 = 23.4, op6= 7.0;

printf("%d\n", op1 < op2);       // imprime 1, porque op1 es menor que op2: Verdadero
printf("%d\n", op5 == op3);      // imprime 0, porque op5 es distinto de op3: Falso
```

Cada vez que una expresión con operadores relacionales se evalúa como verdadera, toda la expresión se convierte en la constante numérica 1. Caso contrario, es 0.

La expresión `op1 < op3 >op5 <= op1` los operadores relacionales se resuelven de izquierda a derecha.

`( ( op1 < op3 ) > op5 ) <= op1` todo igual a 1

```
printf("%d\n", op1<op3>op4<=op1); // imprime 1
```

La expresión op1 > op3 == op5 < op1 != op2 de acuerdo a 2.12.6 se evalúa como sigue:

((op1 > op3) == (op5 < op1)) != op2 todo igual a 1

4.1.2 Expresiones con Operadores Lógicos.

Los operadores lógicos se indican en 2.12.3

Ejemplos:

```
int x = 2, y = 45, z = 3;
double f = 23.5, g = 4.5;
```

La expresión x < y && f > z de acuerdo a la precedencia indicada en 2.12.6 la expresión se evalúa:

(x < y) && (f > z)

1 && 1

1

La expresión z > y && f <= y || 5 se resuelve de la siguiente manera:

((z > y) && (f <= y)) || 5

0 && 0

0 5

1

```
printf("%d\n", z > y && f <= y || 5 );  // imprime 1
```

4.1.3 Expresiones con Operadores de Incremento y Decremento.

El operador de incremento ++ y el de decremento – se aplican directamente a variables enteras. Es posible aplicarlos a variables flotantes, pero es raro hacerlo.

Ejemplos:

```
int x = 20;

x++;      // cambia x a 21
++x;     // cambia x a 21

--x;      // cambia x a 19
x--;     // cambia x a 19
```

Lenguaje C no asegura el orden en que se ejecuta ++i ó i++ , y por lo tanto hay que tener cuidado al usar el incremento o decremento. Por ejemplo:

```
int i = 5;
i = i++;
```

puede entregar el resultado 5 o 6, dependiendo del compilador. En este caso es preferible usar,

```
i = i +1;  // en vez de i = i++;
```

También, usar incremento en una variable que se utiliza tanto al lado izquierdo y derecho del símbolo de asignación genera un resultado impredecible. Ejemplo:

```
beta[i] = i++;  // resultado impredecible en C
```

4.1.4 Expresiones con operadores aritméticos.

En lenguaje C, las conversiones implícitas (2.12.5) pueden inducir a error si no se tiene precaución.
Ejemplo:

```
float alfa, b = 1.2;
alfa = (7/9 ) * b;
printf("%f\n", alfa);   // imprime 0.0
```

Se asume que hay una conversión implícita de los operandos a la derecha del signo de asignación a float, porque la variable b es de tipo float. Sin embargo, esto no ocurre con (7/9), operación que se realiza como entero y su resultado es cero por ser un número menor que 1.
La manera de corregir esto, es mediante un casting al 7 o 9 a float o simplemente poner el 7 o el 9 en decimal. Esto obliga a que el resultado de la división se convierta a flotante.
Ejemplo:

```
float alfa, b = 1.2;
alfa = (7/9.0 ) * b;
printf("%f\n", alfa);   // imprime 0.93333
```

Otro ejemplo que puede inducir a error se muestra en el siguiente ejemplo:

```
int  gama = 100000;
int beta = 100000;
long int var = gama*beta; // gama*beta overflows
```

El resultado de var es erróneo porque gama*beta exceden el máximo número para un tipo int y no se convierte implícitamente a long int. La solución es hacer un casting como se indica:

```
int  gama = 100000;
int beta = 100000;
long long int var = (long long int)gama*beta; // gama*beta ahora OK.
```

El casting obliga a la variable gama a convertirse en un tipo long int y beta hace automáticamente lo mismo, cumpliendo ahora la regla de convertirse al operando mayor que hay al lado derecho del símbolo de asignación (2.12.5).

4.2 Bucles.

4.2.1 Sentencia for().

La sentencia for, permite ejecutar un número determinado de veces un grupo de sentencias que están en un bloque. Un bloque se construye usando paréntesis { } después del for para agrupar las sentencias. Es importante tener en cuenta que cualquier variable que se declare dentro de este bloque, tiene su existencia limitada a este bloque. La variable declarada dentro del for, no existe para efectos del resto del programa.

La sintaxis del for es la siguiente:

```
for(expresión 1; expresión 2; expresión 3)
{
        //grupo de sentencias
        int beta=3;
}
```

La variable beta, declarada dentro del for, tiene existencia sólo dentro del for.
Es ilegal hacer referencia a esta variable fuera del for.

En general, la expresión 1 es una variable que se inicializa en cero, pero no necesariamente debe comenzar con cero. La expresión 2 comprende a la variable de la expresión 1 que se compara con una constante u otra variable. En la expresión 3, la variable de la expresión 1 se incrementa en 1 u otro valor que se desee.

Ejemplo:

```
int i;
for(i = 0; i < 8; i++)
{
        // grupo de sentencias
}
```

Este for se ejecuta 8 veces. En la primera vez, i vale cero, se ejecutan las sentencias y luego i se incrementa en 1 usando la expresión 3. Cuando i es igual a 8, se sale del bloque, terminando el for. En este ejemplo, el grupo de sentencias se ejecuta 8 veces.

Es obligatorio en la versión C89 del lenguaje C, declarar la variable i antes del for. Sin embargo, en las versiones C99 y C11, la variable i se puede declarar directamente dentro del paréntesis del for.

Ejemplo:(versión C99 del C)

```
for(int i=2; i < 10 ; i = i + 2)
{
        // grupo de sentencias
}
```

En este ejemplo, i comienza la iteración con 2 y el for se ejecuta 5 veces, porque la variable i se incrementa de dos en dos.
Si la variable i se declara dentro del for, ésta es válida sólo para el for, pero si se declara antes del for, la variable i es válida para el for y para el resto del programa.

Otras formas válidas del for:

Ejemplo 1:

```
for(int k=2;  k < 10;  )
{
        // grupo de sentencias
        k++;
}
```

Ejemplo 2:

```
int k = 0;
for(  ;  k < 10;  )
{
        // grupo de sentencias
        k++;
}
```

Ejemplo 3:

```
for(int k=0 ;      ; k++)
{
        // grupo de sentencias
}
```

El for de los ejemplos 1 y 2 se ejecuta 10 veces. El for del ejemplo 3, se ejecuta hasta que el valor de k excede el valor máximo aceptado para el Tipo int del C. Al llegar a este punto, el SO coloca una excepción.

El siguiente for,

```
for(    ;     ;    )
{
        // grupo de sentencias
}
```

queda en un lazo infinito porque, en lenguaje C, se asume el for siempre verdadero cuando no está la expresión 2.

La expresión 2 puede ser cualquier expresión que al evaluarse sea Verdadera (distinta de cero) o Falsa (exactamente igual a cero).

Ejemplo:

```
for(int k = 4; k ; k--)
{
   printf("%d\n", k);
}
```

En este ejemplo, for se ejecuta 4 veces e imprime 4 3 2 1 y cuando k=0, se sale del for. Mientras k (expresión 2) sea distinta de cero, el for se ejecuta. Sale del for cuando k es igual a cero.

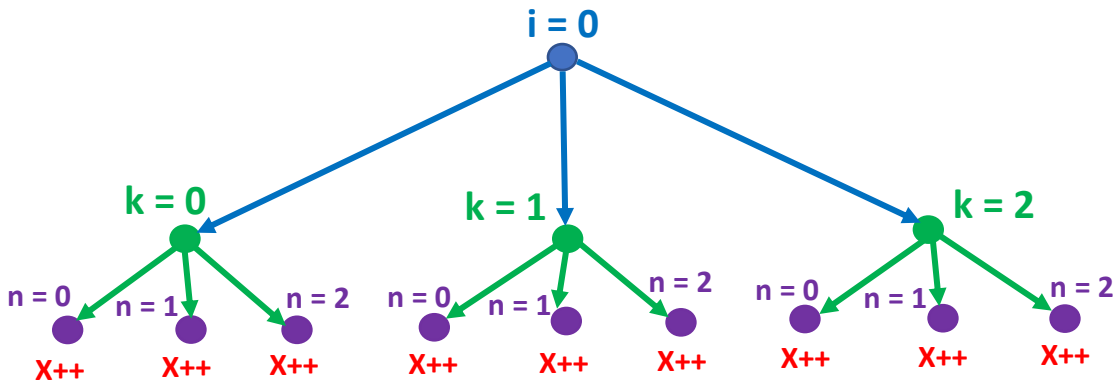
4.2.1.1 Bucles for() anidados

Un bucle for puede estar dentro de otro for, otro dentro de este último for y así sucesivamente. Como ejemplo se muestran tres for anidados:

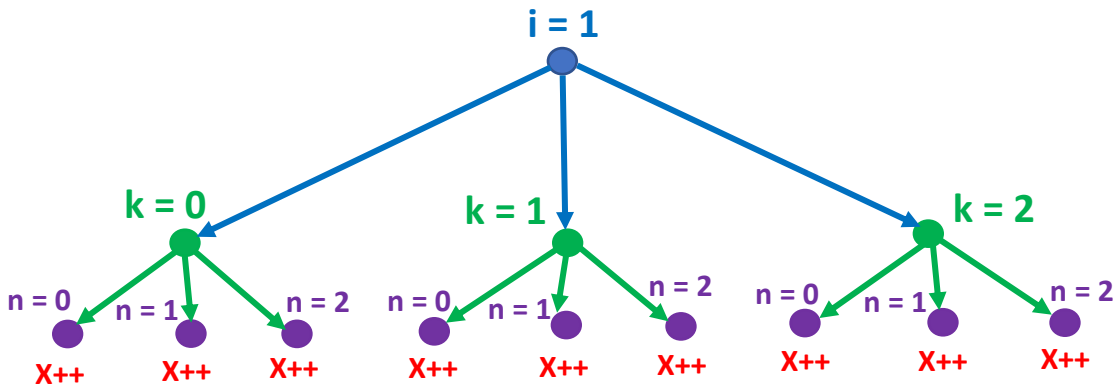
```
int x=0;
for(int i=0; i < 2; i++)
{
        for(int k=0; k < 3; k++)
        {
                for(int n=0; n < 3; n++)
                {
                        X++;
                }
        }
}
```

¿Cuántas veces se ejecuta la sentencia x++; ?

Para ver cómo funcionan estos for anidados se construyen los siguientes grafos:



Para i=0 el for interior (color verde) se ejecuta tres veces, para k=0, k=1 y k=2.
Para cada valor de k (en el for de color verde), el for interior a éste (color morado) se ejecuta tres veces, para n=0, n=1 y n=2. Para cada n, se ejecuta la sentencia x++. En total, x++ se ejecuta nueve veces solo para i=0.



Para i=1 la sentencia x++ se ejecuta también nueve veces.
En total, para estos tres for anidados, la sentencia x++ se ejecuta 18 veces. Con estos grafos es fácil ver la secuencia de ejecuciones de cada for.

4.2.2 Sentencia while.

La sentencia while permite ejecutar, de manera similar al for, un número determinado de veces un grupo de sentencias, mientras una expresión se evalúe como verdadera.
La sintaxis del while es la siguiente:

```
while( expresión)
{
   //grupo de sentencias
}
```

A diferencia del for, while ocupa una sola expresión que mientras sea distinta de cero es verdadera, y falsa cuando es exactamente igual a cero. El grupo de sentencias se ejecuta una vez, se evalúa la expresión y si es verdadera, se vuelven a ejecutar las sentencias en el bloque. El bucle while es muy utilizado para crear lazos infinitos en video juegos. Se puede salir por programa o presionando una tecla u otro dispositivo que actúe con el video juego.
Ejemplo:

```
int k=10;
while(k)
{
        //sentencias
        k--;
}
```

En este ejemplo, las sentencias se ejecutan mientras la variable k sea distinta de cero. Cuando k es cero, se sale del while.

Ejemplo:

```
int k=0;
while( k < 12)
{
        //sentencias
        k++;
}
```

En este ejemplo, se usa como expresión k < 12 , que se mantiene como verdadera mientras k sea menor al valor entero 12. Cuando esta expresión relacional no se cumpla, la expresión se evalúa como falsa, se convierte en un cero y sale del while. La variable k parte en cero, pero se incrementa en 1, cada vez después de ejecutarse las sentencias anteriores a k++. El grupo de sentencias se ejecutan 12 veces.

4.2.3 Sentencia do-while

La sentencia while comienza la ejecución de las sentencias dentro del bloque sólo si la expresión en el paréntesis es verdadera. Es posible entonces, que las sentencias dentro del bloque nunca se ejecuten.
El bucle do-while ejecuta de inmediato una primera vez las sentencias dentro del bloque y después pregunta si la expresión es verdadera o falsa. La sintaxis del do-while es la siguiente:

```
do{
        //sentencias
}while(expresión);
```

Las sentencias dentro del bloque se ejecutan una vez, independiente si la expresión es verdadera o falsa y continúan ejecutándose mientras la expresión es verdadera.
Ejemplo:

```
int k=10;
do{
        printf("dentro del bloque");
        k--;
}while( k );
```

En este ejemplo el printf() se ejecuta 10 veces y sale del while.

4.3 Sentencias de selección.

4.3.1 Sentencia if

La sentencia if permite decidir si una o un grupo de sentencias se ejecutan. Para la decisión, se usa cualquier expresión válida en lenguaje C que al evaluarse tome el valor cero (falsa) o distinta de cero (verdadera). La expresión puede ser una fórmula, una expresión relacional o una expresión lógica. Las sintaxis del if puede ser cualquiera de las siguientes:

1º Forma:

```
if( expresión)
{
   //sentencias
}
```

Las sentencias dentro del bloque se ejecutan si la expresión es verdadera (distinta de cero).

2º Forma:

```
if( expresión)
{
    //grupo sentencias 1
}else
{
   //grupo sentencias 2
}
```

En esta segunda forma del if, las sentencias del grupo 1 se ejecutan si la expresión es verdadera y si la expresión es falsa se ejecutan las sentencias del grupo 2. Sólo un grupo de sentencias se ejecuta a la vez.

3º Forma:

```
if( expresión 1)
{
    //grupo sentencias 1
}else if( expresión 2)
{
   //grupo sentencias 2
}
..... else if   //se puede repetir….
else
{
   //grupo sentencias 3
}
```

En esta forma del if, el grupo de sentencias 3 se ejecuta sólo si ninguna de las expresiones 1 o 2 son verdaderas. La sentencia else puede estar o no estar. Si la expresión 1 no es verdadera, entonces se pregunta por la segunda y por las expresiones de los else if que sigan hasta que una sea verdadera. Sólo un grupo de sentencias es ejecutado.

Los paréntesis curvos { } no son necesarios si un grupo de sentencias es una sola sentencia.

Ejemplo:

```
if( n > 0)
  printf("Una sola sentencia"); // no es necesario incluir { }
```

Ejemplos:

También es posible usar **if** anidados como se muestra en el siguiente ejemplo:

```
if ( n > 2)
   printf("n mayor que 2\n");
else
        if(  n > 5)
          printf("n mayor que 5 verdadero\n");
        else
         printf("n > 5 falso\n");
```

Sin embargo, en programación C se prefiere usar **else if** en vez de **if anidados**, como se muestra en el siguiente ejemplo:

```
if ( n > 2)
   printf("n mayor que 2\n");
else if ( n > 5 )
  printf("n mayor que 5 verdadero \n");
else
  printf("n > 5  falso\n");
```

El uso de paréntesis curvos { } y else if en vez de if anidados, ayuda a una mejor programación exenta de errores que pueden crearse al momento de compilar.

Expresión condicional

La expresión condicional tiene la siguiente sintaxis:

```
expresión1  ?  expresión2   :   expresión3
```

Se evalúa de la siguiente manera: Si la expresión1 es verdadera (distinta de cero), entonces el valor de toda la expresión condicional es el valor de la expresión2. Si la expresión1 es falsa, entonces el valor de toda la expresión condicional es el valor de la expresión3.
La expresión condicional puede ser usada en un if.
Ejemplo:

```
int i = 2, j = 5, k = 20;
float delta = 12.0;

if( delta < k && i > j  ? k==i : delta > j*i)
        printf ("verdadero\n");
else
        printf( " falso\n");
```

El resultado de este if es que imprime verdadero ¿Por qué?

Tipo Booleano en C99

La versión C99 y posteriores del lenguaje C introducen el Tipo _Bool para variables booleanas. Estas variables pueden tomar el valor 0 ó 1. Aún cuando es posible asignar un entero mayor a 1 (positivo o negativo) a una variable booleana, la variable booleana guarda un 1 para cualquier entero asignado mayor a 1.

Ejemplo:

```
_Bool  beta = 1;
if(beta)
   Printf(" beta verdadero\n");
```

También, la versión C99 introduce una nueva biblioteca <stdbool.h> con el tipo bool y las palabras **true** y **false**. El siguiente ejemplo utiliza **#include <stdbool.h>**
Ejemplo:

```
bool  beta = true;   // true es equivalente a 1
if (beta)
   printf(" beta verdadero\n");
```

4.3.2 Sentencia switch.

La sentencia switch permite decidir que sentencias se ejecutan dependiendo del resultado de una expresión. Dentro de un bloque, switch contiene una o más sentencias case seguidas por una expresión-constante. La expresión-constante se evalúa al momento de compilación y no puede contener variables, pero si puede contener caracteres porque en C, los caracteres se tratan como enteros. La expresión del switch, al ser evaluada debe entregar como resultado un entero. Números flotantes y ristras (strings) de caracteres no se pueden usar en la expresión-constante después de un case.

La sintaxis del switch es la siguiente:

```
switch (expresión)
{
        case  expresión-constante:
                //sentencias
                break;
        case expresión-constante:
                //sentencias
                break;

        ……. Se puede repetir case….

        default : // default es opcional
                //sentencias (opcional)
                break;
}
```

La sentencia default es opcional, pero es útil en el sentido de poder detectar si el **switch** no es ocupado correctamente. La sentencia break en el default es necesaria para evitar errores cuando se agregan sentencias case al **switch**. También, si la última sentencia es un case (no hay default), debe ir un break para evitar errores al agregar más sentencias case. Estrictamente, por ser el último case, el break no es necesario, pero útil.
La sentencia default se ejecuta sólo si ninguna sentencia case fue válida. Por esto, default puede ir en cualquier parte del switch.

Ejemplos:

Ejemplo 1:

```
int dat = 2;

switch ( dat )
{
        case 1:
        case 2:
          printf(" caso 2\n");
        break;
        case 3:
          printf(" dat distinto de 1,2 o 3\n");
        break;
        default:
          printf(" default\n");
        break;
}
```

Como resultado de ejecutar este switch se imprime caso 2, porque la expresión dat es un 2. En el ejemplo, **case 1**: no tiene nada y después viene **case 2:** . Esta es la forma, si se desea que sentencias se ejecuten para uno o más valores de la expresión del switch.

Ejemplo 2:

```
int dat = 21, beta = 2;

switch ( dat%2 + beta )
{
        case 1:
          printf(" caso 1\n");
        case 2:
          printf(" caso 2\n");
        break;
        case 3:
          printf(" caso 3\n");
        break;
        default:
        break;
}
```

En este ejemplo se imprime caso 3. ¿Por qué?

Ejemplo 3:

```
unsigned int letra = 65;

switch ( letra )
{
        case 'B':
          printf(" caso letra B\n");
        case 'A':
          printf(" caso letra A\n");
        break;
        case 'C':
          printf(" caso letra C\n");
        break;
        default:
        break;
}
```

En este ejemplo se imprime caso letra A, porque el código ASCII de la letra A es 65. La letra 'A' se lee como un entero de valor 65.

4.4 Variables estáticas.

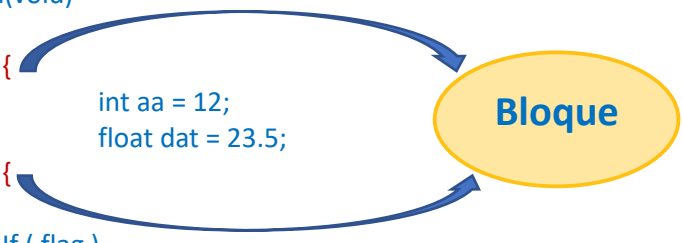
Las variables estáticas al igual que las variables globales tienen existencia como objetos de la memoria por toda la duración de un programa. Es importante establecer su diferencia con las variables locales o automáticas creadas dentro de un bloque, cuya existencia como objeto de memoria, dura mientras se está dentro del bloque. Sea este el bloque main() o bloques if, for, while etc.

4.4.1 Bloques en lenguaje C.

Un bloque en lenguaje C se obtiene cuando una o un grupo de sentencias se encierran por paréntesis { }.

Ejemplo:

```
#include <stdio.h>
#include<bool.h>
bool flag = true;  // Variable global

int main(void)
{
        {
                int aa = 12;
                float dat = 23.5;
        {
        If ( flag )
        {
                double  sat = 45.0;
        }
        return 0;
}
```



En lenguaje C, un bloque no necesariamente es a continuación de un if , while etc., si no que basta usar un par de paréntesis cerrados { } como se indica con color granate en el ejemplo anterior.
En dicho bloque, las variables declaradas dentro del bloque aa y dat, tienen existencia mientras se está dentro del bloque. Las variables aa y dat se crean y guardan en memoria al entrar al bloque. Sin embargo, al salir del bloque, dichas variables aa y dat liberan la memoria ocupada, la cual puede ser ocupada para otros datos por el sistema operativo. Debido a su desaparición como objetos, estas variables aa y dat no pueden ser referenciadas fuera del bloque. Lo mismo sucede con la variable sat, declarada dentro del bloque del if.
La variable flag declarada antes del main(), es una **variable global** y como objeto dura toda la existencia del programa. Esta variable, puede usarse en cualquier parte del programa, a continuación de su declaración.

4.4.2 Declaración de variables estáticas.

Una variable estática se declara colocando la palabra static antes del tipo.
Ejemplo:

```
static int dat = 4;
```

4.4.3 Diferencias entre variables estáticas y locales.

La principal diferencia entre una variable declarada estática y otra declarada localmente o automática es que el objeto para una variable estática se define e inicializa en tiempo de compilación, antes de entrar a ejecutar el main(), mientras que una variable local o automática crea el objeto en tiempo de ejecución del programa y lo inicializa en ese momento.
Por esto, el valor usado para inicializar una variable estática debe ser una expresión-constante, esto es un número, que pueda ser determinado en tiempo de compilación.
Una variable automática puede ser inicializada con una expresión que se calcule en tiempo de ejecución.
Como el objeto de una variable estática se define e inicializa antes de la ejecución del programa, esta variable es declarada una sola vez y no pierde su calidad de objeto durante toda la duración del programa.
La zona de la memoria en la cual se crea la variable estática, **al cargar el programa ejecutable**, se llama **Segmento de Datos** cuando es inicializada, como ocurre en este caso. Cuando la variable estática no es inicializada, se guarda en la zona de memoria llamada **bss**.
Ejemplo:

1º programa:

```
#include <stdio.h>

int main(void)
{
        for(int i = 0; i < 5; i++)
        {
                int beta = 2; // variable automática
                printf("valor de beta = %d\n",beta);
                beta++;
        }
        return 0;
}
```

El printf() imprime los valores 2 2 2 2 2 (en cada pasada del for imprime el número 2).
Como resultado del 1º programa se imprime cinco veces el valor 2. Cuando se entra al bloque del for la variable beta se crea e inicializa con el valor 2. Cuando se sale del bloque del for el objeto beta libera la memoria ocupada y la variable beta deja de existir. Luego, se vuelve a entrar al paréntesis () del for a evaluar la variable **i** y si es menor que 5 se entra de nuevo en el bloque del for. Como la variable beta dejó de existir, se crea de nuevo e inicializa con el valor 2. Esto ocurre cinco veces en este ejemplo y luego se imprime cinco veces un 2. En cada pasada del for, el objeto beta puede ocupar distintas o iguales partes de la memoria, según lo decida el **STACK**. La zona de la memoria en la cual se guarda y se saca la variable beta se llama **STACK**.

2º programa:

```
#include <stdio.h>
int main(void)
{
        for(int i = 0; i < 5; i++)
        {
                static int beta = 2; // variable estática
                printf("valor de beta = %d\n",beta);   // imprime los valores 2 3 4 5 6
                beta++;
        }
        return 0;
}
```

En el 2º programa la variable beta se declara estática. Antes de entrar a ejecutar el main(), en tiempo de compilación, el objeto de la variable estática beta se define e inicializa con el valor 2. La variable beta se almacena en la zona de memoria Segmento de Datos al momento de cargar el programa y no se declara más.
Al comenzar la ejecución del main(), la variable beta vale 2 y en la primera pasada del for se imprime un 2. Luego se incrementa beta a 3. Se sale del bloque del for y se entra a evaluar la variable **i,** como es menor a 5 se entra en el for. La declaración de la variable estática beta se ignora y se imprime el valor 3.
Así, de la misma manera se continúa con la impresión de los valores 4, 5 y 6.

3º programa:

```
#include <stdio.h>
int main(void)
{
        for(int i = 0; i < 5; i++)
        {
                int beta = 2;
                printf("valor de beta = %d\n",beta);
                beta++;
        }
                printf("valor de beta = %d\n",beta);  // ERROR la variable beta no existe fuera del bloque
                                                      // del for.
        return 0;
}
```

4º programa:

```
#include <stdio.h>

int main(void)
{
        for(int i = 0; i < 5; i++)
        {
                static int beta = 2;  // variable estática
                printf("valor de beta = %d\n",beta);
                beta++;
        }
                printf("valor de beta = %d\n",beta);  // ERROR la variable estática beta no existe fuera del
                                                      // bloque del for.

                return 0;
}
```

Tanto las variables no-estáticas como las estáticas, al ser declaradas dentro de un bloque, existen dentro del ámbito del bloque. Por lo tanto, no pueden ser referenciadas fuera del bloque.
Sin embargo, es posible en el caso de las variables estáticas y por el hecho de no eliminarse de la memoria su objeto al salir del bloque, usar un puntero para referenciar una variable estática fuera de su ámbito de existencia.
Esto no es posible hacerlo con las variables no-estáticas, porque su objeto desaparece de la memoria al salir del bloque.

4º programa:

```
#include <stdio.h>

int main(void)
{
        int*  observador = NULL;  // declaración de puntero observador

        for(int i = 0; i < 5; i++)
        {
                static int beta = 2; // variable estática
                printf("valor de beta = %d\n",beta);   // imprime los valores 2 3 4 5 6
                beta++;
                observador = &beta; // asigna dirección de beta a puntero observador
        }
                printf("Valor de beta= %d\n", *observador); // imprime valor de beta fuera de su  ámbito
                //de existencia usando el puntero observador.

        return 0;
}
```

La variable estática beta se crea en la memoria una sola vez y es posible tener acceso a ella sólo si se está dentro del ámbito de su creación, esto es, dentro del bloque del for para el ejemplo del 4º programa.
Sin embargo, toda variable que está en memoria tiene una dirección y si la variable no cambia su posición de memoria en el transcurso del programa, como ocurre con las variables estáticas, entonces es posible usar su dirección para hacer referencia a ella desde fuera de su ámbito de creación. Para el caso del ejemplo anterior, fuera del bloque del for.
El puntero **observador** permite leer o modificar el valor de la variable beta desde fuera de su ámbito de existencia. Ciertamente, esto puede realizarse solamente mediante un puntero.

4.5 Función scanf.

La función scanf permite ingresar valores al programa en ejecución usando el teclado. Sin embargo, se indicarán varias precauciones que es necesario tener al usar esta sentencia o función en lenguaje C.
La sintaxis del scanf es la siguiente:

```
int gama;
scanf("%d", &gama);
```

Aquí, scanf guarda el valor entero ingresado por teclado en la variable gama.
A diferencia del printf, scanf utiliza la dirección dónde se desea guardar el valor ingresado por el teclado.
El formato, para el valor ingresado, es muy similar al del printf con algunas diferencias. A continuación, se muestran formatos que no se pueden usar con el scanf:

1º caso `scanf(" valor ingresado= %d", &gama);`

2º caso `scanf("%d \n ", &gama);`

En ambos casos la función scanf falla. En el primer caso se resuelve con:

```
printf("valor ingresado= ");
scanf("%d", &gama);
```

En el segundo caso, simplemente no usar \n en el scanf.
A continuación, se indican los formatos a usar con un scanf:

d, i, n para números enteros. Si se antepone la letra h minúscula a cualquiera de estos formatos lo convierte en un short. Anteponiendo la letra **l** lo convierte en un long, letra **o** para números octales, **u** para números enteros sin signo y **x** para números hexadecimales.
Anteponiendo la letra h lo convierte en unsigned short y anteponiendo la letra l lo convierte en unsigned long.
e, f, g son formatos para números flotantes. Anteponiendo la letra l lo convierte en double y anteponiendo la letra L en long double.
s para ingresar palabras o strings(sin espacio).
[^\n] para ingresar frases con espacio entre palabras.

A diferencia con el printf, **scanf** debe usar el formato **lf** para dobles.

Ejemplos de uso del scanf:
Ejemplo 1:

```
int k, h, n;
double dat;
scanf("%d%d%d%lf", &k, &h, &n, &dat);
```

Los números deben ingresarse en el teclado separados por un espacio: Ejemplo 2 34 56 78.56
También, puede usarse una coma para separar los formatos:

```
 int k, h, n;
double dat;
scanf("%d,%d,%d,%lf", &k, &h, &n, &dat);
```

Ahora los números deben ingresarse en el teclado separados por una coma: Ejemplo 2 ,34 ,56 ,78.56
No ingresar una coma producirá un error en la lectura del scanf.
Lo que el programador use para separar los formatos, debe usarlos exactamente de la misma manera para separar los números al ingresarlos por el teclado. Lo mismo si mezcla ingreso de números y caracteres.
Ejemplo:

```
int k, h, n;
double dat;
scanf("%d%d,,%d;%lf", &k, &h, &n, &dat);
```

Los números deben ingresarse en el teclado separados por un **espacio coma punto y coma**: Ejemplo 2 34, 56; 78.56
El segundo formato %d está separado del tercero por una coma, el tercero del cuarto por un punto y coma y exactamente estas mismas separaciones deben usarse al ingresar lo números por el teclado. Si se olvida una, la lectura del siguiente número será errónea y el scanf falla.

¿Cómo usar scanf para ingresar valores a un arreglo?

Dos formas posibles (el programador puede usar otras):
1º Forma:
Usando direcciones,

```
int dat[7] ={0};  //inicializa el arreglo con todos los elementos en cero
printf(" Ingrese los valores para el arreglo separados por espacio)=");
scanf("%d%d%d%d%d%d%d", dat,dat+1,dat+2,dat+3,dat+4,dat+5,dat+6);
```

Esta forma se puede usar cuando el arreglo es pequeño.

2º Forma:

```
float data[20] = {0};
for(int i = 0; i < 0; i++)
{
        printf("Ingrese data[%d]= ", i); // %d es reemplazado por el valor de i
        scanf("%f", &data[i]); // También se puede usar data + i
        if(data[i] == 19)
        printf("No hay más valores que ingresar")
}
```

Esta forma es adecuada cuando un arreglo es de tamaño mediano. Para arreglos grandes es preferible usar un archivo para copiarlo a un arreglo.

¿Cómo ingresar un mensaje?
Para ingresar un mensaje se utiliza el formato s:
Ejemplo:

```
char mensaje[20];
scanf("%s", mensaje);  // mensaje es la dirección del arreglo
```

El mensaje debe no debe tener espacios entre los caracteres. El tamaño del arreglo debe ser suficiente para contener el largo del arreglo.

¿Cómo ingresar y escribir un mensaje como: Ciudad del sur?
Ejemplo: Una forma simple de hacerlo es:

```
char mensaje1[10];
char mensaje2[10];
char mensaje3[10];
printf("Ingrese mensaje =");
scanf("%s%s%s", mensaje1, mensaje2, mensaje3);
printf("%s %s %s\n", mensaje1, mensaje2, mensaje3);
```

Al ingresar por teclado: Ciudad del sur se imprime: Ciudad del sur.
Las palabras deben ingresarse en una misma línea separadas por un espacio.
En el último printf, los formatos %s deben estar separados por un espacio para que las palabras al imprimirse queden espaciadas. Esto no es necesario en el scanf, porque scanf no toma en cuenta los espacios. Sin embargo, es mejor usar el formato **[^\n]** para ingresar varias palabras separadas por espacio:
Ejemplo:

```
char frase[100];
printf("Ingrese varias palabras separadas por espacio = ");
scanf("%[^\n]", frase);
printf("%s\n", frase);
```

4.6 Ejercicios resueltos.

IMPORTANTE: NO HACER COPY PASTE DE LOS PROGRAMAS PORQUE ALGUNOS CARACTERES DEL WORD NO FUNCIONAN EN EL PELLES O CODEBLOCK.

4.6.1 Intercambiar el valor entre dos variables.

Para programar este ejercicio, se necesita una variable temporal que guarde el valor de una de las dos variables a intercambiar:

```
int dat = 12;
int gama = 20;
int temp = dat; // variable temporal temp se inicializa con valor de dat
dat = gama; // asigna el valor 20 a dat
gama = temp; // asigna el valor 12 a gama, guardado previamente en temp.
```

después de ejecutar este programa, la variable dat contiene el valor 20 y la variable gama contiene el valor 12.

4.6.2 Escribir un programa que realice la suma de productos, como ocurre en el producto punto de dos vectores:

Sean vector v1 con coordenadas $\{ a_1 , a_2 , a_3 , a_4 \}$

vector v2 con coordenadas $\{ b_1 , b_2 , b_3 , b_4 \}$

El producto punto es $v1 * v2 = a_1 * b_1 + a_2 * b_2 + a_3 * b_3 + a_4 * b_4$

Las coordenadas de cada vector se guardan en un arreglo de una dimensión:

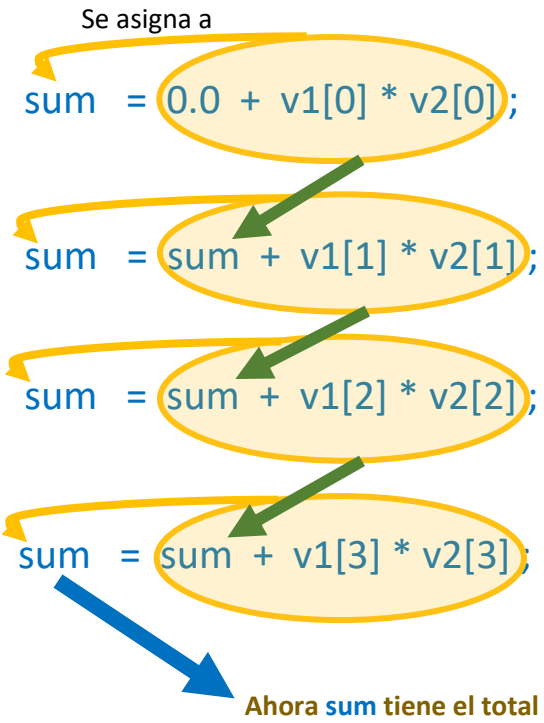
```
double v1[4] = {2.0, 3.4, 5.6, 7.8};
double v2[4] = {1.0, 2.5, 6.7, 8.0};
```

Para realizar la suma de los productos se necesita un for que se repita cuatro veces:
Además, se necesita una variable externa al for inicializada en cero para acumular los productos parciales.

```
double sum = 0.0; // variable que acumula productos parciales.
for(int i=0; i < 4; i++)
{
        sum = sum + v1[i] * v2[i];
}
printf("valor total de sum= %f\n", sum); // imprime valor total de sum
```

Al salir del for la variable sum contiene la suma de los productos parciales de los dos vectores guardados en los arreglos v1 y v2.
La variable sum se declara fuera del for porque se necesita que su ámbito de existencia no se restrinja al bloque del for.
En la primera pasada del for, sum tiene el valor 0.0 y se asigna 0.0 + v1[0]*v2[0] a sum.
En la segunda pasada del for, el valor anterior de sum se suma a v1[1] * v2[1] y esto se asigna a sum.
En la tercera pasada del for, el valor anterior de sum se suma a v1[2] * v2[2] y esto se asigna a sum.

En la cuarta y última pasada del for, el valor anterior de sum se suma a v1[3] * v2[3] y esto se asigna a sum. Se muestra en forma gráfica a continuación:



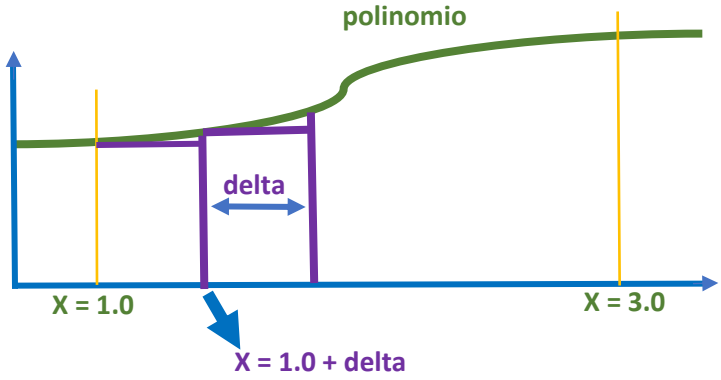
Recordemos que la expresión sum = sum + v1[i] * v2[i] se puede escribir como sum += v1[i]*v2[i].
En adelante usaremos la segunda forma para expresiones similares.

4.6.3 Calcular la integral definida de un polinomio.

Sea el polinomio:

$$3X^3 + 2X^2 + X + 6$$

La integral de este polinomio entre los valores de x {1.0, 3.0} está dada por el área bajo la curva generada por el polinomio al ser evaluado entre los valores x = 1.0 y x = 3.0.



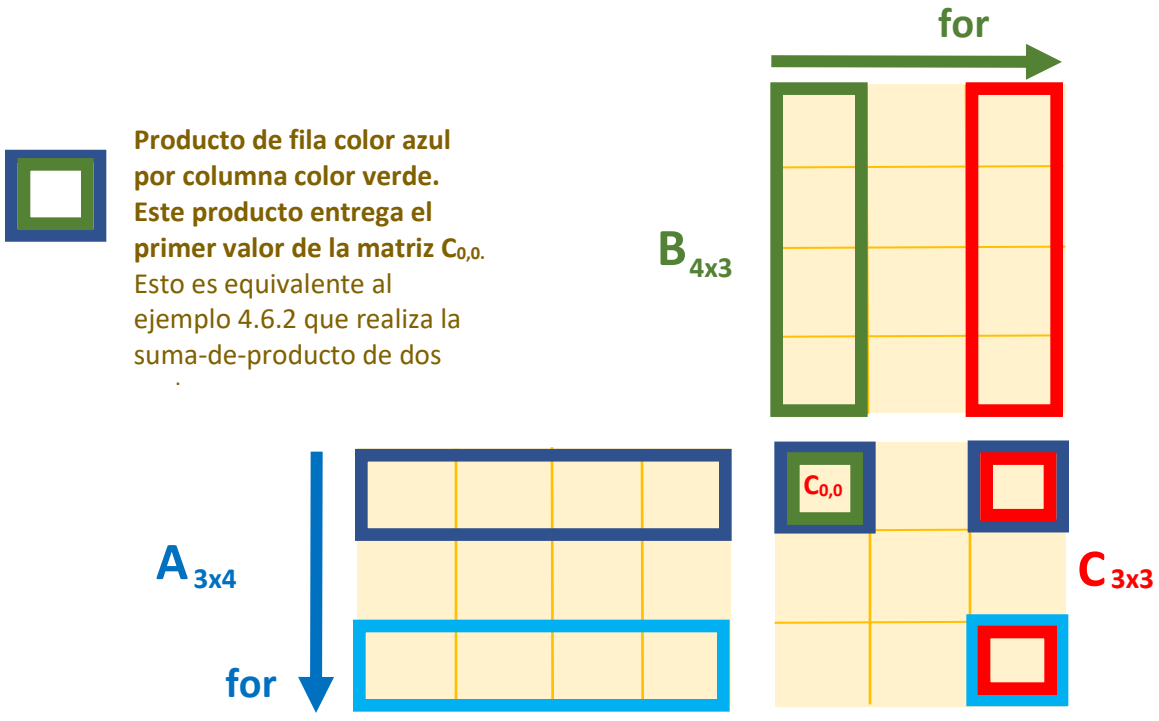
El área bajo la curva del polinomio es la suma de las áreas de los rectángulos que van desde x=1.0 hasta x=3.0. El área de cada rectángulo es alto * ancho. El alto es el valor del polinomio evaluado a un x determinado y el ancho es el valor delta elegido. Mientras más pequeño el valor de delta, más preciso es el cálculo del área bajo la curva y por lo tanto el valor de la integral.
Para escribir un programa que calcule el valor de la integral haremos uso de los resultados obtenidos en los ejercicios 1 y 2.
La integral es la suma de los productos alto * ancho de cada rectángulo. El número total de rectángulos dependerá del valor elegido para la variable delta. Si delta = 0.01, entonces el número total de rectángulos es 2.0/delta, esto es 200. Este número 200 nos servirá para el número de veces que ejecutaremos un for.

```
double x = 1.0;
double delta = 0.01;
double area = 0.0;
for( int i =0; i < (int)(2.0/delta) ; i++)  // (int) es un casting a entero
{
        x=1.0 + i*delta;
        area  +=  (3.0*x*x*x + 2.0*x*x + x + 6.0) * delta; //ver 4.6.2
}
printf("valor integral = %f", area); // imprime el valor de la integral
```

Es interesante aumentar la precisión de la variable delta, por ejemplo, a 0.0001 y ver como el área bajo la curva se ajusta cada vez más a su valor real.

4.6.4 Escribir un programa que multiplique dos matrices.

La multiplicación de dos matrices es una multiplicación repetitiva de dos vectores. Los vectores son las filas y columnas de las matrices. En la siguiente figura se muestra el procedimiento de una multiplicación de dos matrices, A de 3x4 y otra B de 4x3. Recordemos que la multiplicación de dos vectores es una suma de productos (ejercicio 4.6.2).



Para almacenar las matrices A y B se usarán arreglos de dos dimensiones. Sin embargo, como normalmente los arreglos de dos dimensiones guardan una matriz fila a fila, la matriz B se almacenará columna a columna. Esto para facilitar el producto de una fila por una columna para obtener un valor de la matriz de salida C.

```
double matA[3][4] = {{2.0,4.5,3.4,7.8},{1.0,2.0,3.4,6.7},{3.7,6.7,9.0,4.6}};
double matB[4][3] = {{2.0,7.9,5.8},{1.5,3.9,2.5},{3.5,2.4,7.9},{7.9,6.2,8.3}};
```

Cada bloque representa una fila de la matriz.

La matriz B se modifica a una matriz de 4x3 columna a columna.

```
double matBcol[3][4] = {{2.0,1.5,3.5,7.9},{7.9,3.9,2.4,6.2},{5.8,2.5,7.9,8.3}};
```

Para esta matriz modificada, cada bloque representa una columna de la matriz.

En el gráfico anterior se muestra que el primer valor de la matriz se obtiene multiplicando la primera fila de la matriz A por la primera columna de la matriz B. Esta multiplicación es la ya conocida suma de productos. La primera fila de la matriz A se itera con todas las columnas de la matriz B para obtener todos los valores de la primera fila de la matriz C. Así, la iteración sobre las filas de la matriza A será una sentencia for más externa (ver en gráfico línea de color azul grueso). El segundo for, interno al for anterior de las filas de A, itera sobre las columnas de la matriz B (ver en gráfico gruesa línea de color verde). La multiplicación de una fila por una columna se hace con un for más interno, dentro del segundo for. Esto es igual como el programa del producto punto de dos vectores.
El siguiente es el programa completo:

```
#include <stdio.h>
int main(void)    // ver 4.2.1.1 for() anidados
{
        double matA[3][4] = {{2.0,4.5,3.4,7.8},{1.0,2.0,3.4,6.7},{3.7,6.7,9.0,4.6}};
        double matBcol[3][4] = {{2.0,1.5,3.5,7.9},{7.9,3.9,2.4,6.2},{5.8,2.5,7.9,8.3}};
        double C[3][3];

        for(int i = 0; i < 3 ; i++)      // ver for línea color azul en gráfico
        {
                for( int j = 0; j < 3 ; j++)  //ver for línea color verde en gráfico
                {
                        double sum = 0.0;
                        for(int k = 0; k < 4; k++)   //producto fila por columna (4.6.2)
                        {
                                sum  += matA[i][k] * matBcol[j][k];
                        }
                        C[i][j] = sum;
                {

        }

return 0;
}
```

4.6.5 Ordenar números de mayor a menor o viceversa.

Se asume que los números están en un arreglo de una dimensión. La forma más directa y simple de ordenar los números de mayor a menor es comparando el valor en la primera posición del arreglo con todos los otros números. Se compara el primero con el segundo. Si el primero es menor que el segundo, se intercambia. Se compara el primero con el tercero. Si el primero es mayor que el tercero, se continúa. El intercambio sólo se hace cuando el primer valor es menor que aquél con cual se compara. De esta manera al terminar la comparación del primer valor con todos los demás, el primer valor del arreglo es el número mayor de todos los que están en el arreglo.
Se continúa comparando el valor en la segunda posición del arreglo con todos los demás. Así, en la segunda posición quedará el segundo valor mayor de todos los que hay en el arreglo.
Esto se repite hasta la penúltima posición del arreglo. El siguiente es el programa:

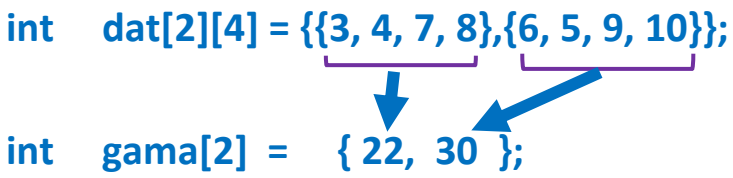
```
#include <stdio.h>

int main(void)
{
        int dat[11] = {10,340,23,67,8,9,45,678,33,20,66};
        int temp;  // Observe que la variable temp se declara antes del for ¿porqué?
        for( int i = 0; i < 10 ; i++) // itera entre el primero y el penúltimo
        {
                for( int k = i + 1; k < 11; k++) // itera y comprueba si es menor
                {
                        if( dat[i] < dat[k]) // intercambia valores
                        {
                                temp = dat[i];
                                dat[i] = dat[k];
                                dat[k] = temp;
                        }
                }
        }
        for(int i = 0; i < 11; i++)
                printf("%d\n", dat[i]); // imprime todos los números ordenados
        return 0;
}
```

Si en el if se cambia el símbolo < por >, entonces el programa ordena de menor a mayor.

No se debe olvidar que para cada valor de la variable i en el for más externo, se ejecutan completamente todas las instrucciones dentro del bloque de este for. Esto implica que para cada valor i, el for más interno con variable k se ejecuta 11 menos i+1 veces.
Los for anidados, como en este caso se ejecutan en secuencia desde el más externo al más interno.

4.6.6 Reducir un arreglo de dos dimensiones a una dimensión.

Reducir un arreglo de dos dimensiones a un arreglo de una dimensión, donde cada elemento del arreglo de una dimensión sea la suma de los valores en cada bloque del primer arreglo.
Ejemplo:



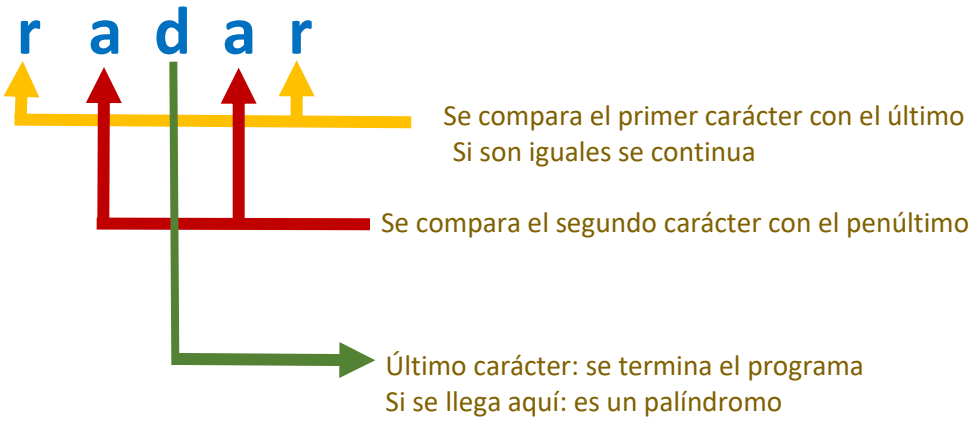
Para escribir este programa se usarán punteros para indicar la dirección de comienzo de cada bloque del primer arreglo:

```
#include < stdio.h >


int main(void)

{
        int dat[2][4] = {{3,4,7,8},{6,5,9,10}};
        int gama[2];
        int* puntero1 = dat[0]; // puntero1 apunta al valor 3
        int* puntero2 = dat[1]; // puntero2 apunta al valor 6
        int sum = 0;

        for( int i = 0; i < 4; i++)
                sum += *(puntero1 + i);
        gama[0] = sum;
        sum = 0;

        for( int i = 0; i < 4; i++)
                sum += *(puntero2 + i);
        gama[1] = sum

        printf("%d %d\n", gama[0], gama[1]);

        return 0;
}
```

4.6.7 Desarrollar un programa que resuelva un palíndromo.

Un palíndromo es una palabra que se lee igual desde la izquierda o desde la derecha:
Ejemplos: **anna radar 56765**

Para desarrollar el programa deben considerarse los casos en que la longitud del palíndromo es par o impar. El programa debe aceptar por teclado el ingreso de un palíndromo y escribir en la pantalla si es o no es un palíndromo. El siguiente esquema describe el algoritmo a programar:



Cualquier comparación intermedia (en este ejemplo: colores amarillo y rojo) que resulte en caracteres desiguales se termina el programa porque no-es-un palíndromo.
Si después de todas las comparaciones intermedias se llega al último carácter, entonces si-es-un palíndromo.


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

int main(void)
        {
                char pal[20];     // Suficiente espacio para un palíndromo
                printf(" Ingresar palindromo=");
                scanf("%s", pal);
                int longitudPal = 0; //largo del palíndromo
                bool flag = true;
```

```
for( int i=0; i < 20; i++)     //este for() calcula el largo del palíndromo
{
        if( pal[i] != '\0')
                longitudPal++;
        else
        break;
}

if(longitudPal%2)  //palíndromo par
{
        for(int i = 0; i < (longitudPal - 1)/2; i++)
        {
                if(pal[i] != pal[longitudPal - 1- i])
                {
                        printf(" No es palíndromo\n");
                        flag = false;
                }
        }
                if(flag)
                printf(" Es palindrome\n");
}
else   //palíndromo impar
{
        for(int i = 0; i < longitudPal/2; i++)
        {
                if(pal[i] != pal[longitudPal - 1 - i])
                {
                        printf(" No es palíndromo\n");
                        flag = false;
                }
        }
if(flag)
printf(" Es palindrome\n");
}
return 0;
}
```

4.6.8 Determinar si un número entero es primo o no es primo.

Un número entero es primo si es divisible por 1 o por si mismo. El número 1 se define como no-primo y el único número par que es primo es el número 2. Cualquier otro número mayor a 2 sería divisible por 2 y por lo tanto, no-primo. Así, un número primo mayor a 2, debe ser un número impar de la forma 2*n+1, donde n es un entero positivo en [1, infinito].

El siguiente programa determina si un número impar mayor a 1 es primo o no lo es.

```
int main(int argc, char *argv[])
{
   printf("Ingrese un numero entero impar mayor a 1=");
        int primo = 0;
        scanf("%d", &primo);
        while((primo < 3) || (primo%2==0))
        {
                printf("Ingrese un numero entero impar mayor a 1=");
                scanf(" %d", &primo);
        }

        for(int i=1;(2*i+1) < (primo-1)/2;i++)
        {
                        if(primo%(2*i+1) == 0)
                        {
                                printf("No es numero primo\n")
                                return 0;
                        }
        }

                        printf("Es numero primo\n");

   return 0;
}
```

El bucle **while()** no permite ingresar números menores a 3 o números pares.
El ciclo **for** recorre todos los números impares menores a la mitad del número primo-1. Pasado la mitad del número primo la división del primo ingresado sería siempre menor a 1 y por lo tanto, no es necesario hacer la comprobación si es o no es primo.

4.6.9 Determinar cuántas veces se repite un carácter en una frase.

El siguiente programa solicita el ingreso de una frase de varias palabras. Enseguida pide ingresar un solo carácter y entonces imprime cuantas veces el carácter se repite o no en la frase ingresada.

```
int main(int argc, char *argv[])

{

   char beta[50];

        char letra;

        printf("Ingrese frase=");   //ejemplo: “Las aventuras de Tom Sawyer”

        scanf("%[^\n]",beta);   // el formato en corchetes permite ingresar palabras y espacios

        printf("%s\n",beta);
```

```
        printf("Ingrese caracter=");
        scanf(" %c",&letra);
        int contador=0;
        for(int i=0; beta[i] != '\0';i++)
        {
                if(beta[i] == letra)
                {
                        contador++;
                }
        }
        if(contador > 0)
        {
                printf("Se repite %d veces\n",contador);
        }else{
                printf("No hay repeticion\n");
        }


  return 0;
}
```

4.6.10 Determina si una secuencia de caracteres se repite en una frase.

A diferencia del problema 4.6.9, ahora se ingresa una secuencia de caracteres (ejemplo: tres caracteres) y el programa determina si esta secuencia está en la frase y cuantas veces se repite.

```
int main(int argc, char *argv[])
{
   char beta[50];
        char letras[3];
        printf("Ingrese frase=");    // Las aventuras de Tom Sawyer
        scanf("%[^\n]",beta);
        printf("%s\n",beta);

        printf("Ingrese tres caracteres seguidos=");     // ejemplo: Tom
        scanf(" %s",letras);

        int largo=0;
```

```
        for(int k=0; beta[k] != '\0' ;k++)
        {
                        largo++;
        }

        printf("%d\n",largo);

        int contador=0;
        for(int i=0; i < (largo-2);i++)
        {
                //el siguiente if() usa &&(AND) para comprobar que las tres letras estén en la frase
                if((beta[i] == letras[0])&&(beta[i+1] == letras[1])&&(beta[i+2] == letras[2]))
                {
                        contador++;
                }

        }

        if(contador > 0)
        {
                printf("Se repite %d veces\n",contador);
        }else{
                printf("No hay repeticion\n");
        }


    return 0;
}
```

FUNCIONES, typedef, PUNTEROS a FUNCIONES, STACKS y RECURSIÓN.

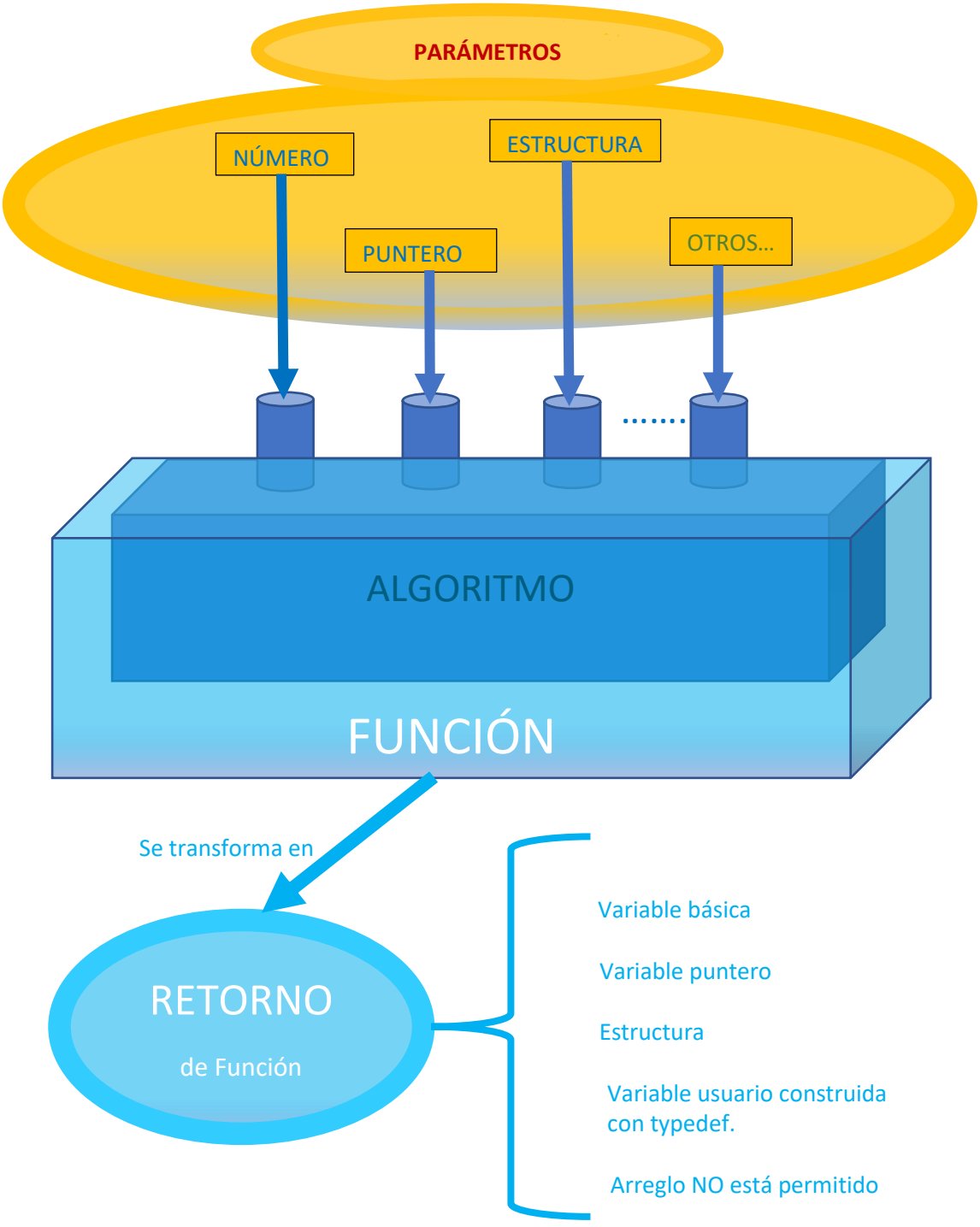
Matemáticas es la Ciencia de las Analogías.
Sir Michael Atiyah ("Fields Medal", 1966)

5.1 Funciones

En un programa, casi siempre es necesario realizar una tarea específica que se repite muchas veces o una tarea que involucra un algoritmo complejo que, por motivos de seguridad o mejor diseño de un programa, es necesario encapsular. También ocurre a menudo, que una tarea que se realiza en un programa es útil para otros programas distintos.
El lenguaje C proporciona lo que llama Funciones que permiten agrupar declaraciones y sentencias para realizar un algoritmo o una tarea específica. Una función puede usarse múltiples veces en un programa o usarse en distintos programas.
Una función también puede navegar por distintos archivos que componen un programa. Las funciones son una parte muy importante del lenguaje C. El lenguaje C es muy utilizado para escribir APIs ("Application Programming Interfaces") como por ejemplo las APIs del software TensorFlow para Inteligencia Artificial de Google, APIs para computación gráfica (Vulkan, OpenGL) etc. Se usa el lenguaje C para escribir estas APIs por su versatilidad y rapidez de ejecución. En general, las APIs son Funciones que transfieren sus resultados al exterior mediante el uso de variables punteros como parámetros. Esto ciertamente lo permite hacer el lenguaje C de forma eficiente.

Usaremos la siguiente analogía para describir una función:
Una función, es un Transformer que acepta parámetros (datos de un cierto Tipo) para ejecutar una tarea o algoritmo y que una vez realizada, la tarea o algoritmo se transforma en un solo resultado o retorno de un Tipo aceptado en el lenguaje C, con ciertas excepciones.

La siguiente figura grafica una función:



La figura anterior representa la forma más usada de una función en un programa, esto es, cuando recibe parámetros y retorna o se transforma en una variable de un tipo de dato.

Otras formas incluyen:

- Función con parámetros y sin retorno (usada por las APIs)
- Función sin parámetros y con retorno (no muy usada)
- Función sin parámetros y sin retorno (raramente usada)

La transformación de la función es una analogía y ciertamente no ocurre de dicha manera, pero nos ayudará más adelante a entender ciertas complejidades de la sintaxis de un llamado a función.

El retorno de un arreglo no está permitido en lenguaje C y es una excepción a los tipos que se pueden retornar. Los arreglos se retornan usando punteros.

Una función, en general, se implementa en un programa en tres partes:

- Declaración
- Definición
- Llamado a la función

En la declaración, se establece el Tipo del retorno de la función y los Tipos de los parámetros a usar en la ejecución de la tarea o algoritmo. La declaración de una función tiene la siguiente forma, y generalmente se coloca antes del main() para que tenga validez en todo el programa:

Ejemplo: **int func(int, double, int*)**; // declaración de una función

Esta es una función que se transforma en/o retorna un entero, tiene el nombre func y acepta como parámetros un entero, un doble y un puntero a entero. En la declaración sólo se incluyen los tipos de los parámetros a utilizar en la definición.

Si no se incluye la declaración de una función; al llamar a la función, el compilador asume que la función retorna un int (entero). Esto puede conducir a un mensaje de error en caso de que el verdadero retorno de la función no sea un entero. Es recomendable usar siempre antes del llamado a una función, la declaración de la función.

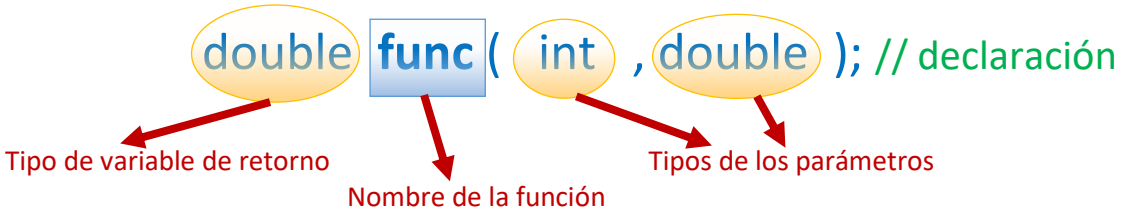
La definición de una función involucra todo lo que la función realiza, ocupando los parámetros que se le asignan desde el llamado a la función. Lo que la función realiza puede ser un algoritmo complejo, un simple cálculo o cualquier simple tarea que permita el lenguaje C. En general, la definición de una función se coloca después del main() para que no interfiera con una buena lectura del programa que está dentro del main() o archivos adicionales.

El llamado a la función se hace exactamente desde el lugar en el cual se desea usar el retorno que entrega dicha función. El llamado a la función debe contener las variables o directamente valores que se asignan a los parámetros de esta función. En lenguaje C, las variables o valores que se indican en el llamado a una función se denominan argumentos. En el siguiente ejemplo se muestra una función con su declaración, definición y llamado a la función con sus parámetros y argumentos.

Las funciones que no retornan un valor o dato se usan igual que una sentencia.

Muchas personas que programan en lenguaje C no hacen distinción entre parámetros y argumentos y los ocupan indistintamente.

En el siguiente ejemplo, de manera gráfica, se muestran todas las partes de una función:




```
#include <stdio.h>

int main(void)
{
        int a = 10;
        double b = ( 30.5 + func( a, 2.3 ) / 2.0 ;  // a se asigna a x ; 2.3 se asigna a y

        printf( "%f\n",  b);

        return 0;

}

double func( int x, double y )
{
        return x*y ;
}
```

Esto es el llamado a la función que se transforma en 23.0

El **retorno x*y** es igual a 10*2.3, esto es 23.0.

La definición de la función es lo que está después del main(void), esto es:

Definición de la función

```
double func(int x, double y)
{
        return x*y;
}
```

Cuerpo de la función

Cuando se llama a la función func(..) desde el main(void), la variable a se asigna al parámetro x en la definición de func() y el valor 2.3 se asigna al parámetro y. El sistema operativo, busca para los parámetros x e y de la función func() al momento de comenzar su ejecución, espacio en memoria del computador, para guardar temporalmente estas variables o parámetros. Terminada la ejecución de la función func(),

los objetos de las variables x e y son eliminados por el sistema operativo, y estos espacios ocupados para otros propósitos.
El CUERPO de la función es aquella parte de la definición encerrada entre paréntesis curvos y representa a todas las sentencias que conforman(algoritmo) lo que ejecuta la función.

Sí una función no retorna nada, se usa el Tipo void para el retorno. En este caso, la función no se transforma en nada al ser ejecutada y sus efectos dependen de las sentencias que se usen en la definición.
Ejemplo:

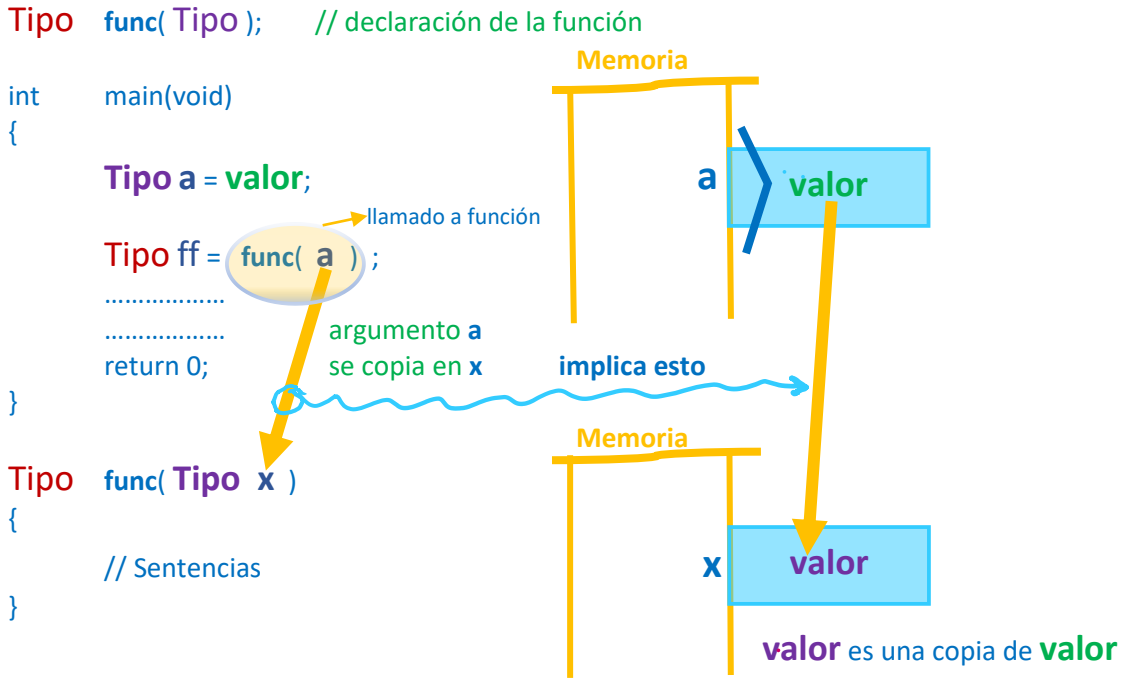
```
void   func2(int x)
{
        printf("%d\n", x);
}
```

Esta función al ser ejecutada imprime en pantalla el valor de la variable x. No retorna nada.
El llamado a una función como func2(..) es simplemente:

```
func2(34); // como cualquier sentencia
           // imprime el valor 34.
```

Un argumento en el llamado a una función, siempre se copia en un parámetro del mismo tipo, en la definición de una función.

Esto se explica en el siguiente esquema:



Ejemplo: Si **Tipo** es un float, entonces **valor** es un número decimal que se copia en **x**. Si **Tipo** es un tipo-a-puntero, entonces **a** es un puntero y **valor** es una dirección (de algún objeto) que se copia en **x**. Lo importante, es que el Tipo del argumento **a** debe ser igual al Tipo del parámetro **x**.
Si el argumento **a** es directamente la dirección de una variable como &variable, entonces se considera una constante y se copia directamente en el parámetro **x**.

Si a fuera un valor numérico, no una variable, se copia directamente en x.

El valor del objeto a se copia al objeto x. El objeto x, se crea en alguna parte de la memoria (distinta dónde está el objeto a) en el momento que se ejecuta la función, y desaparece al término de la ejecución de la función. El objeto x es local al cuerpo de la función (en la definición).

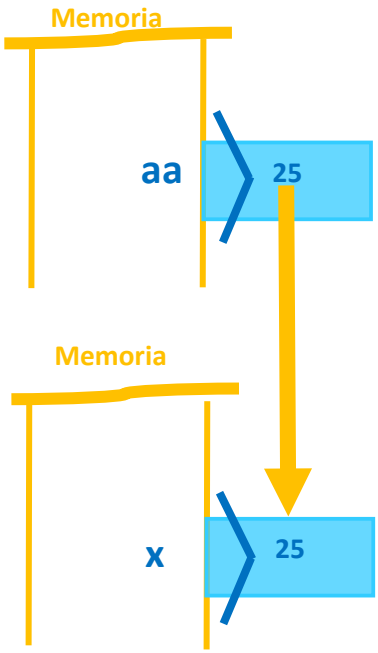
El valor de una variable existente en alguna parte del main(), puede transferirse a la definición de una función de dos formas: por VALOR o por REFERENCIA.
En el caso por VALOR, el argumento del llamado a la función debe ser la variable misma, mientras que, en el caso por REFERENCIA, el argumento debe ser la dirección de la variable a utilizar.

5.2 Paso de argumentos por Valor.

La mejor manera de comprender el paso de argumentos por valor, es haciendo uso de un ejemplo específico, como se muestra en el siguiente programa y esquema:

```
#include <stdio.h>
int     gama( int );  // declaración

int     main( void )
{
        int     aa = 25;
        int     beta = gama( aa );
        ……
        ……
        return 0;
}


int     gama( int  x )
{
        // Sentencias
}
```

El valor de aa se copia en x



El valor de la variable aa, esto es 25, se copia en la variable x, como se muestra en la memoria. En la definición de la función gama, se trabaja entonces con la variable x, la cual si es modificada no afecta en absoluto al valor de la variable aa. Sólo se afecta la copia, esto es la variable x.
El paso por VALOR siempre es una copia del valor de una variable o argumento en otra variable o parámetro.

5.3 Paso de argumentos por REFERENCIA.

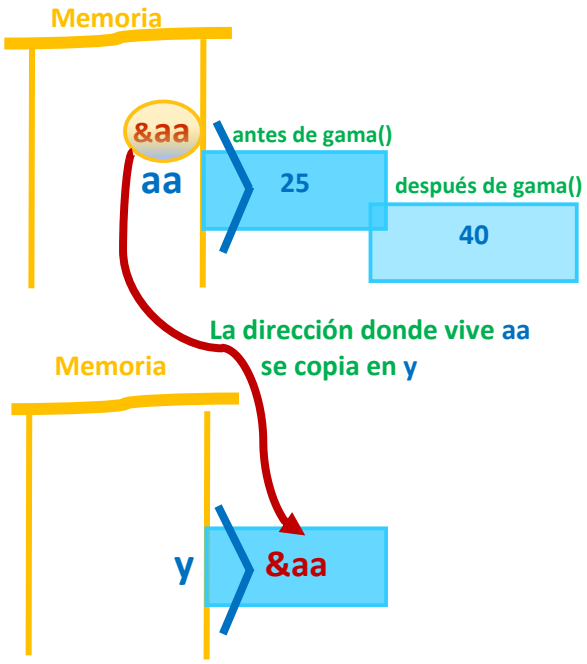
En el paso de argumentos por referencia, siempre se copia la DIRECCIÓN de la variable en un parámetro de la definición de la función, que debe ser del tipo que acepta una dirección, por ejemplo: un puntero.
Puesto que, lo que se transfiere desde el llamado de una función a la definición de la función es la dirección de una variable, declarada generalmente en el programa principal o main(); dentro de la definición de la función, lo que se utiliza es el valor del objeto o variable declarado en el main() que se conoce como el valor original de la variable.
Esto implica, que si se modifica este valor en la función, se cambia el valor de la variable original en el main().
En el paso por valor, esto no ocurre y es su principal diferencia con el paso por referencia.

En el siguiente esquema se muestra el ejemplo anterior, utilizando paso por referencia:

```
#include <stdio.h>
int     gama( int* );  // declaración

int     main( void )
{
        int     aa = 25;
        int     beta = gama( &aa );
        ..............
        ..............
        return 0;
}

int     gama( int*  y )
{
        *y = 40;

        return  *y ;
}
```

se copia en y

Al ejecutarse esta sentencia la variable aa cambia el valor 25 por 40



La dirección donde vive la variable aa es &aa. Esta dirección se copia en la variable y, parámetro que ahora es un puntero que acepta una dirección, como se indica en la memoria. Dentro de la definición de la función gama, la variable y representa una dirección y debe usarse asterisco (*) o corchete [ ] para obtener el valor guardado en esa dirección. Esto indica que el valor a ocupar en la función es el valor original de la variable aa.
En el esquema anterior, *y (en el cuerpo de la función) representa la variable original aa. Luego, la sentencia *y = 40; cambia el valor original de aa desde 25 a 40.
Esto es totalmente diferente al paso de argumentos por valor, que nunca modifica el valor original al cambiar el valor del parámetro dentro del cuerpo de la función.

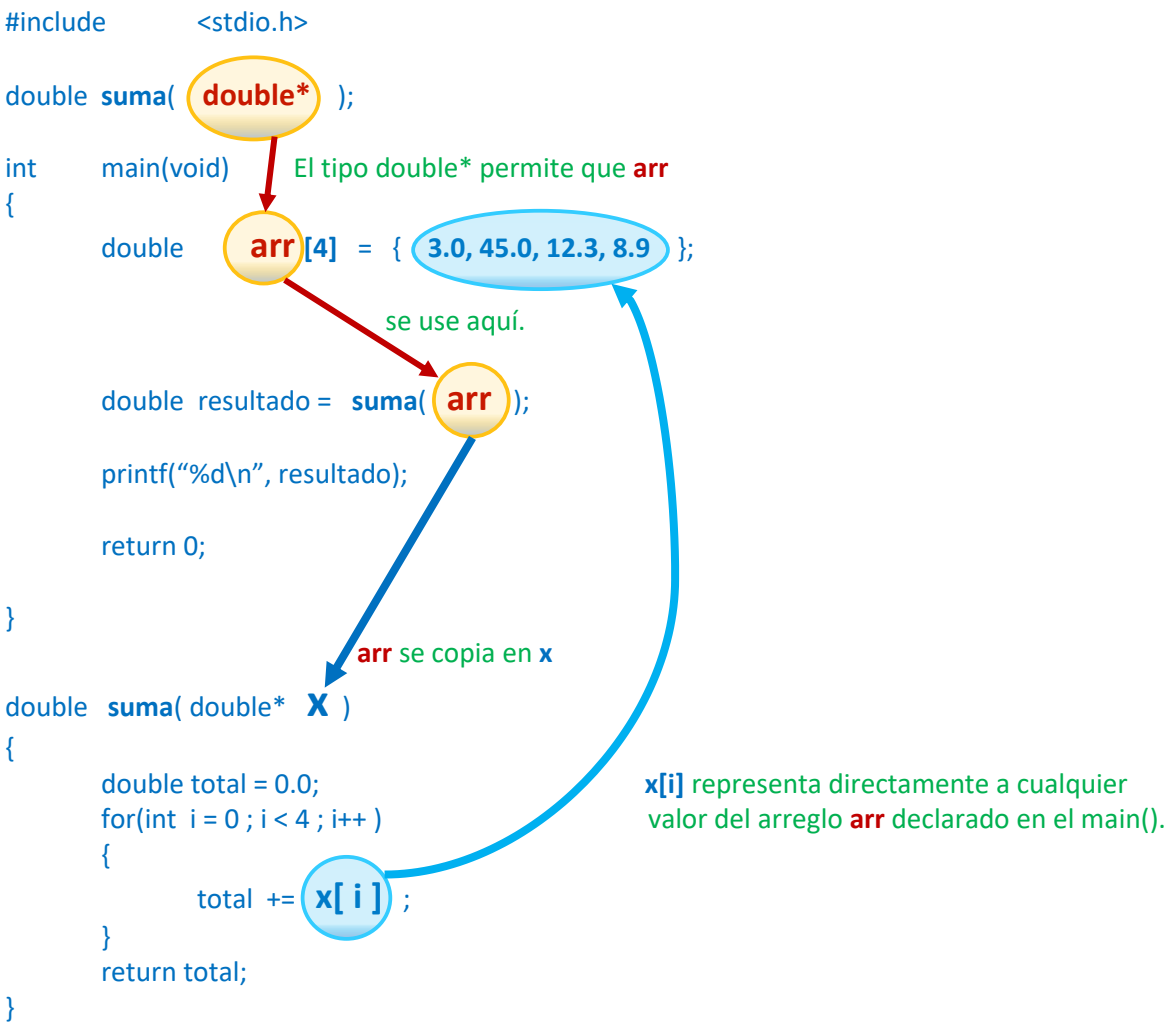
5.4 Paso de un arreglo como argumento de una función.

El uso de arreglos como argumentos de una función es un caso especial. Un arreglo siempre se pasa por referencia. En el Lenguaje C no está permitido copiar un arreglo por valor en un parámetro de una función. Sólo se puede copiar la dirección del arreglo (en general la dirección del primer valor del arreglo) desde el llamado a la función hacia un parámetro en la definición de una función.
Sin embargo, en el capítulo 6 veremos que un arreglo siendo un miembro de una estructura, si se puede pasar por valor porque el Lenguaje C permite que las estructuras puedan pasarse por valor a una función.

Ejemplo:

```
double  arr[4] = {3.0,45.0,12.3,8.9};
```

Usar una función que calcule y retorne la suma de todos los elementos del arreglo arr.
Se debe recordar que el nombre de un arreglo de una dimensión es la dirección del primer elemento del arreglo.



En este ejemplo se copia la dirección del primer elemento del arreglo en la variable x que es un Tipo puntero a double. Al tener la dirección del primer valor del arreglo, se puede tener acceso a cualquier valor usando x[i] (en el cuerpo de la función) o en su defecto *(x + i). Ambos, representan a un valor original de arreglo. El valor que representen depende del índice i. Así, los valores utilizados dentro de la definición de la función son los originales o declarados en el arreglo dentro del main() del programa.

Otra nomenclatura que se usa como parámetros cuando se pasa un arreglo a una función es la siguiente: Para el ejemplo anterior,

```
double   suma( double  [ ] );   // declaración

double  suma(double  x[ ] )    // definición de la función
{
        //sentencias
}
```

El llamado a la función es igual al caso anterior, sólo cambian la declaración y definición. Esta nomenclatura realiza exactamente lo mismo que usando punteros como parámetros. En ambos casos se copia la dirección del primer elemento del arreglo en el parámetro de la definición.
En este libro, no hacemos énfasis en esta última forma de usar arreglos como argumentos.

Importante
Un error común es creer que, al aplicar a un puntero, en el cuerpo de la función, sizeof(x), se obtiene el tamaño del arreglo. Lo que se obtiene al hacer esto, es el tamaño de la dirección arr que son, en general, 4 bytes. La única manera de conocer el tamaño del arreglo que se pasa como argumento, es usar otro parámetro del tipo int, al cual se le asigne el tamaño buscado desde el llamado a la función.

5.5 Retorno de una función.

El retorno de una función es siempre una sola variable que representa a un objeto básico, derivado o complejo.
Retornos de variables básicas son de los tipos float, double, int, char etc. Retornos de variables derivadas son punteros y de variables más complejas son estructuras, funciones o tipos creados con typedef.
La variable u objeto de retorno de una función tiene una existencia temporal y por lo tanto, debe ser asignada a otra variable del mismo tipo, de larga existencia, mediante el llamado a la función.
Cuando se llama a una función que tiene un tipo de retorno distinto a **void**, **la función colapsa** exactamente en el mismo lugar desde dónde se llama, a su variable de retorno. Esta variable de retorno se asigna a otra variable o se utiliza de acuerdo con su entorno. En este libro, usaremos indistintamente las palabras **Transforma** o **Colapsa**, para explicar lo que ocurre al llamado de una función que posee retorno, en el mismo lugar desde dónde se hace el llamado a la función.

Ejemplos:

```
int  func(int); // declaración de la función, retorna un  entero

int main(void)
{
                 9
int    aa   =   func( 3 )  ;  // la función colapsa al retorno que es el valor entero 9
                                // el cual se asigna a la variable aa.

9

   return 0;
}

int func(int   x  )          // x asume el valor 3
{
        return   x*x  ;  // retorna el cuadrado de x, que es el número entero 9
}
```

Cuando el retorno de una función es la palabra void, significa que la función no retorna nada.
En esta situación la función no colapsa a nada y no podría usarse en el ejemplo anterior.

5.6 Uso de exit() en una función.

La función exit(expresión) es parte de la biblioteca estándar <stdlib.h> y se comporta igual a return, excepto que exit() termina el programa, no importando desde que función() se ejecute. En cambio, return termina el programa sólo si se ejecuta en el main().
La expresión en exit() es 0 si el programa termina exitosamente o 1 en caso contrario. También se usan EXIT_SUCCESS o EXIT_FAILURE.

5.7 Variables estáticas en una función.

Lenguaje C no permite que un parámetro en una función sea declarado estático. Sin embargo, es posible declarar variables estáticas dentro de la definición de una función. También, una función puede declararse estática anteponiendo la palabra static antes del tipo de retorno de la función. Una función declarada como estática se utiliza cuando se desea que ésta pueda ser llamada solamente en el archivo en que fue creada. Esto se analizará en detalle en un próximo capítulo.
Cuando se declara una variable como estática dentro de la definición de una función, el objeto de esta variable permanece en memoria por todo el programa. Cada vez que se llama a la función, la variable estática está accesible con el último valor dejado por el anterior llamado a la función. La variable estática se declara una sola vez en tiempo de compilación.
Los siguientes ejemplos muestran la diferencia entre declarar una variable en la definición de una función como estática o automática (no estática).

Ejemplos:

```
#include <stdio.h>

int func(void);

int main(void)
{
        for(int i = 0; i < 5; i++)
        {
                printf("%d\n", func(void));
        }
        return 0;
}

int func(void)
{
        int x = 0;
        x++;
        return x;
}
```

Al ejecutarse el programa anterior se imprimen en la consola los números:

```
1
1
1
1
1
```

Sin embargo, si la variable x en el cuerpo de la función func() se declara estática como se muestra:

```
int func(void)
{
        static int x = 0; // ahora la variable x se declara estática
        x++;
        return x;
}
```

Al ejecutarse el programa con la variable x declarada como **estática**, se imprimen en consola los siguientes números:

```
1
2
3
4
5
```

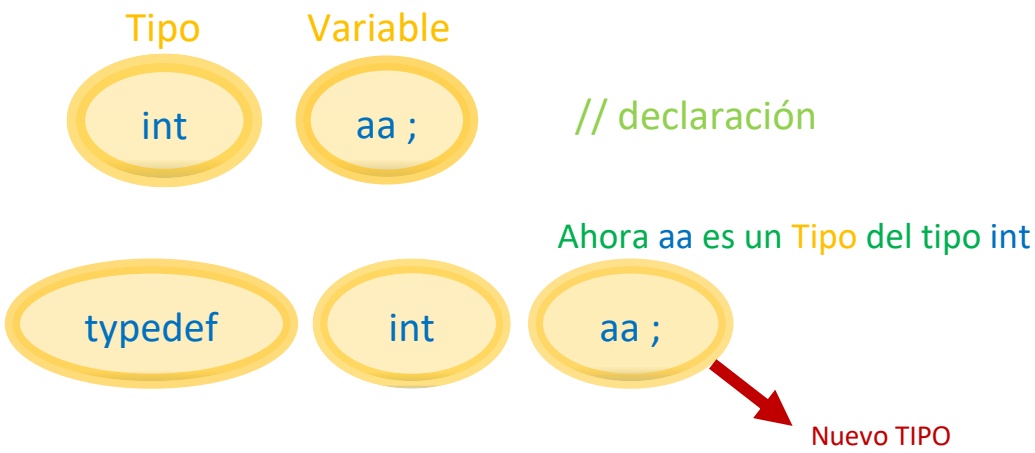
Lo cual es diferente cuando la variable x es no-estática.

En el primer programa, donde la variable x es no-estática; al llamar a la función, la declaración de la variable x obliga a crear un objeto en memoria con el nombre x que guarda el valor 0 (cero). Este objeto y su valor desaparece al término de la ejecución de la función.
Cada vez que dentro del for() se llama a la función func() ocurre lo anterior. Esto implica que la sentencia x++ aumenta en 1 el valor de x que es 0 e imprime cinco veces 1.

En el segundo caso, cuando la variable x es estática, ésta se declara en tiempo de compilación con el valor 0. Así, cuando se entra al run-time o tiempo de ejecución, la variable x no se vuelve a declarar. La sentencia static int x = 0 no se considera cada vez que se llama a la función func(), como ocurre en el programa anterior.
Luego, en el primer llamado a la función func(), la variable x vale 0 y se incrementa en 1 al ejecutarse la sentencia x++, imprimiendo un 1 en la consola. En el segundo llamado a la función dentro del for(),la variable x vale 1 y se incrementa a 2 por la ejecución de la sentencia x++. Para cada pasada por el for() la variable x se incrementa en 1 lo que entrega como resultado una impresión de los números 1 2 3 4 5.

5.8 typedef

En lenguaje C, typedef es una palabra clave o sentencia que permite crear un alias para un Tipo determinado. En otras palabras, convierte el nombre de una variable en un Tipo, igual al tipo con el cual se declaró dicha variable.
Ejemplo:



Luego, se puede usar aa para declarar variables del tipo entero.
Ejemplo:

```
aa   beta = 30;
```

beta es una variable que guarda el entero 30 en memoria.

Importante.
typedef convierte a una variable en un tipo, cuando se aplica a una declaración sin inicialización.

Ejemplo:

```
typedef int aa;   // CORRECTO
typedef int aa = 23;  // ERROR
```

5.9 Punteros a función.

Así como se puede declarar un puntero a objetos int, float, char etc., lenguaje C también permite declarar un puntero a función. La sentencia para ello es la siguiente:

```
int  ( *f ) ( int, double);
```

El * antes de la letra f, y todo ello encerrado en paréntesis, es la forma de declarar un puntero a función. La letra f es el nombre del puntero y puede ser cualquiera permitido por el lenguaje C. En este caso específico, f es un puntero a funciones que retornan un entero y aceptan como parámetros un entero y un decimal de precisión doble. En capítulos anteriores, se vio que todo puntero de una variable básica, adquiere de manera implícita los corchetes [ ] para obtener el valor guardado en la dirección proporcionada por el puntero. De manera análoga, el puntero f adquiere en forma implícita los paréntesis ( ) para tener acceso a la función.
Ejemplo:

```
int func(int, double);  //declaración de una función

int (*ff)(int,double);  //declaración de un puntero a función

ff = &func;  // La dirección donde vive func se asigna al puntero ff
```

La última sentencia también se puede escribir como:

```
ff = func;
```

Lenguaje C asume que cuando se escribe de esta última forma, func decae a su dirección.
Se pueden usar entonces, indistintamente, la función func() o su puntero ff para ejecutar el cuerpo de la función func().

El siguiente programa muestra un posible uso de punteros a función:

```
#include <stdio.h>

int func1(int, int);
int func2(int,int);

int (*ff)(int, int);
```

```
int main(void)
{
        ff = func1;  // también se puede usar &func1 en vez de func1
        printf("%d\n", ff(2,3)); // imprime en consola el valor 5

        ff = func2;
        printf("%d\n", ff(2,3)); // imprime en consola el valor 6

        return 0;
}

int func1(int x, int y)
{
        return x + y;
}

int func2(int x, int y)
{
        return x * y;
}
```

El puntero ff usa los paréntesis () para ejecutar el cuerpo de la función func1 o func2 según el caso.
En el primer printf() del programa, ff(2,3) llama a la función func1 a ser ejecutada con los argumentos 2 y 3.

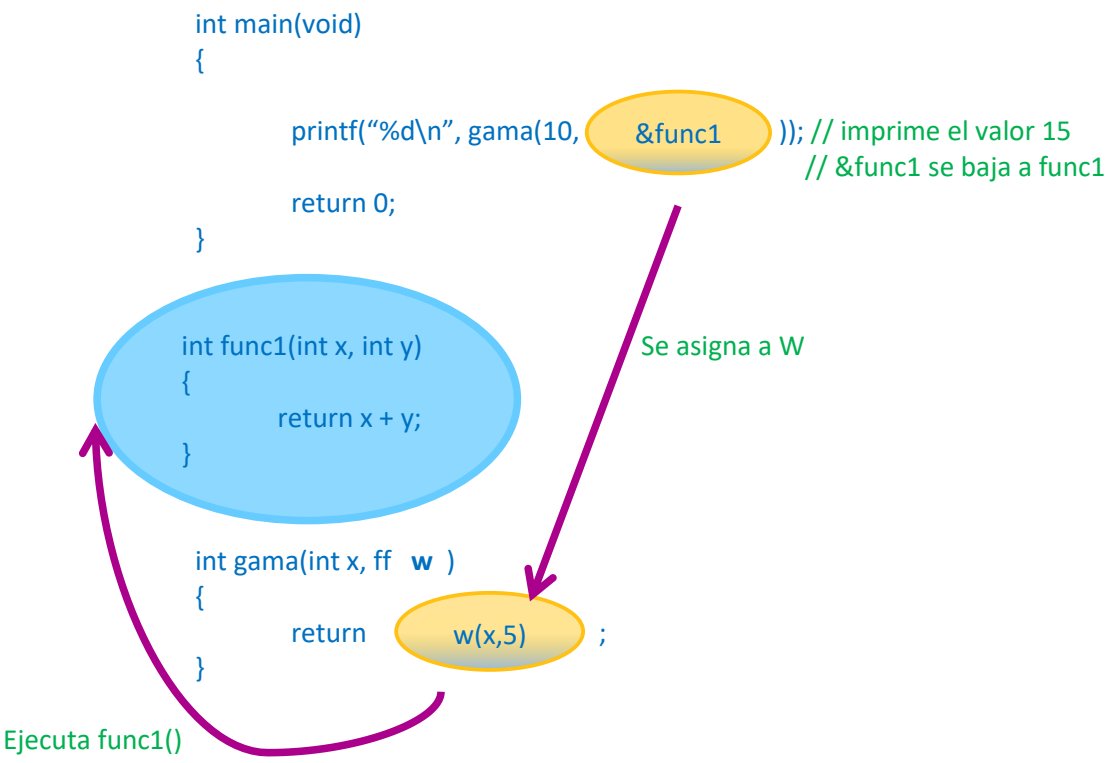
¿Puede una función ser usada como parámetro de otra función?
La respuesta a esta pregunta es no y sí. No es posible directamente, pero convirtiendo una variable puntero-a-función a un nuevo Tipo puntero-a-función es viable.
La sentencia:

```
typedef    int  ( *ff )( int ,int);
```

convierte a ff en un **Tipo puntero-a-función**. Si ff es un **Tipo**, entonces es posible declarar una variable puntero-a-función de este tipo.

El programa anterior es modificado para incluir una función como parámetro de otra función:

```
#include <stdio.h>

typedef int (*ff)(int, int);  // ahora ff es un TIPO puntero-a-función

int func1(int, int);

int gama(int, ff);    // gama incluye como parámetro el tipo ff
```



En la definición de la función gama(), se declara la variable w del Tipo ff, esto es, del tipo puntero_a_función. Esto significa que a la variable w se puede asignar la dirección de cualquier función que retorne un entero y tenga dos parámetros, ambos del tipo int.

En el programa anterior, el llamado a la función gama(10, &func1) dentro del printf, asigna la dirección &func1 al parámetro w. Así, en el cuerpo de la función gama(), cuando se ejecuta el return w(x,5), lo que se hace es ir dónde está en memoria la función func1() y ejecutarla con los parámetros 10 y 5. De esta manera la función gama() retorna el valor 15, que se imprime con el printf().
Vemos, de esta manera, que **typedef** puede crear **Tipos** que permiten declarar variables u objetos que se pueden usar como parámetros de funciones, elementos de arreglos etc., donde estos objetos pueden ser muy complejos.

En el siguiente ejemplo se construye un **Tipo** más complejo usando **typedef** y que es un tipo puntero-a-arreglo cuyos elementos son punteros a funciones:

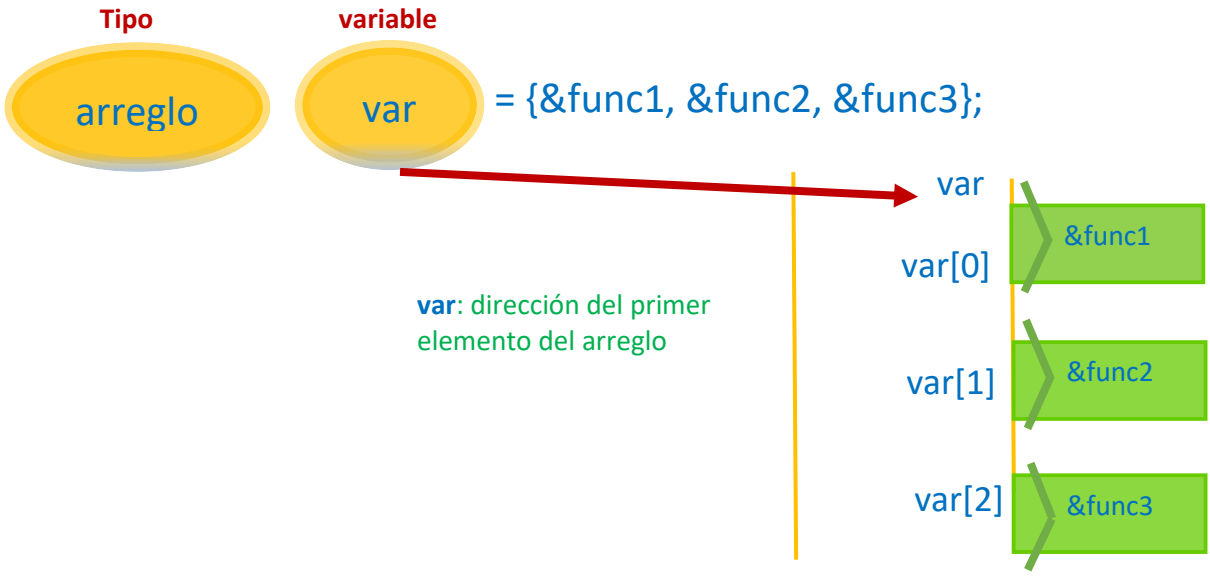
```
        #include <stdio.h>

(1)     typedef  int  (*dat)(double);  // dat es un Tipo puntero-a-función
        int func1(double);
        int func2(double);
        int func3(double);
(2)     typedef  dat  arreglo[3];  // arreglo es un Tipo arreglo con elementos punteros a función
```

```
(3)     int gama(int, arreglo); // gama es una función con parámetros int y arreglo

        int main(void)
        {
(4)             arreglo var = {&func1, &func2, &func3}; // var es una variable del tipo arreglo que
(5)             printf("%d\n" , gama(10, var));          // se inicializa con las direcciones de
                return 0;                                // func1, func2 y func3
        }                                                //¿Por qué no se indica explícitamente
                                                         // la dimensión del arreglo var?

        int gama(int y, arreglo beta)          (6)
        {
                int suma = beta[0](2.0) + beta[1](3.0) + beta[2](4.0);      (7)
                return suma + y;
        }

        int func1(double x)
        {
                return (int)(x+x);
        }

        int func2(double x)
        {
                return (int)(x*x);
        }

        int func3(double x)
        {
                return (int)x;
        }
```

Al compilar y ejecutar este programa se imprime en la consola el número 27. El lector debe comprobar este resultado.
Los números (1), (2), etc. son para hacer una descripción más detallada de las sentencias.

La sentencia (1) `typedef int (*dat)(double);` crea el Tipo dat, que es para variables puntero-a-funciones que retornan un int y aceptan como parámetro un double. Las funciones func1, func2 y func 3 son de este tipo.

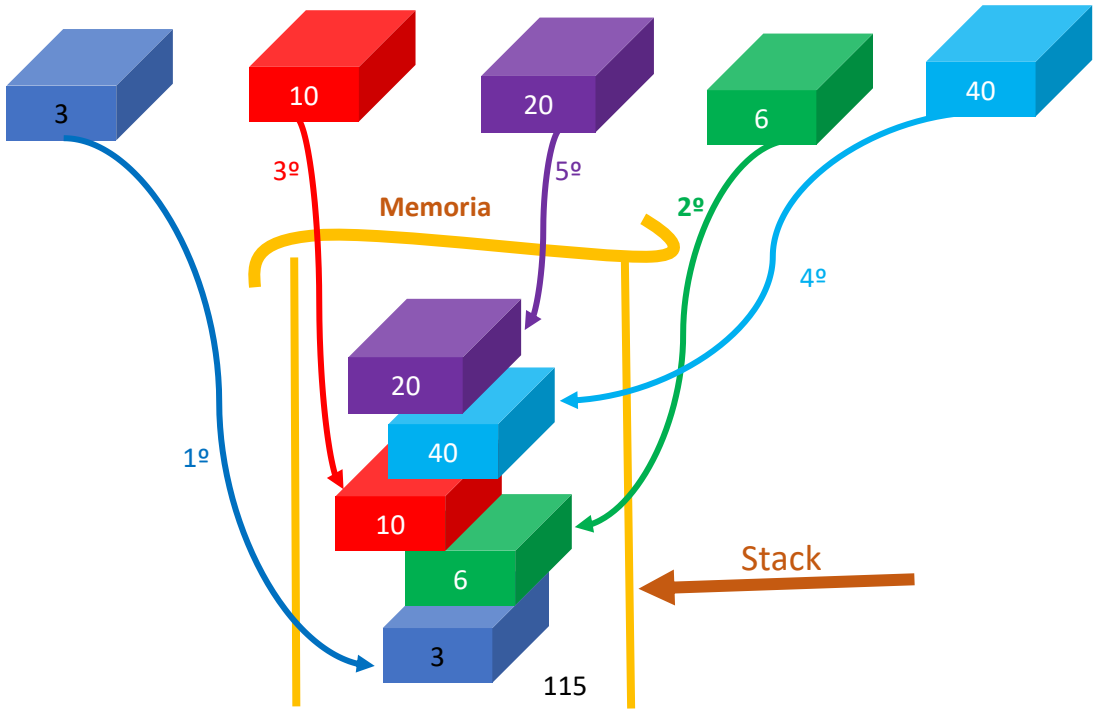
La sentencia (2) `typedef dat arreglo[3];` crea un Tipo arreglo para arreglos de una dimensión que tienen 3 elementos y donde cada elemento es un puntero-a-función del tipo dat.
Esto significa que usando el Tipo arreglo, se puede declarar una variable que en si misma representa a un arreglo de una dimensión con tres elementos que son direcciones dónde viven funciones del tipo dat.
Ejemplo: la sentencia (4) declara la variable var:



var[0] es la dirección en memoria donde vive la función func1(). Como var[0] es un puntero a función, entonces para llamar a la función func1() debe usarse var[0](argumento).
En la definición de la función gama, se declara el parámetro beta (6) y se asigna a éste la dirección var cuando se llama a la función en la sentencia (5). Dentro del cuerpo de la función gama entonces, beta[0] es lo mismo que var[0]. Por ello en la sentencia (7) se utiliza beta[0](2.0) para llamar a la función func1() con el argumento 2.0.
La sentencia (3) es la declaración de la función gama con parámetros del tipo int y arreglo.

5.10 Stack

El **stack** en lenguaje C, es memoria que se estructura y utiliza de una manera determinada. La conveniencia de un stack es que su capacidad de almacenar datos puede crecer o disminuir automáticamente. El stack en lenguaje C es muy utilizado para guardar datos que se usan en el cuerpo de una función. En el capítulo Modelo de Memoria en Lenguaje C se explica el stack con mayor detalle.
En el siguiente esquema se muestra de manera conceptual como funciona un stack:



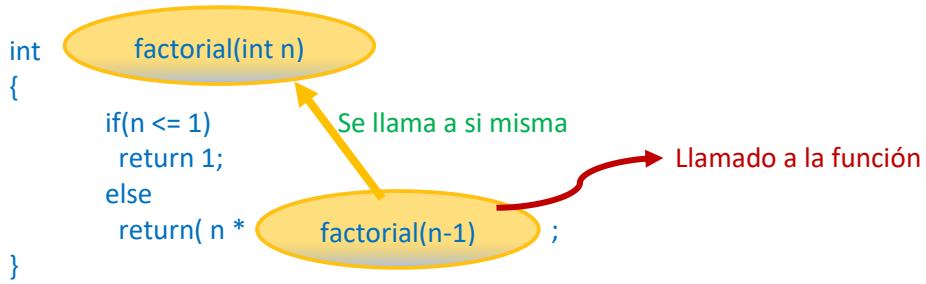
Los bloques son datos o valores que se “empujan” (Push) al stack. Primero se empuja el valor3, luego el valor 6, 10, 40 y finalmente el 20. Los datos al ser empujados al stack se apilan en la memoria. El apuntador SP (Stack Pointer) es la dirección de la memoria donde se guarda el último valor empujado, esto es, el valor 20. El puntero SP siempre apunta al último valor ingresado al stack.
Una vez ingresados los cinco bloques, si se desea “extraer” (Pop) el valor 6, deben primero sacarse los valores 20, 40, 10 y luego el 6. En un stack no es posible extraer valores de manera aleatoria. El último ingresado siempre es el primero en salir. Esto recibe el nombre en inglés **LIFO**, que se lee **last_input – first_output, (última_entrada-primera_salida).**

5.11 Recursión.

Se dice recursión cuando una función se llama a si misma. Lenguaje C proporciona esta forma de ejecutar una función y el clásico ejemplo es usar recursión para calcular la factorial de un número entero.
Ejemplo: calcular 5!
El resultado es 1*2*3*4*5 = 120.

Sin embargo, aunque del punto de vista del cálculo de la factorial es fácil comprender su forma de operar, del punto de vista de la forma como se ejecuta una función que se llama a si misma, no es obvio.

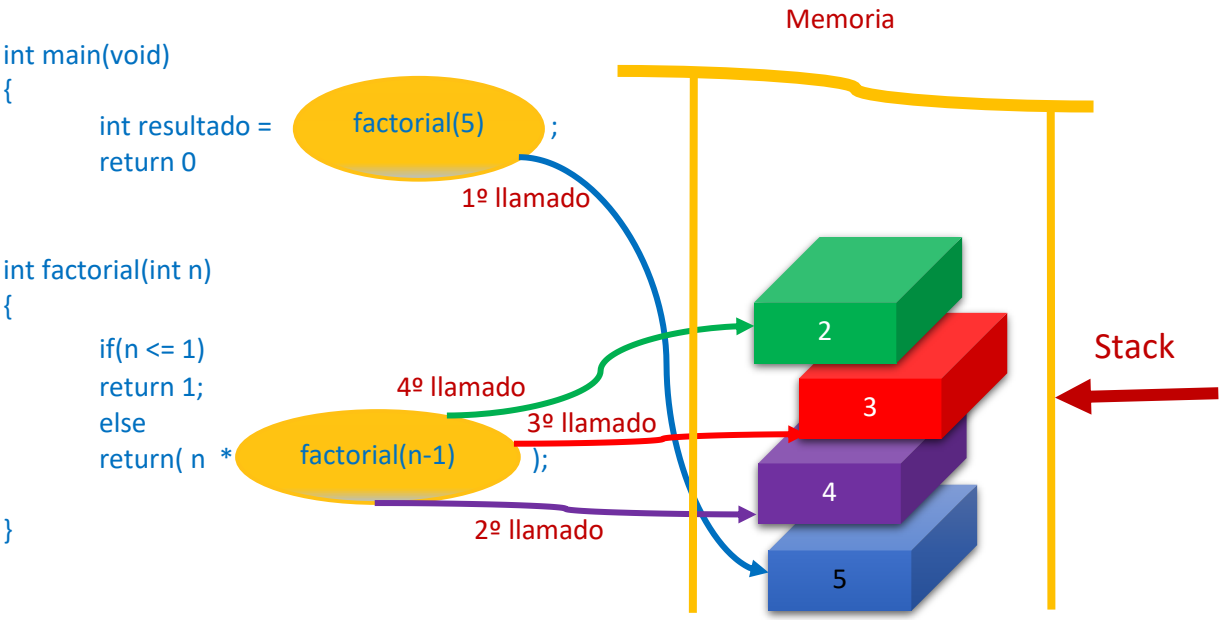
El siguiente programa realiza el cálculo factorial de un número entero n.



La función factoria() se llama a si misma en el cuerpo de dicha función como se indica. En el ejemplo anterior factorial(n – 1) llama a la función factorial con el argumento n-1. Para comprender como se ejecuta esta función factorial() haremos uso del stack visto en 5.10.
Primero es necesario comprender que el primer paso de la compilación de un programa es traducir el texto escrito en lenguaje C a lenguaje assembly. Lenguaje assemply es una etapa intermedia de la compilación que utiliza sentencias que pueden ser traducidas directamente a lenguaje de máquina. El lenguaje assembly de la función factorial() nos dice que la forma de operar de la recursión es primero, guardando en un stack los números n-1 (5 4 3 2), uno por cada llamado de la función, y posteriormente multiplicando estos números 1*2, su resultado * 3, su resultado por *4, su resultado * 5 lo que da el resultado final 120. El número 1 no se guarda en el stack. Se asume como obligatorio.
Conceptualmente la recursión de este ejemplo opera en dos etapas.

1) Cada llamado recursivo guarda el valor n-1 decreciente.
2) Multiplica los números guardados en el stack, de la manera explicada en el párrafo anterior.

El esquema siguiente indica cómo la función recursiva factorial(n) utiliza el stack y obtiene el resultado final:



El primer llamado a la función se hace desde el main() usando factorial(5) y se ingresa al stack el número entero 5. Los tres llamados siguientes se hacen desde el cuerpo de la función usando la sentencia factorial(n-1) e ingresan al stack respectivamente los números enteros 4 3 2 .
Una vez en el stack, el programa assembly multiplica primero el 1*2 , su resultado * 3, su resultado * 4 y finalmente multiplica por 5 obteniendo el valor final 120.

Para aquellos alumnos que deseen ver en profundidad este caso de recursión, se incluye el programa assembly escrito en ARM del programa en lenguaje C:

```
FACTORIAL:  PUSH  {R0,LR}        // va guardando los números 5 4 3 2  en el stack
            CMP   R0,# 1
            BGT   OTRO
            MOV   R0,# 1
            ADD   SP, SP, #8
            MOV   PC, LR
OTRO:       SUB   R0, R0, # 1    // crea los números 4 3 2  guardados por PUSH
            BL    FACTORIAL
            POP   {R1, LR}
            MUL   R0, R1, R0     } estas tres instrucciones multiplican:  1*2*3*4*5 = 120
            MOV   PC, LR
```

Las instrucciones en color rojo comprenden la primera etapa en la cual se guardan los números 5 4 3 2 en el stack. Las instrucciones en color morado guardan el número 1 en un registro y de ahí se salta a las instrucciones de color celeste que comprenden la segunda etapa de multiplicación 1*2*3*4*5.
Como referencia para estudiar este programa assemly se sugiere el texto: Digital Design and Computer Architecture, ARM edition. Autores: Sarah L. Harris & David Money Harris. Capítulo 6. Editorial Morgan Kaufmann, 2016.

5.12 Funciones CallBack

Una función se denomina CallBack cuando el usuario pasa un puntero como argumento, el cual se usa en otra parte que vuelve a llamar a la función principal. Estas funciones son muy adecuadas cuando se desea que cumplan propósitos distintos manejados por punteros de tipo desconocido ("typeless").
En el siguiente ejemplo se tienen dos arreglos, uno con elementos que son enteros y el otro con elementos decimales de alta precisión (double). Se requiere una **función principal** que pueda calcular la suma de los elementos de cualquiera de los dos arreglos. Uno de los argumentos de esta función debe ser sin tipo definido ("typeless") para que pueda recibir cualquiera de los dos arreglos.
Se necesita entonces, dos funciones: una para calcular la suma de los elementos de un arreglo de enteros y otra para calcular la suma de los elementos decimales o double. La función principal debe tener un argumento apropiado para recibir cualquiera de estas **funciones de cálculo**, para lo cual se crea un **Tipo puntero_a_función** para estos efectos.

```
typedef void* (*add)(void*, int);  // declara add como Tipo puntero_a_función


void* suma(add, void*, int);       // función principal
                                   // el segundo argumento es void*, porque no se sabe
                                   // hasta llamar a la función suma() si este argumento
                                   // recibirá un puntero a enteros o double. Ver definición de esta función
                                      // Funciones de cálculo
void* suma_int(void*,int);            // suma_int suma una secuencia de enteros
void* suma_double(void*,int);         //suma_double suma una secuencia de decimales
                                      // los nombres de ambas funciones son del Tipo add

int main(int argc, char *argv[])
{
        int betaInt[5] ={1,2,3,4,15};  // arreglo de enteros
        double betaDouble[5] = {1.0,2.0,3.0,30.0,4.0};  // arreglo de decimales

        double total1 =  * ((double*)suma(suma_double, betaDouble, 5)); //primer uso de suma()
                                  convierte la dirección de retorno sin tipo en una dirección
                                  a un tipo double
                                  el * obtiene el valor que vive en dicha dirección de retorno

        printf("%f\n", total1);  // total1 es la suma de los elementos de arreglo de decimales

        int total2 = *((int*)suma(suma_int, betaInt, 5)); // segundo uso de suma()
```

```
        printf("%d\n", total2);  // total2 es la suma de los elementos de arreglo de enteros

   return 0;
}


void* suma_int(void* x, int n)
{
        static int suma = 0;
        for(int i=0;i<n;i++)
        {
                suma += ((int*)x)[i]; // x es sin tipo, por lo cual se aplica el casting (int*) a x
        }
                return &suma;
}







void* suma_double(void* x, int n)
{
        static double suma = 0.0;
        for(int i=0;i<n;i++)
        {
                suma += ((double*)x)[i];
        }
                return &suma;
}



void* suma(add y,void* z,int n)    // El retorno de suma() es void* porque no se sabe hasta el momento
{                                  // de llamar a esta función que tipo retornará.
        return y(z,n);  // la variable y es del tipo add (puntero_a_función) y puede recibir los punteros
}                       // suma_int  o suma_double. Se debe recordar que se puede usar también
                        // &suma_int  o &suma_double. El efecto es el mismo.
```

La función suma() permite calcular la suma de todos los elementos de un arreglo, sean estos números enteros o decimales.

La función suma_int() calcula la suma de una secuencia de números enteros y devuelve una dirección sin tipo asignado.

La función suma_double calcula la suma de una secuencia de números decimales y devuelve una dirección sin tipo asignado.

En el cuerpo de ambas funciones suma_int y suma_double, la variable suma se declara estática porque se devuelve su dirección. La declaración de variable estática es mandatorio al devolver su dirección.

La función suma() permite calcular la suma total de los elementos de un arreglo, sean enteros o decimales. Para ello, en el llamado a esta función, el primer argumento debe ser un puntero a la función suma_int() o suma_double().

El segundo argumento del llamado a suma() debe ser un puntero al arreglo deseado. El tercer argumento debe entregar el total de elementos en el arreglo.

El retorno de la función suma() es una dirección del tipo void* ("typeless") sin tipo y por lo tanto es necesario hacer un casting al tipo adecuado. Una vez hecho esto, se usa un asterisco a la izquierda para obtener el valor del total calculado.

La función suma() usa punteros a funciones y punteros sin tipo (void*) para decidir en el momento del llamado que clase de resultado se requiere. La función suma() usa otra función (suma_int() o suma_double) que después vuelven a llamar a la función suma() para entregar el resultado final. Esta es una función **CallBack**.

5.13 Función que retorna otra función.

¿Cómo una función puede retornar otra función? Esto es posible si se crea un Tipo puntero-a-función y se utiliza como retorno de una función. En la declaración de una función, se debe establecer como retorno un tipo que es un Tipo puntero-a-función.
Mediante el siguiente ejemplo, es posible clarificar el uso de una función cuyo retorno es otra función.
Ejemplo:

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

typedef double (*f)(int, int);        // f es un Tipo Puntero a Función

typedef double (*f2)(double);     // f2 es un Tipo Puntero-a-Función

f2 func(f, double*); // func() es una función que acepta dos parámetros: el primero es un puntero
                     // a función y el segundo es un puntero a double. Retorna una función del Tipo f2.
                     // El puntero a función debe ser para funciones que aceptan dos enteros y retornan
                     // un double.

double gama(double);  // gama() es una función que acepta un double y retorna un double

double beta(int, int);   // beta() es una función que acepta dos parámetros int y retorna un double

int main(int argc, char *argv[])
```

```
{
   double  m  = 0.0;     // variable m
        double w = func( &beta, &m )(m);      // &beta es la dirección en memoria donde
                                              // está la función beta().
                                              // &beta se asigna a la variable X (abajo en
                                              // la definición de la función func()).
                                              // &m se asigna a la variable Z (abajo en
                                              // la definición de la función func()).
        printf("%f\n",w);
   return 0;
}

f2      func( f x, double* z ) // X es una variable del tipo f
{
        double  a  =  x(2,4)  ;        // X(2.4) es lo mismo que la función beta(2,4).
        *z = a;                        // a modifica la variable original m en el main().
        return &gama;                  // &gama es la dirección en memoria donde
                                       // está la función gama(), por ende, apunta a una
                                       // función del Tipo f2.
}


double gama(double y)
{
        return y+y;
}

double  beta(int h,int k)
{
        return (double)(h+k);
}
```

Todo esto colapsa al retorno &gama que es lo mismo que gama. Luego queda gama(m) que retorna un número double y lo asigna a la variable w.

La función func() retorna &gama, que es la dirección en memoria(stack), donde reside la función gama(). También func() acepta como parámetro la variable X que es del Tipo puntero-a-función y se asigna a ésta la dirección &beta, que es la dirección donde reside la función beta().
En el cuerpo de la función func(), se ejecuta X(2,4) que lo mismo que beta(2,4). El retorno de esta función se asigna a *z, es decir se asigna a la variable m en el main(). Así, en el cuerpo de func() se cambia el valor de m que posteriormente se usa como argumento de gama(m).

5.14 Funciones inline.

Las funciones inline, son funciones donde el llamado a la función es reemplazado por todo el cuerpo de la función y al ser ejecutado en run-time, no tiene tiempo de latencia. El tiempo de latencia, ocurre cuando se tiene que ir a otro lugar de la memoria a ejecutar el cuerpo de la función.
Para que una función sea reconocida como inline por el compilador, debe anteponerse la palabra inline antes del tipo de retorno.
Ejemplo:

```
inline    double promedio(int x, double y){ return (double)x + y };
```

En general, una función inline es una función de muy pocas sentencias y ubicada antes del main() con su completa definición, como en el ejemplo de arriba.
El compilador, no necesariamente obedece a la directiva inline y puede, por lo tanto, no compilarla como inline.
Cuando una función es definida como inline, ésta no puede ser considerada como extern, ya que el llamado de ella desde otros archivos sería reconocido como un error.
Una forma de evitar este error es usar la palabra static antes de inline. De esta forma, la función sólo se puede llamar en el archivo en el cual se creó.

5.15 Ejercicios resueltos

No hacer copy-paste de estos ejercicios, porque algunos símbolos de Word son erróneos para el compilador.

5.15.1 Con los siguientes datos, realizar lo siguiente:

```
float   temperatura[10] = {35.9,36.2,37.0,36.8,38.3,39.0,37.9,35.6,37.1,36.6};
char*   p1 = “juan”;
char*   p2 = “luisa”;
char*   p3 = “jaime”;
char*   p4 = “jasna”;
char*   p5 = “clara”;
char*   p6 = “jose”;
char*   p7 = “antonio”;
char*   p8 = “ricardo”;
char*   p9 = “cecilia”;
char*   p10 = “claudio”;
```

5.15.1.1 Asociar cada nombre con cada valor del arreglo temperatura, de manera consecutiva.

Para asociar cada nombre con cada valor del arreglo, se crea un arreglo de punteros, de manera que cada puntero en el arreglo coincide con una posición de un valor de temperatura.

```
char*   nombres[10] = {p1,p2,p3,p4,p5,p6,p7,p8,p9,p10};
```

Este arreglo nos permite asociar el nombre juan a la temperatura 35.9, el nombre clara a la temperatura 38.3 etc.

5.15.1.2 Crear una función que retorne el nombre que tiene la temperatura más alta.

Se crea directamente la definición de la función. Como se debe tener acceso a arreglos, se usarán como parámetros de la función punteros al tipo que corresponda.

```
char*   nombreTemperaturaMasAlta(float* temp, char** name)
{
        int       index=0;
        float tempInit = temp[0];
        for(int i=1; i<10 ; i++)
        {
                if( tempInit < temp[i])
                {
                        tempInit = temp[i];   // tempInit mantiene la temperatura más alta.
                        index = i;            // index guarda la posición en el arreglo de la
                                                      // temperatura más alta.
                }
        }
        return name[index];
}
```

El parámetro temp recibe el argumento temperatura y el parámetro name recibe el argumento nombres desde el llamado a la función.

5.15.1.3 Modifique la función creada en 5.15.1.2 para que retorne el nombre y se conozca en el programa principal su temperatura.

Como la función ya tiene un retorno utilizado, y sólo se puede devolver una variable, se usa otro parámetro para devolver hacia el programa principal el valor de la temperatura. Esto se consigue usando un puntero a flotante. Se construye otra función con otro nombre.

```
char*   nombreYtemperatura(float* temp, char** name, float* valorTemp)
{
        int     index=0;
        float tempInit = temp[0];
        for(int i=1; i<10 ; i++)
        {
                if( tempInit < temp[i])
                {
                        tempInit = temp[i];  // tempInit mantiene la temperatura más alta.
                        index = i;           // index guarda la posición en el arreglo de la
                                                // temperatura más alta.
                }
        }
        *valorTemp = tempInit;      // devuelve la temperatura más alta al programa principal
        return name[index];
}
```

En el programa principal main(), se debe declarar una variable flotante y pasar como argumento esta variable al parámetro valorTemp. Al ejecutarse la función nombreYtemperatura(), se cargará en dicha variable, el valor de la temperatura más alta.

5.15.1.4 Crear un programa completo con las funciones anteriores.

En el programa principal se declara la variable valtemp, la cual recibirá el valor de la temperatura cuando se ejecute la función nombreYtemperatura() dentro del printf(). El valor de la temperatura se carga en esta variable, porque dentro de esta función, se trabaja con el valor original de la variable valtemp. Esto ocurre cuando en el llamado a la función, se entrega como argumento la dirección de la variable valtemp (&valtemp).

En ambas funciones se requiere que el segundo parámetro name se declare como puntero-a-puntero a carácter. Esto es char** name.
El tipo char** se requiere cuando la variable-puntero a declarar, es la dirección de un objeto el cual guarda otra dirección. En este caso nombres, que es el nombre del arreglo, apunta al primer elemento del arreglo que también es una dirección, p1. Luego, el parámetro name es una dirección a otra dirección, por lo cual para su declaración en las funciones nombreYtemperatura() y nombreTemperaturaMasAlta(), se debe usar el TIPO char**.

```
#include <stdio.h>
char*    nombreTemperaturaMasAlta(float*, char**);
char*    nombreYtemperatura(float*, char**, float*);


int main(int argc, char *argv[])
{
   float  temperatura[10] = {35.9,36.2,37.0,36.8,38.3,39.0,37.9,35.6,37.1,36.6};
          char*    p1 = "juan";
          char*    p2 = "luisa";
          char*    p3 = "jaime";
          char*    p4 = "jasna";
          char*    p5 = "clara";
          char*    p6 = "jose";
          char*    p7 = "antonio";
          char*    p8 = "ricardo";
          char*    p9 = "cecilia";
          char*    p10 = "claudio";
          float valtemp; // declara variable para recibir valor de temperatura

          char*    nombres[10] = {p1,p2,p3,p4,p5,p6,p7,p8,p9,p10};

          // llama a la función nombreTemperaturaMasAlta e imprime el nombre
          printf("%s\n",nombreTemperaturaMasAlta(temperatura,nombres));

          // llama a la función nombreTemperaturaMasAlta e imprime el nombre y temperatura

          printf("nombre= %s  ",nombreYtemperatura(temperatura,nombres,&valtemp));
          printf("temperatura= %f grados celcius\n",valtemp);

   return 0;
}

char*    nombreTemperaturaMasAlta(float* temp, char** name)
          {
                    int       index=0;
                    float tempInit = temp[0];
                    for(int i=1; i<10 ; i++)
                    {
                              if( tempInit < temp[i])
                              {
                                        tempInit = temp[i];   // tempInit mantiene la temperatura más alta.
                                        index = i;                 // index guarda la posición en el arreglo de la
                                                                  // temperatura más alta.
                              }
                    }
                    return name[index];
}
```

```
char*   nombreYtemperatura(float* temp, char** name, float* valorTemp)
        {
                int     index=0;
                float tempInit = temp[0];
                for(int i=1; i<10 ; i++)
                {
                        if( tempInit < temp[i])
                        {
                                tempInit = temp[i];   // tempInit mantiene la temperatura más alta.
                                index = i;            // index guarda la posición en el arreglo de la
                                                      // temperatura más alta.
                        }
                }
                *valorTemp = tempInit;          // devuelve la temperatura más alta al programa principal
                return name[index];
}
```

5.15.2 Dado los siguientes datos, realizar lo siguiente:

```
        char    palabra1[7] ={"Viento"};
        char    palabra2[6 ] ={"Solar"};
```

5.15.2.1 Crear un arreglo que pueda contener ambas palabras concatenadas.

Para crear un arreglo, primero se debe saber cuánta memoria se debe reservar.

```
        int     largo1 = sizeof(palabra1);
        int     largo2 = sizeof(palabra2);
        int     memoria = largo1 + largo2 ;        //memoria es cuanta cantidad de
                                                   //bytes a reservar.
        char    ambasPalabras[memoria];            // crea un arreglo con capacidad para ambas
                                                   // palabras.
```

5.15.2.2 Crear una función que concatene ambas palabras, separadas por un espacio, y las deposite en el nuevo arreglo.

```
void    concatenar(char* una, char* dos, char* ambas, int largo1, int largo2)
{
                for(int i = 0; i < largo1-1; i++)
                {
                        ambas[i] = una[i];
                }
                ambas[largo1-1] = " ";
                for(int i = 0; i < largo2; i++)
                {
                        ambas[i] = dos[i];
                }
}
```

5.15.2.3 Crear un programa completo.

```
#include <stdio.h>
#include <stdlib.h>
void    concatenar(char*,char*,char*,int,int);

int main(int argc, char *argv[])
{
        char    palabra1[7] ={"Viento"};
        char    palabra2[6]={"Solar"};
        int     largo1 = sizeof(palabra1);
        int     largo2 = sizeof(palabra2);

        int     memoria = largo1 + largo2 ;     //memoria es cuanta cantidad de
                                                //bytes a reservar.
        char    ambasPalabras[memoria];
        concatenar(palabra1,palabra2,ambasPalabras,largo1,largo2);

        printf("ambas palabras= %s\n",ambasPalabras);   // imprime palabras concatenadas.

   return 0;
}
void    concatenar(char* una, char* dos, char* ambas, int largo1, int largo2)
        {
                for(int i = 0; i < largo1-1 ; i++)  // copia primera palabra
                {
                        ambas[i] = una[i];
                }

                ambas[largo1-1] =' ';           // pone un espacio.

                for(int i = 0 ; i  <  largo2; i++)      // copia segunda palabra
                {
                        ambas[largo1 + i] = dos[i];
                }

        }
```

5.15.2.4 Modifique el programa para ingresar las palabras por teclado de tamaño desconocido, previamente.

La modificación debe hacerse solamente en el programa principal. Para ingresar las palabras se utiliza scanf(). Primero se ingresan los tamaños máximos de cada palabra. Después se ingresan las palabras.

```
printf(" Ingresar tamaño máximo palabra1= ");
int largo1Maximo, largo2Maximo, largo1=1, largo2=1; // largo1 y largo2 se inicializan en 1
                                                    // ¿Por qué?.
scanf("%d", &largo1Maximo);
printf(" Ingresar tamaño máximo palabra2= ");
scanf("%d", &largo2Maximo);

char    palabra1[largo1Maximo];
char    palabra2[largo2Maximo];
printf(" Ingresar palabra1= ");
scanf("%s", palabra1);
printf(" Ingresar palabra2= ");
scanf("%s", palabra2);
```

El tamaño de las palabras ingresadas es muy probable que siempre sean menores a su tamaño máximo, por lo cual es necesario calcular por código el largo de cada palabra ingresada. Con estos datos se reserva la memoria adecuada para el arreglo ambasPalabras[].

```
int k = 0;
while( palabra1[k] != '\0' )
{
        largo1++;
        k++;
}
k = 0;
while( palabra2[k] != '\0' )
{
        largo2++;
        k++;
}
```

Esto se puede hacer usando strlen() de la biblioteca <string.h>, pero aún no se ha visto.

5.15.2.5 Crear un programa completo con la modificación anterior.

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

void	concatenar(char*,char*,char*,int,int);

int main(int argc, char *argv[])
{
	printf(" Ingresar tamano maximo palabra1= ");
	int largo1Maximo, largo2Maximo, largo1=1, largo2=1;
	scanf("%d", &largo1Maximo);
	printf(" Ingresar tamano maximo palabra2= ");
	scanf("%d", &largo2Maximo);

	char	palabra1[largo1Maximo];
	char	palabra2[largo2Maximo];
	printf(" Ingresar palabra1= ");
	scanf("%s", palabra1);
	printf(" Ingresar palabra2= ");
	scanf("%s", palabra2);

	int k = 0;
	while( palabra1[k] != '\0' )
	{
		largo1++;
		k++;
	}
	k = 0;
	while( palabra2[k] != '\0' )
	{
		largo2++;
		k++;
	}

	int	memoria = largo1 + largo2 ;
	char	ambasPalabras[memoria];
	concatenar(palabra1,palabra2,ambasPalabras,largo1,largo2);
	printf("ambas palabras= %s\n",ambasPalabras);
	 return 0;
}

void	concatenar(char* una, char* dos, char* ambas, int largo1, int largo2)
	{
		for(int i = 0; i < largo1-1 ; i++)
		{
			ambas[i] = una[i];
		}
```

```
        ambas[largo1-1] =' ';

        for(int i=0;i < largo2;i++)
        {
                ambas[largo1 + i] = dos[i];
        }
}
```

¿Qué pasa, si el usuario se equivoca en el ingreso de tamaños de palabras y en vez de ingresar números, ingresa caracteres? Modificar el programa para precaver esta situación.
¿Qué pasa, si el usuario ingresa una palabra con más caracteres que el máximo permitido? Tomar en cuenta esta situación y modificar el programa.

5.15.3 En el siguiente programa, se ingresa por teclado una frase que contiene palabras separadas por espacio. Escribir un código para realizar esto y completar las acciones que se indican posteriormente.

```
int     largoMaximoFrase;
printf("Ingresar tamaño de frase= ");
scanf("%d",&largoMaximoFrase);
char    frase[largoMaximoFrase];
printf("Ingresar frase= ");
scanf(" %[^\n]", frase);          // se usa formato [^\n] y no S ¿Por qué?.
        espacio                   // se deja un espacio después del primer " , esto es, "  %[^^\n]
                                  // ¿Por qué?. Pruebe sin el espacio ¿Qué sucede?.
printf("%s\n", frase);
```

5.15.3.1 Ingrese una palabra o ristra(string), de tres caracteres (sin espacio entre ellos) y codifique una función que indique si esta palabra se encuentra en la frase y cuántas veces se repite.
Ejemplo: sea la frase,

A la hora del a**man**ecer en el sur de chile

se ingresa ristra: **man**
Se encuentra en la palabra amanecer y no se repite en la frase.
código para la ristra:

```
        char    ristra[4];
        printf("Ingrese tres caracteres= ");
        scanf("%s", ristra);
        printf("%s\n",ristra);
        int largo = 0;
        int k = 0;
        while( frase[k] != '\0' )
        {
                largo++;
                k++;
        }
```

```
int     encuentraRistra(char* frase, char* ristra, int largo)
        {
                int repite = 0;
                for(int i = 0; i < largo – 2; i ++)
                {
                   if(ristra[0] == frase[i] && ristra[1] == frase[i + 1] && ristra[2] == frase[i + 2])
                        {
                                repite++;
                        }
                }
                return repite;
        }
```

5.15.4 Ingresar tres o más palabras, separadas por espacio.

Reserve memoria en el main() que permita guardar un igual número de caracteres a los ingresados por teclado.
Cree una función que permita invertir la secuencia de palabras ingresadas y guardarlas en el espacio reservado en el main().
Imprima la nueva secuencia de palabras invertidas.
La inversión que se desea se explica en el siguiente ejemplo:
secuencia ingresada: En el sur llueve mucho
secuencia invertida: mucho llueve sur el En

El programa completo se muestra a continuación:

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

void    invertirFrase(char*, char*, int, int);

int main(int argc, char *argv[])
{
char* frase = malloc(sizeof(char)*100);             //Guarda espacio en memoria para la frase
        printf("Ingrese frase de minimo tres palabras= ");
        scanf("%[^\n]",frase);
        printf("%s\n",frase);
        int largo=0, numeroPalabras=1;
        while(frase[largo] != '\0')                  //Calcula el total de caracteres en la frase
                {                                    //y el número de palabras
                        if(frase[largo] == ' ')
                        {
                                numeroPalabras++;
                        }
                        largo++;
                }
        printf("Total caracteres = %d\n", largo);
        printf("Total palabras en frase = %d\n",numeroPalabras);
        char* fraseReves = malloc(sizeof(char)*(largo+1));          //reserva memoria dinámica
```

```
        invertirFrase(frase, fraseReves, largo, numeroPalabras);
                printf("\n");
                printf("Frase al reves:\n");
                printf("%s\n",fraseReves);          //Imprime frase al revés
                printf("\n");

         free(frase);
         return 0;
}


void    invertirFrase(char* frase, char* fraseReves, int largo, int numeroPalabras)
{
        int k = largo-1;
        int contador=0;
        int espacio=0;
        for(int i=0;i < numeroPalabras;i++)
        {

                while(frase[k] != ' ' && k >= 0)        //Comienza desde el final de la frase
                {                                       //hacia el principio de la frase
                        k--;
                        contador++;
                }
                        for(int j=0; j < contador; j++)
                        {
                                fraseReves[espacio + j] = frase[k+1+j];
                        }

                        fraseReves[espacio+contador] = ' ';
                        espacio += contador+1;
                        contador=0;
                        k--;
        }
                fraseReves[largo+1] = '\0';

}
```

La función usa como parámetros dos punteros a carácter y dos int. El primer parámetro permite el acceso a los caracteres ingresados por teclado y el segundo parámetro permite ingresar la secuencia de palabras invertidas en la memoria reservada en el main(), carácter a carácter. La reserva de memoria en el main(), para la secuencia invertida, usa la función malloc() de manera dinámica al usar la variable largo, para calcular en run-time el total de espacio de memoria requerido.

5.15.5 Ingresar RUT por teclado con errores.

Ingresar el RUT de una persona con errores y crear una función que lo guarde en memoria sin los errores y lo imprima correctamente. El programa debe solicitar el ingreso de nuevo, si el número de dígitos ingresados es incorrecto. Ejemplo:
Rut ingresado: 4. 5,,56-7,78 9—0
Rut corregido: 455677890
Rut impreso: 45.567.789-0

El programa completo es el siguiente:
La función funcRut() cumple tres roles: a) Solicita ingresar el rut por teclado. b) Calcula el número de dígitos; si es distinto de 8 o 9, pide de nuevo ingresar el rut. c) Elimina todos los errores y guarda los dígitos ingresados en memoria sin puntos ni guión.
La función imprimeRut() imprime los dígitos del rut guardado, incorporando los puntos y guión en las posiciones correctas.

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

_Bool funcRut(char*,char*,int*);      // Función que elimina errores y guarda rut correcto en memoria.
void imprimeRUT(char*,int);           // Función que imprime el rut de manera correcta incluyendo puntos
                                      // y guión.

int main(int argc, char *argv[])
{
        char* rut = malloc(sizeof(char)*20);
        char* RUT = malloc(sizeof(char)*10);
        int largoRUT;
        do{                                     // solicita ingresar rut y volver a ingresar si la cantidad de
                printf("Ingrese RUT = ");       // dígitos es distinta de 8 o 9.
                scanf(" %[^\n]",rut);
                                                ¿Por qué es importante dejar un espacio aquí?
                printf("Rut Ingresado: %s\n",rut);
        }while( !funcRut(rut, RUT, &largoRUT));

        imprimeRUT(RUT, largoRUT);

        free(rut);
        free(RUT);
   return 0;
}
```

```
//Definición de funciones
_Bool funcRut(char* rut, char* RUT,int* largoRUT)
{
	int largo=0;
	while(rut[largo] != '\0')  //Calcula el total de caracteres en el RUT
			largo++;

	int k=0;
	for(int i=0;i<largo;i++)
		{
			if(rut[i] >= 48 && rut[i] <= 57)
			{
				RUT[k] = rut[i];
				k++;
			}
		}

		if(k==8 || k==9)
		{
			*largoRUT = k;   // Usando punteros se devuelve el número de dígitos a la variable
			return 1;        // largoRUT declarada en el main().
		}else
			return 0;

}

void imprimeRUT(char* RUT,int largoRUT)
{
		printf("RUT CORRECTO: ");
		if(largoRUT == 8)
		{
			printf("%c%c%c%c%c%c%c%c%c\n",RUT[0],'.',
			RUT[1],RUT[2],RUT[3],'.',RUT[4],RUT[5],RUT[6],'-',RUT[7]);
		}
		else if(largoRUT == 9)
		{
			printf("%c%c%c%c%c%c%c%c%c%c\n",RUT[0],
			RUT[1],'.',RUT[2],RUT[3],RUT[4],'.',RUT[5],RUT[6],RUT[7],'-',RUT[8]);
		}
}
```

5.15.6 Considere el alfabeto inglés de 26 caracteres.

Ingrese por teclado, una frase de varias palabras que contengan una mezcla de letras mayúsculas y minúsculas del alfabeto inglés.

Cree una función que permita encriptar la frase ingresada de acuerdo con las siguientes reglas:

a) Cada carácter debe cambiarse por otro carácter que esté un número determinado de posiciones más adelante en el alfabeto. El número de posiciones, en adelante clave, se debe ingresar por teclado al comenzar el programa.
b) Si el carácter de reemplazo pasa más allá de la z, entonces vuelva al principio del alfabeto y continúe. Ejemplo: si el carácter a cambiar es la letra Y con una clave 3, se obtiene la letra B.

Cree una segunda función que permita desencriptar una frase ya encriptada. Use como clave: 26-clave.

Este algoritmo para encriptar fue inventado por el emperador romano Julio César.

Programa completo.

Se entrega el programa que incluye la función para encriptar. **Ud., programe la función para desencriptar.**

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

void encriptarFrase(char*,char*,int,int, char*, char*);

int main(int argc, char *argv[])
{
        char alfaMIN[27] = {"abcdefghijklmnopqrstuvwxyz"};          //alfabeto inglés
        char alfaMAY[27] = {"ABCDEFGHIJKLMNOPQRSTUVWXYZ"};
        int clave;
        char* frase = malloc(sizeof(char)*100);
        printf("Ingrese frase a encriptar= ");
        scanf("%[^\n]", frase);
        printf("Ingresar Clave entre 1 y 25 = ");
        scanf("%d",&clave);
        if(clave < 1 && clave > 25)
                {
                        printf("Clave erronea\n");
                        exit(-1);
                }
        int largo = 0;
        while(frase[largo] != '\0')
        {
                largo++;
        }

        char* fraseEncriptada = malloc(sizeof(char)*(largo+1));  // reserva memoria para frase encriptada

        encriptarFrase(frase,fraseEncriptada,largo,clave, alfaMIN, alfaMAY);
        fraseEncriptada[largo] = '\0';
        printf("Frase encriptada= %s\n",fraseEncriptada);

   return 0;
}
```

```
void encriptarFrase(char* frase,char* fraseEncriptada,int largo, int clave
	, char* alfaMIN, char* alfaMAY)
{
	int index;				// indica posición de letra en alfabeto inglés
	for(int i=0; i < largo ; i++)
	{
		if(frase[i] <= 122 && frase[i] >= 97)	// si letra es minúscula
		{
			index = frase[i] - 97 + clave;	// se calcula index
			if(index > 25)
				{
					index -= 26;
				}
			fraseEncriptada[i] = alfaMIN[index];
		}
		else if(frase[i] <= 90 && frase[i] >= 65)	// si letra es mayúscula
		{
			index = frase[i] - 65 + clave;	// se calcula index
			if(index > 25)
				{
					index -= 26;
				}
			fraseEncriptada[i] = alfaMAY[index];
		}
		else if(frase[i] == ' ')
		{
			fraseEncriptada[i] = ' ';
		}
		else
		{
			printf("Frase erronea\n");
			exit(-1);
		}
	}
}
```

5.15.7 Evaluar polinomio

Crear un programa que permita evaluar un polinomio de cualquier orden, con un orden máximo de 9. Ingresar por teclado los coeficientes del polinomio, del tipo double, separados por espacio. El número total de coeficientes determinará el orden máximo del polinomio a calcular.
Ejemplo: si se ingresa la siguiente secuencia de datos:

1.2 0.0 3.4 5.6 0.0 98.0

El polinomio para evaluar es:

$$1.2 + 3.4*X^2 + 5.6*X^3 + 98.0*X^5$$

Se debe crear una función que evalúe el polinomio ingresado por teclado.
Los coeficientes se deben ingresar en orden ascendente del orden del polinomio. Esto es, primero a1 seguido de a2 y así sucesivamente. El polinomio que da descrito por:

$$a_0 + a_1*X^1 + a_2*X^2 + a_3*X^3 + \ldots\ldots\ldots + a_n*X^n$$

El programa completo se muestra a continuación:
Se deben ingresar todos los coeficientes, aunque sean ceros, hasta el orden del polinomio por evaluar.

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

double evaluaPolinomio(double*,int);

int main(int argc, char *argv[])
{
   double* a = malloc(10*sizeof(double));
        int orden;
        printf("Ingrese orden de polinomio= ");
        scanf("%d",&orden);
        printf("Ingrese %d coeficientes separados por espacio: ",orden+1);
        for(int i=0;i<orden+1;i++)
        {
                scanf("%lf",&a[i]);
        }

        printf("valor polinomio= %f\n", evaluaPolinomio(a,orden));

   free(a);
   return 0;
}
```

```
double evaluaPolinomio(double* a,int orden)
{
        double x;
        printf("Ingrese valor de X =");
        scanf("%lf",&x);
        double valorPolinomio = 0.0;
        for(int i = 0; i < orden+1; i++)
                valorPolinomio += a[i]*pow(x,(double)i); // pow() es función de la biblioteca estándar
        return valorPolinomio;
}
```

5.15.8 Anagrama

Ingrese una palabra por teclado que tenga entre 5 y 15 caracteres del alfabeto inglés. Después ingrese tres palabras separadas por espacio, donde cada una de las tres palabras, deben tener exactamente el mismo número de caracteres de la primera palabra. Su programa debe detectar si alguna de las tres palabras ingresadas no cumple con el número de caracteres exigidos y entonces, volver a pedir un nuevo ingreso de tres palabras.

Escriba una función …. anagrama(…) que diga si alguna de las tres últimas palabras ingresadas es un anagrama de la primera palabra ingresada por teclado. Imprima todas las palabras que son anagrama de la primera.

Un anagrama es una permutación de los caracteres de una palabra para construir otra.

Ejemplo: **agranda – granada**. Se tiene un anagrama si dos palabras, tienen los mismos caracteres ordenados de distinta forma.

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

// determine que hace cada una de las cuatro funciones declaradas a continuación
_Bool checkPalabras(char*);
_Bool countLetras(char*,int*);
_Bool igualCantidadLetras(char*,int);
void  anagrama(char*,char*,int);

int main(int argc, char *argv[])
{
   int count = 0;
        char palabraM[50];          // reserva espacio para primera palabra.
        char tresPalabras[150];     // reserva espacio para tres palabras
        do{
        printf("Ingrese primera palabra entre 5 y 15 caracteres= ");  // ingresar primera palabra
        scanf(" %s", palabraM);}while(!countLetras(palabraM, &count));
        printf("%s\n",palabraM);
        printf("Primera palabra=%d\n",count);
```

```
        // si una de las tres palabras a ingresar no tiene el mismo número de caracteres de la
        // primera palabra ingresada, se obliga a ingresar de nuevo tres palabras.
        do{
        printf("Ingrese tres palabras separadas por espacio= ");      // ingresar tres palabras
        scanf(" %[^\n]", tresPalabras);}while(!igualCantidadLetras(tresPalabras,count));
        printf("%s\n",tresPalabras);

        anagrama(palabraM,tresPalabras,count);

   return 0;
}



_Bool countLetras(char* x, int* y)
{
        int count = 0, i = 0;
        while(x[i] != '\0')
        {
                count++;
                i++;
        }
        *y = count;

        if(count < 15 && count > 4)
        {
                return true;
        }else{ return false;}

}

_Bool igualCantidadLetras(char* x,int y)
{
        int n=0, espacio=0;
        while(x[n] != '\0')
        {
                if(x[n] == ' ')
                {
                        espacio++;
                }
                n++;
        }
        printf("espacio=%d\n",espacio);
        if(espacio != 2)
                return false;

        int count = 0, i = 0;
        char a = ' ';
```

```
	for(int k=0; k < 3;k++)
	{
		if(k == 2)
			a = '\0';
		while(x[i] != a)
		{
			count++;
			i++;
		}
		printf("palabra%d= %d\n",k+1,count);
		i++;
		if(count != y)
			return false;
		count=0;
	}
		return true;
}

void  anagrama(char* x,char* y,int count)
{
	for(int j=0;j < 3;j++)
	{
		char palabraM[20];
		int par = 0;
		for(int i=0; i < count;i++)
			palabraM[i] = x[i];
		for(int k=0; k < count; k++)
		{
			for(int i=0; i < count; i++)
			{
				if(palabraM[i] == y[k+j*(count+1)])
				{
					palabraM[i] = ' ';
					par++;
					break;
				}
			}
		}
		if(count == par)
		{
			printf("Es Anagrama: ");
			for(int i = 0;i < count;i++)
				printf("%c",y[i+j*(count+1)]);
			printf("\n");
		}else
		{
			printf("No es Anagrama: ");
			for(int i = 0;i < count;i++)
```

```
                        printf("%c",y[i+j*(count+1)]);
                    printf("\n");
                }
        }

}
```

5.15.9 Arreglos dinámicos de dos dimensiones.

Ingresar por teclado un número entero que permita reservar memoria para almacenar un número determinado de enteros, definido por el usuario.
Ingrese por teclado dos números enteros, que permitan crear un arreglo de dos dimensiones de la forma int arreglo[][], siempre que se cumpla que el producto de ambas dimensiones sea menor o igual al total de enteros reservados en memoria.

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

int main(int argc, char *argv[])
{
        printf(" ***Este programa crea un arreglo dinamico de dos dimensiones*** \n");
        int     mem = 0;      // tamaño de memoria reservada
        printf(" Ingrese tama%co de memoria = " ,164); // 164 es código de letra ñ
        scanf(" %d", &mem);

        int*     beta = malloc(sizeof(int)*mem);
        for(int i = 0; i  < mem; i++)
        {
                beta[i] = i+1;
                printf("%d", beta[i]);
                printf("%c",' ');
        }
        int n1 = 1, n2 = 1;
        printf("\n");
        printf(" **Producto de ambas dimensiones debe ser menor o igual a tama%co de memoria**
        \n",164);
        do{
        printf("Ingrese tama%co de primera dimension= ",164); // 164 es para imprimir letra ñ
        scanf(" %d", &n1);
        printf("Ingrese tama%co de segunda dimension= ",164);
        scanf(" %d", &n2);
                }while(n1*n2  != mem || n1*n2  >  mem);
        //printf("%d  %d\n", n1, n2);

        int* arr2[n1];
        for(int i = 0; i < n1; i++)
        {
                arr2[i] = beta + i*n2;
        }
```

```
        int k,m;
        printf("***Ingrese el  valor que desea imprimir***\n");
        printf("Ingrese subíndice de primera dimension= ");
        scanf(" %d", &k);
        printf("Ingrese subíndice segunda dimension= ");
        scanf(" %d", &m);
        printf("valor = %d\n",arr2[k][m]);

         free(beta);
         return 0;
        }
```

5.15.10 Arreglos dinámicos de tres dimensiones.

Ingresar por teclado un número entero que permita reservar memoria para almacenar un número determinado de enteros, definido por el usuario.
Ingrese por teclado tres números enteros, que permitan crear un arreglo de tres dimensiones de la forma int arreglo[][][], siempre que se cumpla que el producto de las tres dimensiones sea menor o igual al total de enteros reservados en memoria.

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

int main(int argc, char *argv[])
{
        printf("***Este programa crea un arreglo dinamico de tres dimensiones***\n");
        int        mem = 0;
        printf("Ingrese tama%co de memoria = ",164); // 164 permite imprimir letra ñ
        scanf(" %d", &mem);

        int*       beta = malloc(sizeof(int)*mem);
        for(int i = 0; i  < mem; i++)
        {
                beta[i] = i+1;
                printf("%d", beta[i]);
                printf("%c",' ');
        }
        int n1 = 1, n2 = 1, n3 = 1;
        printf("\n");
        printf("**Producto de las tres dimesiones debe ser menor o igual a tama%co de
memoria**\n",164);
        do{
        printf("Ingrese tama%co de primera dimension= ",164);
        scanf(" %d", &n1);
```

```
        printf("Ingrese tama%co de segunda dimension= ",164);
        scanf(" %d", &n2);
        printf("Ingrese tama%co de tercera dimension= ",164);
        scanf(" %d", &n3);
                }while(n1*n2*n3  != mem || n1*n2*n3  >  mem);

        int**     p1[n1];  // crea un doble puntero necesario para tener 3 dimensiones
        int m = 0;
        for(int i = 0; i < n1; i++)  // explique Ud. que hacen estos for anidados
        {
                p1[i] = &beta + i*n2*n3;
                for(int k = 0; k < n2; k++)
                {
                        p1[i][k] = beta + m;
                        m = m + n3;
                }
        }

        int k,l,z;
        printf("***Ingrese el valor que desea imprimir***\n");
        printf("Ingrese primer subindice <=  a  %d\n", n1);
        scanf(" %d", &k);
        printf("Ingrese segundo subindice <=  a %d\n", n2);
        scanf(" %d", &l);
        printf("Ingrese tercer subindice <=  a  %d\n", n3);
        scanf(" %d", &z);
        printf("valor = %d\n",p1[k][l][z]);  // imprime un valor usando tres subíndices [][][]
        free(beta);
   return 0;
}
```

Modifique los programas 5.15.9 y 5.15.10, de tal manera que sea una función la encargada de crear el arreglo de dos y tres dimensiones.

5.15.11 Crear un Tipo de un arreglo.

En este ejemplo se muestra porqué se debe usar ** como parámetro de una función cuando se usa un arreglo de arreglos.

```
#include <stdio.h>

typedef int arr[5];          // Ahora arr es un Tipo de un arreglo de cinco elementos enteros.
int func(int**);

int main(int argc, char *argv[])
{
        arr gama1 = {2,3,4,5,7};
        arr gama2 = {23,56,78,90,45};
        arr gama3 = {3,56,8,90,5};

        // beta es la dirección del primer elemento del arreglo, esto es gama1, pero gama1 a su vez es la
        //dirección del primer elemento del arreglo arr, esto es el número 2.
        // Esto implica que para alcanzar a cualquier número de gama1, gama2 o gama3, es necesario
        usar ** en la función func.

        int*     beta[3] = {gama1, gama2, gama3}; //gama1 es el nombre de un arreglo del Tipo arr y por
        //lo tanto, es también una dirección. Esto obliga a que el Tipo de los elementos de beta sea int*.

        printf("%d\n", beta[0][1]);      // imprime el número 3
        printf("%d\n", func(beta));      // imprime el número 78

         return 0;
        }

        int      func(int** x)
        {
                 return x[1][2];     // [1] selecciona gama2 y [2] selecciona el número 78 de gama2
        }
```