



Procesos y Comunicación Interprocesos.

Introducción al lenguaje C

Dr. Arturo Yee Rendón

UNIVERSIDAD AUTÓNOMA DE SINALOA
Facultad de Informática Culiacán



Introducción al Lenguaje C



Marco Teórico

- La primera estandarización del **lenguaje C** fue en ANSI, con el estándar X3.159-1989. El lenguaje que **define** este estándar fue conocido vulgarmente como **ANSI C**.
- En 1990, fue ratificado como estándar **ISO**(ISO/IEC 9899:1990). La adopción de este **estándar** es muy amplia por lo que, si los **programas** creados lo siguen, el **código** es **portable** entre **plataformas** y/o **arquitecturas**



Características

- Programación estructurada
- Economía de las expresiones
- Abundancia en operadores y tipos de datos
- Codificación de alto y bajo nivel simultáneamente



Características

- Utilización natural de las funciones primitivas del sistema
- Reemplaza ventajosamente a la programación en ensamblador
- No está orientado a ninguna área especial
- Producción de código objeto altamente optimizado



Fases de desarrollo de un programa en C

- El preprocesador
 - Transforma el programa fuente, convirtiéndolo en otro archivo fuente “predigerido”. Las transformaciones incluyen:
 - Eliminar los comentarios.
 - Incluir en el fuente el contenido de los archivos declarados con `#include <archivo.h>` (a estos archivos se les suele llamar cabeceras)
 - Sustituir en el fuente las macros declaradas con `#define`(ej. `#define CIEN 100`)



Fases de desarrollo de un programa en C

- El compilador
 - Convierte el fuente entregado por el preprocesador en un archivo en lenguaje máquina: archivo objeto.
 - Algunos compiladores pasan por una fase intermedia en lenguaje ensamblador

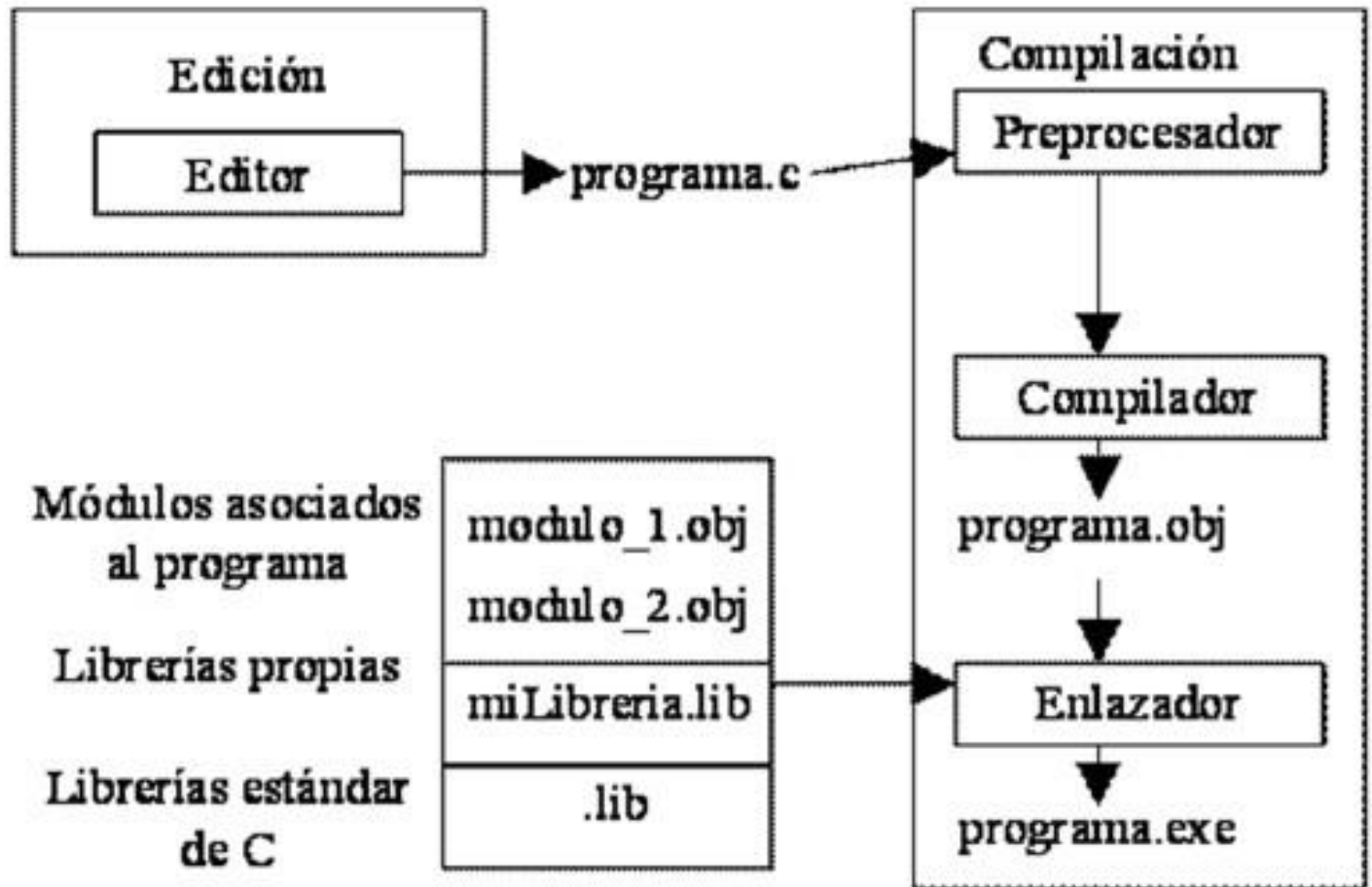


Fases de desarrollo de un programa en C

- El enlazador
 - Un archivo **objeto** es código **máquina**, pero no se puede **ejecutar**, porque le falta **código** que se encuentra en otros **archivos binarios**.
 - El enlazador **genera** el **ejecutable binario**, a partir del contenido de los archivos objetos y de las bibliotecas. Las bibliotecas contienen el **código** de funciones **precompiladas**, a las que el archivo fuente llama (por ejemplo **printf**).



Fases de desarrollo de un programa en C





Ejemplo de programa en C

Los componentes típicos de un archivo fuente del programa son:

- El **archivo** comienza con algunos **comentarios** que describen el propósito del **módulo** e **información** adicional tal como el nombre del autor y fecha, nombre del archivo. Los comentarios comienzan con **/*** y terminan con ***/**.
- **Ordenes** al **preprocesador**, conocidas como **directivas** del preprocesador. Normalmente **incluyen** archivos de **cabecera** y **definición** de **constantes**.



Ejemplo de programa en C

- **Declaraciones** de **variables** y **funciones** son visibles en todo el archivo. En otras palabras, los nombres de estas variables y funciones se pueden utilizar en cualquiera de las funciones de este archivo.
- El resto del archivo incluye **definiciones** de las **funciones** (su cuerpo). Dentro de un cuerpo de una función se pueden definir **variables** que son **locales** a la función y que sólo existe en el código de la función que se está **ejecutando**.



Ejemplo de programa en C

```
/* Mi primer programa escrito en C
Fecha:**-***-***/
#include <stdio.h>
void main()
{
    /* Escribe un mensaje */
    printf ("Hola, mundo\n");
}
```



Ejemplo de programa en C

- Compilación en linux con gcc

```
$ gcc prog.c -o prog
```

- Ejecución

```
$ ./prog
```



Ejemplo de programa en C

- Compilación en linux con gcc y biblioteca matemática

```
$ gcc prog.c -o prog -lm
```

- Ejecución
- \$./prog



Palabras reservadas ANSI C

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



Archivos de cabecera

- Directivas tales como `#include <stdio.h>` indica al compilador que lea el archivo `stdio.h` de modo que sus líneas se sitúan en la posición de la directiva. C ANSI soporta dos formatos para la directiva `#include`:
 1. `#include <stdio.h>`
 2. `#include "demo.h"`
- El primer formato de `#include`, lee el contenido de un archivo - el archivo estándar de C, `stdio.h` . El segundo formato, visualiza el nombre del archivo encerrado entre las dobles comillas que está en el directorio actual.



Definiciones de macros

Una **macro define** un símbolo **equivalente** a una **parte de código** C y se utiliza para ello la directiva **#define**.

Se puede **representar constantes** tales como PI, IVA y BUFFER o **cuadrado(x)**.

```
#define PI 3.14159
```

```
#define IVA 16
```

```
#define DOLAR 12.50
```

```
#define cuadrado(x)  $x*x$ 
```



Comentarios

- El **compilador ignora** los **comentarios** encerrados entre los símbolos **/*** y ***/**.

/* Mi primer programa ***/**

O bien

// Mi primer programa

- Se pueden escribir **comentarios multilínea**

/* Mi segundo programa C
escrito el día ...
en Culiacán Sinaloa, México***/**

- Los **comentarios no pueden anidarse**



Tipos y declaración de variables



Tipos de dato numérico

Enteros		
Nombre	Rango	Tamaño
Int	-2,147,483,648 a 2,147,483,647	4 bytes
unsigned int	0 a 4294967295	4 bytes
short	-32768 a 32767	2 bytes
unsigned short	0 a 65,535	2 bytes
long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	8 bytes
unsigned long	0 a 18,446,744,073,709,551,616	8 bytes



Tipos de dato numérico

Reales

Nombre	Rango	Tamaño
float	10^{-37} a 10^{37}	4 bytes
double	10^{-308} a 10^{308}	8 bytes

Carácter

Nombre	Rango	Tamaño
char	0 a 255	1 byte



Constantes numéricas

- Todos los números sin un punto decimal en programas C se tratan como enteros y todos los números con un punto decimal se consideran reales de coma flotante de doble precisión.
- Si se desea representar números en base 16 (hexadecimal) o en base 8 (octal), se precede al número con el carácter 'OX' para hexadecimal y 'O' para octal. Si se desea especificar que un valor entero se almacena como un entero largo se debe seguir con una "L"
- 025 /* octal 25 o decimal 21 */
- 0x25 /* hexadecimal 25 o decimal 37 */
- 250L /* entero largo 250 */



Tipos `char`

- El tipo `char` se utiliza para representar caracteres o valores enteros.
- Las constantes de tipo `char` pueden ser caracteres encerrados entre comillas ('A', 'b', 'p').
- Caracteres no imprimibles (tabulación, avance de página...) se pueden representar con secuencias de escape ('\t', '\f')



Secuencias de escape

Carácter	Significado	Código ASCII
\a	Carácter de alerta (timbre)	7
\b	Retroceso de espacio	8
\f	Avance de página	12
\n	Nueva línea	10
\r	Retorno de carro	13
\t	Tabulación	9
\\	Barra inclinada	92
\?	Signo de interrogación	63
\'	Comilla	39
\"	Doble comilla	34
\0	Carácter nulo (terminación de (cadena)	-



Variables

- Todas las **variables** en C se **declaran** o **definen** antes de que sean **utilizadas**. Una declaración indica el **tipo** de una **variable**.
- Si la **declaración** también almacena un **valor** (se inicia), entonces en una **definición**.



Identificadores

- un **identificadores** un **nombre** que define a una **variable**, una **función** o un tipo de **datos**.
- Un identificador válido ha de **empezar** por una **letra** o por el **carácter** de **subrayado** `_`, seguido de cualquier **cantidad** de **letras**, **dígitos** o **subrayados**.
- Se **distinguen** **mayúsculas** de **minúsculas**, no se pueden utilizar **palabras reservadas**



Declaración de variables

Ejemplos válidos

- `char letra;`
- `int Letra;`
- `float CHAR;`
- `int __variable__;`
- `int cantidad_envases;`
- `double precio123;`
- `int __;`



Declaración de variables

Ejemplos no válidos:

- `int 123var; /* Empieza por dígitos */`
- `char int; /* Palabra reservada */`
- `int una sola; /* Contiene espacios */`
- `int US$; /* Contiene $ */`
- `int var.nueva; /* Contiene el punto */`
- `int eñe; /* Puede no funcionar */`



Declaración de variables

```
int si =1, no= 0; //definen las variables si y no  
float total = 42.125 //define la variable total
```

Se pueden declarar variables múltiples del mismo tipo de dos formas

```
int v1; int v2; int v3;
```

o bien

```
int v1;  
int v2;  
int v3;
```

Pudiéndose declarar también de la forma siguiente:

```
int v1, v2,v3;
```

o con valores iniciales como:

```
int v1=1, v2=2, v3=3;
```



Declaración de **variables**

- C, no soporta tipos de datos lógicos, pero mediante enteros se pueden representar: 0, significa falso; distinto de cero, significa verdadero (cierto).
- La palabra reservada **const** permite definir determinadas variables con valores constantes, que no se pueden modificar. Así, si se declara

`const int z = 4350`

y si se trata de modificar su valor

`z = 3475`

el compilador emite un mensaje de error similar a
“**asignación de la variable de sólo lectura**”.



Variables tipo `char`

- Las variables de tipo `char` (carácter) pueden almacenar **caracteres individuales**. Por ejemplo la definición

```
char car = 'M';
```

declara una variable `car` y le asigna el valor **ASCII** del **carácter** `M`. El compilador convierte la constante carácter `'M'` en un valor entero (`int`), igual al código ASCII de `'M'` que se almacena a continuación en el byte reservado para `car`.

- Dado que los **caracteres literales** se almacenan **internamente** como valores **`int`**, se puede cambiar la línea.

```
char car;
```

por

```
int car;
```

y el programa **funcionará correctamente**



Operadores



Expresiones y operadores

Las expresiones son operaciones que realiza el programa.

$$a+b+c$$



Operadores aritméticos

Operador	Descripción	Ejemplo
+	Suma	$(a+b)$
-	Resta	$(a-b)$
*	Multiplicación	$(a*b)$
/	División	(a/b)



Operadores relacionales

Operador	Descripción	Ejemplo
<	Menor que	$(a < b)$
<=	Menor que o igual	$(a \leq b)$
>	Mayor que	$(a > b)$
>=	Mayor o igual que	$(a \geq b)$
=	Igual	$(a = b)$
!=	No igual	$(a \neq b)$



Operadores de incremento y decremento

Operador	Descripción	Ejemplo
++	Incremento en i	++i, i++
--	Decremento en i	--i, i--



Operadores de asignación

Operador	Descripción	Ejemplo
=	Operación de asignación simple	<code>a = b;</code>
+=	<code>z += 4;</code>	<i>equivale a <code>z = z + 4;</code></i>
-=	<code>z -= 4;</code>	<i>equivale a <code>z = z -4;</code></i>
* =	<code>z * = 10;</code>	<i>equivale a <code>z = z * 10;</code></i>
/ =	<code>z / = 5;</code>	<i>equivale a <code>z = z / 5;</code></i>
%=	<code>z % = 2;</code>	<i>equivale a <code>z = z % 2</code></i>



Operaciones de entrada y salida



Funciones de entrada y salida

- Las funciones `printf()` y `scanf()` permiten comunicarse con un programa. Se denominan funciones de E/S.
- `printf()` es una función de **salida** y `scanf()` es una función de **entrada** y ambas utilizan una cadena de control y una lista de argumentos.



Función printf()

- La función `printf()` es la función de salida con formato. Esta función se puede utilizar para mostrar números, caracteres o cadena de caracteres con formato, desde un programa

`printf` ("cadena de control", exp1, exp2,...expN);



Función printf()

- La cadena de control tiene tres componentes: texto, identificadores y secuencias de escape.
- Se puede utilizar cualquier texto y cualquier número de secuencias de escape.
- El número de identificadores ha de corresponder con el número de variables o valores a escribir.
- Los identificadores de la cadena de formato determinan como se escriben cada uno de los argumentos.
- cada identificador comienza con un signo porcentaje (%) y un código que indica el formato de salida de la variable.



Función printf()

Identificador	Formato
%d	Entero decimal
%c	Carácter simple
%s	Cadena de caracteres
%f	Coma flotante (decimal)
%e	Coma flotante (notación exponencial)
%u	Entero decimal sin signo
%o	Entero octal sin signo
%x	Entero hexadecimal sin signo



Función printf()

Ejemplo:

```
void main(){  
    Int var1=10;  
    float var2=20;  
    printf ("var1 es %d y el var2 es %f\n", var1, var2);  
}
```



Función scanf()

- La función **scanf()** es la función de entrada con formato. Esta función se puede utilizar para introducir números con formato, caracteres o cadena de caracteres, a un programa

scanf("cadena de control", var1, var2,...varN);

- El formato general de la función **scanf()** es una cadena de control y uno o más direcciones de variables de entrada. La cadena de control consta sólo de identificadores.



Función scanf()

Identificador	Formato
%d	Entero decimal
%c	Carácter simple
%s	Cadena de caracteres
%f	Coma flotante
%u	Entero decimal sin signo
%o	Entero octal sin signo
%x	Entero hexadecimal sin signo



Función scanf()

Ejemplo:

```
void main(){
```

```
    Int var1=10;
```

```
    float  var2=20;
```

```
    printf("Escriba el valor de var1:");
```

```
    scanf("%d", &var1);
```

```
    printf("Escriba el valor de var2:");
```

```
    scanf("%f", &var2);
```

```
    printf ("var1 es %d y  el var2 es %f\n", var1, var2);
```

```
}
```



Ejemplo

```
#include <stdio.h>
#define PI 1.1416

void main(int argc, char *argv[]){
    // Declaración de variables
    float radio, perimetro, area;

    // Solitud y entrada de datos
    printf("Escriba el radio del circulo:");
    scanf("%f", &radio);

    // Calculo de perímetro y área
    perimetro = 2*PI*radio;
    area= PI*radio*radio;

    // Salida de resultados
    printf("El perímetro es: %f \n",perimetro);
    printf("El área es: %f \n", area);
}
```



Ejercicios

1. Escribir un programa (es decir una función main) que pida por teclado una serie de números enteros, los almacene en una tabla estática y posteriormente escriba por pantalla todos los números introducidos indicando además cual es el mayor y el menor. Lo primero que debe hacer el programa es preguntar al usuario cuantos números se van a introducir y comprobar que dicha cantidad es menor que la dimensión de la tabla. Para dicha dimensión debe definirse una constante (por ejemplo MAX_DIMENSION) a la cual se puede asignar un valor razonable (por ejemplo, 100).
2. Construya un programa en que dado el radio de una esfera determine su área y volumen.



Tarea

1. Implemente un programa que al recibir como dato la dimensión del lado de un hexaedro, calcule el área de la base el área lateral, el área total y el volumen.
2. Escribe un programa que dada las dimensiones de un cilindro hueco (radio interior, radio exterior y altura) determine la superficie y el volumen.
3. Desarrolle un programa que al recibir las coordenadas de los vértices de un triangulo determine su área.



Estructuras de control



Introducción

- Lenguaje C cuenta con ciertas **estructuras** que permitan **controlar el flujo** del programa, es decir, **tomar decisiones** y **repetir acciones** donde están involucradas **instrucciones**.
- Una **instrucción** consta de **palabras reservadas**, **expresiones** y otras **instrucción**. cada instrucción termina con un punto y coma (;).
- Un tipo especial de instrucción, la **instrucción compuesta** o **bloque**, es un grupo de **sentencias encerradas** entre **llaves** ({...}). El cuerpo de una función es una instrucción **compuesta**. Una instrucción **compuesta** puede tener **variables locales**.



Estructura de control **if**

- Simple

if (expresión) sentencia;

o bien

- Compuesta

```
if (expresión) {  
    instrucción1;  
    instrucción2;
```

...

```
}
```



Estructura de control if-else

if (expresión)

instrucción1;

else

instrucción2;

o bien compuesta

if (expresión){

instrucción1;

instrucción2;

}

else{

instrucción3;

instrucción4;

}



Estructura de control **if-else** (Ejemplo)

```
// Comparación de números  
#include <stdio.h>
```

```
void main(){  
    int a, b;
```

```
    printf("Valor 1: "); scanf("%d", &a);  
    printf("Valor 2: "); scanf("%d", &b);
```

```
    if (a>b)  
        printf ("El mayor es %d\n ", a);  
    else  
        printf("El mayor es %d\n", b);
```

```
}
```



Estructura de control **if anidadas**

if (expresión1)

 instruccion1;

else if (expresión2)

 instruccion2;

else

 instruccion3;



Estructura de control **if anidadas** (Ejemplo)

```
// Comparación de numeros  
#include <stdio.h>
```

```
void main(){  
    int a, b;  
  
    printf("Valor 1: "); scanf("%d", &a);  
    printf("Valor 2: "); scanf("%d", &b);  
  
    if (a>b)  
        printf ("El mayor es %d\n ", a);  
    else if(a<b)  
        printf("El mayor es %d\n", b);  
    else  
        printf("Los valores son %d\n", a) ;  
}
```




Expresión condicional (?:)

Expresión condicional es una simplificación de una instrucción if-else. Su sintaxis es:

expresión1 ? expresión2 : expresión3;

que equivale a

```
if (expresión1)
    expresión2;
else
    expresión3;
```



Expresión condicional (Ejemplo)

```
// Comparación de numeros
```

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

```
void main(){
```

```
    int a, b, c;
```

```
    printf("Valor 1 y 2: ");
```

```
    scanf("%d %d", &a, &b);
```

```
    c = a > b ? a : b;
```

```
    printf("El mayor es %d\n", c);
```

```
}
```



Estructura de control **switch**

```
switch (expresión) {  
  case valor1:  
    instruccion1; /*se ejecuta si expresión igual a valor1 */  
    break; /* salida de sentencia switch */  
  case valor2:  
    instruccion2;  
    break;  
  case valor3:  
    instruccion3;  
    break;  
  ...  
  default:  
    instruccion4; /* se ejecuta si ningún valor coincide  
                  con expresión */  
}
```



Estructura de control `switch` (Ejemplo)

```
// Calculadora basica
#include <stdio.h>
void main(){
int a, b;
char op;
printf ("Escriba expresión (ej. 5 + 6 ) =");
scanf("%d %c %d", &a, &op, &b);

switch (op){
    case '+':
        printf("%d \n \a", a+b);
        break;
    case '-':
        printf("%d \n \a", a-b);
        break;
    case '*':
        printf("%d \n \a", a*b);
        break;
    case '/':
        printf("%d \n \a", a/b);
        break;
    default :
        printf ("operación  errónea\n \a");
}
}
```



Estructura de control **while**

```
while (expresión)  
    instruccion;
```

o bien

```
while (expresión) {  
    instruccion1;  
    instruccion2;  
    ...  
}
```



Estructura de control **while**

```
int i = 0;  
while (i < 10) {  
    printf("El valor de i es %d\n", i);  
    i++;  
}
```

0 1 2 3 4 5 6 7 8 9



Estructura de control **while** (Ejemplo)

```
// Conversión de dólares a pesos
```

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

```
#define tipoCambioCompra 12.55
```

```
#define tipoCambioVenta 12.75
```

```
void main(){
```

```
int dolares=1;
```

```
float pesosCompra, pesosVenta;
```

```
printf("Dolares   Compra   Venta \n");
```

```
while(dolares<=10){
```

```
    pesosCompra=tipoCambioCompra*dolares;
```

```
    pesosVenta=tipoCambioVenta*dolares;
```

```
    printf("%4d  %10.2f %10.2f \n", dolares, pesosCompra, pesosVenta);
```

```
    dolares++;
```

```
}
```

```
}
```



Estructura de control **do-while**

```
do {  
    instruccion;  
}while (expresión);
```

o bien

```
do {  
    instruccion1;  
    instruccion2;  
    ...  
}while (expresión);
```




Estructura de control **do-while**

```
int i = 0;  
do {  
    printf("El valor de i es %d\n", i);  
    i++;  
} while (i < 10);
```

0 1 2 3 4 5 6 7 8 9



Estructura de control **do-while** (Ejemplo)

// Tabla de multiplicar

#include <stdio.h>

```
void main(){
```

```
    int i=1,N;
```

```
    printf("Escriba numero de tabla:");
```

```
    scanf("%d", &N);
```

```
    do{
```

```
        printf("%4d x %4d = %4d \n", i, N, i*N);
```

```
        i++;
```

```
    }while (i<=10);
```

```
}
```



Estructura de control **for**

```
for (expresión1; expresión2; expresión3)  
    instruccion;
```

O bien

```
for (expresión1; expresión2; expresión3) {  
    instruccion1;  
    instruccion2;  
    ...  
}
```



Instrucción **for**

- La información de actualización está al principio del ciclo

```
for ( i = 0; i < 10; i++) {  
    printf("El valor de i es %d\n", i);  
}
```

0 1 2 3 4 5 6 7 8 9

- Un ciclo for puede iterar varios valores

```
for (i = 0, j = 0; ... ; i++, j++)
```



Estructura de control **for** (Ejemplo)

```
/* Conversión de temperaturas */  
#include <stdio.h>  
main()  
{  
    int fahrenheit;  
    float celsius;  
  
    printf("Fahrenheit Celsius\n");  
    for (fahrenheit=0; fahrenheit<=300; fahrenheit+=20) {  
        celsius=(5.0/9.0)*(fahrenheit-32);  
        printf("%9d %8.2f \n",fahrenheit, celsius);  
    }  
}
```



Ejercicios

1. Dados dos valores enteros (a, b) determinar el valor de la **división entera** y el **resto** de la **división entera** de los mismos (a, b), sin usar los **operadores** de **división entera** ó el de **residuo**.
2. Realizar un **algoritmo** que lea 3 **números** diferentes y determine el **número medio** del **conjunto** de los 3 **números**.
3. Escriba un **programa** que lea una **hora** (hora, minutos, segundos) y diga la hora que será un **segundo después**.



Ejercicios

4. Desarrolle un programa que dados **a** y **b** enteros, informar el producto de **ambos** por **sumas sucesivas**.
5. Realizar un programa que lea 2 **números** x y n y calcule la suma:
$$1 + x^1 + x^2 + x^3 + x^n$$
6. Calcular el número de **combinaciones** de n **elementos** tomando r a la vez
7. Dado un capital **inicial**, a un determinado **porcentaje** de **intereses anual** determine en cuantos meses este **capital** se **duplicara**, suponiendo que el capital y los **interese** generados son **reinvertido**.



Estructuras de datos



Introducción

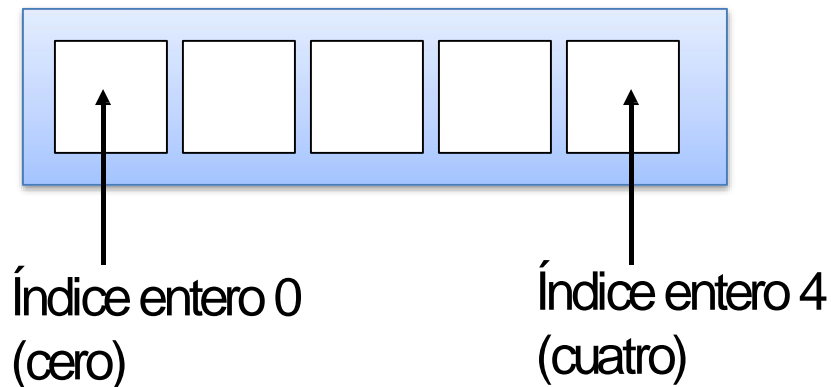
- Los tipos complejos o estructuras de datos en C son:

vectores, estructuras, uniones, cadenas y campos de bits.



Vectores

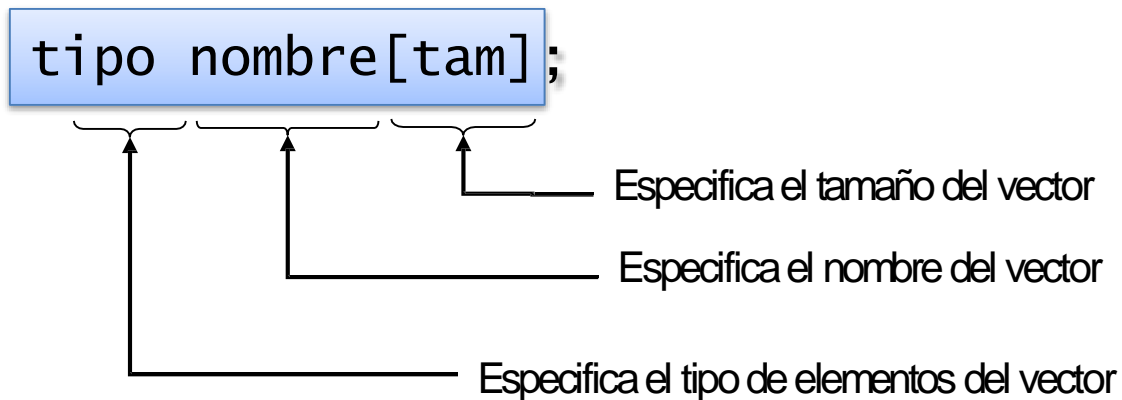
- un vector es una secuencia de elementos
 - Todos los elementos de un vector son del mismo tipo
 - Las estructuras pueden tener elementos de distintos tipos
 - Se accede a elementos individuales usando índices enteros





Declaración de Vectores

- Una variable vector se declara especificando:
 - El tipo de elementos de la vector
 - El nombre de la variable
 - El tamaño de la vector



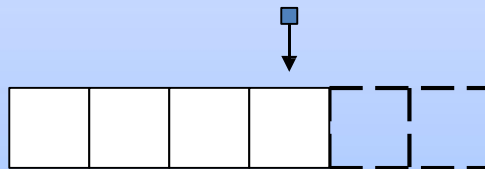


Dimensión de un **vector**

- El rango se conoce también como dimensión de la vector
- El número de índices asociados con cada elemento

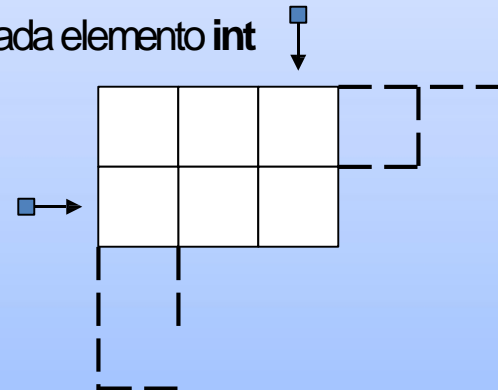
```
int fila[10];
```

Unidimensional
Un solo índice asociado con
cada elemento **int**



```
int cuadrícula[10][5];
```

Bidimensional
Dos índices asociados con
cada elemento **int**





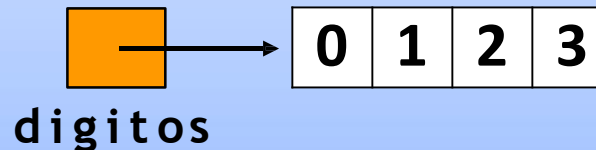
Inicialización un **vector**

- Es posible inicializar explícitamente los elementos de un vector
 - Se puede utilizar una expresión abreviada

```
int digitos[4] = {0, 1, 2, 3};
```

```
int digitos[] = {0, 1, 2, 3};
```

← Equivale






Inicialización de un **vector** multidimensional

- También se pueden inicializar los elementos de un vector multidimensional
 - Hay que especificar todos los elementos

```
int matriz[2][3]={  
    {5, 4, 3},  
    {2, 1, 0}  
};
```

← Nueva vector int[2,3] implícita

 **matriz** →

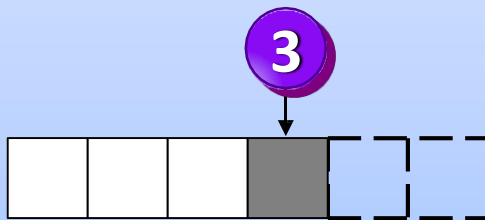
5	4	3
2	1	0



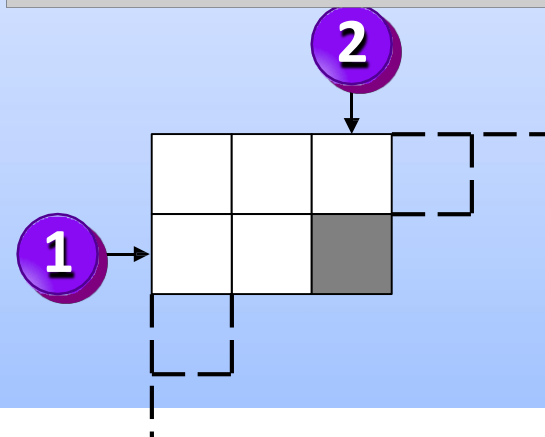
Acceso a los elementos de un **vector**

- Se indica un índice entero para cada dimensión
 - Los índices se cuentan a partir de cero
 - Es responsabilidad del programador el control del tamaño del vector

```
int digitos[10];  
...  
digitos[3];
```



```
int matriz[10][5];  
...  
matriz[1][2];
```





Acceso a los elementos de un vector

- Acceso secuencial a los elementos de un vector unidimensional

```
int digitos[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
for ( i = 0; i < 10; i++) {  
    printf("El valor de elemento %d es %d\n", i,  
          digitos[i]);  
}
```

0 1 2 3 4 5 6 7 8 9



Acceso a los elementos de un vector

- Acceso los elementos de un vector de 2 dimensiones

```
int i,j;
int matriz[2][3]={
    {5, 4, 3},
    {2, 1, 0}
};

for ( i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++){
        printf("El valor de elemento[%d][%d] es %d\n",
            i, j, matriz[i][j]);
    }
}
```

5 4 3 2 1 0



Ejemplo

```
/* Cambio optimo de monedas */
#include <stdio.h>

int monedas[10]= {1000, 500, 200, 100, 50, 20, 10, 5, 2, 1};
void main(){
    int num, cantidad, numonedas;

    printf ("Introduzca el importe exacto: ");
    scanf ("%d", &cantidad);

    printf ("El cambio optimo es el siguiente: \n");
    for (num=0; num<10; num++){

        numonedas=cantidad/monedas[num];
        if (numonedas != 0)
            printf ("%d de %d.\n", numonedas, monedas[num]);

        cantidad= cantidad % monedas[num];
    }
}
```



Cadenas

Las cadenas son simplemente **vectores** de **caracteres**.
las cadenas se terminan siempre con un valor nulo
(`'\0'`)

```
char cadena[80]; /* declara una cadena de 80  
caracteres */
```

```
char mensaje[ ]= "Bienvenidos a Culiacan";
```

```
char frutas[][10] = {"naranja", "platano", "manzana"};
```



Cadenas

El archivo cabecera `string.h` contiene un conjunto de funciones para manipular cadenas: copiar, cambiar caracteres, comparar cadenas, etc. Las funciones más comunes son:

```
char * strcpy (char c1[], char c2[]); //Copia c2 en c1
```

```
char * strcat (char c1[], char c2[]); //Añade c2 al final de c1
```

```
int strlen (cadena); //Devuelve la longitud de la cadena
```

```
int strcmp (char c1[], char c2[]); //Retorna 0 si c1 es igual c2
```

```
//Retorna >0 si c1 es mayor c2
```

```
//Retorna <0 si c1 es menor c2
```



Cadenas - Ejemplo

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

```
void main()
{
```

```
    char completo [80];
    char nombre[32] = "Luis";
    char apellidos [32] = "Beltran Lopez";
```

```
    /* Construye el nombre completo */
    strcpy ( completo, nombre );          /* completo <- "Luis" */
    strcat ( completo, " " );             /* completo <- "Luis " */
    strcat ( completo, apellidos );       /* completo <- "Luis Beltran Lopez" */
```

```
    printf ( "El nombre completo es %s\n", completo );
    printf("La longitud del nombre completo es %d\n", strlen(completo));
}
```



Cadenas - Ejemplo

// Programa que cambia el contenido de una cadena a mayúsculas
#include <stdio.h>

```
void main() {  
    char cadena[80];  
  
    printf ("Escriba una cadena: ");  
    fgets(cadena, 80, stdin);  
  
    int i=0;  
    while (cadena[i]!='\0'){  
        if (cadena[i]>='a' && cadena[i]<='z')  
            cadena[i]-=32;  
        i++;  
    }  
  
    printf ("La cadena es  %s \n", cadena);  
}
```



Ejercicios

1. Escribir un programa que lea elementos de una **matriz** de enteros. Y que realice la **suma** de todos los **elementos** de cada fila de la matriz y los almacene en un vector.
2. Escriba un programa que lea una **cadena** desde el **teclado** y muestre cuanta, **vocales**, **consonantes**, **dígitos**, **espacios contiene**.
3. Escriba un programa que **reciba** 2 **cadena**s de **caracteres** y determine si una es **anagrama** de la otra.
4. Hacer un programa que lea una **cadena** de **caracteres** y **determine** si es número **fraccionario**, **entero** o **no numérico**.



Estructuras

- Las **estructuras** son **colecciones** de **datos**, normalmente de tipos **diferentes** que actúan como un **todo**.
- Puede contener tipos de datos **simples** (carácter, float, int) o **compuestos** (estructuras, vectores)

```
struct coordenada {  
    int x;  
    int y;  
};
```




Estructuras

```
struct libro {  
    char titulo [40];  
    char autor [30];  
    int paginas;  
    char editorial[40];  
};
```

Una variable de tipo estructura se puede declarar así:

```
struct coordenada punto;  
struct libro novela;
```



Estructuras

Para asignar valores a los miembros de una estructura se utiliza la notación punto (.)

```
punto.x = 12;  
punto.y = 15;
```

Existe otro modo de declarar estructuras y variables estructuras.

```
struct coordenada {  
    int x;  
    int y;  
} punto;
```



Estructuras

Para evitar tener que escribir struct cada vez que se declara la estructura, se puede usar typedef en la declaración:

```
typedef struct coordenada {  
    int x;  
    int y;  
}Coordenadas;
```

y a continuación se puede declarar la variable Punto:

```
Coordenadas punto;
```

y utilizar la notación punto (.)

```
punto.x = 12
```

```
punto.y = 15;
```



Estructuras

Inicialización de estructuras

Una variable de tipo `struct` se puede inicializar en su declaración, escribiendo entre llaves los valores de sus campos en el mismo orden en que se declararon estos.

```
struct Persona {  
    char nombre [32];  
    char apellidos [64];  
    unsigned edad;  
};
```

```
struct Persona variable = {"Luis", "Beltran Lopez", 11 };
```



Esctructuras - Ejemplo

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

```
void main(){
```

```
    struct datos {
```

```
        char nombre[20];
```

```
        int edad;
```

```
        float sueldo; };
```

```
    struct datos empleado;
```

```
    // Entrada de datos
```

```
    printf("Nombre:");scanf("%s",empleado.nombre);
```

```
    printf("Edad:");scanf("%d",&empleado.edad);
```

```
    printf("Sueldo:");scanf("%f",&empleado.sueldo);
```

```
    // Salida de datos
```

```
    printf("Sus datos son:\n");
```

```
    printf("Nombre %s \n", empleado.nombre);
```

```
    printf("Edad %d \n", empleado.edad);
```

```
    printf("Sueldo %f \n", empleado.sueldo);
```

```
}
```



Apuntadores



Apuntadores

- Un puntero contiene un valor que es la dirección en memoria de un dato de cierto tipo.
- Los punteros se emplean en C para recorrer vectores, manipular estructuras creadas dinámicamente, pasar parámetros por referencia a funciones, etc.



Apuntadores

- Cuando se declara una variable, se reserva un espacio en la memoria para almacenar el valor de la variable. Ese espacio en memoria tiene una **dirección**.
- Un **puntero** es una **dirección** dentro de la **memoria**, es decir, un apuntador a donde se encuentra una variable.



Apuntadores

- Declaración de apuntadores

Un puntero se declara utilizando el asterisco (*) delante de un nombre de variable.

```
float *longitud;    /* puntero a datos float */  
char *indice;       /* puntero a datos char */  
int *p;             /* puntero a datos int */
```



Apuntadores

- Asignación

El valor que puede adquirir un puntero puede ser dirección de una variable.

El operador **&** devuelve la dirección de una variable:

`puntero = &variable;`



Apuntadores

- Operador de dirección (&)

El operador de **dirección** toma la dirección de una variable para asignarla a un apuntador.

```
int m=10; /* una variable entera */
```

```
int *p /* un puntero */
```

```
p = &m; /*se asigna a p la dirección de la  
variable m */
```



Apuntadores

Operaciones con apuntadores

Se puede alterar la variable a la que apunta un puntero.

Para ello se emplea el operador de **indirección**, que es el asterisco:

```
*puntero = 45;
```

En este caso, se está introduciendo un 45 en la posición de memoria a la que apunta **puntero**.



Apuntadores

Varios punteros pueden apuntar a la misma variable:

```
int *puntero1;
```

```
int *puntero2;
```

```
int var;
```

```
puntero1 = &var;
```

```
puntero2 = &var;
```

```
*puntero1 = 50;           /* mismo efecto que var=50 */
```

```
var = *puntero2 + 13;     /* var=50+13*/
```



Apuntadores

Declaración múltiple de punteros

Si en una misma declaración de variables aparecen varios punteros, hay que escribir el asterisco a la izquierda de cada uno de ellos:

```
int *puntero1, var, *puntero2;
```

Se declaran dos punteros a enteros (**puntero1** y **puntero2**) y un entero (**var**).



Apuntadores

El puntero nulo

El **puntero nulo** es un valor especial de puntero que no apunta a ninguna parte. Su valor es cero.

En **<stdio.h>** se define la constante **NULL** para representar el puntero nulo.



Apuntadores

Parámetros por referencia a funciones

En C todos los parámetros se pasan por valor. Esto tiene en principio dos inconvenientes:

- No se pueden modificar variables pasadas como argumentos
- Si se pasa como parámetro una estructura, se realiza un duplicado de ella, con lo que se pierde tiempo y memoria



Apuntadores

Se puede pasar un puntero como argumento a una función. El puntero no se debe alterar, pero sí el valor al que apunta:

```
void incrementa (int *var) {  
    (*var)++;  
}  
  
main() {  
    int x = 1;  
    incrementa (&x); /* x pasa a valer 2 */  
}
```



Apuntadores

- En el ejemplo anterior, había que poner paréntesis en **(*var)++** porque el operador **++** tiene más precedencia que el de indirección (el asterisco).
- Entonces ***var++** sería como escribir ***(var++)**, que no sería lo que queremos.



Apuntadores

Aritmética de punteros

El lenguaje C permite sumar o restar cantidades enteras al puntero, para que apunte a una dirección diferente, Consideremos un puntero a enteros:

```
int *ptr;
```

`ptr` apuntará a cierta dirección de memoria:

Pero también tendrán sentido las expresiones `ptr+1`, `ptr+2`, etc. La expresión `ptr+k` es un puntero que apunta a la dirección de `ptr` sumándole `k` veces el espacio ocupado por un elemento del tipo al que apunta (en este caso un `int`):



Apuntadores

/* Rellenar de unos los elementos del 10 al 20 */

```
int* ptr;           /* el puntero */
int vector [100];   /* el vector */
int i;              /* variable contadora */

ptr = &vector[0];   /* ptr apunta al inicio del vector */
ptr+=10;            /* ptr apunta a vector[10] */
for ( i=0; i<=10; i++ ) {
    *ptr = 1;        /* asigna 1 a la posición de
                       memoria apuntada por "ptr" */
    ptr++;           /* ptr pasa al siguiente elemento */
}
```



Apuntadores

Punteros y vectores

Si **ptr** es un puntero, la expresión

ptr[k] es equivalente a ***(ptr+k)**

con lo que se puede trabajar con un puntero como si se tratara de un vector:

```
int* ptr;  
int vector [100];  
ptr = &vector[10];  
for ( i=0; i<=10; i++ )  
    ptr[i] = 1; /* equivalente a *(ptr+i) = 1 */
```



Apuntadores

Un vector es en realidad un puntero constante. El nombre de un vector se puede utilizar como un puntero, al que se le puede aplicar la aritmética de punteros (salvo que no se puede alterar).

ptr = vector;

es equivalente a

ptr = &vector[0];



Apuntadores

Paso de vectores como parámetros a funciones

```
void inicia_vector( int n_elem, int* vector, int valor ) {  
    int i;  
    for ( i=0; i<n_elem; i++ )  
        *(vector++) = valor; /* operador de post-incremento */  
}  
main() {  
    int ejemplo [300];  
    int otro_vector [100]; /* pasa la dirección del vector  
                           ejemplo */  
    inicia_vector ( 300, ejemplo, 10 );  
    /* rellena los elems. del 150 al 199 */  
    inicia_vector ( 50, otro_vector+150, 20 );  
}
```



Apuntadores - Ejemplo

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

```
int longstr(char *str);
```

```
void main(){  
    char cad[]="Hola";  
    printf("la longitud es: %d \n", longstr(cad));  
}
```

```
int longstr(char *str){  
    int i=0;  
    while(*str!='\0'){  
        i++;  
        str++;  
    }  
    return i;  
}
```




Apuntadores - Ejemplo

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

```
void cambiar(int *x, int *y);
```

```
void main(){
```

```
    int a=10, b=20;
```

```
    cambiar(&a, &b);
```

```
    printf("El valor de a es %d, y el valor de b es %d\n", a, b);
```

```
}
```

```
void cambiar(int *x, int *y){
```

```
    int temp;
```

```
    temp = *x;
```

```
    *x = *y;
```

```
    *y = temp;
```

```
}
```



Apuntador a apuntador

Los punteros pueden apuntar a otros punteros

```
#include <stdio.h>
void main(){

int x, *ptr1, **ptr2, ***ptr3;

x = 10;
ptr1 = &x;
ptr2 = &ptr1;
ptr3 = &ptr2;

printf("x = %d \n ", x);
printf("x = %d \n ", *ptr1);
printf("x = %d \n ", **ptr2);
printf("x = %d \n ", ***ptr3);

printf("valor x es %d y su dirección x = %p \n ", x, &x);
printf("El valor ptr1 es %p, apunta a %d, y su dirección es %p \n ", ptr1, *ptr1, &ptr1);
printf("El valor ptr2 es %p, apunta a %p, y su dirección es %p \n ", ptr2, *ptr2, &ptr2);
printf("El valor ptr3 es %p, apunta a %p, y su dirección es %p \n ", ptr3, *ptr3, &ptr3);
}
```



Ejercicios

1. Crear una función `char * rev(char *)` que dado un **vector** de caracteres retorne una cadena inversa de la cadena original recibida.
2. Escriba un función que tome tres variables (**a**, **b**, **c**) como parámetros separados y rote los valores almacenados, i.e., **a** tomará el valor de **b**, **b** el de **c** y **c** el de **a**.



Apuntadores

Arreglos

En C se pueden tener arreglos de apuntadores ya que los apuntadores son variables.

```
char *nomb[] = { "Ene", "Feb", "Mar", .... };
```



Apuntadores - Ejemplo

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

```
void main(){  
    char *semana[]={"Domingo",  
                    "Lunes",  
                    "Martes",  
                    "Miércoles",  
                    "Jueves",  
                    "Viernes",  
                    "Sábado"  
};  
  
    int i;  
  
    for(i=0; i<7; i++)  
        printf("%s\n", semana[i]);  
}
```



Apuntadores

Punteros y estructuras

Un puntero puede apuntar a una estructura y acceder a sus campos:

```
struct Dato {  
    int campo1, campo2;  
    char campo3 [30];  
};
```

```
struct Dato x;  
struct Dato *ptr;  
...  
ptr = &x;  
(*ptr).campo1 = 33;  
strcpy ( (*ptr).campo3, "hola" );
```



Apuntadores

El operador ->

Para hacer menos incómodo el trabajo con punteros a estructuras, el C tiene el operador flecha -> que se utiliza de esta forma:

ptr->campo

que es equivalente a escribir

(*ptr).campo



Apuntadores

Así, el ejemplo anterior quedaría de esta forma:

```
ptr = &x;
```

```
ptr->campo1 = 33;
```

```
strcpy ( ptr->campo3, "hola" );
```




Alineación de la memoria en estructuras

Reglas de alineación de memoria - para estructura

1. Antes de cada miembro individual, habrá relleno para que comience en una dirección que sea divisible por su tamaño. por ejemplo, en un sistema de 64 bits, `int` debe comenzar en una dirección divisible por 4, un `long` por 8, `short` por 2.
2. `char` y `char []` son especiales, podría estar en cualquier dirección de memoria, por lo que no necesitan relleno (`padding`) antes que ellos.



Alineación de la memoria en estructuras

Reglas de alineación de memoria - para estructura

3. Para la estructura, aparte de la necesidad de **alineación** para cada miembro **individual**, el tamaño de la estructura en sí se alineará con un tamaño **divisible** por el tamaño del miembro individual **más grande**, rellenándolo al final. por ejemplo, si el miembro más grande de la estructura es:
 - **long, double, void*** , entonces divisible por 8,
 - **int** luego por 4,
 - **short** por 2.
4. El orden de los miembros puede afectar el tamaño real de la estructura.



Alineación de la memoria en estructuras

Ejemplos

```
struct Foo {  
    char x; // 1 byte  
    double y; // 8 bytes  
    char z; // 1 bytes  
};
```

24 bytes.

```
struct Foo_1 {  
    char x; // 1 byte  
    char z; // 1 bytes  
    double y; // 8 bytes  
  
};
```

16 bytes



Apuntadores

Memoria dinámica: malloc y free (stdlib.h)

Por medio de punteros se puede reservar o liberar memoria **dinámicamente**, es decir, según se necesite.

La función **malloc** sirve para solicitar un bloque de memoria del tamaño suministrado como parámetro. Devuelve un puntero a la zona de memoria concedida:

void* malloc (unsigned numero_de_bytes);

El tamaño se especifica en bytes. Se garantiza que la zona de memoria concedida no está ocupada por ninguna otra variable ni otra zona devuelta por **malloc**.

Si **malloc** es incapaz de conceder el bloque (p.ej. no hay memoria suficiente), devuelve un puntero nulo.



Apuntadores

Punteros void*

La función `malloc` devuelve un puntero inespecífico, que no apunta a un tipo de datos determinado. En C, estos punteros sin tipo se declaran como `void*`

Muchas funciones que devuelven direcciones de memoria utilizan los punteros `void*`. Un puntero `void*` puede convertirse a cualquier otra clase de puntero:

```
char *ptr = (char*)malloc(1000);
```



Apuntadores

Operador sizeof

El problema de **malloc** es conocer cuántos bytes se quieren reservar. Si se quiere reservar una zona para diez enteros, habrá que multiplicar diez por el tamaño de un entero.

El tamaño en bytes de un elemento de tipo **T** se obtiene con la expresión

sizeof (T)

El tamaño de un **char** siempre es 1 (uno).



Apuntadores

Función free

Cuando una zona de memoria reservada con **malloc** ya no se necesita, puede ser *liberada* mediante la función **free**.

void free (void* ptr);

ptr es un puntero de cualquier tipo que apunta a un área de memoria reservada previamente con **malloc**.

Si **ptr** apunta a una zona de memoria indebida, los efectos pueden ser desastrosos, igual que si se libera dos veces la misma zona.



Apuntadores

Ejemplo

```
#include <stdio.h>
#include <stdlib.h>
void main(){
    Int *ptr;           /* puntero a enteros */
    Int *ptr2;          /* otro puntero */
    /* reserva memoria para 300 enteros */
    ptr = (int*)malloc ( 300*sizeof(int) );
    ptr[33] = 15;       /* trabaja con el área de memoria */
    inicia_vector (10, ptr, 10);
    /* otro ejemplo */
    ptr2 = ptr + 15;    /* asignación a otro puntero */
    /* finalmente, libera la zona de memoria */
    free(ptr);
}
```




Apuntadores - Ejemplo

```
#include <stdio.h>
#include <stdlib.h>
void main(){
    int *vect, N, i;
    printf ("Tamaño del vector? ");
    scanf ("%d", &N);
    vect = malloc(sizeof(int) * N);
    if (vect!=NULL){
        for(i=0; i<N ; i++)
            vect[i]=i*10;

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

        free(vect);
    }
    else
        printf("Memoria insuficiente\n");
}
```



Apuntadores

Punteros a funciones

Un puntero puede apuntar a código.

```
void (*ptr_fun) (int,int);
```



Apuntadores - Ejemplo

```
int suma (int a, int b) {  
    return a+b;  
}  
  
int resta (int a, int b) {  
    return a-b;  
}  
  
// declaramos un puntero a funciones con dos parámetros  
// enteros y retorna entero  
int (*ptrFuncion) (int,int);  
void main(){  
    ptrFuncion = suma;        // 'funcion' apunta a 'suma'  
    x = ptrFuncion(4,3);      // x=suma(4,3)  
    ptrFuncion = resta;      // 'funcion' apunta a 'resta'  
    x = ptrFuncion(4,3);      // x=resta(4,3)  
}
```



Ejercicios

1. Desarrolle una función llamada menú que reciba una lista de opciones para mostrar un menú de opciones y espere la selección del usuario, retornando la posición de opción seleccionada, las opciones mostradas deben estar enumeradas iniciando con 1, y solo retornara la opción seleccionada si se encuentra en el rango valido.
2. Desarrolle una función que reciba dos apuntadores de cadena de caracteres, para copiar de la primera a la segunda un numero determinado de caracteres desde la izquierda (debe asignar a la segunda cadena la memoria exactamente necesaria).
3. Desarrolle una función que reciba dos apuntadores de cadena de caracteres, para copiar de la primera a la segunda un numero determinado de caracteres desde la derecha (debe asignar a la segunda cadena la memoria exactamente necesaria).

Nota: Utilice la función menú del problema 1 para realizar un ejemplo que ilustre el funcionamiento de todos las funciones.



Listas simplemente **enlazadas**



Introducción

Hay dos **desventajas** serias con respecto a las estructuras **estáticas** de pilas y colas usando **vectores**:

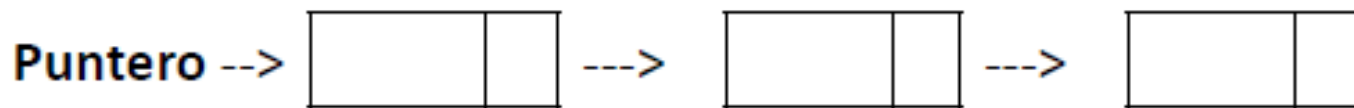
1. Tienen un **espacio limitado** de memoria,
2. Es posible **no ocupar** toda la memoria disponible, haciendo que se **desperdicie** espacio.

Una solución es **usar listas enlazadas**.



Definiciones

- Las listas enlazadas son estructuras de datos **dinámicas**, esto significa que **adquieren** espacio y **liberan** espacio a medida que se necesita.
- Éstas son estructuras de datos semejantes a los **vectores** salvo que el acceso a un elemento no se hace mediante un **índice** sino mediante un **Puntero**.





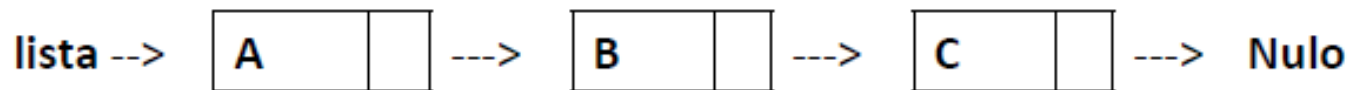
Definiciones

- Una lista simplemente enlazada es una **relación** de elementos (nodos), donde que cada elemento esta **relacionado** con un elemento del **conjunto**, diferente así mismo.
- Como cada elemento puede tener a lo más una **arista dirigida** que **sale** y una **arista dirigida** que **entra**, bien puede tener 0 aristas que salen, o 0 aristas que entran.
 - Si el nodo tiene 0 **aristas que salen**, entonces es el **final** de la lista.
 - Si el nodo tiene 0 **aristas que entran**, entonces es el **inicio** de la lista.



Definiciones

Por razones practicas, se dibujan una **flecha** que **sale** de un identificador de la lista y **entra** al inicio de la lista y otra flecha que sale del **final** de la lista y apunta a un símbolo que se llama **NULO**.





Operaciones

Recorrido

Esta operación consiste en **visitar** cada uno de los nodos que forman la lista. Se comienza con el **primero**, se toma el valor del **campo enlace** para avanzar al segundo nodo, el campo enlace de este nodo nos dará la **dirección del tercer nodo**, y así sucesivamente.

Inserción

Esta operación consiste en **agregar** un nuevo nodo a la lista. Para esta operación se pueden considerar tres casos:

- Insertar un nodo al **inicio**.
- Insertar un nodo **antes** o **después** de cierto nodo.
- Insertar un nodo al **final**.



Operaciones

Borrado

La operación de borrado consiste en **quitar** un nodo de la lista, **redefiniendo** los enlaces que correspondan. Se pueden presentar cuatro casos:

- Eliminar el **primer** nodo.
- Eliminar el **último** nodo.
- Eliminar un **nodo** con **cierta información**.
- Eliminar el nodo **anterior** o **posterior** al nodo cierta con información.

Búsqueda

Esta operación consiste en **visitar** cada uno de los nodos, tomando al campo enlace como **puntero** al **siguiente** nodo a visitar hasta encontrar el deseado.



Implementación de estructura de datos de una lista simplemente enlazada

- Normalmente cada nodo de la lista esta estructurado con dos partes:
 - La parte de información.
 - La parte de dirección al siguiente nodo de la lista.
- El campo de información contiene el elemento real de la lista. El campo de dirección al siguiente contiene un apuntador al elemento con el cual esta relacionado, es decir, al elemento siguiente de la lista.
- La lista completa se accede mediante el identificador de la lista. El campo de la dirección del último nodo apunta a una dirección nula.



Implementación de estructura de datos de una lista simplemente **enlazada**

// Definición de la estructura de datos

```
typedef struct nodo {  
    int info;  
    struct nodo *sig;  
} TNode;
```

// Declaración de la variable para la Lista

```
TNode*miEstructuraLista;
```



Implementación de estructura de datos de una lista simplemente **enlazada**

// Agrega un nodo al inicio de la lista

```
void AgregarInicio(TNodo**lista, int dato){  
    TNodo*nuevo=NULL;
```

// Asigna memoria para el nuevo nodo

```
nuevo = (TNodo*) malloc(sizeof(TNodo));  
nuevo->info=dato; // Asigna el valor al nodo  
nuevo->sig= *lista; // Coloca nodo al inicio de lista  
*lista = nuevo; // El inicio de lista cambia al nuevo  
}
```



Implementación de estructura de datos de una lista simplemente **enlazada**

// Muestra todos los nodos de la lista

```
void VerTodos (TNode*lista) {  
    // Recorre la lista mientras no encuentre el ultimo  
    while (lista!=NULL) {  
        // Muestra el dato del nodo  
        printf ("%d\n", lista->info) ;  
        // Se posiciona en el siguiente nodo  
        lista=lista->sig;  
    }  
}
```



Implementación de estructura de datos de una lista simplemente **enlazada**

// Obtener información de nodo de cierta posición

```
int ObtenerDatoNodo (TNodo**lista,  intpos) {  
    int i;
```

// Toma dirección del primer nodo

```
TNodo*temp=*lista;
```

// Recorre la lista hasta la posición deseada

```
for (i=0; i<pos;  i++)
```

```
temp=temp->sig;
```

// Retorna información de la posición deseada

```
returntemp->info;  
}
```




Implementación de estructura de datos de una lista simplemente **enlazada**

// Borra el primero nodo de la Lista

```
void BorrarInicio(TNodo**lista){
    TNodo*primero, *temp;
    primero =*lista; // Toma la dirección del primer nodo
    if(primero!=NULL){ // Determina si no esta vacía la lista
        temp=primero->sig; // Guarda dirección el segundo nodo
        free(primero); // Libera la memoria del primer nodo
        primero=temp; // El segundo nodo pasa a ser el primero

        if(primero!=NULL) // Pregunta si no quedo vacía
            *lista=primero; // Cambia la dirección del inicio de la lista
        else
            *lista=NULL; // Si quedo vacía pone NULL a la lista
    }
}
```



Implementación de estructura de datos de una lista simplemente **enlazada**

// Liberar todos los nodos de la lista

```
void BorrarTodosNodos(TNodo**lista) {
    TNodo*primero, *temp;
    primero =*lista; // Toma la dirección del primer nodo
    // Determina si no esta vacía la lista
    while(primero!=NULL) {
        // Guarda temporalmente el segundo nodo
        temp=primero->sig;
        // Libera la memoria del primer nodo
        free(primero);
        // El segundo nodo pasa a ser el primero
        primero=temp;
    }
    *lista=NULL; // La referencia se hace nula.
}
```



Ejercicios en clase

1. Dadas dos listas L_1 y L_2 obtener una lista L_3 con los elementos ordenados de L_1 y L_2 .
2. Escriba un algoritmo que dada una lista L_1 devuelva otra lista L_2 conteniendo los elementos repetidos de L_1 . Por ejemplo, si L_1 almacena los valores 5, 2, 7, 2, 5, 5, 1, debe construirse una lista L_2 con los valores 5, 2. Si en L_1 no hay elementos repetidos, L_2 será la lista vacía.



Ejercicios en casa

1. Escriba un algoritmo que dada una lista L_1 devuelva otra lista L_2 , inversa de L_1 .
2. Escriba un algoritmo que dada una lista L_1 y un valor devuelva dos listas L_2 y L_3 , cada una de ellas conteniendo, respectivamente, los elementos de L_1 mayores y menores al valor proporcionado.
3. Escriba un algoritmo que dada una lista L , mueva un elemento de posición i , n lugares hacia delante, si es posible, o al final de L , en caso contrario.



Procesos



Definiciones

- Un proceso es una entidad **activa** que tiene asociada un conjunto de atributos: código, datos, pila, registros e identificador único.
- Representa la entidad de ejecución utilizada por el Sistema Operativo. Frecuentemente se conocen también con el nombre de **tareas** (“tasks”).
- Un programa representa una entidad **pasiva**. Cuando un programa es reconocido por el Sistema Operativo y tiene asignado recursos, se convierte en proceso.



Definiciones...

Generalmente un proceso:

- **Es la unidad de asignación de recursos:** el Sistema Operativo va asignando los recursos del sistema a cada proceso.
- **Es una unidad de despacho:** un proceso es una entidad activa que puede ser ejecutada, por lo que el Sistema Operativo conmuta entre los diferentes procesos listos para ser ejecutados o despachados.



Definiciones...

- Los programas se ejecutan en un contexto llamado proceso
- Un proceso es un programa en ejecución
- Componentes de un proceso: código y una estructura de datos
- Contexto de un proceso: memoria, archivos abiertos, sockets, etc.
- Los procesos tienen un identificador (PID)
- Procesos se alternan en el uso del CPU



Objetivos de un Sistema Operativo

- Un Sistema Operativo **multiusuario** y **Multitarea** pretende crear la ilusión a sus usuarios de que se dispone del sistema al completo.
- La **capacidad** de un **procesador** de cambiar de tarea o **contexto** es infinitamente más **rápida** que la que pueda tener una persona normal, por lo que habitualmente el sistema cumple este **objetivo**



Objetivos de un Sistema Operativo

Los S.O Administran la ejecución de varios procesos para maximizar la utilización del CPU y proveer tiempo de respuesta razonable:

- Asignar recursos a procesos
- Soportar comunicación entre procesos
- Creación de procesos por parte de los usuarios



Multitareas (Multiprogramación)

- El término Multitareas significa que **más** de un **proceso puede** estar listo para su **ejecución**.
- El sistema operativo escoge uno de estos procesos. Cuando el proceso necesita esperar por un recurso (por ejemplo, que algún usuario presione una tecla o un acceso a disco), el sistema operativo guarda toda la información necesaria para continuar el proceso donde éste se quedó y entonces selecciona otro para ejecutar.



Multitareas (Multiprogramación)

- Unix/Linux no sólo hace la multiprogramación, sino que también es de tiempo compartido. La idea es que parezca que el sistema operativo ejecuta varios procesos simultáneamente.
- Si existe una sola CPU, entonces en cualquier momento puede ejecutarse únicamente una instrucción de un solo proceso.



Estado del proceso

- En ejecución (Running)
- Esperando (Waiting)
- Detenido (Stopped)
- Zombi



Identificadores de procesos

- de proceso (pid)
- del padre (ppid)
- de usuario (uid)
- de grupo (gid)



Creación de procesos

- Para ejecutar comandos y aplicaciones
- Para proporcionar servicios, por ejemplo, impresión, Web etc.
- Creados por otros procesos por clonación: copia casi exacta (excepto PID)
- Un proceso especial (init), con PID=1, es la raíz del árbol de procesos



Ciclo de vida de procesos

- Creación
- Se alternan en el uso del CPU, E/S y otros recursos
- Termina (voluntariamente, error, etc.) retornando los recursos asignados (memoria, archivos, tablas en kernel)
- Algunos procesos no mueren (demonios o daemons):
 - httpd, sshd, ...



Algunas herramientas de software

- top: muestra el consumo recursos de procesos
- ps: reporta estatus de procesos
- pstree: muestra arbol de procesos
- kill [signal] PID : envía señal a proceso



Tratamiento de Procesos - Identificación

- `getpid ()`. Retorna el ID de proceso
- `getppid ()`. Retorna el ID del proceso padre



Tratamiento de Procesos - Ejemplo

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

```
void main () {  
    printf ("El ID de este proceso es %d\n", (int) getpid ());  
    printf ("El ID del padre de este proceso es %d\n", (int) getppid  
    ());  
}
```



Creación de Procesos

Función `system(char [] comando)`

Con `system()` se consigue detener la ejecución de proceso al llamar a un comando del shell y retornar cuando éste haya acabado.

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

```
main () {
```

```
    system ("ls -l ");
```

```
}
```



Creación de Procesos...

Función `fork()`

- Esta función crea un proceso nuevo o “proceso hijo” que es exactamente igual que el “proceso padre”.
- Si `fork()` se ejecuta con éxito devuelve:
 - Al padre: el PID del proceso hijo creado.
 - Al hijo: el valor 0.



Creación de Procesos...

```
#include <stdio.h>

void main() {
    int hijo=fork();
    if (hijo==0){
        printf ("Yo soy el proceso Hijo...\n");
        printf("PID = %d\n", getpid());
        printf("PPID = %d\n", getppid());
    }
    else{
        printf ("Yo soy el proceso Padre...\n");
        printf("PID = %d\n", getpid());
        printf("PPID = %d\n", getppid());
    }
}
```



Creación de Procesos...

`wait (int *estadoHijo)`

- Espera la terminación del proceso hijo
- Una vez que salimos del **`wait()`**, tenemos unas macros (incluir **`wait.h`** y **`stdlib.h`**) que nos permiten evaluar el contenido de `estadoHijo`:
- **`WIFEXITED(estadoHijo)`** es 0 si el hijo ha terminado de una manera anormal (caída, eliminado con un kill, etc). Distinto de 0 si ha terminado porque ha hecho una llamada a la función `exit()`
- **`WEXITSTATUS(estadoHijo)`** devuelve el valor que ha pasado el hijo a la función `exit()`, siempre y cuando la macro anterior indique que la salida ha sido por una llamada a `exit()`.



Creación de Procesos...

```
#include <stdio.h>

void main() {
    int hijo=fork();
    int estado;
    if (hijo==0){
        printf ("Yo soy el proceso Hijo...\n");
        printf("PID = %d\n", getpid());
        printf("PPID = %d\n", getppid());
    }
    else{
        printf ("Yo soy el proceso Padre...\n");
        printf("PID = %d\n", getpid());
        printf("PPID = %d\n", getppid());
        wait(&estado);
    }
}
```




Ejercicio

Desarrolle un programa que pueda crear un proceso padre y un proceso hijo que compartan la tarea de suma un vector, cada proceso debe suma la mitad del vector y al padre mostrara la suma total.

Los datos del vector serán pasados desde la línea de comandos.



Comunicación entre Procesos (Tuberías)

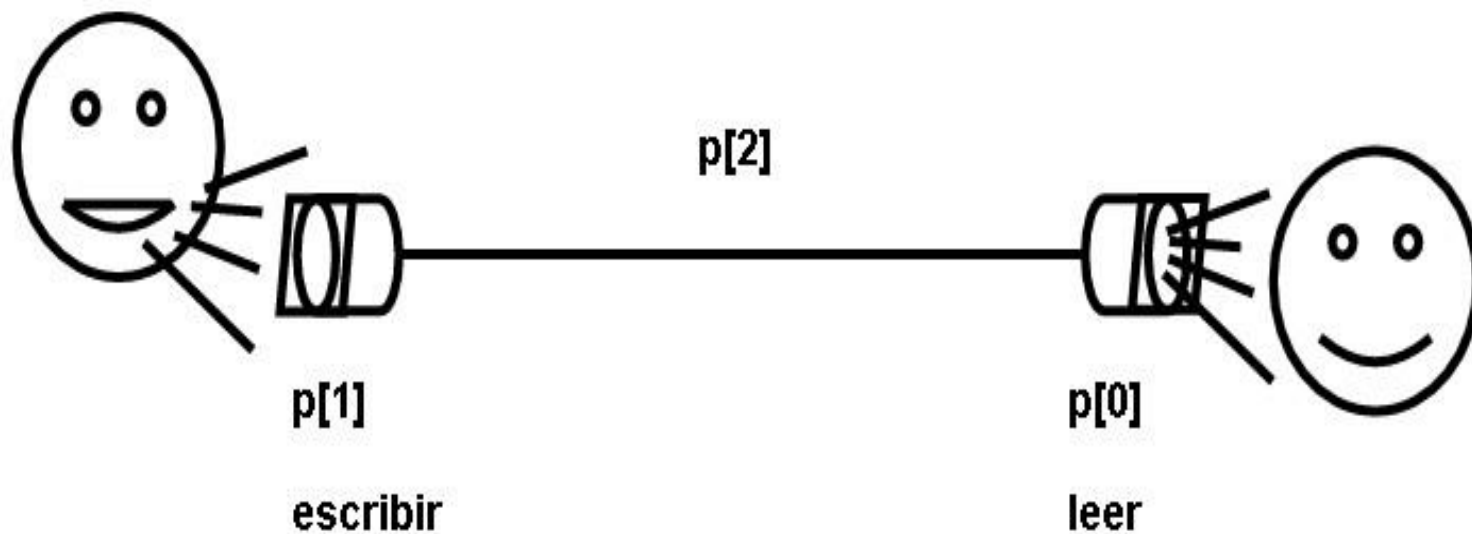


Definición

- Las tuberías o “**pipes**” simplemente conectan la salida estándar de un proceso con la entrada estándar de otro. Normalmente las tuberías son de un solo sentido.
- Las tuberías suelen ser “**half-duplex**”, es decir, de un único sentido, y se requieren dos tuberías “half-duplex” para hacer una comunicación en los dos sentidos, es decir “full-duplex”.
- Las tuberías son, por tanto, flujos unidireccionales de bytes que conectan la salida estándar de un proceso con la entrada estándar de otro proceso.



Definición





Programación de Pipes

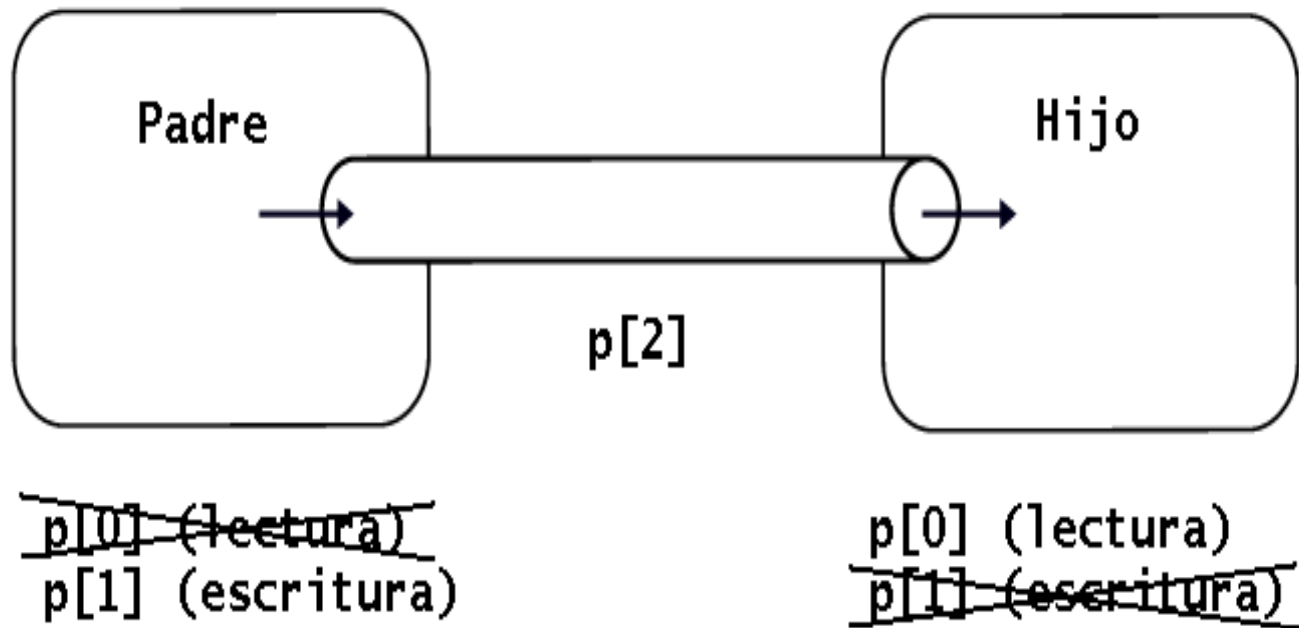
- Una tubería tiene en realidad dos descriptores de archivo: uno para el extremo de escritura y otro para el extremo de lectura.

```
int p[2];  
pipe(p);
```

- Una vez creado un pipe, se podrán hacer lecturas y escrituras de manera normal, como si se tratase de cualquier archivo.
- Para asegurar la unidireccionalidad de la tubería, es necesario que tanto padre como hijo cierren los respectivos descriptores de archivos que no serán usados.



Programación de Pipes





Ejemplo 1

```
#include <stdio.h>
#define LEER 0
#define ESCRIBIR 1
void main(int argc, char *argv[]){

    int estado;
    int valores[]={1,2,3,4,5,6,7,8,9,10};
    int sumapadre=0, sumahijo=0, i, tuberia[2];
    int sumaparcial;

    pipe(tuberia);
    int hijo=fork();

    if(hijo==0){        //Codigo el proceso hijo
        for(i=0; i<5; i++)
            sumahijo+=valores[i];

        close(tuberia[LEER]); //cierra la lectura
        write(tuberia[ESCRIBIR], &sumahijo, sizeof(sumahijo));
    }
}
```



Ejemplo 1

```
else { //Codigo del proceso padre
    for(i=5; i<10; i++)
        sumapadre+=valores[i];

    wait(&estado);
    close(tuberia[ESCRIBIR]); //cierra la escritura
    read(tuberia[LEER], &sumaparcial, sizeof(sumaparcial));
    float promedio = (sumapadre + sumaparcial)/10.0;
    printf("El promedio es: %f\n", promedio);
}
}
```




Ejemplo 2

```
#include <stdio.h>
#define LEER 0
#define ESCRIBIR 1

void main(int argc, char *argv[]){
    int estado;
    FILE *da;
    int valores[]={1,2,3,4,5,6,7,8,9,10};
    int sumapadre=0, sumahijo=0, i, tuberia[2];
    int sumaparcial;

    pipe(tuberia);
    int hijo=fork();
    if(hijo==0){        //Codigo el proceso hijo
        for(i=0; i<5; i++) sumahijo+=valores[i];

        close(tuberia[LEER]); //cierra la lectura
        da=fdopen(tuberia[ESCRIBIR],"w"); // Convierte el descriptor de archivo a flujo de datos
        fprintf(da,"%d", sumahijo); // Envía la suma parcial al padre
    }
}
```



Ejemplo 2

```
else { //Codigo del proceso padre
    for(i=5; i<10; i++)
        sumapadre+=valores[i];

    wait(&estado);
    close(tuberia[ESCRIBIR]); //cierra la escritura
    da=fdopen(tuberia[LEER], "r");
    fscanf(da, "%d", &sumaparcial); // Recibe la suma parcial del proceso hijo
    float promedio = (sumapadre + sumaparcial)/10.0;
    printf("El promedio es: %f\n", promedio);
}
}
```



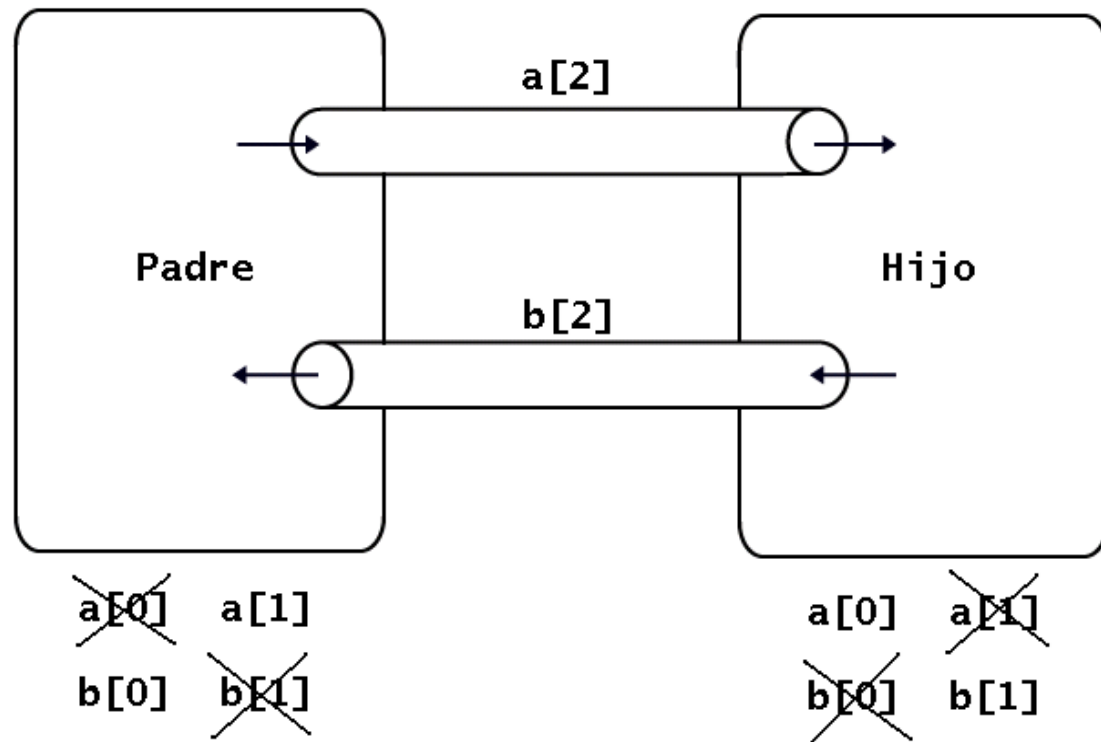
Programación de Pipes

La implementación de una comunicación bidireccional entre dos procesos mediante tuberías, requiere la creación de dos tuberías diferentes ($a[2]$ y $b[2]$), una para cada sentido de la comunicación.

En cada proceso habrá que cerrar descriptores de archivo diferentes que no serán utilizados.



Programación de Pipes





Ejercicio

Desarrolle un programa donde se genere un proceso padre y un proceso hijo, el proceso padre recibirá 2 valores desde la línea de comandos. Estos datos los enviara al proceso hijo para que calcule la suma y la multiplicación de ambos y los resultados serán retornados al proceso padre para su impresión.



Comunicación Cliente Servidor (Sockets)



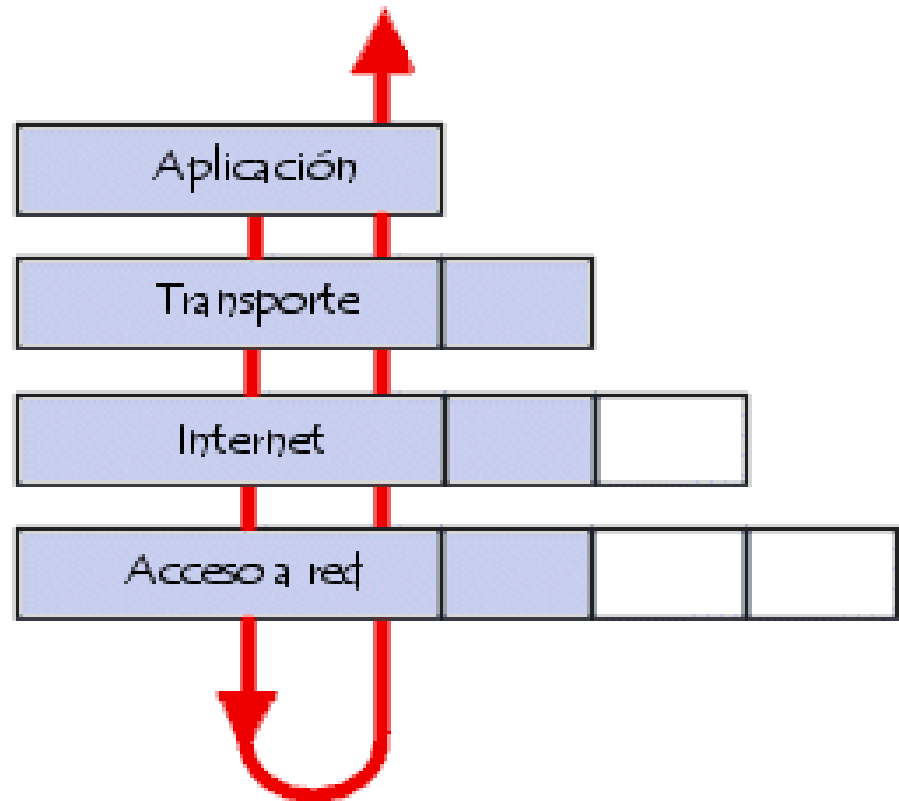
Comunicación por red

- Muchas de las aplicaciones que usamos todos los días, como el correo electrónico o los navegadores web, utilizan protocolos de red para comunicarse.
- El protocolo mas utilizado para estas aplicaciones esta basado en el modelo TCP/IP.



El modelo TCP/IP

- El modelo TCP/IP, influenciado por el modelo OSI, también utiliza el enfoque modular (utiliza módulos o capas), pero sólo contiene cuatro:





El modelo TCP/IP

- **Capa de acceso a la red:** especifica la forma en la que los datos deben enrutar, sea cual sea el tipo de red utilizado
- **Capa de Internet:** es responsable de proporcionar el paquete de datos.
- **Capa de transporte:** brinda los datos de enrutamiento, junto con los mecanismos que permiten conocer el estado de la transmisión;
- **Capa de aplicación:** incorpora aplicaciones de red estándar (Telnet, SMTP, FTP, HTTP, etc.).



Socket

- Interfaz normalizado de comunicación entre procesos utilizado en la arquitectura TCP/IP.
- Extremo de una conexión entre procesos de aplicación que se ejecutan en una red TCP/IP.
- Cada extremo se identifica mediante su dirección IP o nombre DNS y un número de puerto (entre 1 y 65535).
- Permiten que un proceso pueda enviar o recibir datos a través de la red.



Sockets: Modelo cliente-servidor

Una aplicación consta de dos partes o extremos que se ejecutan en la misma máquina o en máquinas distintas:

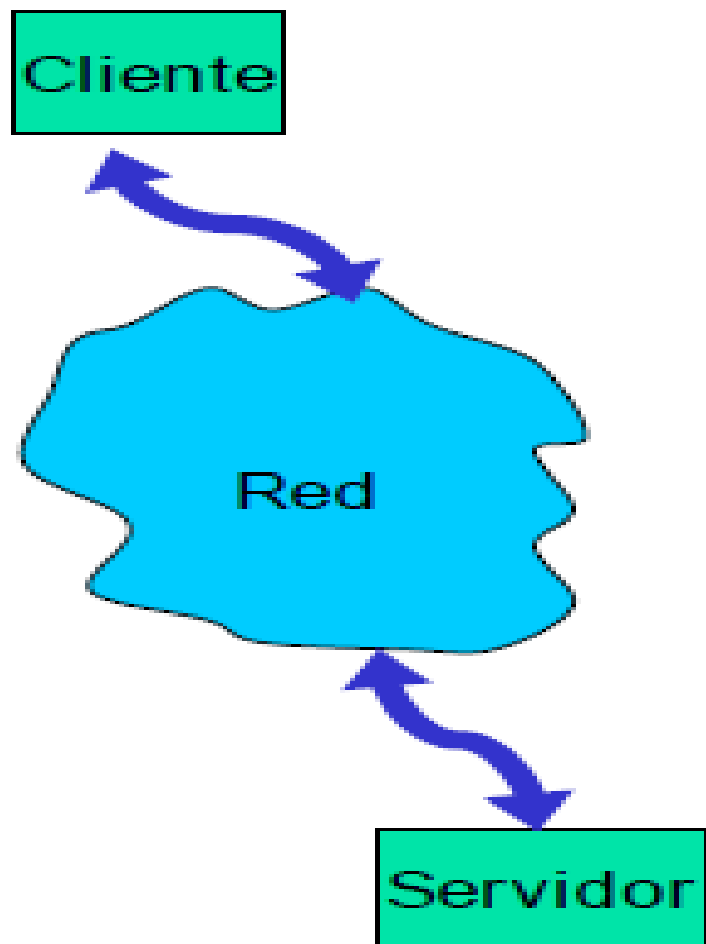
Cliente: realiza peticiones.

Servidor: responde a las peticiones (ofrece un servicio).

Ambos extremos dialogan mediante un protocolo de aplicación.



Sockets: Modelo cliente-servidor





Sockets: Tipos de servicio

Orientado a la conexión (TCP).

Flujos de bytes (**streams**).

Con control de errores y control del flujo.

Comunicación uno a uno.

Sin conexión (UDP).

Bloques de datos de usuario de hasta 64 Kb (**datagramas**).

Sin control de errores ni control del flujo.

Comunicación:

Uno a uno.

Uno a varios (multicast).



Conexión

Para poder realizar la conexión entre un cliente y un servidor se requiere:

- **Dirección del servidor.**
- el cliente debe conocer a qué computadora desea conectarse, el nombre si esta registrador en algún DNS o la dirección IP
- **Puerto (Servicio)**
- Cuando un cliente desea conectarse, debe indicar qué servicio quiere. Son números enteros normales y van de 1 a 65535, del 1 a 1023 están reservados para servicios habituales de los sistemas operativos (www, ftp, mail, ping, etc)



El Servidor

Los pasos que debe seguir un programa servidor son los siguientes:

- **Apertura de un socket**, mediante la función `socket()`.
- Avisar al sistema operativo de que hemos abierto un socket y queremos que asocie nuestro programa. Se consigue mediante la función `bind()`.
- Avisar al sistema de que comience a atender dicha conexión de red. Se consigue mediante la función `listen()`.



El Servidor

- Pedir y aceptar las conexiones de clientes al sistema operativo. Para ello hacemos una llamada a la función `accept()`.
- Escribir y recibir datos del cliente, por medio de las funciones `read()` y `write()`.
- Cierre de la comunicación del socket, por medio de la función `close()`.



El Cliente

Los pasos que debe seguir un programa cliente son los siguientes:

- Abrir un socket como el servidor, por medio de la función `socket()`.
- Solicitar conexión con el servidor por medio de la función `connect()`. En esta llamada se debe facilitar la dirección IP del servidor y el número de puerto que se desea.
- Escribir y recibir datos del servidor por medio de las funciones `read()` y `write()`.
- Cerrar la comunicación por medio de `close()`.



Estructuras de datos

```
struct sockaddr_in{  
    short sin_family;  
        unsigned short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8];  
};
```



Estructuras de datos

sin_family: Indica la “familia de direcciones”, en nuestro caso siempre tendrá el valor “AF_INET”.

sin_port: Indica el numero de puerto.

sin_addr: Indica la dirección IP.

sin_zero: Se rellenan a cero. Simplemente tiene sentido para que el tamaño de esta estructura coincida con el de **sockaddr**.



Estructuras de datos

La estructura `in_addr` utilizada en `sin_addr` tiene la siguiente definición:

```
struct in_addr {  
    unsigned long s_addr;  
}
```



Ejemplo de estructura

```
struct hostent *hostinfo
```

```
hostinfo = gethostbyname("nombrehost.net")
```

```
int puerto=80;
```

```
struct sockaddr_in  dir;
```

```
dir.sin_family = AF_INET;
```

```
dir.sin_port = htons(puerto);
```

```
dir.sin_addr = *((struct in_addr *) hostinfo->h_addr);
```

```
memset( &(dir.sin_zero), '\0', 8);
```



Funciones

int **socket**(int domain, int type, int protocol);

Donde:

domain: dominio de dirección, que fijaremos a
AF_INET

type: tipo de servicio (SOCK_STREAM o SOCK_DGRAM)

potocol: lo fijaremos a 0 para lo lo determine
automáticamente

Ejemplo:

```
Int socket_fd = socket(PF_INET, SOCK_STREAM, 0);
```



Funciones

Recuperación de la dirección del cliente

Ejemplo:

```
#include <arpa/inet.h>
```

```
char *ip = inet_ntoa(cliente.sin_addr);
```



Servidores Iterativos

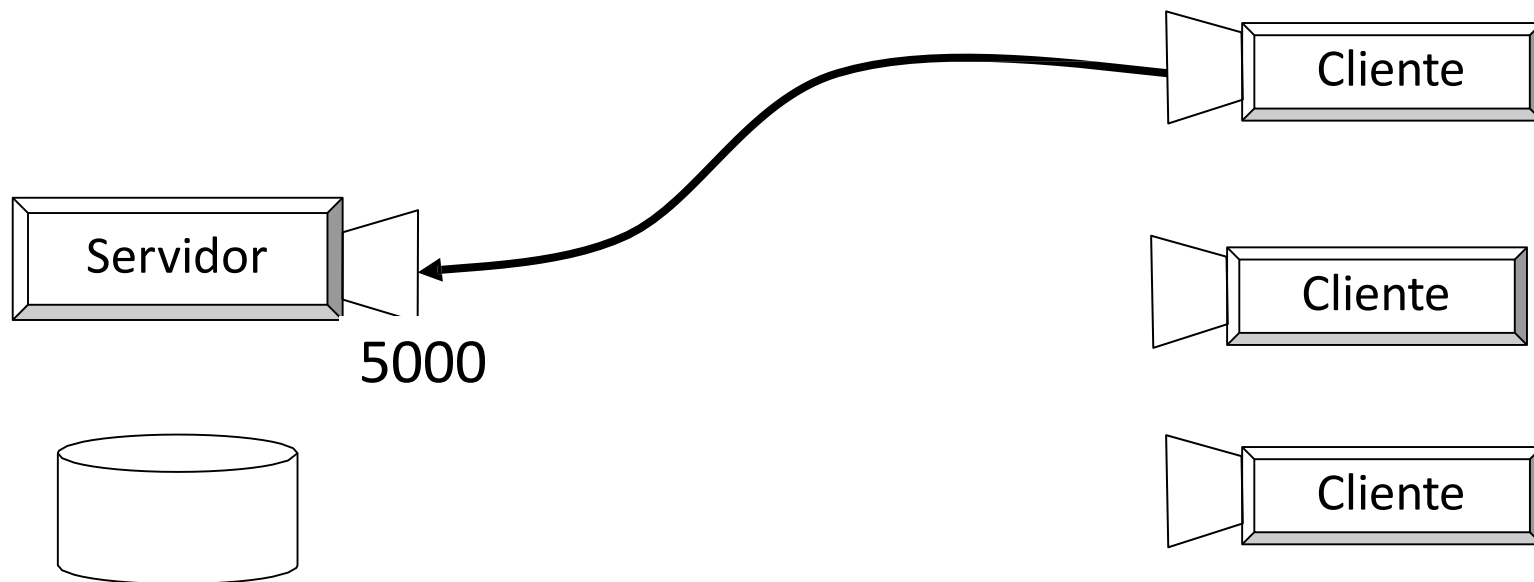
Qué pasa cuando varios clientes tratan de **conectarse** al mismo tiempo a un **servidor**

- Una forma es ir atendiéndolos de a uno en un ciclo: como en el programa que atiende pedidos de archivos
 - Se **acepta** una **conexión**
 - Se **lee** la **petición**
 - Se **lee** desde el **archivo** y se escribe en el socket hasta encontrar una marca de fin de archivo
- A este tipo de servidores se les llama **servidores iterativos**
- El problema es que todo cliente tiene que **esperar** su **turno** para ser atendido Si uno de ellos pide un archivo muy grande los demás tienen que **esperar** La mayor parte de la **espera** es debido a **operaciones** de IO, hay capacidad de CPU **ociosa** !



Servidores Iterativos

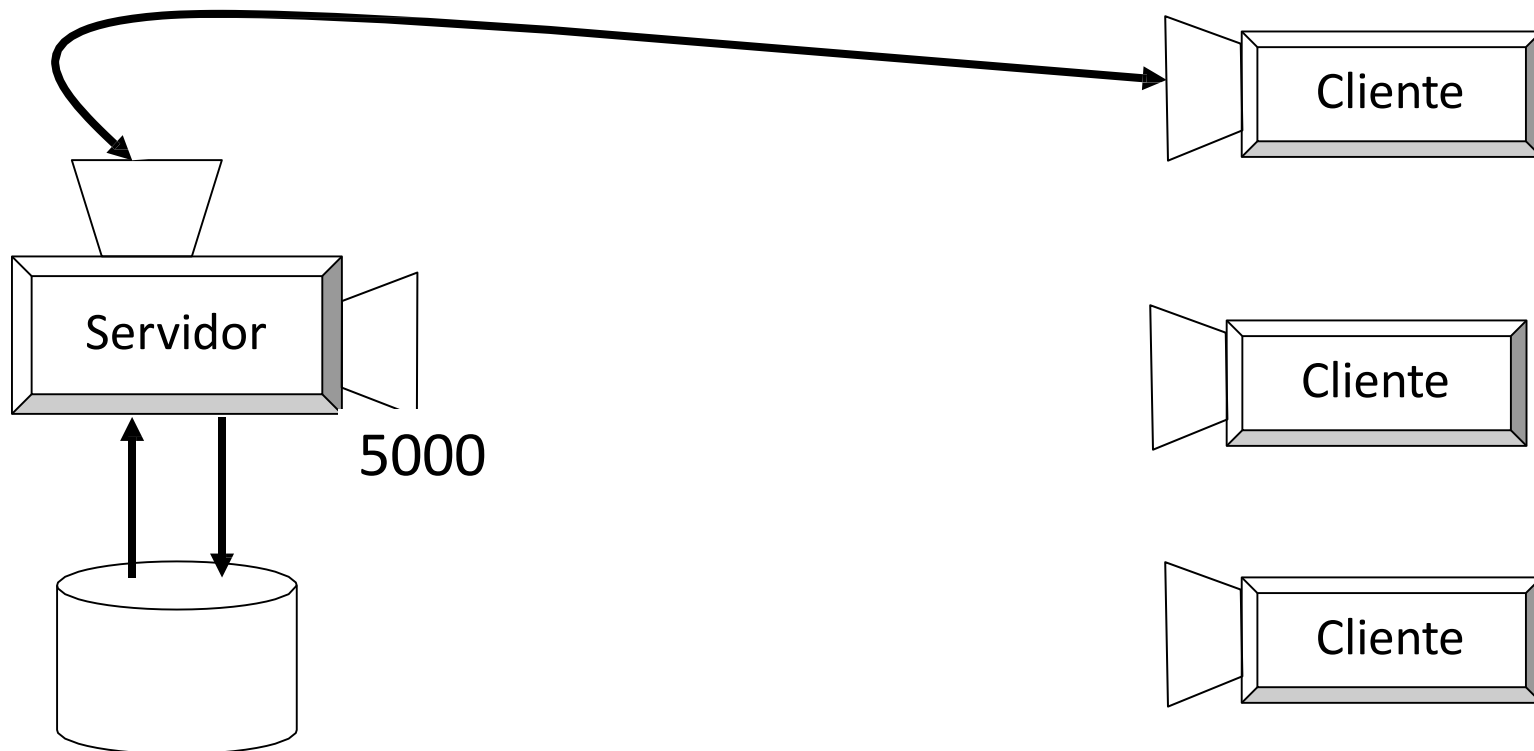
Un servidor **secuencial** (iterativo) **atendiendo** a más de un **cliente**





Servidores Iterativos

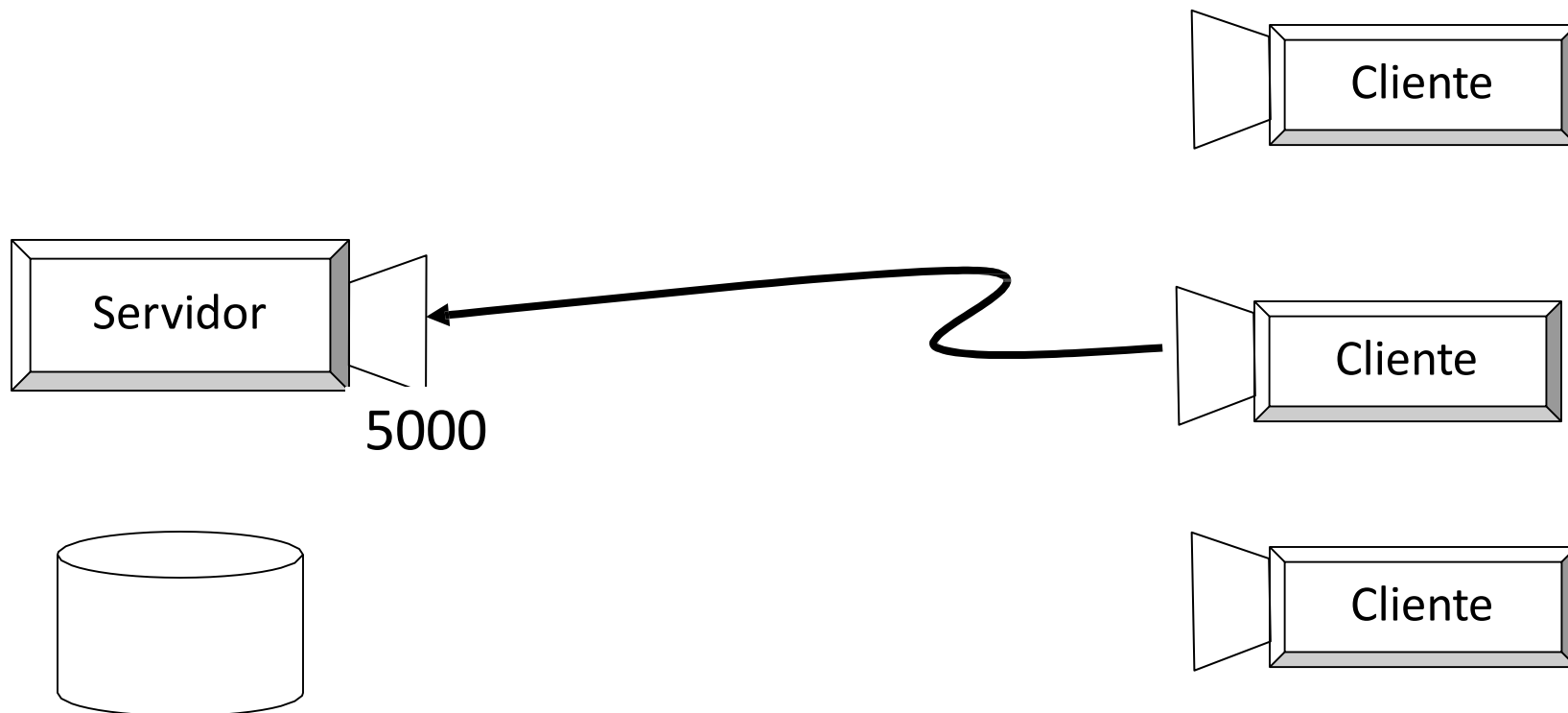
Durante la conversación no puede oír por el puerto 5000





Servidores Iterativos

Sólo después de efectuar la transmisión se pone a escuchar de nuevo por el 5000





Servidor Concurrente

- Un servidor concurrente atiende a varios clientes al mismo tiempo.
- Más aún, mientras está atendiendo sigue escuchando
- Se trata de crear un nuevo proceso o línea de ejecución cada vez que un cliente “llega” a pedir un servicio.
- La alternativa es crear procesos hijos (fork) que atiendan cada solicitud en lo individual.



Ejercicios

1. Desarrolle un programa que ofrezca el servicio de hora siempre y cuando el cliente se encuentre en nuestra lista de direcciones validas.
2. Desarrolle una programa cliente donde este envíe una cadena de caracteres a un programa servidor concurrente y este regrese otro cadena de caracteres con la siguiente información:
 - Numero de caracteres
 - Numero de palabras
 - Numero de líneas

La información deberá se mostrada por el cliente.