

Tema 2. Introducción al Lenguaje de Programación C

Vicente Benjumea García

Programación-I
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 2. Introducción al lenguaje de programación C

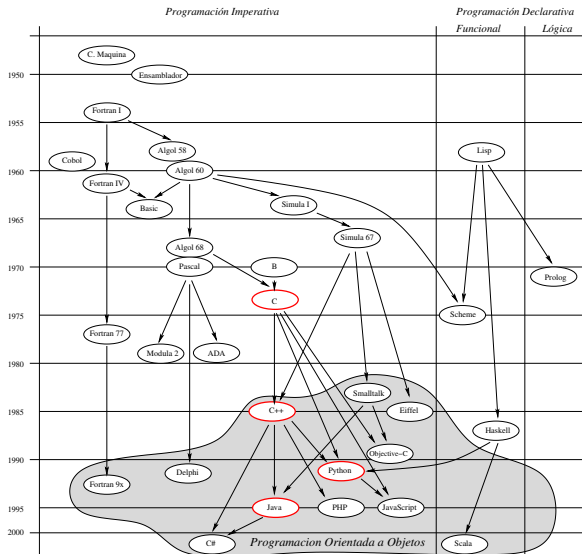
- Introducción al lenguaje de programación C.
- Esquema general de un programa en C. Elementos básicos de C.
- El preprocesador: inclusión de bibliotecas y definición de símbolos.
- Tipos de datos simples predefinidos.
- Variables, asignaciones, operadores aritméticos y conversiones de tipos.
- Entrada y salida de datos básica.
- Expresiones relacionales y lógicas.
- Control del flujo de ejecución.
- Estructuras de selección.
- Estructuras de iteración.
- Tipo puntero, dirección de memoria, y desreferenciación.
- Tipos Enumerados.
- Entrada y salida de datos avanzada.
- Problemas derivados de la implementación de los números.

Esta obra se encuentra bajo una licencia Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) de Creative Commons.

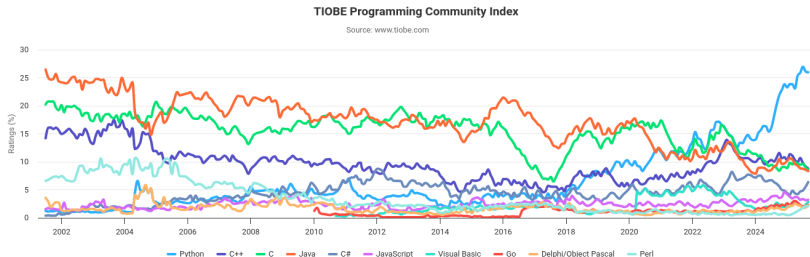


Introducción a C

- Influencias de los lenguajes de programación:



- Clasificación de los lenguajes más *populares* (2025) según *Tiobe* (<https://www.tiobe.com/tiobe-index/>):



Esquema general de un programa en C

Programa

Un **programa** es la implementación de un **algoritmo**, codificado en un lenguaje de programación concreto, adecuado para ser ejecutado en una computadora, con el objetivo de resolver un problema determinado.

En este curso utilizaremos el *lenguaje de programación C*.

Elementos de un Programa

Un programa se compone de los siguientes elementos:

- Las **DECLARACIONES** y **DEFINICIONES** especifican conceptos y elementos.
- Las **ACCIONES** permiten manipular la información.
- Los **OBJETOS** almacenan información, y son manipulados por la acciones.
- Estructuras de **CONTROL** del flujo de ejecución secuencial.
- Mecanismos para la **MODULARIZACIÓN** adecuada del programa.

Esquema general de un programa en C

Esquema General de un Programa en C

- En la asignatura utilizaremos un **subconjunto simple** del lenguaje de programación **C** para describir los algoritmos.

```
// Inclusión de módulos de biblioteca necesarios
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>

// Declaraciones y definiciones de Constantes y Tipos

// Declaraciones y definiciones de Subprogramas

int main()    // Definición del Programa Principal
{
    // Declaración de variables locales

    // Secuencia de sentencias (acciones)
}
```

Esquema general de un programa en C


Ejemplo de un Programa en C


- Diseñe un programa que lea del teclado una cantidad de *pesetas* y muestre en pantalla su equivalente en *euros*, sabiendo que 1 *euro* equivale a 166.386 *pts*.

Algoritmo:

- 1 Mostrar mensaje en pantalla solicitando la cantidad de pesetas a convertir.
- 2 Leer la cantidad de pesetas de teclado y almacenarla en una variable
- 3 Calcular la cantidad de euros equivalente
 - 1 Dividir la cantidad de pesetas entre 166.386.
 - 2 Almacenar el resultado en una variable.
- 4 Mostrar en pantalla la cantidad de euros resultado de la conversión.

Ejecución del Programa

Introduzca cantidad de pts: 500 
500 pts equivalen a 3.005061 euros

El valor 500 lo introduce el usuario desde el teclado
El símbolo  representa la tecla Entrar / Enter

Esquema general de un programa en C


Ejemplo de un Programa en C


- Diseñe un programa que lea del teclado una cantidad de *pesetas* y muestra en pantalla su equivalente en *euros*, sabiendo que 1 *euro* equivale a 166.386 *pts*.

```
#include <stdio.h>
// -- Constantes -----
const double PTS_1_EUR = 166.386;
/* -- Programa Principal ----- */
int main()
{
    int pts;
    printf("Introduzca cantidad de pts: ");
    scanf(" %d", &pts);
    double euros = pts / PTS_1_EUR;
    printf("%d pts equivalen a %lg euros\n", pts, euros);
}
```

El programa C se debe
COMPILAR para generar
un programa ejecutable

Ejecución del Programa

Introduzca cantidad de pts: 500 
500 pts equivalen a 3.005061 euros

El valor 500 lo introduce el usuario desde el teclado
El símbolo  representa la tecla Entrar / Enter

Esquema general de un programa en C

Compilación de un programa C en el entorno de desarrollo

- Un programa codificado en **C** está codificado en un lenguaje de programación de **alto nivel**, que el procesador no puede ejecutar directamente.
- El proceso de **compilación** traduce el programa en **C** a un programa equivalente en **código máquina**, que sí es ejecutable por el procesador.
- El entorno de desarrollo **Code::Blocks** tiene un **editor** y un **compilador de C** integrado, que facilitará realizar esta tarea. Sin embargo, es necesario **configurar** adecuadamente el proceso de compilación con algunas **opciones** importantes que permitirán realizar una **compilación más segura**.
 - -O2: aumenta el nivel de análisis y optimización del código.
 - -Wall: permite detectar errores frecuentes durante la compilación.
 - -Wextra: permite detectar errores adicionales durante la compilación.
 - -Werror: compilación más estricta, trata los *avisos* como **errores**.

Véase el documento correspondiente a la Guía del Entorno de Programación CodeBlocks para saber como se activan.

Esquema general de un programa en C

Elementos Básicos de un Programa en C

- Palabras reservadas: `main`, `const`, `int`, `double`, etc.
- Tipos: `int`, `double`, etc.
- Identificadores: `pts`, `euros`, `PTS_1_EUR`, etc.
- Constantes literales: `166.386`, `" Pts equivalen a "`, etc.
- Constantes simbólicas: `PTS_1_EUR`, etc.
- Variables: `pts`, `euros`, etc.
- Operadores: `=` `+` `-` `*` `/` `%` `>>` `<<` etc.
- Delimitadores: `() { } ; ,` etc.
- Dos clases de comentarios:
 - `//` Comentario de línea
 - `/*` Comentario de bloque `*/`

Esquema general de un programa en C

Las Acciones Manipulan la Información

- Se ejecutan en **SECUENCIA**: cuando se termina de ejecutar la sentencia actual, entonces se ejecuta la siguiente sentencia.
- Se representan mediante **SENTENCIAS** ejecutables:
 - Sentencias de Inicialización y Asignación.
 - Sentencias de Entrada y Salida de Datos.
 - Sentencias de Control del Flujo de Ejecución secuencial.
 - Llamadas a Subprogramas.

Información, Datos, Variables y Constantes

- Las **variables** y **constantes** almacenan y representan la **información**, en forma de **datos**, que manipula el programa.
 - Las **Constantes** representan **información** cuyo **valor** se establece antes de la ejecución del programa, en **Tiempo de Compilación**, y **permanece constante** (no cambia) durante la ejecución del mismo, por ejemplo 1 euro equivale a 166.386 pesetas.
 - Las **Variables** almacenan **información** cuyo **valor** se establece durante la ejecución del programa, en **Tiempo de Ejecución**, y **sí puede cambiar** durante la ejecución del mismo, por ejemplo, el valor de pts y euros.

Esquema general de un programa en C

Ejemplo de un Programa en C


- Desarrolle un programa que lea del teclado una cantidad de *gramos* y muestre en pantalla su desglose en *toneladas*, *kilogramos* y *gramos*.

Análisis de la Solución

- Gramos en 1 Kilogramo: *1000*
- Kilogramos en 1 Tonelada: *1000*
- Gramos en 1 Tonelada: $1000 \times 1000 = 1000000$
- Por ejemplo para un valor de *3254875* gramos:

$$\begin{array}{r} 3254875 \\ 254875 \end{array} \quad \begin{array}{r} 1000000 \\ \hline 3 \text{ (Tn)} \end{array} \quad \begin{array}{r} 254875 \\ (gr) \text{ } 875 \end{array} \quad \begin{array}{r} 1000 \\ \hline 254 \text{ (Kg)} \end{array}$$

Ejecución del Programa

Introduzca la cantidad de gramos: *3254875* 
3254875 gr equivalen a 3 Tn, 254 Kg, y 875 gr

El valor 3254875 lo introduce el usuario desde el teclado

Esquema general de un programa en C

Ejemplo de un Programa en C

- Desarrolle un programa que lea del teclado una cantidad de *gramos* y muestre en pantalla su desglose en *toneladas*, *kilogramos* y *gramos*.

```
#include <stdio.h>
// -- Constantes -----
const int GRAMOS_1_KILOGRAMO = 1000;
const int KILOGRAMOS_1_TONELADA = 1000;
const int GRAMOS_1_TONELADA = GRAMOS_1_KILOGRAMO * KILOGRAMOS_1_TONELADA;
/* -- Programa Principal ----- */
int main()
{
    int gramos_totales;
    printf("Introduzca la cantidad de gramos: ");
    scanf(" %d", &gramos_totales);
    int toneladas = gramos_totales / GRAMOS_1_TONELADA;
    int resto_gramos = gramos_totales % GRAMOS_1_TONELADA;
    int kilos = resto_gramos / GRAMOS_1_KILOGRAMO;
    int gramos = resto_gramos % GRAMOS_1_KILOGRAMO;
    printf("%d gr equivalen a %d Tn, %d Kg, y %d gr\n",
           gramos_totales, toneladas, kilos, gramos);
}
```

El preprocesador: incl. bibliotecas y definición de símbolos

- El preprocesador **transforma**, de forma automática, el código fuente **antes** de la compilación.

Directivas de inclusión de bibliotecas y definición de símbolos

- #include** <header-file.h>
incluye, en el código actual, el contenido del **fichero del sistema** especificado.
- #include** "header-file.h"
incluye, en el código actual, el contenido del **fichero local** especificado.
- #define** *SÍMBOLO* *valor-de-sustitución*
reemplaza, en el código actual, el *SÍMBOLO* por el *valor de sustitución*. Se recomienda poner el *valor de sustitución* **entre paréntesis**.

```
// fichero: ejemplo.h
#ifndef ejemplo_h__
#define ejemplo_h__ 1

#define PI (3.14)
struct Fecha {
    int dia, mes, anyo;
};
```

```
// fichero: ejemplo.c
#include "ejemplo.h"

#define E (2.71)

int main()
{
    double valor = PI*E;
```

```
// resultado del preprocesador

struct Fecha {
    int dia, mes, anyo;
};
double media(int a, int b);

int main()
```

Tipos de datos simples predefinidos

Tipo de Datos

- Define el **conjunto de valores** que puede tomar el objeto.
- Determina las **operaciones** que se pueden aplicar al objeto.
- Determina el **espacio** que es necesario para almacenar su valor en memoria.
- Determina la **interpretación** del valor binario almacenado en memoria.

Tipos de Datos Simples o Escalares

- Formados por elementos **indivisibles** y **ordenados**, es decir, les son aplicables los operadores relacionales (== != < > <= >=).
- Predefinidos: **bool**, **char**, short, unsigned short, **int**, unsigned, long, unsigned long, float, **double**, long double.
- Definidos por el programador: **enum** (tipo enumerado).

Tipos de Datos Compuestos o Estructurados (Tema 4)

- Definidos por el programador:
 - Estructuras (registros).
 - Arrays y Cadenas de Caracteres.

Tipos de datos simples predefinidos

Tamaño de los tipos de Datos simples predefinidos

Tipo	Valores	Tamaño	Intervalo Valores	printf	scanf
bool	Lógicos	1 byte	false true	%d	-
char	Caracteres	1 byte	Tabla ASCII (256 valores)	%c	%c
short	Enteros	2 bytes	±32767	%hd	%hd
unsigned short	Naturales	2 bytes	+65535	%hu	%hu
enum	Enumerados	4 bytes	Valores de la enumeración	%d	-
int	Enteros	4 bytes	±2147483647	%d	%d
unsigned	Naturales	4 bytes	+4294967295	%u	%u
long	Enteros	4/8 bytes	±9223372036854775807	%ld	%ld
unsigned long	Naturales	4/8 bytes	+18446744073709551615	%lu	%lu
long long	Enteros	8 bytes	±9223372036854775807	%lld	%lld
unsigned long long	Naturales	8 bytes	+18446744073709551615	%llu	%llu
float	Reales	4 bytes	±1.17549e±38 (7 dec.)	%g, %f, %e	%g, %f, %e
double	Reales	8 bytes	±1.79769e±308 (15 dec.)	%lg, %lf, %le	%lg, %lf, %le
long double	Reales	10/16 bytes	±1.18973e±4932 (18 dec.)	%Lg, %Lf, %Le	%Lg, %Lf, %Le

- El operador `sizeof(tipo)` devuelve el número de bytes necesario para representar un valor del tipo especificado.

```
printf("Tamaño del tipo char: %d\n", (int)sizeof(char));
```

- Nota:** es necesario incluir el fichero `<stdbool.h>` para utilizar el tipo `bool` y los

Tipos, Constantes, Variables y Asignaciones

Constantes y Variables

- **Almacenan y representan los datos** que manipula el programa.
 - **Constantes:** representan información cuyo **valor** se establece **antes de la ejecución** del programa, y no cambia durante la ejecución del mismo.
 - **Variables:** almacenan información cuyo **valor** se establece **durante la ejecución** del programa, y sí puede cambiar durante la ejecución del mismo.
-
- **Nombre** (identificador): identifica cada entidad de nuestro programa y permite referenciarlo. Comienza por **letra**, seguida por **letras, dígitos** y **_**
 - **Tipo:** que define las características de los datos que son manipulados.

<code>bool</code>	<code>char</code>	<code>int</code>	<code>double</code>
Lógicos	Caracteres	N ^{os} Enteros	N ^{os} Reales
false y true	Tabla ASCII	±2147483647	±1.79769e±308
1 byte	1 byte	4 bytes	8 bytes

Constantes Literales, Definidas y Simbólicas

- Las **constantes literales** aparecen representadas con su valor explícito en el programa. Normalmente, salvo excepciones, su uso **no** es recomendable.
`media = (num1 + num2) / 2.0; // 2.0 es constante literal`
- El preprocesador permite **definir símbolos** asociados con valores. Se recomienda definir su valor **entre paréntesis**, en otro caso pueden producir problemas. Se suelen definir en la zona de constantes al principio del programa.

```
#define NOMBRE ( valor )
```

- Las **constantes simbólicas** asocian un **nombre** y un **significado** a un determinado valor constante del tipo especificado. Se suelen definir en la zona de constantes al principio del programa, y su ámbito y tiempo de vida abarca todo el programa.

```
const tipo NOMBRE = valor ;
```

Ejemplo

```
#define LIMITE      (30)
const bool DEPURACION = true;
const char LETRA = 'a';
const int MAXIMO = 1000;
const double ERROR_PRECISION = 1.0e-6; // 1.0E-6 equivale a 1.0×10-6
```

Variables

- Una **variable almacena un valor** según su **tipo**. El valor almacenado **puede cambiar** durante la ejecución del programa. De todos los valores que especifica su tipo, en un momento dado, una variable **sólo almacena uno** de ellos.
- Las **variables locales automáticas** se definen **dentro del cuerpo** de los **subprogramas** y de **main**. Su ámbito y tiempo de vida abarca el bloque en el que están definidas.
- Las **variables globales** se definen **fuera del cuerpo** de los **subprogramas** y de **main**. Su ámbito y tiempo de vida abarca todo el programa.
- **Se recomienda que NO se DEFINAN ni UTILICEN VARIABLES GLOBALES.**
- Una variable tiene asignada una zona de memoria donde se almacena el valor que toma en un momento dado. Se puede especificar el valor que almacenará **inicialmente**, en otro caso, el valor que almacena queda **inespecificado**.

```
tipo nombre ;
```

```
tipo nombre = expresión ;
```

Ejemplo

```
int main() {  
    bool encontrado = false; // valor inicial false  
    char letra = 'z';        // valor inicial 'z'  
    int cnt_1, cnt_2;        // valor inicial inespecificado  
    int maximo = 56;         // valor inicial 56  
    double media;           // valor inicial inespecificado  
}
```

Inicialización, Asignación y Expresiones Aritméticas

- La **inicialización** asigna a una **variable** un valor **inicial** resultado de evaluar una expresión.
- La **asignación** asigna a una **variable** un **nuevo valor** resultado de evaluar una expresión. El valor que estaba anteriormente almacenado se **pierde**.

```
nombre_de_variable = expresión ;
```

- La evaluación de una expresión se realiza según unas reglas de precedencia y de asociatividad.

de mayor a menor	Precedencia	Asociat.
paréntesis	()	Izq*
unarios	! - (tipo)	Dch
multiplicativos	* / %	Izq
sumatorios	+ -	Izq
relacionales	< > <= >=	Izq*
igualdad	== !=	Izq*
lógico y	&&	Izq
lógico o		Izq

```
int main()
{
    int x1 = 4 + 2 * 6;

    x1 = 2 + x1 * 3 / 2 - 3;

    x1 = 2 + (x1 * 3) / 2 - 3;
    x1 = 2 + ((x1 * 3) / 2) - 3;
    x1 = (2 + ((x1 * 3) / 2)) - 3;
    x1 = ((2 + ((x1 * 3) / 2)) - 3);
}
```

- Nótese la diferencia entre los símbolos de **asignación** (=) y de **igualdad** (==).
- A todos los tipos simples (incluido el **char**) se le pueden aplicar los operadores relacionales (== != < > <= >=).

Orden de Precedencia y Asociatividad de los Operadores

de mayor a menor	Precedencia	Asoc.
paréntesis, acceso	() [] -> .	Izq
unarios	! + - (tipo) ++ -- * & ~	Dch
multiplicativos	* / %	Izq
sumatorios	+ -	Izq
desplazamientos bits	<< >>	Izq
relacionales	< > <= >=	Izq*
igualdad	== !=	Izq*
bits AND	&	Izq
bits XOR	^	Izq
bits OR		Izq
lógico AND	&&	Izq
lógico OR		Izq
condicional (ternario)	?:	Dch
asignación	= += -= *= /= %= &= ^= = <= >=	Dch
coma	,	Izq

Operadores

Operador	Significado
(<code>expr</code>)	Paréntesis, modifica el orden de evaluación.
<code>! X</code>	Negación lógica unaria (véase tabla de verdad)
<code>- X</code>	Menos unario, negativo, cambio de signo.
<code>X * Z</code>	Multiplicación aritmética.
<code>X / Z</code>	División aritmética (<code>entera</code> o <code>real</code>).
<code>X % Z</code>	Resto de la división aritmética entera (módulo).
<code>X + Z</code>	Suma aritmética.
<code>X - Z</code>	Resta aritmética.
<code>X > Z</code>	Mayor, true si <code>X</code> mayor que <code>Z</code> , false en otro caso.
<code>X >= Z</code>	Mayor o igual, true si <code>X</code> mayor o igual que <code>Z</code> , false en otro caso.
<code>X < Z</code>	Menor, true si <code>X</code> menor que <code>Z</code> , false en otro caso.
<code>X <= Z</code>	Menor o igual, true si <code>X</code> menor o igual que <code>Z</code> , false en otro caso.
<code>X == Z</code>	Igualdad, true si <code>X</code> igual a <code>Z</code> , false en otro caso.
<code>X != Z</code>	Distinto, true si <code>X</code> distinto a <code>Z</code> , false en otro caso.
<code>X && Z</code>	Conjunción (Y)/(AND) lógica (véase tabla de verdad).
<code>X Z</code>	Disyunción (O)/(OR) lógica (véase tabla de verdad).

Sentencias de Asignación Especiales

```
int main()
{
    int x1 = 4 + 2 * 6; // Inicialización (16)

    ++x1;                // x1 = x1 + 1;

    --x1;                // x1 = x1 - 1;

    x1++;                // x1 = x1 + 1;

    x1--;                // x1 = x1 - 1;

    x1 += expr;          // x1 = x1 + ( expr );

    x1 -= expr;          // x1 = x1 - ( expr );

    x1 *= expr;          // x1 = x1 * ( expr );

    x1 /= expr;          // x1 = x1 / ( expr );

    x1 %= expr;          // x1 = x1 % ( expr );
}
```

Conversiones de Tipo

- En las expresiones, cuando se evalúa una operación aplicada sobre **operandos de tipos distintos**, se realiza una conversión de tipo **automática** al tipo simple más **amplio** de los operandos.

• **bool** ▶ **char** ▶ short ▶ enum ▶ **int** ▶ unsigned ▶ long ▶ unsigned long ▶ float ▶ **double** ▶ long double

```
double a = 5 / 2;    // valor de a ? -> 2.0 = (5 / 2)    división entera
double b = 5 / 2.0;  // valor de b ? -> 2.5 = (5.0 / 2.0) división real
```

- En la **asignación**, el valor se convierte de forma **automática** al tipo de la variable.
- En la evaluación de expresiones, a veces nos puede interesar que un determinado operando o expresión sea **evaluado** a un valor con un **tipo distinto al suyo propio**. En estos casos, se realiza una **conversión explícita** de tipos (*casting*). Para ello, se precede el valor o expresión entre paréntesis por el símbolo del tipo, **entre paréntesis**, al que queremos convertir:

```
int    a = (int>true;    // produce el entero 1
bool   b = (bool)0;      // produce el valor lógico false

int    c = (int)'a';      // produce el entero 97    (según tabla ASCII)
char   d = (char)65;      // produce el carácter 'A' (según tabla ASCII)

double i = (double)2;     // produce el real 2.0
int    j = (int)3.6;      // produce el entero 3 (pierde decimales)
```

- La precedencia del *casting* es muy alta, por lo que es necesario utilizar paréntesis en el caso de expresiones.

```
int    a = (int)(7.8 / 2.4); // produce el entero 3    (int)(3.25)
double b = (int)7.8 / 2.4;   // produce el double 2.9166 (7.8 / 2.4)
```


Conversiones de Tipo. Ejemplo

- ¿ Cuales serán los valores que tomen las variables z1, z2 y z3 ?

```
int main()
{
    int    x  = 1;
    int    y  = 2;
    double z1 = x / y * 1e6;
    double z2 = (double)(x / y) * 1e6;
    double z3 = (double) x / y * 1e6;
}
```

La división es una operación propensa a errores

- Atención a la **división entera** en vez de la **división real**, y a la **división entre cero**.
 - La **división entera** produce el **cociente sin decimales**.
 - La **división entera entre cero** aborta la ejecución del programa.
 - La **división real entre cero** produce el valor **Inf** (“Infinito”).
 - La **división real de cero entre cero** produce el valor **NaN** (“Not a Number”).

Entrada y Salida de Datos Básica (I)

- Para realizar operaciones de entrada y salida básicas es necesario **incluir** la biblioteca estándar `<stdio.h>` al principio del programa.
`#include <stdio.h>`
- Se puede utilizar para todos los **tipos simples predefinidos**.
- **No** está definida para el **tipo enumerado** (tipo definido por el programador).
- También se puede utilizar para las **cadenas de caracteres** (véase tema 4).

Salida de Datos printf

La **función de salida** (`printf`) permite **mostrar** en el flujo de salida a pantalla (`stdout`) el valor de una constante, variable o expresión (`\n` \equiv *Salto-de-Línea*).

```
#include <stdio.h>
int main()
{
    char c1 = 'a';
    int x1 = 20, x2 = 30;
    double d3 = 3.14;

    printf("Carácter: %c\n", c1);           // %c para mostrar un caracter
    printf("Número 1: %d ; Número 2: %d\n", x1, x2); // %d para mostrar un número entero
    printf("Número Real: %lg\n", d3);       // %lg para mostrar un número real
}
```

Entrada y Salida de Datos Básica (II)

Entrada de Datos scanf

La **función de entrada** (`scanf`) permite asignar a una **variable** un determinado valor *introducido* desde el flujo de entrada de teclado (`stdin`).

- Nótese el símbolo **ampersand** (&) antes de cada variable.
 - El símbolo **ampersand** (&) representa la dirección en memoria de la variable.

```
int main()
```

```
{
```

```
    int x1, x2;
```

```
    double d1;
```

```
    char c1;
```

x1: 120:	20
x2: 124:	30
d1: 128:	3.14
c1: 136:	'A'

Variable: **x1**, tipo: **int**, dirección de memoria: **120**, valor: **20**

Variable: **x2**, tipo: **int**, dirección de memoria: **124**, valor: **30**

Variable: **d1**, tipo: **double**, dirección de memoria: **128**, valor: **3.14**

Variable: **c1**, tipo: **char**, dirección de memoria: **136**, valor: **'A'**

```
    printf("Introduce un caracter: ");
```

```
    scanf(" %c", &c1);
```

// símbolo %c para leer un caracter

```
    printf("Introduce dos números enteros: ");
```

```
    scanf(" %d %d", &x1, &x2);
```

// símbolo %d para leer un número entero

```
    printf("Introduce un número real: ");
```

```
    scanf(" %lg", &d1);
```

// símbolo %lg para leer un número real

```
}
```

- La cadena que especifica el formato de entrada de `scanf` **NO DEBE TENER ESPACIOS NI \n AL FINAL**.

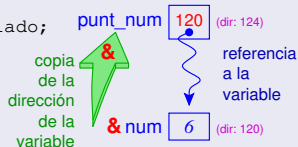
Entrada y Salida de Datos Básica (III)

```
int scanf(const char* formato, int* punt_num)
{
    * punt_num = valor-leido-de-teclado;
}
```

```
int main()
```

```
{
    int num;
```

```
    scanf(" %d", & num );
}
```



Entrada y Salida de Datos Básica (IV)


Ejemplo

```
#include <stdio.h>
int main()
{
    int x1, x2 ;
    printf("Introduce dos números: ");
    scanf(" %d %d", &x1, &x2);



    printf("Núm 1: %d ; Núm 2: %d\n", x1, (3*x2));
    printf("FIN\n"); // \n Salto-de-Línea
}
```


```
#include <stdio.h>
int main()
{
    int x1, x2 ;
    printf("Introduce dos números: ");
    scanf(" %d", &x1);
    scanf(" %d", &x2);
    printf("Núm 1: %d ; Núm 2: %d\n", x1, (3*x2));
    printf("FIN\n"); // \n Salto-de-Línea
}
```

Ejecución del Programa

Introduce dos números: 12 15 
Núm 1: 12 ; Núm 2: 45
FIN

*Los valores 12 y 15 los introduce el usuario desde el teclado
Los datos de entrada se separan por espacios o ENTER
Al final de la entrada se pulsa la tecla ENTER*

Introduce dos números: 12 
15 
Núm 1: 12 ; Núm 2: 45
FIN

*Los valores 12 y 15 los introduce el usuario desde el teclado
Los datos de entrada se separan por espacios o ENTER
Al final de la entrada se pulsa la tecla ENTER
El símbolo  representa la tecla Entrar / Enter*

Ejemplos Simples

- ❶ Programa que lea dos números enteros y los intercambie.
- ❷ Programa que lea un número entero y muestre 1 si el número leído es par, y muestre 0 si es impar.
- ❸ Programa que lea un carácter, suponemos que es una letra mayúscula, la convierta a minúscula y muestre el resultado.
 - Los dígitos del '0' al '9' están consecutivos en la tabla ASCII.
 - Las letras de la 'A' a la 'Z' están consecutivas en la tabla ASCII.
 - Las letras de la 'a' a la 'z' están consecutivas en la tabla ASCII.

Ejemplo 1

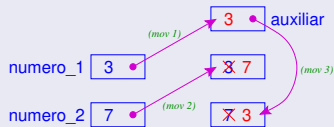
Programa que lea dos números enteros y los intercambie.

```
#include <stdio.h>

int main()
{
    int numero_1, numero_2;
    printf("Introduce dos números: ");
    scanf(" %d %d", &numero_1, &numero_2);

    int auxiliar = numero_1;           // (mov 1)
    numero_1 = numero_2;               // (mov 2)
    numero_2 = auxiliar;               // (mov 3)

    printf("Resultado: %d %d\n", numero_1, numero_2);
}
```



Ejemplo 2

Programa que lea un número entero y muestre 1 si el número leído es par, y muestre 0 si es impar.

```
#include <stdio.h>
int main()
{
    int numero;
    printf("Introduce un número: ");
    scanf(" %d", &numero);
    int resto = numero % 2;
    int espar = (resto == 0); // espar = ((numero % 2) == 0);
    printf("El número %d es par: %d\n", numero, espar);
}
```

Se debe incluir el fichero `<stdbool.h>` para poder utilizar el tipo `bool`.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int numero;
    printf("Introduce un número: ");
    scanf(" %d", &numero);
    int resto = numero % 2;
    bool espar = (resto == 0); // espar = ((numero % 2) == 0);
    printf("El número %d es par: %d\n", numero, espar);
}
```


Ejemplo 3

Programa que lea un carácter, suponemos que es una letra mayúscula, la convierta a minúscula y muestre el resultado.

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

```
int main()
{
```

```
    char letra_mayuscula;
    printf("Introduce una letra mayúscula: ");
    scanf(" %c", &letra_mayuscula);
```

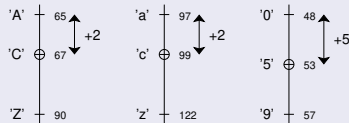
```
    // Suponemos que es una letra mayúscula
```

```
    int distancia = letra_mayuscula - 'A';
    char letra_minuscula = ('a' + distancia);
```

```
    // char letra_minuscula = ('a' + (letra_mayuscula - 'A'));
```

```
    printf("Letra mayúscula: %c -> Letra minúscula: %c\n", letra_mayuscula, letra_minuscula);
```

```
}
```



En nuestros programas, nunca debemos trabajar directamente con los valores de la tabla ASCII, ya que son valores que podemos calcular directamente dentro de nuestro código. Respecto a la tabla ASCII, sólo debemos tener en consideración lo siguiente:

- Los dígitos del '0' al '9' están consecutivos en la tabla ASCII.
- Las letras de la 'A' a la 'Z' están consecutivas en la tabla ASCII.
- Las letras de la 'a' a la 'z' están consecutivas en la tabla ASCII.

Expresiones Lógicas o Booleanas

- Una **expresión lógica** es una expresión que será evaluada a un valor lógico *verdadero* (**true**) o *falso* (**false**). Por ejemplo: `((x + 6) >= 10) && (z < 20)`
- Permiten comprobar el **estado** de la computación en un momento determinado. Son la base para la **toma de decisiones** en nuestros programas.
- La evaluación de una expresión lógica se realiza según unas reglas de precedencia y de asociatividad, y la tabla de verdad de los operadores lógicos.
- Los operadores relacionales (`<` `<=` `>` `>=` `==` `!=`) producen un valor lógico como resultado de la comparación.
- Los operadores lógicos (`!` `&&` `||`) producen un valor lógico como resultado.
- Los operadores lógicos (`&&` `||`) se evalúan en **CORTOCIRCUITO**.

Tabla de verdad de los operadores lógicos

x	y	x <code>&&</code> y	x <code> </code> y	! x
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

Evaluación en Cortocircuito de Expresiones Lógicas

- Los operadores **AND** y **OR** (`&&` , `||`) se evalúan en **CORTOCIRCUITO**.
- Cuando ya se conoce el resultado de la operación lógica tras la evaluación del primer operando, entonces el **segundo operando NO se evalúa**.
 - Para el operador **AND**, cuando el primer operando se evalúa a **false**, entonces el resultado de la operación AND es **false** sin necesidad de evaluar el segundo operando.
 - Para el operador **OR**, cuando el primer operando se evalúa a **true**, entonces el resultado de la operación OR es **true** sin necesidad de evaluar el segundo operando.
- Sin embargo, si tras la evaluación del primer operando no se puede calcular el resultado de la operación lógica, entonces el **segundo operando SÍ se evalúa**.
 - Para el operador **AND**, cuando el primer operando se evalúa a **true**, entonces el resultado de la operación AND es el resultado de evaluar el segundo operando.
 - Para el operador **OR**, cuando el primer operando se evalúa a **false**, entonces el resultado de la operación OR es el resultado de evaluar el segundo operando.

Expresiones Lógicas: Ejemplos

Es necesario incluir el fichero `<stdbool.h>` para utilizar el tipo `bool` y los valores `false` (0) y `true` (1).

- Codifique las siguientes expresiones lógicas.

```
#include <stdbool.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    bool valor1 = ...;           // Toma el valor falso
    bool valor2 = ...;           // Toma el valor verdadero
    bool par = ...;              // True si NUM es par
    bool tres_digitos = ...;     // True si NUM tiene tres dígitos
    bool tres_digitos_par = ...; // True si NUM tiene tres dígitos y es par

    bool primo_10 = ...;         // True si NUM un número primo menor que 10
    bool divisor_100 = ...;      // True si NUM es un divisor de 100
}
```

- Desarrolle un programa que lea un número entero y muestre 1 si tiene cuatro dígitos, y además es capicúa. En otro caso muestre 0.

Expresiones Lógicas: Ejemplo 1

- Codifique las siguientes expresiones lógicas.

```
#include <stdbool.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    bool valor1 = false;
    bool valor2 = true;
    bool par = (num % 2) == 0;
    bool tres_digitos = (num >= 100) && (num <= 999);
    // bool tres_digitos = (100 <= num) && (num <= 999);
    bool tres_digitos_par = tres_digitos && par;
    bool tres_digitos_par = ((num >= 100) && (num <= 999)) && ((num % 2) == 0);
    bool primo_10 = ((num >= 2) && (num <= 3)) || (num == 5) || (num == 7);
    bool divisor_100 = (num != 0) && ((100 % num) == 0); // CORTOCIRCUITO
}
```

- Los operadores relacionales (== != < > <= >=) **NO son asociativos**. Es decir la siguiente expresión lógica para comprobar si un número tiene tres dígitos es **INCORRECTA**.

```
bool tres_digitos = (100 <= num <= 999); // INCORRECTA
```

Expresiones Lógicas: Ejemplo 2

- Desarrolle un programa que lea un número entero y muestre 1 si tiene cuatro dígitos, y además es capicúa. En otro caso muestra 0.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int numero;
    printf("Introduce un número: ");
    scanf(" %d", &numero);

    int digito_1 = numero % 10;
    int digito_2 = (numero / 10) % 10;
    int digito_3 = (numero / 100) % 10;
    int digito_4 = (numero / 1000) % 10;
    bool cuatro_cap = (numero >= 100) && (numero <= 9999)
                      && (digito_2 == digito_3)
                      && (digito_1 == digito_4);

    printf("%d tiene 4 dígitos y es capicúa: %d\n", numero, cuatro_cap);
}
```

Expresiones Lógicas. Equivalencias

$$\begin{aligned}! (! b) &\Leftrightarrow (b) \\! (a \&\& b) &\Leftrightarrow (! a || ! b) \\! (a || b) &\Leftrightarrow (! a \&\& ! b)\end{aligned}$$

$$\begin{aligned}! (x == y) &\Leftrightarrow (x != y) \\! (x > y) &\Leftrightarrow (x <= y) \\! (x < y) &\Leftrightarrow (x >= y)\end{aligned}$$

$$\begin{aligned}\cancel{(b == true)} &\Leftrightarrow (b) \\\cancel{(b == false)} &\Leftrightarrow (! b)\end{aligned}$$

$$\begin{aligned}\cancel{(b != true)} &\Leftrightarrow (! b) \\\cancel{(b != false)} &\Leftrightarrow (b)\end{aligned}$$

$$\begin{aligned}((a \&\& b) || (a \&\& c)) &\Leftrightarrow a \&\& (b || c) \\((a || b) \&\& (a || c)) &\Leftrightarrow a || (b \&\& c)\end{aligned}$$

Expresiones Lógicas. Ejercicios

- Desarrolle un programa que lea de teclado datos dos números enteros (x e y) y un carácter (c), y muestre **1** por cada una de las siguientes propiedades que cumplan, y **0** en caso contrario.

- 1 $x \in \{ 3, 4, 5, 6, 7 \}$
- 2 $x \in \{ 1, 2, 3, 7, 8, 9 \}$
- 3 $x \in \{ 1, 3, 5, 7, 9 \}$
- 4 $x \in \{ 2, 5, 6, 7, 8, 9 \}$
- 5 $x \in \{ 3, 4, 6, 8, 9 \}$, $y \in \{ 6, 7, 8, 3 \}$
- 6 Ni x ni y sean mayores que **10**
- 7 x no sea múltiplo de y
- 8 c es una letra mayúscula
- 9 c es una letra (mayúscula o minúscula)
- 10 c es un alfanumérico (mayúscula, minúscula o dígito)

- Simplifique las siguientes expresiones lógicas:

- 1 $! (x > y) \ \&\& \ (x < z)$
- 2 $! ((x > y) \ \&\& \ (x < z))$
- 3 $! ((x == y) \ || \ (x != z))$
- 4 $! ((x == y) \ || \ ((x < y) \ \&\& \ (x > z)))$
- 5 $(x == y) \ || \ (((x < y) \ \&\& \ (x > z)) == \text{false})$

Control del Flujo de Ejecución de las Sentencias

- El flujo de ejecución de nuestros programas es **secuencial**.
 - Cuando se termina de ejecutar una sentencia, entonces se ejecuta la siguiente.
- Hasta ahora, el flujo de ejecución de las sentencias coincide con el **orden** en el que han sido codificadas.
 - Esto no es suficiente para resolver problemas.
 - Necesitamos nuevas sentencias que permitan **controlar el flujo de ejecución** bajo determinadas **condiciones** (expresiones lógicas).

Control del Flujo de Ejecución de las Sentencias

Secuencia

- Cuando se termina de ejecutar una sentencia, entonces se ejecuta la siguiente.

Sentencias de Selección

- Permiten **seleccionar** la ejecución **alternativa** y **excluyente** de unas sentencias u otras dependiendo del valor de una **expresión lógica**.

Sentencias de Iteración

- **Repiten** la ejecución de una secuencia de sentencias.
- El **número de iteraciones** depende de una **expresión lógica** cuyo valor evoluciona durante la ejecución del cuerpo del bucle.

Anidamientos

- Estas estructuras de control se pueden **anidar**, es decir, se pueden poner unas dentro de otras para que el flujo de ejecución sea el adecuado para resolver el problema.
- El **ámbito y t. de vida de una variable abarca el bloque { }** en el que está definida.
- Cuando se declara una variable **dentro de un bucle**, entonces la variable se y Cien

Ejercicios

- ➊ Programa que lea un número, calcule su valor absoluto y lo muestre en pantalla.
- ➋ Programa que lea dos números enteros y muestre un mensaje indicando si *num1* es divisible por *num2*.
- ➌ Programa que lea un carácter, si el caracter leído es una letra minúscula, la convierte a mayúscula, si el caracter leído es una letra mayúscula, la convierte a minúscula, en otro caso no la modifica. Finalmente, muestra el resultado de la transformación.
- ➍ Desarrolle un programa que lea un número real comprendido entre 0 y 10 y muestre la nota asociada.
 - Matricula de Honor ($n = 10$), Sobresaliente ($9 \leq n < 10$), Notable ($7 \leq n < 9$), Aprobado ($5 \leq n < 7$), Suspenso ($0 \leq n < 5$), Error.
- ➎ Desarrolle un programa que muestre el mayor de tres números leídos de teclado.
- ➏ Desarrolle un programa que muestre el mayor de cuatro números leídos de teclado.
- ➐ Calcular el resultado de una operación aritmética (+ - * /) y dos números leídos de teclado.
- ➑ Desarrolle un programa que lea de teclado tres números (día, mes y año), si la fecha es correcta mostrará el mensaje **Fecha Correcta**, en otro caso mostrará **Error**.

Sentencias de Selección

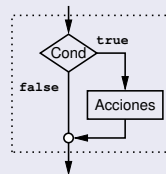
Sentencias de Selección

- Permiten **seleccionar** la ejecución **alternativa** y **excluyente** de unas sentencias u otras dependiendo del valor de una **expresión lógica**.

Sentencia de Selección Simple (if)

- Se evalúa la condición (expresión lógica).
 - Si el resultado es true, entonces se ejecutan las sentencias del **cuerpo del if**.
 - En otro caso, **no se ejecuta nada**.
- Finalmente, se pasa a ejecutar la siguiente sentencia.

```
int main()
{
    // Sentencia de Selección Simple
    if ( EXPR_LOGICA_1 ) {
        // Acciones_1
    }
    // Siguiente sentencia
}
```



Sentencia de Selección Simple (if)

- Calcular el valor absoluto de un número entero.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    if (num < 0) {
        num = -num;
    }

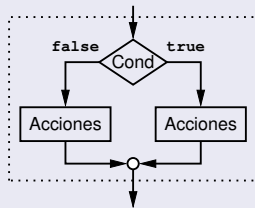
    printf("%d\n", num);
}
```

Sentencias de Selección

Sentencia de Selección Compuesta (if-else)

- Se evalúa la condición (expresión lógica).
 - Si el resultado es true, entonces se ejecutan las sentencias del **cuerpo del if**.
 - En otro caso, se ejecutan las sentencias del **cuerpo del else**.
- Finalmente, se pasa a ejecutar la siguiente sentencia.

```
int main()  
{  
    // Sentencia de Selección Compuesta  
    if ( EXPR_LOGICA_1 ) {  
        // Acciones_1  
    } else {  
        // Acciones_otro_caso  
    }  
    // Siguiente sentencia  
}
```



Sentencia de Selección Compuesta (if-else)

- Calcular si un número es divisible por otro.

```
#include <stdio.h>
int main()
{
    int num1, num2;
    printf("Introduce dos números: ");
    scanf(" %d %d", &num1, &num2);

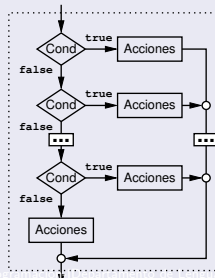
    if ((num2 != 0) && ((num1 % num2) == 0)) {
        printf("%d es divisible por %d\n", num1, num2);
    } else {
        printf("%d no es divisible por %d\n", num1, num2);
    }
}
```

Sentencias de Selección

Sentencia de Selección Compuesta Encadenada (if-else-if)

- Se van evaluando las expresiones lógicas, en el orden especificado, si la evaluación de una expresión logica es **false**, se continúa evaluando la siguiente expresión lógica, hasta que la evaluación de una expresión logica sea **true**, entonces se ejecutan las acciones correspondientes a dicha expresión lógica, y el resto de expresiones lógicas y acciones correspondientes son desechadas y no se ejecutan.
- Si ninguna expresión lógica se ha evaluado a **true**, entonces se ejecutan las acciones correspondientes a la cláusula **else**, en caso de que no exista clausula **else**, entonces no se hace nada.
- Finalmente, se pasa a ejecutar la siguiente sentencia.

```
int main()
{
    // Sentencia de Selección Encadenada
    if ( EXPR_LOGICA_1 ) {
        // Acciones_1
    } else if ( EXPR_LOGICA_2 ) {
        // Acciones_2
    } <...> todas las condiciones que se deseen
    } else if ( EXPR_LOGICA_3 ) {
        // Acciones_3
    } else {
        // Acciones_otro_caso
    }
    // Siguiente sentencia
}
```



Sentencias de Selección. Ejemplo 1

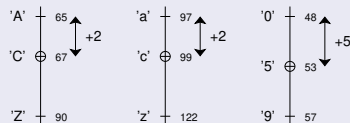
- Programa que lea un carácter, si el carácter leído es una letra minúscula, la convierte a mayúscula, si el carácter leído es una letra mayúscula, la convierte a minúscula, en otro caso no la modifica. Finalmente, muestra el resultado de la transformación.

```
#include <stdio.h>

int main()
{
    char caracter, nuevo;
    printf("Introduce un caracter: ");
    scanf(" %c", &caracter);

    if ('A' <= caracter && caracter <= 'Z') {
        nuevo = ('a' + (caracter - 'A'));
    } else if ('a' <= caracter && caracter <= 'z') {
        nuevo = ('A' + (caracter - 'a'));
    } else {
        nuevo = caracter;
    }

    printf("Resultado: %c\n", nuevo);
}
```



Sentencias de Selección. Ejemplo 2

- Desarrolle un programa que lea un número real comprendido entre 0 y 10 y muestre la nota asociada.

- Matricula de Honor ($n = 10$), Sobresaliente ($9 \leq n < 10$), Notable ($7 \leq n < 9$), Aprobado ($5 \leq n < 7$), Suspenso ($0 \leq n < 5$), Error.

```
#include <stdio.h>
int main()
{
    double nota;
    printf("Introduce la nota: ");
    scanf(" %lg", &nota);

    if ( ! ((nota >= 0.0) && (nota <= 10.0))) {
        printf("Error: 0 <= n <= 10\n");
    } else if (nota == 10.0) {
        printf("Matricula de Honor\n");
    } else if (nota >= 9.0) {
        printf("Sobresaliente\n");
    } else if (nota >= 7.0) {
        printf("Notable\n");
    } else if (nota >= 5.0) {
        printf("Aprobado\n");
    } else {
        printf("Suspenso\n");
    }
}
```

Sentencias de Selección. Ejemplo 3 (v1)

- Desarrolle un programa que muestre el mayor de tres números leídos de teclado.

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

```
int main()
{
    int a, b, c, mayor;
    printf("Introduce tres números: ");
    scanf(" %d %d %d", &a, &b, &c);

    if (a >= b && a >= c) {
        mayor = a;
    } else if (b >= a && b >= c) {
        mayor = b;
    } else {
        mayor = c;
    }
    printf("Mayor: %d\n", mayor);
}
```

Sentencias de Selección. Ejemplo 3 (v2)

- Desarrolle un programa que muestre el mayor de tres números leídos de teclado. *Esta solución es más flexible y adaptable.*

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

```
int main()
{
    int a, b, c, mayor;
    printf("Introduce tres números: ");
    scanf(" %d %d %d", &a, &b, &c);

    mayor = a;
    if (b > mayor) {
        mayor = b;
    }
    if (c > mayor) {
        mayor = c;
    }
    printf("Mayor: %d\n", mayor);
}
```

Sentencias de Selección. Ejemplo 4

- Desarrolle un programa que muestre el mayor de cuatro números leídos de teclado.

```
#include <stdio.h>

int main()
{
    int a, b, c, d, mayor;
    printf("Introduce cuatro números: ");
    scanf(" %d %d %d %d", &a, &b, &c, &d);

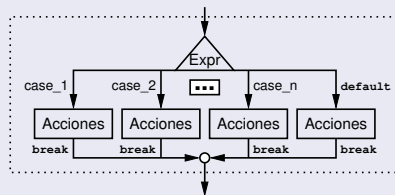
    mayor = a;
    if (b > mayor) {
        mayor = b;
    }
    if (c > mayor) {
        mayor = c;
    }
    if (d > mayor) {
        mayor = d;
    }
    printf("Mayor: %d\n", mayor);
}
```

Sentencias de Selección

Sentencia de Selección Múltiple (switch)

- Se evalúa la expresión y se compara su resultado con los valores de los diferentes **casos**.
 - En caso de que el valor de la expresión sea igual al valor de algún caso, entonces se ejecutan las sentencias del **cuerpo del caso** (hasta el **break**, que no se debe **olvidar**).
 - Si el valor de la expresión no es igual a ningún valor de los casos, entonces se ejecutan las sentencias del **cuerpo del default** (hasta el **break**, que no se debe **olvidar**).
- Finalmente, se pasa a ejecutar la siguiente sentencia.
- La expresión y los valores constantes de los **casos** deben ser del mismo **tipo simple**, los números reales (float, double, long double) no están permitidos.
- NO** está permitido declarar variables dentro de los casos de la sentencia **switch**.

```
int main() {  
    switch ( EXPRESION ) {  
        case CTE_1:  
            // Acciones_CTE_1  
            break;  
        case CTE_2:  
        case CTE_3:  
            // Acciones_CTE_2_CTE_3  
            break;  
        default:  
            // Acciones_otro_caso  
            break;  
    }
```



Sentencias de Selección. Ejemplos

- Desarrolle un programa que lea un número que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

1	->	Lunes
2	->	Martes
3	->	Miércoles
4	->	Jueves
5	->	Viernes
6	->	Sábado
7	->	Domingo
otro caso	->	Error

- Desarrolle un programa que lea una letra que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

l L	->	Lunes
m M	->	Martes
x X	->	Miércoles
j J	->	Jueves
v V	->	Viernes
s S	->	Sábado
d D	->	Domingo
otro caso	->	Error

Sentencias de Selección. Ejemplo 1

- Desarrolle un programa que lea un número que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

```
#include <stdio.h>
int main()
{
    int numero_semana;
    printf("Introduce un número: ");
    scanf(" %d", &numero_semana);

    switch (numero_semana) {
        case 1:
            printf("Lunes\n");
            break;
        case 2:
            printf("Martes\n");
            break;
        case 3:
            printf("Miércoles\n");
            break;
        case 4:
            printf("Jueves\n");
            break;
        case 5:
            printf("Viernes\n");
            break;
        case 6:
            printf("Sábado\n");
            break;
        case 7:
            printf("Domingo\n");
            break;
        default:
            printf("Error\n");
            break;
    }
}
```


Sentencias de Selección. Ejemplo 2

- Desarrolle un programa que lea una letra que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

```
#include <stdio.h>
int main()
{
    char letra_semana;
    printf("Introduce una letra: ");
    scanf(" %c", &letra_semana);

    switch (letra_semana) {
        case 'l':
        case 'L':
            printf("Lunes\n");
            break;
        case 'm':
        case 'M':
            printf("Martes\n");
            break;
        case 'x':
        case 'X':
            printf("Miércoles\n");
            break;
        case 'j':
        case 'J':
            printf("Jueves\n");
            break;
        case 'v':
        case 'V':
            printf("Viernes\n");
            break;
        case 's':
        case 'S':
            printf("Sábado\n");
            break;
        case 'd':
        case 'D':
            printf("Domingo\n");
            break;
        default:
            printf("Error\n");
            break;
    }
}
```

Sentencias de Selección. Ejemplo 3

- Calcular el resultado de una operación aritmética (+ - * /) y dos números leídos de teclado.

```
#include <stdio.h>
#include <stdbool.h>
int main() {
    char operador; double num1, num2, resultado; bool ok = true; // compacto por espacio
    printf("Introduce el operador y dos operandos: ");
    scanf(" %c %lg %lg", &operador, &num1, &num2);
    switch (operador) {
        case '+': resultado = num1 + num2;
                break;
        case '-': resultado = num1 - num2;
                break;
        case '*': resultado = num1 * num2;
                break;
        case '/': if (num2 != 0) {
                    resultado = num1 / num2;
                } else {
                    ok = false;
                }
                break;
        default: ok = false;
                break;
    }
    if (ok) {
        printf("Resultado: %lg\n", resultado);
    } else {
        printf("Error\n");
    }
}
```

Sentencias de Selección. Anidamientos. Ejemplo 1

Programa que lea de teclado una fecha, dada por tres números enteros (día, mes y año), si la fecha es correcta mostrará el mensaje **Fecha Correcta**, en otro caso mostrará **Error**, considerando que:

- ❶ Cualquier valor del año es correcto.
- ❷ Sólo son correctos los meses desde el número **1** hasta el número **12**, ambos inclusive, de tal forma que el número **1** corresponde al mes de **Enero**, el número **2** al mes de **Febrero**, y así sucesivamente hasta el número **12** que corresponde al mes de **Diciembre**.
- ❸ El número correspondiente al día debe ser **mayor que cero** y menor o igual al número de días del mes en que se encuentra, teniendo en cuenta si el año en que se encuentra es **bisiesto** o no.
 - a. Los meses *Enero, Marzo, Mayo, Julio, Agosto, Octubre, y Diciembre* tienen **31** días.
 - b. Los meses *Abril, Junio, Septiembre y Noviembre* tienen **30** días.
 - c. El mes *Febrero* tiene **29** días si el año en que se encuentra es **bisiesto**, y en otro caso tiene **28** días.
 - d. Un año es bisiesto si es divisible entre **400**, o si es divisible entre **4** y no es divisible entre **100**.

Sentencias de Selección. Anidamientos. Ejemplo 1

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int dia, mes, anyo;
    printf("Introduce día, mes y año: ");
    scanf(" %d %d %d", &dia, &mes, &anyo);

    bool ok = false;
    if (1 <= mes && mes <= 12) {
        int ndias; // su ámbito y tiempo de vida abarca el bloque { } en el que está definida
        if (mes == 2) {
            if ((anyo % 400 == 0) || ((anyo % 4 == 0) && (anyo % 100 != 0))) {
                ndias = 29;
            } else {
                ndias = 28;
            }
        } else if ((mes == 4) || (mes == 6) || (mes == 9) || (mes == 11)) {
            ndias = 30;
        } else {
            ndias = 31;
        }
        ok = (1 <= dia && dia <= ndias);
    }
    if (ok) {
        printf("Fecha Correcta\n");
    } else {
        printf("Error\n");
    }
}
```

```
if (1 <= dia && dia <= ndias) {
    ok = true;
} else {
    ok = false;
}
```

Sentencias de Selección. Anidamientos. Ejemplo 1

```
int main() {
    int dia, mes, anyo;
    printf("Introduce día, mes y año: ");
    scanf(" %d %d %d", &dia, &mes, &anyo);

    bool ok = false;
    if (1 <= mes && mes <= 12) {
        int ndias;    // su ámbito y tiempo de vida abarca el bloque { } en el que está definida
        switch (mes) {
            case 2:
                if ((anyo % 400 == 0) || ((anyo % 4 == 0) && (anyo % 100 != 0))) {
                    ndias = 29;
                } else {
                    ndias = 28;
                }
                break;
            case 4: case 6: case 9: case 11:    // compacto por falta de espacio
                ndias = 30;
                break;
            default:
                ndias = 31;
                break;
        }
        ok = (1 <= dia && dia <= ndias);
    }
    if ( ok ) {
        printf("Fecha Correcta\n");
    } else {
        printf("Error\n");
    }
}
```

El Operador Condicional Ternario (?:)

El operador condicional ternario (?:)

`(cond ? op2 : op3)`

- El operador condicional ternario `(cond ? op2 : op3)` evalúa la condición (`cond`), si es verdadera, entonces el resultado de la expresión es el valor del segundo operando (`op2`). En otro caso, el resultado es el valor del tercer operando (`op3`).
 - Se debe utilizar de forma simple, ya que puede dar lugar a expresiones complejas.

```
int mayor = (x > y) ? x : y;    // Asigna el valor mayor de X e Y
int menor = (x < y) ? x : y;    // Asigna el valor menor de X e Y
```

Sentencias de Iteración. Bucles

- **Repiten** la ejecución de una secuencia de sentencias, el **cuerpo del bucle**.
- El **número de iteraciones** depende de una *expresión lógica*, denominada **condición de control del bucle**, cuyo valor evoluciona durante la ejecución del cuerpo del bucle.
- El número de iteraciones realizadas por un bucle puede ser:
 - **Predeterminado**: el número de iteraciones a realizar se conoce con anterioridad al inicio de las iteraciones. Bucle **for**
 - **No predeterminado**: el número de iteraciones a realizar depende de la propia ejecución de las sentencias iterativas. Bucles **while** y **do-while**

- Para el diseño correcto de un bucle se deben considerar los siguientes aspectos:

1 Control de la iteración.

1 ¿Cuándo debe terminar la ejecución del bucle?

Estudiar la condición de control del bucle y cuando debe terminar la iteración.

2 Inicialización de la condición de control del bucle.

Las variables implicadas en la condición deben tener valores iniciales adecuados.

3 Evolución de la condición de control del bucle.

Las variables implicadas en la condición deben evolucionar durante el proceso iterativo, desde sus valores iniciales, hasta que finalmente tomen valores que permitan terminar la iteración al evaluar la condición de control del bucle.

2 Proceso iterativo.

1 Acción iterativa

¿Cual es el proceso cuya ejecución queremos repetir ?

2 Inicialización del proceso iterativo.

Las variables implicadas en el proceso iterativo deben tener valores iniciales adecuados.

3 Evolución del proceso iterativo.

Las variables implicadas en el proceso iterativo pueden evolucionar durante el mismo.

- A veces se entremezclan ambos aspectos, el control de la iteración y el proceso iterativo.

Sentencias de Iteración. Ejercicios

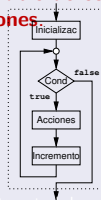
- 1 Programa que muestre los N primeros números enteros mayores o iguales a cero, siendo N un número leído desde teclado, ej. para $N=6$: 0 1 2 3 4 5
- 2 Programa que muestre los N primeros números enteros mayores o iguales a cero (en orden **inverso**), siendo N un número leído desde teclado, ej. para $N=6$: 5 4 3 2 1 0
- 3 Programa que muestre los N primeros números enteros impares mayores que cero, siendo N un número leído desde teclado, ej. para $N=6$: 1 3 5 7 9 11

Sentencias de Iteración

Sentencia de Iteración For

- Se utiliza cuando el **número de iteraciones** que se van a realizar está **predeterminado** antes de su ejecución, y no depende de la ejecución del cuerpo del bucle.
- El proceso iterativo está **controlado por los valores** que toma una **variable de control**.
- INICIALIZACIÓN**: **inicializa la variable de control**. Se ejecuta **una única vez** antes del proceso iterativo. El ámbito de vida de la variable de control abarca todo el proceso iterativo.
- EXPR_LÓGICA**: una **expresión lógica** que utiliza la **variable de control** para determinar el **número de iteraciones**. En cada iteración, se **evalúa y comprueba** antes de ejecutar el cuerpo del bucle, si es **Verdadera**, se ejecuta el cuerpo del bucle. Si es **Falsa**, termina el bucle.
- INCREMENTO**: **evoluciona la variable de control**. En cada iteración, se ejecuta **justo después** de ejecutar el cuerpo del bucle.
- El **cuerpo del bucle** se ejecuta **para cada valor** que toma la **variable de control**.
- Se recomienda que en el cuerpo del bucle, no sean modificadas ni la variable de control del bucle, ni el valor de la expresión que determina el número de iteraciones.**

```
int main()  
{  
    for ( INICIALIZACION; EXPR_LOGICA; INCREMENTO ) {  
        // Acciones (cuerpo del bucle)  
    }  
}
```



Sentencias de Iteración. Bucle For. Ejemplo 1

- Programa que muestra los N primeros números enteros mayores o iguales a cero, siendo N un número leído desde teclado, ej. para $N=6$: 0 1 2 3 4 5

```
#include <stdio.h>
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int i = 0; i < n; ++i) {
        printf("%d ", i);
    }
    printf("\n");
}
```

Sentencias de Iteración. Bucle For. Ejemplo 2 (v1)

- Programa que muestra los N primeros números enteros mayores o iguales a cero (en orden **inverso**), siendo N un número leído desde teclado, ej. para $N=6$: 5 4 3 2 1 0

```
#include <stdio.h>
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int i = 0; i < n; ++i) {
        printf("%d ", (n - i - 1));
    }
    printf("\n");
}
```

n	i	v
6	0	5
6	1	4
6	2	3
6	3	2
6	4	1
6	5	0

$$v == n - i - 1$$

Sentencias de Iteración. Bucle For. Ejemplo 2 (v2)

- Programa que muestra los N primeros números enteros mayores o iguales a cero (en orden **inverso**), siendo N un número leído desde teclado, ej. para $N=6$: 5 4 3 2 1 0

```
#include <stdio.h>
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int i = n-1; i >= 0; --i) {
        printf("%d ", i);
    }
    printf("\n");
}
```

Sentencias de Iteración. Bucle For. Ejemplo 3 (v1)

- Programa que muestra los N primeros números enteros impares mayores que cero, siendo N un número leído desde teclado, ej. para $N=6$: 1 3 5 7 9 11

```
#include <stdio.h>

int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int i = 0; i < n; ++i) {
        printf("%d ", ((i * 2) + 1));
    }
    printf("\n");
}
```

n	i	v
6	0	1
6	1	3
6	2	5
6	3	7
6	4	9
6	5	11

$$v == i*2+1$$

Sentencias de Iteración. Bucle For. Ejemplo 3 (v2)

- Programa que muestra los N primeros números enteros impares mayores que cero, siendo N un número leído desde teclado, ej. para $N=6$: 1 3 5 7 9 11

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

```
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int i = 1; i < 2 * n; i += 2) {
        printf("%d ", i);
    }
    printf("\n");
}
```

Sentencias de Iteración. Anidamientos. Ejemplos

- Desarrolle varios programas donde cada uno lee de teclado un número y muestra una figura como en los siguientes ejemplos para el número **7**.

```
*****↵
*****↵
*****↵
*****↵
*****↵
*****↵
*****↵
```

prg.1

```
*****↵
 *   *↵
  *  *↵
   * *↵
    **↵
     *↵
```

prg.2

```
      *↵
     **↵
    * *↵
   *  *↵
  *   *↵
 *    *↵
*****↵
```

prg.3

```
*↵
**↵
* *↵
*  *↵
*   *↵
*    *↵
*     *↵
*****↵
```

prg.4

```
*****↵
 *   *↵
  *  *↵
   * *↵
    **↵
     *↵
    *↵
```

prg.5

- Nótese que después del último asterisco de cada fila debe haber un salto de línea ↵ ("`\n`");

Sentencias de Iteración. Anidamientos. Ejemplo 1

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número 5.

```
#include <stdio.h>
const char SIMBOLO = '*';
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n; ++c) {
            printf("%c", SIMBOLO);
        }
        printf("\n");
    }
}
```

```
*****↵
*****↵
*****↵
*****↵
*****↵
```

- Nótese que después del último asterisco de cada fila debe haber un salto de línea ↵ ("`\n`");

Sentencias de Iteración. Anidamientos. Ejemplo 2

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número 5.

```
#include <stdio.h>
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n; ++c) {
            if ((f == 0) || (c == n-1) || (c == f)) {
                printf("%c", SIMBOLO);
            } else {
                printf("%c", ESPACIO);
            }
        }
        printf("\n");
    }
}
```

```
*****
*   *
*   *
**
*   *
```

- Nótese que después del último asterisco de cada fila debe haber un salto de línea `\n` ("\\n");

Sentencias de Iteración. Anidamientos. Ejemplo 3

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número 5.

```
#include <stdio.h>
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n; ++c) {
            if ((f == n-1) || (c == n-1) || (c == n-f-1)) {
                printf("%c", SIMBOLO);
            } else {
                printf("%c", ESPACIO);
            }
        }
        printf("\n");
    }
}
```

```
*↵
**↵
* *↵
* *↵
*****↵
```

n	f	c
5	0	4
5	1	3
5	2	2
5	3	1
5	4	0

$c == n-f-1$

- Nótese que después del último asterisco de cada fila debe haber un salto de línea ↵ ("`\n`");

Sentencias de Iteración. Anidamientos. Ejemplo 4

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número 5.

```
#include <stdio.h>
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < f+1; ++c) {
            if ((f == n-1) || (c == 0) || (c == f)) {
                printf("%c", SIMBOLO);
            } else {
                printf("%c", ESPACIO);
            }
        }
        printf("\n");
    }
}
```

```
*↵
**↵
* *↵
* *↵
*****↵
```

n	f	columnas
5	0	1
5	1	2
5	2	3
5	3	4
5	4	5

columnas == f+1

- Nótese que después del último asterisco de cada fila debe haber un salto de línea ↵ ("`\n`");

Sentencias de Iteración. Anidamientos. Ejemplo 5

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número 5.

```
#include <stdio.h>
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    printf("Introduce un número: ");
    scanf(" %d", &n);

    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n-f; ++c) {
            if ((f == 0) || (c == 0) || (c == n-f-1)) {
                printf("%c", SIMBOLO);
            } else {
                printf("%c", ESPACIO);
            }
        }
        printf("\n");
    }
}
```

```
*****
*   *
* * 
***
*   
```

n	f	columnas
5	0	5
5	1	4
5	2	3
5	3	2
5	4	1

columnas == n-f

- Nótese que después del último asterisco de cada fila debe haber un salto de línea `\n` ("\\n");

Sentencias de Iteración. Ejercicios

- Un número entero es **primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

Sentencias de Iteración

Sentencia de Iteración For con Final Anticipado

- Se utiliza cuando el **número máximo de iteraciones** que se pueden realizar está **predeterminado** antes de su ejecución, pero es posible **finalizar anticipadamente** las iteraciones, dependiendo de alguna **condición extra**.
- La expresión lógica que controla la iteración tiene dos partes, **conectadas mediante el operador lógico &&** (**NO se deben conectar con ||**):
 - La **primera parte** está destinada a controlar el **número máximo de iteraciones** que se pueden realizar, utilizando para ello la **variable de control** del bucle.
 - La **segunda parte** añade la **condición** necesaria para **finalizar anticipadamente** el proceso iterativo.
- El **cuerpo del bucle** sí podrá modificar los valores que afectan a la condición de finalización anticipada.
- **Se recomienda que en el cuerpo del bucle, no sean modificadas ni la variable de control del bucle, ni el valor de la expresión que determina el número máximo de iteraciones.**

```
int main()
{
    for (int i = 0; (i < MAX_ITER) && ( cond_final_anticipado ); ++i) {
        // Acciones (cuerpo del bucle)
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v1)

- Un número entero es **primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    int cnt = 0; // cuenta de divisores, inicialmente a cero
    for (int div = 1; div <= num; ++div) {
        if ((num % div) == 0) {
            ++cnt; // si tiene algún divisor, incrementar la cuenta de divisores
        }
    }

    if (cnt == 2) {
        // si sólo tiene dos divisores entre 1 y NUM, entonces es primo
        printf("El número %d es primo\n", num);
    } else {
        printf("El número %d no es primo\n", num);
    }
}
```


Sentencias de Iteración. Ejemplo 1 (v2)

- Un número entero es **primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    int cnt = 0;
    for (int div = 2; div <= num/2; ++div) {
        if ((num % div) == 0) {
            ++cnt;
        }
    }

    if ((num > 1)&&(cnt == 0)) {
        // si (NUM>1) y no tiene ningún divisor entre 2 y NUM/2, entonces es primo
        printf("El número %d es primo\n", num);
    } else {
        printf("El número %d no es primo\n", num);
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v3)

- Un número entero es **primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    int cnt = 0;
    for (int div = 2; (div <= num/2) && (cnt == 0); ++div) { // final anticipado
        if ((num % div) == 0) {
            ++cnt;
        }
    }

    if ((num > 1) && (cnt == 0)) {
        // si (NUM>1) y no tiene ningún divisor entre 2 y NUM/2, entonces es primo
        printf("El número %d es primo\n", num);
    } else {
        printf("El número %d no es primo\n", num);
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v4)

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    bool es_primo = (num > 1);
    for (int div = 2; (div <= num/2) && es_primo ; ++div) { // final anticipado
        if ((num % div) == 0) {
            es_primo = false;
        }
    }

    if (es_primo) {
        printf("El número %d es primo\n", num);
    } else {
        printf("El número %d no es primo\n", num);
    }
}
```

Sentencias de Iteración. Bucles. Ejercicios

- El *máximo común divisor* de dos números enteros es el mayor número entero que es divisor de ambos números.
- Se puede calcular el *máximo común divisor* de dos números enteros positivos mediante el *algoritmo de Euclides*:
 - Dados dos números enteros positivos, si son distintos, entonces se realiza un proceso iterativo hasta que los dos números sean iguales, en el cual, en cada iteración, al número mayor se le restará el número menor.
 - Al finalizar el proceso iterativo, el *máximo común divisor* será igual al valor de ambos números (*que ahora son iguales*).
- Calcular el **máximo común divisor**, según el método de Euclides, de dos números enteros (> 0) leídos de teclado.

	M	N	
	24	18	
24-18 \Rightarrow	6	18	
	6	12	$\Leftarrow 18-6$
	6	6	$\Leftarrow 12-6$

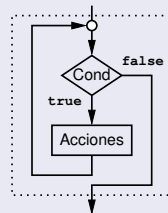
Sentencias de Iteración While y Do-While

- Las sentencias **WHILE** y **DO-WHILE** se utilizan cuando el número de iteraciones que se van a realizar **no está predeterminado** antes de la ejecución del proceso iterativo, sino que depende de la ejecución de este proceso iterativo.
 - Una condición (expresión lógica) controla la iteración, si es **verdadera**, entonces continúa el proceso iterativo.
 - En el cuerpo del bucle habrá sentencias que dirijan la **evolución** de la iteración afectando al valor de la condición de control.

Sentencia de Iteración While

- La condición de control se evalúa **antes** de ejecutar el cuerpo del bucle.
 - Si la condición se evalúa a **Verdadera**, entonces **se ejecuta** el cuerpo y se **repite** el proceso.
 - Si la condición se evalúa a **Falsa**, entonces **NO se ejecuta** el cuerpo y **finaliza** la ejecución del bucle.

```
int main()
{
    while ( Expr_Logica ) {
        // Acciones (cuerpo del bucle)
    }
}
```



Sentencias de Iteración. Ejemplo 1 (v1)

- Calcular el *máximo común divisor*, según el método de Euclides, de dos números enteros (> 0) leídos de teclado.

Para ello, se le debe restar **sucesivamente** al número **mayor** el número **menor**, hasta que los dos números sean iguales. El valor final es el **MCD**.

```
#include <stdio.h>
int main()
{
    int num1, num2;
    printf("Introduce dos números positivos: ");
    scanf(" %d %d", &num1, &num2);

    if ((num1 <= 0) || (num2 <= 0)) {
        printf("Error. Números erróneos\n");
    } else {
        while (num1 != num2) {
            if (num1 > num2) {
                num1 = num1 - num2;
            } else {
                num2 = num2 - num1;
            }
        }

        printf("El MCD es: %d\n", num1);
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v2)

- Calcular el *máximo común divisor*, según el método de Euclides, de dos números enteros (> 0) leídos de teclado.

Para ello, se le debe restar **sucesivamente** al número **mayor** el número **menor**, hasta que los dos números sean iguales. El valor final es el **MCD**.

```
#include <stdio.h>
int main()
{
    int num1, num2;
    printf("Introduce dos números positivos: ");
    scanf(" %d %d", &num1, &num2);
    while ((num1 <= 0) || (num2 <= 0)) {
        printf("Error. Introduce dos números positivos: ");
        scanf(" %d %d", &num1, &num2);
    }

    while (num1 != num2) {
        if (num1 > num2) {
            num1 = num1 - num2;
        } else {
            num2 = num2 - num1;
        }
    }

    printf("El MCD es: %d\n", num1);
}
```


Sentencias de Iteración. Ejemplo 2 (v1)

- Un número entero es **primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    bool es_primo = (num > 1);
    int div = 2;
    while ((div <= num/2) && es_primo) {
        if ((num % div) == 0) {
            es_primo = false;
        }
        ++div;
    }

    if (es_primo) {
        printf("El número %d es primo\n", num);
    } else {
        printf("El número %d no es primo\n", num);
    }
}
```

Sentencias de Iteración. Ejemplo 2 (v2)

- Un número entero es **primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce un número: ");
    scanf(" %d", &num);

    int div = 2;
    while ((div <= num/2)&&((num % div) != 0)) { // si encuentro un divisor
        ++div;                                // (NUM%DIV==0), entonces
    }                                         // terminar el bucle

    if (div == num/2+1) {
        // si (DIV == NUM/2+1), entonces no se ha encontrado ningún
        // divisor entre 2 y NUM/2, por lo tanto es primo
        printf("El número %d es primo\n", num);
    } else {
        printf("El número %d no es primo\n", num);
    }
}
```

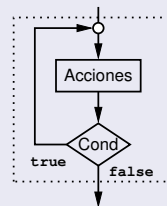
- Programa que lea dos números enteros (numerador y denominador). En la lectura del denominador, si el número leído es igual a cero, entonces se repetirá el proceso de lectura del denominador hasta que el número introducido sea distinto de cero. Finalmente se mostrará el resultado de dividir el numerador entre el denominador.

Sentencias de Iteración. Bucles

Sentencia de Iteración Do-While

- La condición de control se evalúa **después** de ejecutar el cuerpo del bucle.
 - Primero **se ejecuta** el cuerpo, y posteriormente, si la condición se evalúa a **Verdadera**, entonces se **repite** el proceso.
 - Si la condición se evalúa a **Falsa**, entonces **finaliza** la ejecución del bucle.

```
int main()  
{  
    do {  
        // Acciones (cuerpo del bucle)  
    } while ( EXPR_LOGICA ) ; // Nótese el punto-y-coma  
}
```



Sentencias de Iteración. Ejemplo 1

- Programa que lea dos números enteros (numerador y denominador). En la lectura del denominador, si el número leído es igual a cero, entonces se repetirá el proceso de lectura del denominador hasta que el número introducido sea distinto de cero. Finalmente se mostrará el resultado de dividir el numerador entre el denominador.

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

```
int main()
{
    int numerador, denominador;
    printf("Introduce el numerador: ");
    scanf(" %d", &numerador);
    do {
        printf("Introduce el denominador: ");
        scanf(" %d", &denominador);
    } while (denominador == 0);

    int resultado = numerador / denominador;

    printf("Resultado: %d\n", resultado);
}
```

Definición de Variables

- La **variables globales** se definen **fuera** de los subprogramas.
 - La **definición y/o utilización de variables globales está prohibida**.
- La **variables locales** se definen **dentro** de los subprogramas.
- Cuando se define una **variable sin un valor inicial**, entonces la variable almacena un valor **inespecificado (basura)**.
 - **Utilizar un valor inespecificado de una variable es un ERROR en nuestro programa**.
 - Cuando sea posible, se recomienda asignar un valor inicial en la definición de la variable.
- El **ámbito** y tiempo de vida de una variable local abarca la zona comprendida entre su definición, y el final del bloque donde ha sido definida.
- Una variable se debe definir con el **menor ámbito** (en el bloque anidado más interno) posible que permita resolver el problema.
 - Aumenta la localidad del código y disminuye las dependencias externas.
 - Si una variable no está en ámbito, entonces no puede ser mal-utilizada, ni corrompida.
 - Simplifica el análisis y la modificación del programa, así como la corrección de errores.
- Cuando se define una **variable de control de un bucle for**, entonces la variable se destruye cuando se termina de ejecutar todas las iteraciones del bucle.
- Cuando se define una variable **dentro de un bucle**, entonces la variable se **crea** y se

Sentencias de Iteración. Ejercicios

- ❶ Multiplicar, mediante sumas, dos números enteros (≥ 0) leídos de teclado.

$$2 \times 5 \equiv 2 + 2 + 2 + 2 + 2$$

- ❷ Potencia de la base elevada al exponente, dos números enteros (≥ 0) leídos de teclado (error si 0^0). En **C** x^y **no** es equivalente a x *elevado a y*.

$$2^5 \equiv 2 \times 2 \times 2 \times 2 \times 2$$

- ❸ Calcular el factorial de un número (≥ 0) leído de teclado.

$$6! \equiv 1 \times 2 \times 3 \times 4 \times 5 \times 6$$

- ❹ Dividir, mediante restas, dos números enteros (≥ 0) leídos de teclado.

$$\begin{array}{lcl} 23 \div 4 \Rightarrow & 23 - 4 \equiv 19 & \\ & 19 - 4 \equiv 15 & \\ & 15 - 4 \equiv 11 & \\ & 11 - 4 \equiv 7 & \\ & 7 - 4 \equiv 3 & \Rightarrow \text{resto} \equiv 3; \text{cociente} \equiv 5 \end{array}$$

Sentencias de Iteración. Ejercicio 1

- Multiplicar, mediante sumas, dos números enteros (≥ 0) leídos de teclado.

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

```
int main()
```

```
{
```

```
    int m, n;
```

```
    printf("Introduce dos números: ");
```

```
    scanf(" %d %d", &m, &n);
```

```
    if ((m < 0) || (n < 0)) {
```

```
        printf("Error, números negativos\n");
```

```
    } else {
```

```
        if (n > m) {           // la multiplicación es conmutativa
```

```
            int aux = n;      // N debe contener el número MENOR
```

```
            n = m;
```

```
            m = aux;
```

```
        }
```

```
        // Sumar: m + m + m + ... + m    (n veces)
```

```
        int total = 0; // 0 es el elemento neutro de la suma
```

```
        for (int i = 0; i < n; ++i) {
```

```
            // Proceso iterativo: Añadir el valor de 'm' al total
```

```
            total = total + m;      // total += m;
```

```
        }
```

```
        printf("Resultado: %d\n", total);
```

```
    }
```

```
}
```


Sentencias de Iteración. Ejercicio 2

- Potencia de la base elevada al exponente, dos números enteros (≥ 0) leídos de teclado (error si 0^0). En C x^y no es equivalente a x elevado a y .

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

```
int main()
{
    int base, exp;
    printf("Introduce base y exponente: ");
    scanf(" %d %d", &base, &exp);

    if ((base < 0) || (exp < 0) || (base == 0 && exp == 0)) {
        printf("Error, números no válidos\n");
    } else {
        // Multiplicar: base * base * base * ... * base (exp veces)
        int total = 1; // 1 es el elemento neutro de la multiplicación
        for (int i = 0; i < exp; ++i) {
            // Proceso iterativo: Multiplicar el valor de 'base' al total
            total = total * base; // total *= base;
        }
        printf("Resultado: %d\n", total);
    }
}
```

Sentencias de Iteración. Ejercicio 3

- Calcular el factorial de un número (≥ 0) leído de teclado.

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

```
int main()
```

```
{
```

```
    int n;
```

```
    printf("Introduce un número: ");
```

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

```
    if (n < 0) {
```

```
        printf("Error, número negativo\n");
```

```
    } else {
```

```
        // Multiplicar: 1 2 3 4 5 6 7 ... n
```

```
        long total = 1; // el tipo LONG permite almacenar números mayores
```

```
        for (int i = 2; i <= n; ++i) {
```

```
            // Proceso iterativo: multiplicar numero y acumular
```

```
            total = total * i; // total *= i;
```

```
        }
```

```
        printf("Factorial: %ld\n", total);
```

```
    }
```

```
}
```

Sentencias de Iteración. Ejercicio 4

- Dividir, mediante restas, dos números enteros (≥ 0) leídos de teclado.

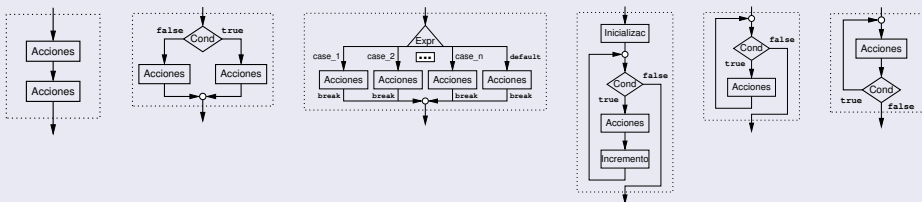
```
#include <stdio.h>

int main()
{
    int dividendo, divisor;
    printf("Introduce el dividendo y el divisor: ");
    scanf(" %d %d", &dividendo, &divisor);

    if ((dividendo < 0) || (divisor <= 0)) {
        printf("Error, números erróneos\n");
    } else {
        int resto = dividendo;
        int cociente = 0;
        while (resto >= divisor) {
            resto = resto - divisor;    // resto -= divisor;
            ++cociente;
        }
        printf("Cociente: %d\n", cociente);
        printf("Resto: %d\n", resto);
    }
}
```

Programación Estructurada

- Un programa sigue una metodología de **programación estructurada** si todas las estructuras de control que se utilizan (secuencia, selección e iteración) tienen un **único punto de entrada** y un **único punto de salida**.



- Bohm y Jacopini demostraron que todo problema computable puede resolverse usando únicamente estas estructuras. Es la base de la **programación estructurada**. Un algoritmo que use tan sólo las estructuras de control tratadas en este tema, se denomina estructurado.
- Se recomienda utilizar las sentencias y estructuras de selección e iteración (if, switch, while, do-while y for).
- Se recomienda no utilizar las sentencias goto ni continue. Se recomienda que la sentencia break sólo se utilice en la estructura switch, y no se utilice para terminar la ejecución de un bucle.

Control de Errores.

- Durante la ejecución de un programa, se pueden dar **situaciones inesperadas o errores inesperados** para los cuales el programa no tiene un comportamiento adecuado. Puede suceder:
 - Que el programa **aborte** su ejecución, por ejemplo al realizar una división entre cero.
 - Que el programa muestre un **malfuncionamiento**, por ejemplo que produzca un resultado diferente al correcto.
- El programador puede utilizar las estructuras de selección y de iteración para controlar algunos de estos errores.

```
#include <stdio.h>
int main()
{
    int numerador, denominador, resultado;
    printf("Introduce el numerador y denominador: ");
    scanf(" %d %d", &numerador, &denominador);
    if (denominador == 0) {
        printf("Error: división entre cero\n");
    } else {
        resultado = numerador / denominador;
        printf("Resultado: %d\n", resultado);
    }
}
```

Tipo puntero, dirección de memoria, y desreferenciación (I)

- El valor de las **variables** de un programa se almacenan en la **memoria** del ordenador, en posiciones que son direccionables y accesibles.
- El operador **ampersand** (&) permite conocer la dirección de memoria donde se almacena el valor de una variable.
 - Si la variable es de un determinado tipo, por ejemplo **int**, entonces la dirección de esa variable es de tipo **puntero a int** (**int***), y se puede almacenar en otra variable de ese mismo tipo.
- El operador ***** permite acceder (**desreferenciar**) a la zona de memoria referenciada por un puntero, tanto para conocer su valor, como para cambiarlo.

```
int main()
{
    int numero = 3;

    int* puntero_a_numero = &numero; // Dirección de la variable (puntero a la variable)

    int valor = *puntero_a_numero;    // Desreferenciar el puntero (acceso al valor)

    *puntero_a_numero = 7;           // Desreferenciar el puntero (modificar el valor)
}
```

numero: 120:	3 7	Variable: numero , tipo: int , dirección de memoria: 120, valor-anterior: 3, valor-nuevo: 7
puntnum: 124:	120	Variable: puntnum , tipo: int* , dirección de memoria: 124, valor: 120
valor: 128:	3	Variable: valor , tipo: int , dirección de memoria: 128, valor: 3

Tipo puntero, dirección de memoria, y desreferenciación (II)

```
int main()
{
    int numero = 3;
    int* puntero_a_numero = &numero ;
    int valor = * puntero_a_numero ;
    * puntero_a_numero = 7;
}
```



Tipo puntero, dirección de memoria, y desreferenciación (III)

```
#include <stdio.h>

int main()
{
    int numero = 3;
    printf("Numero inicial: %d\n", numero);           // muestra 3

    printf("Introduce un número: ");
    scanf(" %d", &numero);                           // se introduce 5
    printf("Numero: %d\n", numero);                   // muestra 5

    int* puntero_a_numero = &numero; // Dirección de la variable (puntero a la variable)

    printf("Dirección: %p\n", puntero_a_numero); // muestra 0x7ffdac6c156c (otra dirección es posible)
    printf("Dirección: %p\n", &numero);          // muestra 0x7ffdac6c156c (otra dirección es posible)

    int valor = *puntero_a_numero; // Desreferenciar el puntero (acceso al valor)

    printf("Numero: %d\n", numero); // muestra 5
    printf("Valor: %d\n", valor);    // muestra 5
    printf("Desref: %d\n", *puntero_a_numero); // muestra 5

    *puntero_a_numero = 7; // Desreferenciar el puntero (modificar el valor)

    printf("Numero: %d\n", numero); // muestra 7
    printf("Valor: %d\n", valor);    // muestra 7
    printf("Desref: %d\n", *puntero_a_numero); // muestra 7
}
```

Diagrama de memoria:

numero:	120:	3	Variable: numero , tipo: int , dirección de memoria: 120 , valor-anterior: 3 , valor-anterior: 5 , valor-nuevo: 7
puntnum:	124:	120	Variable: puntnum , tipo: int* , dirección de memoria: 124 , valor: 120
valor:	128:	5	Variable: valor , tipo: int , dirección de memoria: 128 , valor: 5

Diagrama de puntero:

numero:	120:	3
puntnum:	124:	•
valor:	128:	5

Una flecha apunta desde el punto en la dirección de memoria 124 (puntero) hacia la dirección de memoria 128 (valor).

Tipos Enumerados (I)

- Los **tipos simples predefinidos son suficientes** para resolver problemas que sólo requieran tipos de datos simples.
- Sin embargo, también es posible definir otros tipos simples para especificar la interpretación de un valor, o para especificar los valores posibles.

- El programador puede definir un **tipo enumerado** como una secuencia de símbolos separados por comas. Estos símbolos son los “valores” asociados al tipo definido.

```
enum Colores { ROJO, VERDE, AZUL } ;           // punto-y-coma  
enum Palos { OROS, COPAS, ESPADAS, BASTOS } ; // punto-y-coma
```

- Se pueden **declarar** variables y constantes de un tipo enumerado definido. Una variable del tipo enumerado puede **tomar el valor** de cualquier símbolo de la enumeración.

```
enum Colores mi_color = VERDE ;
```

- El tipo enumerado es un **tipo simple escalar**, se pueden aplicar los operadores relacionales a sus valores.
- Un valor del tipo enumerado se puede **convertir** al tipo entero. A cada elemento de la enumeración le corresponde un número según la posición que ocupa en la secuencia:
 $0 \equiv (\text{int})\text{ROJO}$ $2 \equiv (\text{int})\text{ESPADAS}$ $\text{VERDE} \equiv (\text{enum Colores})1$
- **No** es posible realizar **entrada** (scanf) directa de valores enumerados. La **salida** (printf) se realiza como números enteros (int).

Tipos Enumerados (II)

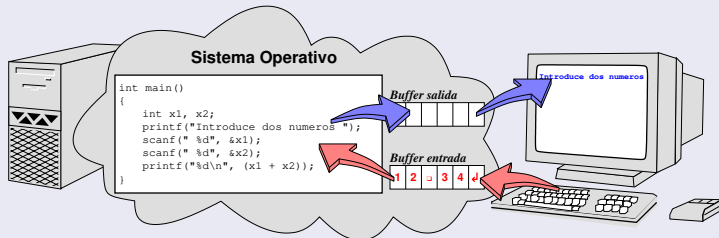
- Los tipos enumerados también se pueden utilizar para definir **símbolos constantes** con valores enteros (de tipo `int`).

```
enum {  
    MAXIMO = 20,  
    LIMITE = 10,  
} ;                                // punto-y-coma  
int main()  
{  
    int valor = LIMITE;  
    if (valor < MAXIMO) {  
        printf("%d %d %d\n", valor, LIMITE, MAXIMO);  
    }  
}
```

Entrada de Datos Avanzada

El Buffer de Entrada

- La entrada y salida de datos se realiza indirectamente a través de **buffers** controlados por el *Sistema Operativo* y son independientes de nuestro programa.
- Cuando se pulsa alguna tecla, los caracteres correspondientes a las teclas pulsadas se almacenan en una zona de memoria intermedia: el **buffer de entrada**. Cuando un programa realiza una operación de entrada de datos, accede al buffer de entrada y obtiene los caracteres allí almacenados si los hubiera, o esperará hasta que los haya (cuando se pulsen una serie de teclas seguida por la tecla **ENTER**).
- Una vez obtenidos los caracteres asociados a las teclas pulsadas, se **convertirán** a un valor del tipo de la variable especificada por la operación de entrada, **asignándole** dicho valor a la misma.




Entrada de Datos Avanzada

- La entrada de datos con `scanf`, excepto para el especificador `%c` (tipo `char`) se realiza **eliminado** los espacios en **blanco** y saltos de línea **iniciales** hasta encontrar algún carácter distinto de ellos (datos).
- Después se **leen** caracteres hasta encontrar algún carácter no adecuado para el tipo de datos de la lectura.
- Si el dato leído se puede convertir al tipo de la variable que lo almacenará, entonces la operación será correcta, los datos leídos se **convierten** al tipo de la variable y se **asignan** a la variable. En otro caso, la operación será incorrecta, el flujo de entrada pasa a modo **erróneo**, y cualquier entrada siguiente **fallará**.

```
int main() {  
    int x; char letra1, letra2;  
    printf("Introduce número seguido por letra espacio letra: ");  
    scanf("%d %c %c", &x, &letra1, &letra2);  
    printf("Resultado: %d [%c] [%c]\n", x, letra1, letra2);  
}
```

Ejecución del Programa

Introduce número seguido: `123z` 
Resultado: 123 [z] [x]

3 espacios número 123 letra z
espacio letra x ENTER

Entrada de Caracteres Avanzada


- La entrada de datos con `scanf`, para el especificador `%c` (tipo `char`), **lee** el siguiente carácter (de tipo `char`) **sin eliminar** los espacios en blanco ni saltos de línea **iniciales**.

```
char letra;  
scanf("%c", &letra);
```

- Sin embargo, si se especifica un **espacio en blanco** antes del especificador `%c`, entonces se realiza **eliminación** los espacios en **blanco** y saltos de línea **iniciales** hasta encontrar algún carácter distinto de ellos, y después **lee** el siguiente carácter (de tipo `char`), y se **asigna** a la variable.

```
int main() {  
    char c1, c2, c3 ;  
    printf("Introduce tres caracteres: ");  
    scanf(" %c%c%c", &c1, &c2, &c3); // nótese el espacio antes del primer %c  
    printf("Resultado: [%c] [%c] [%c]\n", c1, c2, c3);  
}
```

Ejecución del Programa

Introduzca 3 caracteres: `a b` 
Resultado: `[a] [] [b]`

Se introducen 3 espacios letra a espacio letra b ENTER
Se han leído los tres caracteres consecutivos

Entrada de Datos Avanzada. Procesamiento Secuencial (I)

Ejemplos. Procesamiento Secuencial

- ❶ Programa que lea de teclado una secuencia de números enteros terminada en el número cero, y muestra el resultado de la suma de todos los números de la secuencia (sin incluir al cero terminador), por ejemplo, para la secuencia de números **20 34 12 54 0** muestra el resultado **120**.

Introduce una secuencia de números terminada en cero:


20 34 12 54 0 

| números separados por espacios y ENTER

Resultado: 120

- ❷ Programa que lea de teclado una secuencia de números enteros terminada en el número cero, y muestra aquellos números de la secuencia de entrada que son pares (sin incluir al cero terminador), por ejemplo, para la secuencia de números **11 12 13 14 15 16 17 18 19 0** muestra el resultado **12 14 16 18**.

Introduce una secuencia de números terminada en cero:

11 12 13 14 15 16 17 18 19 0 

| números separados por espacios y ENTER

Resultado: 12 14 16 18

- ❸ Programa que lee de teclado una secuencia de números enteros terminada en el número cero, y muestra el mayor, menor y la media de todos los números de la secuencia (sin incluir al cero terminador), por ejemplo, para la secuencia de números **20 34 12 54 17 0** muestra el resultado **54 12 27.4**.

Introduce una secuencia de números terminada en cero:

20 34 12 54 17 0 

| números separados por espacios y ENTER

Resultado: 54 12 27.4

Entrada de Datos Avanzada. Procesamiento Secuencial (II)

El Buffer de Entrada. Procesamiento Secuencial. Ejemplo 1

```
#include <stdio.h>
int main()
{
    int num;
    int suma = 0;
    printf("Introduce una secuencia de números terminada en cero: \n");
    scanf(" %d", &num);
    while (num != 0) {
        suma = suma + num; // suma += num;
        scanf(" %d", &num);
    }
    printf("Resultado: %d\n" << suma);
}
```

Ejecución del Programa

Introduce una secuencia de números terminada en cero:

20 34 12 54 0 

Resultado: 120

*cinco números separados por espacios y ENTER
los caracteres tecleados se almacenan en el Buffer
los números serán leídos desde el Buffer*


Entrada de Datos Avanzada. Procesamiento Secuencial (III)

El Buffer de Entrada. Procesamiento Secuencial. Ejemplo 2

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduce una secuencia de números terminada en cero: \n");
    scanf(" %d", &num);
    printf("Resultado: ");
    while (num != 0) {
        if (num % 2 == 0) {
            printf("%d ", num);
        }
        scanf(" %d", &num);
    }
    printf("\n");
}
```

Ejecución del Programa

Introduce una secuencia de números terminada en cero:

11 12 13 14 15 16 17 18 19 0 

Resultado: 12 14 16 18

*diez números separados por espacios y ENTER
los caracteres tecleados se almacenan en el Buffer
los números serán leídos desde el Buffer*

Entrada de Datos Avanzada. Procesamiento Secuencial (IV)

El Buffer de Entrada. Procesamiento Secuencial. Ejemplo 3

```
#include <stdio.h>
int main()
{
    int num;
    int cnt = 0;
    int suma = 0;
    printf("Introduce una secuencia de números terminada en cero: \n");
    scanf(" %d", &num);
    int menor = num;
    int mayor = num;
    while (num != 0) {
        ++cnt;
        suma += num;
        if (num > mayor) {
            mayor = num;
        } else if (num < menor) {
            menor = num;
        }
        scanf(" %d", &num);
    }
    if (cnt == 0) {
        printf("Secuencia vacía\n");
    } else {
        double media = double(suma) / double(cnt);
        printf("Resultado: %d %d %d\n", mayor, menor, media);
    }
}
```

Salida Formateada de Datos (I)

- Representación de los *números enteros* (*decimal, hexadecimal, octal*).

```
#include <stdio.h>
int main()
{
    printf(" %d\n", 27);           // muestra 27 (decimal)
    printf(" %x\n", 27);           // muestra 1b (hexadecimal)
    printf(" %o\n", 27);           // muestra 33 (octal)
}
```

Salida Formateada de Datos (II)

- Es posible especificar la *anchura de campo* y la *precisión*.

```
#include <stdio.h>
int main()
{
    // Anchura de campo 5
    printf("%5d\n", 234);           // muestra  234
    printf("%-5d\n", 234);          // muestra 234
    printf("%05d\n", 234);          // muestra 00234

    // Por defecto precisión 6
    printf("%lg %lg %lg %lg\n", 0.00006789, 0.0006789, 678900.0, 6789000.0);
    //          6.789e-05  0.0006789  678900  6.789e+06

    // Salida con precisión 3
    printf("%.3lg %.3lg %.3lg %.3lg\n", 0.00006789, 0.0006789, 678900.0, 6789000.0);
    //          6.79e-05  0.000679  6.79e+05  6.79e+06

    printf("[%9.3lg] [%9.3lg] [%9.3lg] [%9.3lg]\n", 0.00006789, 0.0006789, 678900.0, 6789000.0);
    //          [ 6.79e-05] [ 0.000679] [ 6.79e+05] [ 6.79e+06]
}
```

- Notación general: `printf("%lg", valor)`.
 - La precisión especifica el número de cifras significativas. La precisión por defecto es 6.
 - Si $(10^{-5} < |\text{valor}| < 10^{\text{precision}})$ entonces usa *punto-fijo* (`%lf`), en otro caso usa *notación científica* en *punto flotante* (`%le`).

Salida Formateada de Datos (III)

- Representación de los *números reales* (*precisión, punto fijo, notación científica*).

```
#include <stdio.h>
int main()
{
    // Notación general: (%g) Por defecto precisión 6
    printf("%lg %lg %lg %lg\n", 0.00006789, 0.0006789, 678900.0, 6789000.0);
    //          6.789e-05  0.0006789  678900  6.789e+06

    // Notación en punto fijo: (%f) Por defecto precisión 6
    printf("%lf %lf %lf %lf\n", 0.00006789, 0.0006789, 678900.0, 6789000.0);
    //          0.000068  0.000679  678900.000000  6789000.000000

    // Notación científica en punto flotante: (%e) Por defecto precisión 6
    printf("%le %le %le %le\n", 0.00006789, 0.0006789, 678900.0, 6789000.0);
    //          6.789000e-05  6.789000e-04  6.789000e+05  6.789000e+06
}
```

- La precisión por defecto es 6.
- Notación general: `printf("%lg", valor)`.
 - Si $(10^{-5} < |valor| < 10^{precision})$ entonces usa *punto-fijo* (`%lf`), en otro caso usa *punto-flotante* (`%le`).
 - La precisión especifica el número de cifras significativas.
- Notación en punto fijo: `printf("%lf", valor)`.
 - Representa los números en *punto-fijo* (sin factor de escala).
 - La precisión especifica el número de cifras decimales exactas.
- Notación científica en punto flotante: `printf("%le", valor)`.
 - Representa los números en *notación científica en punto-flotante* (con factor de escala).
 - La precisión especifica el número de cifras decimales exactas.

Desbordamiento

- El número de bits utilizados para representar los números *naturales*, *enteros* y *reales* es limitado.
- El resultado de aplicar algún operador a dos valores de un determinado tipo puede dar lugar a un valor no representable (fuera de límites) dando lugar a lo que se conoce como **desbordamiento** (**overflow**).
- Por ejemplo, la siguiente operación matemática produce un resultado erróneo.

```
#include <stdio.h>
int main()
{
    unsigned resultado = 7 + 4294967290;
    printf("%u\n", resultado);    // Muestra 1 debería mostrar 4294967297
}
```

- Ejemplo, si la representación de números naturales fuese de 4 bits:

$$\begin{array}{r} 7 \\ + 10 \\ \hline 17 \end{array} \qquad \begin{array}{r} 0111 \\ + 1010 \\ \hline 1\ 0001 \end{array}$$

X

► El resultado final en realidad es 1 (y no 17)

Problemas Derivados de la Implementación de los Números

Representación aproximada de números reales y pérdida de precisión

- El número de bits utilizado para representar la parte fraccionaria de los números reales es limitado.
 - Implica una **representación aproximada** y una **pérdida de precisión**.
- Esta pérdida de precisión afecta a los cálculos realizados con números reales.
- Debido a la pérdida de precisión en las operaciones con números reales, nunca debemos realizar comparaciones de **igualdad** (`==`/`!=`) con el resultado de éstas.
- Por ejemplo, la siguiente operación matemática produce un resultado erróneo.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    bool ok = ((3.0 * 0.1) / 3.0) == (3.0 * (0.1 / 3.0));
    printf("%d\n", ok);           // Muestra 0 (false)
}                                // Matemáticamente debería ser 1 (true)
```

- Ejemplo, si la precisión fuese de 4 dígitos:

- $3.0000 \times 0.1000 = 0.3000 \rightarrow 0.3000 / 3.0000 = 0.1000$
- $0.1000 / 3.0000 = 0.0333\text{X} \rightarrow 3.0000 \times 0.0333\text{X} = 0.0999$

Problemas Derivados de la Implementación de los Números

El tipo unsigned

- La representación de los números de tipo **unsigned** se realiza en **binario puro**, y la representación de los números de tipo **int** se realiza en **complemento-a-dos**:
- Cuando se realizan operaciones aritméticas o comparaciones entre una variable de tipo **int** y otra de tipo **unsigned**, el valor de la variable de tipo **int** se convierte a **unsigned**, que en caso de ser un valor negativo, no puede ser representado adecuadamente, por lo que puede producir resultados erróneos.
- Por ello, el tipo **unsigned** sólo se debe utilizar en aquellas situaciones donde se desee realizar aritmética modular o manipulaciones de bits.

- Ejemplo de representación de números de 3 bits:

Binario	unsigned	int
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

- Las siguientes operaciones producen resultados erróneos.

```
#include <stdio.h>
int main()
{
    int num21 = -4;
    unsigned num22 = 2;
    double num23 = num21 + num22;           // ERROR: 4.29497e+09 vs. -2
    printf("%lg\n", num23);                 // ERROR: 4.29497e+09 vs. -2
    int num24 = 1.0 * (num21 + num22);      // ERROR: 2147483647 vs. -2
    printf("%d\n", num24);                  // ERROR: 2147483647 vs. -2
    if (num21 > num22) {                     // ERROR: ¿ (-4 > 2) ? vs. false
        printf("ERROR\n");                 // ERROR: (-4 < 2) vs. false
    }
}
```

Problemas Derivados de la Implementación de los Números

El tipo float

- El tipo **float** no puede representar a todos los valores del tipo **int**.
 - El tipo **float** sólo utiliza 32 bits para representar los números en formato de punto flotante, de los cuales, se utilizan 8 bits para el exponente, y 24 bits para la mantisa.
 - El tipo **int** utiliza también 32 bits en formato **complemento-a-dos** para representar los números enteros.
 - Por lo tanto, los 24 bits de la mantisa del tipo **float** no son suficientes para representar a todos los números enteros de tipo **int**, ya que son necesarios 32 bits, por lo que se produce una pérdida de precisión en la conversión entre números de tipo **int** y números de tipo **float**. El número 16777215 es el mayor número **int** que puede representar el tipo **float** sin pérdida de precisión.
- En el siguiente ejemplo, el resultado de la multiplicación de los dos números **int** (9527×15937) es 151831799, que cuando se convierte a **float**, se produce una pérdida de precisión (pasa a tomar el valor 151831792).

```
#include <stdio.h>
int main()
{
    int num21 = 9527;           // 9527 * 15937 = 151831799
    int num22 = 15937;
    float res2a = num21 * num22; // ERROR: 151831792
    printf("%f\n", res2a);       // ERROR: 151831792 vs. 151831799
    float num23 = 1.0;
    int res2i = num21 * num22 * num23; // ERROR: 151831792
    printf("%d\n", res2i);       // ERROR: 151831792 vs. 151831799
    double res2d = num21 * num22 * num23; // ERROR: 151831792
    printf("%lf\n", res2d);      // ERROR: 151831792 vs. 151831799
}
```


Interpretación del Valor Binario Almacenado en Memoria

- Internamente toda la información se representa en **binario** como una secuencia de bits (de un tamaño especificado).
- El **tipo**, asociado a esa representación, permite interpretarlo y asociarlo con la información abstracta que representa.
- ¿ Que valor contiene la variable **v** ?

v:	...
	01000001
	...

Si **v** es de tipo entero, entonces **v** almacena el número **65**

Si **v** es de tipo carácter, entonces **v** almacena la letra '**A**' ya que el código binario 01000001 (65) es el valor de la letra '**A**' en la tabla ASCII para el tipo carácter

Tildes en el Sistema Operativo Windows

- Tanto el IDE *Code::Blocks*, como *VsCode*, en el sistema operativo *Windows*, suelen dar problemas en el tratamiento y manipulación de caracteres con tildes. Para resolver este problema, se puede incluir la biblioteca `<windows.h>` e invocar a los procedimientos que se indican en el siguiente ejemplo:

// CODE::BLOCKS

```
#include <stdio.h>
#include <windows.h>
/* resto del programa */
int main()
{
    SetConsoleOutputCP(1252);
    SetConsoleCP(1252);
    /* resto del programa */
}
```

// VSCODE

```
#include <stdio.h>
#include <windows.h>
/* resto del programa */
int main()
{
    SetConsoleOutputCP(65001);
    SetConsoleCP(65001);
    /* resto del programa */
}
```

Errores de Codificación Frecuentes

- 1 Usar el operador de **asignación** = en lugar del operador de **igualdad** ==, cuando se desea realizar una comparación de igualdad.
- 2 Usar el operador de **asignación** con **más** += en lugar del operador de **incremento** ++, cuando se desea incrementar el valor de la variable.
- 3 Usar el operador de **asignación** con **menos** -= en lugar del operador de **decremento** --, cuando se desea decrementar el valor de la variable.
- 4 Usar el símbolo & en lugar de &&. Usar el símbolo | en lugar de ||.
- 5 Usar el símbolo ^ para calcular la potencia.
- 6 Definir una operación relacional erróneamente, por ejemplo (0 <= x <= 9) en vez de ((0 <= x) && (x <= 9)).
- 7 Errores al crear un expresión lógica sin tener en cuenta que se **evalúa en cortocircuito**. Por ejemplo, codificar ((num1 % num2) == 0) && (num2 != 0)) en lugar de ((num2 != 0) && ((num1 % num2) == 0))
- 8 Es fácil equivocarse cuando se mezclan los operadores lógicos (! && ||) en la misma expresión lógica compleja.
- 9 Olvidar la sentencia **break** al final de las sentencias **case** del **switch**.
- 10 Es un error poner ; en la sentencia **if**, **else**, **for** o **while**:
if (...) ; {...} else ; {...} en lugar de if (...) {...} else {...}
for (...) ; {...} en lugar de for (...) {...}
while (...) ; {...} en lugar de while (...) {...}