UNIVERSIDAD DE GUANAJUATO

Estructuras de datos fundamentales

Dr. Mario Alberto Ibarra Manzano
Departamento de Ingeniería Electrónica
División de Ingeniería del Campus Irapuato-Salamanca

Estructuras de datos fundamentales

- Estructuras de datos básicas

❖ Arreglos
❖ Cadenas
❖ Estructuras
❖ Uniones
❖ Punteros
❖ Memoria estática
❖ Memoria dinámica

- Estructuras de datos abstractas

❖ Pilas
❖ Colas
❖ Listas enlazadas simples y dobles
❖ Listas enlazadas múltiples

Definición de estructuras de datos fundamentales

En el lenguaje de programación C, las estructuras de datos fundamentales son los bloques básicos que se utilizan para almacenar y organizar datos en la memoria de una manera eficiente. Estas estructuras permiten manejar y manipular datos de manera sistemática, proporcionando una base para construir estructuras más complejas y algoritmos eficientes.

- Arreglos (Arrays)
- Cadenas (Strings)
- Estructuras (Struct)
- Uniones (Unions)
- Enumeraciones (Enums)
- Punteros (Pointers)

➢ Gestión de almacenamiento de datos
- Memoria estática
- Memoria dinámica

Definición de estructuras de datos fundamentales

En el contexto de un curso de Ingeniería de Datos e Inteligencia Artificial, las estructuras de datos fundamentales son los bloques básicos de almacenamiento y organización de datos que permiten la eficiente manipulación, acceso y procesamiento de grandes volúmenes de información. Estas estructuras son esenciales para el manejo de datos en aplicaciones de inteligencia artificial, donde el rendimiento y la eficiencia son cruciales.

Almacenamiento Eficiente

- Permiten guardar datos de manera óptima en términos de uso de memoria y tiempo de acceso. Esto es especialmente importante en grandes volúmenes de datos, donde la eficiencia puede afectar significativamente el rendimiento de los algoritmos.

Manipulación de Datos

- Facilitan operaciones como la inserción, eliminación, búsqueda y actualización de datos, que son operaciones comunes en la preparación y análisis de datos.

Optimización de Algoritmos

- Los algoritmos de inteligencia artificial, como los de aprendizaje automático, se basan en estas estructuras para manejar datos de entrada y salida, optimizar el entrenamiento de modelos y mejorar la velocidad de procesamiento.

Escalabilidad

- Ayudan a diseñar sistemas que pueden escalar con el crecimiento de datos, asegurando que las operaciones sigan siendo rápidas y eficientes a medida que el volumen de datos aumenta.

Definición de estructuras de datos fundamentales

Ejemplos de Estructuras de Datos en IA e Ingeniería de Datos

Arreglos y Listas: Para almacenar secuencias de datos, como conjuntos de características en un modelo de aprendizaje automático.

Matrices y Tensores: Utilizados extensamente en operaciones de álgebra lineal y redes neuronales, donde los datos se representan en forma de matrices de múltiples dimensiones.

Tablas Hash y Diccionarios: Permiten búsquedas rápidas y son útiles para gestionar conjuntos de datos etiquetados o realizar búsquedas en grandes bases de datos.

Árboles y Grafos: Cruciales para representar relaciones jerárquicas y de red, como en los modelos de decisiones y grafos de conocimiento en inteligencia artificial.

Colas y Pilas: Utilizadas en algoritmos de búsqueda, procesamiento de tareas y gestión de datos en memoria.

Arreglos (Array)

En el lenguaje de programación C, un arreglo (o array en inglés) es una estructura de datos que consiste en una colección de elementos del mismo tipo de datos, almacenados en ubicaciones de memoria contiguas. Los arreglos permiten almacenar múltiples valores en una sola variable y acceder a ellos mediante un índice, lo que facilita la manipulación y el procesamiento de datos en bloque.

Características

- **Tipo Homogéneo:** Todos los elementos de un arreglo deben ser del mismo tipo, como enteros (‘int’), caracteres (‘char’), flotantes (‘float’), etc.
- **Tamaño Fijo:** El tamaño de un arreglo en C se debe especificar en el momento de su declaración y no puede cambiar durante la ejecución del programa. Esto significa que la cantidad de memoria asignada para el arreglo es constante.
- **Acceso mediante Índices:** Cada elemento del arreglo se puede acceder utilizando un índice, que comienza en 0 para el primer elemento y se incrementa por 1 para cada elemento sucesivo. Esto permite un acceso rápido a cualquier elemento del arreglo.
- **Memoria Contigua:** Los elementos de un arreglo se almacenan en ubicaciones de memoria contiguas. Esto permite un acceso eficiente a los elementos y optimiza el rendimiento del programa.

Arreglos (Array)

Aplicaciones de los Arreglos

- **Almacenamiento de Datos:** Se utilizan para almacenar colecciones de datos, como listas de números, caracteres o cualquier otro tipo de datos homogéneos.
- **Procesamiento de Datos en Bloque:** Facilitan la manipulación de datos en bloque, como realizar operaciones matemáticas en series de números, ordenar listas, buscar elementos, etc.
- **Matrices y Vectores:** Los arreglos son la base para la implementación de estructuras más complejas como matrices (arreglos bidimensionales) y vectores en álgebra lineal, fundamentales en aplicaciones de ingeniería de datos e inteligencia artificial.

Conceptos Importantes sobre Arreglos

1. **Definición de Arreglo:** Un arreglo es una colección de elementos del mismo tipo de datos, almacenados en ubicaciones de memoria contiguas. Se accede a los elementos utilizando índices.
2. **Tipos de Arreglos:** Unidimensionales y Multidimensionales.
3. **Índices y Acceso:** Los índices en un arreglo comienzan en 0, lo que significa que el primer elemento está en el índice 0, el segundo en el índice 1, y así sucesivamente. El acceso a un elemento se realiza utilizando su índice, por ejemplo, array[2] accede al tercer elemento del arreglo.
4. **Tamaño de Arreglo:** El tamaño de un arreglo debe ser conocido en tiempo de compilación en C y no puede cambiar durante la ejecución del programa. Esto significa que los arreglos tienen un tamaño fijo.
5. **Inicialización de Arreglos:** Los arreglos se pueden inicializar en el momento de la declaración utilizando llaves {} con una lista de valores. Si no se especifican valores para todos los elementos, los restantes se inicializan a 0. En un indice especifico se utiliza corchetes [], por ejemplo, float A[5] = {5, [2] = 4, [1] = 6, 7}.
6. **Memoria Contigua:** Los elementos de un arreglo se almacenan en ubicaciones de memoria contiguas, lo que permite un acceso eficiente y rápido a los elementos.

Arreglos (Array)

Generación de números aleatorios

La función *rand()* en C genera números pseudoaleatorios que siguen una distribución uniforme discreta en el rango de valores posibles que puede producir. Esto significa que, en teoría, cada número entero dentro del rango posible tiene la misma probabilidad de ser seleccionado.

1. **Distribución Uniforme Discreta:** Los números generados por *rand()* están distribuidos uniformemente, lo que significa que cada número entero en el rango de salida tiene la misma probabilidad de aparecer. Sin embargo, es importante tener en cuenta que *rand()* genera números pseudoaleatorios, lo que significa que siguen un patrón determinado por una fórmula matemática, y no son verdaderamente aleatorios.
2. **Rango de Salida:** El rango de salida de *rand()* depende de la implementación de la biblioteca estándar C en el sistema. Normalmente, rand() devuelve un número entero entre 0 y *RAND_MAX*, donde *RAND_MAX* es una constante definida en <*stdlib.h*> que representa el valor máximo posible devuelto por *rand()*. El valor de *RAND_MAX* puede variar entre implementaciones, pero es al menos 32,767 según el estándar C.
3. **Semilla (seed):** *rand()* genera números pseudoaleatorios basándose en una semilla inicial. Si no se establece una semilla usando la función *srand()*, *rand()* generará la misma secuencia de números cada vez que se ejecute el programa, lo que puede ser útil para pruebas repetibles. Para obtener una secuencia diferente de números aleatorios cada vez que se ejecuta el programa, se puede usar srand() con un valor variable, como el tiempo actual:

Arreglos (Array)

Distribución Uniforme Discreta (DU)

$$x = rand() \qquad x = \{a, a+1, a+2, \dots, b\} = [0, RAND\_MAX]$$

$$P(x) = \frac{1}{b-a+1}$$

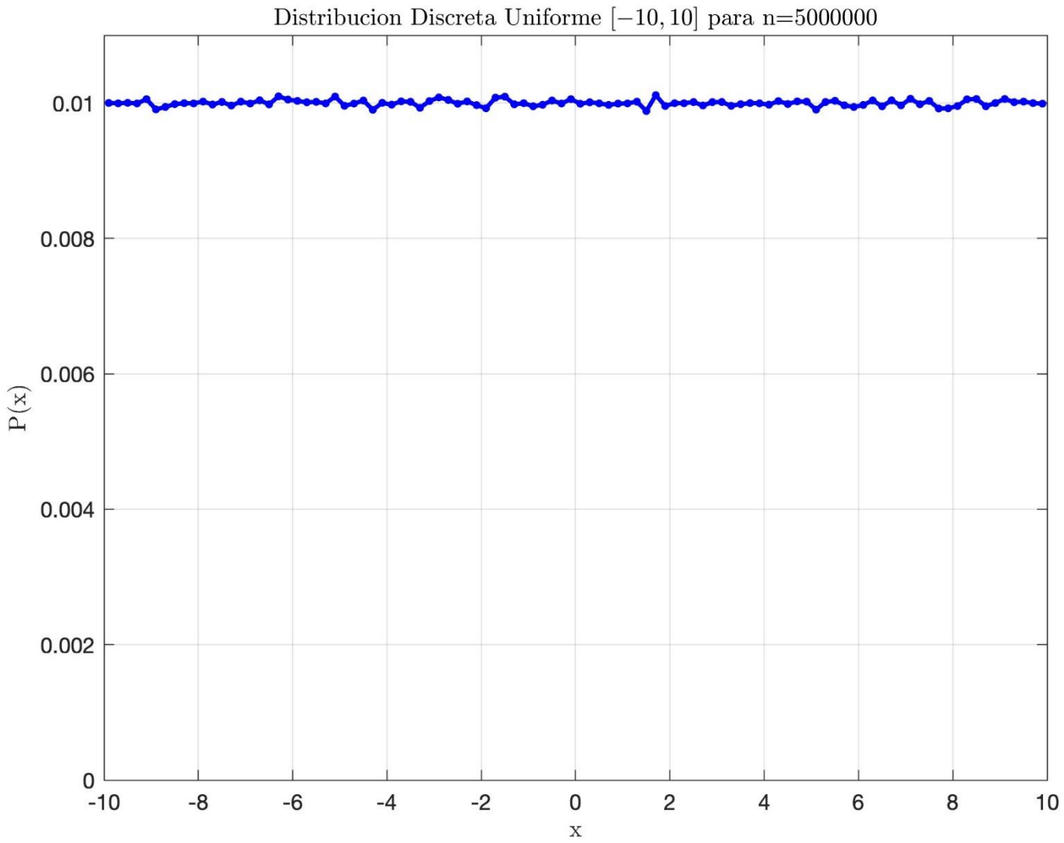
$$\mu = \frac{a+b}{2} \qquad \bar{x} = \frac{\sum x_i}{n}$$

$$\sigma^2 = \frac{(b-a+1)^2 - 1}{12} \qquad s^2 = \frac{\sum (x-\mu)^2}{n} = \frac{\sum x^2}{n} - \mu^2$$

$$\hat{a} = \left\lfloor \bar{x} + {}^1\!/_2 - \frac{\sqrt{12s^2+1}}{2} \right\rfloor \qquad \hat{a} = \min(x)$$

$$\hat{b} = \left\lceil \bar{x} - {}^1\!/_2 + \frac{\sqrt{12s^2+1}}{2} \right\rceil \qquad \hat{b} = \max(x)$$

$$P(w \le x) = \frac{x-a+1}{b-a+1} \qquad N = b-a+1$$



Arreglos (Array)

Distribución Uniforme Discreta (DU)

Estimación de Inventario en un Entorno Industrial

En una fábrica de componentes electrónicos, se producen diariamente piezas de un componente crítico para un cliente importante. Cada componente producido tiene un número de serie único que se asigna secuencialmente a medida que se fabrica.
Recientemente, un error en el sistema de gestión de inventarios ha causado la pérdida de datos sobre cuántos componentes en total se han producido en los últimos tres meses. Afortunadamente, un lote de n componentes ha sido inspeccionado manualmente, y se han registrado los números de serie de estos componentes. Los números de serie capturados están listados como (x_1, x_2 ,…,x_n).

Estimación de la Población en un Evento Comunitario

En una comunidad local, se ha organizado un evento social al aire libre en un parque. Cada asistente recibe un boleto con un número de serie único al ingresar. Desafortunadamente, debido a un problema técnico, el contador digital que registra el número total de asistentes dejó de funcionar a mitad del evento, y los organizadores no pueden determinar exactamente cuántas personas asistieron.
Para estimar la cantidad total de asistentes, un grupo de voluntarios ha recogido de manera aleatoria n boletos desechados que encontraron en el parque al final del evento. Los números de serie de estos boletos recogidos están registrados como (x_1, x_2, … ,x_n).

Arreglos (Array)

Distribución Uniforme Discreta (DU)

Estimación de Inventario en un Entorno Industrial

En una fábrica de componentes electrónicos, se producen diariamente piezas de un componente crítico para un cliente importante. Cada componente producido tiene un número de serie único que se asigna secuencialmente a medida que se fabrica.
Recientemente, un error en el sistema de gestión de inventarios ha causado la pérdida de datos sobre cuántos componentes en total se han producido en los últimos tres meses. Afortunadamente, un lote de n componentes ha sido inspeccionado manualmente, y se han registrado los números de serie de estos componentes. Los números de serie capturados están listados como (x_1, x_2 ,…,x_n).

$$x = (x_1, x_2, \dots, x_n)$$

$$\bar{x} = \frac{\sum x_i}{n}$$

$$s^2 = \frac{\sum (x-\mu)^2}{n} = \frac{\sum x^2}{n} - \mu^2$$

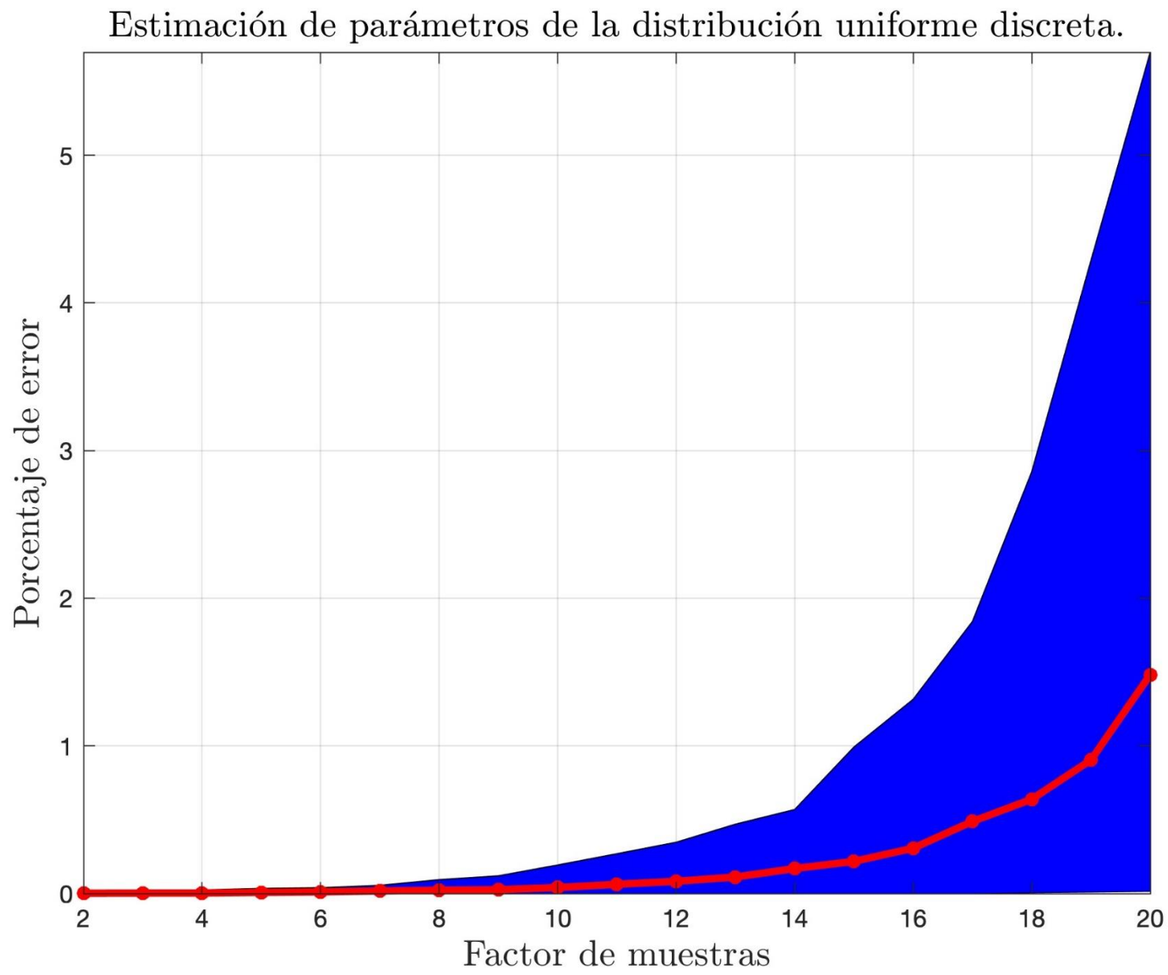
$$\hat{a} = \left[\bar{x} + {}^1\!/_2 - \frac{\sqrt{12s^2+1}}{2} \right]$$

$$\hat{b} = \left[\bar{x} - {}^1\!/_2 + \frac{\sqrt{12s^2+1}}{2} \right]$$

$$\hat{N} = \hat{b} - \hat{a} + 1$$

Arreglos (Array)

Distribución Uniforme Discreta



Arreglos (Array)

Distribuciones discretas.

Rango admisible: Las distribuciones discretas tienen una variable aleatoria, x, con un rango admisible:

$$a \leq x \leq b$$

Función de probabilidad: La función de probabilidad de x, P(x), es:

$$P(x) \geq 0 \qquad\qquad a \leq\leq x \leq b$$

donde:

$$\sum_{a}^{b} P(x) = 1.0$$

Probabilidad acumulada: La función de probabilidad acumulada de x, F(x), es:

$$F(x) = \sum_{a}^{x} P(w)$$

Esto nos da la probabilidad acumulada hasta $x=x_0$ o menor, como:

$$F(x_0) = P(x \leq x_0) = \sum_{a}^{x_0} P(x)$$

Arreglos (Array)

Distribuciones discretas.

Probabilidad complementaria: La probabilidad complementaria de x_0 es la probabilidad de x superior que x_0 como:

$$H(x_0) = 1 - F(x_0) = P(x > x_0)$$

Valor esperado o media: El valor esperado de x, E(x), también conocido como media de x, μ, se define como:

$$\mu = E(x) = \sum_{a}^{b} xP(x)$$

Varianza y desviación estándar: La varianza de x, σ^2, es obtenida de la siguiente forma:

$$\sigma^2 = E[(x-\mu)^2] = \sum_{a}^{b} (x-\mu)^2 P(x)$$

La desviación estándar, σ, es calculada de la siguiente forma:

$$\sigma = \sqrt{\sigma^2}$$

Arreglos (Array)

Distribuciones discretas.

Mediana: La mediana en x, denotada por $\mu_{0.5}$, es el valor de x con probabilidad acumulada de 0.5 como se muestra:

$$F(\mu_{0.5}) = 0.5$$

Moda: La moda, $\tilde{\mu}$, es el valor que más se repite de x y está localizado donde la probabilidad es mayor en el rango admisible, como sigue:

$$P(\tilde{\mu}) = max\{P(x) \quad a \leq x \leq b\}$$

Índice léxico (Lexis Ratio): El índice léxico, τ de x es la razón de la varianza sobre la media, se define como:

$$\tau = \frac{\sigma^2}{\mu}$$

Estadísticas básicas sobre datos muestras.

Considerando n datos muestra, $(x_1, \dots, x_n)$, recolectados, varias medidas estadísticas pueden calcularse:

$$n = \#(x_1, \dots, x_n)$$
$$x(1) = \min(x_1, \dots, x_n)$$
$$x(n) = \max(x_1, \dots, x_n)$$

Arreglos (Array)

Estadísticas básicas sobre datos muestras.

$$\bar{x} = \sum_{i=1}^{n} \frac{x_i}{n}$$

$$s^2 = \sum_{i=1}^{n} \frac{(x_i - \bar{x})^2}{n-1}$$

$$s = \sqrt{s^2}$$

$$cov = \frac{s}{\bar{x}}$$

$$\tilde{x} = \max(F(x))$$

$$x_{0.5} = \begin{cases} x(k) & k = \dfrac{n+1}{2} \\ \dfrac{x(k) + x(k+1)}{2} & k = \dfrac{n}{2} \end{cases}$$

$$\tau = \frac{s^2}{\bar{x}}$$

Arreglos (Array)

Generación de Tallas para una Empresa de Ropa.

Una empresa de ropa desea estimar cuántos productos de cada talla deben fabricar en función de la distribución de las alturas de los consumidores. Las alturas de las personas muestran una media (μ) de 170 cm y una desviación estándar (σ) de 10 cm.

Requisitos:

La empresa divide las tallas en cinco categorías, y cada categoría cubre un rango específico de alturas:

- Talla XS (Extra Small): Alturas menores a 155 cm.
- Talla S (Small): Alturas entre 155 cm y 165 cm.
- Talla M (Medium): Alturas entre 165 cm y 175 cm.
- Talla L (Large): Alturas entre 175 cm y 185 cm.
- Talla XL (Extra Large): Alturas mayores a 185 cm.

Simular un número determinado de consumidores (por ejemplo, 2000) y estimar cuántos productos de cada talla deben fabricarse, en función de la altura simulada de los consumidores.

Arreglos (Array)

Distribución Normal (Gaussiana).

El método de Box-Müller se define por las siguientes fórmulas:

$$Z_0 = \sqrt{-2lnU_1}cos(2\pi U_2)$$
$$Z_1 = \sqrt{-2lnU_1}sin(2\pi U_2)$$

donde U_1 y U_2 son dos números aleatorios independientes uniformemente distribuidos en el rango de $(0,1)$. Los valores resultantes Z_0 y Z_1 son números aleatorios con una distribución normal estándar (media = 0, desviación estándar 1).

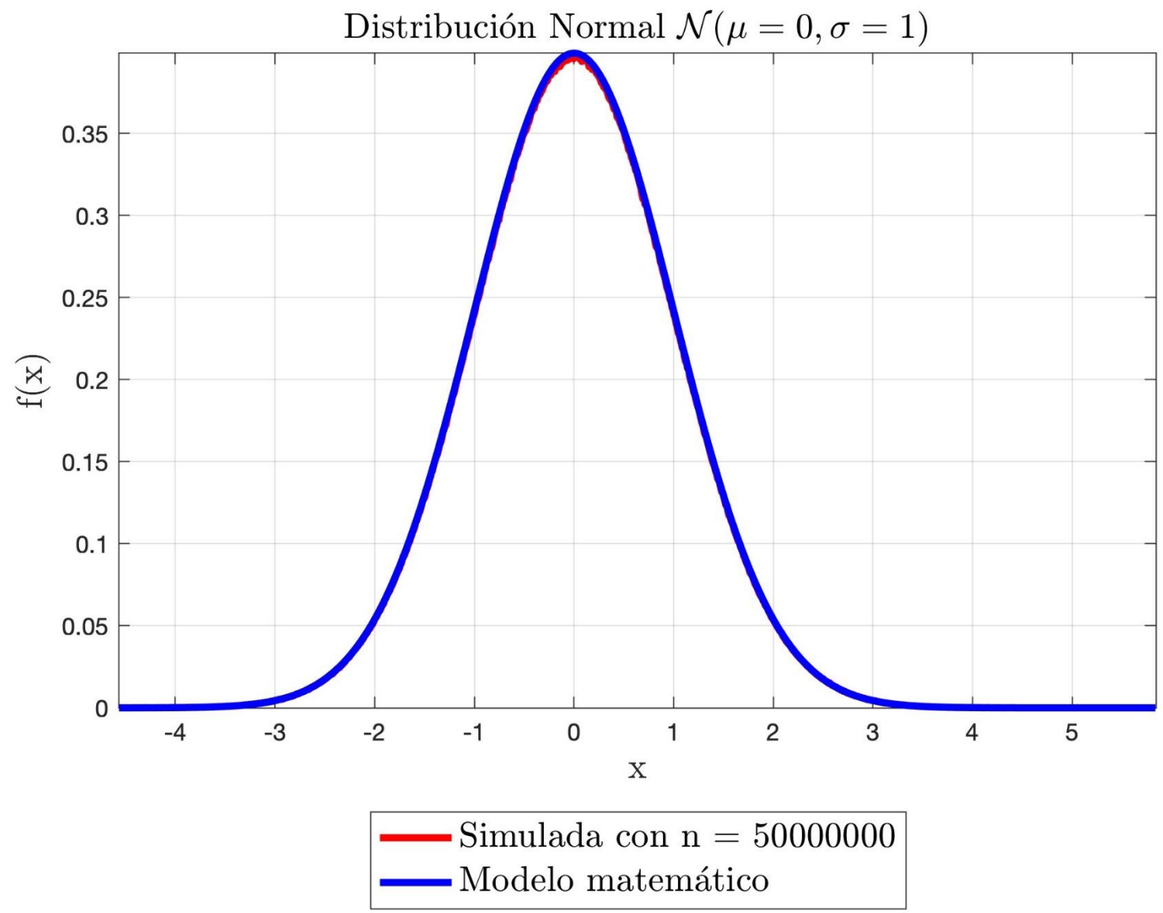
El método de Box-Müller genera números con una distribución normal estándar ($\mu = 0$, $\sigma = 1$). Para generar números con una media (μ) y desviación estándar (σ) específicas, puedes ajustar el resultado de la siguiente manera:

$$X = \mu + \sigma Z$$

Donde Z es el número con distribución normal estándar generado.

Arreglos (Array)

Distribución Normal (Gaussiana).



Arreglos (Array)

Generación de Tallas para una Empresa de Ropa.

Una empresa de ropa desea estimar cuántos productos de cada talla deben fabricar en función de la distribución de las alturas de los consumidores. Las alturas de las personas muestran una media (μ) de 170 cm y una desviación estándar (σ) de 10 cm.

Requisitos:

La empresa divide las tallas en cinco categorías, y cada categoría cubre un rango específico de alturas:

- Talla XS (Extra Small): Alturas menores a 155 cm.
- Talla S (Small): Alturas entre 155 cm y 165 cm.
- Talla M (Medium): Alturas entre 165 cm y 175 cm.
- Talla L (Large): Alturas entre 175 cm y 185 cm.
- Talla XL (Extra Large): Alturas mayores a 185 cm.

Simular un número determinado de consumidores (por ejemplo, 2000) y estimar cuántos productos de cada talla deben fabricarse, en función de la altura simulada de los consumidores.

Arreglos (Array)

Generación de Tallas para una Empresa de Ropa.

```
m = 170;
S = 10;

// Distribución uniforme discreta
U1 = (float)rand()/RAND_MAX;
U2 = (float)rand()/RAND_MAX;

// Distribución normal normalizada
# define PI 3.1415926535897932
Z = sqrt(-2*log(U1))*cos(2*PI*U2)

// Distribución normal con m y s
X = m+s*Z;

//Indice
M[4] = {155, 165, 175, 185};
XS, Sí X<M[0] entonces id=0
S, Sí X<M[1] entonces id=1
M, Sí X<M[2] entonces id=2
L, Sí X<M[3] entonces id=3
XL, Sí X>M[3] entonces id=4
```

Índice	Talla	Mínimo	Máximo
0	XS	-	155
1	S	155	165
2	M	165	175
3	L	175	185
4	XL	185	-

Arreglos (Array)

Generación de Tallas para una Empresa de Ropa.

XS, Sí c_3=X<M[0] entonces id=0
S, Sí c_2= X<M[1] entonces id=1
M, Sí c_1= X<M[2] entonces id=2
L, Sí c_0= X<M[3] entonces id=3
XL, Sí c'_0=X>M[3] entonces id=4

C	$c_3\ c_2\ c_1\ c_0$	id	$b_3\ b_2\ b_1\ b_0$
15	1 1 1 1	0	0 0 0 0
7	0 1 1 1	1	0 0 0 1
3	0 0 1 1	2	0 0 1 0
1	0 0 0 1	3	0 0 1 1
0	0 0 0 0	4	0 1 0 0

b_0

$c_3c_2\backslash c_1c_0$	00	01	11	10
00	0	1	0	d
01	d	d	1	d
11	d	d	0	d
10	d	d	d	d

b_1

$c_3c_2\backslash c_1c_0$	00	01	11	10
00	0	0	1	d
01	d	d	1	d
11	d	d	0	d
10	d	d	d	d

b_2

$c_3c_2\backslash c_1c_0$	00	01	11	10
00	1	0	0	d
01	d	d	0	d
11	d	d	0	d
10	d	d	d	d

$b_0=c'_3c_2+c'_1c_0$ $\quad$ $b_1=c'_3c_1$ $\quad$ $b_2=c'_0$

id = (b2<<2)|(b1<<1)|(b0) = ((X>=M[3])<<2)|(((X>=M[0])&(X<M[2]))<<1)|(((X>=M[0])&(X<M[1]))|((X>=M[2])&(X<M[3])));

Arreglos (Array)

Generación de Tallas para una Empresa de Ropa.

XS, Sí c_3=X<M[0] entonces id=0
S, Sí c_2= X<M[1] entonces id=1
M, Sí c_1= X<M[2] entonces id=2
L, Sí c_0= X<M[3] entonces id=3
XL, Sí c'_0=X>M[3] entonces id=4

Id=y	$c_3\ c_2\ c_1\ c_0$	C=x	x+1	2^y	$2^{(4-y)}$
0	1 1 1 1	15	16	1	16
1	0 1 1 1	7	8	2	8
2	0 0 1 1	3	4	4	4
3	0 0 0 1	1	2	8	2
4	0 0 0 0	0	1	16	1

$$2^{4-y} = x + 1$$

$$log_2(2^{4-y}) = log_2(x+1)$$

$$(4-y)log_2(2) = log_2(x+1)$$

$$y = 4 - log_2(x+1)$$

$$y = 4log_2(2) - log_2(x+1)$$

$$y = log_2\left(\frac{2^4}{x+1}\right) = log_2\left(\frac{16}{x+1}\right)$$

$$y = \frac{ln\left(\frac{16}{x+1}\right)}{ln(2)}$$

$$y = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

$$4 = a_0$$

$$3 = a_4 + a_3 + a_2 + a_1 + 4$$

$$2 = 81a_4 + 27a_3 + 9a_2 + 3a_1 + 4$$

$$1 = 2401a_4 + 343a_3 + 49a_2 + 7a_1 + 4$$

$$0 = 50625a_4 + 3375a_3 + 225a_2 + 15a_1 + 4$$

$$a_0 = 4 \qquad a_1 = -\frac{1693}{1363} \qquad a_2 = \frac{1817}{6720} \qquad a_3 = -\frac{197}{6720} \qquad a_4 = \frac{1}{960}$$

$$y = \frac{1}{960}x^4 - \frac{197}{6720}x^3 + \frac{1817}{6720}x^2 - \frac{1693}{1363}x + 4$$

$$y = \left(\left(\left(\frac{1}{960}x - \frac{197}{6720}\right)x + \frac{1817}{6720}\right)x - \frac{1693}{1363}\right)x + 4$$

Arreglos (Array)

Análisis de Tiempos de Procesos en una Fábrica.

Una fábrica tiene un proceso de producción de piezas que sigue una distribución normal. El tiempo que una máquina tarda en producir una pieza depende de varios factores y se distribuye normalmente con una media de 30 minutos y una desviación estándar de 5 minutos.

La fábrica quiere analizar los tiempos de producción para 1000 piezas y, en particular, quiere saber cuántas piezas se producen en menos de 25 minutos (procesos rápidos), cuántas se producen entre 25 y 35 minutos (procesos normales), y cuántas tardan más de 35 minutos (procesos lentos). Este análisis ayudará a mejorar la planificación del proceso y a identificar oportunidades de mejora.

Prueba de Calidad en una Línea de Producción

Una fábrica de electrónicos realiza pruebas de calidad en sus productos para verificar si cumplen con los estándares de calidad. De cada lote de producción de 1,000 productos, se seleccionan 10 productos al azar y se verifica si son defectuosos. Según la experiencia, la probabilidad de que un producto sea defectuoso es del 5% (probabilidad de éxito p=0.05).

El objetivo es determinar cuántos productos defectuosos se pueden esperar en una muestra de 10 productos, utilizando una distribución binomial, y simular esta prueba de calidad en 100 lotes de producción. Queremos analizar cuántos productos defectuosos encontramos en cada lote y cómo se distribuyen esos resultados.

Arreglos (Array)

Distribución Binomial.

Una distribución binomial describe el número de éxitos en una serie de ensayos de Bernoulli independientes (donde cada ensayo tiene dos resultados posibles, típicamente "éxito" o "fracaso") con una probabilidad constante de éxito p.

La probabilidad de obtener exactamente k éxitos en n ensayos se calcula usando la fórmula de la función de probabilidad binomial:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

donde:
n = número de ensayos.
k = número de éxitos.
p = probabilidad de éxito en un ensayo individual.
$\binom{n}{k}$ = coeficiente binomial, que representa el número de maneras de elegir k éxitos de n ensayos.

Para una distribución binomial con parámetros n y p las medidas estadísticas se definen como:

$$\mu = np$$
$$\sigma^2 = np(1-p)$$
$$CV = \frac{\sqrt{1-p}}{\sqrt{np}}$$

Cadenas (String)

En el lenguaje de programación C, una cadena (o string en inglés) es una secuencia de caracteres terminada con un carácter nulo (“\0”). Las cadenas en C se representan como arreglos de caracteres y se utilizan para almacenar y manipular texto.

Características

- **Arreglo de Caracteres:** Una cadena es esencialmente un arreglo de caracteres, donde cada carácter es un elemento del arreglo. Por ejemplo, la cadena "Hola" se almacena como un arreglo de caracteres {'H', 'o', 'l', 'a', '\0'}.
- **Carácter Nulo (“\0”):** Todas las cadenas en C están terminadas con un carácter nulo (“\0”), que indica el final de la cadena. Esto permite que las funciones de manejo de cadenas sepan dónde termina la cadena, ya que no existe una longitud intrínseca almacenada en la cadena misma.
- **Manipulación de Cadenas:** Las cadenas en C se pueden manipular utilizando una variedad de funciones estándar definidas en la biblioteca <string.h>. Estas funciones permiten operaciones como copiar cadenas, concatenarlas, medir su longitud, compararlas, buscar caracteres, y más.

Cadenas (String)

Asignación de cadenas

En C, no puedes asignar directamente una cadena a otra usando el operador = como lo harías en otros lenguajes. Debes usar la función *strcpy()* de la biblioteca *string.h*.

```
char cadena1[20];
char cadena2[] = "Hola Mundo";
strcpy(cadena1, cadena2);
```

Comparación de cadenas.

Para comparar cadenas en C, se utiliza la función *strcmp()* de la biblioteca *string.h*. El operador == no se puede utilizar directamente, ya que compararía las direcciones de memoria en lugar del contenido.

```
char cadena1[] = "Hola";
char cadena2[] = "Hola";
cmp = strcmp(cadena1, cadena2);
```

strcmp(cadena1, cadena2) devuelve:

- 0 si las cadenas son iguales.
- Un valor negativo si *cadena1* es menor que cadena2 (orden lexicográfico).
- Un valor positivo si *cadena1* es mayor que cadena2.

Cadenas (String)

Concatenación de Cadenas

Para concatenar cadenas en C, se utiliza la función *strcat()*.

```
char cadena1[50] = "Hola, ";
char cadena2[] = "Mundo!";
strcat(cadena1, cadena2);
```

Longitud de Cadenas.

Para obtener la longitud de una cadena (sin contar el carácter nulo \0), se usa la función *strlen()*.

```
char cadena[] = "Hola Mundo";
int longitud = strlen(cadena);
```

Búsqueda de Subcadenas

Para buscar una subcadena dentro de una cadena, se utiliza la función *strstr()*.

```
char cadena[] = "Hola Mundo";
char subcadena[] = "Mundo";

char *posicion = strstr(cadena, subcadena);
```

Cadenas (String)

Copia de Cadenas

Para copiar una cadena a otra, se utiliza *strcpy()* o su versión segura *strncpy()*.

```
char destino[20];
char origen[] = "Hola C";
strcpy(destino, origen);
```

Comparación Insensible a Mayúsculas/Minúsculas.

Para comparar cadenas ignorando mayúsculas y minúsculas, se usa *strcasecmp()* (no estándar, pero común en sistemas POSIX).

```
char cadena1[] = "Hola";
char cadena2[] = "HOLA";

strcasecmp(cadena1, cadena2) == 0
```

Formado de Cadenas

Para dar formato a una cadena y guardarla en otra, se puede usar *sprintf()*.

```
char buffer[50];
int edad = 25;

sprintf(buffer, "Tengo %d años", edad);
```

Cadenas (String)

Localización de la primera aparición de un carácter en una cadena

Para localizar la localidad de memoria de la primera aparición de un carácter en unca cadena, se utiliza strchr(). Regresa un puntero con la localidad o NULL en el caso que el carácter no se encuentre en la cadena.

```
char cadena[] = "Universidad de Guanajuato";
char *bc;
bc = strchr(cadena, 'a');
```

Comprobación si el carácter es alfanumérico

Para comprobar si un carácter es un digito decimal o una letra mayúscula o minúscula, se usa *isalnum()*. El resultado es verdadero *isalpha()* o *isdigit()* también devuelven verdadero.

```
char cadena[] = "División";
int i;
while(isalnum(cadena[i])) i++;
```

Conversión entre mayúsculas y minúsculas.

Para convertir de minúsculas a mayúsculas, se puede usar *toupper()*. En el caso de convertir de mayúsculas a minúsculas, se puede usar *tolower()*.

```
char cadena1[] = "Universidad de Guanajuato";
toupper(cadena[i]);
tolower(cadena[i]);
```

Cadenas (String)

Validación de dirección de correo electrónico

Para validar una dirección de correo electrónico de manera correcta, es importante seguir ciertas reglas y convenciones establecidas por los estándares de Internet, particularmente el RFC 5321 y RFC 5322. Aunque la validación completa de un correo electrónico puede llegar a ser compleja debido a las múltiples combinaciones válidas, a continuación, te proporciono las reglas más importantes y comunes que se deben seguir para una validación efectiva.

Reglas Básicas para Validar un Correo Electrónico:

1. *Estructura Básica:* Un correo electrónico válido debe seguir la estructura básica: Un **local part** (parte antes del @), seguido por el símbolo @, y luego un dominio (parte después del @). Formato: *local-part@dominio*
2. *Longitud del Correo Electrónico:* Máximo: Un correo electrónico completo no debe exceder los 254 caracteres. Local Part: La parte antes del @ puede tener hasta 64 caracteres. Dominio: La parte después del @ puede tener hasta 255 caracteres en total, aunque cada etiqueta de dominio (sección separada por un punto) debe tener un máximo de 63 caracteres.
3. *Reglas para la Parte Local (Local Part):* Caracteres válidos: Letras (a-z, A-Z), Números (0-9), Caracteres especiales permitidos: ! # $ % & ' * + / = ? ^ _ { | } ~ Puntos (.) pueden ser utilizados entre caracteres, pero no al principio o al final. El carácter punto (.) no puede aparecer consecutivamente. Comillas: La parte local puede estar entre comillas dobles (" "), lo que permite el uso de caracteres especiales adicionales, pero esto no es muy común y no siempre es soportado.
4. *Reglas para la Parte del Dominio:* Caracteres válidos: Letras (a-z, A-Z), Números (0-9), Guiones (-) pueden ser utilizados, pero no pueden aparecer al principio o al final de una etiqueta. Cada parte del dominio, separada por puntos, no debe exceder los 63 caracteres. El dominio debe terminar en una etiqueta válida, como .com, .org, .net, etc.

Cadenas (String)

Validación de dirección de correo electrónico

Para validar una dirección de correo electrónico de manera correcta, es importante seguir ciertas reglas y convenciones establecidas por los estándares de Internet, particularmente el RFC 5321 y RFC 5322. Aunque la validación completa de un correo electrónico puede llegar a ser compleja debido a las múltiples combinaciones válidas, a continuación, te proporciono las reglas más importantes y comunes que se deben seguir para una validación efectiva.

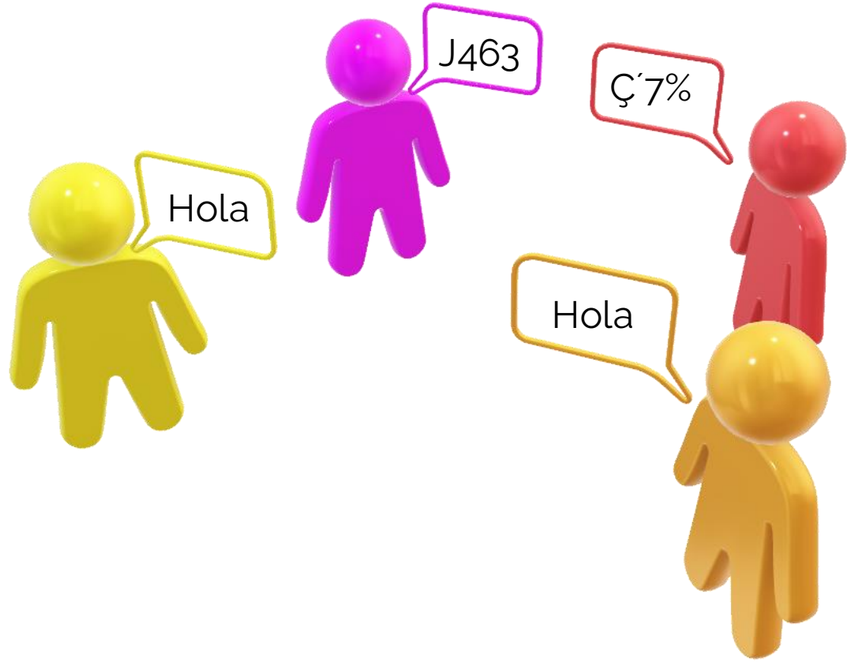
Reglas Básicas para Validar un Correo Electrónico:

5. *Sufijos del Dominio:* El dominio debe tener un TLD (Top Level Domain) válido, como .com, .org, .net, .edu, etc. Además de los dominios tradicionales, algunos TLD pueden ser más largos como .international, pero deben ser válidos según las reglas del sistema de nombres de dominio (DNS).
6. *No se permite Espacios en Blanco:* No se permiten espacios en blanco ni en la parte local ni en la parte del dominio.
7. *Casos Especiales:* Cuentas con Etiquetas (Tagging): Algunos servicios de correo electrónico permiten usar un + en la parte local para etiquetar correos. Por ejemplo, usuario+trabajo@dominio.com es válido. Correos Internacionalizados: Los dominios pueden incluir caracteres especiales (por ejemplo, letras con acentos) bajo la norma IDN (Internationalized Domain Names).
8. *Caracteres No Permitidos:* No se permiten caracteres especiales como (),:;<>[]@\" (excepto si están entre comillas dobles en la parte local, lo cual es raro). Backslash (\) solo puede usarse como un carácter de escape dentro de una dirección entre comillas.

Cadenas (String)

Encriptación de información

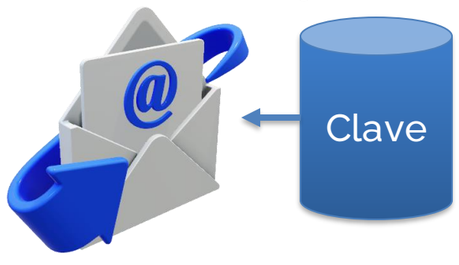
La criptografía es el procedimiento que traduce el texto sin formato (texto plano) en una secuencia ilegible de caracteres mediante una clave. El objetivo es que el contenido del texto secreto resultante o criptograma (texto cifrado) solo sea accesible para aquellos que disponen de la clave para descifrarlo. Los métodos criptográficos son aplicados a cualquier tipo de información electrónica como mensajes de voz, archivos de imagen o códigos de programación, además de a mensajes de texto.



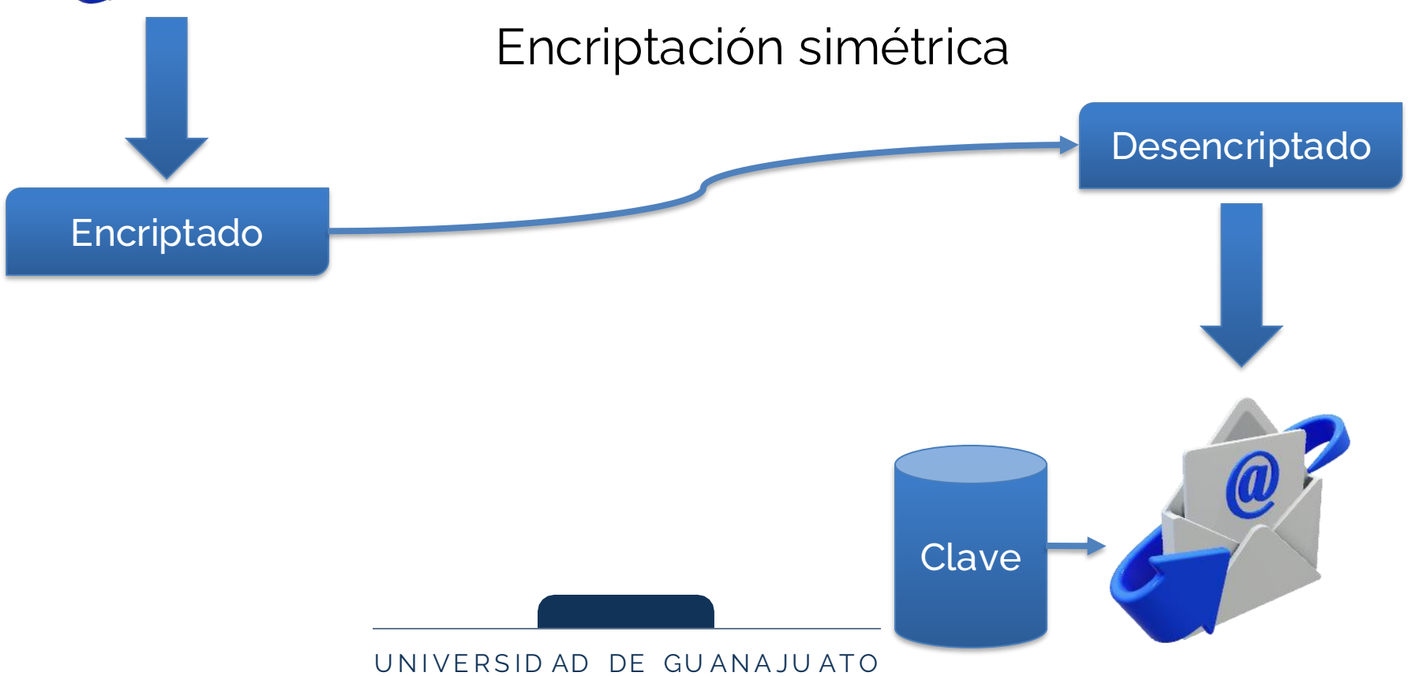
Cadenas (String)

Encriptación de información

La criptografía es el procedimiento que traduce el texto sin formato (texto plano) en una secuencia ilegible de caracteres mediante una clave. El objetivo es que el contenido del texto secreto resultante o criptograma (texto cifrado) solo sea accesible para aquellos que disponen de la clave para descifrarlo. Los métodos criptográficos son aplicados a cualquier tipo de información electrónica como mensajes de voz, archivos de imagen o códigos de programación, además de a mensajes de texto.

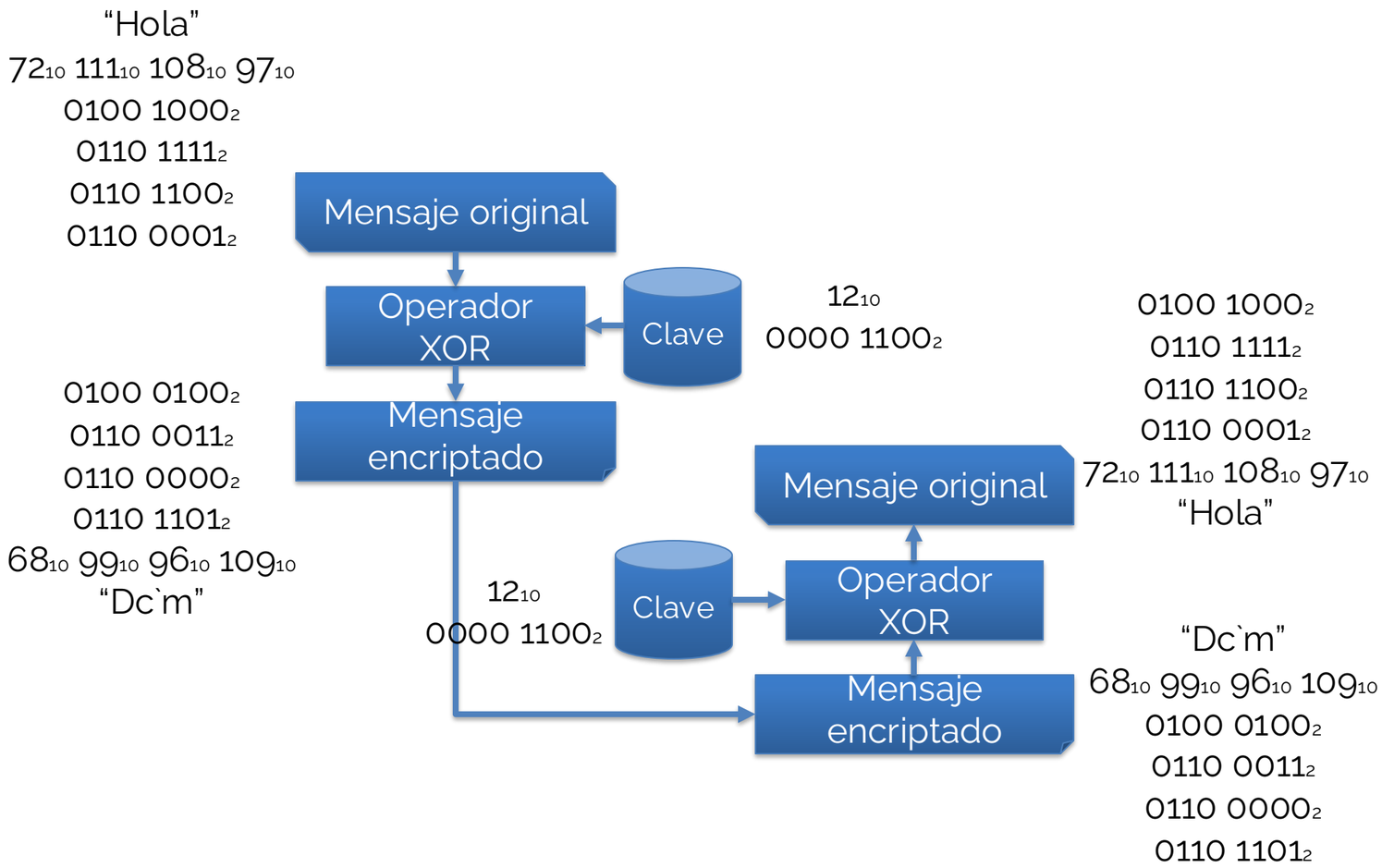


Encriptación simétrica



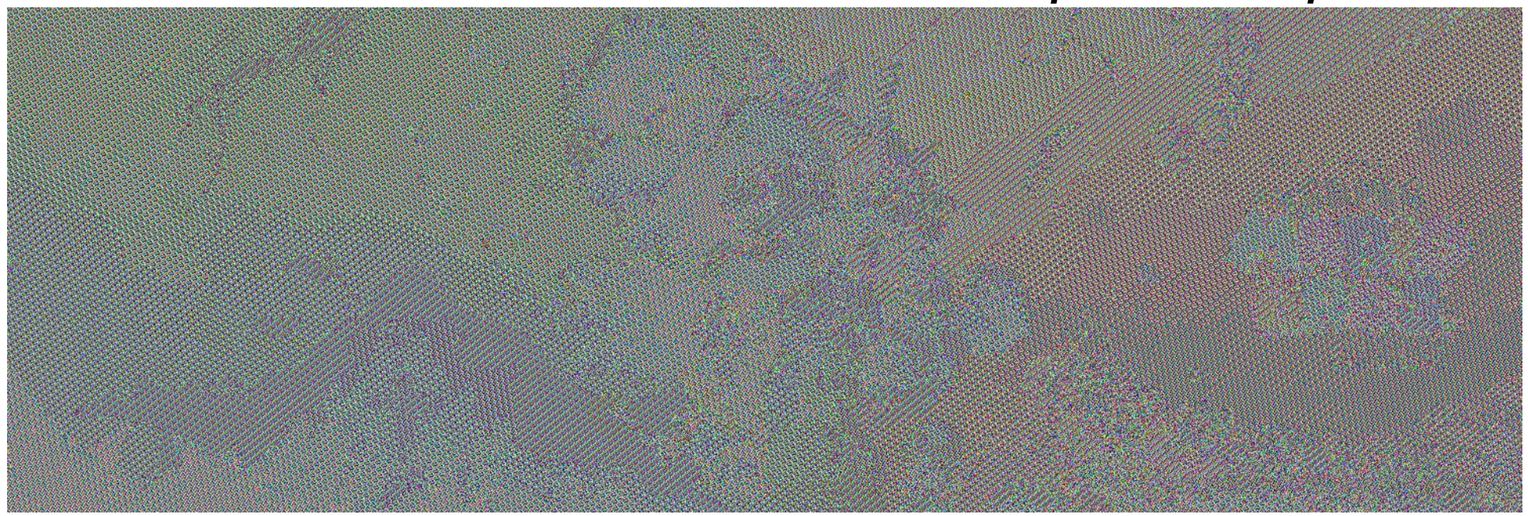
Cadenas (String)

Encriptación de información



Cadenas (String)

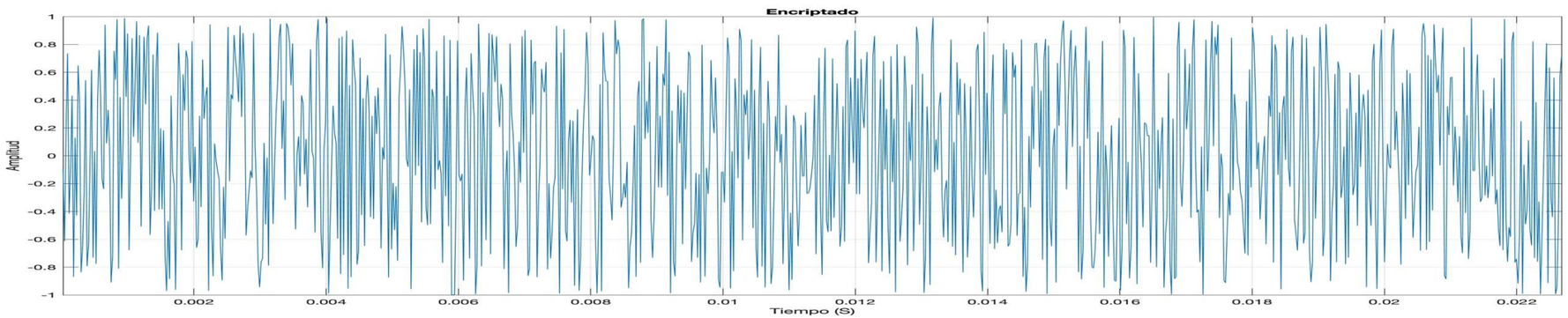
Encriptación de información

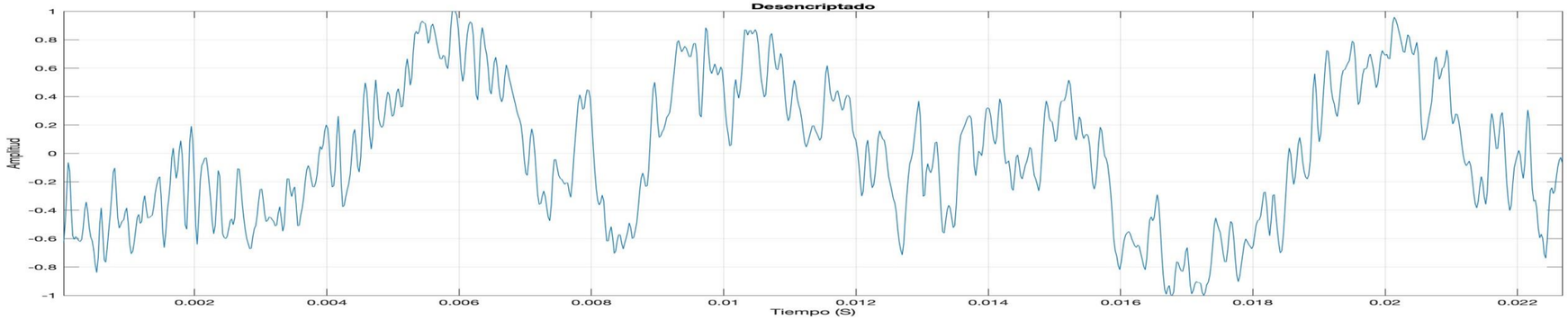


Clave = [229 198 184 43 51 87 245 217 56 40 223 44 133 116 83 18 174 178 46 210 183 20 192 106 221 103 5 209 244 75 65 80 168 207 151 177 217 226 6]

Cadenas (String)

Encriptación de información





Cadenas (String)

RC4 (Rivest Cipher 4)

El algoritmo RC4 es un cifrado de flujo simple y eficiente diseñado por Ron Rivest en 1987. Es un cifrado simétrico que utiliza una clave secreta para generar un flujo de bytes pseudoaleatorios que luego se XOR-ean con los datos de entrada (texto en claro) para producir el texto cifrado.

RC4 es un cifrado de flujo en el que se genera un flujo de bytes pseudoaleatorios que luego se aplica al texto mediante una operación XOR, de manera similar al Cifrado XOR, pero con un flujo de claves generado dinámicamente. Es fácil de implementar, y aunque tiene ciertas vulnerabilidades, sigue siendo mucho más seguro que algoritmos simples como César o XOR fijo.

Pasos del Algoritmo RC4:

1. **Inicialización del estado - KSA (Key Scheduling Algorithm):** Se inicializa un arreglo de 256 bytes llamado S con valores de 0 a 255. Luego, se mezcla el arreglo S con la clave secreta.

2. **Generación del flujo de claves - PRGA (Pseudo-Random Generation Algorithm):** El estado S se utiliza para generar un flujo de bytes pseudoaleatorios que se XOR-ean con el texto en claro para producir el texto cifrado.

3. **Cifrado y descifrado:** El proceso de cifrado y descifrado es idéntico, simplemente se XOR-ea el flujo de claves con los datos de entrada (texto en claro o texto cifrado).

Cadenas (String)

RC4 (Rivest Cipher 4)

Pasos del Algoritmo RC4:

- **KSA (Key Scheduling Algorithm).**

clave = “LID-IA” = [76 73 68 45 73 65] NC = #{clave} = 6
S[10] = [0 1 2 3 4 5 6 7 8 9] NS = #{S} = 10

j = (j+S[i]+clave[i%NC])%NS
SWAP(S[i], S[j])

i	j	S
0	0	[0 1 2 3 4 5 6 7 8 9]
0	(0+0+76)%10=6	[6 1 2 3 4 5 0 7 8 9]
1	(6+1+73)%10=0	[1 6 2 3 4 5 0 7 8 9]
2	(0+2+68)%10=0	[2 6 1 3 4 5 0 7 8 9]
3	(0+3+45)%10=8	[2 6 1 8 4 5 0 7 3 9]
4	(8+4+73)%10=5	[2 6 1 8 5 4 0 7 3 9]
5	(5+4+65)%10=4	[2 6 1 8 4 5 0 7 3 9]
6	(4+0+76)%10=0	[0 6 1 8 4 5 2 7 3 9]
7	(0+7+73)%10=0	[7 6 1 8 4 5 2 0 3 9]
8	(0+3+68)%10=1	[7 3 1 8 4 5 2 0 6 9]
9	(1+9+45)%10=5	[7 3 1 8 4 9 2 0 6 5]

S= [7 3 1 8 4 9 2 0 6 5]

Cadenas (String)

RC4 (Rivest Cipher 4)

Pasos del Algoritmo RC4:

- **PRGA (Pseudo-Random Generation Algorithm).**

S[10] = [7 3 1 8 4 9 2 0 6 5] NS = #[S] = 10
MSG[9] = “Algoritmo” NM = #[MSG] = 9
MSG[9] = [65 108 103 111 114 105 116 109 111]

i = (i + 1) % NS
j = (j + S[i]) % NS
SWAP(S[i], S[j])
S[(S[i] + S[j]) % NS]

k	i	j	S	S[(S[i]+S[j])%NS]
0	0	0	[7 3 1 8 4 9 2 0 6 5]	
0	1	3	[7 8 1 3 4 9 2 0 6 5]	S[1] = 8
1	2	4	[7 8 4 3 1 9 2 0 6 5]	S[5] = 9
2	3	7	[7 8 4 0 1 9 2 3 6 5]	S[3] = 0
3	4	8	[7 8 4 0 6 9 2 3 1 5]	S[7] = 3
4	5	7	[7 8 4 0 6 3 2 9 1 5]	S[2] = 4
5	6	9	[7 8 4 0 6 3 5 9 1 2]	S[7] = 9
6	7	8	[7 8 4 0 6 3 5 1 9 2]	S[0] = 7
7	8	7	[7 8 4 0 6 3 5 9 1 2]	S[0] = 7
8	9	9	[7 8 4 0 6 3 5 9 1 2]	S[4] = 6

S = [8 9 0 3 4 9 7 7 6]

Cadenas (String)

RC4 (Rivest Cipher 4)

Pasos del Algoritmo RC4:

- **Cifrado y descifrado.**

MSG[9] = “Algoritmo” NM = #[MSG] = 9
MSG[9] = [65 108 103 111 114 105 116 109 111]
S[9] = [8 9 0 3 4 9 7 7 6]

MSG[k] = MSG[k]^S[k]

MSG	MSG	MSG	S	S	MSG^S	MSG^S	MSG^S
A	65	0100 0001	8	0000 1000	0100 1001	73	I
l	108	0110 1100	9	0000 1001	0110 0101	101	e
g	103	0110 0111	0	0000 0000	0110 0111	103	g
o	111	0110 1111	3	0000 0011	0110 1100	108	l
r	114	0111 0010	4	0000 0100	0111 0110	118	v
i	105	0110 1001	9	0000 1001	0110 0000	96	`
t	116	0111 0100	7	0000 0111	0111 0011	115	s
m	109	0110 1101	7	0000 0111	0110 1010	106	j
o	111	0110 1111	6	0000 0110	0110 1001	105	i

Estructuras (Struct)

En el lenguaje de programación C, una estructura (o struct en inglés) es un tipo de dato definido por el usuario que permite agrupar bajo un mismo nombre diferentes variables de distintos tipos. Las estructuras proporcionan una manera de encapsular datos relacionados en una sola entidad, lo que facilita la organización y manejo de datos complejos.

Características

- **Composición Heterogénea:** A diferencia de los arreglos, que almacenan elementos del mismo tipo, una estructura puede contener variables de distintos tipos (enteros, flotantes, caracteres, otros structs, etc.).
- **Definición y Declaración:** Las estructuras se definen usando la palabra clave struct seguida de un nombre y un conjunto de miembros (variables) entre llaves {}. Una vez definida, una estructura se puede usar para declarar variables que tengan ese tipo de estructura.
- **Acceso a los Miembros:** Los miembros de una estructura se acceden utilizando el operador punto (.) para referirse a cada uno de los elementos que la componen.
- **Memoria Contigua:** Los miembros de una estructura se almacenan en ubicaciones de memoria contiguas, aunque la alineación de los miembros puede depender de restricciones específicas del compilador.

Estructuras (Struct)

Sintaxis básica:

```
struct identificador {
        tipo miembro_1;
        tipo miembro_2;
        ...
}

struct identificador variable_1, variable_2;
variable_1.miembro_1
```

Ejemplo:

```
struct usuario {
        char nombre[50];
        int edad;
        float altura;
}

struct usuario u1, u2;

u1.edad
```

Sintaxis básica:

```
typedef struct identificador {
        tipo miembro_1;
        tipo miembro_2;
        ...
}

identificador variable_1, variable_2;
variable_1.miembro_1
```

Ejemplo:

```
typedef struct usuario {
        char nombre[50];
        int edad;
        float altura;
}

usuario u1, u2;

u1.edad
```

Estructuras (Struct)

Operaciones con números complejos

Los números complejos están integrados por dos partes, una real (a) y una imaginaria (b), la combinación de estas dos partes forma un número complejo (Z)

$$z = a + bi$$

Las operaciones en espacio complejo se definen como:

$$z_1 \pm z_2 = (a_1 + a_2) \pm (b_1 + b_2)i$$

$$z_1 \times z_2 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i$$

$$z_1^* = a_1 - b_1 i$$

$$\frac{1}{z_1} = z_1^{-1} = \frac{a_1}{a_1^2 + b_1^2} - \frac{b_1}{a_1^2 + b_1^2} i$$

$$\frac{z_1}{z_2} = z_1 \times z_2^{-1} = \left(\frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2}\right) + \left(\frac{a_2 b_1 - a_1 b_2}{a_2^2 + b_2^2}\right) i$$

$$|z_1| = \sqrt{a_1^2 + b_1^2}$$

$$\theta = tan^{-1}\left(\frac{b}{a}\right)$$

Uniones (Union)

En el lenguaje de programación C, una unión (o union en inglés) es un tipo de dato definido por el usuario que permite agrupar varias variables bajo un mismo nombre, de manera similar a una estructura. Sin embargo, a diferencia de las estructuras, en una unión, todos los miembros comparten el mismo espacio de memoria, lo que significa que solo uno de los miembros puede almacenar un valor en un momento dado.

Características

- **Memoria Compartida:** Todos los miembros de una unión comparten el mismo espacio de memoria. La cantidad de memoria asignada para una unión es igual a la del miembro más grande. Esto permite ahorrar espacio de memoria, pero con la restricción de que solo un miembro puede contener un valor válido a la vez.
- **Acceso Exclusivo a Miembros:** Aunque se pueden definir múltiples miembros dentro de una unión, en un momento dado solo se puede acceder a un miembro correctamente, ya que la modificación de un miembro afecta a todos los demás.
- **Definición y Declaración:** Las uniones se definen utilizando la palabra clave union seguida de un nombre y un conjunto de miembros entre llaves {}. Las variables de tipo unión se pueden declarar después de la definición.
- **Flexibilidad en el Uso de Tipos de Datos:** Las uniones son útiles cuando se necesita almacenar diferentes tipos de datos en la misma ubicación de memoria, en diferentes momentos del programa.

Unión (Union)

Sintaxis básica:

```
union identificador {
        tipo miembro_1;
        tipo miembro_2;
        ...
}

union identificador variable_1, variable_2;

variable_1.miembro_1
```

Ejemplo:

```
union error {
        int numero;
        char cadena[20];
        float valor;
}

union error e1, e2;

e1.numero
```

Sintaxis básica:

```
typedef union identificador {
        tipo miembro_1;
        tipo miembro_2;
        ...
}

identificador variable_1, variable_2;

variable_1.miembro_1
```

Ejemplo:

```
typedef union error {
        int numero;
        char cadena[20];
        float valor;
}

error e1, e2;

e1.numero
```

Unión (Union)

Ejemplo básico:

```
typedef union dato{
        float f;
        int i;
        char c[4];
}

dato d;
```

d.i = 2024;
d.c[3] = 0 d.c[2] = 0 d.c[1] = 7 d.c[0] = 232
7*256+232 = 2024

d.i = -5323
d.c[3] = 255 d.c[2] = 255 d.c[1] = 235 d.c[0] = 53

d.c[3] = $1111\ 1111_2$ d.c[2] = $1111\ 1111_2$ d.c[1] = $1110\ 1011_2$ d.c[0] = $0011\ 0101_2$

d.c[3] = $0000\ 0000_2$ d.c[2] = $0000\ 0000_2$ d.c[1] = $0001\ 0100_2$ d.c[0] = $1100\ 1010_2$

d.c[3] = 0 d.c[2] = 0 d.c[1] = 20 d.c[0] = 202
20*256+202+1=5323

Unión (Union)

Ejemplo básico:

$$(-1)^S \times (1.M) \times 2^{(E-bias)}$$

IEEE 754
1. 32 bits – 1 bit signo (0, positivo), 8 bits exponente (127 simple y 1023 doble) y 23 bits mantisa
2. 64 bits – 1 bit signo, 11 bits exponencial y 52 bits mantisa

d.f = 1;
d.c[0] = 63 d.c[1] = 128 d.c[2] = 0 d.c[3] = 0
d.c[0] = $0011\ 1111_2$ d.c[1] = $1000\ 0000_2$ d.c[2] = $0000\ 0000_2$ d.c[3] = $0000\ 0000_2$

$$(-1)^0 \times (1.00000000000000000000000) \times 2^{01111111_2-127} = 1 \times 2^0 = 1$$

d.f = 3.1415926535897 93;
d.c[0] = 64 d.c[1] = 73 d.c[2] = 15 d.c[3] = 219
d.c[0] = $0100\ 0000_2$ d.c[1] = $0100\ 1001_2$ d.c[2] = $0000\ 1111_2$ d.c[3] = $1101\ 1011_2$

$$(-1)^0 \times (1.10010010000111111011011) \times 2^{10000000_2-127} = 1.570796370506287 \times 2^1 = 3.141592741012573$$

$$\frac{10010010000111111011011}{10000000000000000000000} = \frac{4788187}{8388608} = 0.570796370506287$$

Enumeración (Enum)

En el lenguaje de programación C, una enumeración (o enum en inglés) es un tipo de dato definido por el usuario que permite asignar nombres simbólicos a un conjunto de valores enteros constantes. Las enumeraciones se utilizan para mejorar la legibilidad y la claridad del código al dar nombres significativos a valores enteros que representan estados, opciones o categorías.

Características

- **Conjunto de Constantes Enteras:** Una enumeración define un conjunto de constantes enteras con nombres simbólicos, lo que facilita la comprensión y el mantenimiento del código.
- **Asignación Automática de Valores:** De forma predeterminada, los valores enteros asignados a los nombres de una enumeración comienzan en 0 y se incrementan en 1 por cada elemento sucesivo. Sin embargo, es posible especificar valores específicos para cualquier miembro de la enumeración.
- **Definición y Uso:** Las enumeraciones se definen usando la palabra clave enum seguida de un nombre y una lista de identificadores entre llaves {}. Después de la definición, se pueden declarar variables de tipo enumeración que solo pueden tomar los valores definidos en la enumeración.
- **Mejora de la Legibilidad:** Usar nombres simbólicos en lugar de valores enteros crudos hace que el código sea más fácil de leer y entender, especialmente en contextos donde ciertos valores enteros tienen un significado específico.

Punteros (Pointer)

En el lenguaje de programación C, un puntero es una variable que almacena la dirección de memoria de otra variable. Los punteros son una característica poderosa y fundamental de C, que permiten la manipulación directa de la memoria y el manejo eficiente de datos.

Características

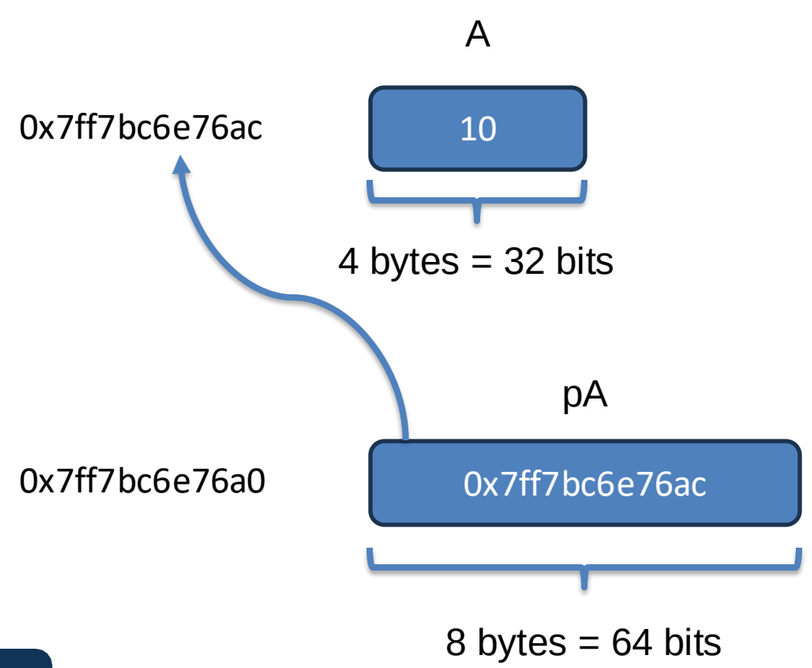
- **Almacenamiento de Direcciones:** Un puntero no almacena un valor directamente, sino la dirección en memoria donde se encuentra almacenado el valor de otra variable. Por ejemplo, un puntero a un entero (int) almacenará la dirección en la memoria donde reside ese entero.
- **Declaración de Punteros:** Los punteros se declaran utilizando el operador asterisco (*) junto con el tipo de dato al que apuntan. Por ejemplo, int *ptr declara un puntero a un entero.
- **Operador de Dirección (&):** Este operador se utiliza para obtener la dirección de memoria de una variable. Por ejemplo, &x obtiene la dirección de la variable x.
- **Operador de Indirección (*):** Cuando se utiliza con un puntero, este operador se llama "operador de indirección" y se usa para acceder al valor almacenado en la dirección de memoria a la que apunta el puntero.
- **Manipulación de Memoria:** Los punteros permiten la manipulación directa de la memoria, lo que es útil para la creación de estructuras de datos dinámicas, la realización de operaciones de bajo nivel, y la optimización del rendimiento en aplicaciones de alto rendimiento.

Punteros (Pointer)

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int A, *pA;
        A = 10;
        pA = &A;
        printf("A = %d\npA = %p\n*pA = %d\n&pA = %p\n", A, pA, *pA, &pA);
        printf("sizeof(A) = %lu\nsizeof(pA) = %lu\n",sizeof(A), sizeof(pA));
        return 0;
}
```

A = 10
pA = 0x7ff7bc6e76ac
*pA = 10
&pA = 0x7ff7bc6e76a0
sizeof(A) = 4
sizeof(pA) = 8



Punteros (Pointer)

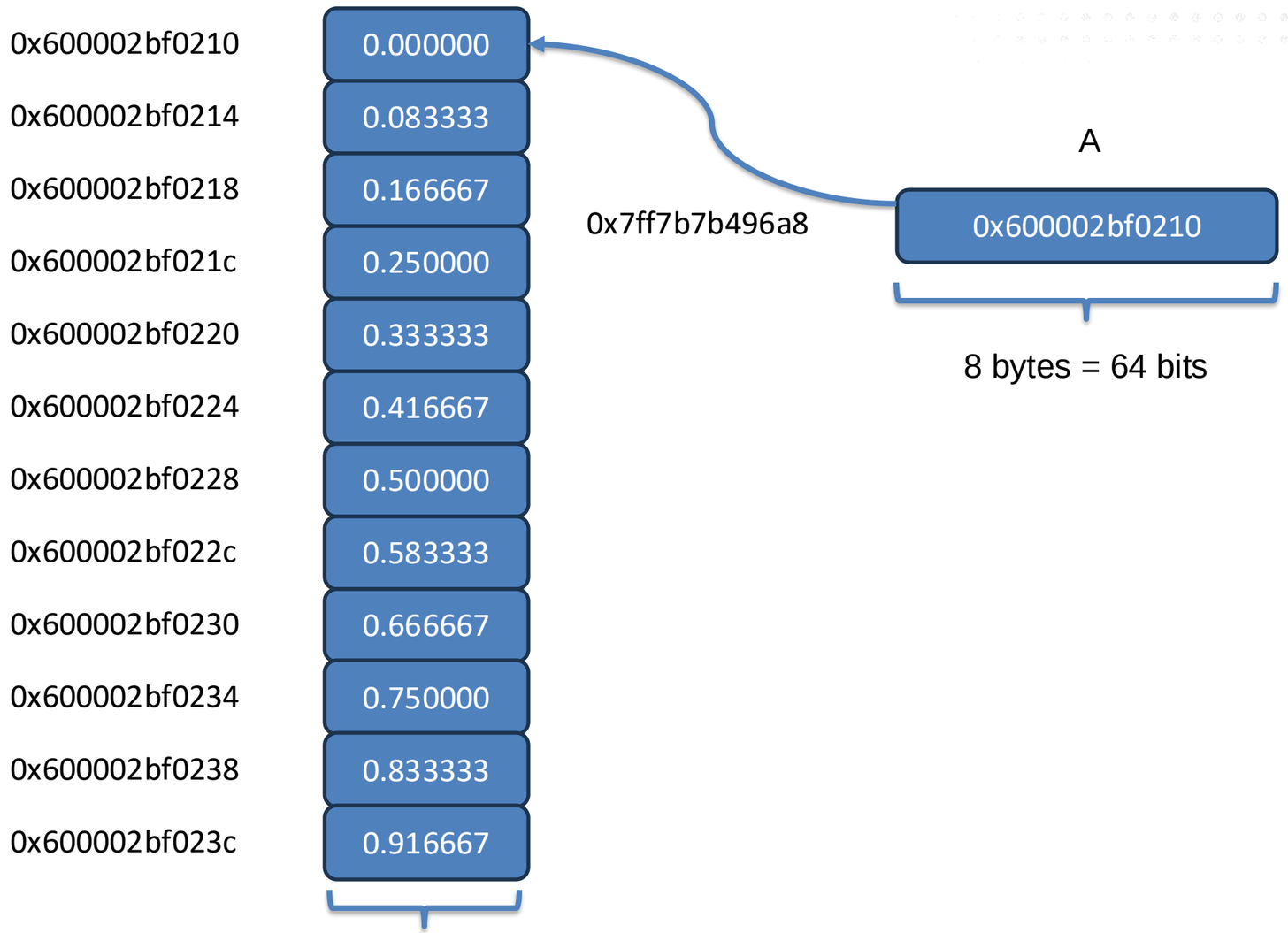
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        float *A;
        int n, i;
        n = 12;
        A=(float*)malloc(n*sizeof(float));
        if(A==NULL)
                return 1;
        for(i=0; i<n; i++)
                A[i] = (float)i/n;
        printf("A = %p\n&A = %p\n", A, &A);
        printf("sizeof(float) = %lu\nsizeof(float*) = %lu\n",
                sizeof(A[0]), sizeof(A));
        for(i=0; i<n; i++)
                printf("%p\tA[%d] = %f\n", A+i, i+1, A[i]);
        free(A);
        return 0;
}
```

```
A = 0x600002bf0210
&A = 0x7ff7b7b496a8
sizeof(float) = 4
sizeof(float*) = 8
0x600002bf0210 A[1] = 0.000000
0x600002bf0214 A[2] = 0.083333
0x600002bf0218 A[3] = 0.166667
0x600002bf021c A[4] = 0.250000
0x600002bf0220 A[5] = 0.333333
0x600002bf0224 A[6] = 0.416667
0x600002bf0228 A[7] = 0.500000
0x600002bf022c A[8] = 0.583333
0x600002bf0230 A[9] = 0.666667
0x600002bf0234 A[10] = 0.750000
0x600002bf0238 A[11] = 0.833333
0x600002bf023c A[12] = 0.916667
```

Punteros (Pointer)

A = 0x600002bf0210
&A = 0x7ff7b7b496a8
sizeof(float) = 4
sizeof(float*) = 8
0x600002bf0210 A[1] = 0.000000
0x600002bf0214 A[2] = 0.083333
0x600002bf0218 A[3] = 0.166667
0x600002bf021c A[4] = 0.250000
0x600002bf0220 A[5] = 0.333333
0x600002bf0224 A[6] = 0.416667
0x600002bf0228 A[7] = 0.500000
0x600002bf022c A[8] = 0.583333
0x600002bf0230 A[9] = 0.666667
0x600002bf0234 A[10] = 0.750000
0x600002bf0238 A[11] = 0.833333
0x600002bf023c A[12] = 0.916667



Punteros (Pointer)

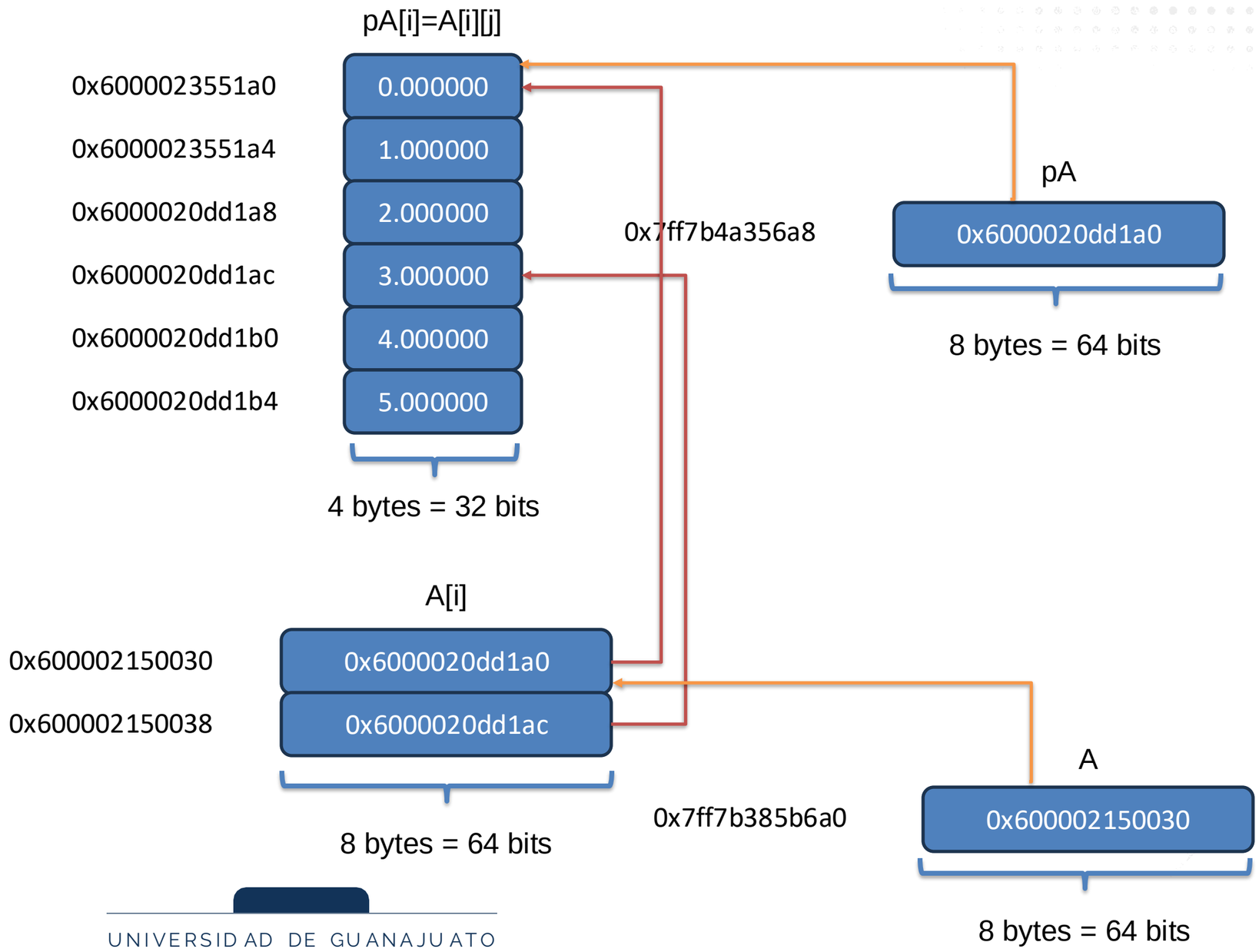
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        float *pA, **A;
        int n, m, i, j;
        n = 3;
        m = 2;
        pA=(float*)malloc(n*m*sizeof(float));
        if(pA==NULL)
                return 1;
        A=(float**)malloc(m*sizeof(float*));
        if(A==NULL)
        {
                free(pA);
                return 2;
        }
        for(i=0; i<m; i++)
                A[i]=pA+i*n;
        for(i=0; i<m; i++)
                for(j=0; j<n; j++)
                        A[i][j] = i*n+j;
        printf("pA = %p\n", pA);
        printf("&pA = %p\n", &pA);
        printf("*pA = %f\n", *pA);
        printf("sizeof(pA) = %lu\n", sizeof(pA));
        printf("A = %p\n", A);
        printf("&A = %p\n", &A);
        printf("*A = %p\n", *A);
        printf("**A = %f\n", **A);
        printf("sizeof(A) = %lu\n", sizeof(A));
        for(i=0; i<m; i++)
                printf("%p\tA[%d] = %p\n", A+i, i, A[i]);
        for(i=0; i<m; i++)
                for(j=0; j<n; j++)
                        printf("%p\tA[%d][%d] = %f\n", pA+i*n+j, i, j, A[i][j]);
        free(pA);
        free(A);
        return 0;
}
```

```
pA = 0x6000023551a0
&pA = 0x7ff7b385b6a8
*pA = 0.000000
sizeof(pA) = 8
A = 0x600002150030
&A = 0x7ff7b385b6a0
*A = 0x6000023551a0
**A = 0.000000
sizeof(A) = 8
0x600002150030 A[0] = 0x6000023551a0
0x600002150038 A[1] = 0x6000023551ac
0x6000023551a0 A[0][0] = 0.000000
0x6000023551a4 A[0][1] = 1.000000
0x6000023551a8 A[0][2] = 2.000000
0x6000023551ac A[1][0] = 3.000000
0x6000023551b0 A[1][1] = 4.000000
0x6000023551b4 A[1][2] = 5.000000
```

Punteros (Pointer)

```
pA = 0x6000023551a0
&pA = 0x7ff7b385b6a8
*pA = 0.000000
sizeof(pA) = 8
A = 0x600002150030
&A = 0x7ff7b385b6a0
*A = 0x6000023551a0
**A = 0.000000
sizeof(A) = 8
0x600002150030 A[0] = 0x6000023551a0
0x600002150038 A[1] = 0x6000023551ac
0x6000023551a0 A[0][0] = 0.000000
0x6000023551a4 A[0][1] = 1.000000
0x6000023551a8 A[0][2] = 2.000000
0x6000023551ac A[1][0] = 3.000000
0x6000023551b0 A[1][1] = 4.000000
0x6000023551b4 A[1][2] = 5.000000
```



Punteros (Pointer)

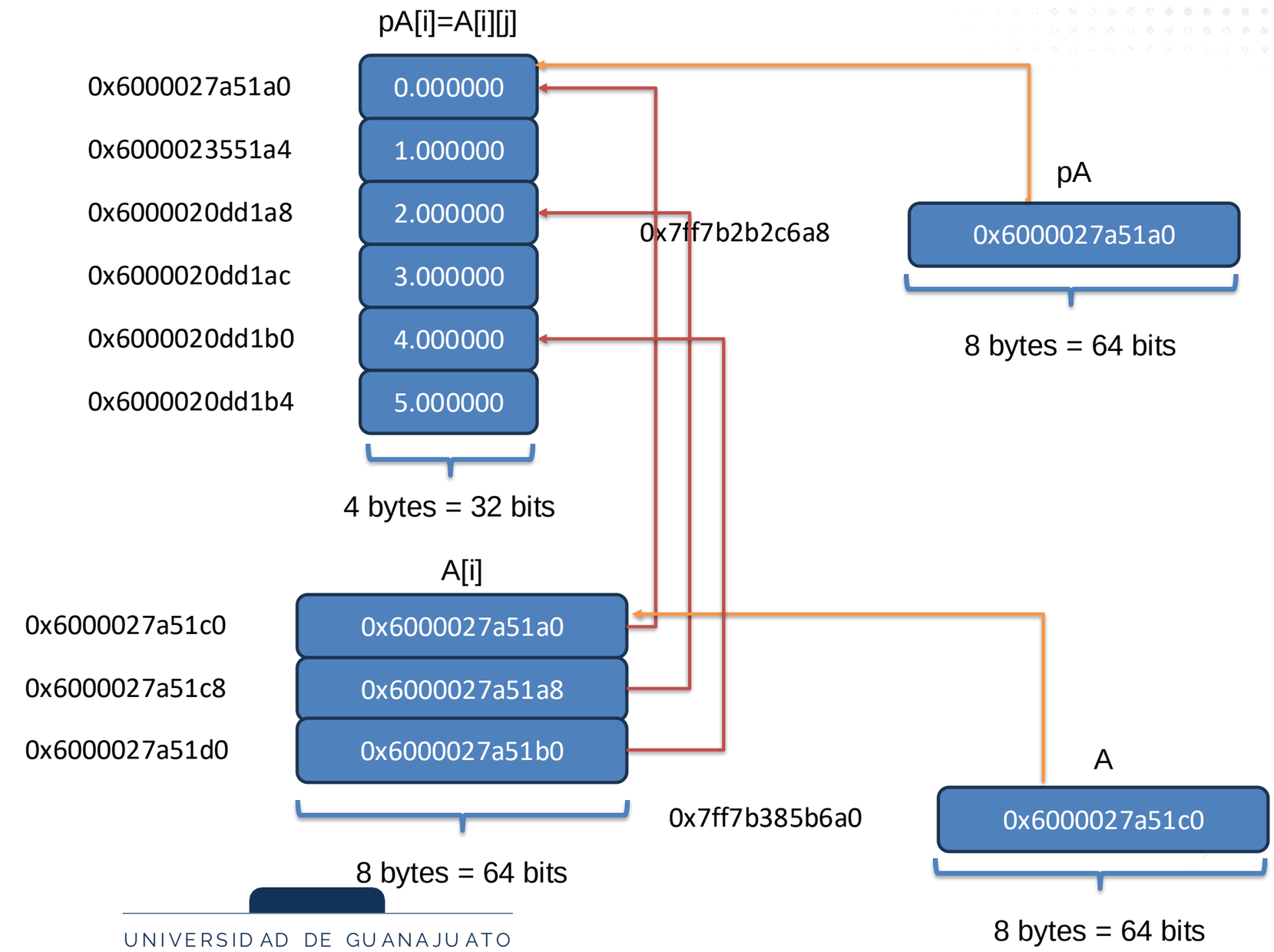
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        float *pA, **A;
        int n, m, i, j;
        n = 3;
        m = 2;
        pA=(float*)malloc(n*m*sizeof(float));
        if(pA==NULL)
                return 1;
        A=(float**)malloc(n*sizeof(float*));
        if(A==NULL)
        {
                free(pA);
                return 2;
        }
        for(i=0; i<n; i++)
                A[i]=pA+i*m;
        for(i=0; i<n; i++)
                for(j=0; j<m; j++)
                        A[i][j] = i*m+j;
        printf("pA = %p\n", pA);
        printf("&pA = %p\n", &pA);
        printf("*pA = %f\n", *pA);
        printf("sizeof(pA) = %lu\n", sizeof(pA));
        printf("A = %p\n", A);
        printf("&A = %p\n", &A);
        printf("*A = %p\n", *A);
        printf("**A = %f\n", **A);
        printf("sizeof(A) = %lu\n", sizeof(A));
        for(i=0; i<n; i++)
                printf("%p\tA[%d] = %p\n", A+i, i, A[i]);
        for(i=0; i<n; i++)
                for(j=0; j<m; j++)
                        printf("%p\tA[%d][%d] = %f\n", pA+i*m+j, i, j, A[i][j]);
        free(pA);
        free(A);
        return 0;
}
```

```
pA = 0x6000027a51a0
&pA = 0x7ff7b2b2c6a8
*pA = 0.000000
sizeof(pA) = 8
A = 0x6000027a51c0
&A = 0x7ff7b2b2c6a0
*A = 0x6000027a51a0
**A = 0.000000
sizeof(A) = 8
0x6000027a51c0 A[0] = 0x6000027a51a0
0x6000027a51c8 A[1] = 0x6000027a51a8
0x6000027a51d0 A[2] = 0x6000027a51b0
0x6000027a51a0 A[0][0] = 0.000000
0x6000027a51a4 A[0][1] = 1.000000
0x6000027a51a8 A[1][0] = 2.000000
0x6000027a51ac A[1][1] = 3.000000
0x6000027a51b0 A[2][0] = 4.000000
0x6000027a51b4 A[2][1] = 5.000000
```

Punteros (Pointer)

```
pA = 0x6000027a51a0
&pA = 0x7ff7b2b2c6a8
*pA = 0.000000
sizeof(pA) = 8
A = 0x6000027a51c0
&A = 0x7ff7b2b2c6a0
*A = 0x6000027a51a0
**A = 0.000000
sizeof(A) = 8
0x6000027a51c0 A[0] = 0x6000027a51a0
0x6000027a51c8 A[1] = 0x6000027a51a8
0x6000027a51d0 A[2] = 0x6000027a51b0
0x6000027a51a0 A[0][0] = 0.000000
0x6000027a51a4 A[0][1] = 1.000000
0x6000027a51a8 A[1][0] = 2.000000
0x6000027a51ac A[1][1] = 3.000000
0x6000027a51b0 A[2][0] = 4.000000
0x6000027a51b4 A[2][1] = 5.000000
```



Punteros (Pointer)

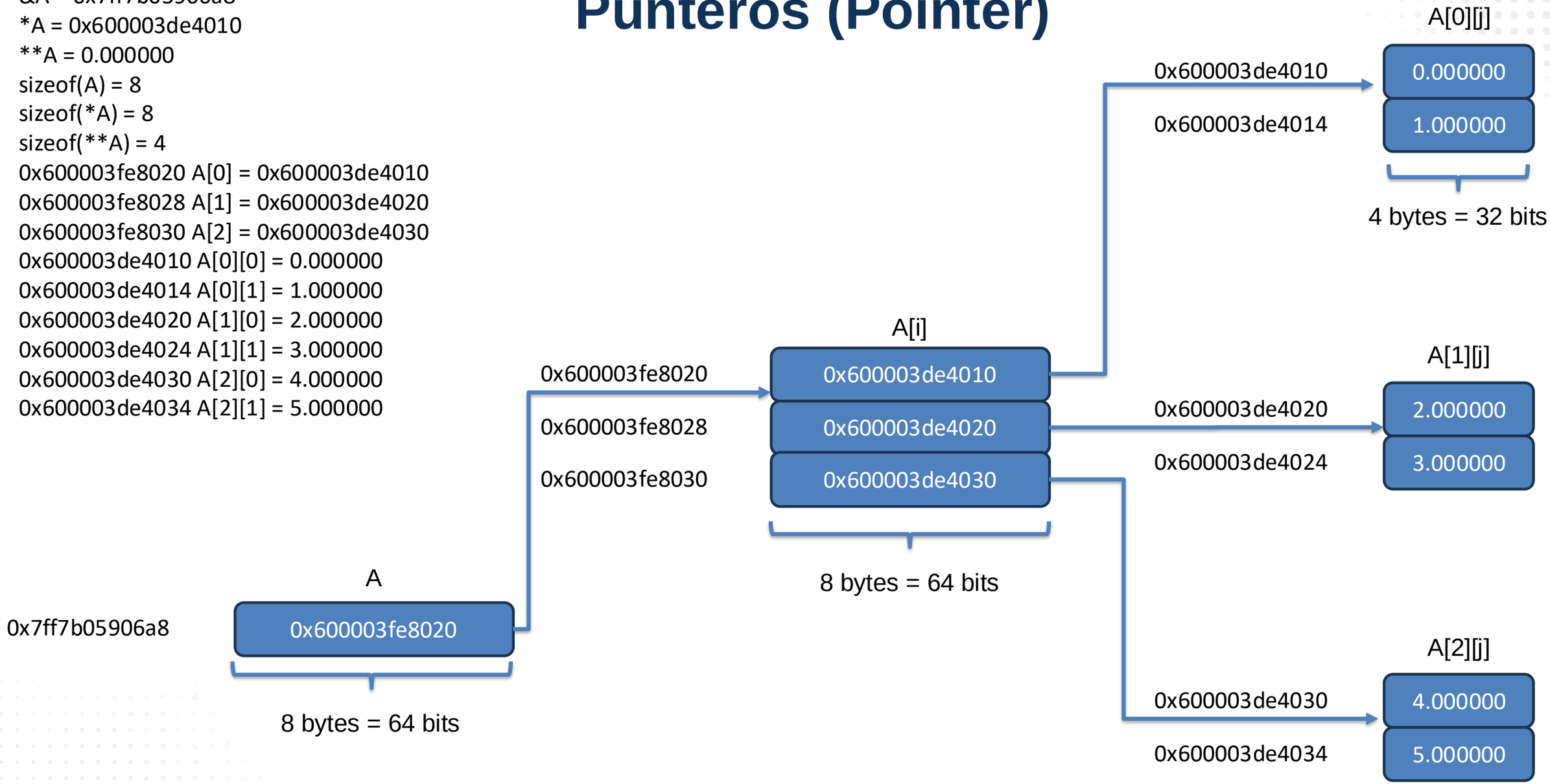
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        float **A;
        int n, m, i, j;
        n = 3;
        m = 2;
        A=(float**)malloc(n*sizeof(float*));
        if(A==NULL)
                return 1;
        for(i=0; i<n; i++)
        {
                A[i] = (float*)malloc(m*sizeof(float*));
                if(A[i]==NULL)
                {
                        for(i--; i>-1; i--)
                                free(A[i]);
                        return 2;
                }
        }
        for(i=0; i<n; i++)
                for(j=0; j<m; j++)
                        A[i][j] = i*m+j;
        printf("A = %p\n", A);
        printf("&A = %p\n", &A);
        printf("**A = %p\n", *A);
        printf("***A = %f\n", **A);
        printf("sizeof(A) = %lu\n", sizeof(A));
        printf("sizeof(*A) = %lu\n", sizeof(*A));
        printf("sizeof(**A) = %lu\n", sizeof(**A));
        for(i=0; i<n; i++)
                printf("%p\tA[%d] = %p\n", A+i, i, A[i]);
        for(i=0; i<n; i++)
                for(j=0; j<m; j++)
                        printf("%p\tA[%d][%d] = %f\n", A[i]+j, i, j, A[i][j]);
        for(i=0; i<n; i++)
                free(A[i]);
        free(A);
        return 0;
}
```

```
A = 0x600003fe8020
&A = 0x7ff7b05906a8
*A = 0x600003de4010
**A = 0.000000
sizeof(A) = 8
sizeof(*A) = 8
sizeof(**A) = 4
0x600003fe8020 A[0] = 0x600003de4010
0x600003fe8028 A[1] = 0x600003de4020
0x600003fe8030 A[2] = 0x600003de4030
0x600003de4010 A[0][0] = 0.000000
0x600003de4014 A[0][1] = 1.000000
0x600003de4020 A[1][0] = 2.000000
0x600003de4024 A[1][1] = 3.000000
0x600003de4030 A[2][0] = 4.000000
0x600003de4034 A[2][1] = 5.000000
```

Punteros (Pointer)

A = 0x600003fe8020
&A = 0x7ff7b05906a8
*A = 0x600003de4010
**A = 0.000000
sizeof(A) = 8
sizeof(*A) = 8
sizeof(**A) = 4
0x600003fe8020 A[0] = 0x600003de4010
0x600003fe8028 A[1] = 0x600003de4020
0x600003fe8030 A[2] = 0x600003de4030
0x600003de4010 A[0][0] = 0.000000
0x600003de4014 A[0][1] = 1.000000
0x600003de4020 A[1][0] = 2.000000
0x600003de4024 A[1][1] = 3.000000
0x600003de4030 A[2][0] = 4.000000
0x600003de4034 A[2][1] = 5.000000



Punteros (Pointer)

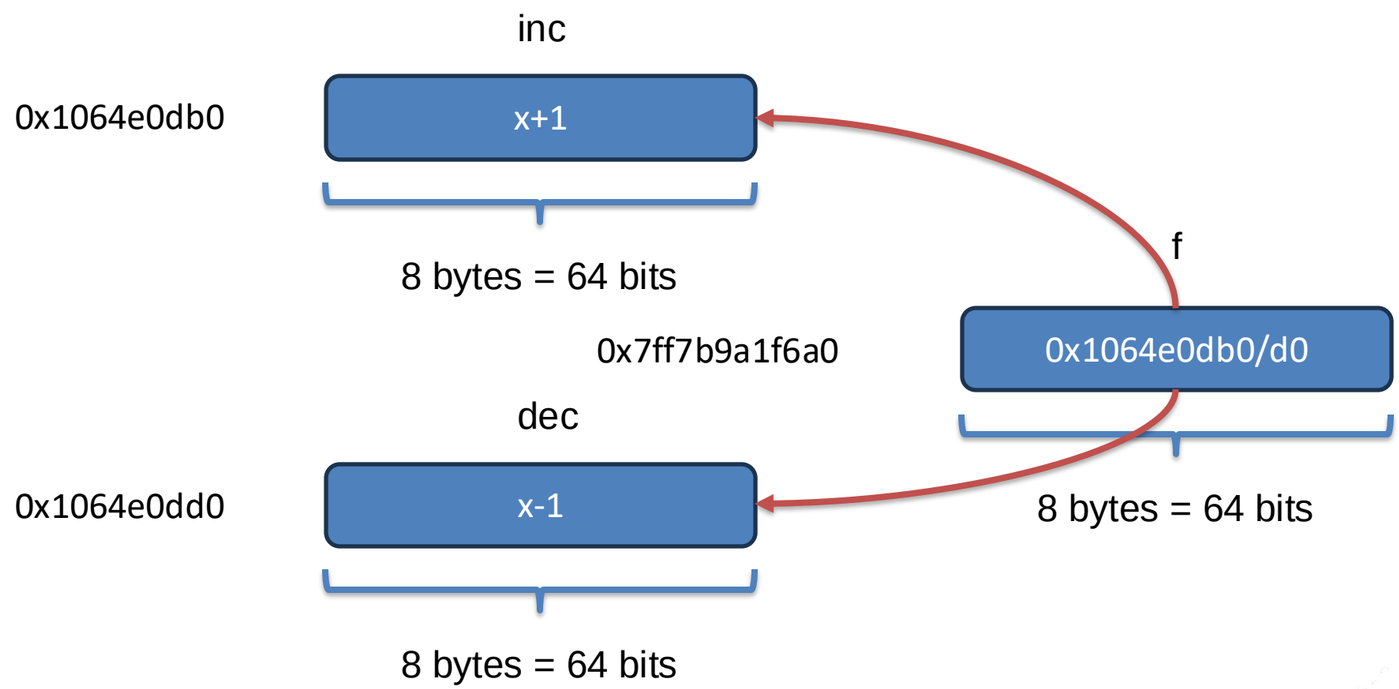
```
#include <stdio.h>
#include <stdlib.h>

float inc(float x)
{
        return x+1;
}

float dec(float x)
{
        return x-1;
}

int main(int argc, char *argv[])
{
        float x1 = 10;
        float (*f)(float);
        printf("sizeof(f) = %lu\n", sizeof(f));
        printf("sizeof(&inc) = %lu\n", sizeof(&inc));
        printf("sizeof(dec) = %lu\n", sizeof(dec));
        printf("&inc = %p\n", &inc);
        printf("&dec = %p\n", &dec);
        printf("&f = %p\n", &f);
        f=&inc;
        printf("%p\tf(%f) = %f\n", f, x1, f(x1));
        f=dec;
        printf("%p\tf(%f) = %f\n", f, x1, f(x1));
        return 0;
}
```

```
sizeof(f) = 8
sizeof(&inc) = 8
sizeof(dec) = 1
&inc = 0x1064e0db0
&dec = 0x1064e0dd0
&f = 0x7ff7b9a1f6a0
0x1064e0db0 f(10.000000) = 11.000000
0x1064e0dd0 f(10.000000) = 9.000000
```



Punteros (Pointer)

FILE: Tipo de objeto que identifica un flujo y contiene la información necesaria para controlar este, incluyendo un puntero al buffer,

Definición de funciones

En la práctica es importante dividir un programa en unidades razonables e independientes, por ello, la mayoría de los lenguajes de programación, incluido C, proporcionan una forma de dividir un programa en segmentos, cada uno de los cuales puede escribirse de forma más o menos independiente de los demás.

En C, estos segmentos se denominan funciones. El código de programa en el cuerpo de una función está aislado del de otras funciones. Una función tendrá una interfaz específica con el mundo exterior en términos de cómo se le transfiere la información y cómo se transmiten los resultados generados por la función. Esta interfaz se especifica en la primera línea de la función, donde aparece el nombre de la función.

En C, el concepto de función se puede explicar de varias formas.

Por ejemplo:

- Una función es un subprograma. Te permite romper una gran tarea de computación en otras más pequeñas.
- Una función es un fragmento de código delimitado por llaves que realiza alguna tarea bien definida y también devuelve un valor.
- Una función es un componente básico de un programa. Es útil para hacer código existente reutilizable.
- Una función también se trata como un tipo derivado en C.
- Una función es una forma de ampliar el repertorio del lenguaje C