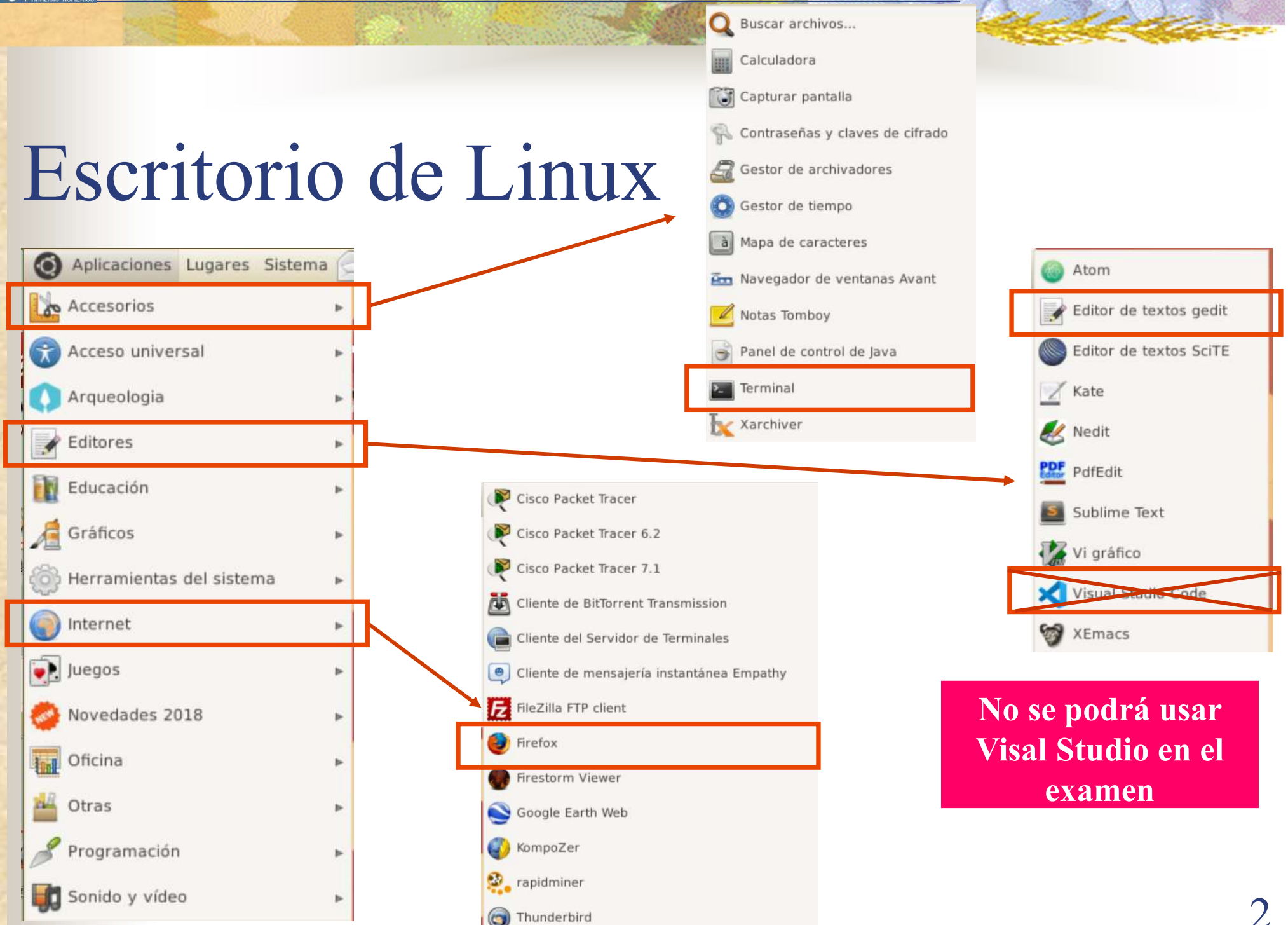
Práctica 0

Eva Lucrecia Gibaja Galindo
Dpto. Informática y Análisis Numérico

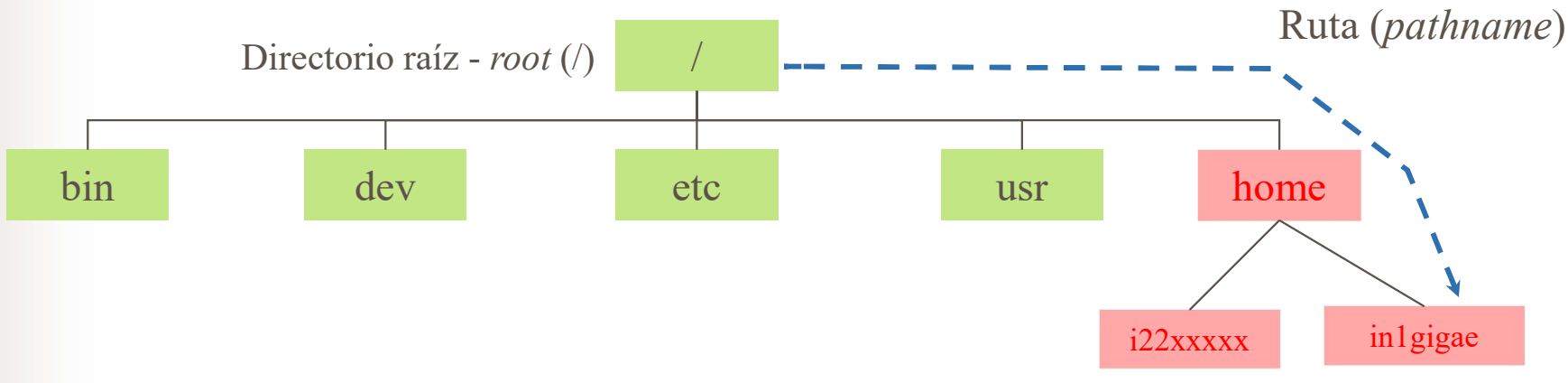
Escritorio de Linux



No se podrá usar Visal Studio en el examen

Ficheros y directorios

- **Fichero**. Colección de información que se almacena en disco
  - Organizados en directorios y subdirectorios forman un sistema jerárquico de archivos
- **Directorio** (carpeta). Ficheros especiales que permiten agrupar ficheros dentro de el. Pueden anidarse dando lugar a subdirectorios
- Directorios básicos en Linux:
  - **/.** Directorio raíz
  - **bin**. Programas utilizados en la inicialización del sistema
  - **dev**. Ficheros de dispositivo
  - **etc**. Ficheros de configuración y programas de administración
  - **usr**. Programas, librerías, etc. accesibles por la mayoría de usuarios
  - **home**. Cuentas de usuarios. Ahí están las carpetas i22xxxxx



Ficheros y directorios

- **Nombre de un fichero**. Identifica el fichero y su contenido. Está formado por:
  - *Nombre*. Combinación de números y caracteres
    - Linux es *casesensitive (Hola.c <> hola.c)*
    - *Recomendación.* No utilizar ni espacios en blanco ni caracteres especiales para los nombres de ficheros
      - trabajo de programación.docx ✗
      - trabajoProgramacion.docx ✓
      - trabajo_programacion.docx ✓
  - *Extensión*. Va precedida de punto (.) y está relacionado con el tipo de fichero
- **Tipos de ficheros**
  - *Ordinarios*. De datos, texto y ejecutables
  - *Directorios*. Contienen otros ficheros
  - *Otros: Vínculos, especiales*
- **Ficheros ocultos**
  - Su nombre comienza por punto (.)
  - No aparecen cuando se muestra el contenido de un directorio
  - Almacenan información que el sistema utiliza automáticamente
  - Para verlos: ls –a

Ficheros y directorios

- Especificación de los caminos (*pathname*):
  - **Camino absoluto**
    - Especifica la localización de un fichero desde el directorio raíz
    - Comienza con *slash* (/)
      - /home/in1gigae/primero/FI/practica1
      - /home/in1gigae/primero/FI/practica1/hola.c
  - **Camino relativo**.
    - Especifica la localización de un fichero con respecto al directorio en que se está trabajando
    - No comienza con *slash* (/)

Símbolo	Significado
~	Directorio *home* del usuario Alt Gr 4; Alt 126
.	Directorio de trabajo actual
..	Directorio padre

Si no consigues escribirlo, utiliza $HOME

  - **Sin especificar camino**
    - Nos estamos refiriendo a un fichero del directorio actual

Manejo de directorios y ficheros

- ls *[-opciones] [fichero]*. Lista el contenido de un directorio
  - -l: Lista los ficheros en formato largo
  - -a: Lista además los ficheros ocultos
- pwd. Muestra el directorio de trabajo actual
- cd *[directorio]*. Cambia el directorio de trabajo
  - *cd ..* Cambia el directorio de trabajo al directorio padre
  - Para volver al *home* del usuario:
    - cd *intro*
    - *cd ~*
    - *cd $HOME*
- mkdir *[directorio]*. Crea directorio
- rmdir *[directorio]*. Borra un directorio. El directorio debe estar vacío
- *cp* [-opciones] *origen destino*. Copia ficheros y directorios
  - -R: Copia recursiva del contenido de un directorio
  - -i: Pedir confirmación para copiar ficheros con idéntico nombre
- ./programa.exe Ejecuta un archivo ejecutable en el directorio actual
- *. Carácter especial. Sustituye a cualquier cadena de caracteres

Manejo de directorios y ficheros

- *mv* [-opciones] *origen destino*. Renombra ficheros y mueve ficheros entre directorios
  - -i: Pedir confirmación para mover copiar ficheros
- *rm [-opciones] fichero* . Borra un fichero o directorio.
  - -R: Borrado recursivo del contenido de un directorio (no tiene que estar vacío)
  - -i: Pedir confirmación para borrar
- *more fichero*. Visualiza en pantalla el fichero
  - q: salir
- *cat fichero*. Visualiza en pantalla el fichero
- *cat > fichero*. Crea un fichero de texto con el contenido que escribamos en el terminal
  - Termina de escribir el fichero con ctrl+D
- *man orden*. Visualiza ayuda sobre orden o instrucción de C
- *mandato &*. ejecuta en segundo plano el mandato

Protección de ficheros y directorios

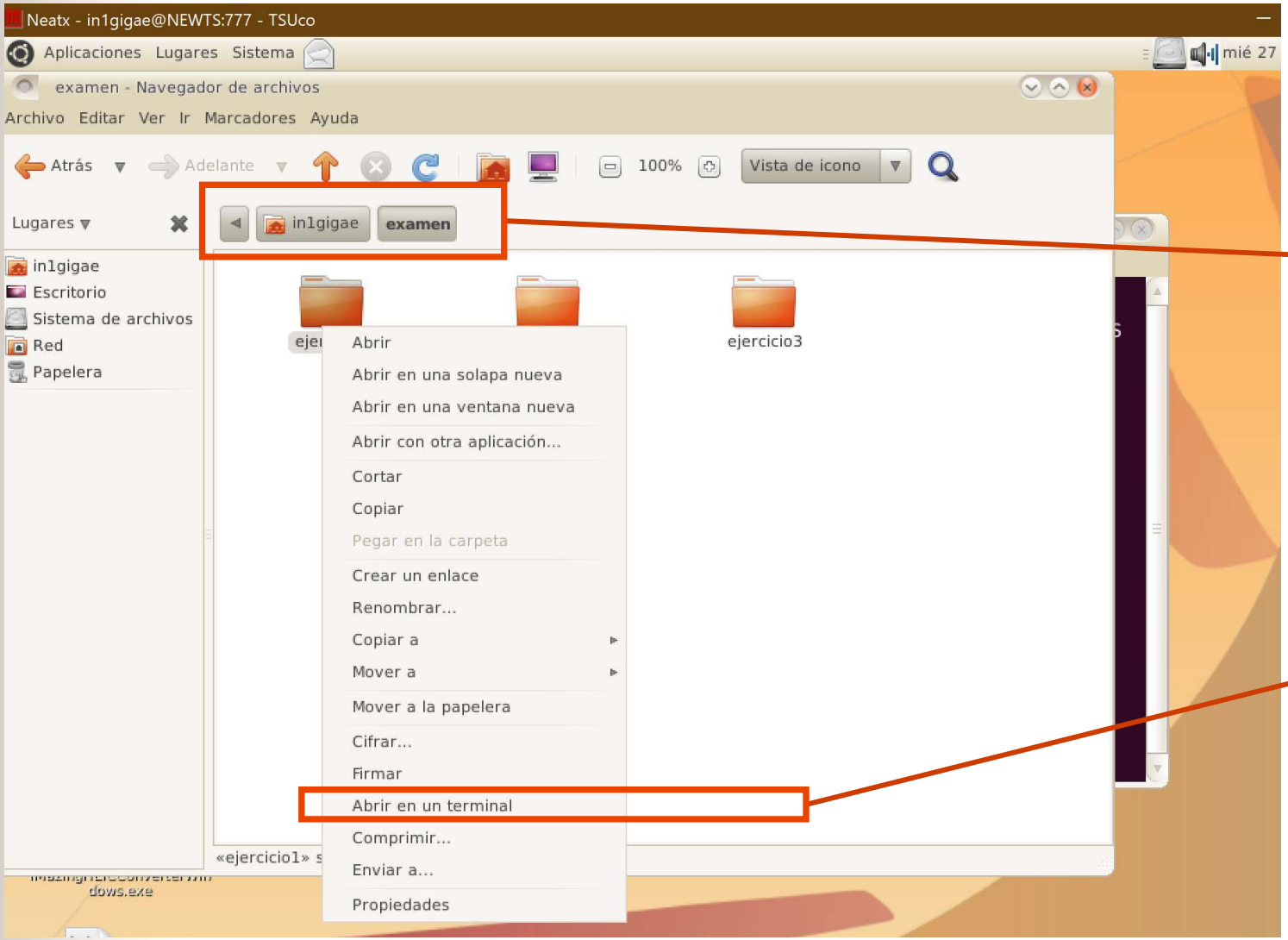
- chmod *[-opciones] modo fichero*. Permite cambiar los permisos de un fichero o directorio
  - -R: recursivo para todos los subdirectorios
  - ls –l para ver los permisos

- **Modo**:
  - Modo relativo: *quien código permisos*
    - quien: u, g, o, a (*owner*, *group*, *others*, *all*)
    - código: +/-
    - permisos: r,w,x
      - chmod u+rw nombreFichero
      - chmod ug+rw nombreFichero
  - Modo absoluto: 3 dígitos decimales
    - chmod 700 nombreFichero
    - Primer dígito: permisos de usuario
    - Segundo dígito: permisos de grupo
    - Tercer dígito: permisos del resto

Binario	Decimal	Permiso
000	0	ninguno
001	1	X
010	2	W
011	3	WX
100	4	R
101	5	RX
110	6	RW
111	7	RWX

- **Permisos:**
  - Lectura (*r*): visualizar fichero o listar directorio
  - Escritura (*w*): editar fichero, borrar y crear ficheros
  - Ejecución (*x*): ejecutar un fichero, entrar en directorio

Abrir una carpeta en un terminal



Ponernos encima de la ruta y pulsar CTRL+L

Pinchar en la carpeta y con botón derecho "Abrir en un terminal"

Estructura básica de un programa C

```
Directivas del preprocesador
•    #include
•    #define

Declaraciones globales:
*    prototipos de funciones
*    variables globales //No se deben utilizar

int main()
{
   Declaración de variables locales

   Sentencias ejecutables

   return 0;
}

Definición de otras funciones
```

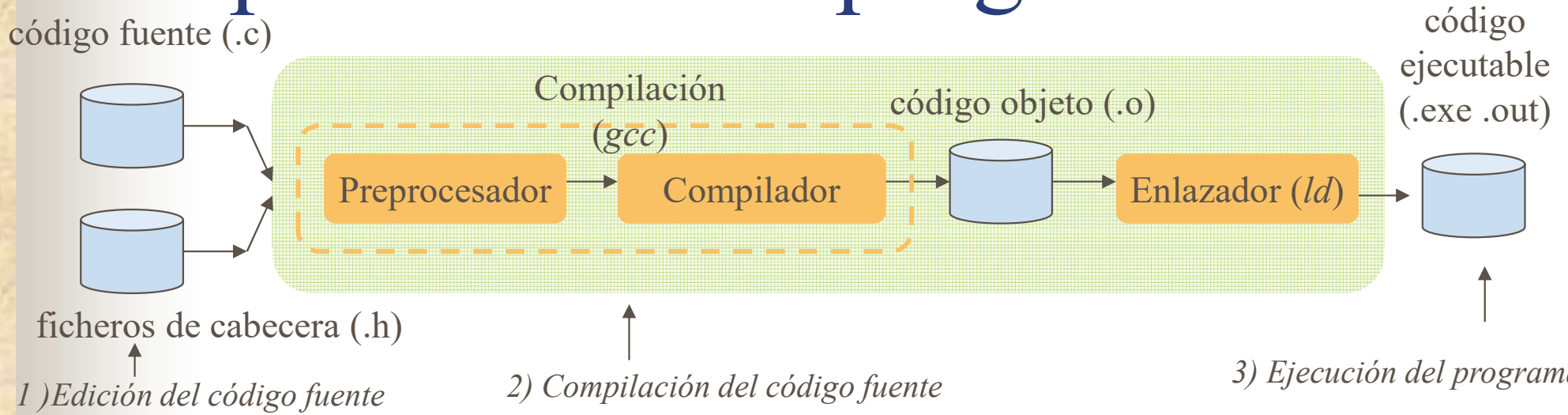


```
# include <stdio.h>

int main()
{
   printf("Hola mundo");
   return 0;
}
```

•guardar el código fuente con extensión .c
•hola.c, hola.o y hola.exe son tres ficheros diferentes

Compilación de un programa C



- Llamada implícita al enlazador. No poner la opción -c
  - `gcc hola.c` → compila y genera *`a.exe`* (Windows) o bien `a.out` (Linux)
  - `gcc hola.c -o hola.exe` → compila y genera ejecutable *`hola.exe`*
- Activar todos los *warnings*
  - **`gcc -Wall hola.c -o hola.exe`**
- **Ejecutar el *.exe* desde el terminal**
  - **`./hola.exe`**
- Genera solo el correspondiente .o (lo utilizaremos cuando veamos bibliotecas)
  - `gcc -c hola.c` → compila y genera `hola.o`
  - `gcc -c hola.c -o otroNombre.o` → compila y genera `otroNombre.o`

Salida de datos. printf

- Los *especificadores de formato,* se marcan con el carácter **%** y van acompañados de una letra que indica el tipo de la expresión correspondiente

Tipo de dato	Especificador de formato
int	%i %d
float	%f %e(notación científica)
char	%c %d(código ASCII)
cadena de caracteres	%s
long	%ld %u
double	%lf
puntero	%lu %p

```
printf("Cuadrado de X = %f y su raiz = %.3f",x*x, sqrt(x));
```

Entrada de datos.scanf

- Es la operación de entrada de datos más común y se encuentra en la biblioteca `stdio`

```
scanf("<especif. formato>", &<variable>);
```

  - `scanf("%i",&salario);` espera a que el usuario introduzca un valor entero `(int)` en el teclado y, cuando se pulsa la tecla `Intro`, lo almacena en la variable `salario`
  - `scanf("%f",&retencion);` espera a que el usuario introduzca un valor real en el teclado y lo almacena en `retencion`
  - `scanf("%c",&opcion);` espera a que el usuario introduzca un carácter en el teclado y lo almacena en `opcion`
  - `scanf("%s", nombre);` espera a que el usuario introduzca una cadena de caracteres por el teclado y lo almacena en `nombre`
  - **Ejercicio: Descargar y hacer el ejemplo 1**

Ojo!! con %s no se pone &

Biblioteca estándar de C

- Conjunto de funciones para realizar entrada y salida de datos, administración de memoria, manipulación de cadenas, etc.
  - ***<stdio.h>*** Funciones de E/S
  - ***<string.h>*** Funciones para cadenas
  - ***<math.h>*** Funciones matemáticas. En este caso habrá que compilar con la opción ***-lm***
  - ***<stdlib.h>*** Reserva de memoria, conversión de datos, números aleatorios
  - ***<limits.h>*** ***<float.h>***. Límites de implementación, INT_MAX, INT_MIN, etc.
  - ***<time.h>*** Funciones de fecha y hora

Ojo!! Para incluir la librería matemática hay que compilar con la opción –lm: gcc –lm casting.c

Funciones de la librería math

Tipo del valor devuelto

Tipo(s) de los parámetros

`int abs(int)`	Regresa el valor absoluto
`double pow(double x,double y)`	Calcula **x** ^{y}. Puede producirse un error de dominio si **x** es negativo e **y** no es un valor entero. También se produce un error de dominio si el resultado no se puede representar cuando **x** es cero e **y** es menor o igual que cero. Un error de recorrido puede producirse.
`double cos(double x)`	Calcula el coseno de **x** (medido en radianes).
`double sin(double x)`	Calcula el seno de **x** (medido en radianes).
`double sqrt(double x)`	Calcula la raíz cuadrada del valor no negativo de **x**. Puede producirse un error de dominio si **x** es negativo.
`double tan(double x)`	Calcula la tangente de **x** (medido en radianes).
`double log(double x)`	Calcula el logaritmo natural (o neperiano). Se produce un error de dominio si el argumento es negativo y un error de recorrido si el argumento es cero.
`double exp(double x)`	Calcula la función exponencial de **x**.
`double log10(double x)`	Calcula el logaritmo en base 10 de **x**. Puede producirse un error de dominio si el argumento es negativo y un error de recorrido si el argumento es cero.
`double ceil(double x)`	La función *ceil* retorna el resultado de la función "techo" de **x**.
`double floor(double x)`	La función *floor* retorna el resultado de la función "suelo" de **x**.

Módulos y Prototipos

- Cada vez que se utiliza un módulo (ya sea de una biblioteca como definida por el programador) es necesario conocer previamente su cabecera o prototipo
  - Esto permite al compilador realizar la comprobación de tipos y número de argumentos
- **Declaración** de una función o prototipo:

```
<tipo devuelto> <nombreFuncion>(<parametros formales>);
```

El prototipo de una función termina en ;

- **Definición** de una función:

```
<tipo devuelto> <nombreFuncion>(<parametros formales>)
{
  Cuerpo de la función
}
```

```
int factorial(int n); //prototipo
int factorial(int n)  //definición
{
    int i, aux;       //variables locales
    aux = 1           //sentencias
    for (i=2; i<=n; i++)
        {aux = aux * i;}
    return aux;       //return
}
```

Módulos y Prototipos

- **Todo el proyecto en un solo fichero .c**
  - Compilación:
    - *gcc main.c –o main.exe*
  - Ejecución
    - *./main.exe*

main.c

#include
prototipos
main()
definición de las funciones

- **Dividir el proyecto en varios .c y varios .h**
  - Compilación:
    - *gcc main.c funciones.c –o main.exe*
  - Ejecución:
    - *./main.exe*

main.c

#include
main()

funciones.c

#include
definición de las funciones

funciones.h

#include
prototipos

- **Ejercicio: Descargar y hacer el ejemplo 2**

Módulos y prototipos

- Los ficheros *.h* NO llevan nunca la implementación (definición) de las funciones
- En la llamada a *gcc* no se ponen los ficheros *.h*
- No haremos NUNCA *include* de ficheros *.c*
- Los *include* de nuestros ficheros de cabecera van entre comillas: `#include "funciones.h"`
- En cada fichero *.h* o *.c* haremos solo aquellos *include* que sean necesarios. Es decir, supongamos que queremos compilar un único fichero, haremos *include* de aquello que debería conocerse para poder llevara a cabo la compilación

Ejemplos de #include

```
#include <stdio.h>
struct punto
{
  int x;
  int y;
}
```

- No es necesario incluir <*stdio.h*>, dentro del fichero no hay ninguna llamada a funciones de *stdio*

```
#include <stdio.h>
struct punto
{
  int x;
  int y;
}
int suma(struct punto);
```

- No es necesario incluir <*stdio.h*>, dentro del fichero no hay ninguna llamada a funciones de *stdio*

```
#include <string.h>
#include <stdio.h>
void main()
{
  printf("Hola");
}
```

- No es necesario incluir <*string.h*>, dentro del fichero no hay ninguna llamada a funciones de *string*

Ejemplos de #include

```
#include <stdio.h>
int suma(int a, int b);
```

- No es necesario incluir <*stdio.h*>, dentro del fichero no hay ninguna llamada a funciones de *stdio*

```
#include <stdio.h>
#include "prototipos.h"
int suma(int a, int b)
{
  return(a+b);
}
```

- No es necesario incluir <*stdio.h*>, dentro del fichero no hay ninguna llamada a funciones de *stdio*
- Si que incluiremos su correspondiente *.h*

```
#include <string.h>
#include "prototipos.h"

#include <stdio.h>
void main()
{
  printf("Hola: %d",
     suma (2,3));
}
```

- No es necesario incluir <*string.h*>, dentro del fichero no hay ninguna llamada a funciones de *string*
- Si es necesario <*stdio.h*> porque estamos llamando a printf
- Si es necesario *prototipos.h* porque estamos llamando a suma

Tablas orientativas. El rango de valores para cada tipo depende de cada SO y compilador

Tipos de datos numéricos

- Enteros

Tipo	Memoria	Rango
`int`	2 *bytes*	[-32768, 32767]
`long`	4 *bytes*	[-2147483648, 2147483647]
`char`	*1 byte*	[0, 255]

- Reales

Tipo	Memoria	Precisión
`float`	4 *bytes*	7 dígitos
`double`	8 *bytes*	15 dígitos

- **Importante: División con decimales**

```
float r;

r = 5.0 * 7.0;      /* Asigna a r el valor  35.0      */
r = 5.0 / 7.0;      /* Asigna a r el valor 0.7142857 */
r = 5.0 / 7;        /* Asigna a r el valor 0.7142857 */
r = 5 / 7;          /* Asigna a r el valor 0          */
```

Tipo de dato carácter

- Un carácter es cualquier elemento de un conjunto de caracteres predefinidos o alfabeto
  - Código ASCII (256 caracteres). Cada carácter tiene un código numerado de 0 a 255
- Dos representaciones:
  - Como número (código ASCII). Podemos utilizar tanto el tipo entero *char* (es el más adecuado), como el tipo entero *int* (de 0 a 255)
  - Como literal de carácter. Se representan entre comillas simples: ’!’ , ’A’, ’a’, ’5’

```
char c;
c = 'A';      // Almacena la 'A'
c = 65;       // Almacena 'A' (código 65)
c = c + 2;    // Almacena la 'C'. Equivale a c= 65+2
c = '7';      // Almacena '7' (código 55)
c = 'A' + 1   // Almacena 66  (carácter 'B')
c = 65 + 1    // Almacena 66
c = 'C' - 2   // Almacena 65, es decir, 'A'
c = 'A' + '1' // Cuidado!!! devuelve 114 (65+49).
c = 7;        // Almacena el carácter cuyo código es 7
//Imprime el caracter y su codigo ASCII
printf("caracter:%c codigo:%d\n", c, c);
```

Código ASCII

Caracteres de Control no imprimibles

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^		`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Tipo lógico

- Se utiliza para representar los valores verdadero
- El tipo de dato lógico no existe en lenguaje C, pero se puede simular:
  - **En C, es verdadero todo aquello distinto de 0. Sólo el 0 es falso**
    - 105: verdadero
    - -1: verdadero
    - 0: falso

```
main () {
    int a, b, c, d;
    a = 1;
    b = 2;
    c = 0;
    d = (a && b); //Asigna verdadero a d
    d = (b && c); // Asigna falso a d
    d = (c || c); // Asigna falso a d
    d = !a;       // Asigna falso a d
    d = (c && (a && b)) || (b && c);
}
```

Vectores

```
#include <stdio.h>
#define TAM 100

void leerVector(float V[], int nEle);
void escribirVector(float V[], int nEle);

int main()
{
  float V[TAM];
  int nEleV=7;

  //Leer nEleV

  leerVector(V, nEleV);
  escribirVector(V, nEleV);

   return 0;
}
```

Tamaño máximo

Declaración

Llamada

Parámetro

```
void leerVector(float V[], int nEle)
{
    int i;
    for(i=0; i<nEle; i++)
    {
      printf("\nV[%d]: ", i);
      scanf("%f", &V[i]);
    }
}

void escribirVector(float V[], int nEle)
{
    int i;
    for(i=0; i<nEle; i++)
    {
      printf("\nV[%d]: %f", i, V[i] );
    }

}
```

Recorrer

Acceso

Ejercicio: Descargar y hacer el ejemplo 4

Tipo de dato cadena de caracteres

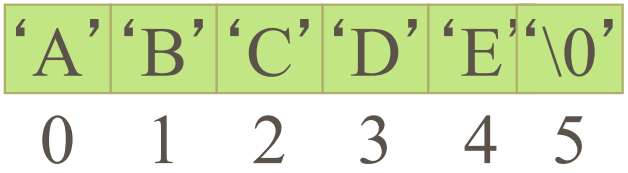
- Conjunto de caracteres, tales como "ABCDEFG"
- Las cadenas se señalan incluyendo un carácter '\0' al final de
- Siempre hay que definir las cadenas con un espacio más del previsto como máxima longitud para el carácter fin de cadena
- Declaración:

  char <nombre> [<longitud>];

  ```
  char cadena [6];
  ```

- Declaración con inicialización:

  ```
  char cadena[6] = "ABCDE";
  ```



  - Con las comillas dobles "" el compilador inserta automáticamente un carácter '\0' al final de la cadena

Entrada de Cadenas de Caracteres

- Leer una cadena. Ejemplo: *char **cadena**[**80**]*;

1. No lee cadenas con espacios en blanco. Ej. al leer la cadena "`hola amigos`" sólo leería la cadena "`hola`"

```
scanf("%s", cadena);
getchar(); //quitar el \n final del buffer de teclado
```

NO necesita el &

2. Lee cadenas con espacios en blanco

```
scanf("%80[^\n]", cadena);
getchar(); //quitar el \n final del buffer de teclado
```

3. Lee cadenas con espacios en blanco. Hay que limpiar el *\n* final que se guarda en la cadena y hacer *#include <string.h>*

```
fgets(cadena, 80, stdin);
cadena[strlen(cadena)-1]='\0'; //limpia \n final
```

4. Lee espacios en blanco, pero dará *warning* o puede no compilar

```
gets(cadena); //No recomendado
```

Manejo de Cadenas de Caracteres

- Las funciones más útiles de C para manejar cadenas se encuentran declaradas en el archivo de cabecera **`string.h`**. Algunas de estas funciones son las siguientes:
- **`int strlen(char *)`**. Devuelve la longitud de la cadena de caracteres (sin incluir el carácter nulo)
- **`int strcmp(char *cad1, char *cad2)`**. Compara ambas cadenas y devuelve:
  - `0` si las dos son idénticas
  - `<0` si la primera cadena precede *alfabéticamente* a la segunda
  - `>0` si la segunda cadena precede *alfabéticamente* a la primera
- **`char *strcpy(char *dest, char *orig)`**
  - Copia `orig` en `dest`. Devuelve `dest`. Se suele ignorar el valor devuelto y se usa como si fuera una función `void`

Ejercicio: Descargar y hacer el ejemplo 3

Estructuras

No se pueden comparar dos estructuras con ==

- Estructura (`struct`) es una colección de uno o más datos (no necesariamente del mismo tipo), agrupados bajo el mismo nombre
- Cuando definimos un `struct` definimos una plantilla, un nuevo tipo de dato
- A continuación, podremos declarar variables que sigan esa plantilla

```
struct punto
{
   float x;
   float y;
}; //No olvidar el ;
int main()
{
   struct punto punto1 ={0.0, 0.0}; //Declaración e Inicialización
   struct punto punto2; //Declaración
   punto2=punto1; //Asignación

   float valor = punto1.x; //Acceso a un campo
}
```

Plantilla, definimos un nuevo tipo de nombre *struct punto*

Variable de tipo *struct punto*

En C podemos asignar dos estructuras

En C podemos devolver una estructura

Paso de estructuras a funciones

```
#include <stdio.h>
struct punto
{
  float x;
  float y;
};

void escribirPunto(struct punto p);
struct punto leerPunto();
int compara(struct punto p1, struct punto p2);
struct punto suma(struct punto p1, struct punto p2);

main()
{
  struct punto p1, p2;
  p1 = leerPunto();
  p2 = leerPunto();
  if (compara(p1, p2))
    printf("\nLos puntos son iguales");
  else
    printf("\nLos puntos son diferentes");
  escribirPunto(p1);
}
```

Definir struct

Declaración

Llamada

Llamada

Devolver struct

```
struct punto leerPunto()
{
    struct punto p;
    printf("Introducir coordenada x: ");
    scanf("%f", &p.x);
    printf("Introducir coordenada y: ");
    scanf("%f", &p.y);
    return p;
}

void escribirPunto( struct punto p )
{
   printf("\np.x: %f\t p.y:%f", p.x, p.y);
}

int compara(struct punto p1, struct punto p2)
{
  if((p1.x==p2.x) && (p1.y==p2.y))
    return 1;
  else
    return 0;
}
```

Parámetro

Arrays de estructuras

```
#include <stdio.h>
#define MAX_ELE 100
struct punto
{
   float x;
   float y;
};
void leeVector(struct punto vector[], int utiles);
void escribeVector(struct punto vector[],
     int utiles);

void main()
{
  int i;
  struct punto vectorPuntos[MAX_ELE];

  struct punto vector2[2]={{-2,-2}, {-3, -3}};

  leeVector(vectorPuntos, MAX_ELE);
  escribeVector(vectorPuntos, MAX_ELE);
  escribeVector(vector2, 2);

}
```

Tamaño máximo

Declaración

Llamada

Parámetro

```
void leeVector(struct punto vector[] , int utiles)
{
  int i;
  for(i=0; i<utiles; i++)
  { printf("\nvector[%d].x: ", i);
    scanf("%f", &(vector[i].x));
    printf("\nvector[%d].y: ", i);
    scanf("%f", &(vector[i].y));
  }
}
void escribeVector(struct punto vector[], int utiles)
{ int i;
  for(i=0; i<utiles; i++)
  {
     printf("\nvector[%d].x: %f vector[%d].y: %f", i,
            vector[i].x ,i, vector[i].y);
  }
}
```

Recorrer

Acceso

Ejercicio: Descargar y hacer el ejemplo 5

Matrices

```
#include <stdio.h>
#define TAM 100

void leerMatriz(float M[][TAM], int nFil, int nCol);
void escribirMatriz(float M[][TAM], int nFil, int
nCol);

int main()
{
  float M[TAM][TAM];
  int nFil=3, nCol=5;

  //Leer nFil, nCol

  leerMatriz(M, nFil, nCol);
  escribirMatriz(M, nFil, nCol);

   return 0;
}
```

Tamaño máximo

Declaración

Lllamada

Parámetro
primer corchete en blanco

```
void leerMatriz(float M[][TAM], int nFil, int nCol)
{
   int i,j;
   for(i=0; i<nFil; i++){
        for(j=0; j<nCol; j++){
           printf("\nM[%d][%d]: ", i, j);
           scanf("%f", &M[i][j]);
        }
   }
}

void escribirMatriz(float M[][TAM], int nFil, int
nCol)
{
   int i,j;
   for(i=0; i<nFil; i++){
        for(j=0; j<nCol; j++){
           printf("\nM[%d][%d]: %f", i, j, M[i][j] );
        }
   }

}
```

Recorrer

Acceso

Ejercicio: Descargar y hacer el ejemplo 6

Tamaño de los datos

- ***sizeof*** es un operador unario en tiempo de compilación que devuelve el **tamaño** en *bytes* de un tipo de dato o variable en memoria
  - Si es un tipo de dato debe ir entre paréntesis
  - Si es una variable no son necesarios los paréntesis
  - Dados `int a, v[7];`
    - `sizeof(int)` devuelve el tamaño de un entero
    - `sizeof(a)`, `sizeof a` devuelven el tamaño de la variable `a`
    - `sizeof v`, `sizeof(v)` devuelven el tamaño total del vector `v`
    - `sizeof v[3]`, `sizeof(v[3])` devuelven el tamaño de un elemento del vector `v`
  - Aplicado sobre un *struct* devuelve su tamaño en memoria
    - Este tamaño no siempre coincide con la suma del tamaño de sus campos ya que puede incluir caracteres de relleno internos y finales utilizados para ajustar los miembros de la estructura a los límites de memoria

Conversión de tipos

- **Conversión implícita de tipos**
  - Los tipos fundamentales pueden ser mezclados libremente en asignaciones y expresiones
  - Las conversiones se realizan **automáticamente**
    - `<DatoGrande> = <DatoPequeño>;`
      - No ocasiona problemas
    - `<DatoPequeño> = <DatoGrande>;`
      - Si `<DatoGrande>` cabe en `<DatoPequeño>` no ocasiona problemas
      - En otro caso podría truncarse, perder información o producirse otros efectos no deseados
- **Conversión explícita de tipos**
  - La solicita específicamente el programador
  - Utiliza el **operador de molde (*cast*)**, hablamos de "*hacer un casting*"
  - Se aplica con frecuencia a los valores de retorno de las funciones

```
(float) i; // convierte i a float
(int) 3.4; // convierte 3.4 a entero
k = (int)1.7* (int)masa;
```

Misceláneo

limits.h	`CHAR_MAX` `CHAR_MIN`	mayor/menor `char`
	`INT_MAX` `INT_MIN`	mayor/menor `int`
	`LONG_MAX` `LONG_MIN`	mayor/menor `long`
float.h	`FLT_MAX` `FLT_MIN`	mayor/menor `float`
	`DBL_MAX` `DBL_MIN`	mayor/menor `double`

- El **rango** de valores para un tipo de dato depende de la cantidad de memoria que dedique el compilador a su representación interna
  - Está definido en *limits.h* y *float.h*
- Una **expresión condicional** tiene el formato:
  - `Condicion ? Expresión1: Expresión2;`
    - El operador **:?** es un operador ternario
  - Suponiendo que `a` vale 7
    - `(a>=5)? printf("Aprobado"): printf("Suspenso");`
    - Escribe *Aprobado*
  - Suponiendo que `m` vale 50
    - `n=(m==99)?1:2;` ⇨ Asigna a `n` el valor 2. Equivale a

```
if(m==99)
   n=1;
else
  n=2;
```

  - Suponiendo que `x` vale 10
    - `y=(x>9)?100:200;` ⇨ Asigna a `y` el valor 100. Equivale a

```
if (x>9)
   y=100;
else
   y=200;
```