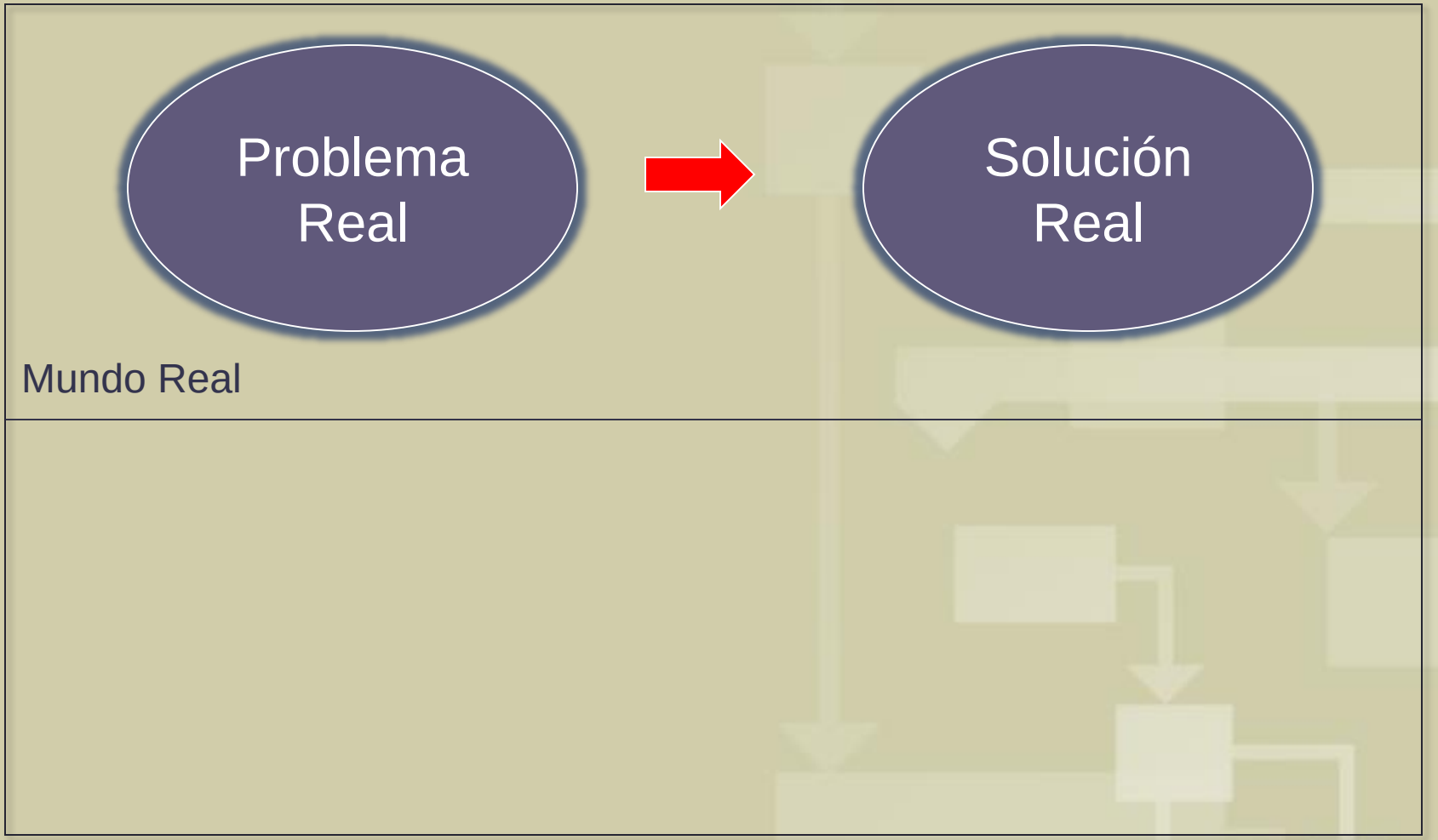
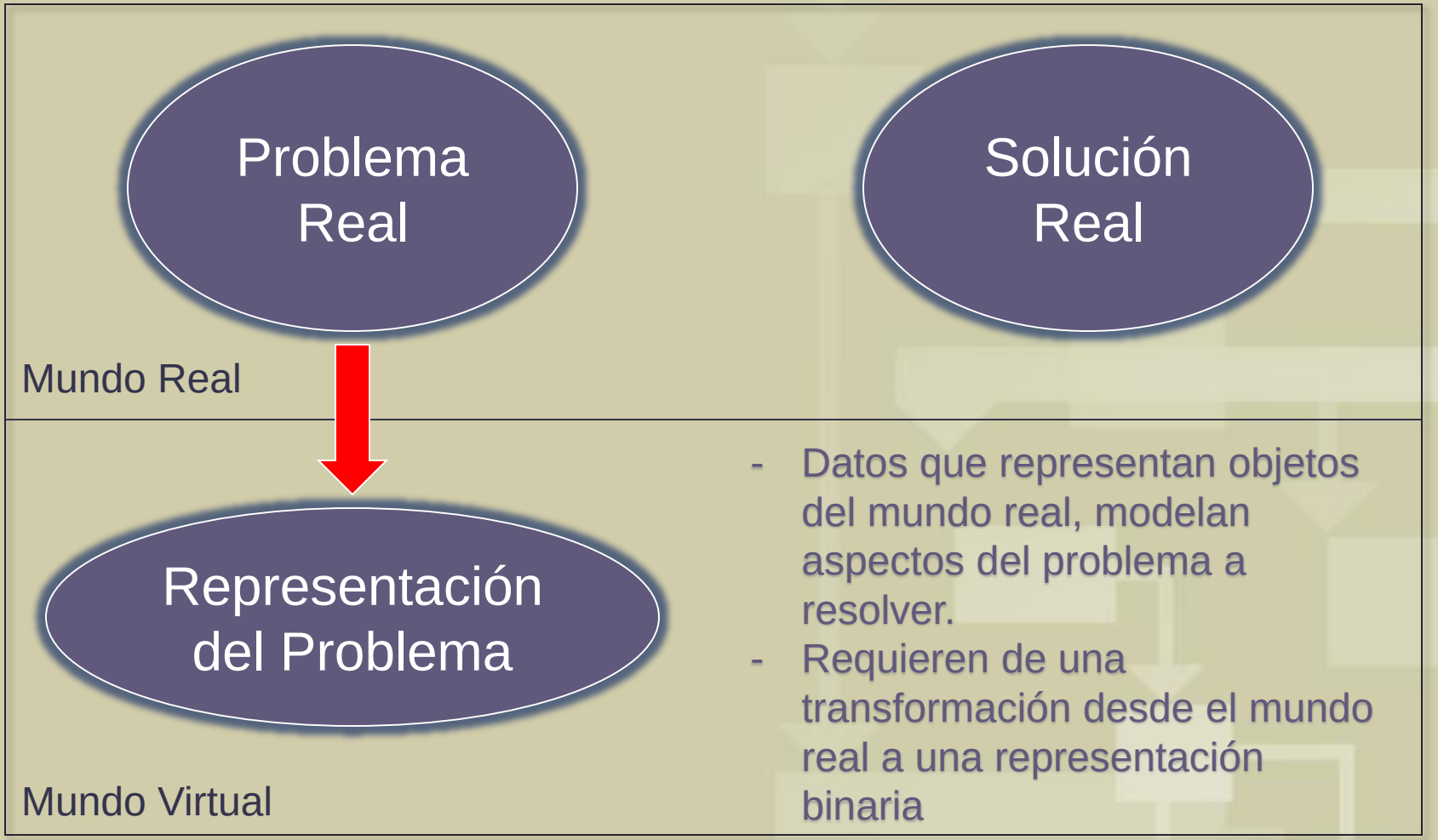
Introducción al Cómputo

Algoritmos y Resolución de Problemas

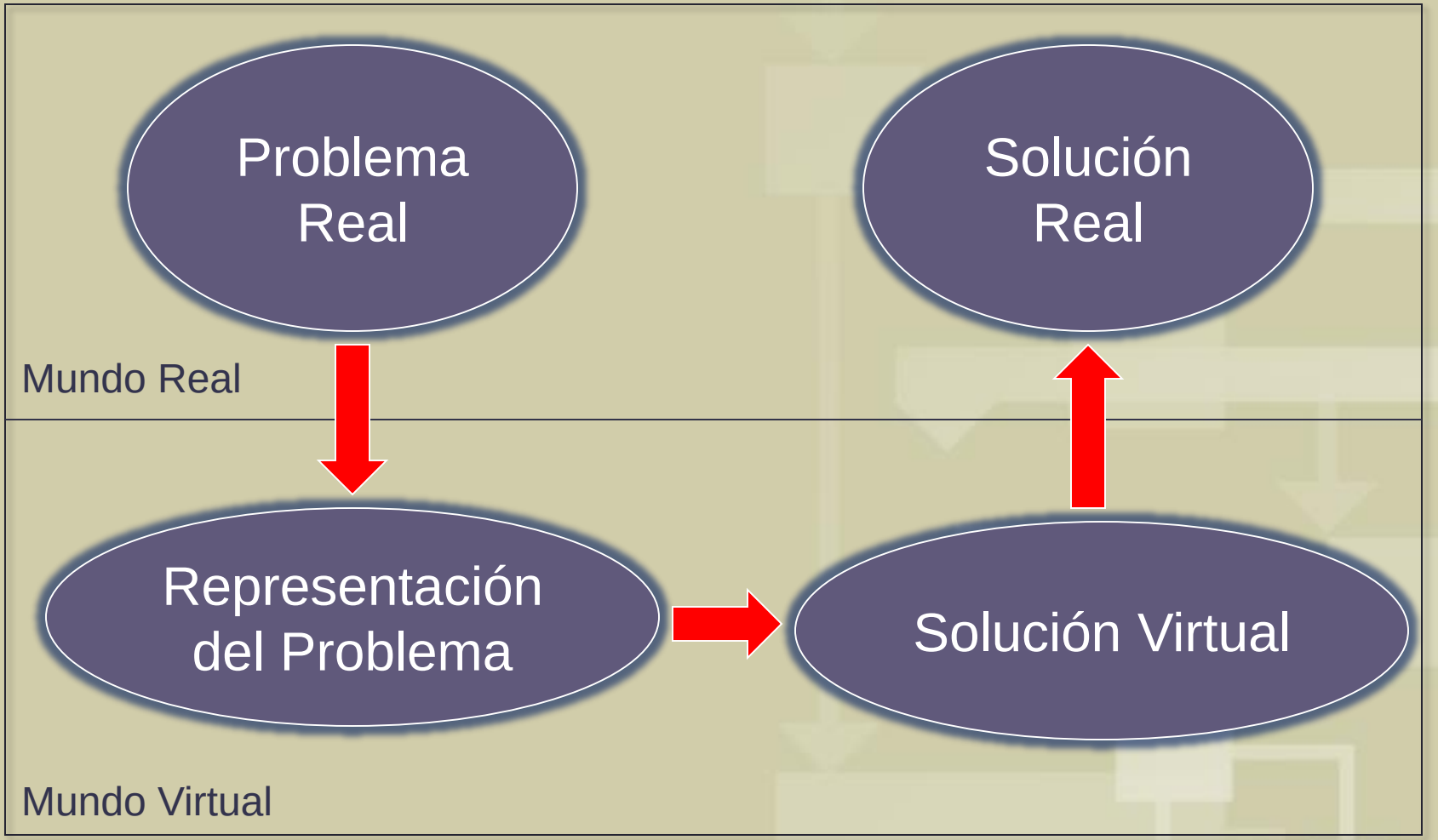
Resolución de un Problema



Resolución de un Problema



Resolución de un Problema



Resolución de un Problema

Análisis del Problema

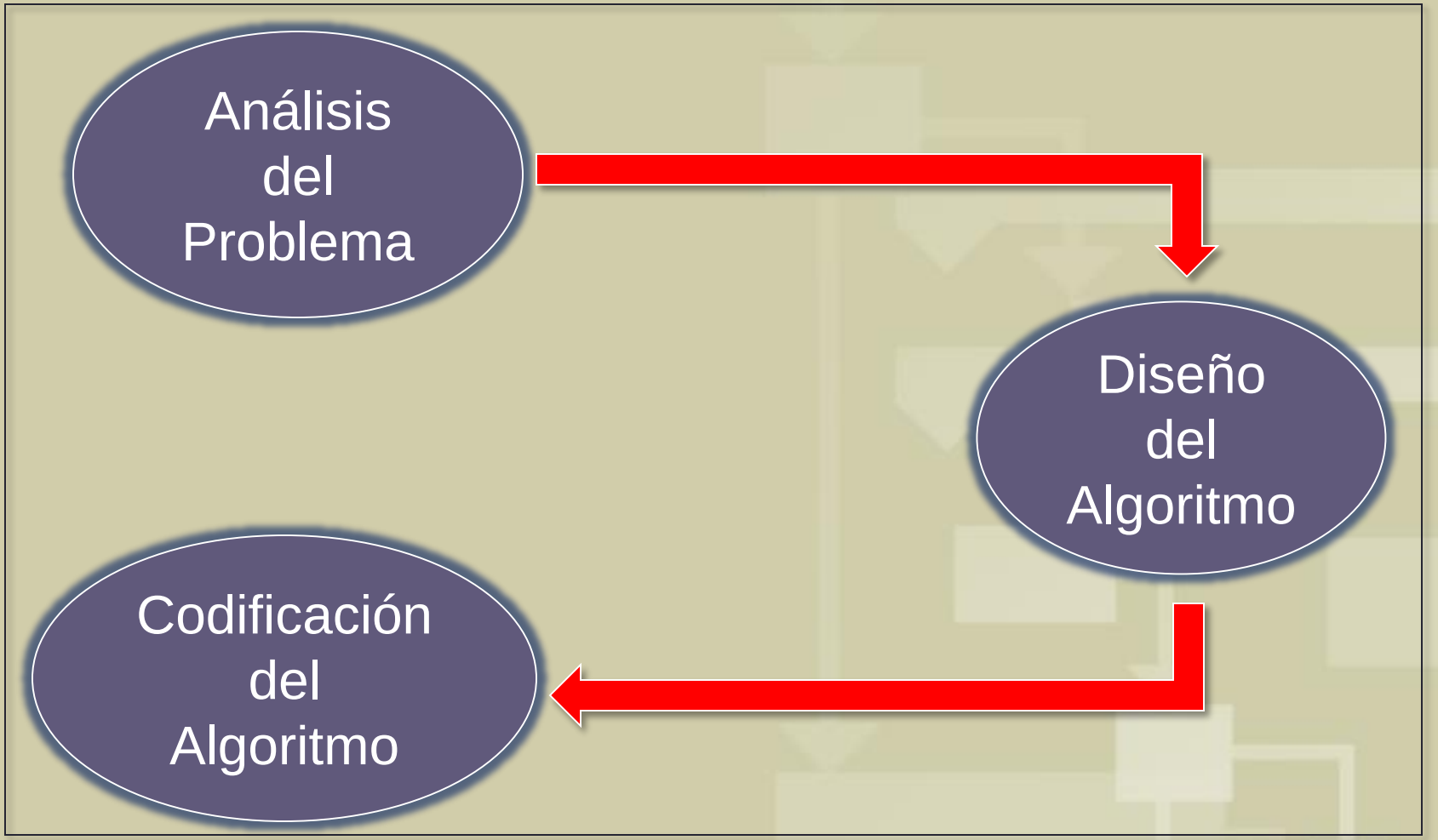
- Estudio detallado del problema.
- Especificación de los requerimientos que debe cumplir la solución.

- En ésta etapa se determina **qué** deberá hacer el programa

Resolución de un Problema

- Determinación del conjunto de acciones a ejecutar (algoritmo) para resolver el problema planteado.
- En ésta etapa se determina **cómo** el programa hará la tarea solicitada

Resolución de un Problema



Algoritmo:

Procedimiento a seguir para resolver un problema en términos de:

1) Las acciones a ejecutar

2) El orden de dichas acciones

Características:

- Preciso (se indica el orden de realización en cada paso)

- Definido (se repite el resultado)

- Finito (número determinado de pasos)

Algoritmo:

Ejemplo:

- Ingresar primer número

- Ingresar segundo número

- Ingresar tercer número

- Sumar los tres números

- Dividir el resultado obtenido en el paso anterior por 3

- Mostrar el cociente obtenido

Algoritmo:

Ejemplo:

- **Ingresar primer número**
- **Ingresar segundo número**
- **Ingresar tercer número**

Entrada

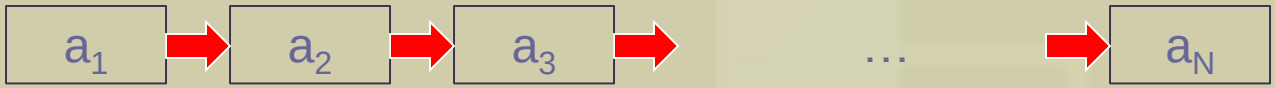
- **Sumar los tres números**
- **Dividir el resultado obtenido en el paso anterior por 3**

Procesamiento

- **Mostrar el cociente obtenido**

Salida

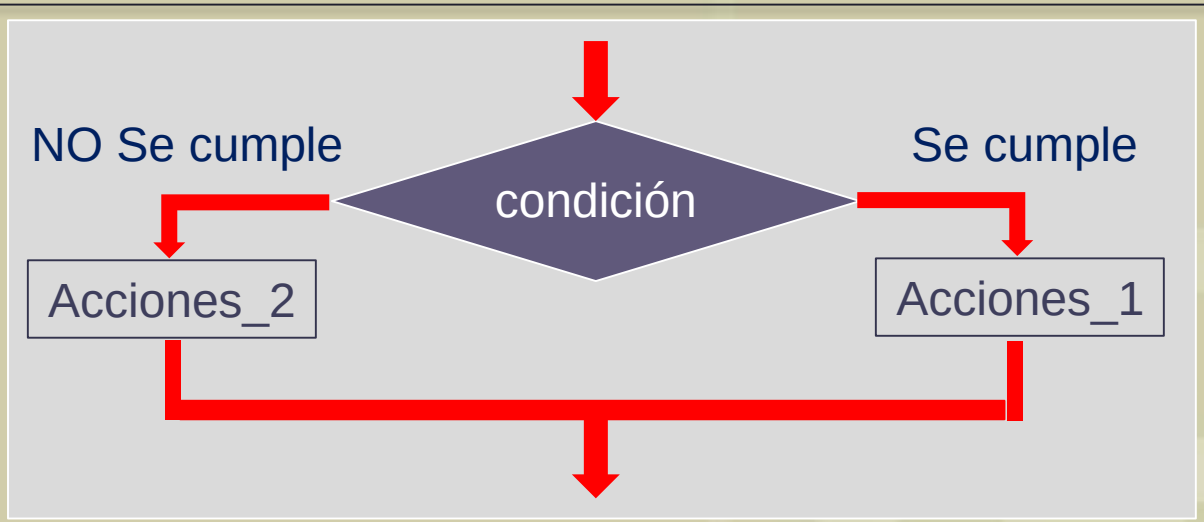
Control de flujo: Secuencial



Pseudocódigo:

```
acción_1
acción_2
...
acción_N
```

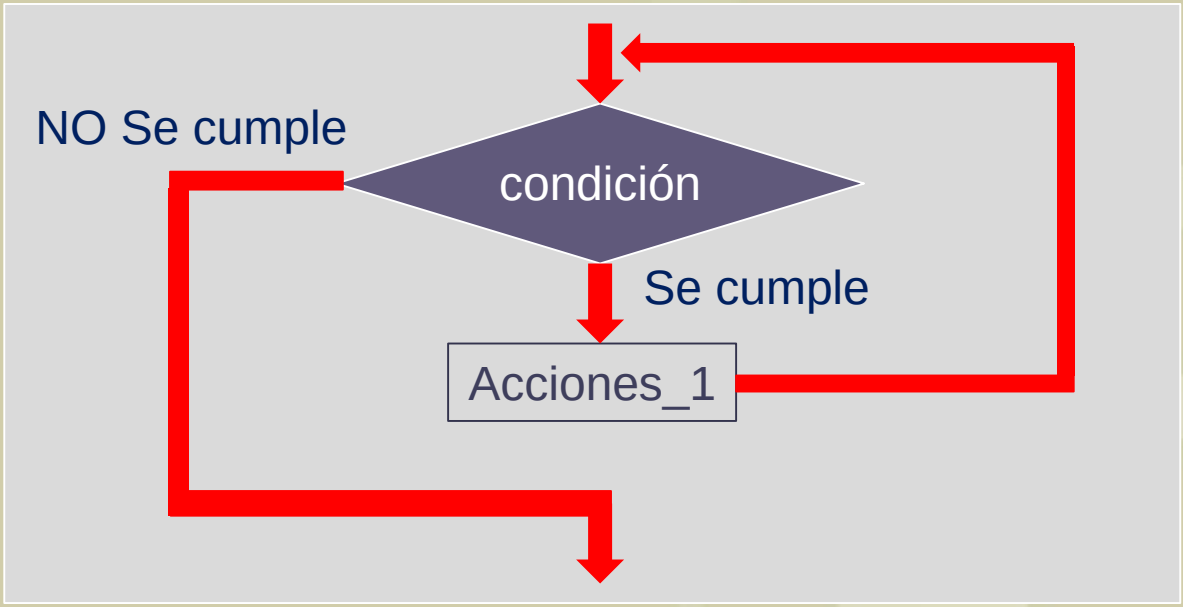
Control de flujo: Condicional



Pseudocódigo:

```
…
si condición se cumple hacer
    acciones_1
Sino hacer
    acciones_2
fin si
```

Control de flujo: Repetitivo



Pseudocódigo:

```
…
mientras condición se cumple hacer
    acciones_1
fin mientras
```

Descomposición: Funciones

Dentro del proceso de resolución de problemas, la identificación de subproblemas cumple un rol fundamental (tarea de diseño) y sienta las bases para la descomposición del problema.

La pericia de un programador no está en ser veloz para escribir líneas de programa, sino en saber descubrir, en el proceso de diseño, cuáles son las partes del problema, y luego resolver cada una de ellas abstrayéndose de las otras.

Las funciones definen la abstracción a la solución de cada subproblema individual.

Descomposición: Funciones

En programación, una función es una secuencia de instrucciones, que pueden incluir bloques de control de flujo o llamadas a otras funciones, que tiene un objetivo en particular y que se ejecuta cuando es activada desde otra función.

Resolución de un Problema

DIVIDE

Y

CONQUISTA

Funciones

Funciones como cajas negras:


```
// función que retorna verdadero si el valor de X es múltiplo de Y,
// o retorna falso en caso contrario

esMultiplo( X, Y )
    si modulo(X, Y) = 0
        retornar verdadero
    si no
        retornar falso
    fin si
fin esMultiplo
```

```
esMultiplo( X, Y )
    si modulo(X, Y) = 0
        retornar verdadero
    fin si
    retornar falso
fin esMultiplo
```

Funciones

- Una función recibe los datos del problema que tiene que resolver a través de sus argumentos/parámetros. Estos son variables y sus valores dependen de cómo fue llamado.

```
Si esMultiplo(541, 11)
   imprimir "541 es múltiplo de 11"
Sino
   imprimir "541 NO es múltiplo de 11"
Fin si
...
Si esMultiplo(5437,13)
   imprimir "5437 no es un número primo"
Fin Si
```

- No resuelven un problema específico, lo que las define como base de reutilización.
- Dentro de la secuencia de instrucciones que ejecutan pueden llamar a otras funciones, incluso, llamarse a si misma.

Funciones: Ejemplo

```
// Suma de los primeros N números naturales,
// retorna su resultado

Sumar( N )
      suma = 0
      índice = 1
      mientras índice <= N hacer
            incrementar suma en índice
            incrementar índice en 1
      fin mientras
      retornar suma
Fin Sumar
```

Inserción Ordenada

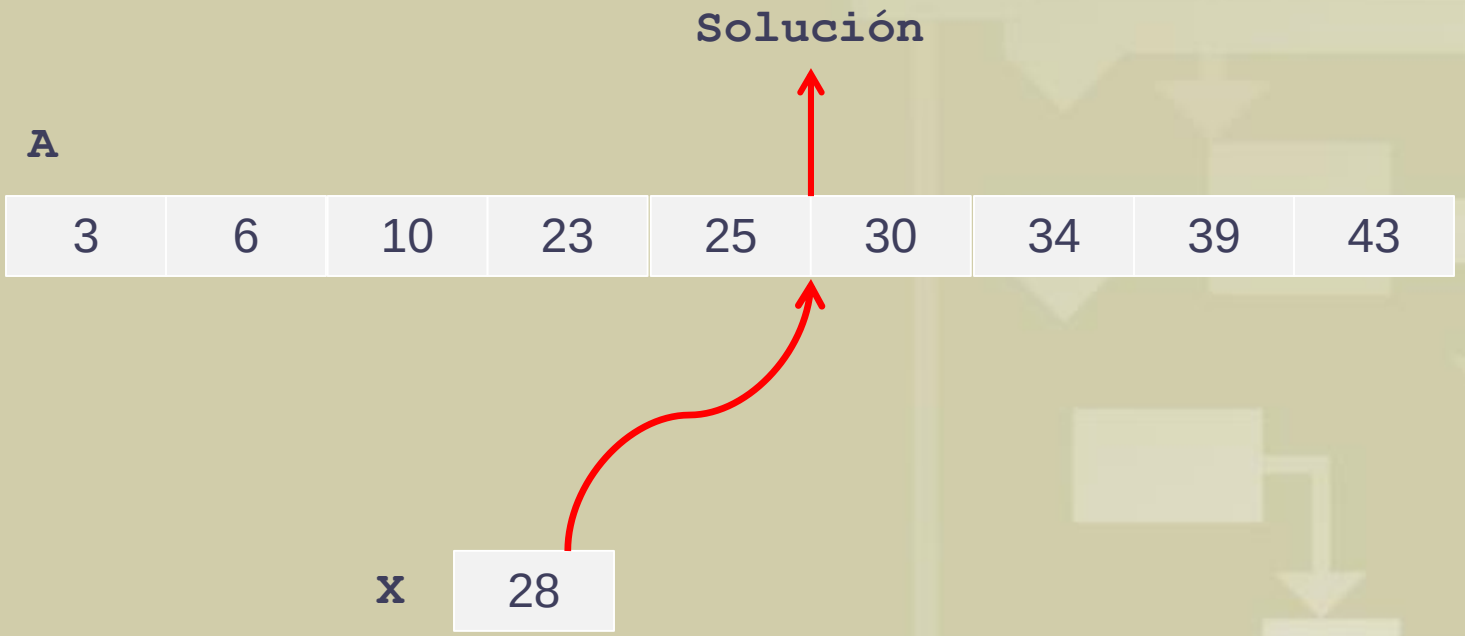
```
// función que ordena el vector A

ordenarPorInsersion(A)
  Para i desde 1 hasta tamaño(A)
    InsertarOrdenado (A(i), A, i-1)
  Fin para
Fin ordenarPorInsersion
```

Inserción Ordenada

```
// procedimiento que inserta X en
// forma ordenada dentro del conjunto
// A que tiene N elementos

InsertarOrdenado (X, A, N)
      idx = PuntoInsercion(X, A, N)
      Desplazar(A, N, idx)
      A(idx) = X
Fin InsertarOrdenado
```

Inserción Ordenada

PuntoInsercion(X, A, N)

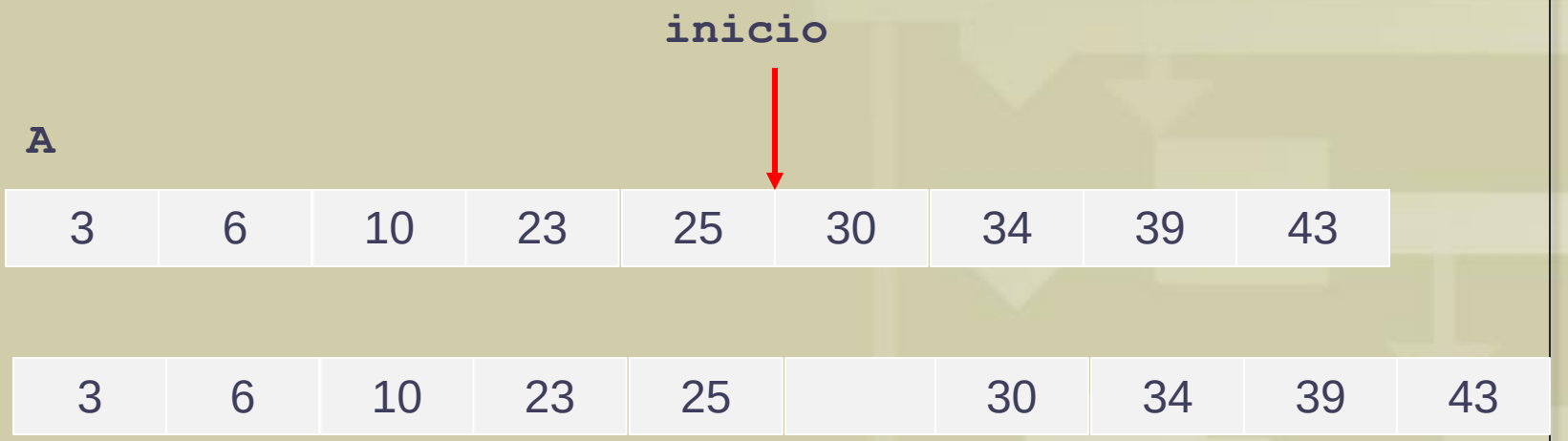


Inserción Ordenada

```
// procedimiento que busca el punto
// correcto para insertar ordenada-
// mente a X dentro del conjunto
// A que tiene N elementos ordenados

PuntoInsercion(X, A, N)
     idx = 0
     mientras idx < N Y X > A(Idx)
          idx = idx + 1
     fin mientras
     retornar idx
Fin PuntoInsercion
```

Inserción Ordenada

Desplazar(A, N, inicio)



Inserción Ordenada

```
// procedimiento que desplaza a todos
// los elementos de A, de tamaño N,
// en una posición a partir de la
// componente inicio

Desplazar(A, N, inicio)
      idx = N
      mientras idx > inicio
            A(idx) = A(idx - 1)
            idx = idx - 1
      fin mientras
Fin Desplazar
```

Ordenamiento

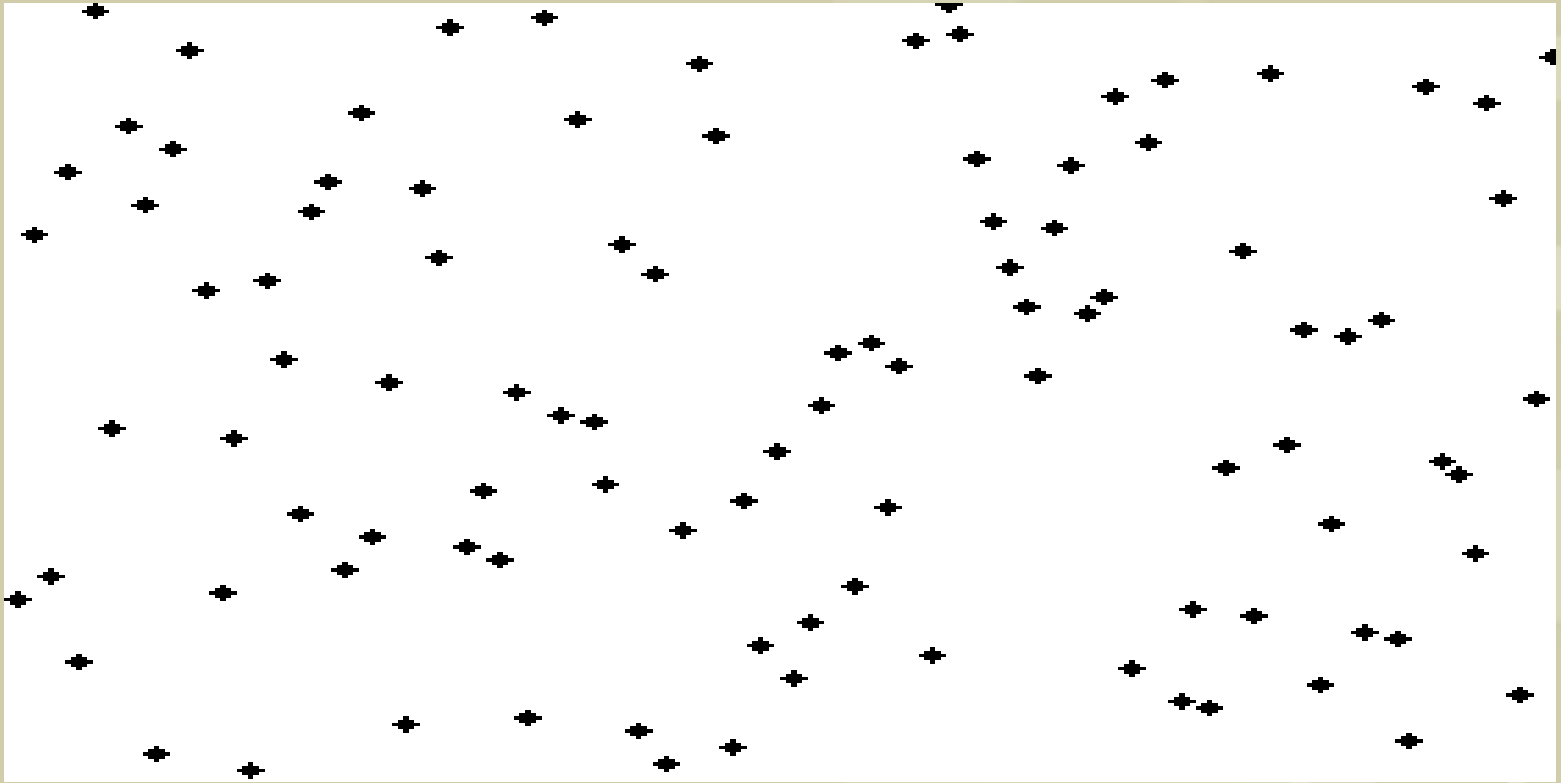
El problema consiste en dado un conjunto A, ordenar los elementos con algún criterio, por ejemplo que $A_i < A_{i+1}$

A_0	A_1	A_2	A_3						A_{N-1}

Existen un gran número de algoritmos que resuelven este problema

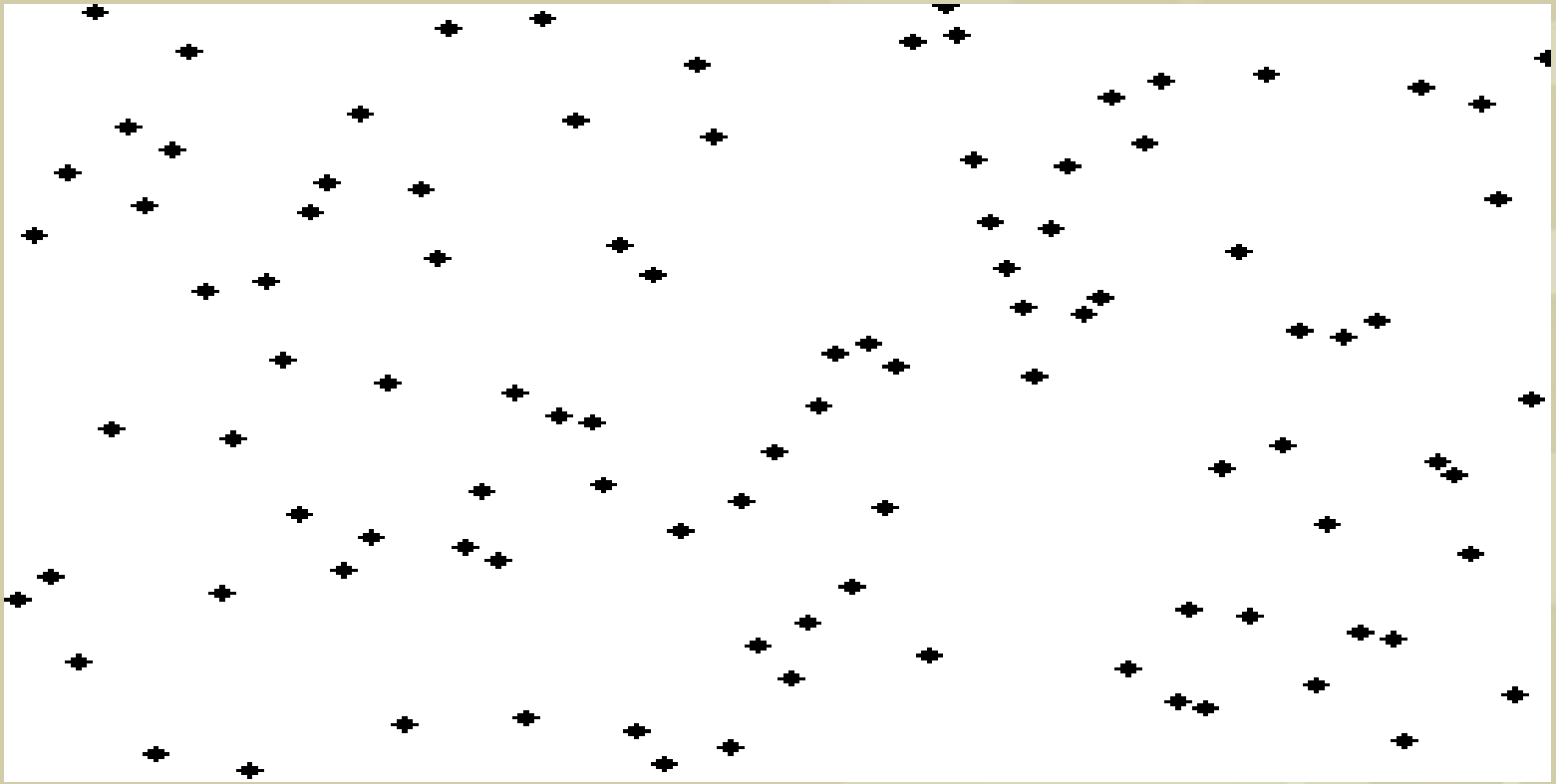
Ordenamiento

Método de Inserción



Ordenamiento

Método de la Burbuja

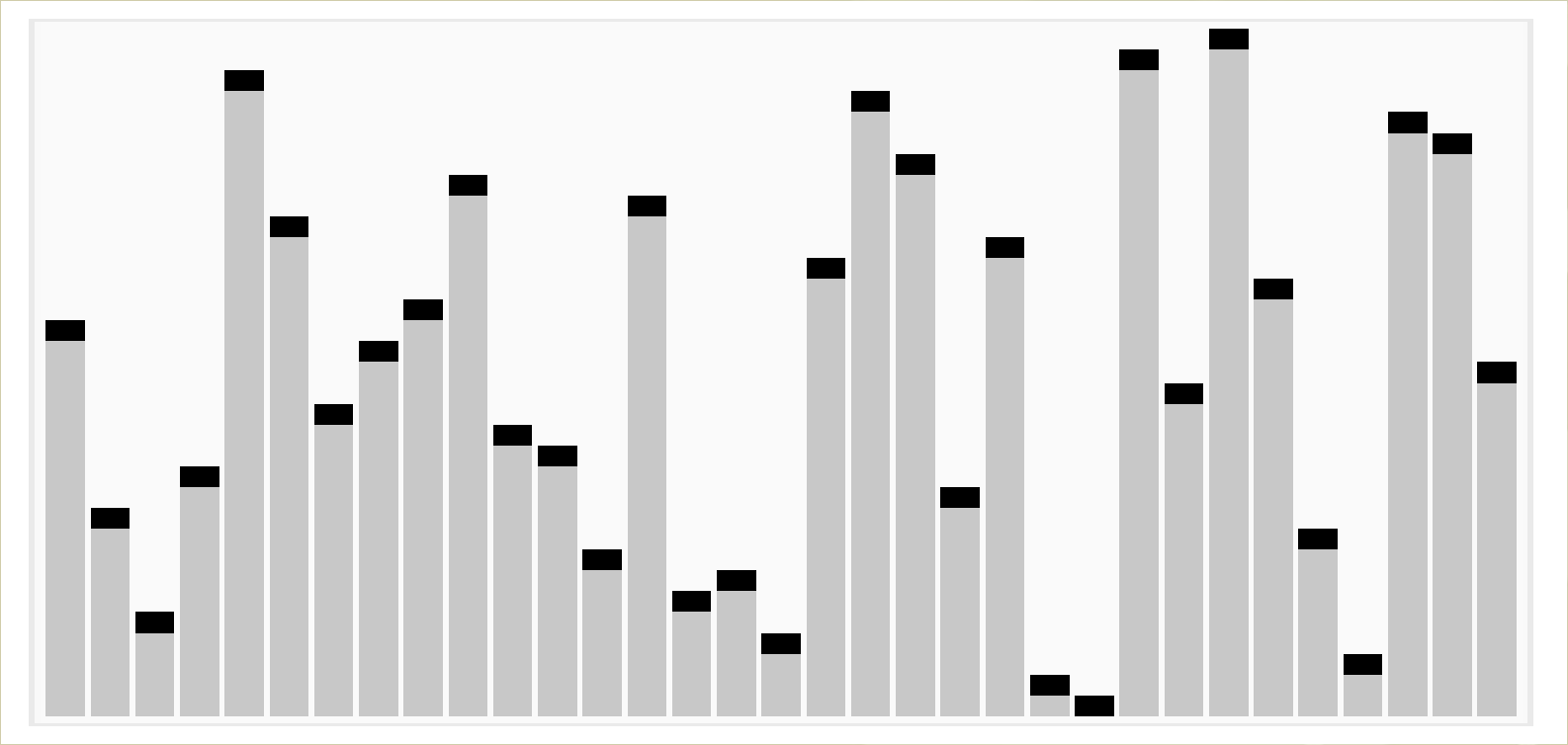


QuickSort

```
quicksort(A)
    si tamaño(A) ≤ 1
        retornar A
    fin si
    seleccionar y remover un elemento PIVOT de A
    para cada X en A
        si x ≤ PIVOT
            agregar X a MENORES
        sino
            agregar X to MAYORES
        fin si
    fin para
    retornar Unir(quicksort(MENORES), PIVOT, quicksort(MAYORES))
Fin quicksort
```

Ordenamiento

Método QuickSort



Comparación

tamaño	n2	t [seg]		n.log(n)	t [seg]	
10	100	0.1		33	0.03	
100	10000	10		664	0.66	
1000	1000000	1000		9966	9.97	
1000000	1E+12	1E+09		19931569	19,931.57	
		11574.07	dias		5.54	Horas
		31.70979	años			

BuscaPosicionMayor (p1 ej3a)

```
BuscaPosicionMayor(A())
  posMayor = 0
  para idx = 1 hasta dim(A) - 1
    si A(idx) > A(posMayor)
      posMayor = idx
    fin si
  fin para
  retornar posMayor
Fin BuscaPosicionMayor
```

BuscaMayor (p1 ej3b)

```
BuscaPosicionMayor(A())
  posMayor = 0
  para idx = 1 hasta dim(A) - 1
    si A(idx) > A(posMayor)
      posMayor = idx
    fin si
  fin para
  retornar posMayor
Fin BuscaPosicionMayor


BuscaMayor(A())
  retornar A(BuscaPosicionMayor(A))
Fin BuscaMayor
```

Función Sumar (p1 ej6a)

		9	8	3
	+		9	7
	1	0	8	0

Función Sumar (p1 ej6a)

A()		9	8	3
B()	+		9	7
C()	1	0	8	0

```
Sumar(A(), B(), C())
  SetZero(C())              // pone a 0 todos los elem.
  para i = 0 hasta mayor(dim(A),dim(B))-1
    V      = C(i) + A(i) + B(i)
    C(i)   = Resto(V, 10)
    C(i+1) = PartEntera(V/10)
  fin para
Fin Sumar
```

Función Multiplicar (p1 ej6b)

			9	8	3
		x		9	7
		6	8	8	1
	8	8	4	7	
	9	5	3	5	1

Función Multiplicar (p1 ej6b)

A()			9	8	3
B()		x		9	7
		6	8	8	1
	8	8	4	7	
C()	9	5	3	5	1

```
Mult(A(), B(), C())
  SetZero(C())
  para j = 0 hasta dim(B)-1
    para i = 0 hasta dim(A)- 1
      V        = C(i+j) + A(i) * B(j)
      C(i+j)   = Resto(V, 10)
      C(i+j+1) = C(i+j+1) + PartEntera(V/10)
    fin para       // -> próximo i
  fin para         // -> próximo j
Fin Mult
```

SetZero

```
SetZero(A())
  para idx = 0 hasta dim(A) - 1
    A(idx) = 0
  fin para
Fin BuscaPosicionMayor
```

Movimientos caballo (p1 ej9)

```
BuscaSoluciones()
    Tablero t(8,8)
    limpiarTablero(t)
    para y = 0 hasta NumFilas(t) - 1
        para x = 0 hasta NumColumnas(t) - 1
            IntentaLlenado(t, x, y, 1)
        fin para  // siguiente x
    fin para  // siguiente y
Fin BuscaSoluciones
```

Movimientos caballo (p1 ej9)

```
// intenta completar el tablero 't', poniendo la 'ficha'
// en la posición ('x','y') y tratando de hacer evolucionar
// la solución
//
IntentaLlenado( t, x, y, ficha)
    PoneFicha(t, x, y, ficha)
    si ficha == NumFilas(t) * NumColumnas(t)
        Imprimir "Encontré una solución!!"
        ImprimeTablero(t)
    sino
        c = BuscaCandidatos(t, x, y)
        para i = 0 hasta dim(c) - 1
            IntentaLlenado(t, c[i].x, c[i].y, ficha + 1)
        fin para
    fin si
    SacaFicha(t, x, y)
Fin IntentaLlenado
```

Movimientos caballo (p1 ej9)

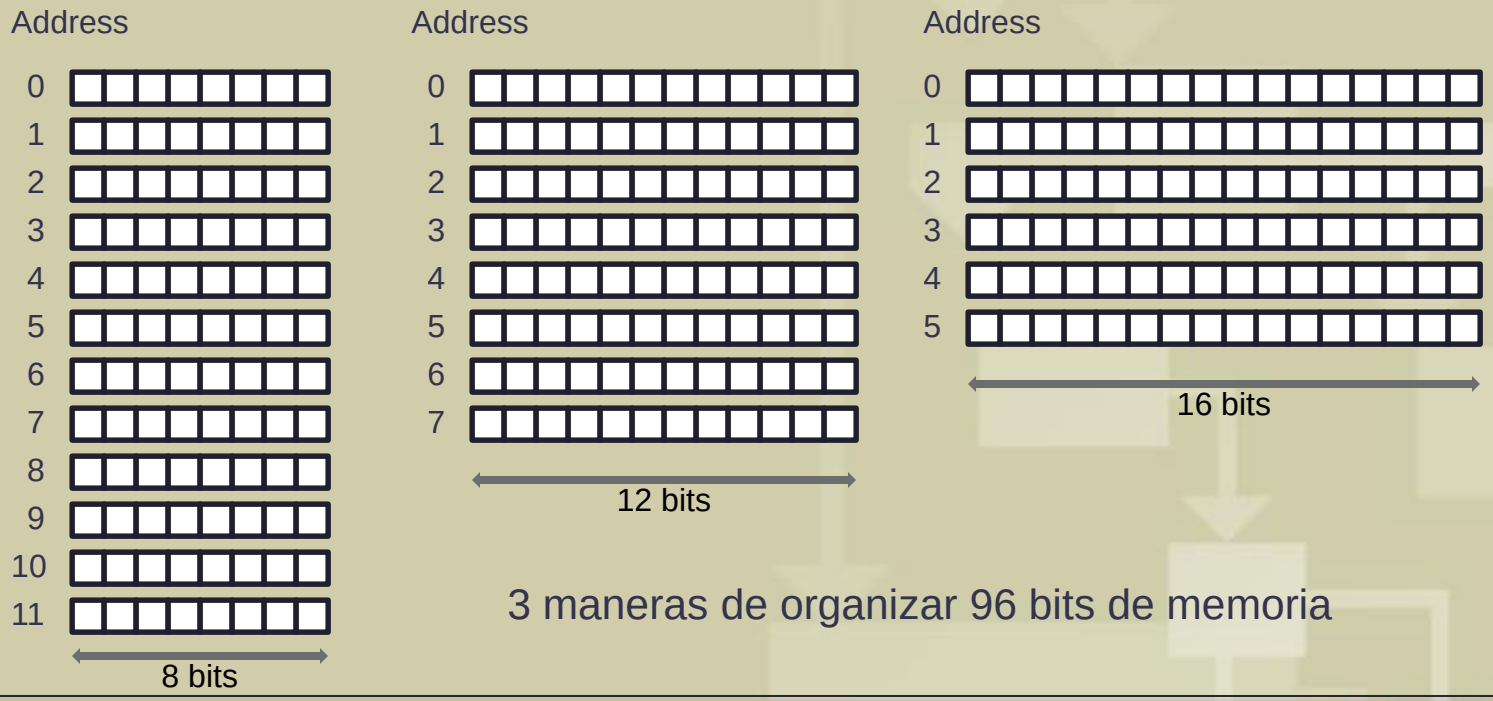
```
// Arma un arreglo de pares (X, Y), con posibles movimientos
// a partir de la posición 'x', 'y'
//
BuscaCandidatos( t, x, y)
    candidatos lc
    deltas = {{ 1,  2}, { 2,  1}, { 2, -1}, { 1, -2},
              {-1, -2}, {-2, -1}, {-2,  1}, {-1,  2} }
    para d = 0 hasta dim(deltas) - 1
        si EsPosible(t, x+deltas[d][0], y+deltas[d][1])
            AgregaPos(lc, x+deltas[d][0], y+deltas[d][1])
        fin si
    fin para
    retornar lc;
Fin BuscaCandidatos
```

Movimientos caballo (p1 ej9)

```
// función que retorna verdadero si ('x','y') cae dentro
// del tablero y además es una posición libre, retorna falso
// en caso contrario
//
EsPosible(t, x, y)
    si x >= 0 AND x < NumColumnas(t) AND
       y >= 0 AND y < NumFilas(t) AND EstaLibre(t, x, y)
        retornar true
    fin si
    retornar false
Fin EsPosible
```

Organización de la memoria

Unidad de memoria: dígito binario, **bit**
1 byte = 8 bits 1 nibble = 4 bits
1 word = 2, 4 u 8 bytes = 16, 32 ó 64 bits



3 maneras de organizar 96 bits de memoria

Números de precisión finita

Enteros positivos de 3 dígitos decimales, sin punto decimal, sin signo.

Exactamente 1000 miembros: *000*, *001*, *002* … *999*

Con estas restricciones, hay números imposibles de representar:

- Número mayores a *999*
- Números negativos
- Fracciones
- Números irracionales
- Números complejos

Números de precisión finita

Se pierde la propiedad aritmética de cierre con respecto a la suma, resta y multiplicación:

600 + 600 = 1200 (muy grande)
003 – 005 = -2 (negativo)
050 x 050 = 2500 (muy grande)
007 / 002 = 3.5 (no es entero)

Violaciones de 2 clases mutuamente exclusivas:

- Operaciones cuyo resultado es mayor que el número más grande (error de overflow) o menor que el más chico (error de underflow).
- Operaciones cuyo resultado no es ni muy grande ni muy chico, si no que simplemente no es un número que pertenece al conjunto de los números representables

Números de precisión finita

Ley asociativa:

$a + (b - c) = (a + b) - c$

Para $a = 700$, $b = 400$ y $c = 300$, el primer término da 800 y el segundo da overflow al evaluar $(a + b)$.
El orden de las operaciones es importante.

Ley distributiva:

$a \times (b - c) = a \times b - a \times c$

Para $a = 5$, $b = 210$ y $c = 195$, el primer término da 75 y el segundo da overflow al evaluar $a \times b$.

Parecería por estos ejemplos que las computadoras son inapropiadas para hacer cálculos aritméticos debido a su naturaleza finita. Esta conclusión es obviamente falsa, pero los ejemplos muestran la importancia de entender como funcionan las computadoras y sus limitaciones.

Sistemas de numeración

Número decimal, en base 10:

2009.14

Centenas	Decenas	Unidades	Punto decimal	Décimas	Centésimas
↓	↓	↓		↓	↓

$d_n \quad \ldots \quad d_2 \quad d_1 \quad d_0 \quad . \quad d_{-1} \quad d_{-2} \quad \ldots \quad d_{-k}$

$$\text{Número} = \sum_{i=-k}^{n} d_i \, 10^i$$

$2009.14 = 2 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 9 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2}$

Sistemas de numeración

Utilizando computadoras, es frecuente utilizar otras bases distintas a 10. Las más utilizadas son 2, 8 y 16. Estos sistemas de numeración son llamados binario, octal y hexadecimal.

Para un sistema de base *k*, se requieren *k* símbolos diferentes para representar los dígitos de *0* a *k-1*.

Sistema decimal:	*0 1 2 3 4 5 6 7 8 9*
Sistema binario:	*0 1*
Sistema hexadecimal:	*0 1 2 3 4 5 6 7 8 9 A B C D E F*

$2009_{10} = 11111011001_2 = 3731_8 = 7D9_{16}$

$11111011001_2 = 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 2^3 + 2^0$

$7D9_{16} = 7 \times 16^2 + D \times 16^1 + 9 \times 16^0 = 7 \times 256 + 13 \times 16 + 9$

Números binarios negativos

- Magnitud con signo: el bit de más a la izquierda se utiliza como signo, 0 es positivo y 1 es negativo, el resto de los bits representan el valor absoluto del número.

- Complemento a 1: el bit de más a la izquierda es el signo, 0 es positivo y 1 es negativo, para negar un número se reemplaza cada 1 por 0 y cada 0 por 1.

- Complemento a 2: ídem con el bit de signo, para negar un número se reemplaza cada 1 por 0 y cada 0 por 1 (como en complemento a 1) y luego se le suma 1 al resultado.

- Excess 2^{m-1} (offset binary): para un número con *m* bits, el número se representa como la suma del mismo con 2^{m-1} . Para un número de 8 bits, se le suma *128*, ejemplo $-3 \rightarrow 128 + (-3) = 125$.

Números binarios negativos

00000110	+6
11111001	-6 en complemento a 1
11111010	-6 en complemento a 2
01111010	-6 en excess 128

00010001	+17
11101110	-17 en complemento a 1
11101111	-17 en complemento a 2
01101111	-17 en excess 128

01111111	+127
10000000	-127 en complemento a 1
10000001	-127 en complemento a 2
00000001	-127 en excess 128

No existe	+128
No existe	-128 en complemento a 1
10000000	-128 en complemento a 2
00000000	-128 en excess 128

Problemas: o dos representaciones distintas para 0 o diferentes cantidades de números positivos y negativos.

Aritmética binaria

- Suma de bits:

	0	0	1	1
	+ 0	+ 1	+ 0	+ 1
Suma	0	1	1	0
Carry	0	0	0	1

- Suma de números binarios:

Se empieza la suma con los bits de más a la derecha y se va llevando el bit de carry, como con los números decimales. Si se usa representación de complemento a 1, el carry del bit de más a la izquierda se vuelve a sumar al resultado, en complemento a 2 se descarta.

Decimal		Compl. 1		Compl. 2
10		00001010		00001010
+ (-3)		+ 11111100		+ 11111101
+7	1	00000110	1	00000111
		+ 1		
		00000111		Se descarta

Operaciones entre bits

- OR – "|" :

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

- XOR – "^" :

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

- AND – "&" :

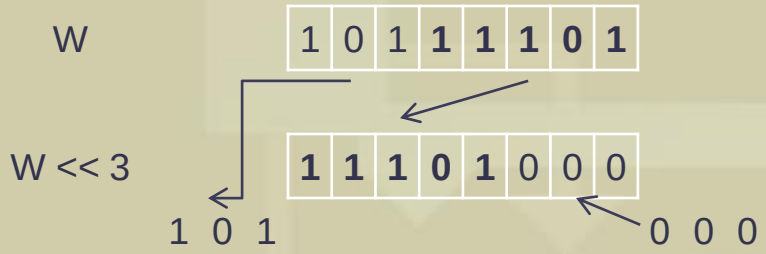
a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

- NOT – "~"

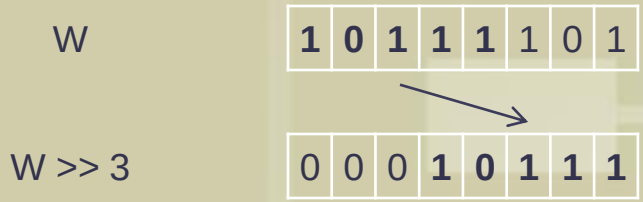
a	~a
0	1
1	0

Operaciones entre bits

- Left shift – “<<” :



- Right shift – “>>” :



Números de punto flotante

$$n = f \times 10^{e}$$

f es la fracción o mantisa
e es el exponente

3.14	= 3.14	x 10^{0}	= 0.314	x 10^{1}
0.000001	= 1.0	x 10^{-6}	= 0.1	x 10^{-5}
2009	= 2.009	x 10^{3}	= 0.2009	x 10^{4}

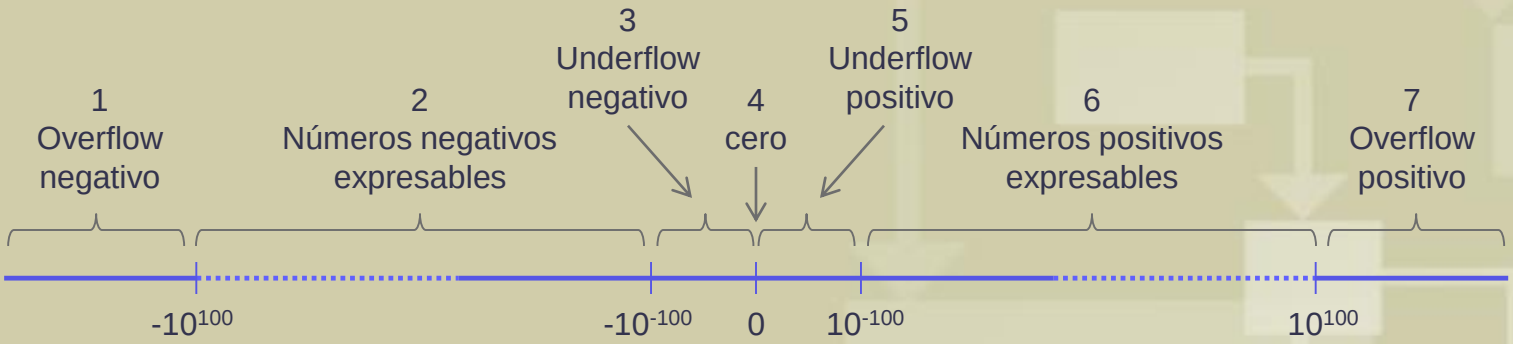
Normalizado: $f = 0$ ó $0.1 \leq |f| < 1$

El *rango* está determinado por la cantidad de dígitos del exponente.
La *precisión* está determinada por la cantidad de dígitos en la fracción.

Números de punto flotante

Representación con 3 dígitos decimales con signo para la fracción y 2 dígitos con signo para el exponente: de 0.100×10^{-99} a $0.999 \times 10^{+99}$.

1. Números negativos grandes, menores a -0.999×10^{99}
2. Números negativos entre a -0.999×10^{99} y -0.100×10^{-99}
3. Números negativos chicos, mayores a -0.100×10^{-99}
4. Cero
5. Números positivos chicos, menores a -0.100×10^{-99}
6. Números positivos entre a $+0.100 \times 10^{-99}$ y $+0.999 \times 10^{99}$
7. Números positivos grandes, mayores a $+0.999 \times 10^{99}$



Error de redondeo

Las operaciones con números de punto flotante pueden llevar a resultados que no se pueden expresar exactamente. El resultado es llevado al número representable más cercano.

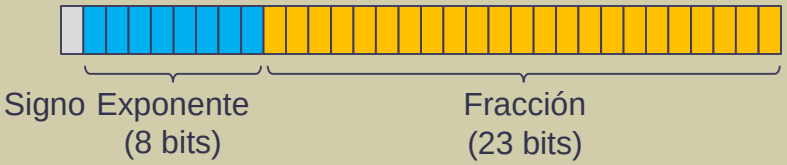
Aparece error de redondeo. Ej: $0.100 \times 10^3 / 3 \rightarrow 0.333 \times 10^2$

- El espacio entre números expresables adyacentes no es constante.
Ej: $0.999 \times 10^{99} - 0.998 \times 10^{99}$ vs. $0.999 \times 10^{-10} - 0.998 \times 10^{-10}$

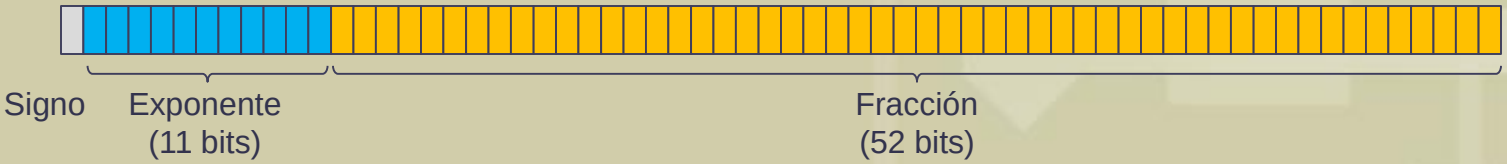
- El error relativo es prácticamente el mismo.
Ej: $0.999 \times 10^{99} / 0.998 \times 10^{99}$ vs. $0.999 \times 10^{-10} / 0.998 \times 10^{-10}$

IEEE 754

Simple precisión (32 bits) 1 bit signo, 8 bits exponente y 24 bits de fracción



Doble precisión (64 bits) 1 bit signo, 11 bits exponente y 53 bits de fracción



En la fracción se omite el primer bit que siempre es 1.
El exponente está codificado con un offset (excess) igual a $(2^{e-1})-1$.

$$Número = (-1)^{Signo} \times 2^{Exponente-Offset} \times (1 + Fracción)$$

IEEE 754

Normalizado	±	$0 < Exp < (2^{e-1})-1$	Patrón de bits
Desnormalizado	±	0	Patrón de bits distinto de cero
Cero	±	0	0
Infinito	±	$(2^{e-1})-1$	0
Not a Number	±	$(2^{e-1})-1$	Patrón de bits distinto de cero

	Precisión Simple	Precisión Doble
Bits totales	32	64
Bits de signo	1	1
Bits de exponente	8	11
Bits en la fracción	23	52
Sistema del exponente	Exceso 127	Exceso 1023
Rango exponente	-126 a 127	-1022 a 1023
Número más chico	2^{-126}	2^{-1022}
Número más grande	$\approx 2^{128}$	$\approx 2^{1024}$
Rango decimal	$\approx 10^{-38}$ a 10^{38}	$\approx 10^{-308}$ a 10^{308}
Precisión decimal	≈ 7 dígitos decimales	≈ 16 dígitos decimales

Codificación IEEE 754

Codificación en IEEE 754 simple precisión del número decimal −118.625.

- Como el número es negativo, el bit de signo es 1.
- 118.625 es en binario 1110110.101. El 101 después del punto representa $0.625 = 2^{-1} + 2^{-3}$.
- El número binario normalizado queda 1.110110101×2^{6}.
- El primer bit de un número binario normalizado es siempre 1 y no se codifica.
- La mantisa queda 11011010100000000000000, completando a 23 bits.
- El exponente es 6, codificandolo en excess 127 queda 6 + 127 = 133.
- 133 en binario es 10000101.

$C2ED4000_{16}$

Links

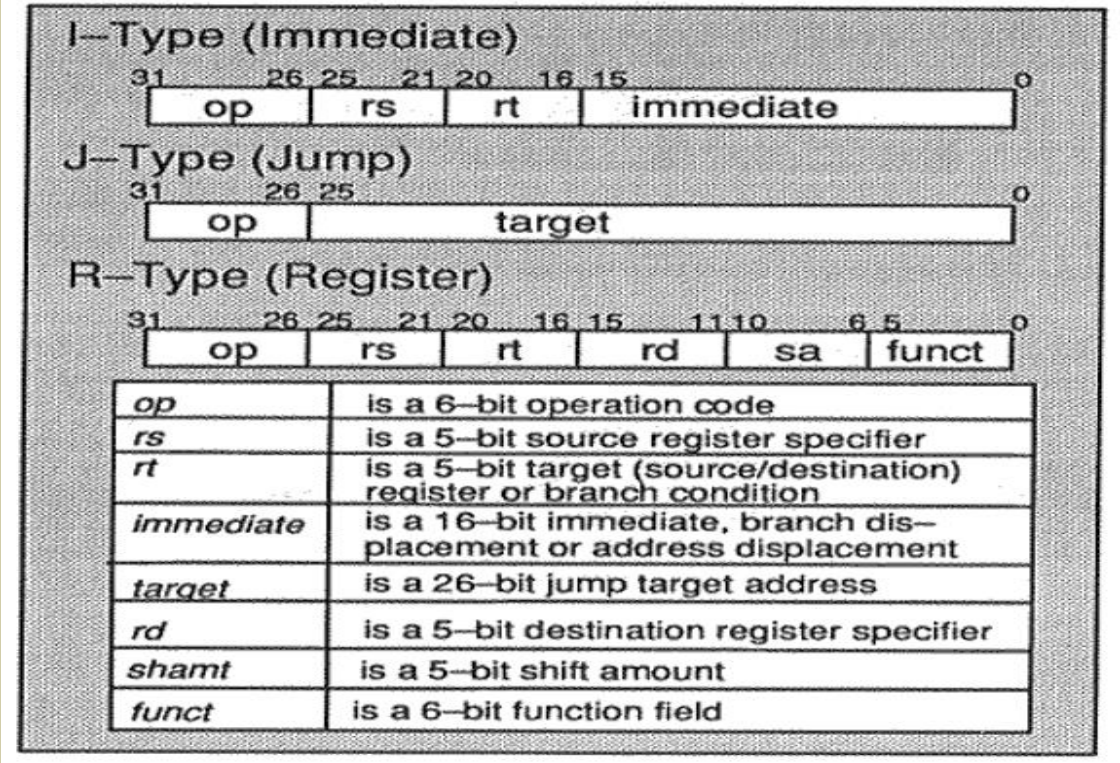
- http://en.wikipedia.org/wiki/Computer_architecture
- http://en.wikipedia.org/wiki/Computer_numbering_format
- http://en.wikipedia.org/wiki/Floating_point
- http://en.wikipedia.org/wiki/Kahan_summation_algorithm
- http://en.wikipedia.org/wiki/IEEE_754-1985
- http://en.wikipedia.org/wiki/Single_precision_floating-point_format
- http://en.wikipedia.org/wiki/Double_precision_floating-point_format

Programación

- Computadoras como máquinas tontas: solo hacen lo que se les pide que hagan.
- Realizan operaciones elementales: por ejemplo sumar 2 números, chequear si un número es igual a 0, operaciones lógicas entre bits, etc.
- **Set de instrucciones**: Operaciones básicas que puede realizar un procesador.
- Para resolver un problema utilizando una computadora, se debe expresar su solución en términos de las instrucciones de un procesador en particular.

Nivel de abstracción

- Codificación de instrucciones (CPU-dependiente)



op	is a 6–bit operation code
rs	is a 5–bit source register specifier
rt	is a 5–bit target (source/destination) register or branch condition
immediate	is a 16–bit immediate, branch dis–placement or address displacement
target	is a 26–bit jump target address
rd	is a 5–bit destination register specifier
shamt	is a 5–bit shift amount
funct	is a 6–bit function field

Nivel de abstracción

- Programación en términos de números.

```
<ToMemMgr::portMRAlloc(unsigned char)>:
0xe1a03000
0xe20110ff
0xe3a02d4f
0xe282203c
0xe0801001
0xe7d10002
0xe080c080
0xe083310c
0xe5933804
0xe7c13002
0xe1a0f00e
```

Nivel de abstracción

- Uso de un ***assembly language***: Se mantiene una correspondencia 1 a 1 entre las sentencias del lenguaje y el set de instrucciones. Bajo nivel y no portable.

```
<ToMemMgr::portMRAlloc(unsigned char)>:
0xe1a03000      mov     r3, r0
0xe20110ff      and     r1, r1, #255
0xe3a02d4f      mov     r2, #5056
0xe282203c      add     r2, r2, #60
0xe0801001      add     r1, r0, r1
0xe7d10002      ldrb    r0, [r1, r2]
0xe080c080      add     ip, r0, r0, lsl #1
0xe083310c      add     r3, r3, ip, lsl #2
0xe5933804      ldr     r3, [r3, #2052]
0xe7c13002      strb    r3, [r1, r2]
0xe1a0f00e      mov     pc, lr
```

Nivel de abstracción

- Lenguajes de alto nivel. Permiten la manipulación de las entidades del problema sin involucrarse con los detalles de su implementación interna. (**FORTRAN**, **Pascal**, **C**, **Basic**, **Java**, **C++, Objective C**, **C#,** etc. , etc. , etc.). Portabilidad.

```
Port_t ToMemMgr::portMRAlloc (u8 szLog)
{
    Port_t ret = rdFree[szLog];
    rdFree[szLog] = MRDescTab[ret].refStrAddr;

    return ret;
}
```

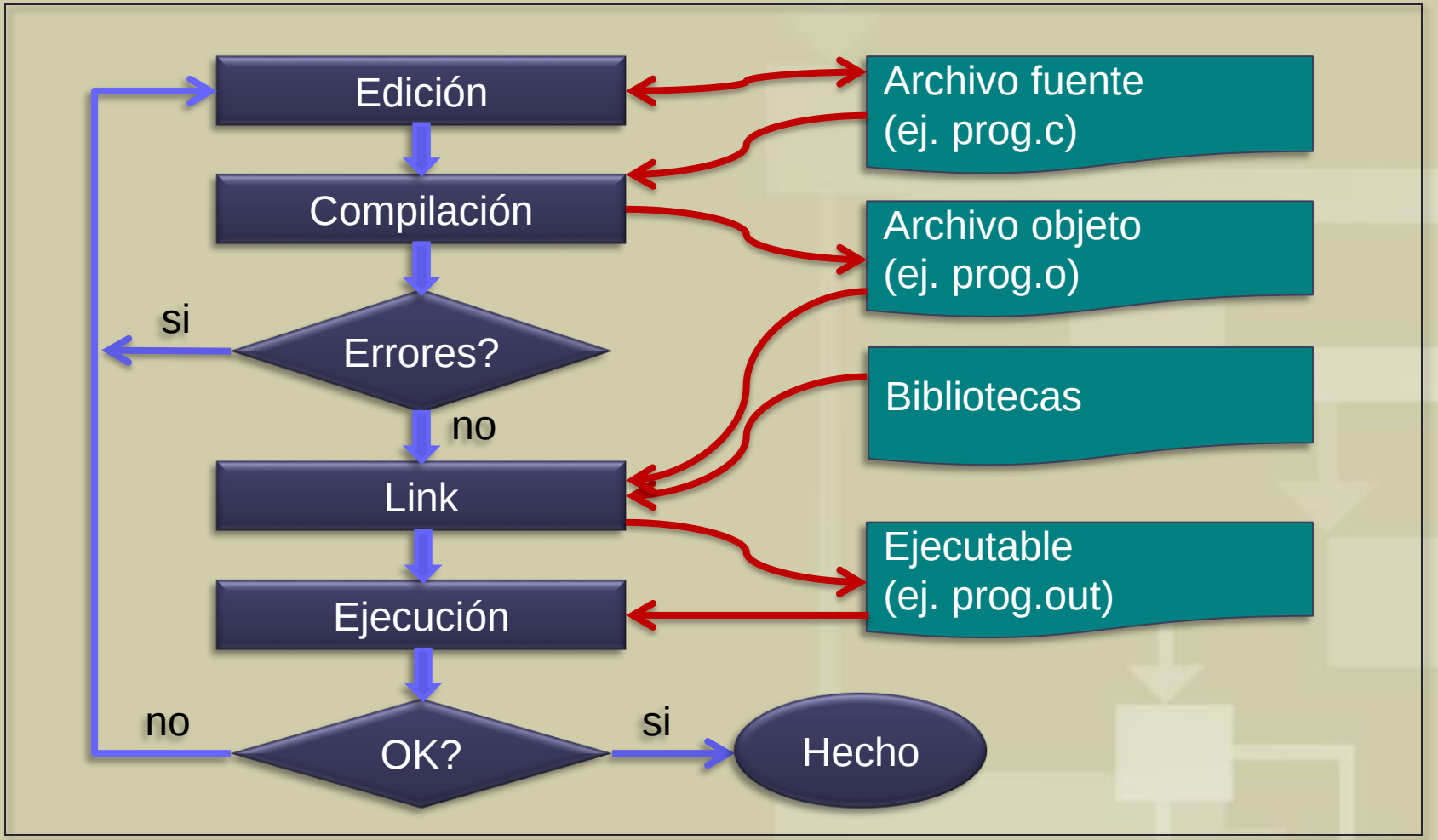
Paradigmas de programación

- **Lenguajes imperativos**: se debe especificar en forma detallada el flujo del programa.
- **Lenguajes declarativos**: Se pone énfasis en la definición del problema. La solución descansa en mecanismos propios del lenguaje.
- **Orientados a objetos**: Soporte de mecanismos de representación de entidades y sus interrelaciones.

Lenguaje C

- De propósito general. Imperativo. Estructurado.
- De alto nivel, pero tiene características que le permiten al programador acercarse al hardware y hacer operaciones a bajo nivel.
- Riqueza de operadores y tipos de datos.
- Economía de expresión.
- Eficiencia como uno de los objetivos de diseño.
- Fácil de aprender ;-)

Generación de un programa



Estructura de un programa en C

- Comentarios.
- Directivas al preprocesador.
- Definición de tipos de datos.
- Declaración de variables.
- Definición de funciones.

hola.c

```
/*
 Primer programa en C
*/
#include <stdio.h>

int main(void)   // función que toma control
{
    printf("hola mundo!\n");
    return 0;
}
```

Proceso de compilación

- Depende de la plataforma y entorno de desarrollo utilizado.
- Para ambientes Linux/Cygwin(Windows), la compilación y ligado (link) se puede realizar de una manera unificada desde una consola de comandos:

```
$ gcc hola.c –o hola
```

- Esto genera (si no existieron errores de compile/link) un archivo ejecutable de nombre ‘hola’ (hola.exe en plataforma Windows). La ejecución se realiza con:

```
$ ./hola
hola mundo!
```

Variables

- En lenguajes de bajo nivel de abstracción, la representación de datos y su manipulación involucran la interacción directa con el hardware (sumamente tedioso).
- Los lenguajes de alto nivel implementan una abstracción que facilita estas tareas: **la variable**.

“Relación entre un nombre simbólico y una porción de memoria.”

Variables

- Los nombres de variables en C deben comenzar con una letra o el carácter '_' seguida por cualquier combinación de letras (mayúsculas o minúsculas), '_' o dígitos (0-9). No se pueden utilizar palabras reservadas.

Nombres válidos	Nombres inválidos
sumValue	sum$value
is_even	is even
_sysFlag	3Flag
J5x7	int

Tipos básicos y constantes

- C soporta los siguientes 5 tipos básicos:

```
int
char
_Bool
float
double
```

- La cantidad de memoria asociada a una variable de un tipo dado depende del compilador/plataforma (no lo define el lenguaje).
- Constante: Cualquier número, carácter o cadena de caracteres (string)

```
58          -> constante entera en base decimal
0x1F        -> constante entera en base hexadecimal (31)
0664        -> constante entera en base octal (436)

'q'         -> constante caracter
"Hola"      -> constante string

8.4e3       -> constante double
3.1415f     -> constante float
```

Uso de tipos y constantes

```
#include <stdio.h>

int main (void)
{
  int    intVar = 100;
  float  floatVar = 331.79;
  double doubleVar = 8.44e+11;
  char   charVar = 'W';
  _Bool  boolVar = 0;

  printf ("intVar(%d bytes)    = %d\n", sizeof(intVar),    intVar);
  printf ("floatVar(%d bytes)  = %f\n", sizeof(floatVar),  floatVar);
  printf ("doubleVar(%d bytes) = %e\n", sizeof(doubleVar), doubleVar);
  printf ("doubleVar(%d bytes) = %g\n", sizeof(doubleVar), doubleVar);
  printf ("charVar(%d bytes)   = %c\n", sizeof(charVar),   charVar);
  printf ("boolVar(%d bytes)   = %d\n", sizeof(boolVar),   boolVar);

  return 0;
}
```

Uso de tipos y constantes

```
$ gcc tipos.c -o tipos
$
$ ./tipos
intVar(4 bytes)     = 100
floatVar(4 bytes)   = 331.790009
doubleVar(8 bytes) = 8.440000e+11
doubleVar(8 bytes) = 8.44e+11
charVar(1 bytes)    = W
boolVar(1 bytes)    = 0
$
```

Especificadores de tipo

- Los especificadores: **`signed`**, **`unsigned`**, **`short`**, **`long`**, **`long long`** pueden utilizarze para cambiar la representación de un tipo de variable.
- La implementación depende del compilador.
- Se utilizan anteponiendo el especificador al tipo de datos:

```
short int X;
```

- Cuando se utiliza un especificador sin tipo, se presume que es sobre el tipo int:

```
short X;                    // short int X
unsigned U;                 // unsigned int U
unsigned long ul;           // unsigned long int ul
```

Especificadores de tipo

Tipo	Constantes	Printf format	bytes
char	'a', '\n'	%c	1
_Bool	0,1	%d, %i, %u	1
short int		%hd,%hi,%hx,%ho	2
int	12, -97, 0xFFE0, 0177	%d, %i, %x, %o	4
unsigned int	12u, 100U, 0XFFu	%u, %x, %o	4
long int	12L, -2001, 0xffffL	%ld,%li,%lx,%lo	4
unsigned long int	12UL, 100ul, 0xffeeUL	%lu, %lx, %lo	4
long long int	0xe5e5e5e5LL, 5001l	%lld,%lli,%llx,%llo	8
unsigned long long int	12ull, 0xffeeULL	%llu, %llx, %llo	8
float	12.34f, 3.1e-5f	%f, %e, %g	4
double	12.34, 3.1e-5	%lf, %e, %g	8
long double	12.341, 3.1e-5l	%Lf, %Le, %Lg	12

Operadores

- Algebráicos

  + - * / % =

  += -= *= /= %=

  ++ --

- Relacionales

  > < >= <= == !=

- Lógicos

  && || !

- De bits

  & | ^ >> << ~

  &= |= ^= >>= <<=

Ejemplos

ejem1.c

```
#include <stdio.h>
int main (void)
{
    int a = 100;
    int b = 2;
    int c = 25, d = 4, result;

    printf ("a: %d\tb: %d\tc: %d\td: %d\n", a, b, c, d);
    result = a - b;                          // resta
    printf ("a - b = %i\n", result);
    result = b * c;                          // multiplicacion
    printf ("b * c = %i\n", result);
    printf ("a / c = %i\n", a / c);          // division
    printf ("a + b * c = %i\n",  a + b * c); // precedencia
    printf ("a * b + c * d = %i\n", a * b + c * d);
    printf ("c / d = %d\n", c / d);          // division **

    return 0;
}
```

Ejemplos

```
$ gcc ejem1.c -o ejem1
$
$ ./ejem1
a: 100  b: 2    c: 25   d: 4
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
c / d = 6
$
```

Ejemplos

ejem2.c

```
#include <stdio.h>
int main (void)
{
    float f1 = 123.125, f2;
    int i1, i2 = -150;

    i1 = f1; // floating to integer conversion
    printf ("%f assigned to an int produces %i\n", f1, i1);
    f1 = i2; // integer to floating conversion
    printf ("%i assigned to a float produces %f\n", i2, f1);
    f1 = i2 / 100; // integer divided by integer
    printf ("%i divided by 100 produces %f\n", i2, f1);
    f2 = i2 / 100.0; // integer divided by a float
    printf ("%i divided by 100.0 produces %f\n", i2, f2);
    f2 = (float) i2 / 100; // type cast operator
    printf ("(float) %i divided by 100 produces %f\n", i2, f2);

    return 0;
}
```

Ejemplos

```
$ gcc ejem2.c -o ejem2
$
$ ./ejem2
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
$
```

Ejemplos

ejem3.c

```
/* Programa que duplica el valor ingresado  */
/* y lo imprime en pantalla                 */
#include <stdio.h>

int main()
{
    int valor,
        valorX2;

    printf("Ingrese el valor a duplicar:");
    scanf("%d", &valor);
    valorX2 = valor * 2;
    printf("%d x 2 = %d\n", valor, valorX2);

    return 0;
}
```

Ejemplos

```
$ gcc ejem3.c –o ejem3
$
$ ./ejem3
Ingrese el valor a duplicar:35
35 x 2 = 70
$
```

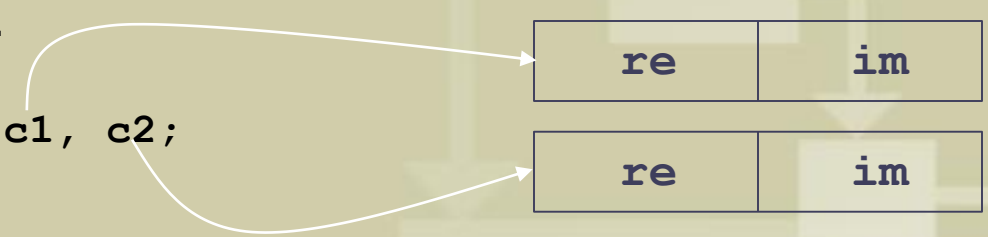
Tipos definidos por el usuario

- C permite que el usuario (programador) defina sus propios tipos de datos. Esto se realiza a través de la definición de estructuras:

```
struct Complejo {
    double re;
    double im;
};

typedef struct Complejo Complejo_t;
```

- Cuando se declara una variable del tipo **`Complejo_t`** (o **`struct Complejo`**) el compilador reserva memoria suficiente para almacenar todos sus atributos.

```
Complejo_t c1, c2;
```



Tipos definidos por el usuario

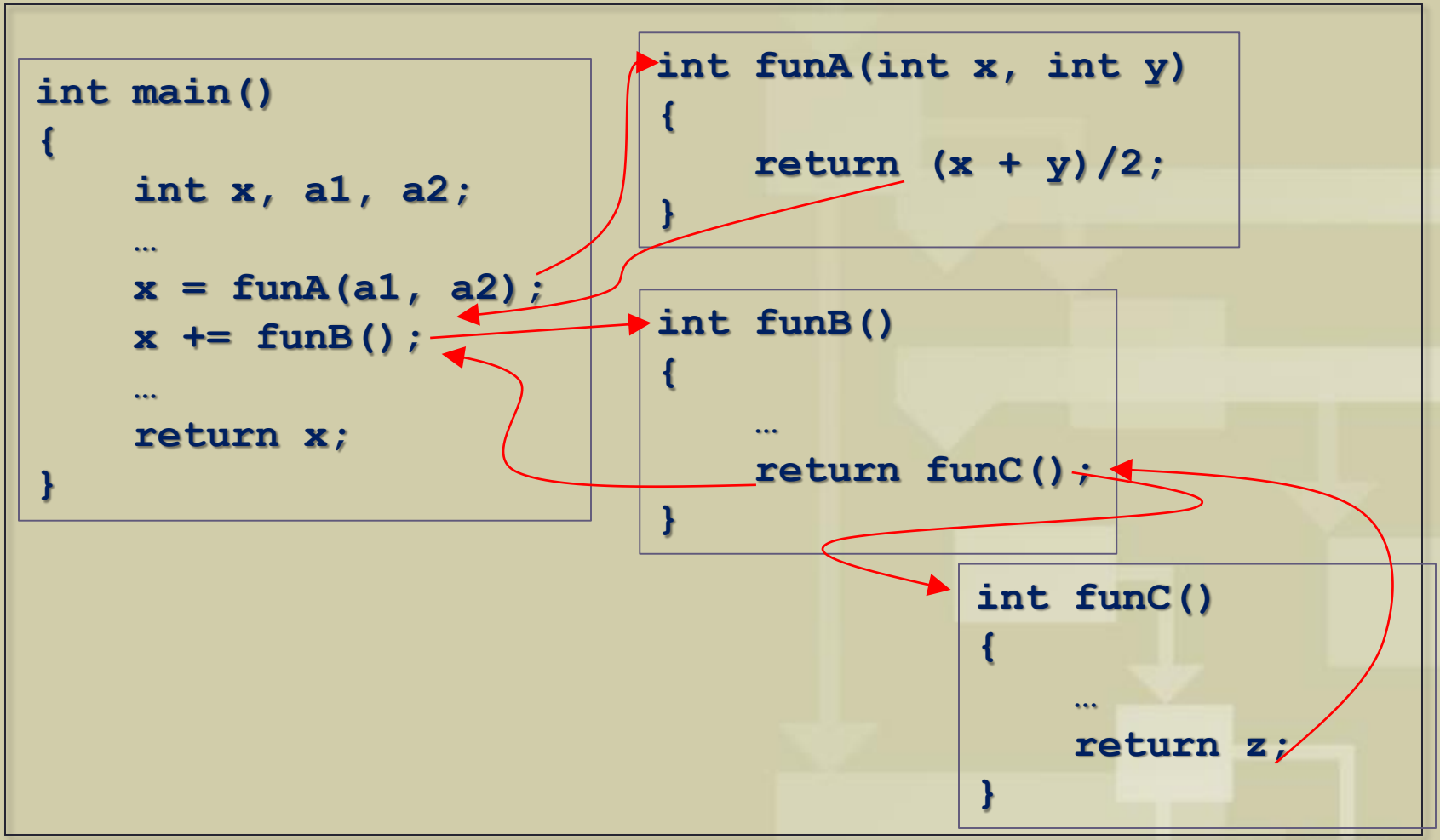
- Si bien una variable de este tipo contiene todos los atributos definidos en la estructura, se puede individualizar un atributo en particular a través de lo que se denomina “especificador de campo”: el símbolo ‘.’

```
Complejo_t c1, c2, suma;
c1.re = 4;
c1.im = 8;

c2.re = 1;
c2.im = 32;

suma.re = c1.re + c2.re;
suma.im = c1.im + c2.im;
```

Flujo de ejecución de un programa

- La función **`main`** toma el control del programa, ejecutando todas las sentencias que contiene su cuerpo.
- La ejecución de una función es secuencial, pero pueden incluirse bloques condicionales, repetitivos o llamados a funciones (inclusive a ella misma).
- Llamar a una función es transferirle el control de ejecución. Cuando la función termina, el control vuelve a la sentencia posterior a la llamada original.
- Una función termina cuando llega al final de su cuerpo, o alcanza una sentencia "**`return`**".

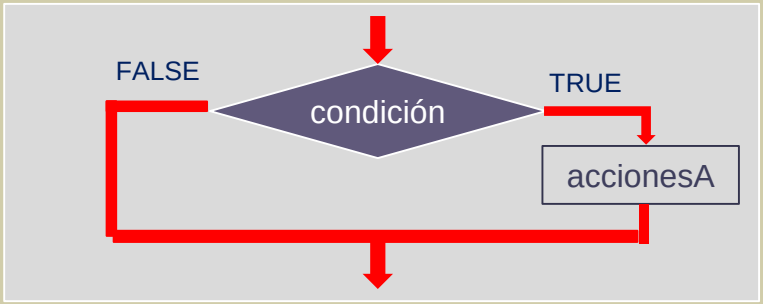
Flujo de ejecución de un programa



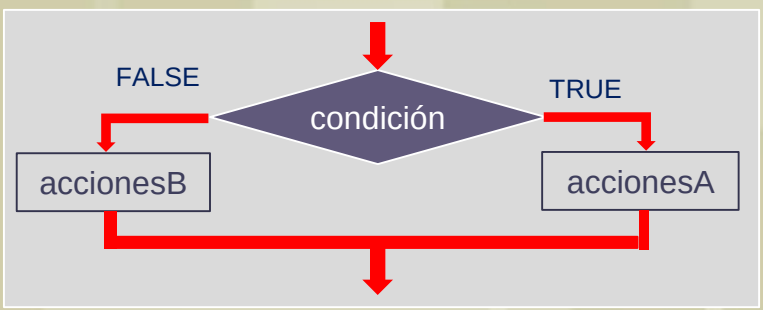
Bloque Condicional `if`

- El bloque **`if`** es una estructura de control condicional. Su sintaxis es:

```
if (condición) {
    accionA1;
    accionA2;
}
```




```
if (condición) {
    accionA1;
    accionA2;
} else {
    accionB1;
    accionB2;
}
```



Condición lógica

- La condición es una expresión que establece la condición lógica que debe satisfacerse para que el cuerpo de la estructura de control se ejecute.
- La expresión puede contener operadores relacionales, lógicos, algebraicos y de bits.
- En el caso de que la expresión termine en un valor numérico, este se tratará como verdadero si su valor es distinto de 0, y como falso si su valor es 0.

Condición lógica

- Ejemplos de condiciones lógicas:

```
X <  5                 // verdadero si X es menor a 5
X > 1 && X <= 5        // verdadero si X esta en (1,5]
(X>>3) & 1             // verdadero si el 4to bit de X es 1
(X % 7) == 0           // verdadero si X es múltiplo de 7
X                      // verdadero si X es distinto de 0
(X % 2)                // verdadero si X es impar
(X & 1)                // verdadero si X es impar
8                      // verdadero siempre
0                      // falso siempre
```

Ejemplo de `if`

paridad.c

```
/* Programa que imprime si un número ingresado por
   el usuario es par o no, aprovechando en operador
   módulo (%)     */
#include <stdio.h>

int main()
{
    int N;

    printf("Ingrese el valor N: ");
    scanf("%d", &N);
    if(N % 2)
        printf("El numero %d es IMPAR\n", N);
    else
      printf("El numero %d es PAR\n", N);

    return 0;
}
```

Bloque Repetitivo **`while`**

- El bloque **`while`** es una estructura de control repetitivo. Su sintaxis es:

```
while ( condición ) {
    accion1;
    accion2;
    …
}
```

En el caso de que el cuerpo posea una sola sentencia puede omitirse las { }. Esto vale para cualquier estructura de control de flujo.

Ejemplo de **while**

conversor.c

```
/* Programa que imprime la tabla de conversión de
Fahrenheit a Celsius para F = 0, 20, 40,..., 300 */
#include <stdio.h>

int main()
{
    int lower = 0, upper = 300, step = 20;
    float fahr, celsius;

    fahr = lower;

    while(fahr <= upper) {
        celsius = (5.0/9.0) * (fahr - 32.0);
        printf("%4.0f F -> %6.1f C\n", fahr, celsius);
        fahr = fahr + step;
    } /* fin del while */
    return 0;
} /* fin del main */
```

Ejemplo de `while`

```
$ ./conversor
   0 F ->  -17.8 C
  20 F ->   -6.7 C
  40 F ->    4.4 C
  60 F ->   15.6 C
  80 F ->   26.7 C
 100 F ->   37.8 C
 120 F ->   48.9 C
 140 F ->   60.0 C
 160 F ->   71.1 C
 180 F ->   82.2 C
 200 F ->   93.3 C
 220 F ->  104.4 C
 240 F ->  115.6 C
 260 F ->  126.7 C
 280 F ->  137.8 C
 300 F ->  148.9 C
```

Bloque Repetitivo **`for`**

- El bloque **`for`** es una estructura de control repetitivo. Su sintaxis es:

```
for( a; b; c ) {
    accion1;
    accion2;
}
```

- La parte **`a`** es la inicialización, y puede tener una, ninguna o varias sentencias separados por ‘,’.
- La parte **`b`** establece la condición de continuidad de la iteración.
- La parte **`c`** es ejecutada al final de cada ciclo.

Ejemplo de `for`

conversor2.c

```
/* Tabla de conversión de grados F a Celsius
   utilizando constantes simbólicas y bloque for */
#include <stdio.h>

#define LOWER 0
#define UPPER 300
#define STEP 20

int main()
{
  int fahr;

  for(fahr = LOWER; fahr <= UPPER; fahr += STEP)
    printf("%4d F -> %6.1f C\n", fahr, (5.0/9.0)*
            (fahr - 32));
  return 0;
}
```

Bloque Repetitivo **do-while**

- El bloque **do-while** es una estructura de control repetitivo levemente diferente al **while**. Su sintaxis es:

```
do {
    accion1;
    accion2;
} while (condición);
```

Ejemplo de **do-while**

dowhile.c

```
/* Programa que pide al usuario un número entero entre 1 y 10.
Se continúa pidiendo el valor hasta que cumpla la condición */
#include <stdio.h>

int main()
{
    int n, error;

    do {
        printf("Ingrese un número entero entre 1 y 10: ");
        scanf("%d", &n);
        if (error = (n < 1 || n > 10))
            printf("\nERROR: Intentelo nuevamente!!\n\n");
    } while(error);
/* ahora puedo procesar el valor ingresado sabiendo que es correcto. */
    // ...
    return 0;
} /* fin del main */
```

Sentencia **break**

- El **break** produce la salida inmediata del **while**, **for**, **do** o **switch** en que se encuentra, por ej.:

```
while(1) {
    scanf("%lf",&x);
    if (x < 0.0)  /* si x es negativo */
       break;       /* salgo del while */
    printf("%lf\n",sqrt(x));
}
/* ... y vengo a parar acá */
printf("El número es negativo!!\n");
```

Sentencia **`continue`**

- El **`continue`** termina la iteración actual del **`while`**, **`for`**, o **`do`** en que se encuentra, por ej.:

```
for(i=0; i < 20; i++) {
   if (i == 13)
      continue; /* supersticioso yo? */
   printf("%d\n",i);
}
```

Sentencia `goto`

- El **`goto`** produce un salto incondicional a una sentencia con una etiqueta que se encuentra en algún lugar dentro de la misma función, por ej.:

```
   for(i=1; i<100; i++) {
      for (j=1; j<100; j++) {
         if (i*j == i+j )
            goto afuera;
         printf("%d %d\n",i, j);
      }
   }
afuera:
   printf("Salida de emergencia?\n");
```

Formateo del código

```
#include <stdio.h>

int main()
{
    int Lower = 0, Upper = 300, Step = 20;
    float Fahr, Celsius;

    Fahr = Lower;

    while(Fahr <= Upper) {
        Celsius = (5.0/9.0) * (Fahr - 32.0);
        printf("%4.0f F -> %6.1f C\n", Fahr, Celsius);
        Fahr = Fahr + Step;
    } /* fin del while */
    return 0;
} /* fin del main */
```

Formateo del código

```
#include <stdio.h>

int main(){int Lower=0,Upper=300,Step=20;float
Fahr,Celsius;Fahr=Lower;while(Fahr<=Upper){Celsius=(5.0/
9.0)*(Fahr-32.0);printf("%4.0f F -> %6.1f C\n",Fahr,
Celsius);Fahr=Fahr+Step;}/* fin del while */ return 0;}/*
fin del main */
```

Operador condicional

- El operador condicional tiene la sintaxis:

```
cond ? expr1 : expr2;
```

Si **cond** es verdadero, evalúa y toma el resultado de **expr1**, si **cond** es falso evalúa y toma el resultado de **expr2**

```
if (y < z)
   x = y;
else
   x = z;
```

```
/* equivalente a: */
x = (y < z) ? y : z;
```

Bloque **switch**

- Variación del bloque if para casos múltiples

```
switch(expr) {   // expr define el caso de entrada
    case Caso1:  // los casos son constantes
        Accion1_1;
        ...
        break;   // si no existiese, continuaría
    case Caso2:
        Accion2_1;
        ...
        break;
    ...
    default:     // puede no existir un default
        AccionD_1;
        ...
        break;
}
```

```
/* Simula el comportamiento de una entidad que se mueve al azar en 2D */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define IZQUIERDA  0
#define DERECHA    1
#define ARRIBA     2
#define ABAJO      3
#define QUIETO     4

#define NUM_PASOS   100000

typedef struct Pos {
   int x;
   int y;
} Pos_t;
```

```
int main() {
    Pos_t pos = { 0, 0 };
    int contador;

    srand(time(NULL));          /* Inicializa generador pseudo-aleatorio */

    for(contador = 0; contador < NUM_PASOS; contador++) {
      switch(rand() % 5) {      /* Sortea una dirección                  */
        case IZQUIERDA:
            pos.x--;
            break;
        case DERECHA:
            pos.x++;
            break;
        case ARRIBA:
            pos.y++;
            break;
        case ABAJO:
            pos.y--;
            break;
      }
    }
    printf("PosX: %d, PosY: %d\nDistancia: %lf\n",
                pos.x, pos.y, sqrt(pos.x*pos.x + pos.y*pos.y));
    return 0;
}
```

Funciones

- Sientan la base para la reutilización de código.
- Definen los módulos constructivo de resolución de subproblemas en el paradigma de DIVIDE y CONQUISTA.
- Permiten producir programas que son más fáciles de escribir, leer, entender, depurar, modificar y mantener.
- Las funciones deben ser **declaradas** y **definidas**.

Funciones

- La ***declaración*** (también llamada ***prototipo)*** de una función consiste en indicar el nombre de la función, el tipo del valor retornado (**`void`** si no retorna nada) y la cantidad y tipo de los argumentos que deben ser suministrados al llamar a la función.

```
int main(void);
void exit(int);
double sqrt(double);
double pow(double, double);
```

- Es muy importante para la validación de argumentos:

```
double sr2 = sqrt(2);          // conversión
double sr3 = sqrt("tres");     // error
```

Funciones

- La ***definición*** de una función es una declaración seguida por el cuerpo de la función.

```
void printMessage(void) {
    printf("Hello world!\n");
}

int main(void) {
    printMessage();
    printMessage();
    return 0;
}
```

Argumentos de funciones

- Los argumentos que se pasan a la función sirven para parametrizar su comportamiento (en general no resuelven un único problema, sino una familia de ellos).

```
void triangular(int n) {
    int i, triang = 0;
    for( i = 1; i <= n; i++ )
        triang += i;
    printf("Num triangular %d es %d\n", n, triang);
}

int main(void) {
    triangular(10);
    triangular(25);
    return 0;
}
```

Variables locales automáticas

- Las variables definidas dentro de una función se conocen como variables locales automáticas.
- Son creadas automáticamente cada vez que la función es activada.
- Estas variables son sólo accesibles desde la función que las definió.
- El valor inicial es asignado cada vez que se ejecuta la función.
- Variables locales no inicializadas tienen valores no controlados.
- Los argumentos de una función son variables automáticas que se inicializan con los valores que se les dieron al llamarse la función.

```
void triangular(int n) {
    int i, triang = 0;
    for( i = 1; i <= n; i++ )
        triang += i;
    printf("Num triangular %d es %d\n", n, triang);
}
```

Valor de retorno

- Las funciones que no haya sido declaradas como **`void`** deben retornar un valor del tipo declarado.
- Las funciones declaradas **`void`** no pueden retornar un valor.
- El valor a retornar se indica con la sentencia **`return`**.
- Puede haber múltiples sentencias **`return`** en una función.

```
int  f1() { }              // error
void f2() { }              // ok
int  f3() { return 1; }    // ok
void f4() { return 1; }    // error
int  f5() { return; }      // error
void f6() { return; }      // ok

int fac(int n)  { return (n > 1) ? n * fac(n-1) : 1; }

int fac2(int n) {
    if( n > 1 )
       return n * fac2(n-1);
    return 1;
}
```

Valor de retorno

```
int triangular(int n) {
    int i, triang = 0;
    for( i = 1; i <= n; i++ )
        triang += i;
    return triang;
}

int main(void) {
    int result = triangular(10) + triangular(25);
    printf("Resultado: %d\n", result);
    return 0;
}

float absValue(float x) {
    if( x < 0 )
        x = -x;
    return x;
}
```

Dispositivos predefinidos de I/O

- Dispositivos de I/O: un programa tiene 3 dispositivos predefinidos: de entrada, de salida y de error.
- Por defecto:
  - Entrada: teclado
  - Salida: consola
  - Error: consola

funciones básicas de I/O

```
// retorna el próximo caracter disponible
// en el dispositivo estándar de entrada,
// o EOF para indicar error o final de
// entrada (End-Of-File)
int getchar();

// escribe el caracter c en el
// dispositivo estándar de salida y
// retornan el caracter escrito o EOF
// para indicar error
int putchar(int c);
```

Ejemplos de I/O

copia.c

```
/* Copiador de archivos     */
#include <stdio.h>
int main()
{
    int c;
    while((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

Redireccionamiento

- Los sistemas operativos proveen mecanismos de redireccionamiento:
  - dispositivos de entrada (<)
  - Dispositivo de salida (>, >>)
  - Dispositivo de errores (2>, 2>>)

Redireccionamiento, ejemplos

Redirecciona la salida al archivo “resultados.dat”

```
$ ./prog > resultados.dat
```

Redirecciona la entrada desde archivo “datos.dat”

```
$ ./prog < datos.dat
```

Redirecciona entrada, salida y error

```
$ ./prog < input.dat > output.dat 2> errores.dat
```

Ejemplos de I/O

contador.c

```
/* Contador de caracteres */
#include <stdio.h>
int main()
{
    int nc = 0;

    while(getchar() != EOF)
        ++nc;
    printf("Leí %d carácteres\n",nc);
    return 0;
}
```

Ejemplos de I/O

lineas.c

```
/* Contador de líneas */
#include <stdio.h>
int main()
{
    int c, nl = 0;

    while((c = getchar()) != EOF)
        if(c == '\n')
            nl++;

    printf("Cantidad de líneas leídas: %d\n",nl);
    return 0;
}
```

Arrays

- Conjunto ordenado de datos del mismo tipo.
- El conjunto entero está identificado con un nombre.
- A cada dato individual se accede mediante un índice.
- Definición de un array:

```
int array[50];
```

tipo nombre cantidad de elementos

Arrays

- Acceso a los elementos:

```
int iarray[50];
double darray[100];
int i = 25;

iarray[0]  = 23;
iarray[49] = i;

for( i = 0; i < 100; i++ ) {
    darray[i] = i * sqrt(i);
    printf("%d %f\n", i, darray[i]);
}

i = iarray[0];
iarray[i] = i * i;
```

Arrays de estructuras

- Acceso a los elementos y a campos de elementos:

```
typedef struct {
    double x;
    double y;
} Punto2D_t;
...
Punto2D_t parray[10];
...
...
parray[i]  = parray[i+1];  // copia un elemento
...                        // completo
...
parray[i].x = parray[i+1].x;   // copia el
                               // campo "x"
```

Arrays

- El índice del primer elemento de un array es 0.
- El índice del último elemento de un array de **N** elementos es **N-1**.
- C no verifica límites.

```
int v[10];

v[0] = v[7] = 3;

v[10] = 8;
v[-1] = 5;
```

Ejemplo con arrays

```
#include <stdio.h>

#define N 20

int main(void)
{
    int Fibonacci[N], i;

    Fibonacci[0] = 0;
    Fibonacci[1] = 1;

    for( i = 2; i < N; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

    for( i = 0; i < N; ++i )
        printf("%i\n", Fibonacci[i]);

    return 0;
}
```

Inicialización de arrays

```
int integers[5] = { 0, 1, 2, 3, 4 };
char vocales[] = { 'a', 'e', 'i', 'o', 'u' };
float datos[500] = { 100.0, 300.0, 505.5 };

double x[500] = { [2] = 505.5, [0] = 100.0, [1] = 300.0 };

Punto2D_t parray[3] = {{2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0}};

#include <stdio.h>

int main(void)
{
    char word[] = { 'H', 'e', 'l', 'l', 'o' , '!' };
    int  i;

    for( i = 0; i < sizeof(word)/sizeof(word[0]); ++i )
        putchar(word[i]);
    putchar('\n');
    return 0;
}
```

Arrays multidimensionales

```
#define N 5

double mat[N][N];
double tr = 0;
int i, j;

for( i = 0; i < N; ++i )
    for( j = 0; j < N; ++j )
        mat[i][j] = (i == j ? 1 : 0);

for( i = 0; i < N; ++i )
    tr += mat[i][i];

for( i = 0; i < N; ++i ) {
    for( j = 0; j < N; ++j )
        printf("%lf  ", mat[i][j]);
    putchar('\n');
}
```

Arrays como argumentos

```
double sumaVec(double vec[], int n) {
    double s = 0;
    int i;
    for( i = 0; i < n; i++ )
        s += vec[i];
    return s;
}

int main(void) {
    double result;
    double val[] = { 2.3, 3.14, 0.125, 8.9 };
    result = sumaVec(val, sizeof(val)/sizeof(val[0]));
    printf("Suma: %lf\n", result);
    return 0;
}
```

Arrays multidimensionales

```
#define N 3
double traza(double mat[N][N]) {
    double tr = 0;
    int i;
    for( i = 0; i < N; i++ )
        tr += mat[i][i];
    return tr;
}

int main(void) {
    double result;
    double mat[][N] = {{2.3, 3.14, 0.12},{2.1, 8.9, 0},
                       {0,   2,    4.56}};
    result = traza(mat);
    printf("traza: %lf\n", result);
    return 0;
}
```

Arrays de chars

```
void concat(char result[], const char str1[], int n1,
            const char str2[], int n2);

int main(void)
{
    char s1[5] = { 'T', 'e', 's', 't', ' ' };
    char s2[6] = { 'w', 'o', 'r', 'k', 's' , '.' };
    char s3[20];
    int i;

    concat(s3, s1, 5, s2, 6);

    for( i = 0; i < 11; ++i )
        putchar(s3[i]);
    putchar('\n');

    return 0;
}
```

Strings: Arrays de chars + convención

```
printf("Hello world!\n");
```

- `"Hello world!\n"` es un string
- Un string es un array de **`char`** terminado con un 0.

```
char s1[] = { 'T', 'e', 's', 't', ' ', 0 };

char s2[] = "works.";
char s3[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char s4[] = { "world!" };
```

- No es necesario pasar la longitud de los strings:

```
void concat(char result[], const char str1[],
            const char str2[]);
```

Strings: Arrays de chars + convención

```
void concat(char result[], const char str1[],
            const char str2[])
{
    int idx1 = 0, idx2 = 0;
    char c;

    while( (c = str1[idx1]) != 0 )
        result[idx1++] = c;

    while( (c = str2[idx2]) != 0 )
        result[idx1 + idx2++] = c;

    result[idx1 + idx2] = 0;
}
```

String input/output

- Output:

```
printf("Hello world!\n");

char s[] = "Hola mundo!";
puts(s);

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

- Input:

```
char string[64];

scanf("%s", string);   // Notar la falta de &
```

Manejo de archivos

- Se define una variable de un tipo especial:

```
FILE *stream;
```

- Antes de usarlo el archivo se “abre”:

```
stream = fopen("datos.txt", "w");
```

- Se usa:

```
fprintf(stream, "Hola mundo!\n");
```

- Se “cierra” cuando no se usa más:

```
fclose(stream);
```

Apertura de un archivo

- **`fopen(NOMBRE_ARCHIVO, MODO)`**
  - **`NOMBRE_ARCHIVO`** es un string
  - **`MODO`** es un string

- Modos de apertura:
  - **`"r"`** archivo para lectura
  - **`"w"`** archivo para escritura, crea o trunca existente
  - **`"a"`** archivo para escritura, crea o continúa

- **`fopen`** retorna **`NULL`** cuando falla.

Escritura de un archivo

- Tiene que haber sido abierto con modo `"w"` o `"a"`
- Escritura de datos formato “texto”

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

FILE *stream = fopen("salida.txt", "w");
assert(stream != NULL);

fprintf(stream, "Hola mundo!\n");
fprintf(stream, "%d %lf\n", 2, 3.14);

fclose(stream);
```

Lectura de un archivo

- Tiene que haber sido abierto con modo **"r"**
- Escritura de datos formato "texto"

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

int x; double d; char str[128];

FILE *stream = fopen("salida.txt", "r");
assert(stream != NULL);

fscanf(stream, "%d", &x);
fscanf(stream, "%lf", &d);
fscanf(stream, "%s", str);

fclose(stream);
```

I/O de caracteres

- Lee un carácter de un archivo, símil **`getchar()`**

```
int fgetc(FILE *stream);
```

- Escribe un carácter a un archivo, símil **`putchar()`**

```
int fputc(int c, FILE *stream);
```

- Se pueden escribir datos binarios a un archivo.
- 3 archivos previamente abiertos:

```
stdin  stdout  stderr
```

Ejemplo I/O a archivo

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

int main()
{
    int x, ret;
    FILE *in = fopen("entrada.txt", "r");
    FILE *out = fopen("salida.txt", "w");

    if( in == NULL ) {
        fprintf(stderr, "error abriendo entrada.txt\n");
        return 1;
    }

    if( out == NULL ) {
        perror("Abriendo salida.txt");
        out = stdout;
    }
```

Ejemplo I/O a archivo, cont.

```
    while( (ret = fscanf(in, "%d", &x)) && ret != EOF )
    {
        fprintf(out, "%d\n", 2 * x);
        fprintf(stderr, "Lei: %d\n", x);
    }

    fclose(in);
    fclose(out);

    return 0;
}
```

- Podría haber sido:

```
    while( fscanf(in, "%d", &x) == 1 )
```

Convención de pasaje de argumentos

- **C utiliza la convención de pasaje de argumentos por valor**

```
void duplica(int x)
{
    x = x * 2;
    printf("en duplica, x vale: %d\n", x);
}


int main()
{
    int a = 4;

    printf("en main a vale: %d\n", a);

    duplica(a);

    printf("en main después de duplica a vale: %d\n", a);

    return 0;
}
```

Convención de pasaje de argumentos

```
void duplica(int x[1])
{
    x[0] = x[0] * 2;
    printf("en duplica, x[0] vale: %d\n", x[0]);
}


int main()
{
    int a[1] = { 4 };

    printf("en main a[0] vale: %d\n", a[0]);

    duplica(a);

    printf("en main después de duplica a[0] vale: %d\n", a[0]);

    return 0;
}
```

Ámbito de variables

El ámbito de una variable define:

- Visibilidad: dónde es visible.
- Tiempo de vida: cuando se crea y cuando deja de existir.

Existen principalmente 3 ámbitos para variables:

- Variables locales o automáticas o de stack.
- Variables globales.
- Variables estáticas.

Variables Locales

A las variables con ámbito local también se las refiere como variables locales, o automáticas o de stack.

- **Visibilidad**: bloque de código en que se la define. (puede ser declarada al inicio de una función o dentro de un bloque interno a una función)

- **Tiempo de vida**: el tiempo que dure la ejecución del bloque que la definió.

Variables Locales

- Las variables **local**es se crean **automática**mente cuando su alcance (“scope”) se activa, y el lugar donde se crean es en una zona denominada “**stack**” o pila.

- Dejan de tener sentido cuando salen de scope. La memoria que estaba asociada a esas variables puede reutilizarse.

Variables Locales

```
void f1(int a1, double a2) // a1 y a2 locales a f1
{
    int a, b, c;     // variables locales a f1
    …
    …
    while(…) {
        int d;       // variable local al bloque
        …
    }
}
void f2(char a1)      // a1 local a f2
{
    int a;           // variables locales a f2
    …
}
```

Variables Locales

```
#include <time.h>
#include <stdio.h>
int main()
{
    int a = 1, c = 2;
    {
        int a = 3;
        printf("interno a = %d\n", a);
    }
    printf("local a = %d\n", a);
    return 0;
}
=======================================
$ ./test
interno a = 3
local a = 1
```

Variables Locales

```
void f() {
    int a;
    printf("a de f() esta en 0x%x\n", &a);
}
void g() {
    int b;
    printf("b de g() esta en 0x%x\n", &b);
    f();
}

int main()
{
    f();
    g();
    return 0;
}
```

```
$ ./test
a de f() esta en 0x28cd14
b de g() esta en 0x28cd14
a de f() esta en 0x28ccf4
```

Ámbito Global

Las variables con ámbito global son aquellas que se declaran fuera de cualquier función.

- **Visibilidad**: Son visibles desde cualquier función.
- **Tiempo de vida**: el tiempo que dure la ejecución del programa.
- **Linkage**: Por defecto tienen “linkage” externo, lo que las hace visibles desde otros módulos del programa.

Variables Globales

```
int a = 1;  // variables globales
int b = 4;

void f() {
    int a;
    a = 5;
    b = 8;
    printf("en f, \t\ta: %d   b: %d\n", a, b);
}
int main()
{
    printf("antes de f, \ta: %d   b: %d\n", a, b);
    f();
    printf("después de f, \ta: %d   b: %d\n", a, b);
    return 0;
}
```

```
$ ./test
antes de f,     a: 1   b: 4
en f,           a: 5   b: 8
después de f,   a: 1   b: 8
```

Variables Globales

main.c

```
void f();
void g();
int a;
int main()
{
    f();
    print(A = %d\n", a);
    g();
    print(A = %d\n", a);
    return 0;
}
```

m1.c

```
extern int a;
void g()
{
    a = 14;
}
```

m2.c

```
extern int a;
void f()
{
    a = 8;
}
```

```
$ gcc main.c m1.c m2.c –o main
$ ./main
A = 8
A = 14
```

Variables Globales

- Se debe minimizar el uso de variables globales.
- Las funciones deberían enterarse del problema que tienen que resolver sólo a través de sus argumentos.
- Una función que utiliza variables globales es muy difícil de reutilizar en otro contexto.
- Variables locales sumado a la convención de pasajes de argumentos por valor otorga gran controlabilidad de modificación de una variable.

Variables Estáticas

- Aplicadas sobre variables de ámbito local: Le modifican donde son alojadas, en lugar del stack, se crean en la zona de «datos», modificando también su tiempo de vida.

- Aplicadas sobre variables de ámbito global o sobre funciones: le modifica el linkage (linkage estático), evitando que pueda ser vista desde afuera del módulo.

Variables Estáticas

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

void f()
{
    static int s = 0;
    int b = 0;
    s++;
    b++;
    printf("s: %d  b: %d\n", s, b);
}
int main() {
    f(); f(); f(); f();
    return 0;
}
```

Recursión

- Especificación de un proceso basado en su propia definición. Involucra:

  - Una o más soluciones triviales (directas)

  - Una o más soluciones que recurren en el mismo problema, directa o indirectamente

Recursión

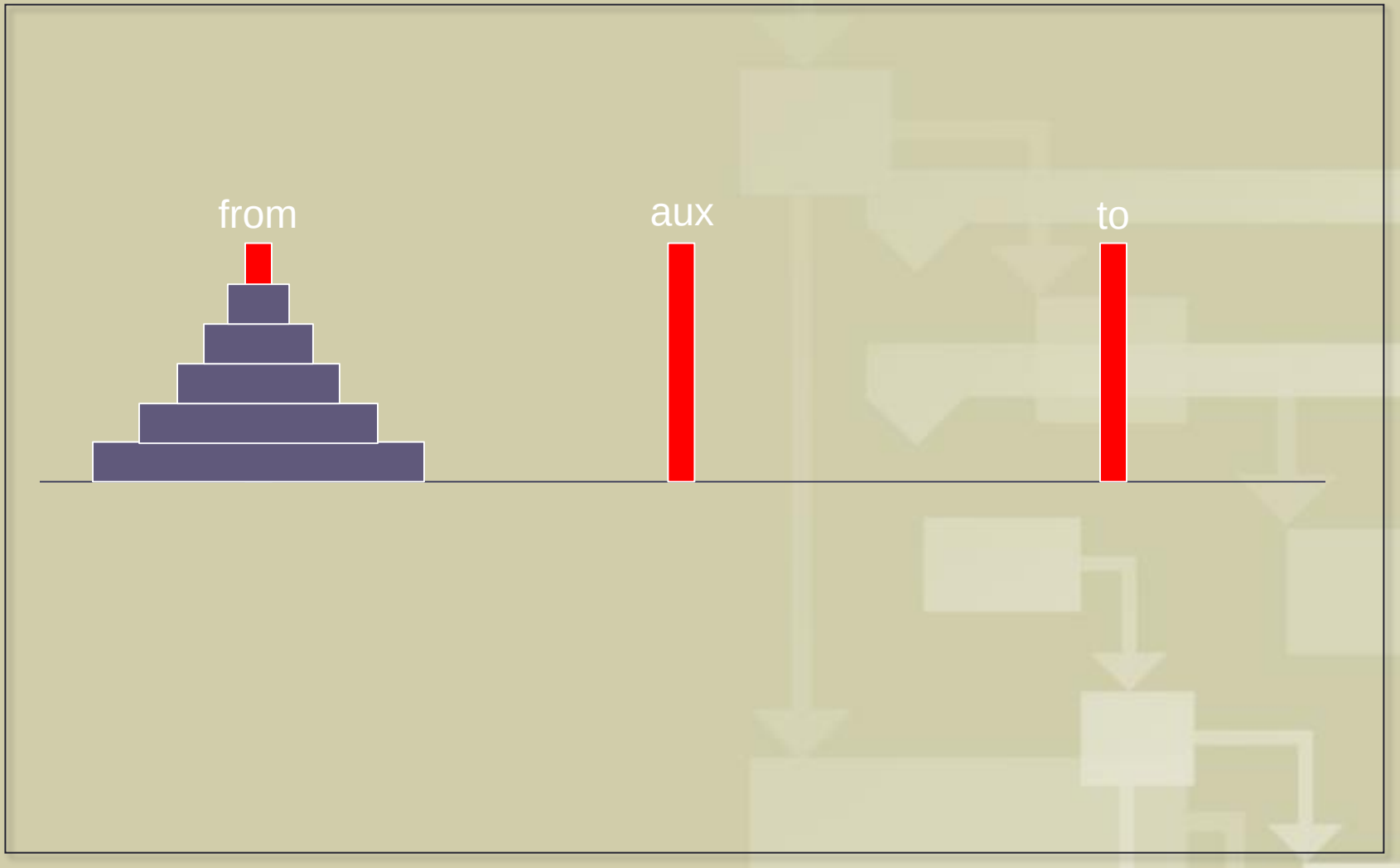
$$\mathbf{n!} = \begin{cases} \Rightarrow \mathbf{1} & \boldsymbol{si\ n == 0} \\ \Rightarrow \boldsymbol{n(n-1)!} & \boldsymbol{si\ n > 0} \end{cases}$$

$$\mathbf{Fb(n)} = \begin{cases} \Rightarrow \mathbf{0} & \boldsymbol{si\ n == 0} \\ \Rightarrow \mathbf{1} & \boldsymbol{si\ n == 1} \\ \Rightarrow \boldsymbol{Fb(n-1) + Fb(n-2)} & \boldsymbol{si\ n > 1} \end{cases}$$

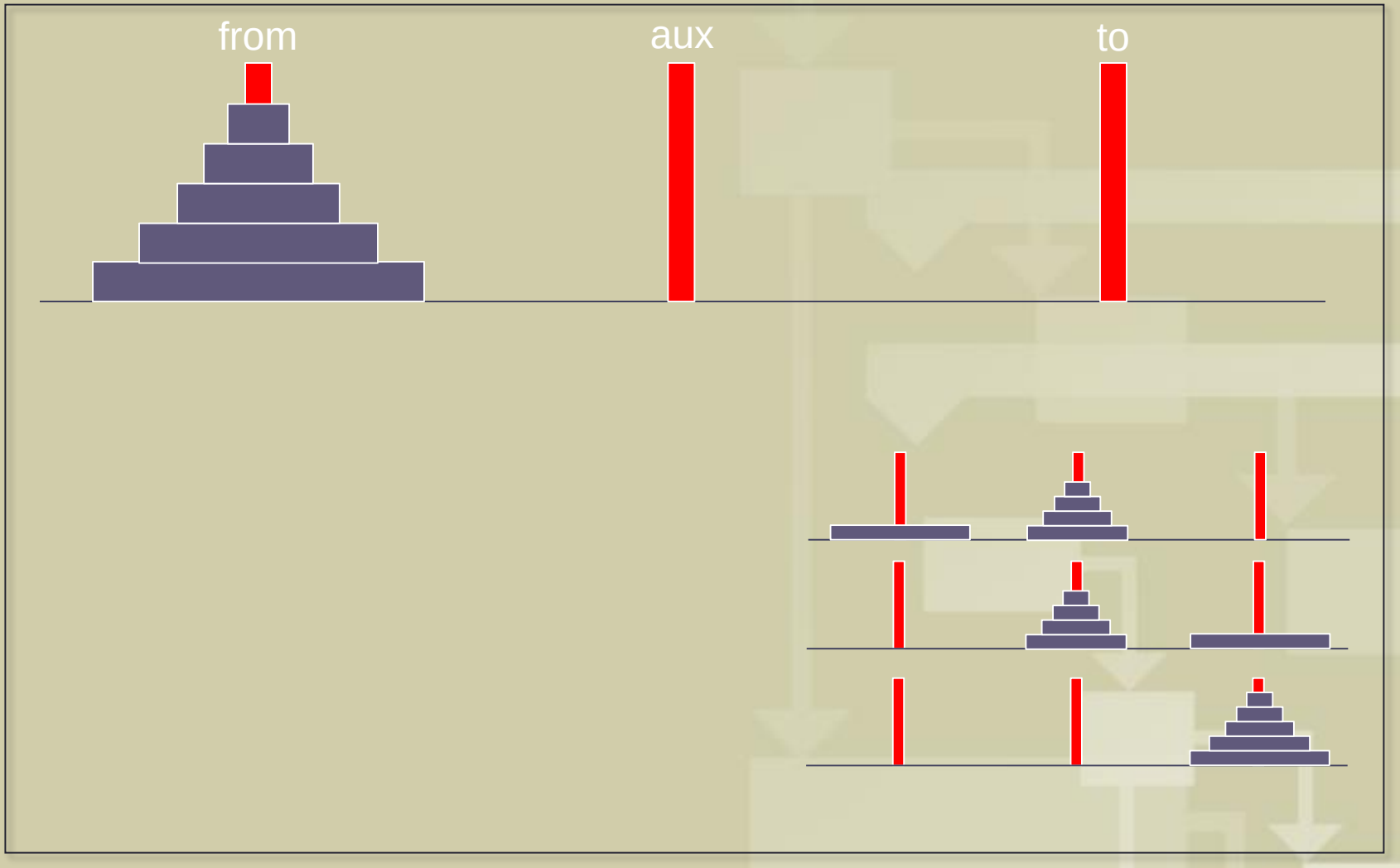
Recursión

```
int factorial(int n) {
    if( n == 0 )
        return 1;
    return n * factorial(n - 1);
}
int fb(int n) {
    if( n == 0 || n == 1)
        return n;
    return fb(n - 1) + fb(n - 2);
}
int main()
{
    printf("7! = %d\n", factorial(7));
    printf("fb(7) = %d\n", fb(7));
    return 0;
}
```

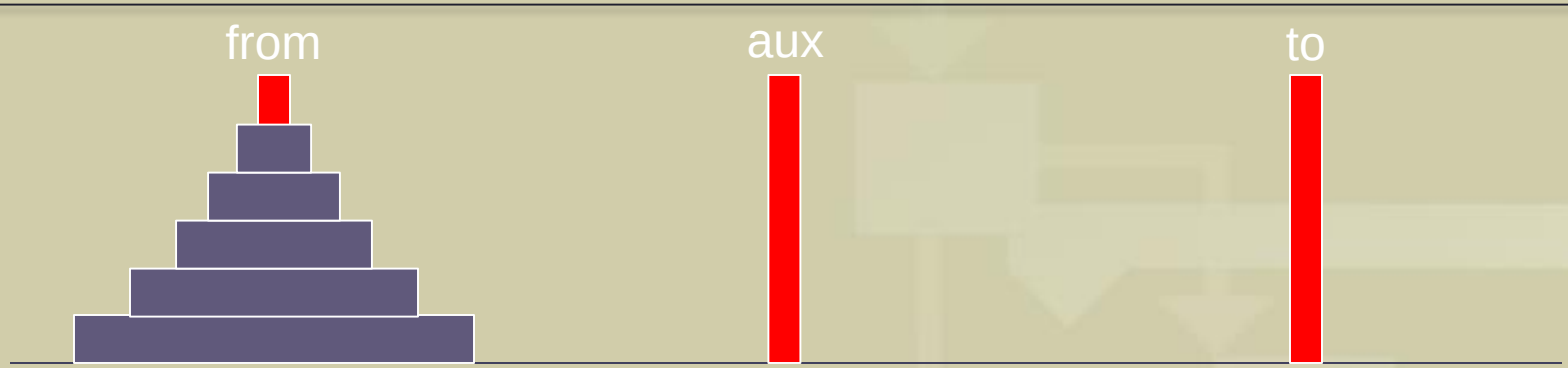
Hanoi



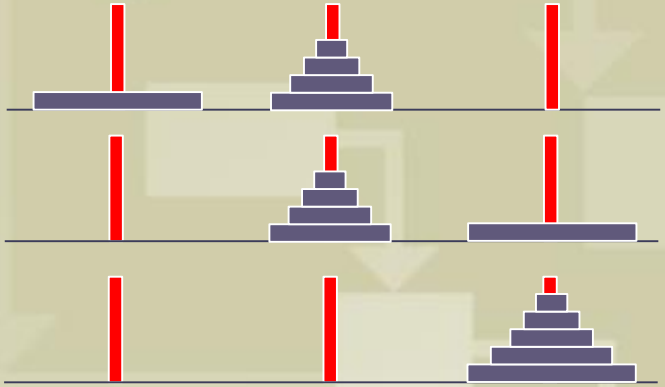
Hanoi



Hanoi

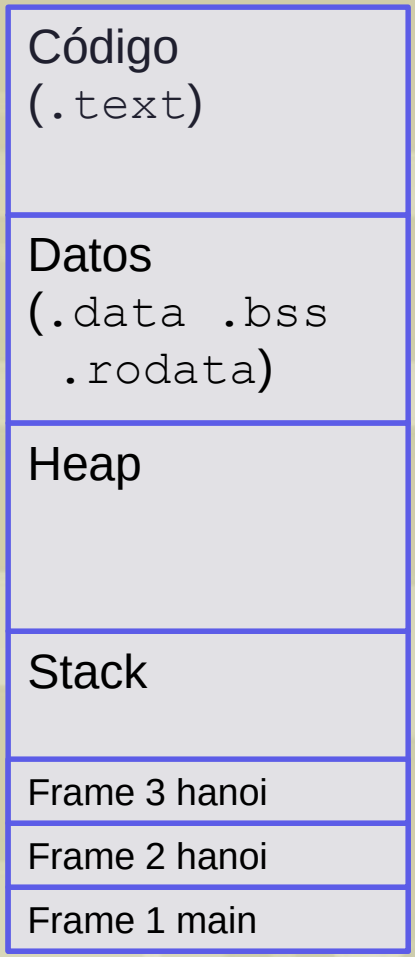



```
void hanoi(int n, int from, int to, int aux) {
    if( n == 1 ) {
        printf("Mover de %d a %d\n", from, to);
        return;
    }
    hanoi( n-1, from, aux, to);
    hanoi(   1, from, to, aux);
    hanoi( n-1, aux, to, from);
}

int main() {
    hanoi(3, 1, 3, 2);
    return 0;
}
```



Variables en Hanoi

```
#include <stdio.h>

int nmvs;

void hanoi(int n, int from, int aux, int to) {
    if( n == 1 ) {
        nmvs++; return;
    }
    hanoi(n-1, from, to, aux);
    hanoi(1, from, aux, to);
    hanoi(n-1, aux, from, to);
}

int main() {
    int i;
    for( i = 2; i < 20; i++ ) {
        nmvs = 0;
        hanoi(i, 1, 2, 3);
        printf("%i nops %d\n", i, nmvs);
    }
    return 0;
}
```



Mapa de Memoria

Unidad de direccionamiento: byte

Cada variable o byte en la memoria tiene una dirección de memoria asociada.

Arquitecturas de 32 bits:

- Rango de 0 a 2^{32}-1.
- Región reservada al sistema operativo
- Múltiples regiones de código y datos
- Múltiples stacks en programas multithreaded
- Regiones vacías

Nuevos operadores

Dos nuevos operadores unarios:

- Dirección de memoria de: **&**
- Contenido de: ***** (derreferencia o indirección)

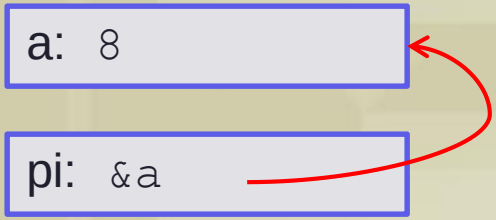
```
#include <stdio.h>

int g = 6;

int main()
{
    int a = 8;

    printf("a  : %8d  \t g  : %8d\n",   a,   g);
    printf("&a : %08x \t &g : %08x\n", &a,  &g);
    printf("*&a: %8d  \t *&g: %8d\n", *&a, *&g);

    return 0;
}
```

```
a  :        8  g  :        6
&a : f517738c  &g : 00600914
*&a:        8  *&g:        6
```

Punteros

Nuevos tipos de variables que sirven para almacenar una dirección de memoria. Los punteros tienen un ***tipo***, que es el tipo de dato que se encuentra en la dirección de memoria.

```
int a = 8;
int *pi = &a;
```



Los punteros son una de las características más sofisticadas del lenguaje C. La potencia y flexibilidad que provee C en el uso de punteros lo diferencia de la mayoría de los otros lenguajes.

Los punteros permiten representar estructuras complejas de datos, cambiar valores pasados como argumentos a funciones, trabajar con memoria “allocada dinámicamente” y trabajar de una manera eficiente y concisa con arrays.

Punteros

- Los punteros tienen un ***tipo***.

```
int a = 8;
int *pi = &a;
```

pi es un puntero a entero, también se puede leer que el contenido de **pi** es un entero.

```
double *pd2 = &a;
```
<<< Error

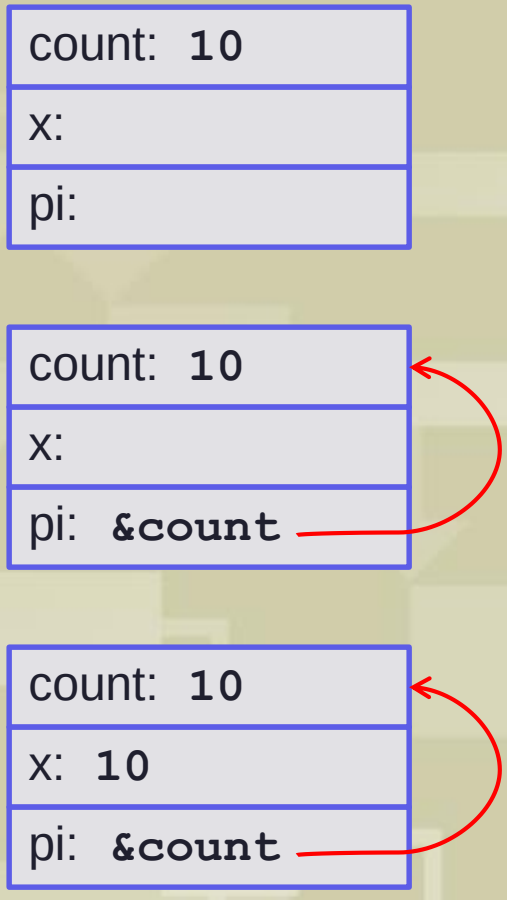
Otros ejemplos de punteros:

```
double *pd;
struct complex *pc;
char **ppc;
int *ap[15];
int (*fp)(char *);
int *f(char *);
```

Ejemplo de Punteros

```
#include <stdio.h>

int main()
{
    int count = 10, x;
    int *pi;

    pi = &count;
    x  = *pi;

    printf("count = %d, x= %d\n",
           count, x);

    return 0;
}
```

```
count = 10, x = 10
```



Ejemplo de Punteros

```
#include <stdio.h>

int main()
{
    char c = 'Q';
    char *char_ptr = &c;

    printf ("%c %c\n", c, *char_ptr);

    c = '/';
    printf ("%c %c\n", c, *char_ptr);

    *char_ptr = '(';
    printf ("%c %c\n", c, *char_ptr);

    return 0;
}
```

```
Q Q
/ /
( (
```

Ejemplo de Punteros

```
#include <stdio.h>

void swap_ints(int *pa, int *pb) {
    int aux = *pa;
    *pa = *pb;
    *pb = aux;
}

int main(){
    int x = 10, y = 25;

    swap_ints(&x, &y);

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

    return 0;
}
```

```
x = 25, y = 10
```

Punteros a estructuras

- Definición similar a un puntero a un tipo nativo:

```
struct Complex {
    double re;
    double im;
};
struct Complex c;
struct Complex *pc;
pc = &c;
```

- Nuevo operador de selector de campo: `->`

```
pc->re = 8;
printf("re: %lf im: %lf\n", pc->re, pc->im);
```

- `pc->re ≡ (*pc).re`

Punteros a estructuras

```
int main() {
    Complex c1 = { 1.2, 3.4 }, c2 = { 2.0, 4.0 };
    Complex c3 = ComplexMultiply(&c1, &c2);
    ComplexPrint(c3);
    return 0;
}

Complex ComplexMultiply(const Complex *pa,
                        const Complex *pb)
{
     Complex res;
     res.re = pa->re * pb->re - pa->im * pb->im;
     res.im = pa->re * pb->im + pa->im * pb->re;
     return res;
}
```

Vectores y Punteros

- Un vector es un puntero al primer elemento del vector.

```
int vec[100];
int *pi = vec;
```

- Los vectores son punteros que no pueden ser “reapuntados”.

```
int a, v1[10], v2[10], *pi;
pi = &a;
pi = v1;
pi = &v2[3];

v2 = v1; <<< Error: v2 es un puntero, pero no lo puedo reasignar
```

- Se puede derreferenciar un puntero como un vector.

```
a = pi[5];
```

- Los vectores son tales cuando se definen, cuando se pasan como argumento a una función van como un puntero, que no es mas que la copia de la dirección de memoria del primer elemento del vector.

Operaciones con Punteros

- Comparar 2 punteros con los operadores **==**, **!=**, **<**, **>**, **<=** ó **>=**, el resultado es un valor verdadero o falso.

```
int a, b, *pia = &a, *pib = &b;
if( pia == pib )
    puts("problemas...");
```

- Suma o resta de un entero a un puntero, el resultado es un puntero.

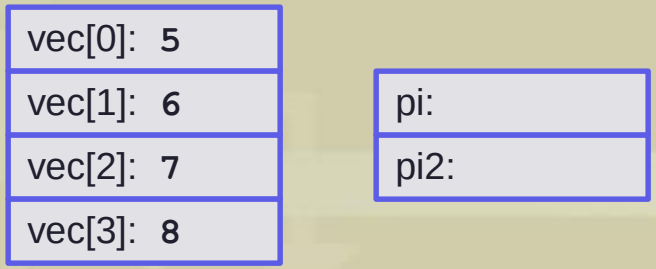
```
int vec[10], *pi = vec, *pi2;
pi2 = pi + 2;
*pi2 = 3;
*(pi + 1) = 7;
```

- Resta de 2 punteros del mismo tipo. El resultado es un entero con signo (entero de tipo **ptrdiff_t**).

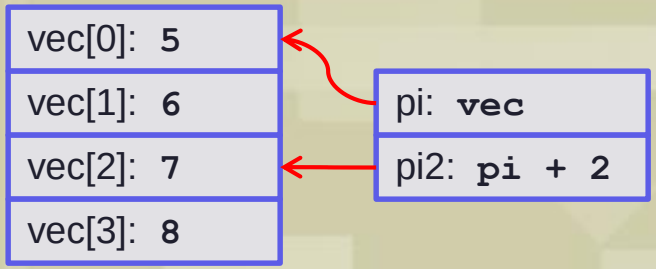
```
int n, vec[10], *pi = &vec[0], *pi2 = &vec[5];
n = pi2 - pi;
```

Suma de Puntero y entero

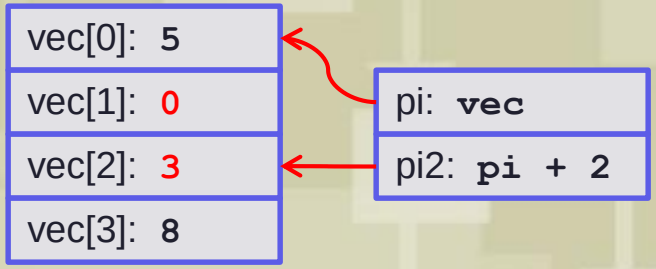
```
int vec[4] = {5,6,7,8};
int *pi, *pi2;
```




```
pi = vec;
pi2 = pi + 2;
```

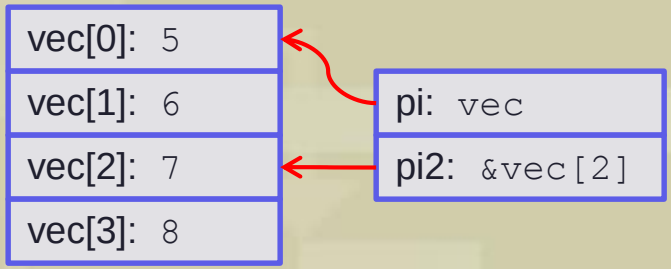



```
*pi2 = 3;
*(pi + 1) = 0;
```

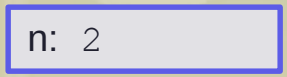


Resta de 2 Punteros

```
int vec[4] = {5,6,7,8};
int *pi = vec;
int *pi2 = &vec[2];
```




```
int n = pi2 - pi;
```




```
if( n != 2 )
    puts("problemas...");
```

- Para ser estrictos **`n`** debería haber sido definido como **`ptrdiff_t`**.

Operaciones con Punteros

```
#include <stdio.h>

void copyString(char *to, char *from);

int main() {
    char string1[] = "A string to be copied.";
    char string2[50];

    copyString(string2, string1);
    printf("%s\n", string2);

    return 0;
}

void copyString(char *to, char *from) {
    while( *to++ = *from++ )
        ;
}
```

```
A string to be copied.
```

Operaciones con Punteros

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    printf("argc: %d\n", argc);

    while( *argv ) {
        printf("arg %d: \"%s\"\n", i, *argv);
        argv++;
        i++;
    }

    return 0;
}
```

Más de Punteros y Vectores

- Los punteros se pueden derreferenciar como los vectores.

```
int a, *ptr, v[] = { 0,1,2,3,4,5 };
ptr = v;
a = ptr[2];
ptr[1] = 567;
```

- Equivalencias.

```
*(ptr + i) ≡ ptr[i]

 (ptr + i) ≡ &ptr[i]
```

Punteros a función

Los punteros a función son punteros que contienen la dirección de memoria donde se encuentra el código de una función. El tipo de un puntero a función está dado por el prototipo de la función a la que apunta. Esto es, el tipo de retorno y la cantidad y tipo de sus argumentos.

- Declaración de una variable puntero a función:

```
int (*fnPtr)(void);
```

Esto declara la variable **`fnPtr`** como un puntero a una función que retorna un entero y no recibe argumentos.

- Asignación de un puntero a función:

```
fnPtr = rand;
```

- Llamado a una función a través de un puntero a función:

```
int r = fnPtr();
```

Punteros a función

El tipo de puntero a función lo utiliza el compilador para asegurar que al ser llamada la función, la cantidad y el tipo de los argumentos y el tipo del valor de retorno sean los adecuados.

- Puntero a función como argumento de una función:

```
#include <math.h>

typedef double(*fun_ptr_type)(double);

double integra(double x0, double x1, fun_ptr_type fun);

int main() {
    double i;

    i = integra(0, M_PI, sin);
    i = integra(0, M_PI, cos);

    return 0;
}
// double integra(double x0, double x1, double(*fun)(double));
```

Allocación dinámica de memoria

```
#include <stdlib.h>

void *malloc(int sizeBytes);
```

➢ Función que alloca una región de memoria de **`sizeBytes`** bytes en el heap y devuelve un puntero a la misma. Cuando no puede satisfacer el requerimiento de memoria, **`malloc()`** retorna **`NULL`**.

```
void free(void *ptr);
```

➢ Función que libera la región de memoria en el heap apuntada por **`ptr`**, previamente allocada con **`malloc()`**.

❖ Toda región allocada con **`malloc()`** debe ser liberada con **`free()`**.

malloc y free

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

double *allocaVector(int pn);
void liberaVector(double *vec);

void ingresaVector(int n, double *vec);
void imprimeVector(int n, double *vec);

int main() {
    int     n;
    double *vector;

    printf("Ingresar la dimensión del vector: ");
    scanf("%d", &n);

    vector = allocaVector(n);

    ingresaVector(n, vector);
    imprimeVector(n, vector);

    liberaVector(vector);

    return 0;
}
```

malloc y free

```
double *allocaVector(int n) {
    double *v = (double *) malloc(n * sizeof(double));
    assert(v != NULL);
    return v;
}

void liberaVector(double *vec) {
    free(vec);
}

void ingresaVector(int n, double *vec) {
    int i;
    for( i = 0; i < n; i++ ) {
        printf("Ingresar componente %d: ", i);
        scanf("%lf", vec + i);
    }
}

void imprimeVector(int n, double *vec) {
    int i;
    for( i = 0; i < n; i++ )
        printf("%lf\n", vec[i]);
}
```

malloc y free

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

void allocaIngresaVector(int *pn, double **pvec);
void imprimeVector(int n, double *vec);
void liberaVector(double *vec);

int main() {
    int     n;
    double *vector;

    allocaIngresaVector(&n, &vector);
    imprimeVector(n, vector);
    liberaVector(vector);

    return 0;
}
```

malloc y free

```
void allocaIngresaVector(int *pn, double **pvec) {
    int i;
    printf("Ingresar la dimensión del vector: ");
    scanf("%d", pn);

    *pvec = (double *) malloc(*pn * sizeof(double));
    assert(*pvec);

    for( i = 0; i < *pn; i++ ) {
        printf("Ingresar componente %d: ", i);
        scanf("%lf", (*pvec)+i);
    }
}
```

Estructuras – allocación dinámica

```
#define COMPLEX_COUNT  20

int main()
{
    Complex *pa = ComplexArrayAlloc(COMPLEX_COUNT);
    assert(pa);

    for( i = 0; i < COMPLEX_COUNT; i++)
       ComplexRead(&pa[i];   // === (pa+i)
    // ...
    ComplexArrayFree(pa);
    return 0;
}
Complex *ComplexArrayAlloc(int cnt)
{
    return (Complex *) malloc(cnt * sizeof(Complex));
}
```

Punteros y Matrices

Las matrices, o arreglos de más de una dimensión, no tienen una equivalencia directa con los punteros.

```
int *ptr, v[5], a[3][3];

ptr = &v[1];        <<< OK
ptr = v;            <<< OK
ptr[4] = 8;         <<< OK

ptr = a;            <<< Error
ptr[1][1] = 8;      <<< Error

ptr = &a[0][0];     <<< OK
ptr[3*1+1] = 8;     <<< OK
```

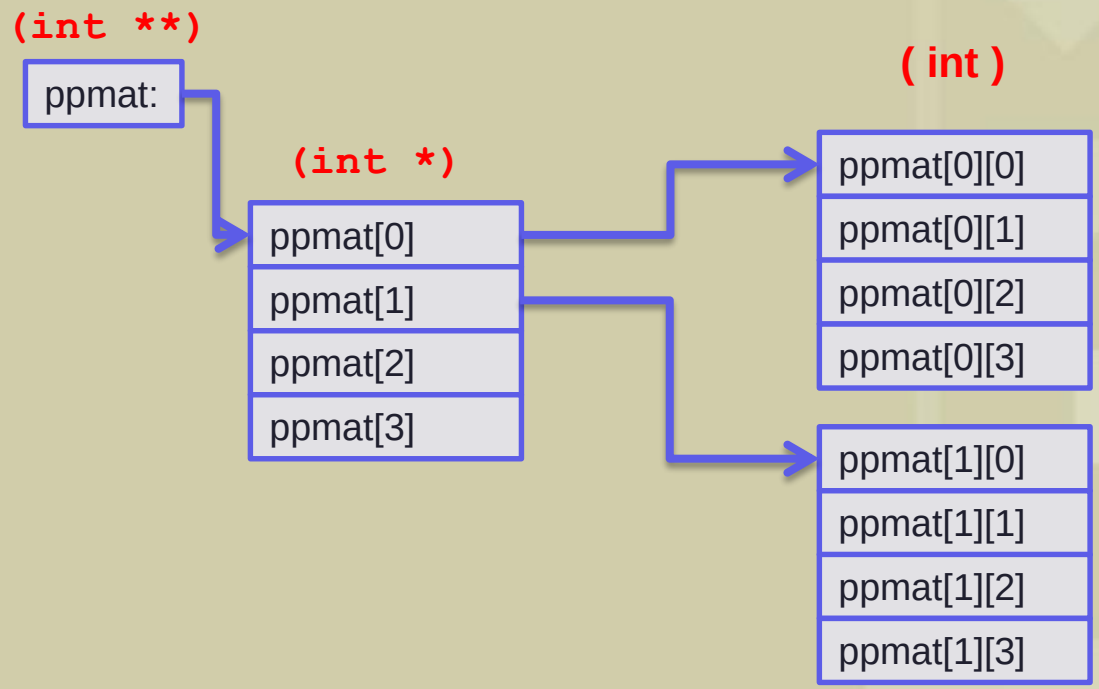
Punteros a Punteros a …

Las punteros pueden apuntar a su vez a otros punteros. De esta forma se puede imitar la sintaxis de los arreglos multidimensionales.

```
int ***ptr, t[3][3][3];

// Inicializando apropiadamente ptr podríamos
// usarlo como si fuera un arreglo similar a t

t[1][0][3] = 3;
ptr[1][0][3] = 8;

// Pero no podemos inicializarlo así
ptr = t;            <<< Error

// Ya que t es equivalente a un int *
```

Allocación dinámica de Matrices

```
int **ppmat;
```



Allocación dinámica de Matrices

```
#include <stdlib.h>
int ** allocaMatriz(int nfil, int ncol);
void liberaMatriz(int ** pmat, int nfil);
void imprimeMatriz(int** pmat,int nfil, int ncol);

int main() {
  int i,j,nf,nc, **ppmat;
  printf("ingrese el numero de filas"); scanf("%d",&nf);
  printf("ingrese el numero de columnas"); scanf("%d",&nc);

  ppmat = allocaMatriz(nf,nc);

  // uso la matriz como si fuera un arreglo bidimensional
  for( i = 0; i < nf; i++ )
    for( j = 0; j < nc; j++ )
        ppmat[i][j] = i*j;

  imprimeMatriz(pmat,nf,nc);
  liberaMatriz(ppmat,nf);

  return 0;
}
```

Allocación dinámica de Matrices

```
int ** allocaMatriz(int nfil, int ncol) {
   int i, **pmat;

   // pmat apunta a un arreglo de nfil punteros
   pmat = (int **) malloc(nfil * sizeof(int*));
   if( pmat == NULL )
      return NULL; // si falla retorno NULL como hace malloc

   // cada elemento de pmat apunta a un arreglo de ncol enteros
   for( i=0; i<nf; i++){
      pmat[i]=(int *) malloc(ncol * sizeof(int));
      if( pmat[i] == NULL ){
         liberaMatriz(pmat,i);
         return null;
      }
   }
   return pmat;
}
```

Allocación dinámica de Matrices

```
void liberaMatriz(int ** pmat, int nfil) {
   int i;

   // primero debo liberar los elementos de pmat
   for( i=0; i<nfil; i++)
      free(pmat[i]);

   // y finalmente libero pmat
   free(pmat);

   return;
}
```

Estructuras con Punteros

```
typedef struct {
    int grado;
    double *coefs;
} Polinomio_t;      // representa un polinomio
// los coeficientes represetan al polinomio:
// coef[grado]*x^grado +…+ coef[1]*x^(1)+coef[0]

Polinomio_t *PolAlloc( int grado );

void PolFree(Polinomio_t *pPol);

int PolSetCoef(Polinomio_t *pPol, int cIdx, double v);

double PolEval(Polinomio_t *pPol, double x);

Polinomio_t *PolDeriv(Polinomio_t *pPol);

Polinomio_t *PolInteg(Polinomio_t *pPol);
```

Estructuras – Ejemplos

```
Polinomio_t *PolAlloc( int grado )
{
    int i;
    // allocacion de la estructura
    Polinomio_t *pPol = (Polinomio_t *)
                malloc(sizeof(Polinomio_t));
    assert(pPol);

    // inicializacion de miembros
    pPol->grado = grado;
    pPol->coefs = (double *)
                  malloc((grado+1) * sizeof(double));
    assert(pPol->coefs);
    for( i = 0; i <= grado; i++ )
        pPol->coefs[i] = 0;
    return pPol;
}
```

Estructuras – Ejemplos

```
void PolFree(Polinomio_t *pPol)
{
    free(pPol->coefs);
    free(pPol);
}


int PolSetCoef(Polinomio_t *pPol, int cIdx, double v)
{
    if(cIdx >= 0 && cIdx <= pPol->grado) {
        pPol->coefs[cIdx] = v;
        return 1;
    }
    return 0;
}
```

Enumeraciones

- Una enumeración es un conjunto de constantes enteras.

```
enum ColorE {
    ROJO,
    VERDE,
    AZUL
};
```

- A la enumeración se le puede asignar un nombre y se comporta como tipo.

```
enum ColorE c1, c2, c3;
```

- Una variable de ese tipo de dato solo podrá contener los valores especificados en la enumeración (no estrictamente cierto en C)

```
c1 = VERDE;
```

Enumeraciones

```
enum Keyword {
    ASM,                    // => 0
    AUTO = 22,              // => 22
    BREAK                   // => 23
};

void f(enum Keyword key)
{
    switch(key) {
        case ASM:
            // do something
            break;
        case BREAK:
            // do something
            break;
        default:
            // do something  (→AUTO)
            break;
    }
}
```

Uniones

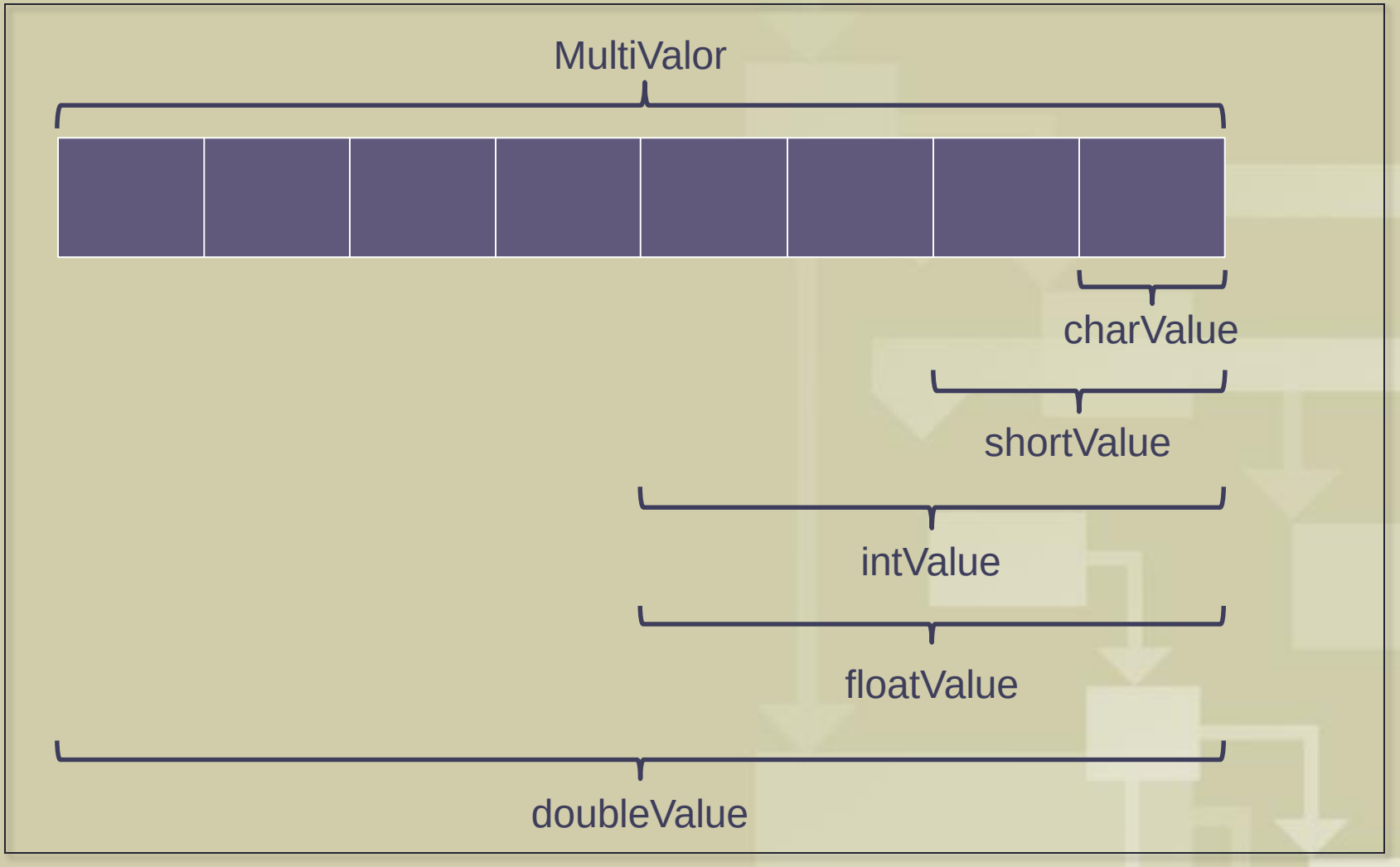
- Las uniones son similares a las estructuras, con la diferencia que en lugar de reservar lugar para todos los campos de la estructura, sólo se reserva espacio para el campo mas grande.

- Los campos comparten el espacio.

- Se utilizan cuando a un mismo espacio se le debe dar diferentes interpretaciones excluyentes entre sí.

```
typedef union {
    char   charValue;
    short  shortValue;
    int    intValue;
    float  floatValue;
    double doubleValue;
} MultiValor;
```

Uniones


MultiValor
charValue
shortValue
intValue
floatValue
doubleValue

Uniones

```
typedef enum {
    TYPE_CHAR,
    TYPE_SHORT,
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_DOUBLE
} ValueType;

typedef struct {
    ValueType  type;
    MultiValor valor;
} Variant_t;
```

```
void printVar(Variant_t *pV)
{
  switch(pV->type) {
    case TYPE_CHAR:
      printf("%c", pV->valor.charValue);
      break;
    case TYPE_SHORT:
      printf("%hd", pV->valor.shortValue);
      break;
    case TYPE_INT:
      printf("%d", pV->valor.intValue);
      break;
    case TYPE_FLOAT:
      printf("%f", pV->valor.floatValue);
      break;
    case TYPE_DOUBLE:
      printf("%lf", pV->valor.doubleValue);
      break;
    default:
      printf("UNKNOWN TYPE");
  }
}
```

Bit Fields

Los campos de bits permiten que miembros enteros de estructuras se puedan almacenar dentro de espacios de memoria mas pequeños que los que el compilador podría usar en forma ordinaria

Permiten que se pueda especificar el ancho (en bits) de cualquier miembro entero de una estructura.

Se utilizan frecuentemente en casos donde hay que forzar algún empaquetamiento especial, ya sea para ajustarlo a una representación de hardware específica o datos empaquetados sobre variables enteras.

Bit Fields

```
union flt {

    struct ieee754 {
        unsigned int mantisa    : 23;
        unsigned int exponente :  8;
        unsigned int signo      :  1;
    } raw;

    float f;
};
```

Bit Fields

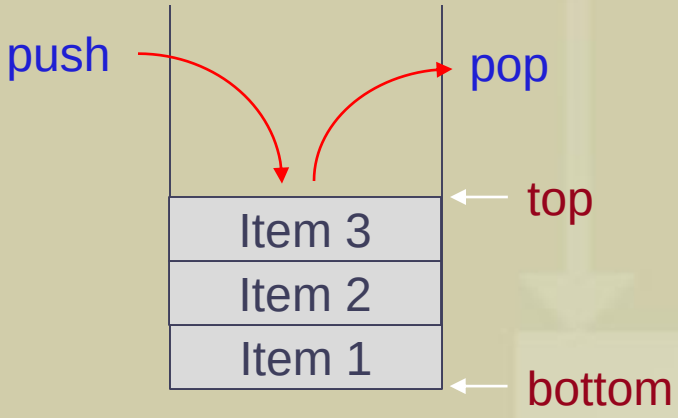
```
int main()
{
    union flt f;
    f.f = 4.2;

    printf("%d %d %x\n", f.raw.signo,
                         f.raw.exponente,
                         f.raw.mantisa);
    return 0;
}
```

Contenedores de datos

- Son estructuras que tienen la propiedad de contener datos.
- Existen distintos tipos de contenedores.
- El tipo define sus características y semántica de uso.
- Múltiples maneras de implementarlos.

Estructura de Pila (Stack)

- Un Stack es una estructura lineal ordenada de elementos, en donde las inserciones y remociones toman lugar en un mismo extremo. Implementa una política **L**ast **I**n **F**irst **O**ut (**LIFO**).



Estructura de Pila (Stack)

- Operaciones fundamentales sobre un stack ‘s’:

Operación	Semántica
`push(s,e)`	agrega un nuevo elemento al stack (tope del stack)
`pop(s)`	Retorna y remueve el elemento más nuevo disponible (tope del stack). No puede estar vacío.
`isEmpty(s)`	Retorna verdadero/falso indicando si el stack está o no vacío.

- Dependiendo de la implementación, pueden existir las operaciones **`create`** y **`destroy`**

Estructura de Pila (Stack)

- Es una estructura dinámica en donde la cantidad de elementos en un dado momento esta dado por la cantidad de operaciones de **`push`** menos la cantidad de operaciones de **`pop`** que se realizaron.
- Conceptualmente no hay límite para la cantidad de elementos que puede contener.
- La implementación puede poner límites.

Una implementación de stack

```
// representación  ********************
typedef struct {
    int top;
    int size;
    int *pElements;
} Stack_t;

// interfase ************************
// crea y retorna un nuevo stack de capacidad 'size'
Stack_t *StackCreate(int size);

// destruye un stack liberando reursos asociados
void StackDestroy(Stack_t *ps);

// agrega el elemento 'e' y retorna true/false(falta de lugar)
int StackPush(Stack_t *ps, int e);

// remueve elemento y lo coloca en *pe, retorna true/false(vacio)
int StackPop(Stack_t *ps, int *pe);

// chequea por stack vacio, retorna true/false
int StackIsEmpty(Stack_t *ps);
```

Una implementación de stack

```
...
#include "stack.h"

Stack_t *StackCreate(int size)
{
    Stack_t *ns = (Stack_t *) malloc(sizeof(Stack_t));
    assert(ns != NULL);

    ns->pElements = (int *) malloc(size * sizeof(int));
    assert(ns->pElements);

    ns->size = size;
    ns->top = 0;

    return ns;
}
void StackDestroy(Stack_t *ps)
{
    free(ps->pElements);

    free(ps);
}
```

Una implementación de stack

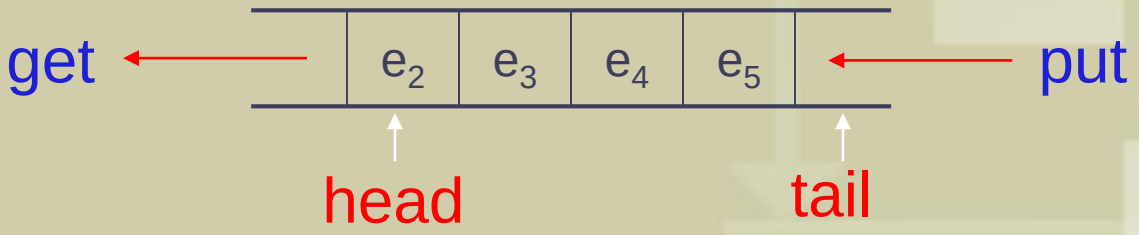
```
int StackPush(Stack_t *ps, int e)
{
    if(ps->top == ps->size)
        return 0;
    ps->pElements[ps->top++] = e;
    return 1;
}

int StackPop(Stack_t *ps, int *pe)
{
    if(ps->top == 0)
        return 0;
    *pe = ps->pElements[--ps->top];
    return 1;
}

int StackIsEmpty(Stack_t *ps)
{
    return ps->top == 0;
}
```

Uso del stack

```
...
#include "stack.h"

int main()
{
    Stack_t *ps = StackCreate(10);
    StackPush(ps, 1);
    StackPush(ps, 3);
    StackPush(ps, 5);
    StackPush(ps, 7);

    int e;
    while(!StackIsEmpty(ps)){
        int r = StackPop(ps, &e);
        assert(r);
        printf("-> %d\n", e);
    }
    StackDestroy(ps);

    return 0;
}
```

Estructura de Cola (Queue)

- Una Cola es una estructura lineal ordenada de elementos, en donde las inserciones y remociones toman lugar en extremos distintos. Implementa una política **F**irst **I**n **F**irst **O**ut (**FIFO**).



Estructura de Cola (Queue)

- Operaciones fundamentales sobre un cola ‘q’:

Operación	Semántica
`put(q,e)`	agrega un nuevo elemento a la cola
`get(q)`	Retorna y remueve el elemento más viejo disponible. No puede estar vacío.
`numElements(q)`	Retorna la cantidad de elementos que contiene la cola.

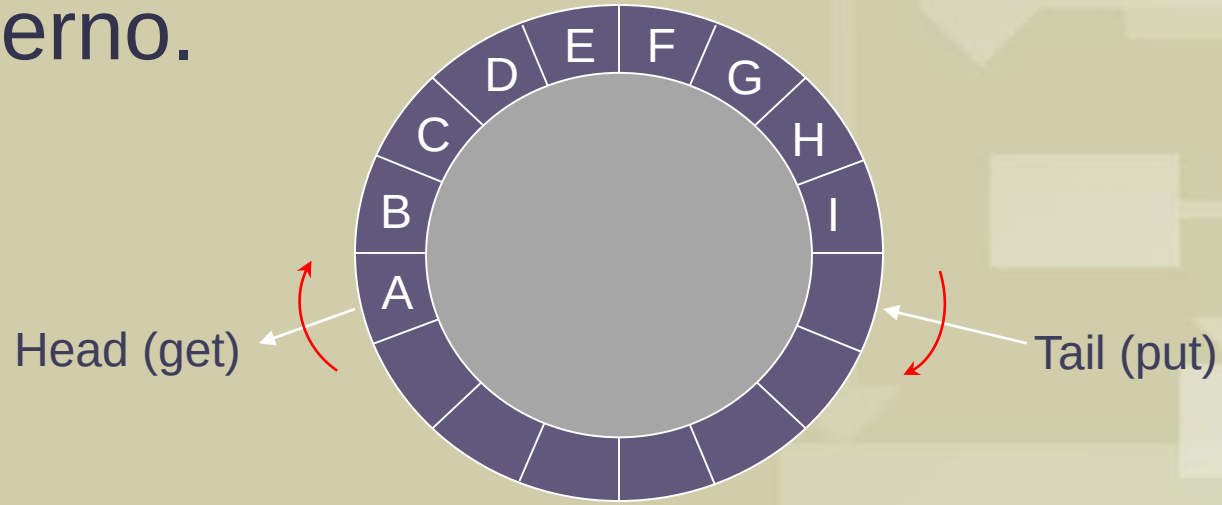
- Dependiendo de la implementación, pueden existir las operaciones **`create`** y **`destroy`**

Estructura de Cola (Queue)

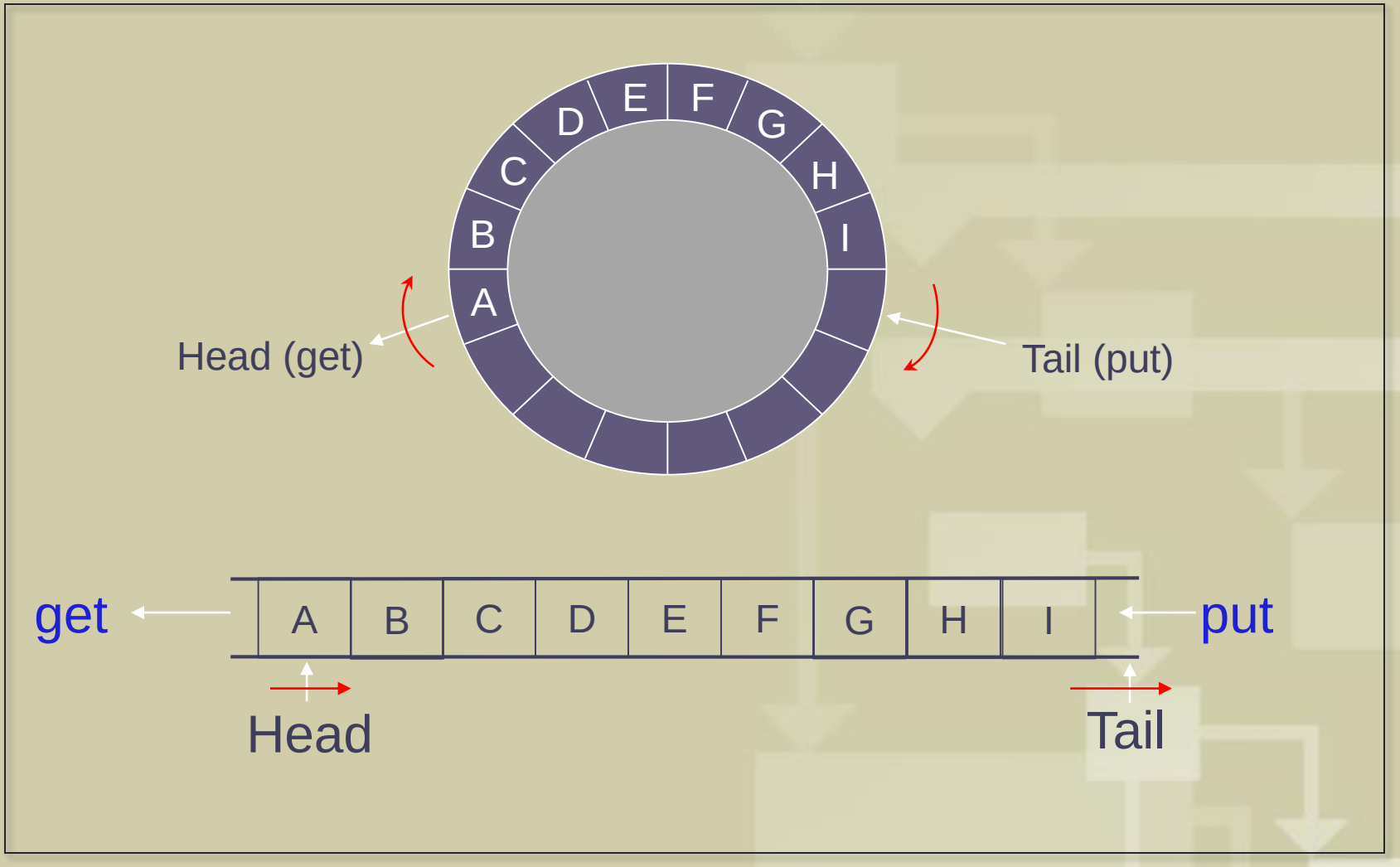
- Es una estructura dinámica en donde la cantidad de elementos en una dado momento esta dado por la cantidad de operaciones de **`put`** menos la cantidad de operaciones de **`get`** que se realizaron.
- Conceptualmente no hay límite para la cantidad de elementos que puede contener.
- La implementación puede poner límites.

Implementación de Colas

- Existen múltiples maneras de ser implementadas. Una comúnmente encontrada es la de utilizar un vector en forma circular como almacenamiento interno.



Implementación de Colas



Implementación de Colas

```
typedef struct {
    unsigned int tail;
    unsigned int head;
    unsigned int qSize;
    int *pElements;
} Queue_t;
```

Semánticas posibles para los campos **`tail`** y **`head`**:

1. Indices directos sobre **`pElements`**: Se pueden mantener a lo sumo **`(qSize – 1)`** elementos. En cada **`put/get`** hay que tener en cuenta la circularización de índices. El total de elementos presentes depende de las posiciones relativas de **`tail/head`**.
2. Contadores de **`put/get`** respectivamente: Se pueden mantener **`qSize`** elementos. Los indices sobre **`pElements`** se obtienen modulando **`tail/head`** con **`qSize`**. Total de elementos es **`(tail – head)`**

Conveniencia de utilizar una potencia de 2 para **`qSize`**.

Estructuras Auto-referenciadas

Estructuras recursivas

- Recordemos: Algoritmos recursivos para resolver un problema:
  - Una o más soluciones directas a problemas triviales (soluciones triviales).
  - Una o más soluciones que recurren a aplicar el mismo algoritmo sobre subproblemas del original.
- Similarmente se pueden definir tipos de datos que utilizan en su definición al tipo que se esta definiendo.

Listas

L = [ ] Lista Vacía

(e, L) element(e) + Lista (TAIL(L))

Funciones sobre Listas

NumNodos(L) =

- 0 si L == [ ]
- 1 + NumNodos(Tail(L))

EsMiembro(e, L) =

- falso si L == [ ]
- verdadero si L == (e, ?)
- EsMiembro(e, Tail(L))

InsOrden(e, L) =

- (e, L) si L == [ ] || (L == (x,?) && x>=e)
- (x, InsOrden(e, Tail(L)))

Implementación de Listas

```
typedef struct Node {
    int cont;
    struct Node *pNext;
} *List_t;        //  List_t es sinomimo de puntero a struct Node

#define LIST_EMPTY     NULL
#define LIST_CONT(lst) ((lst)->cont)
#define LIST_TAIL(lst) ((lst)->pNext)

int ListNumNodes(List_t lst){
    if(lst == LIST_EMPTY)
        return 0;
    return 1 + ListNumNodes(LIST_TAIL(lst));
}

//int ListNumNodes(List_t lst){
//    if(lst == NULL)
//        return 0;
//    return 1 + ListNumNodes(lst->pNext);
//}
```

NumNodos(L) =
- 0 si L == [ ]
- 1 + NumNodos(Tail(L))

Implementación de Listas

EsMiembro(e, L) =

- falso — si L == [ ]
- verdadero — si L == (e, ?)
- EsMiembro(e, Tail(L))

```
int ListIsMember(int e, List_t lst){
    if(lst == LIST_EMPTY)
        return 0;
    if(LIST_CONT(lst) == e)
        return 1;
    return ListIsMember(e, LIST_TAIL(lst));
}

// int ListIsMember(int e, List_t lst){
//     if(lst == NULL)
//         return 0;
//     if(lst->cont == e)
//         return 1;
//     return ListIsMember(e, lst->pNext);
// }
```

Implementación de Listas

InsOrden(e, L) =

- (e, L) si L == [ ] || (L == (x,?) && x>=e)
- (x, InsOrden(e, Tail(L)))

```
static List_t __ListNodeCreate(int e, List_t tail)
{
    List_t newL = (List_t ) malloc(sizeof(struct Node));
    assert(newL);
    LIST_CONT(newL) = e;
    LIST_TAIL(newL) = tail;
    return newL;
}

List_t ListInsert(List_t lst, int e)
{
    if(lst == LIST_EMPTY|| e <= LIST_CONT(lst) )
        return __ListNodeCreate(e, lst);
    LIST_TAIL(lst) = ListInsert(LIST_TAIL(lst), e);
    return lst;
}
```

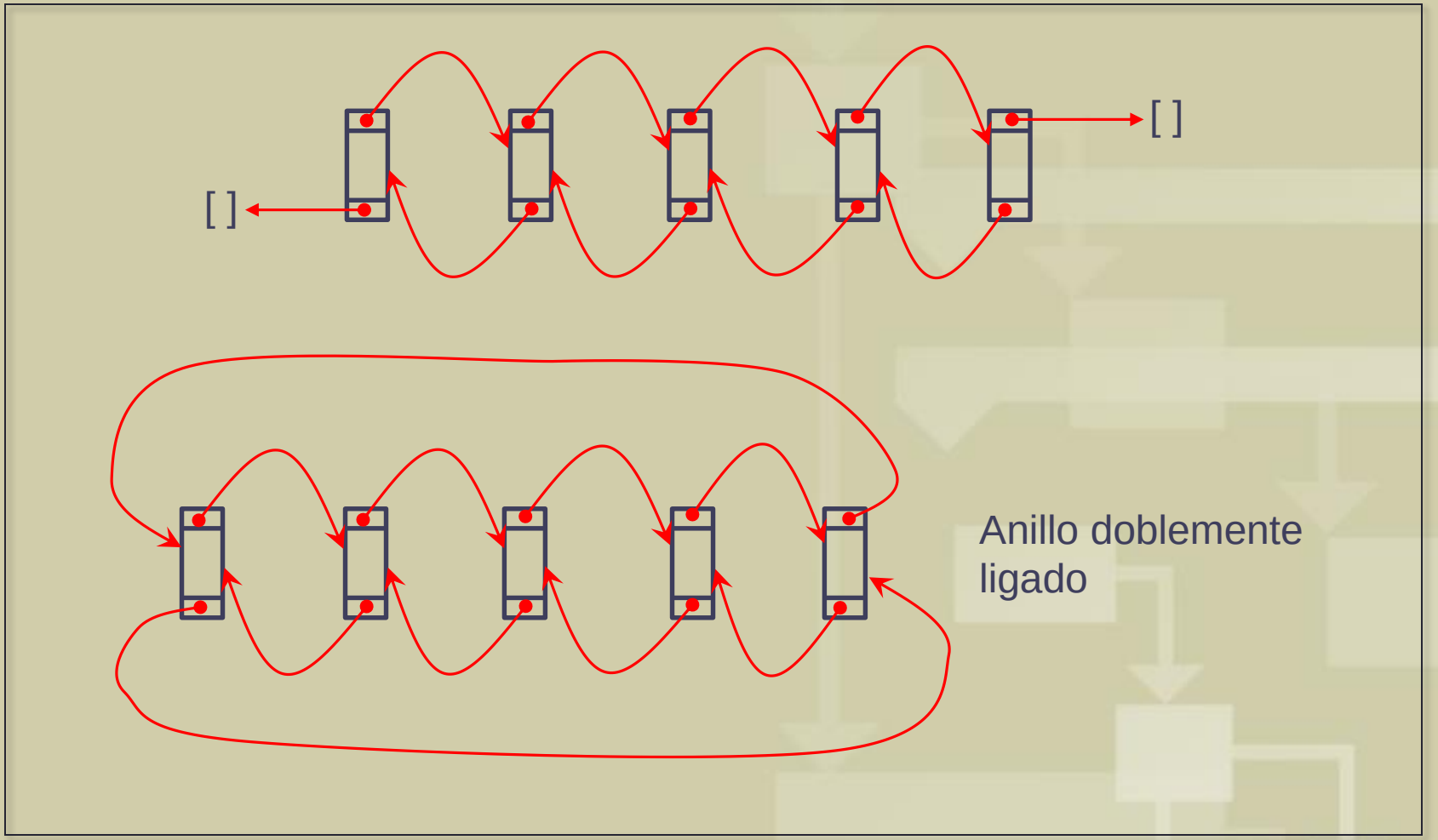
Implementación de Listas

```
// implementaciones no recursivas

int ListNumNodesNR(List_t lst)
{
    int nn = 0;
    while( lst != LIST_EMPTY ) {
        ++nn;
        lst = LIST_TAIL(lst);
    }
    return nn;
}

int ListIsMemberNR(int e, List_t lst)
{
    while( lst != LIST_EMPTY ) {
        if(LIST_CONT(lst) == e)
            return 1;
        lst = LIST_TAIL(lst);
    }
    return 0;
}
```

Implementación de Listas

```
void ListPrint(List_t lst)
{
    while(lst) {
        printf("%d ", LIST_CONT(lst));
        lst = LIST_TAIL(lst);  // lst = lst->pNext
    }
    puts("");
}

int main()
{
    List_t lst = LIST_EMPTY;  // = NULL

    lst = ListInsert(lst, 4);
    lst = ListInsert(lst, 5);
    lst = ListInsert(lst, 3);
    lst = ListInsert(lst, 1);
    ListPrint(lst);
    printf("Num Nodos: %d\n", ListNumNodes(lst));
    return 0;
}
```

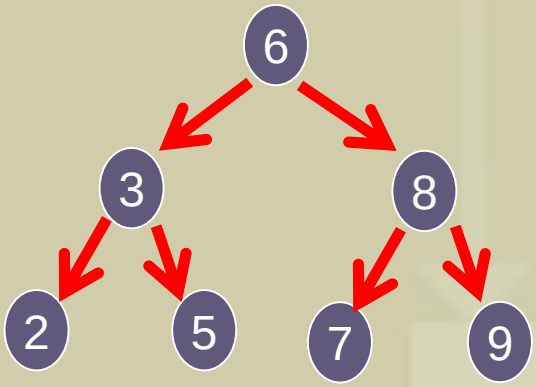
Listas doblemente ligadas



Arboles Binarios

$$A = \begin{cases} [\,] & \text{Árbol Vacío} \\ (e, Ai, Ad) & \text{elem+Árbol izq+Árbol der} \end{cases}$$

Diccionario binario == Árbol con orden



Funciones sobre Árboles

NumNodos(A) =

0	si A == [ ]
1 + NumNodos(Ai) + NumNodos(Ad)	si A == (e, Ai, Ad)

EsMiembro(e, A) =

falso	si A == [ ]
verdadero	si A == (e, ?, ?)
EsMiembro(e, A i)	si A=(x,Ai,Ad) && x > e
EsMiembro(e, A d)	si A=(x,Ai,Ad) && x<= e

InsOrden(e, A) =

(e, [ ], [ ])	si A == [ ]
(x, InsOrden(e, Ai) , Ad)	si A=(x,Ai,Ad) && x > e
(x, Ai, InsOrden(e, Ad))	si A=(x,Ai,Ad) && x <= e

Funciones sobre Árboles

TreeHeight(A) =

0	si A == [ ]
1 + max(TreeHeight(Ai), TreeHeight(Ad))	si A == (e, Ai, Ad)

TreePrint(A) =

Done	si A == [ ]
TreePrint(Ai); Print(e); TreePrint(Ad);	si A == (e, Ai, Ad)

Implementación de Árboles Binarios

```
typedef struct TNode {
    int cont;
    struct TNode *pLeft;
    struct TNode *pRight;
} *Tree_t;

#define TREE_EMPTY     NULL
#define TREE_CONT(t)    ((t)->cont)
#define TREE_LEFT(t)    ((t)->pLeft)
#define TREE_RIGHT(t)   ((t)->pRight)
```

NumNodos(A) =

0	si A == [ ]
1 + NumNodos(Ai) + NumNodos(Ad)	si A == (e, Ai, Ad)

```
// retorna la cantidad de nodos del árbol
int TreeNumNodes(Tree_t t)
{
    if(t == TREE_EMPTY)
        return 0;
    return 1 + TreeNumNodes(TREE_LEFT(t))
             + TreeNumNodes(TREE_RIGHT(t));
}
```

Implementación de Árboles Binarios

```
// retorna verdadero/falso indicando si e existe o nó dentro de t
int TreeIsMember(int e, Tree_t t)
{
    if(t == TREE_EMPTY)    // árbol vacío -> no existe
        return 0;
    if(TREE_CONT(t) == e)  // contenido a la cabeza -> existe
        return 1;
    if( TREE_CONT(t) > e)  // si está, esta en el lado izquierdo
        return TreeIsMember(e, TREE_LEFT(t));
    return TreeIsMember(e, TREE_RIGHT(t));  // si no en el derecho
}

int TreeHeight(Tree_t t)
{
    int hl, hr;
    if(t == TREE_EMPTY)
        return 0;
    hl = TreeHeight(TREE_LEFT(t));
    hr = TreeHeight(TREE_RIGHT(t));
    return 1 + ((hl > hr) ? hl : hr);
}
```

Implementación de Árboles Binarios

```
// función auxiliary para crear un árbol de un solo nodo
Tree_t TreeCreateNode(int e)
{
    Tree_t nT = (Tree_t) malloc(sizeof(struct TNode));
    assert(nT);
    TREE_CONT(nT) = e;
    TREE_LEFT(nT) = TREE_RIGHT(nT) = TREE_EMPTY;
    return nT;
}

// hace insersión ordenada en el árbol
Tree_t TreeInsertOrdered(int e, Tree_t t)
{
    if(t == TREE_EMPTY)
        return TreeCreateNode(e);
    if(TREE_CONT(t) >= e)
        TREE_LEFT(t) = TreeInsertOrdered(e, TREE_LEFT(t));
    else
        TREE_RIGHT(t) = TreeInsertOrdered(e, TREE_RIGHT(t));
    return t;
}
```

Implementación de Árboles Binarios

```
// Imprime el contenido del árbon en forma ordenada
void TreePrint(Tree_t t)
{
    if(t != TREE_EMPTY) {
        TreePrint(TREE_LEFT(t));
        printf("%d\n", TREE_CONT(t));
        TreePrint(TREE_RIGHT(t));
    }
}

// destruye el árbol, liberando recursos
void TreeDestroy(Tree_t t)
{
    if(t != TREE_EMPTY) {
        TreeDestroy(TREE_LEFT(t));
        TreeDestroy(TREE_RIGHT(t));
        free(t);
    }
}
```

Mecanismos de búsqueda

- Un algoritmo de búsqueda esta diseñado para localizar un elemento dentro de un conjunto de datos (por ejemplo en un contenedor).

- Búsqueda secuencial: se utiliza cuando el conjunto de datos no está ordenado. Consiste en ir haciendo una búsqueda secuencial del elemento en el conjunto. Orden N (Depende linealmente de la cantidad de elementos del conjunto.

- Búsqueda binaria: se utiliza cuando el conjunto de datos esta ordenado. Consiste en ir reduciendo el espacio de búsqueda de a mitades hasta encontrar el elemento a quedar con un espacio vacío. Orden log(N) (Todavía depende de N)

- Existe algún mecanismo de búsqueda que no dependa de N?

Funciones de Hash

- Se denomina función de Hash a una función que aplicada a una clave de búsqueda termina en un número entero.

Clave	Hash(Clave)
Jorge Alberto Mengano Gutiérrez	0x221AD7F1
Carla Gonzalez	0xF12AC027
Dionisio Ramírez	0x0234E962
Ana	0xE2103274

- Propiedades buscadas:
  - Bajo costo: computacional, memoria.
  - Determinista: para la misma clave, da el mismo valor de hash.
  - Compresión: Mapear dominios grandes a dominios pequeños.
  - Uniforme: distribuye uniformemente los valores de hash en su rango.

Búsquedas con Tablas de Hash

Es un mecanismo muy eficiente de búsqueda utilizando tablas prearmadas aplicando funciones de hash sobre las claves de búsqueda.

Conjunto de datos

	K	Atributos
0	K_0	$Attr_0$
1	K_1	$Attr_1$
2	K_2	$Attr_2$
3	K_3	$Attr_3$
.		
N-1	K_{N-1}	$Attr_N$

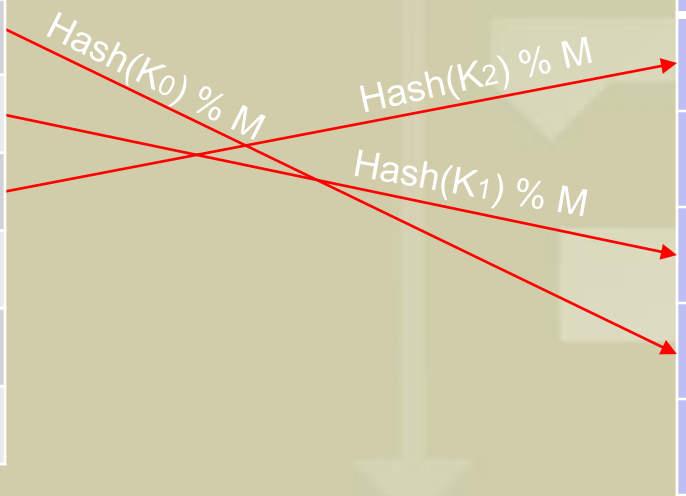


Tabla de Hash

	Orden
0	2
1	
2	1
3	0
.	
M-1	

Búsquedas con Tablas de Hash

Pueden ocurrir colisiones: para dos o mas claves distintas, terminan el la misma entrada de la tabla de hash.

Conjunto de datos

	K	Atributos
0	K_0	$Attr_0$
1	K_1	$Attr_1$
2	K_2	$Attr_2$
3	K_3	$Attr_3$
.		
N-1	K_{N-1}	$Attr_N$

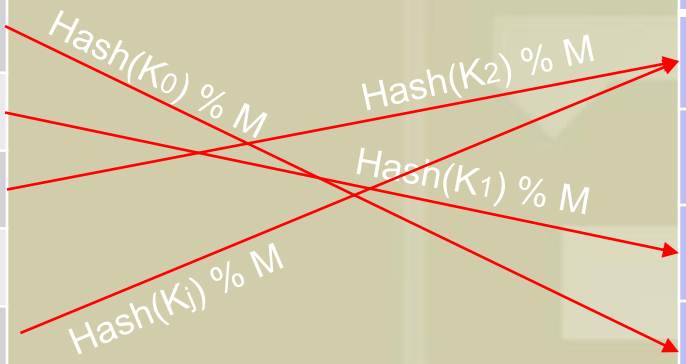
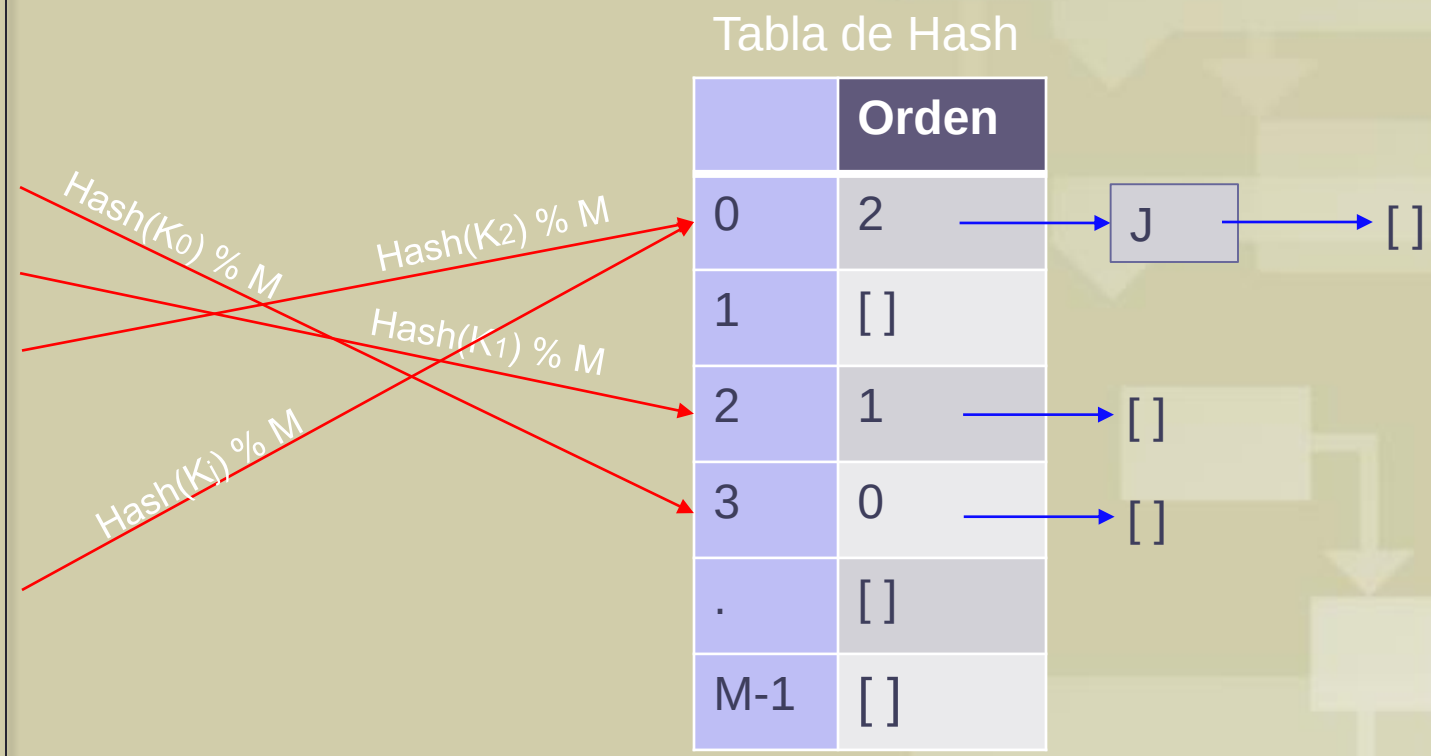


Tabla de Hash

	Orden
0	2
1	
2	1
3	0
.	
M-1	

Resolución de colisiones

Existen múltiples mecanismos de resolución de colisiones: Lista ligada de colisiones:



Resolución de colisiones

Existen múltiples mecanismos de resolución de colisiones: Siguiente libre:

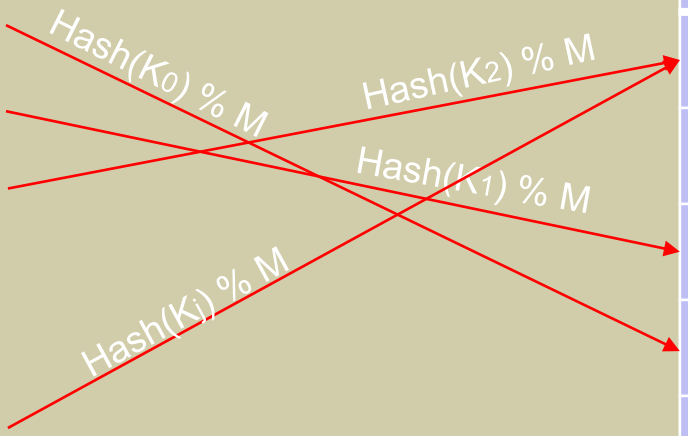


Tabla de Hash

	Orden
0	2
1	J
2	1
3	
.	
M-1	

FIN