Índice general

Tema 0. Presentación

José Enrique Cano Ocaña, Teoría grupo A y prácticas A1 (Jueves mañana) y A2 (Viernes mañana)

Eugenio Aguirre Molina, Teoría grupo B.

Mari Carmen Pegalajar Jiménez, practicas grupo B2.

Antonio Manjavacas Lucas, practicas grupo B1.

Departamento de Ciencias de la Computación e Inteligencia Artificial

Tutorías Martes 8:30-10:30 y de 12:30-14:30 Jueves 9:30-11:30 Lugar sala de operadores (planta 2 aulas junto aula 2.1).

Correo: eco@ugr.es

Temario: Aprender a programar en C.

Método de evaluación:

- **Ordinaria**
  - **2 puntos Asistencia y Participación**
  - **3 puntos examen de prácticas ordenador.**
  - **5 puntos examen de teoría.**
- **Extraordinaria**
  - **5 puntos examen de prácticas.**
  - **5 puntos examen de teoría.**

TEMA 1. INTRODUCCIÓN

El marrón se utilizará para los títulos de las secciones, apartados, etc

El azul se usará para los términos cuya definición aparece por primera vez

El rojo fuerte se usará para destacar párrafos especialmente importantes

1.1. EL ORDENADOR, ALGORITMOS Y PROGRAMAS

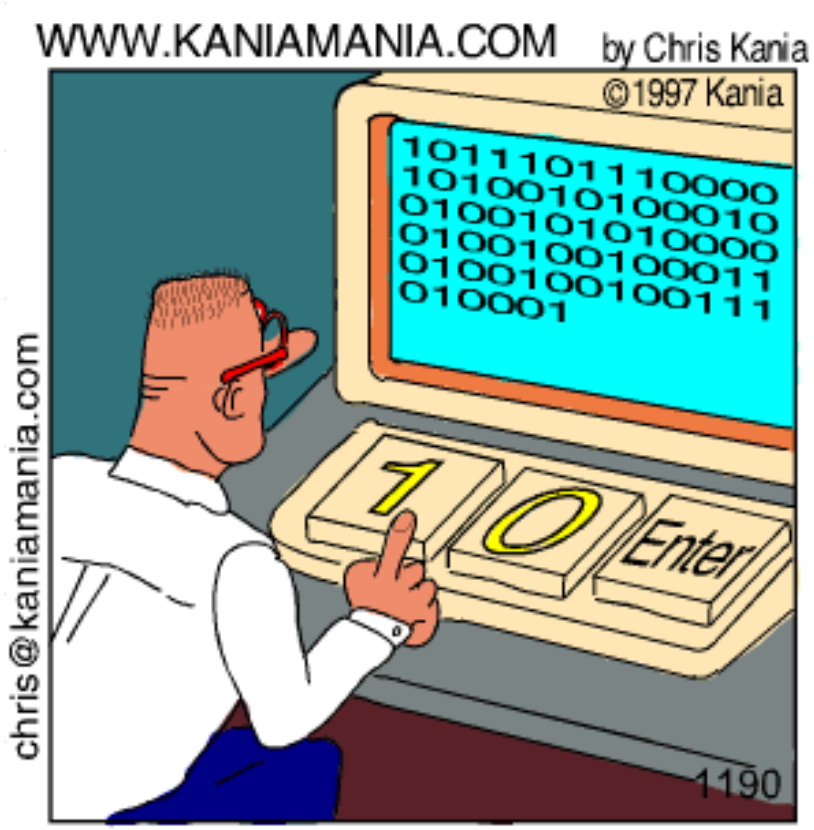
1.1.1. EL ORDENADOR: CONCEPTOS BÁSICOS

- ▷ *Hardware*
- ▷ *Software (Programa)*
- ▷ *Usuario*
- ▷ *Programador*
- ▷ *Lenguaje de Programación*

1.1.2. LENGUAJES DE PROGRAMACIÓN

- ***Programa*: Es el conjunto de instrucciones especificadas en un lenguaje de programación concreto, que pueden ejecutarse en un ordenador y que resuelven un problema**

  ***Código binario*:**



Real programmers code in binary.

- ***Lenguaje de programación***: **Lenguaje formal utilizado para comunicarnos con un ordenador e imponerle la realización concreta de un conjunto de órdenes.**
  - ***Lenguaje ensamblador***. **Depende del micropocesador (Intel 8086, Intel iWARP, Motorola 88000, etc) Se usa para programar drivers, microcontroladores (es un circuito integrado que lleva CPU, memoria y periféricos), compiladores, etc. Se ve en otras asignaturas.**

```
.model small
.stack
.data
   Cadena1 DB 'Hola Mundo.$'
.code
   mov ax, @data
   mov ds, ax
   mov dx, offset Cadena1
   mov ah, 9
   int 21h
end
```

  - ***Lenguajes de alto nivel*** **(C, C++, Java, Lisp, Prolog, Perl, Visual Basic, ...) En esta asignatura usaremos C.**

```
#include <stdio.h>

int main(){
   printf("Hola Mundo");
}
```

▶ *Reseña histórica del lenguaje C++*

1967 Martin Richards: BCPL para escribir S.O.

1970 Ken Thompson: B para escribir UNIX (inicial)

1972 Dennis Ritchie: C

1978 Libro *The C Programming Language, B. Kernigham, D. Ritchie.*

1983 Comité Técnico X3J11: ANSI C

1983 Bjarne Stroustrup: C++

1989 Comité técnico X3J16: ANSI C++

1990 Internacional Standarization Organization

`http://www.iso.org`

- ▷ **Comité técnico JTC1: Information Technology**
  - – **Subcomité SC-22: Programming languages, their environments and system software interfaces.**
    - ○ **Working Group 14: C**

      `http://www.open-std.org/jtc1/sc22/wg14/`
    - ○ **Working Group 21: C++**

      `http://www.open-std.org/jtc1/sc22/wg21/`

      Última revisión en 2003
      (y en 2006 sobre rendimiento)

Documentos *no oficiales*:
Borrador final de C++ estándar

`http://www.kuzbass.ru/docs/isocpp`

Incompatibilidades entre ISO C y ISO C++

`http://david.tribble.com/text/cdiffs.htm`

1.1.3. ALGORITMOS

Algoritmo*: es una secuencia ordenada de instrucciones que resuelve un problema concreto, atendiendo a las siguientes *características básicas:

- ▷ **Corrección (sin errores).**
- ▷ **Precisión (no puede haber ambigüedad).**
- ▷ **Repetitividad (en las mismas condiciones, al ejecutarlo, siempre se obtiene el mismo resultado).**
- ▷ **Finitud (termina en algún momento). Número finito de órdenes no implica finitud.**

► ***Características esenciales:***

- ▷ **Validez (resuelve el problema pedido)**
- ▷ **Eficiencia (lo hace en un tiempo y espacio aceptable) $\longrightarrow$ Proceso de Optimización.**

Usualmente, los algoritmos reciben unos *datos de entrada* con los que operan, y a veces o mejor dicho casi siempre, calculan unos nuevos *datos de salida*.

***Ejemplo.* Algoritmo de la media aritmética de N valores.**

- ▷ ***Datos de entrada*: N, valor1, valor2, ..., valorN**
- ▷ ***Datos de salida*: media**
- ▷ ***Instrucciones en lenguaje natural*:**
  Sumar los N valores y dividir el resultado por N

Ejemplo. **Algoritmo para la resolución de una ecuación de primer grado**
$ax + b = 0$

- ▷ ***Datos de entrada*****:** a, b
- ▷ ***Datos de salida*****:** x
- ▷ ***Instrucciones en lenguaje natural*****:**
  Calcular x **como el resultado de la división** $-b/a$

Podría mejorarse el algoritmo contemplando el caso de ecuaciones degeneradas.

Ejemplo. **Algoritmo para el cálculo de la hipotenusa de un triángulo rectángulo.**

- ▷ ***Datos de entrada*****: lado1, lado2**
- ▷ ***Datos de salida*****: hipotenusa**
- ▷ ***Instrucciones en lenguaje natural*****:**
  Calcular la hipotenusa como el resultado de la operación

$$\text{hipotenusa} = \sqrt{\text{lado1}^2 + \text{lado2}^2}$$

Ejemplo. **Algoritmo para ordenar una lista de valores numéricos.**

$$(9, 8, 1, 6, 10, 4) \longrightarrow (1, 4, 6, 8, 9, 10)$$

- ▷ ***Datos de entrada*****: el vector**
- ▷ ***Datos de salida*****: el mismo vector**
- ▷ ***Instrucciones en lenguaje natural*****:**
  - **– Calcular el mínimo valor de todo el vector**
  - **– Intercambiarlo con la primera posición**
  - **– Volver a hacer lo mismo con el vector formado por todas las componentes menos la primera.**

$$(9, 8, 1, 6, 10, 4) \rightarrow (1, 8, 9, 6, 10, 4) \rightarrow (X, 8, 9, 6, 10, 4)$$

$$\rightarrow (X, 4, 9, 6, 10, 8) \rightarrow (X, X, 9, 6, 10, 8) \rightarrow \cdots$$

***Implementación de un algoritmo*: Transcripción a un lenguaje de programación de un algoritmo.**

***Ejemplo.* Implementación del algoritmo para el cálculo de la media (para 4 valores) en C:**

```
suma = valor1 + valor2 + valor3 + valor4;
media = suma / 4;
```

***Ejemplo.* Implementación del algoritmo para el cálculo de la hipotenusa:**

```
hipotenusa = sqrt(lado1*lado1 + lado2*lado2);
```

Un programa incluirá la implementación de uno o más algoritmos.

***Ejemplo.* Programa para dibujar planos de pisos.**
Utilizará algoritmos para dibujar cuadrados, de medias aritméticas, salidas gráficas en plotter, etc.

Muchos de los programas que se verán en esta asignatura implementarán un único algoritmo.

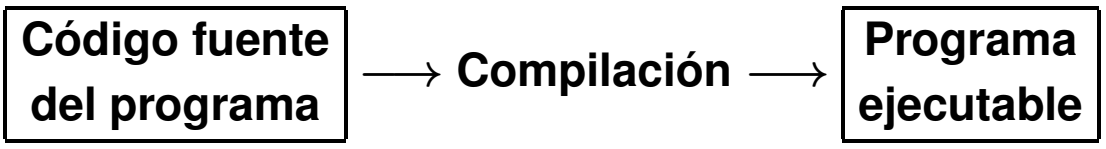
1.1.4. COMPILACIÓN

Al código escrito en un lenguaje concreto se le denomina *código fuente*. Éste se guarda en ficheros de texto normales (en el caso de C tienen extensión `.c`).

`Pitagoras.c`

```
#include <stdio.h>  /* Inclusión de recursos de E/S */
#include <math.h>   /* Inclusión de recursos matemáticos*/

int main(){            /* Programa Principal */
   float lado1;        /* Declara variables para guardar */
   float lado2;        /* los dos lados y la hipotenusa */
   float hip;


   printf("Introduzca la longitud del primer cateto: ");
   scanf("%f", &lado1);
   printf("Introduzca la longitud del segundo cateto: ");
   scanf("%f", &lado2);

   hip = sqrt(lado1*lado1 + lado2*lado2);

   printf("\nLa hipotenusa vale %f\n\n", hip) ;
}
```

`Pitagoras.c` es un simple fichero de texto. Para obtener el programa ejecutable (el fichero en binario que puede ejecutarse en un ordenador) a partir del código fuente, se utiliza un *compilador*:



La extensión usual del programa ejecutable en Windows es `.exe` También existen programas pequeños (como mucho 64K) con extensión .com

---

Código Fuente: `Pitagoras.c`

```
#include <stdio.h>



int main() {
 float lado1;
 ..........
 printf("Introduzca la longitud ...
 scanf("%f", &lado1);
 ..........
}
```

Compilador

Programa Ejecutable: `Pitagoras.exe`

```
10011000010000
10010000111101
00110100000001
11110001011110
11100001111100
11100101011000
00001101000111
00011000111100
```

1.2. Especificación de programas

1.2.1. Escritura y organización de un programa

- **Los programas en C pueden dividirse en varios ficheros aunque por ahora vamos a suponer que cada programa está escrito en un único fichero (por ejemplo,** `Pitagoras.c`**).**
- **Se pueden incluir comentarios en lenguaje natural.**

```
/* Comentario en
   varias líneas */
// Comentario 1 sola línea (en C++ no valido para C)
```

  El texto de un comentario no es procesado por el compilador.

- **Al principio del fichero se indica que vamos a usar una serie de recursos definidos en un fichero externo o *biblioteca***

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

- **A continuación aparece** `int main(){` **que nos indica que comienza el programa principal. Éste se extiende desde la llave abierta** `{`**, hasta encontrar la correspondiente llave cerrada** `}`**.**
- **Dentro del programa principal van las *sentencias* que componen el programa. Una sentencia es una parte del código fuente que el compilador puede traducir en una instrucción en código binario.**
- **Las sentencias van obligatoriamente separadas por punto**

y coma (`;`)

- ▷ **El compilador ejecuta las sentencias secuencialmente de arriba abajo. En el tema 2 se verá como realizar *saltos*, es decir, interrumpir la estructura secuencial.**
- ▷ **Cuando llega a la llave cerrada `}` correspondiente a `main()`, y si no ha habido problemas, el programa termina de ejecutarse y el Sistema Operativo libera los recursos asignados a dicho programa.**

Existen varios tipos de sentencias:

- ▷ **Sentencias de declaración de *datos*. Los datos son como variables matemáticas. Cada dato que usemos en un programa debe *declararse* al principio (después de `main`), asociándole un tipo de dato concreto.**

```
float lado1;
float lado2;
float hip;
```

Declara tres datos (*variables*) de tipo real que el programador puede usar en el programa.

- ▷ **Sentencias de cálculo, a través de la asignación `=`**

```
lado1 = 7;
lado2 = 5;
hip = sqrt(lado1*lado1 + lado2*lado2);
```

asigna a la variable `hip` el resultado de evaluar lo que aparece a la derecha de la asignación.
***Nota*. No confundir `=` con la igualdad en Matemáticas**

▷ **Sentencias para mostrar información en el periférico de salida establecido por defecto.**
En esta asignatura, dicho periférico será la pantalla, pero podría ser un fichero en disco o dispositivo externo (impresora, plotter, puerto FireWire, etc.).
Se construyen usando `printf`**, que es un recurso externo incluido en la biblioteca** `stdio`**.**

- **Lo que se vaya a imprimir va entre comillas y entre paréntesis después de** `printf`**.**
- **Lo que haya dentro de un par de comillas dobles se muestra tal cual, excepto los caracteres precedidos de** `\`**. Por ejemplo,** `\n` **hace que el cursor salte al principio de la línea siguiente.**

```
printf("Salto a la siguiente linea ->\n");
printf("\nEmpiezo en una nueva linea.");
```

```
%d --- Enteros
%f --- Numero coma flotante
%f --- Doubles
%c --- Caracteres
%s --- String
```

- **Los números se escriben entre comillas dobles.**

```
printf("3.1415927");
```

- **Si ponemos una variable, se imprime su contenido.**

```
printf("Longitud del primer cateto: ");
.....
hip = sqrt(lado1*lado1 + lado2*lado2);
printf("\nLa hipotenusa vale :");
printf("%f", hip);
```

Podemos usar una única sentencia:

```
printf("\nLa hipotenusa vale :%f",hip);
```

– **Aunque no es recomendable, también pueden ponerse *expresiones* como:**

```
printf("\nLa hipotenusa vale %f",
        sqrt(lado1*lado1+lado2*lado2));
```

Es importante dar un formato adecuado a la salida de datos en pantalla. El usuario del programa debe entender claramente el significado de todas sus salidas.

```
/* MAL: */
printf("%f %f", totalVentas, numeroVentas);

/* BIEN: */
printf("\nSuma total de ventas = %f", totalVentas);
printf("\nNúmero total de ventas = %f", numeroVentas);
```

▷ **Sentencias para leer datos desde el dispositivo de entrada establecido por defecto. Por ahora, será el teclado. Se construyen usando** `scanf`**, que es un recurso externo incluido en la biblioteca** `stdio`**.**
Su sintaxis más elemental es la siguiente

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

Por ejemplo, `scanf("%f",&lado1)`**, espera a que el usuario introduzca un valor real (**`float`**) desde el teclado (dispositivo de entrada) y, cuando se pulsa la tecla** `Intro`**, lo almacena en la variable** `lado1`**. Conforme se va escribiendo el valor, éste se muestra en pantalla, incluyendo el salto de línea.**

***Nota*. Cuando la entrada es un fichero (y no el teclado), la lectura es automática sin que se espere la tecla** `Intro`**.**

Es una asignación en tiempo de ejecución

Las entradas de datos deben etiquetarse adecuadamente:

```
/* MAL: */
scanf("%f", &lado1);
scanf("%f", &lado2);

/* BIEN: */
printf("Introduzca longitud del primer cateto: ");
scanf("%f", &lado1);
printf("\nIntroduzca longitud del segundo cateto: ");
scanf("%f", &lado2);
```

Estructura básica de un programa (los corchetes delimitan secciones opcionales):

[/* Breve descripción en lenguaje natural
de lo que hace el programa */]

[*Inclusión de recursos externos*]

```
int main(){
```
[*Declaración de datos*]

[*Sentencias del programa separadas por ;*]
```
}
```

1.2.2. ELEMENTOS BÁSICOS DE UN LENGUAJE DE PROGRAMACIÓN

1.2.2.1. TOKENS Y REGLAS SINTÁCTICAS

Cada lenguaje de programación tiene una sintaxis propia que debe respetarse para poder escribir un programa. Básicamente, queda definida por:

a) Los *componentes léxicos* o *tokens*. Un token es un carácter o grupo de caracteres alfanuméricos o simbólicos que representa la unidad léxica mínima que el lenguaje entiende. Por ejemplo:

```
main   ;   (   double   hip   *
```

son tokens de C. Pero por ejemplo, ni `|` ni `((*` son tokens válidos.

b) *Reglas sintácticas* que determinan cómo han de combinarse los tokens para formar sentencias.

Algunas reglas sintácticas se especifican con tokens especiales (formados usualmente por símbolos):

- **Separador de sentencias `;`**
- **Para agrupar varias sentencias se usa `{ }`**
  Se verá su uso en el tema 2. Por ahora, sirve para agrupar las sentencias que hay en el programa principal.
- **Para agrupar expresiones (fórmulas) se usa `( )`**

```
hip = sqrt( (lado1*lado1) + (lado2*lado2) );
```

1.2.2.2. PALABRAS RESERVADAS

Suelen ser tokens formados por caracteres alfabéticos.
Tienen un significado específico para el compilador, y por tanto, el programador no puede definir variables con el mismo identificador.

Algunos casos:

- ▷ `main`
- ▷ **Para definir tipos de datos como por ejemplo** `double`
- ▷ **Para establecer el *flujo de control*, es decir, para especificar el orden en el que se han de ejecutar las sentencias, como** `if`, `while`, `for` **etc.**
  Estos se verán en el tema 2.

```
/* Errores sintácticos */

int main(){
   float main;   /* main es un nombre de dato incorrecto */
   double double; /* double es un nombre de dato incorrecto */
}
```

Palabras reservadas comunes a C (C89) y C++

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

Palabras reservadas adicionales de C++

and	**and_eq**	**asm**	**bitand**	**bitor**	**bool**	**catch**
class	**compl**	**const_cast**	**delete**	**dynamic_cast**	**explicit**	**export**
false	**friend**	**inline**	**mutable**	**namespace**	**new**	**not**
not_eq	**operator**	**or**	**or_eq**	**private**	**protected**	**public**
reinterpret_cast	**static_cast**	**template**	**this**	**throw**	**true**	**try**
typeid	**typename**	**using**	**virtual**	**wchar_t**	**xor**	**xor_eq**

Palabras reservadas añadidas en C99

_Bool	**_Complex**	**_Imaginary**	**inline**	**restrict**

1.2.3. TIPOS DE ERRORES EN LA PROGRAMACIÓN

▷ ► ***Errores en tiempo de compilación*.**
Ocasionados por un fallo de sintaxis en el código fuente. No se genera el programa ejecutable.

```
/*  CONTIENE ERRORES  */
#include <stdi o>


int main{}(
  float lado1:
  float lado 2,
  float hip:

  lado1 = 2;
  lado2 = 3
  hip = sqrt(lado1**lado1 + ladp2*ladp2);
  printf("La hipotenusa vale %f",  hip);
)
```

▷ ► ***Errores en tiempo de ejecución*.**
Se ha generado el programa ejecutable, pero se produce un error durante la ejecución.

```
int dato_entero;
int otra_variable;

dato_entero = 0;
otra_variable = 7 / dato_entero;
.........
```

▷ ► ***Errores lógicos*.**

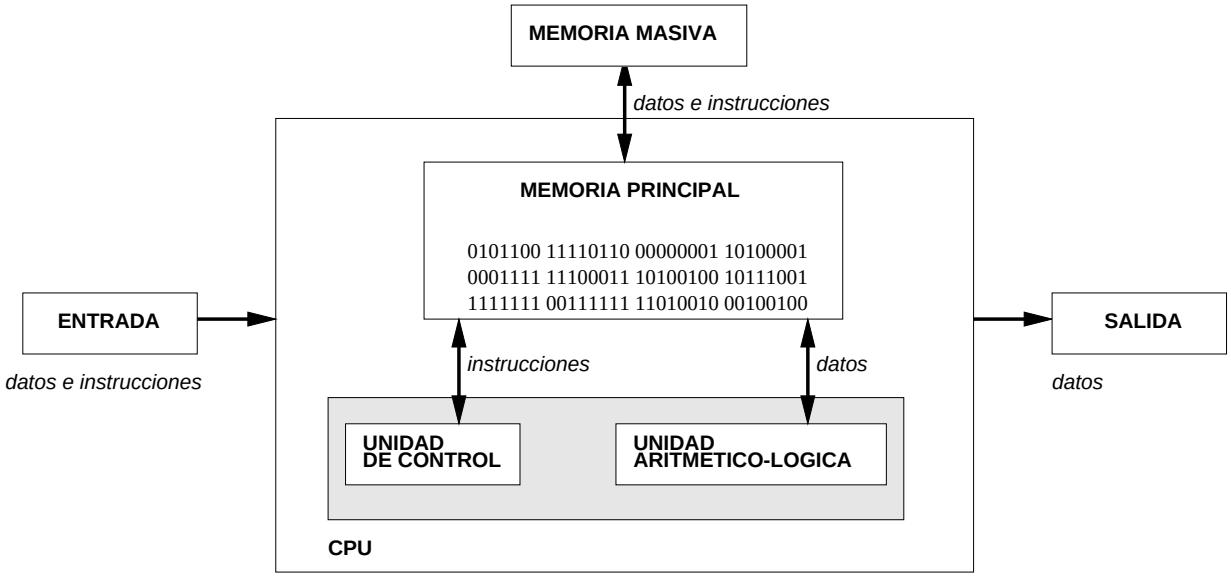
Se ha generado el programa ejecutable, pero el programa ofrece una solución equivocada.

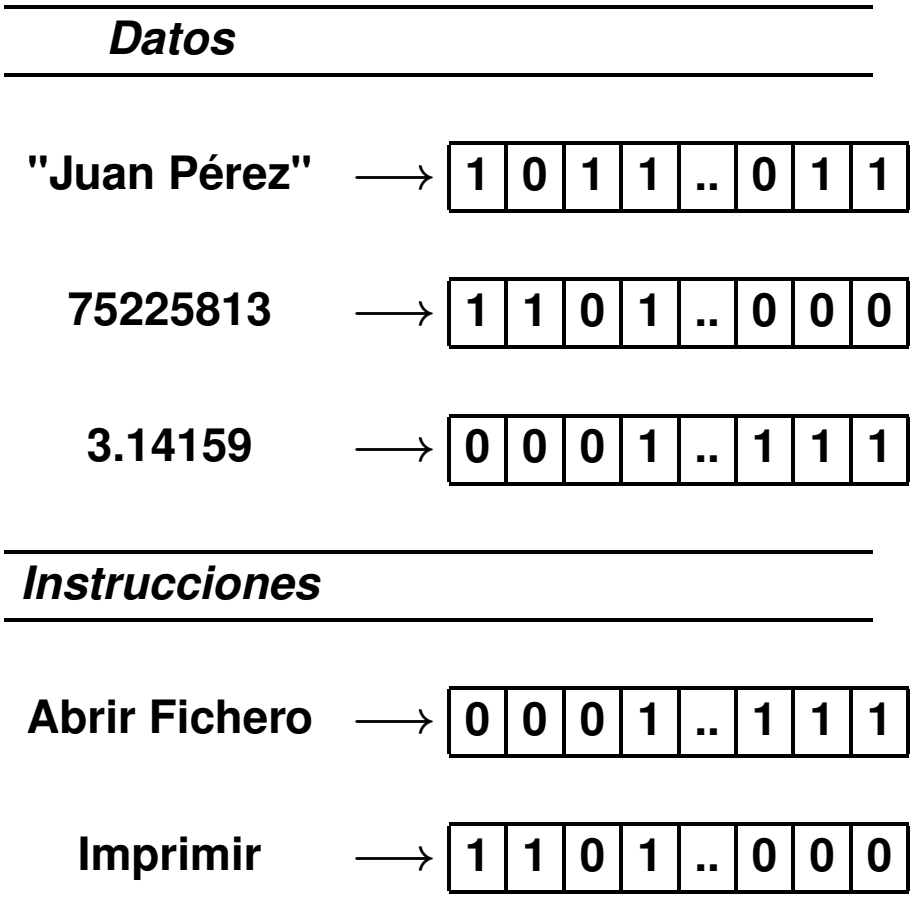
```
.........
lado1 = 4;
lado2 = 9;
hip = sqrt(lado1+lado1 + lado2*lado2);
.........
```

1.3. Datos y tipos de datos

1.3.1. Representación en memoria de datos e instrucciones

Un programa realiza instrucciones y opera con datos. Una vez compilado el programa, todo son combinaciones adecuadas de 0 y 1.



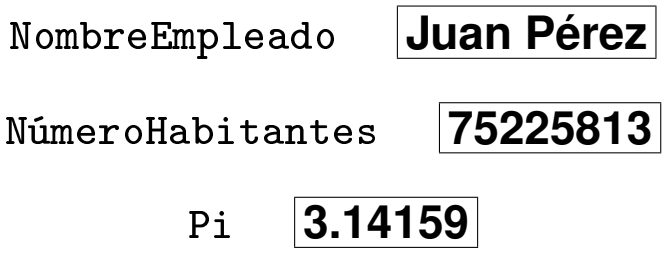


Nos centramos en los datos.

1.3.2. DATOS Y TIPOS DE DATOS

Al trabajar con un lenguaje de alto nivel, no haremos referencia a la secuencia de 0 y 1 que codifican un valor concreto, sino a lo que representa para nosotros.

Un *dato* es un conjunto de celdas o posiciones de memoria que tiene asociado un nombre (*identificador*) y un contenido (*valor*).



El compilador reconoce distintas categorías de datos (numéricas, texto, etc.). Necesitamos almacenar muchos tipos de información (enteros, reales, caracteres, cadenas de caracteres, etc.). Para cada tipo, el lenguaje ofrece un *tipo de dato*.
Por ejemplo, en C:

- ***Tipos***
  - ***Simples***
    - ***Numéricos***
      - ***Enteros*** : `int, long, ...`
      - ***Reales*** : `float, double, ...`
    - ***Caracteres*** : `char`
    - ***Lógico*** : `bool (C++)`
    - ***Puntero*** : `*`
  - ***Compuestos***
    - ***Vectores y matrices***
    - ***Cadena de caracteres*** :
      - ***Vectores de caracteres***, `string`
    - ***Registros (Estructuras)***
    - ***Clases (C++)***

Por ahora sólo usaremos `float`, `double`, `int` y `char`.

1.3.2.1. LITERALES

Son la especificación de un valor concreto de un tipo de dato. Dependiendo del tipo, tenemos:

- ***Literales numéricos*: son tokens numéricos.**
  Para representar datos reales, se usa el punto . para especificar la parte decimal:
  `2` `3` `3.5`
- ***Literales de cadenas de caracteres*: Son cero o más caracteres encerrados entre comillas dobles:**
  `"Hola"`
- **Literales de otros tipos, como *literales de caracteres*, etc.**

1.3.2.2. DECLARACIÓN DE VARIABLES

Cada variable que se use en un programa debe declararse al principio (antes o después de `main`), asociándole un tipo de dato concreto y un identificador. Serán variables locales a main si se declaran después de main. Si se declaran antes de main serán variables globales.

- ▷ **Al declarar una variable, el compilador reserva una zona de memoria para trabajar con ella. Ninguna otra variable podrá usar dicha zona.**
- ▷ **Cada variable debe estar asociado a un *único* tipo de dato (el tipo no puede cambiarse durante la ejecución).**
  El valor que se le puede asignar a una variable depende del tipo de dato con el que es declarado.

▶ *Formas de declarar variables*:

▷ Un único identificador por declaración:

```
<tipo> <identificador1>;
<tipo> <identificador2>;
...
```

Ejemplo. :

```
float lado1;
float lado2;
float hip;
```

▷ Varios identificadores por declaración:

```
<tipo> <identificadores separados por coma>;
```

Ejemplo:

```
float lado1, lado2, hip;
```

▷ Opcionalmente, se puede dar un valor inicial durante la declaración:

```
<tipo> <identificador> = <valor_inicial>;
<tipo> <identificador> = <valor_inicial>,
       <identificador>...;
```

Ejemplo. :

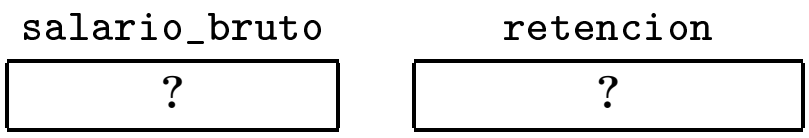
```
float dato = 4.5;
```

equivale a:

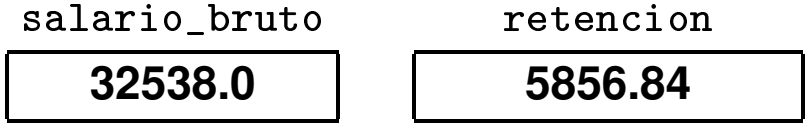
```
float dato;
dato = 4.5;
```

Cuando se declara una variable y no se inicializa, ésta no tiene ningún valor asignado *por defecto*. Puesto que desconocemos su valor, lo podremos considerar como *basura*, y lo representaremos gráficamente por `?`.

```
/* Programa para calcular la retención a aplicar en
   el sueldo de un empleado
*/
#include <stdio.h>

int main(){
   float salario_bruto; /* Salario bruto, en euros */
   float retencion;     /* Retención a aplicar, en euros */
```




```
   printf("Introduzca salario bruto: ");
   scanf("%f", &salario_bruto);

   retencion = salario_bruto * 0.18;

   printf("Retención a aplicar: %f" ,retencion);
}
```



Un error *lógico* muy común es usar una variable no asignada:

```
int main(){
   float salario_bruto; /* Salario bruto, en euros */
   float retencion;     /* Retención a aplicar, en euros */

   retencion = salario_bruto * 0.18;  /* :-( */

   printf("Retención a aplicar: %f", retencion);
}
```

1.3.2.3. DATOS CONSTANTES

Podríamos estar interesados en usar variables a las que sólo permitimos tomar un único valor, fijado de antemano. Es posible con las *constantes*. Se declaran como sigue:

```
const <tipo><identif>= <expresión>;
```

- **A las constantes se les aplica las mismas consideraciones que hicimos con las variables, sobre tipo de dato y reserva de memoria.**
- **Suelen usarse identificadores sólo con mayúsculas para diferenciarlos de las variables.**

```
/* Programa que pide el radio de una circunferencia e imprime
   su longitud y el área del círculo
*/

#include <stdio.h>

int main() {
   const float PI = 3.1416;
   float area, radio, longitud;

   printf("Introduzca el valor del radio ");
   scanf("%f", &radio);
   area = PI * radio * radio;
   longitud = 2 * PI * radio;
   printf("El área vale %f", area);
   printf("La longitud vale %f", longitud);

   PI = 3.15;  /* <- Error de compilación :-) */
}
```

Ventaja declaración de constantes:

- **Información al propio programador**
- **Imposibilidad de cambiarlo por error (`PI`)**
- **Código menos propenso a errores. Para cambiar el valor de `PI`, (por ejemplo por `3.1415927`), sólo hay que tocar en la línea de declaración de la constante.**

```
/* Programa para calcular la retención a aplicar en
   el sueldo de un empleado
*/
#include <stdio.h>
int main(){
   const float IRPF=0.18;  /* Porcentaje gravamen fiscal */
   float retencion;        /* Retención a aplicar, en euros */
   float salario_bruto;    /* Salario bruto, en euros */
```


```
   printf("Introduzca salario bruto: ");
   scanf("%f", &salario_bruto);

   retencion = salario_bruto * IRPF;

   printf("\n Retención a aplicar: %f", retencion);
}
```

Formalmente, al conjunto de variables y constantes, se le denomina *datos*. Así pues, los datos se pueden clasificar como:

$$\text{datos} \begin{cases} \text{variables} \\ \text{constantes} \end{cases}$$

- ▷ **Las variables son datos cuyo valor puede variar a lo largo del programa, p.e.,** `lado1`**.**
- ▷ **Las constantes son datos cuyo contenido no varía a lo largo de la ejecución del programa, p.e., el número π.**

Una sintaxis de programa algo más completa

```
[ /* Breve descripción en lenguaje natural
     de lo que hace el programa */ ]

[ Inclusión de recursos externos ]


int main(){
     [Declaración de constantes]
     [Declaración de variables]

     [Sentencias del programa separadas por ;]
}
```

1.3.2.4. NORMAS PARA LA ELECCIÓN DEL IDENTIFICADOR

Cada dato necesita un identificador único. Un identificador de un dato es un token formado por caracteres alfanuméricos con las siguientes restricciones:

- **Debe empezar por una letra o subrayado (_)**
- **No pueden contener espacios en blanco ni ciertos caracteres especiales como letras acentuadas, la letra eñe, las barras \ o /, etc.**
  Ejemplo: `lado1` `lado2` `precio_con_IVA`
- **El compilador determina la máxima longitud que pueden tener (por ejemplo, 31 caracteres)**
- **Sensibles a mayúsculas y minúsculas.**
  `lado` **y** `Lado` **son dos identificadores distintos.**
- **No se podrá dar a un dato el nombre de una palabra reservada.**
  No es recomendable usar el nombre de algún identificador usado en las *bibliotecas estándar* (por ejemplo, `printf`**)**

```
#include <stdio>

int main(){
   float printf; /* Error de sintaxis */
   float main; /* Error de sintaxis */
   ..........
}
```

Ejercicio. **Determinar cuáles de los siguientes son identificadores válidos. Si son inválidos explicar por qué.**

a) `registro1` **b)** `1registro` **c)** `archivo_3` **d)** `main`

e) `nombre y direccion` **f)** `dirección` **g)** `diseño`

► ***Consejos muy importantes para elegir el identificador de un dato*:**

- ▷ **El identificador de un dato debe reflejar su semántica (contenido).**
  - ☹ `l1, l2, hp`
  - ☺ `lado1, lado2, hip` **o, incluso mejor,** `hipotenusa`
- ▷ **Usaremos minúsculas, salvo la primera letra de nombres compuestos (y los identificadores de las constantes).**
  - ☹ `precioventapublico, tasaanual`
  - ☺ `precioVentaPublico, precio_venta_publico`
  - ☺ `pvp, TasaAnual, tasa_anual`
- ▷ **No se nombrarán dos datos con identificadores que difieran únicamente en la capitalización, o un sólo carácter.**
  - ☹ `cuenta, cuentas, Cuenta`
  - ☺ `cuenta, coleccionCuentas, cuentaPpal`

En el examen, baja puntos no seguir las anteriores normas.

1.4. Operadores y Expresiones

1.4.1. Terminología en Matemáticas

Notaciones usadas en los operadores matemáticos:

- **Prefija. El operador va antes de los argumentos. Estos suelen encerrarse entre paréntesis.**
  `seno(3), tangente(x), media(valor1, valor2)`
- **Infija. El operador va entre los argumentos.**
  `3+5 x/y`

Según el número de argumentos, diremos que un operador es:

- **Unario. Sólo tiene un argumento:**
  `seno(3), tangente(x)`
- **Binario. Tienes dos argumentos:**
  `media(valor1, valor2) 3+5 x/y`
- **n-ario. Tiene más de dos argumentos.**

1.4.2. Operadores en un lenguaje de programación

Cuando hablamos de Matemáticas, pensamos en valores numéricos enteros o reales, pero en general los *operadores matemáticos* trabajan con cualquier tipo de dato (lógico, matrices, complejos, etc) y son más generales ya que, por ejemplo, podrían modificar el dato sobre el que se aplican.

- ▷ **Suma de dos matrices**
- ▷ $p \wedge q$
- ▷ **Transposición de una matriz**

Los lenguajes de programación proporcionan operadores que permiten manipular los datos. Se denotan a través de tokens alfanuméricos o simbólicos.

► ***Tipos de operadores:***

- ▷ **Los definidos en el núcleo del compilador.**
  No hay que incluir ninguna biblioteca
  Suelen usarse tokens simbólicos para su representación y suelen ser infijos
  Ejemplos: `+` (suma), `-` (resta), `*` (producto), `/` (división), etc.
- ▷ **Los definidos en bibliotecas externas.**
  Suelen usarse tokens alfanuméricos para su representación y suelen ser prefijos. Si hay varios argumentos se separan por una coma.

```
sqrt(4)    sin(6.4)    pow(3,6)
```

  Las funciones de este ejemplo pertenecen a `math`.

Tradicionalmente se usa el término *operador* a secas para denotar los primeros, y el término *función* para los segundos.

Los operadores y funciones suelen devolver un dato y actúan sobre los datos (variables o constantes), literales y sobre el resultado que dan otros operadores y funciones.

```
int main() {
   const float PI = 3.1415926;
   float area, radio;

   printf("Introduzca el valor del radio ");
   scanf("%f", &radio);

   /* Asignaciones válidas: */

   area = PI * pow(radio, 2);
   area = PI * radio * radio;
```

En general:

```
   variable = Expresión
```

1.4.3. EXPRESIONES

Una *expresión* es una combinación de datos y operadores sintácticamente correcta, que el compilador evalúa y devuelve un valor.

```
3
3+5
lado1
lado1*lado1
lado1*lado1 + lado2*lado2
sqrt(lado1*lado1 + lado2*lado2)
```

Así pues, los operadores y funciones actúan, en general, sobre expresiones.

Las expresiones pueden aparecer a la derecha de una asignación, pero no a la izquierda.

```
3+5 = lado;  /* Error de compilación */
```

***Nota*. El operador = NO representa una igualdad matemática.**

¿Qué haría lo siguiente?

```
float dato = 4;
dato = dato + 3;
```

Una expresión NO es una sentencia de un programa:

- ▷ **Expresión:** `sqrt(lado1*lado1 + lado2*lado2)`
- ▷ **Sentencia:** `hip = sqrt(lado1*lado1 + lado2*lado2)`

Cuando el compilador evalúa una expresión, devuelve un valor de un tipo de dato (entero, real, carácter, etc.). Diremos que la expresión es de dicho tipo de dato.

Por ejemplo:

`3+5` **es una expresión entera**

`3.5+6.7` **es una expresión de reales**

```
int main(){
   float dato_real;
   int dato_entero;

   dato_real = 3.5+6.7;
   datos_entero = 3+5;
   ......
}
```

***Nota*. Cuando se usa una expresión dentro de `cout`, el compilador detecta el tipo de dato y la imprime de forma adecuada. Con `printf` tenemos que indicar el tipo con el formato.**

```
cout << "\nResultado = " << 3+5;       // C++
printf("\nResultado = %d", 3+5);       /* C */
cout << "\nResultado = " << 3.5+6.7;   // C++
printf("\nResultado = %f", 3.5+6.7);   /* C */
```

Imprime en pantalla:

```
Resultado = 8
Resultado = 10.2
```

1.5. Tipos de datos comunes en C

1.5.1. Rango y operadores aplicables a un tipo de dato

Las características que definen el comportamiento de un tipo de dato son:

- **El *rango* de valores que puede representar, que depende de la cantidad de memoria que dedique el compilador a su representación interna. Intuitivamente, cuanta más memoria se dedique para un tipo de dato, mayor será el número de valores que podremos representar.**

  ***Ampliación*:**

  - **La cantidad de memoria que el compilador asigna a cada tipo se puede consultar mediante el operador `sizeof(<tipo>)`[1] cuyo resultado es un entero que contiene el número de bytes asignados a `<tipo>`.**

- **El conjunto de *operadores* que pueden aplicarse a los datos de ese tipo.**

[1] `sizeof` **es un operador pues está en el núcleo del compilador, pero se usa en forma prefija, encerrando el argumento entre paréntesis, como las funciones**

1.5.2. LOS TIPOS DE DATOS ENTEROS

1.5.2.1. REPRESENTACIÓN DE LOS ENTEROS

Propiedad fundamental: Cualquier entero puede descomponerse como la suma de determinadas potencias de 2. Por ejemplo, para el 53:

$$53 = 0*2^{15}+0*2^{14}+0*2^{13}+0*2^{12}+0*2^{11}+0*2^{10}+0*2^{9}+0*2^{8}+0*2^{7}+$$

$$+0*2^{6}+1*2^{5}+1*2^{4}+0*2^{3}+1*2^{2}+0*2^{1}+1*2^{0}$$

La representación en binario sería la secuencia de los factores (1,0) -*bits*- que acompañan a las potencias:

$$0000000000110101$$

Ampliación*. La forma *fácil* de representar el signo sería añadiendo otro bit al principio para indicar si es positivo o negativo (*bit de signo*). En los ordenadores actuales se usan otras técnicas que agilizan los cómputos. Consultad en la Wikipedia: *Representación de números con signo

- ▷ **Dos elementos a combinar:** `1, 0`
- ▷ **r posiciones. Por ejemplo, $r = 16$**
- ▷ **Se permiten repeticiones e importa el orden**

$$0000000000110101 \neq 0000000000110110$$

- ▷ **Número de datos distintos representables = 2^r**

Ejecutad el applet

`http://www.mathsisfun.com/combinatorics/combinations-permutations-calculator.html`

1.5.2.2. RANGO DE LOS ENTEROS

Subconjunto del conjunto matemático Z. La cardinalidad dependerá del número de bits (r) que cada compilador utiliza para su almacenamiento.

Los compiladores suelen ofrecer distintos tipos enteros. En C: `short`, `int`, `long`, `__int64`**, etc. El más usado es** `int`**.**

El estándar de C++ no obliga a los compiladores a usar un tamaño determinado.

Lo usual es que un `int` **ocupe la longitud de la *palabra* del segmento de datos del sistema operativo. Hoy en día, lo usual es 32 bits (4 bytes) y 64 bits (8 bytes). En el caso de 32 bits el rango de un** `int` **es:**

$$\left[-\frac{2^{32}}{2}, \frac{2^{32}}{2} - 1\right] = [-2147483648, 2147483647]$$

```
int entero;
entero = 53;
```

Cuando necesitemos un entero mayor, podemos usar `__int64`**. Ocupa 64 bits (8 bytes) y el rango es:**

$$[-9223372036854775808, 9223372036854775807]$$

Literales enteros

Los *literales* son la especificación (dentro del código escrito de C) de un valor concreto de un tipo de dato. Los *literales enteros* se construyen con tokens formados por símbolos numéricos. Pueden empezar con un signo negativo `-`

```
53      -406778      0
```

***Nota*. En el código se usa el sistema decimal (53) pero internamente, el ordenador usa el código binario (000000000110101)**

¿Qué tipo de dato se usa para representar un literal entero? Depende de la memoria reservada y los tipos disponibles por cada compilador

- **El compilador usará el tipo de dato que se ajuste al tamaño del literal entero. Por ejemplo, si un** `int` **se representa con 32 bits en un compilador,** `600000000` **será un literal de tipo** `int`**.**

```
int entero;
entero = 600000000;              /* int = int */


__int64 gran_entero;
gran_entero = 600000000000000; /* __int64 = __int64 */
```

- **Los literales *demasiado grandes* provocarán un error de compilación.**

```
int entero;
entero = 6000000000000000000; /* Error compilación */
```

Posteriormente, se analizarán casos más complicados.

1.5.2.3. OPERADORES

Operadores binarios

`+ - * / %`

suma, resta, producto, división entera y módulo.
El operador módulo (%) representa el resto de la división entera
Devuelven un entero.
Binarios. Notación infija: `a*b`

```
int n;
n = 5 * 7;       /* Asigna a la variable n el valor 35 */
n = n + 1;       /* Asigna a la variable n el valor 36 */
n = 25 / 9;      /* Asigna a la variable n el valor 2 */
n = 25 % 9;      /* Asigna a la variable n el valor 7 */
n = 5 / 7;       /* Asigna a la variable n el valor 0 */
n = 7 / 5;       /* Asigna a la variable n el valor 1 */
n = 173 / 10;    /* Asigna a la variable n el valor 17 */
n = 5 % 7;       /* Asigna a la variable n el valor 5 */
n = 7 % 5;       /* Asigna a la variable n el valor 2 */
n = 173 % 10;    /* Asigna a la variable n el valor 3 */
5 / 7 = n;       /* Sentencia Incorrecta. */
5 / 7;           /* Sentencia Incorrecta. */
```

Operadores unarios de incremento y decremento

`++` **y** `--`

Incrementan y decrementan, respectivamente, el valor de la variable entera sobre la que se aplican (no pueden aplicarse sobre una expresión).
Unarios de notación postfija o prefija.

```
<variable>++;    /* Incrementa la variable en 1 */
++<variable>;    /* Es equivalente a:
                    <variable> = <variable> + 1; */

<variable>--;    /* Decrementa la variable en 1 */
--<variable>;    /* Es equivalente a:
                    <variable> = <variable> - 1; */
```

```
int dato=4;
dato = dato+1;        /* Asigna 5 a dato */
dato++;               /* Asigna 6 a dato */
```

Operador unario de cambio de signo

`-`

Cambia el signo de la variable sobre la que se aplica.
Unario de notación prefija.

```
int dato=4, aux1;
aux1 = -dato;         /* Asigna -4 a aux1 */
aux1 = -aux1;         /* Asigna 4 a aux1 */
```

1.5.2.4. EXPRESIONES ENTERAS

Son aquellas expresiones, que al evaluarlas, devuelven un valor entero.

```
entera        56        (entera/4+56)%3
```

¿Cómo evalúa el compilador la siguiente expresión?

```
dato = 3+5*7;
```

Posibilidad 1:

1. **Primero hacer 3+5**
2. **Después, el resultado se multiplica por 7**

El resultado sería: 56

Posibilidad 2:

1. **Primero hacer 5*7**
2. **Después, el resultado se suma a 3**

El resultado sería: 38

La forma que adopta el compilador depende de la *precedencia* de los operadores. Ante la duda, forzad la evaluación deseada mediante la utilización de paréntesis:

```
dato = 3+(5*7);        dato = (3+5)*7;
```

Reglas de precedencia:

```
()
-  (operador unario de cambio de signo)
* / %
+ -
```

Cualquier operador de una fila superior tiene más prioridad que cualquiera de la fila inferior.

```
variable = 3+5*7;  /* equivale a 3+(5*7) */
```

Los operadores de una misma fila tienen la misma prioridad. Se evalúan de izquierda a derecha según aparecen en una expresión.

```
variable = 3/5*7;  /* equivale a (3/5)*7 */
```

Ejercicio. **Teniendo en cuenta el orden de precedencia de los operadores, indicad el orden en el que se evaluarían las siguientes expresiones:**

a) `a + b * c -d` **b)** `a * b / c` **c)** `a * c % b - d`

Ejemplo. **Incrementar el salario en 100 euros y calcular el número de billetes de 500 euros a usar en el pago.**

```
int salario, num_bill500;
salario = 43000;
num_bill500 = (salario + 100) / 500;  /* ok */
num_bill500 = salario + 100 / 500;    /* Error lógico */
```

1.5.3. Los tipos de datos reales

Reales: subconjunto finito de R

- ▷ **Parte entera de $4{,}56 \longrightarrow 4$**
- ▷ **Parte real de $4{,}56 \longrightarrow 56$**

C ofrece distintos tipos para representar valores reales. Principalmente, `float` (usualmente 32 bits) y `double` (usualmente 64 bits).

```
float valor_real;
valor_real = 541.341;
```

1.5.3.1. Literales reales

Son tokens formados por dígitos numéricos y con un único punto que separa la parte entera de la parte real. Pueden llevar el signo `-` al principio.

```
800.457
4.0
-3444.5
```

También se puede utilizar notación científica:

```
3e-5
```

que representa el valor $3 \times 10^{-5} = 0{,}00003$

Importante:

- ▷ **El literal `3` es un entero.**
- ▷ **El literal `3.0` es un real.**

Los compiladores suelen usar el tipo `double` para representar literales reales.

1.5.3.2. REPRESENTACIÓN DE LOS REALES

¿Cómo podría el ordenador representar $541{,}341$?

Lo *fácil* sería:

- ▷ **Representar la parte entera 541 en binario**
- ▷ **Representar la parte real 341 en binario**

De esa forma, con 64 bits (32 bits para cada parte) podríamos representar:

- ▷ **Partes enteras en el rango $[-2147483648, 2147483647]$**
- ▷ **Partes reales en el rango $[-2147483648, 2147483647]$**

Este modo de representación recibe el nombre de representación en *coma fija.*

Sin embargo, la forma usual de representación no es así. Se utiliza la representación en *coma flotante* (*floating point*). En esta representación la parte entera se expresa en binario y la parte real se representa usando potencias inversas de 2. Por ejemplo:

$$1011$$

representaría

$$1 * \frac{1}{2^1} + 0 * \frac{1}{2^2} + 1 * \frac{1}{2^3} + 1 * \frac{1}{2^4} =$$

$$= 1 * \frac{1}{2} + 0 * \frac{1}{4} + 1 * \frac{1}{8} + 1 * \frac{1}{16} = 0{,}6875$$

***Problema*: Usando este método NO es posible representar de forma exacta muchos reales.**
Por ejemplo, $0{,}1$ o $0{,}01$ no se pueden representar de forma exacta. Por tanto, ¡*todas* las operaciones realizadas con los reales serán aproximadas!

1.5.3.3. Rango y Precisión

La codificación en coma flotante permite trabajar (en un mismo tipo de dato) con magnitudes muy grandes y muy pequeñas.

```
double valor_real;
valor_real = 100000000000000000000.0;   /* ok */
valor_real = 0.00000000000000000001;    /* ok! */
```

Pero el precio a pagar es muy elevado: muy poca precisión (tanto en la parte entera como la real). Veamos la parte entera.

```
double valor_real;
valor_real = 123456789012345.0;     /* :-) */
valor_real = 12345678901234567.0;   /* :-( */
    /* Se almacenará un valor aproximado. Por ejemplo:
       12345678901234568.0; */
```

La regla a usar es que con 32 bits, se consigue una precisión aproximada de 7 dígitos y con 64 bits, se consiguen aproximadamente 16 dígitos para la parte entera:

```
valor_real = 0.12345678901234567;
    /* Se almacenará un valor aproximado. Por ejemplo:
       0.12345678901234568; */

valor_real = 0.1;
    /* Ya lo vimos antes
       También se almacenará un valor aproximado.
       Por ejemplo: 0.09999899897 */
```

Cuanto mayor sea la parte entera a representar, menor será la precisión en decimales que obtendremos (sin contar que, además, muchos decimales como 0.1 no se pueden representar de forma exacta)

Por ejemplo, usando 32 bits, hay aproximadamente 8388607 números entre $1{,}0$ y $2{,}0$, mientras que sólo hay 8191 entre $1023{,}0$ y $1024{,}0$

Debido a la poca precisión, los programas financieros no trabajan con reales en coma flotante. Soluciones:

- ▷ **Trabajar con enteros y potencias de 10.**

```
int salario;
salario = 175023; /* 1750 euros y 23 céntimos */
salario = 143507; /* 1435 euros y 7 céntimos */
salario = 130000; /* 1300 euros */
```

En resumen:

- ▷ **Los tipos enteros representan datos enteros de forma exacta.**
- ▷ **Los tipos reales representan la parte entera de forma exacta, siempre que trabajemos con menos de 16 dígitos (en una representación con 64 bits)**
  La parte real será sólo aproximada. Además, la aproximación será mejor cuanto menor sea la parte entera del número.

1.5.3.4. OPERADORES

`+, -, *, /`

Binarios, de notación infija. También se puede usar el operador unario de cambio de signo (`-`)

Aplicados sobre reales, devuelven un real.

```
float r;
r = 5.0 * 7.0;     /* Asigna a r el valor  35.0 */
r = 5.0 / 7.0;     /* Asigna a r el valor 0.7142857 */
```

¡Cuidado! El comportamiento del operador `/` depende del tipo de los operandos: si todos son enteros, es la división entera. Si alguno es real, es la división real.

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

Los reales en coma flotante también permiten representar valores especiales como *infinito* `INF` y una *indeterminación* `NaN` (*Not a Number*)

```
float valor_real, divisor;
divisor = 0.0;
valor_real = 17/divisor;       /* Almacena INF */
valor_real = 1/valor_real;     /* Almacena 0.0 */
valor_real = divisor/divisor; /* Almacena NaN */
valor_real = 1/valor_real;     /* Almacena NaN */
```

1.5.3.5. FUNCIONES ESTÁNDAR

Hay algunas bibliotecas *estándar* que proporcionan funciones que trabajan sobre datos numéricos (enteros o reales) y que suelen devolver un real. Por ejemplo, `math`

`pow(), cos(), sin(), sqrt(), tan(), log(), log10(), fabs(), ....`

Todos los anteriores son unarios excepto `pow`**, que es binario. Devuelven un real.**

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


int main(){
   float real, r;

   real = 5.4;
   r = sqrt(real);
   r = pow(r, real);
}
```

1.5.3.6. EXPRESIONES REALES

Son expresiones cuyo resultado es un número real.
En general, diremos que las *expresiones aritméticas* o *numéricas* son aquellas expresiones bien enteras, bien reales.

```
float a, x, y, c, resultado;

resultado = x - sqrt(sqrt(a)-a*c))/(x-y);
```

Reglas de precedencia:

```
()
-  (operador unario de cambio de signo)
* /
+ -
```

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

`-b+sqrt(b*b-4.0*a*c)/2.0*a`	MAL
`((-b)+(sqrt(b*b - (4.0*a)*c)))/(2.0*a)`	BIEN
`(-b + sqrt(b*b - 4.0*a*c))/(2.0*a)`	RECOMENDADO

1.5.4. OPERACIONES CON TIPOS NUMÉRICOS DISTINTOS

1.5.4.1. USANDO ENTEROS Y REALES EN UNA MISMA EXPRESIÓN

Muchos operadores numéricos permiten como argumentos, expresiones de tipos distintos.

```
5.0 * 7.0;      Devuelve 35.0
5.0 / 7.0;      Devuelve 0.7142857
5 / 7;          Devuelve 0
5.0 / 7;        Devuelve 0.7142857
```

Para evaluar una expresión con tipos distintos, el compilador usará el tipo adecuado que sea capaz de albergar el resultado. Por ejemplo, usará un `float` **para almacenar el resultado de** `5.0 / 7`**, es decir,** `0.7142857`**.**

El operador de asignación también permite trabajar con tipos distintos. Muy importante: Primero se evalúa la expresión en la parte derecha de la asignación y luego se realiza la asignación.

```
float real;
int entero = 5;


real = 5;          /* float = int */
real = entero;     /* float = int */
```

Internamente, transforma (*casting* en inglés) el valor `5` **(en su representación como un** `int`**) al valor** `5.0` **(en su representación como un** `float`**)**

```
float real;
real = 5.0 / 7; /* float = float */
                /* Asigna a real el valor 0.7142857 */
real = 5 / 7;   /* float = int */
                /* Asigna a real el valor 0.0 */
```

Otro ejemplo:

```
int edad1 = 10, edad2 = 5;
float media;
media = (edad1 + edad2)/2;  /* :-( */
```

asigna a `media` **el valor** `7.0`**.**

Es un error lógico, difícil a veces de detectar.

Posible solución: ¿Declarar `edad1` **y** `edad2` **como** `float`**? El resultado sería** `7.5`**, pero no debemos cambiar el tipo asociado a la edad, pues su semántica es de un entero. Forzaremos la división real usando un literal real en un operando.**

```
int edad1 = 10, edad2 = 5;
float media;
media = (edad1 + edad2)/2.0;  /* :-) */
```

¿Qué pasa cuando asignamos un real a un entero? Simplemente, se pierde la parte decimal, es decir, se *trunca*.

```
float real;
int   entera;

real = 5.7;
entera = real;  /* Asigna a entera el valor 5 */
```

Este truncamiento también ocurre si ejecutamos

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

e introducimos un valor real.

***Casos especiales*:**

El casting a entero de un `float` que contenga infinito o indeterminación es un valor basura.

```
int entero, denominador;
float real;

denominador = 0;
real   = 17/(denominador*1.0); /* Almacena INF */
entero = real;                 /* Almacena basura */
real   = 0/(denominador*1.0);  /* Almacena NaN */
entero = real;                 /* Almacena basura */
entero = 17/denominador; /* Error ejecución ¿Por qué? */
```

1.5.4.2. PROBLEMAS DE DESBORDAMIENTO

Recordemos que con un literal demasiado grande, el compilador da un error al intentar asignarlo a una variable. Pero al operar con varios datos enteros, el compilador no puede realizar ninguna comprobación. Por tanto, es fácil salirse del rango representable. En este caso, se produce un desbordamiento y el resultado es un valor basura. Se produce un Error Lógico.

```
int entero;
entero = 600000000000;      /* Error Compilación :-) */
entero = 600000000*1000;    /* Almacena basura :-( */


int valor, incremento;
valor = 600000000;
incremento = 1000;
entero = valor * incremento; /* Almacena basura :-( */
```

***Nota*. El compilador es capaz de evaluar la expresión**

```
600000*1000000
```

El resultado `600000000000` lo puede almacenar en un `__int64`. El problema viene al hacer la asignación de una expresión de tipo `__int64` a una variable de tipo `int`.

Esto mismo ocurre también con los reales.

En general, debemos tener cuidado con las asignaciones a datos de un tipo más pequeño que la expresión de la asignación. Algunos compiladores pueden advertir de este hecho, durante el proceso de compilación, con un *aviso* (pero no es un error de compilación).

1.5.5. El tipo de dato carácter

1.5.5.1. Rango

Es un conjunto finito y ordenado de caracteres: letras minúsculas, mayúsculas, dígitos del 0 al 9 y otros caracteres especiales.

Hay varios tipos de dato de carácter: `char`, `wchar`, `signed char`, **etc. En esta asignatura usaremos** `char`**.**

Código ASCII (256 caracteres) Cada carácter tiene un código numerado de 0 a 255.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Literales de carácter

Son tokens formados por:

- **O bien un único carácter encerrado entre comillas simples:** `'!' 'A' 'a' '5' 'ñ'`

  Observad que `'5'` **es un literal de carácter y** `5` **es un literal entero**

  ¡Cuidado!: `'cinco'` **o** `'11'` **no son literales de carácter.**

- **O bien una *secuencia de escape*, es decir, el símbolo** `\` **seguido de otro símbolo, como por ejemplo:**

Secuencia	Significado
`\n`	Nueva línea (retorno e inicio)
`\t`	Tabulador
`\b`	Retrocede 1 carácter
`\r`	Retorno de carro
`\f`	Salto de página
`\'`	Comilla simple
`\"`	Comilla doble
`\\`	Barra inclinada

  Las secuencias de escape también deben ir entre comillas simples, por ejemplo, `'\n'`, `'\t'`, **etc.**

```
#include <stdio.h>

int main(){
   const char NUEVA_LINEA = '\n';
   char car;

   car = 'B';                          /* Almacena 'B' */
   printf("%cienvenidos",car);
   printf("%cEmpiezo a escribir en la siguiente línea",'\n');
   printf("%c%cAcabo de tabular esta línea",'\n','\t');
   printf(NUEVA_LINEA);
   printf("\nEsto es una comilla simple: %c",'\'');
}
```

Escribiría en pantalla:

```
   Bienvenidos
   Empiezo a escribir en la siguiente línea
           Acabo de tabular esta línea

   Esto es una comilla simple: '
```

1.5.5.2. FUNCIONES ESTÁNDAR

El fichero de cabecera `ctype` contiene varias funciones relativas a caracteres. Por ejemplo:

`tolower` `toupper`

```
#include <ctype.h>


int main(){
   char car;

   car = tolower('A');        /* Almacena 'a' */
   car = toupper('A');        /* Almacena 'A' */
   car = tolower('B');        /* Almacena 'b' */
   car = tolower('?');        /* Almacena '?' */
   ..................
```

1.5.5.3. EL TIPO `char` COMO UN TIPO ENTERO

En C, el tipo `char` es realmente un tipo entero pequeño. Cuando hacemos

```
   char c;
   c = 'A';
```

estamos asignando a la variable `c` el número de orden en la tabla ASCII del carácter `'A'`, es decir, el `65`.

```
   char c;
   c = 'A';    /* c contiene 65 */
```

De hecho, también podríamos poner:

```
char c;
c = 65;     /* c contiene 65 */
```

Lo anterior es posible con cualquier tipo entero:

```
int c;

c = 'A';     /* Almacena 65 */
c = 65;      /* Almacena 65 */
c = '7';     /* Almacena 55 */
c = 7;       /* Almacena 7 */
```

En definitiva, el tipo `char` es un tipo entero *pequeño* ya que está pensado para almacenar los números de órdenes de la tabla ASCII.

Veamos algunas consecuencias:

- **Podemos operar con los caracteres como si fuesen números:**

```
char c;          /* También valdría int c; */

c = 'A'+1;       /* Almacena 66 = 'B' */
c = 65+1;        /* Almacena 66 = 'B' */
c = '7'-1;       /* Almacena 54 = '6' */
c = 'A' + '7'; /* Almacena 120 ='x' */
```

▷ **También podemos usar el operador - con dos argumentos de carácter. Devuelve un entero:**

```
int orden;

orden = 'c' - 'a';  /* Almacena 2 */
```

1.5.6. EL TIPO DE DATO CADENA DE CARACTERES

Los literales de cadenas de caracteres son una sucesión de caracteres encerrados entre comillas dobles:
`"Hola"`, `"a"` **son literales de cadena de caracteres**

```
printf("Esto es un literal de cadena de caracteres");
```

Hay que destacar que las secuencias de escape también pueden aparecer en los literales de cadena de caracteres presentes en `printf`

```
int main(){
   printf("Bienvenidos");
   printf("\nEmpiezo a escribir en la siguiente línea");
   printf("\n\tAcabo de tabular esta línea");
   printf("\n");
   printf("\nEsto es una comilla simple '");
   printf(" y esto es una comilla doble \"");
}
```

Escribiría en pantalla:

```
Bienvenidos
Empiezo a escribir en la siguiente línea
        Acabo de tabular esta línea

Esto es una comilla simple ' y esto es una comilla doble "
```

Ejercicio. **Determinar cuales de las siguientes son constantes de cadena de caracteres válidas, y determinar la salida que tendría si se pasase como argumento a** `printf`

a) `"8:15 P.M."` **b)** `"'8:15 P.M."` **c)** `'"8:15 P.M."'`

d) `"Dirección\n"` **e)** `"Dirección'n"` **f)** `"Dirección\'n"`

g) `"Dirección\\'n"`

C ofrece dos alternativas para trabajar con estos datos:

- ▷ ***Vectores de caracteres*** **con terminador** `'\0'`**. Se verán dentro de un par de meses.**
- ▷ **usar punteros, el tipo char* de C**

```
int main(){
   char *mensaje_bienvenida = "\tTeleco Corporation\n";
   printf("%s", mensaje_bienvenida);
   ........
}
```

Las funciones para poder operar con un string nos vienen en la librería:

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

1.5.7. El tipo de dato lógico o booleano

Es un tipo de dato muy común en los lenguajes de programación que se utiliza para representar los valores verdadero y falso que suelen estar asociados a una condición.
En C++ se usa el tipo `bool`. En C no existe un 0 sera falso y todo aquello distinto de 0 (cero) será true.

1.5.7.1. Rango

Un dato lógico, sólo puede tomar dos valores: `true` (distinto de cero) (verdadero) y `false` (falso) (cero).

Una expresión lógica es una expresión cuyo resultado es un tipo de dato lógico.

1.5.7.2. Operadores

Son los operadores clásicos de la lógica NO, Y, O que en C son los operadores `!`, `&&`, `||`, respectivamente.

		&& (Y)	|| (O)
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

	! (NO)
true	false
false	true

Muchas funciones de `ctype` devuelven un valor lógico

```
isalpha     isalnum    isdigit   ...
```

Por ejemplo,

```
isalpha('3')
```

es una expresión lógica (devuelve `false`)

```
/*  Ejemplo de uso de dato lógico  */
#include <ctype.h>

int main()  {
   int es_alfa, es_alfanum, es_digito_numerico;
   int compuesto;

   es_alfa = isalpha('3');      /* Asigna 0 a es_alfa */
   es_alfanum = isalnum('3');   /* Asigna 1 a es_alfanum */
   es_digito_numerico = isdigit('3');
                        /* Asigna 1 a es_digito_numerico */

   compuesto = (es_alfa && es_alfanum);  /* Resultado: 0 */
   compuesto = (es_alfa || es_alfanum);  /* Resultado: 1 */
   compuesto = !es_alfa;                 /* Resultado: 1 */
   .......
}
```

1.5.7.3. Ilusión de Tipo bool en C

Utilizando la directiva de C para hacer sustitución simbólica `#define` podemos crear la ilusión de que existe el tipo booleano `bool`. Esta directiva se preprocesa antes de compilar el programa y se sustituyen todas las ocurrencias de la primera columna por la segunda columna.

```
 /*  Ilusión de tipo bool  */
#include <ctype.h>

#define bool int
#define true 1
#define false 0

 int main()  {
    bool es_alfa, es_alfanum, es_digito_numerico;
    bool compuesto;

    es_alfa = isalpha('3');       /* Asigna 0 a es_alfa */
    es_alfanum = isalnum('3');    /* Asigna 1 a es_alfanum */
    es_digito_numerico = isdigit('3');
                          /* Asigna 1 a es_digito_numerico */

    compuesto = (es_alfa && es_alfanum);  /* Resultado: 0 */
    compuesto = (es_alfa || es_alfanum);  /* Resultado: 1 */
    compuesto = !es_alfa;                 /* Resultado: 1 */
    .......
 }
```

1.5.7.4. Operadores Relacionales

Son los operadores habituales de comparación de expresiones numéricas.
Pueden aplicarse a operandos tanto enteros, reales, como de caracteres y tienen el mismo sentido que en Matemáticas. El resultado es booleano.

`==` **(igual),** `!=` **(distinto),** `<`**,** `>`**,** `<=` **(menor o igual) y** `>=` **(mayor o igual)**

Algunos ejemplos:

- **La expresión** `(4 < 5)` **devuelve valor** `true` **1**
- **La expresión** `(4 > 5)` **devuelve el valor** `false` **0**
- **La relación de orden entre caracteres se establece según la tabla ASCII.**
  La expresión `('a' > 'b')` **devuelve el valor** `false` **0.**

***Nota*:**

- `!=` **es el operador relacional distinto.**
- `!` **es la negación lógica.**
- `==` **es el operador relacional de igualdad.**
- `=` **es la operación de asignación.**

Tanto en `==` **como en** `!=` **se usan 2 signos para un único operador**

```
/*   Ejemplo de operadores relacionales */

int main(){
  int entero1 = 3, entero2 = 5;
  float real1, real2;
  int menor, iguales;

  menor = entero1 < entero2;     /* menor=1 */
  menor = entero2 < entero1;     /* menor=0 */
  menor = (entero1 < entero2) && !(entero2 < 7); /* menor=0 */
  real1 = 3.8;
  real2 = 8.1;
  menor = real1 > entero1;       /* menor=1 */
  menor = !menor && (real1 > real2); /* menor=0 */
  iguales = real1 == real2;      /* iguales=0 */
}
```

Veremos su uso en la *sentencia condicional*:

```
  if (4 < 5)
     printf("4 es menor que 5");

  if (!(4 > 5))
     printf("4 es menor o igual que 5");
```

Reglas de Precedencia:

```
()
!
< <= > >=
==  !=
&&
||
```

A es menor o igual que B y A no es mayor que B

```
int A=40, B=34;
int condicion;
condicion = A <= B && !A > B;        /* MAL */
condicion = (A <= B) && (!(A > B)); /* BIEN */
condicion = (A <= B) && !(A > B);   /* Recomendado */
condicion = (A <= B) && (A <= B);   /* Mucho mejor */
```

Observa que dejamos los paréntesis aunque según las reglas de precedencia no son necesarios. En este caso estamos acostumbrados a verlo así.

Ejercicio: Escribir una expresión lógica que devuelva `true` si un número entero `edad` está en el intervalo [0,100]

1.5.8. EL TIPO ENUMERADO

Supongamos un programa de contabilidad en el que queremos representar tres posibles valores de desgravación, a saber 7, 16 y 30 %. Alternativas:

- **Usar una única variable** `desgravacion`

```
int desgravacion;

desgravacion = 7;
......
```

  Propenso a errores: podemos poner por error el literal 17, que es aceptado en un `int`**.**

- **Usar tres constantes:**

```
const int desgravacion_7 = 7;
const int desgravacion_16 = 16;
const int desgravacion_30 = 30;
```

  Tedioso, pues siempre tendremos que manejar tres variables

▷ **Usar un tipo *enumerado*. Es un tipo entero que sólo acepta unos valores concretos especificados por el programador.**

```
float salario;
enum TipoDesgravacion {baja=7, media=16, alta=33};
enum TipoDesgravacion desgravacion;

printf("%d", baja);           /* Imprime 7 */
desgravacion = baja;   /* Almacena internamente 7 */
scanf("%f", &salario);
salario = salario - salario*desgravacion/100.0;

desgravacion = 5;       /* Error compilación :-) */
```

Si a una constante sólo se le podía asignar un único valor, a un tipo enumerado sólo se le puede asignar un número muy limitado de valores.

***Nota*. Si no se pone ningún valor en la definición del tipo, C le asigna a cada valor del enumerado el siguiente entero del asignado al valor anterior. Al primero le asigna el cero.**

```
enum TipoUsual {baja, media, alta=4, muy_alta};
enum TipoUsual otro;

otro = baja;
printf("%d", otro);       /* 0 */
printf("%d", media);      /* 1 */
printf("%d", alta);       /* 4 */
printf("%d", muy_alta);   /* 5 */
```