Tema 3. Estructura y componentes de un programa

1. Concepto de programa

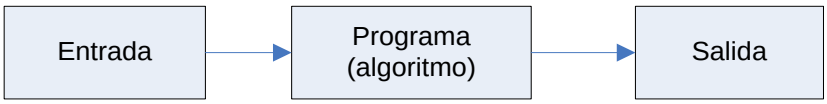
Se llama “programa” al conjunto de instrucciones (órdenes) que se dan a la computadora para la ejecución de una tarea determinada.

Se debe entender el programa como un *medio para conseguir un fin*. **La idea de fin u objetivo del programa está relacionada con la información que se requiere para resolver un problema determinado**.

El proceso de programación es un proceso de resolución de problema que se realiza siguiendo una metodología determinada. En esta asignatura proponemos el “ciclo de vida” de desarrollo tradicional que comprende las siguientes actividades: a) Definición o análisis del problema, b) Diseño del algoritmo de solución, c) Codificación, d) Compilación y ejecución, e) Verificación y depuración, f) Documentación, g) Mantenimiento.

Al iniciar su tarea, el programador debe tener bien claro qué tiene que hacer el programa y determinar de acuerdo al problema: *Entrada, Salida y Algoritmo de la solución.*

Conceptualmente, un programa puede ser considerado como una “caja negra”. Esta caja negra es el algoritmo de solución que permitirá obtener la salida en función de las entradas.



El programador debe definir de donde provienen las entradas al programa, es decir, debe indicar desde que dispositivo de entrada (teclado, disco, ...) ingresarán los datos.

El proceso de ingresar la información de entrada – **datos** – en la memoria de la computadora se denomina *entrada de datos*, *operación de lectura o* acción de *leer*.

Las salidas de los datos se deben presentar en dispositivos periféricos de salida: pantalla, impresora, disco, etc. La operación de salida se conoce también como *escritura* o acción de *escribir*.

2. Instrucciones y tipos de instrucciones

El proceso de *diseño del algoritmo* consiste en definir las **ACCIONES** o **INSTRUCCIONES** que resolverán el problema

Las *acciones o instrucciones* se deben escribir y posteriormente almacenar en memoria en la secuencia en que deben ejecutarse.

2.1. Tipos de instrucciones

Las instrucciones disponibles en un lenguaje de programación dependen de cada lenguaje específico. Las instrucciones básicas que soportan la mayoría de los lenguajes son:

1. Inicio/fin
2. Lectura (captura de datos)
3. Escritura (presentación de resultados)
4. Asignación (asignar valores a las variables o constantes)
5. Bifurcación (alteración de la secuencia del flujo de instrucciones)

Tipo de instrucción	Seudocódigo inglés	Seudocódigo español
Comienzo de proceso	BEGIN	INICIO
Fin de proceso	END	FIN
Entrada (lectura)	READ	LEER
Salida (escritura)	WRITE	ESCRIBIR
Asignación	A = 5	B = 7
Bifurcación condicional	IF	SI
Bifurcación incondicional	GOTO	IR

a) Instrucciones de inicio y fin

Estas instrucciones indican a la computadora el inicio del programa y la finalización del mismo. La indicación de fin de programa puede estar en cualquier lugar del programa y puede haber más de una instrucción de fin, sujeta a las condiciones que evalúa el programa. Por ejemplo, se finaliza el programa cuando se procesaron todos los datos de un conjunto de datos o cuando se encuentra un error severo de datos en una operación aritmética, etc.

b) Instrucciones de lectura de datos (entrada)

Las instrucciones de lectura dan ingreso a los datos que se procesan, esto es, indican las variables o posiciones de memoria en las que se almacenarán los datos en el momento de la ejecución del programa.

Ejemplo: LEER A, B, C

¿Cuál es el significado de esta instrucción?
Si se ingresan por teclado los valores 100, 50 y 30, entonces las variables de lectura tomarán los valores: A=100, B=50, C=30

Una instrucción de lectura deposita valores en las posiciones de memoria indicadas por los nombres de las variables.

c) Instrucciones de escritura de resultados (salida)

Una instrucción de escritura toma valores de las posiciones de memoria indicadas por los nombres de variables y escribe los datos en un dispositivo de salida.
¿Cuál es el significado de la instrucción siguiente?

ESCRIBIR A, B, C

Se mostrarán en pantalla los datos 100, 50 y 30 (se asume pantalla cuando no se especifica otro dispositivo).

d) Instrucciones de asignación

Una instrucción de asignación coloca un valor determinado en una variable o posición de memoria.

Ejemplo 1: **A = 80** la variable denominada A toma el valor 80

Otra forma usual de indicar la asignación es mediante una flecha. El sentido de la flecha indica el sentido de la asignación. En este caso sería: **A ← 80.**

Ejemplo 2: Indicar el valor de A, B y AUX al ejecutarse la instrucción 5.

1. A=10
2. B=20
3. AUX=A
4. A=B
5. B=AUX

El resultado de la evaluación después de ejecutar la instrucción 5 será: A=20, B=10, AUX=10.

*Ejemplo 3: ¿*Cuál es el valor de N después de la ejecución de esta asignación?:

N = N + 5 (Consideramos que N tiene un valor previo igual a 2).

En este ejemplo, el resultado de N es 7 porque al evaluar N + 5, N toma el valor de su contenido (que es 2), luego realiza la suma (2 + 5), y el resultado de la expresión (7) se guarda en la variable N.

Como ejemplo de instrucciones de asignación, tenemos 2 casos que se usarán con mucha frecuencia en la resolución de problemas:

Acumulador:

Un acumulador es una variable, definida por el programador, que hace referencia a una dirección de memoria que almacenará un "total móvil" de valores individuales a medida que vayan apareciendo en el proceso. Por ejemplo, las notas de los alumnos. Esta dirección o posición de memoria debe ser inicializada en cero.

Contador:
Es una variable que se incrementa en un valor constante y se utiliza para registrar el número de veces que se presenta un evento. Ejemplo: para contar los alumnos procesados, se incrementa en 1 por cada lectura de datos de alumnos.

e) Instrucciones de bifurcación

La alteración de la secuencia lineal de las instrucciones de un programa se realiza a través de las instrucciones de bifurcación.

Las bifurcaciones pueden ser *condicionales*, si depende del cumplimiento de una determinada condición (por ejemplo: IF N > 0 THEN fin); o *incondicionales* si la bifurcación no depende de ninguna condición (por ejemplo, GOTO a la instrucción 5). En general, las bifurcaciones incondicionales (GOTO) no son una práctica recomendada en programación, debido a que no respetan las estructuras de control propuestas por la programación estructurada.

Ejemplo de la implementación de los distintos tipos de instrucciones:

En el ejemplo siguiente podemos ver un algoritmo sencillo que lee datos de alumnos: Nº de libreta universitaria (LU) y las notas obtenidas en exámenes parciales (N1, N2). Se desea obtener el promedio de notas de cada alumno y su identificación.

El seudocodigo muestra la secuencia de instrucciones para implementar el algoritmo de solución. Una vez que el programa es codificado y ejecutado, las instrucciones irán procesando, paso a paso, los datos que se introduzcan. A la derecha se muestran:

a) Por un lado, los datos de entrada válidos que corresponden a tres alumnos. Los últimos datos se crean intencionalmente para generar la condición de fin de programa.

b) Por otra parte, se muestra el contenido de cada variable almacenada en la memoria RAM, después de cada iteración del proceso.

c) Finalmente, los datos de salida que se mostrarán en cada instrucción de salida.

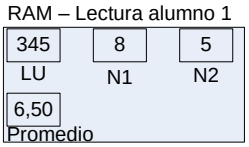
```
ALGORITMO PromedioNotas
VAR
      ENTERO: LU, N1, N2
      REAL: promedio
INICIO
      LEER LU, N1, N2
      MIENTRAS N1<> 0
           promedio = (N1 + N2)/2
           ESCRIBIR promedio
          LEER LU, N1, N2
       FIN-MIENTRAS
FIN
```

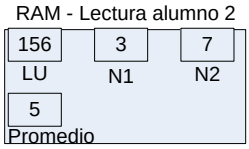
ENTRADA

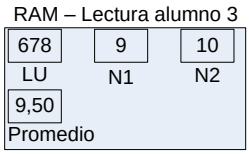
LU	N1	N2
345	8	5
156	3	7
678	9	10
000	0	0

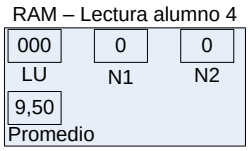
SALIDA

LU	Promedio
345	6,50
156	5
678	9,50









Es importante comprender los siguientes conceptos:

1) Las **variables** indicadas en un programa son **posiciones de memorias** (celdas) que internamente serán reconocidas con una "**dirección**" (un número binario).

2) El programa opera con las posiciones de memoria o variables, es decir, procesa el contenido de las posiciones que representan el valor de los datos.

3) Las instrucciones de lectura asignan datos de entrada a las posiciones de memoria, en forma "destructiva", es decir, cada nueva lectura destruye el valor anterior de la variable y coloca un nuevo valor.

4) La instrucción de repetición de un bucle (MIENTRAS) repite el conjunto de instrucciones hasta que se cumpla una condición N=0 (bifurcación condicional) que posibilita cambiar la linealidad de ejecución de las instrucciones, en este caso, para terminar el programa.

5) La instrucción de asignación para el cálculo del promedio evalúa la expresión a la derecha del signo igual y el resultado lo coloca en la variable a la izquierda del signo.

6) La instrucción de salida muestra, en el dispositivo indicado (en este caso una impresora en el diagrama), el contenido de las variables referenciadas en la instrucción.

El algoritmo propuesto puede ejecutarse para N alumnos con la misma eficacia. En cada iteración se dispone de los datos de UN ALUMNO, de aquí que los datos deben ser usados (operados) mientras se encuentren en la memoria principal, es decir, en el ciclo de lectura.

Para consolidar estos conceptos, modifique este algoritmo para que además de la salida indicada, calcule y muestre la cantidad de alumnos procesados. Siga las instrucciones paso a paso registrando las modificaciones de las variables en la memoria principal, como resultado de las instrucciones ejecutadas, utilizando los mismos casos de prueba.

Esta actividad forma parte de la etapa de **Verificación y depuración de un programa**, consiste en la definición de casos de prueba para comprobar los resultados.

3. Elementos básicos de un programa

En programación se debe tener presente la diferencia entre diseño del algoritmo y su implementación en un lenguaje de programación específico. Sin embargo, una vez que se comprendan los conceptos de programación, la codificación en un nuevo lenguaje de programación será relativamente fácil.

Los lenguajes de programación tienen elementos básicos que se usan como bloques constructivos, así como reglas, que componen su ***sintaxis***. Solamente las instrucciones sintácticamente correctas serán reconocidas por la computadora, los programas con errores de sintaxis no serán ejecutados.

Los elementos básicos constitutivos de un programa o algoritmo son:

- Identificadores (nombres de variables, funciones, procedimientos, etc.)
- Palabras reservadas (INICIO, FIN, SI, MIENTRAS…)
- Caracteres especiales (coma, apóstrofos, etc.)
- Constantes
- Variables
- Expresiones
- Instrucciones

a) Un identificador es una secuencia de caracteres, letras dígitos y subrayados (_), usados para identificar a las variables, funciones, procedimientos, que se utilizan en el programa. En general, el primer carácter debe ser una letra.
Ejemplos: *edad*, *notaExamen*, *índice*, …

El lenguaje C es sensible a las mayúsculas (***case sensitive***) por tanto, los identificadores *Alfa* y *alfa* son distintos.

Se recomienda escribir identificadores de variables en *minúsculas*, las constantes en *mayúsculas* y las funciones *iniciando con minúscula y cada letra intermedia con mayúscula.*

b) Las **palabras reservadas** son identificadores predefinidos (tienen un significado especial) que no es posible utilizar para otros fines distintos para los que han sido definidas. En todos los lenguajes de programación existe un conjunto de palabras reservadas. Por ejemplo, en C, algunas de ellas:

void, int, char, case, const, float

c) Comentarios

En C los comentarios tienen el formato: /* …. */ y pueden extender a varias líneas. Constituyen una valiosa documentación interna del programa. Ejemplo:

```
/* Calcula el promedio de notas de los alumnos */
```

d) Signos de puntuación y separadores

En C las instrucciones deben terminar con un punto y coma. Otros signos de puntuación usados son:

% & * () { } / : [ ] < >, entre otros

Los separadores son espacios en blanco, tabulaciones, retornos de carro y avance de línea.

Los últimos dos son códigos de control dependientes de los sistemas operativos y las aplicaciones.

e) Archivos de cabecera

Un archivo de cabecera es un archivo especial que contiene las declaraciones de elementos y funciones de biblioteca. Para utilizar macros, constantes, tipos y funciones almacenadas en una biblioteca, un programa debe usar la directiva **`#include`** para insertar el archivo de cabecera correspondiente.

Por ejemplo, si un programa utiliza la función **pow** (potencia) que se encuentra en la biblioteca **math.h,** debe contener la directiva:

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

Además de estos elementos básicos, existen otros cuya comprensión es esencial para el diseño correcto de algoritmos y programas. Estos elementos son: bucles, contadores, acumuladores, estructuras de control (secuencia, selección, repetitivas).

El conocimiento del funcionamiento de estos elementos y de cómo se integran en un programa, constituyen las **técnicas de programación** que todo buen programador debe conocer.

3.1. Datos, tipos de datos y operaciones primitivas

Un *dato* es la expresión general que describe los objetos con los que opera la computadora. En general, las computadoras pueden operar con varios *tipos de datos*. Los programas y algoritmos operan sobre esos tipos de datos.

En el proceso de resolución de problemas el diseño de las estructuras de los datos es tan importante como el diseño del algoritmo y del programa que se basa en el mismo.

Ejemplos de datos son: el nombre de una persona, el valor de una temperatura, una cifra de venta de supermercado, la fecha de un cheque, etc.

Tipos de datos:

Los distintos tipos de datos se representan en diferentes formas en la computadora. A nivel de máquina, un dato es un conjunto de bits (dígitos 0 o 1). Los lenguajes de alto nivel permiten basarse en abstracciones e ignorar los detalles de la representación interna. Aparece el concepto de *tipo de dato*, así como de su representación.

El tipo de un dato asocia **un determinado rango de valores** que el dato puede tomar durante la ejecución del programa. Si se intenta asignar un valor fuera del rango se producirá un **error**.
La asignación de tipos a los datos tiene **dos objetivos** principales:

- detectar errores en las operaciones
- determinar cómo ejecutar estas operaciones

Un lenguaje de programación se dice que es fuertemente **tipado** cuando:

- todos los datos deben tener un *tipo* declarado explícitamente

- existen ciertas restricciones en las expresiones en cuanto a los tipos de datos que en ellas intervienen.

Una ventaja de los lenguajes *fuertemente tipados* es que se requiere menos esfuerzo en depurar (corregir) los programas gracias a la cantidad de errores que detecta el compilador.

Los datos pueden ser: **simples o compuestos**

Un tipo de dato **simple** es aquel cuyo contenido se trata como una unidad que no puede separarse en partes más elementales. Los más usuales son:

- Numéricos (*enteros, reales*)
- Lógicos (*booleano*)
- Carácter (*char*)

Un tipo de dato **compuesto o estructurado** es aquel que permite almacenar un conjunto de elementos bajo una estructura particular, darle un único nombre, pero con la posibilidad de acceder en forma individual a cada componente. Ejemplo de datos estructurados son los arreglos y registros, que veremos en detalle en temas posteriores.

A continuación, veremos las características de los principales tipos de **datos simples**:

a) datos numéricos

En la sección final de este documento, **Representación de información** se brinda más información acerca de la representación de números.

El tipo numérico es el conjunto de los valores numéricos. Pueden representarse en dos formas distintas:

- **Numérico *entero***
- **Numérico *real***

Entero: El tipo entero es un subconjunto finito de los números enteros. Pueden ser positivos o negativos y no tienen decimales. Ejemplos: 5, 15, -30, 12567.

Los lenguajes de programación ofrecen distintos tipos de datos enteros. En la mayoría se encuentran enteros de 2 y 4 bytes.


En C algunos de los **tipos de datos enteros** se muestran en la tabla 1:

Tabla 1: datos de tipo entero en C

Tipo de dato	Rango		Tamaño en bytes
	desde	hasta	
short	-128	127	1 byte
unsigned short	0	255	1 byte
int	-32768	32767	2 bytes
unsigned int	0	65535	2 bytes
long	-2147483648	2147483647	4 bytes
unsigned long	0	4294967295	4 bytes

La cantidad de bytes puede variar según el compilador y la plataforma de hardware, cada uno de estos tipos de dato puede ocupar desde 1 byte hasta 8 bytes en memoria.

Real: El tipo real es un subconjunto de los números reales. Pueden ser positivos o negativos y tienen punto decimal. Un número real consta de una parte entera y de una parte decimal. Ejemplos: -45,78, 3,0, 0,008.

En aplicaciones científicas se requiere una representación especial para manejar números muy grandes, como la masa de la Tierra, o muy pequeños, como la masa de un electrón.

Una computadora sólo puede representar un número fijo de dígitos que depende del tamaño de la *palabra.* Una palabra es el número de bits con los que opera internamente una computadora. El tamaño de la palabra puede ser 1 byte, 2 bytes, 4 bytes u 8 bytes. Esta cantidad limitada de bits

puede afectar la representación de números muy grandes o muy pequeños, de aquí surge la coma flotante para disminuir estas dificultades.

En C los **tipos de datos reales** son: **float** (4 bytes) y **double** (8 bytes).

Tabla 2: datos de tipo real en C

Tipo de dato	Rango		Tamaño en bytes
	desde	hasta	
float	3.4E-38	3.4E38	4 bytes
double	1.7E-308	1.7E308	8 bytes

Para datos de este tipo se utiliza la *notación* científica. Igual que en las calculadoras, el dígito que se encuentra a continuación de la **E** representa la potencia a la que se elevará el número 10 para multiplicarlo por la cantidad a la izquierda de dicha **E**: Ejemplos:

3.0E5 = 3.0 * 10^5 = 3.0 * 100000 = 300000

1.5E-4 = 1.5 * 10^-4 = 1.5 * 0.0001 = 0.00015

Los tipos de datos reales, empiezan a perder precisión cuando se sobrepasa el valor de sus bits de mantisa. Así, por ejemplo, en 24 bits solo se puede almacenar un número de 2^{24}, esto es, aproximadamente 16.7 millones, cuando supera este valor la variable que contiene el número pierde precisión y almacena un valor aproximado. **Cuanto mayor es la precisión, menor pérdida de datos habrá.**

b) datos lógicos (booleanos)

El tipo *lógico*, también denominado *booleano,* sólo puede tomar 2 valores: **verdadero** (*true*) o **falso** (*false*).

La asignación **prueba = true**, asigna el valor lógico verdadero a una variable de tipo lógico.

Los compiladores de C que siguen la norma ANSI no incorporan el tipo de dato lógico, pero simula este tipo de dato utilizando el tipo de dato **int**. C interpreta todo valor distinto de 0 como "verdadero" y el valor 0 como "falso".

Ejemplo:

```
indicador = 0;          /* indicador toma el valor Falso */
indicador = suma > 10   /* indicador toma el valor 1 (Verdadero) si la suma es mayor
                           que 10, caso contrario, toma el valor 0 (Falso) */
```

c) datos de tipo Carácter (Char) y tipo Cadena (String)

El tipo carácter (*char*) es el conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato de este tipo contiene un solo símbolo. Los caracteres que reconocen las diferentes computadoras no son estándar, pero la mayoría reconoce los caracteres: *alfabéticos*: letras A la Z en mayúsculas y minúsculas, los *dígitos* de 0 al 9, caracteres *especiales*: signos de puntuación y otros símbolos. En general, estos caracteres se almacenan en ASCII y el orden de los caracteres está dado por el valor decimal que le asigna este código. Por ejemplo, en la tabla 5 podemos ver que la letra "A" mayúscula tiene el valor decimal 65 y que la letra "a" minúscula tiene el valor decimal 97.

En la sección final de este documento, **Representación de información** se brinda más información acerca de los códigos de representación de datos más utilizados, como **ASCII** y **UNICODE**.

Un valor constante de tipo carácter se escribe entre apóstrofos, por ejemplo, la expresión **letra = 'X'**, asigna a la variable de tipo carácter *letra* el valor constante "X".

Una cadena (*string*) de caracteres representa un conjunto de caracteres. Se puede trabajar con una variable de este tipo como si se tratara de una unidad, sin embargo, **no es un dato de tipo simple** dado que está integrado por elementos a los cuales se puede acceder en forma individual. Se trata de un tipo de datos **estructurado.**

Una cadena de caracteres tiene dos características importantes: la *longitud física* y la *longitud dinámica* o *lógica*. La longitud física se define en la declaración del tipo de dato y permite al procesador reservar el espacio máximo de memoria necesario para almacenar el valor de una variable de ese tipo. Ejemplo:

STRING nombre [10]

nombre = “Ema”

En este ejemplo, la longitud física de la variable “*nombre”* es 10 y la longitud dinámica es 3.

C procesa datos de tipo carácter utilizando el tipo de dato **char**. Utilizando la estructura array, que se verá más adelante, puede representar cadenas de caracteres.

Ejemplo:
```
char letra, respuesta;
letra = 'A';
respuesta = 'S';
```

Internamente los caracteres se almacenan como números. Por ejemplo, la letra ‘A’ es el valor 65. Por tanto, se pueden realizar operaciones aritméticas.

Por ejemplo: para convertir la “a” minúscula “A” mayúscula, se resta 32:
```
char letra= 'a';              /* asigna a minúscula        */
letra = letra – 32;       /* convierte a mayúscula */
```
La resta cambia el valor decimal de la representación ASCII: **a (97**) y **A (65).** Verificar con la tabla 5 de la sección Representación de información.

3.2. Constantes y variables

Constantes

Los programas muchas veces requieren valores que no deben cambiar durante la ejecución del programa. Estos valores se denominan *constantes*. Son espacios de la memoria principal cuyo contenido permanece sin variaciones durante la ejecución. Se identifican con un nombre y poseen un tipo de dato.

Ejemplos de constantes:

Constante entera: 125 12599 (no se utiliza punto separador de miles)

Constante real: 3.141592, -0.1234 (válidas) 1,456.63 (inválida, no se permiten comas)

Observaciones: Tener en cuenta que el separador decimal (punto o coma) depende del lenguaje de programación o de la configuración del sistema operativo. Para C, el punto decimal es el punto (“.”).

Una constante de tipo carácter (char) es un carácter del código ASCII encerrado entre apostrofes.

Una constante de tipo cadena contiene un conjunto de caracteres ASCII.

En C, las constantes de tipo **char** se encierran entre apostrofes y las constantes de tipo **cadena** se encierran entre comillas.

Ejemplos:

Constante de tipo char: ‘A’, ‘m’, ‘5’

Constante de tipo cadena: “9 de julio 1449”, “Ciencias Exactas”, “Juan Pérez”

 Tipo de datos en C: https://www.youtube.com/watch?v=pyfi1FMWmtk

Variables

Los valores que cambian durante la ejecución del programa se llaman **variables**, y corresponden a espacios de la memoria principal que se identifican con un nombre y son de un tipo de dato, y pueden cambiar de contenido si el programador así lo especifica con las instrucciones adecuadas.

Una variable se identifica con los siguientes atributos: **Nombre** que la identifica y **Tipo** que describe el tipo de contenido de la variable.

El **nombre** de una variable es lo que permite utilizar el espacio de memoria reservado para la misma y el valor que allí se almacena. Los posibles nombres de las variables dependerán del lenguaje de programación utilizado (longitud, caracteres válidos), pero en general se buscará respetar lo siguiente:

- Debe ser descriptivo de los valores que se le asignarán (esto se conoce como *mnemotécnico*). Los nombres de variables *edad, importeVenta, precioUnitario, cantMaterias, totalGeneral*, resultarán descriptivos en el contexto del problema.
- No utilizar palabras reservadas ni caracteres especiales (espacio, letras acentuadas, ñ, coma, punto). En general se recomienda limitarse a letras (a b … z), dígitos (0 1 2 … 9) y guion bajo (_) para separar palabras. Algunos lenguajes distinguen mayúsculas de minúsculas en los nombres de las variables (por ejemplo, NOTA no es lo mismo que Nota o nota). C es sensible a las mayúsculas.

El **tipo de dato** se especifica cuando se declaran las variables. Los datos a utilizar ya sean de entrada, de salida, o para cálculos intermedios o auxiliares, se declaran antes de iniciar cada algoritmo. Tendremos entonces una primera sección donde se declara el nombre y tipo de cada dato, y respetaremos estas condiciones durante todo el desarrollo del algoritmo.
Por ejemplo, en seudocódigo:

```
VAR
    REAL: precio_unitario, precio_final
    ENTERO: cantidad
    STRING: domicilio [30]
    CHAR: codigo
```

Se declaran tres variables numéricas: dos de tipo real y una de tipo entero, una variable de tipo carácter y otra de tipo cadena.

No todos los lenguajes de programación requieren que se declaren las variables a utilizar: algunos permiten usar variables que no fueron declaradas previamente, y las crean cuando se les asigna un valor.

 En C las variables se declaran antes de usarlas, al principio del bloque que conforma su ámbito.

Ejemplos:

```
short diasSemana;
int horasAcumuladas;
float importeFactura;
char codigoSexo;
```

3.3. Expresiones

Las expresiones son combinaciones de constantes, variables, operadores, paréntesis y nombres de funciones. Maneja las mismas ideas que la notación matemática convencional.

Por ejemplo: a + (b+3) + √C. Los paréntesis indican el orden de cálculo y la √ representa la raíz cuadrada.

El resultado de cada expresión es un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas. Una expresión consta de ***operandos*** y ***operadores***. Según sea el tipo de objetos que manipulan, las expresiones se clasifican en:

- Aritméticas *(Resultado de tipo Numérico)*
- Lógicas (*Resultado de tipo Lógico)*
- Carácter (R*esultado de tipo Carácter)*
- *Cadena de caracteres (resultado de tipo Cadena)*

a) Expresiones aritméticas:

Son análogas a las fórmulas matemáticas. Las variables y constantes son numéricas (entera o real) y las operaciones son las aritméticas.

Operadores	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
**, ^, ↑	Exponenciación
div	división de enteros
mod	resto de la división de enteros

Operadores **div** y **mod**: Generalmente, los lenguajes de programación utilizan dos operadores para la división: el operador "**/**" para la división de números de tipo real y el operador "**div**" para la división de números de tipo entero.

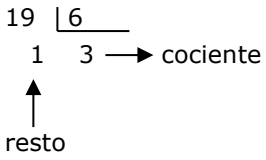
El operador **mod** devuelve el resto de la división como un valor de tipo entero.

En ambos casos, esta representación puede diferir de acuerdo al lenguaje de programación.
Ejemplos: A = 19; B=6

A / B el resultado es 3,166 (real)

A div B el resultado es 3 (entero)

A mod B el resultado es 1 (entero)



C *Operadores aritméticos en C:*

+	Suma
-	Resta
*	Multiplicación
/	División
%	resto de la división de enteros

Aclaración: El lenguaje C no utiliza el operador DIV para la división de enteros.

Reglas de prioridad

Las expresiones que tienen dos o más operandos requieren reglas matemáticas que permitan determinar el orden de las operaciones, se denominan *reglas de prioridad* o *precedencia* y son:

1) Las operaciones encerradas entre paréntesis se evalúan primero. Si existen diferentes paréntesis anidados (interiores unos a otros), las expresiones más internas se evalúan primero.

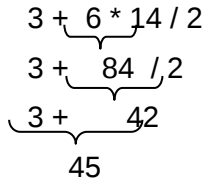
2) Las operaciones aritméticas dentro de una expresión tienen el siguiente orden de prioridad:

   1° - Operador de exponenciación (**, ^, ↑)

2° - Operadores *, /, **div** y **mod**

4° - Operadores +, -

Cuando hay varios operadores con igual prioridad en la expresión, el orden de prioridad es de izquierda a derecha. Ejemplo: Observar el orden de resolución en la siguiente expresión:



Uso de paréntesis: Para cambiar el orden de evaluación de una expresión determinada por su prioridad, se deben utilizar paréntesis. Las expresiones entre paréntesis se evalúan en primer lugar. Si los paréntesis están "anidados", se ejecutan en primer lugar los más internos.
Ejemplo:

(7 * (10 – 5) % 3) * 4 + 9

Se evalúa primero la expresión entre paréntesis, produciendo:
(7 * 5 % 3) * 4 + 9

Luego se evalúa de izquierda a derecha la expresión (7 * 5 % 3):
(35 % 3) * 4 + 9
(2 * 4 + 9)
(8 + 9)
17 Resultado final de la expresión

Operadores de incrementación y decrementación

C incorpora operadores de incremento ++ y de decremento --, estos suman o restan 1 a su argumento, respectivamente, cada vez que se aplican a una variable.

Incrementación	Decrementación
++n	--n
n += 1	n -= 1
n = n + 1	n = n - 1

Ejemplo: `a++` es equivalente a la expresión `a = a + 1`

b) Expresiones lógicas

Una *expresión lógica* es una expresión que puede tomar dos valores: *verdadero* o *falso*.

Las expresiones lógicas se forman combinando constantes lógicas, variables lógicas y otras expresiones lógicas utilizando los *operadores lógi*cos **not**, **and** y **or** y los *operadores relacionales* (de relación o comparación) que se muestran en la tabla 3.

Operadores de relación:
Comprueban una relación entre dos operandos. Se usan normalmente en sentencias de selección (if) o de iteración (while, for), que sirven para comprobar una condición. Los operadores usuales se muestran en la tabla 3. La notación de los mismos puede variar en los distintos lenguajes de programación.

Tabla 3: Operadores de relación

Operadores de relación	Significado	Operadores en C
=	igual	==
>	mayor que	>
<	menor que	<
≥	mayor o igual que	>=
≤	menor o igual que	<=
<>	distinto de	!=

Ejemplo: Dadas las variables A y B, con los valores A=4 y B=3:

- La expresión A > B es *verdadera*
- (A-2) < (B-4) es *falsa*

Si reemplazamos las variables A y B con sus valores tendremos: (4-2) **<** (3-4), esto es 2 < -1, es falso.

Los operadores de relación se pueden aplicar a cualquiera de los tipos de datos estándar: *enteros, real, lógicos* y de *carácter*. Para realizar comparaciones de datos de tipo carácter, la comparación se realizará con el valor numérico decimal de su representación interna (ASCII).

Operadores lógicos:

Los operadores lógicos o booleanos básicos se muestran en la tabla 4:

Tabla 4: Operadores lógicos

Operadores lógicos	Expresión lógica	Significado	**Operadores lógicos en C**
NO (NOT)	**NO** p (**NOT** p)	Negación de p	!
Y (AND)	p **Y** q (p **AND** q)	Conjunción de p y q	&&
O (OR)	p **O** q (p **OR** q)	Disyunción de p y q	||

Las definiciones de las operaciones **NO**, **Y**, **O** se resumen en las llamadas *tablas de verdad.*

A	**NO** A
verdadero	falso
falso	verdadero

A	B	A **Y** B
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	Falso	falso

A	B	A **O** B
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

En las expresiones lógicas se pueden mezclar operadores de relación y lógicos.

Así, por ejemplo:

(11 < 20) **Y** (3 < 9) → la evaluación de la expresión es **verdadera**

(15 > 30) **O** ('X' < 'Z') → la evaluación de la expresión es **verdadera**

Los operadores lógicos se utilizan generalmente en expresiones condicionales. Ejemplos en instrucciones C:

1. Operador AND

```
if ((a > b) && (c > d)) {
   printf ("Resultados correctos");
}
```

Si a = 5, b = 3, c = 12, d = 10, se cumple que las 2 subexpresiones son verdaderas. El operador AND requiere que ambas sean verdaderas para que el resultado sea verdadero. Por tanto, la evaluación es verdadera y se mostrará el mensaje "Resultados correctos".

2. Operador OR

```
if ((a > b) || (c > d)) {
    printf ("Resultados correctos");
}
```

Si a = 5, b = 3, c = 2, d = 10, la primera subexpresión es verdadera y la segunda es falsa. El operador OR requiere que una de las expresiones, o las dos, sean verdaderas para que el resultado sea verdadero. Por tanto, la evaluación es verdadera y se mostrará el mensaje "Resultados correctos".
Nota: la barra vertical es el ASCII alt 124.

3. Operador NOT

```
if (! (a > b) {
   printf ("Resultado correcto");
}
```

Si a = 5, b = 13, la expresión a > b es falsa. El operador NOT aplica la negación, y debe entenderse como "si la expresión no es verdadera entonces…". En este caso, la evaluación es verdadera y se mostrará el mensaje "Resultados correctos".

Prioridad de los operadores lógicos:

Operador	Prioridad
NO (not)	Más alta (primera ejecutada)
Y (**AND**)	↓
O (**OR**)	Más baja (última ejecutada)

c) Expresiones de carácter y cadena

Las expresiones de tipo carácter o de tipo cadena se verán más adelante, en el tema específico.

Características de C vinculadas con el tratamiento de los datos:

a) **Operadores especiales:** C admite algunos operadores especiales que sirven para propósitos diferentes:

- El operador ( ): Operador de llamada a funciones, encierra los argumentos de una función.
- El operador [ ]: Sirve para dimensionar arreglos y designar un elemento de arreglo.

```
int notas [20] /* define un arreglo de 20 elementos.
```

b) **El operador sizeof:** Permite conocer el tamaño en byte de un tipo de dato o variable.
`sizeof (unsigned int)` devuelve un valor entero que es la cantidad de bytes que asigna esa arquitectura particular al tipo de dato ingresado.

c) **Conversión de tipos:** Con frecuencia se necesita convertir un tipo de dato a otro, sin cambiar el valor que representa. Las conversiones de tipo pueden ser *implícitas* (ejecutadas automáticamente) o *explicitas* (solicitadas específicamente por el programador).

c1) Conversión implícita: Los operandos de tipo más bajo se convierten en los de tipo más alto:
Ejemplos:

```
int k = 12;
float m = 4;
m = m + k;  /* el valor de k se convierte temporalmente en float antes de la suma */
m = k / 5;  /* realiza la división entera k/5, el resultado es 2 y se asigna a m */
m = 4.0;
m = m / 5;  /* Convierte 5 a float, realiza la división y el resultado asigna a m */
```

c2) Conversión explícita: C fuerza la conversión de tipo mediante el operador *cast* para realizar la operación. El dato original no cambia.
El formato es: **`(tipo dato) variable`**
Ejemplos:

```
(int)m;      /* Convierte a los efectos de la operación el valor de m a entero */
(float)k;    /* Convierte a los efectos de la operación el valor de k en real de simple
                precisión */
m = (float)k/5;  /* realiza la división k / 5, convierte el resultado a float y asigna a m */
```

En la última operación, si **k** es 12 el valor que se almacena es **m** es 2.4.

Representación de información

1. Código de representación de TEXTO

La computadora utiliza codificación de caracteres de texto para almacenar los datos alfanuméricos, tales como, nombre y apellido, dirección, etc. Cuando se presiona una tecla cualquiera en la computadora, ésta convierte el impulso eléctrico que proviene del teclado en un conjunto de bits. Para realizar esta transformación se utilizaron y se utilizan distintos códigos. Los dos más utilizados en la actualidad son:

1.1. ASCII: Código Estándar Americano para el Intercambio de Información

El código ASCII básico utiliza 7 bits para representar los caracteres. Como otros códigos de formato de representación de caracteres, es un método para una correspondencia entre cadenas de bits y una serie de símbolos (alfanuméricos y otros), permitiendo de esta forma la comunicación entre dispositivos digitales. El código de caracteres ASCII, o una extensión compatible, se usa casi en todas las computadoras.

Existen otras versiones ampliadas de este código que utilizan 8 bits y respetan los códigos normalizados del ASCII básico, aprovechando las combinaciones no usadas para representar símbolos adicionales. Entre ellas se encuentran los códigos ISO 8859-n, donde n es el número que identifica el juego de los nuevos caracteres introducidos dependiendo de los lenguajes. Por ejemplo, la norma, ISO 8859-1, también denominada ISO-Latin1, se proyectó para América y Europa occidental e incluye vocales con acentos, tildes, diéresis y otras letras latinas no usadas en los países anglosajones.

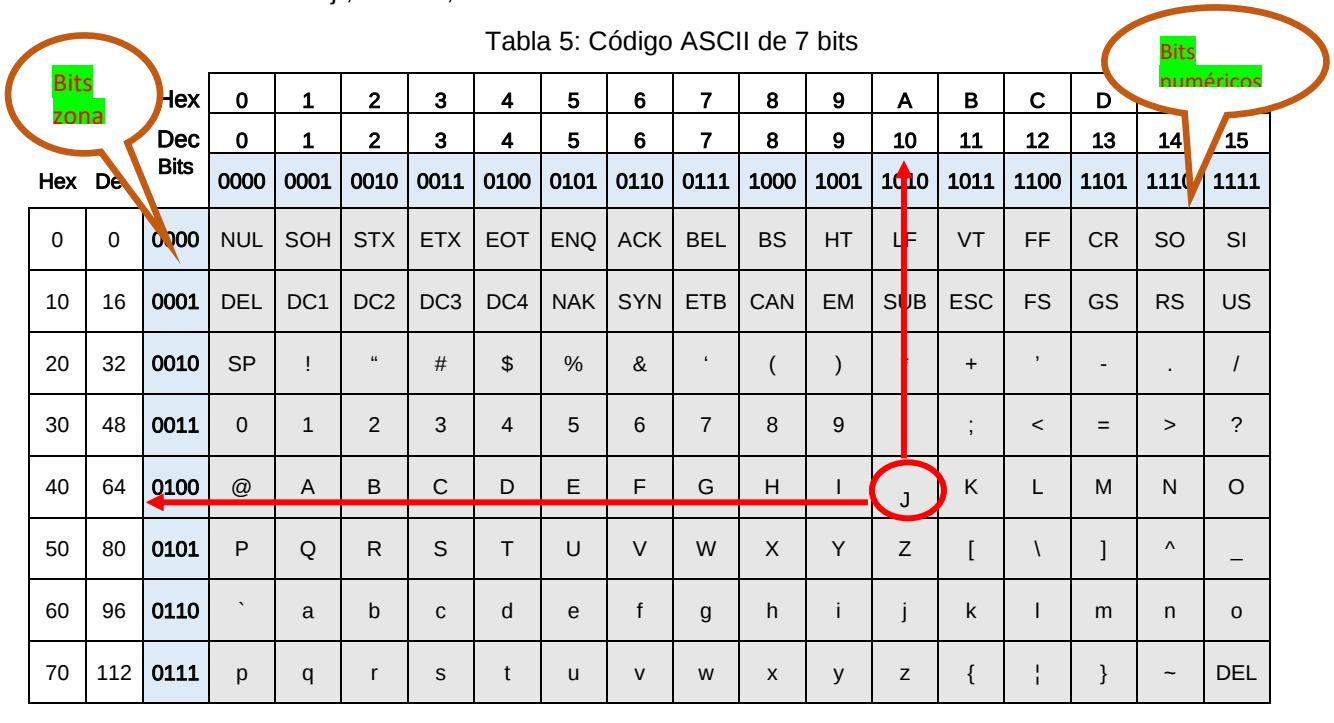
El código ASCII reserva los primeros 32 códigos (numerados del 0 al 31 en decimal) para **caracteres de control**: códigos pensados para controlar los dispositivos. Por el avance tecnológico la mayoría de ellos ya no se utiliza, excepto LF (decimal 10) representa la función "nueva línea" (*line feed*) que hace que una impresora avance el papel, el CR que indica el “retorno de carro” (tecla *Enter*), y el símbolo ESC (decimal 27) representa la tecla "Escape" que se utiliza para cancelar acciones.

Los códigos del 33 al 126 se conocen como **caracteres imprimibles**, y representan letras, dígitos, signos de puntuación y varios símbolos. El código 32 el carácter espacio, designa al espacio entre palabras, y se produce cuando se oprime la barra espaciadora de un teclado.

En la tabla 5 podemos ver que cada carácter tiene una representación interna dada por un conjunto determinado de bits, que también pueden expresarse en decimal y hexadecimal, siguiendo las reglas de conversión entre sistemas. Por ejemplo: al carácter ‘**J**’ le corresponde el código ASCII decimal **74** que en hexadecimal se representa como **4A**, y en binario como **01001010**, como se puede comprobar en la tabla.

El decimal 74 se obtiene sumando los valores decimales de la intersección de la fila y la columna en la que se encuentra el carácter “j”, esto es, al decimal 64 de la fila 5 se le suma el decimal 10 de la columna 11.

Tabla 5: Código ASCII de 7 bits



Hex	Dec	Bits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	0	0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	16	0001	DEL	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	32	0010	SP	!	“	#	$	%	&	‘	(	)	*	+	’	-	.	/
30	48	0011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	64	0100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	80	0101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	96	0110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	112	0111	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	DEL

Caracteres ASCII Extendido:

El código ASCII de 7 bits contiene 128 caracteres y contiene todos lo necesarios para escribir en idioma inglés. En 1986, se modificó el estándar para agregar nuevos caracteres latinos, necesarios para la escritura de textos en otros idiomas, como por ejemplo el español, así fue como se agregaron los caracteres que van del ASCII 128 al 255, denominados ASCII Extendido.

En la tabla 6 se puede ver que las letras propias del español, como la ñ, se codifican como decimal 164 y 165, respectivamente.

Las letras acentuadas como la á minúscula se representa con el decimal 160, la í es 161, la ó es 162 y la ú es 163.

Tabla 6: Código ASCII extendido de 8 bits

128	Ç	144	É	160	á	176	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	161	í	177	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	162	ó	178	▓	195	├	211	╙	227	π	243	≤
131	â	147	ô	163	ú	179	|	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	164	ñ	180	┤	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	165	Ñ	181	╡	198	╞	214	╓	230	µ	246	÷
134	å	150	û	166	ª	182	╢	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	167	º	183	╖	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	168	¿	184	╕	201	╔	217	┘	233	⊛	249	·
137	ë	153	Ö	169	_	185	╣	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	170	¬	186	║	203	╦	219	▉	235	δ	251	√
139	ï	156	£	171	½	187	╗	204	╠	220	▄	236	∞	252	_
140	î	157	¥	172	¼	188	╝	205	═	221	▌	237	ϕ	253	²
141	ì	158	_	173	¡	189	╜	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	174	«	190	╛	207	╧	223	▀	239	∩	255	
143	Å	192	└	175	»	191	┐	208	╨	224	α	240	≡		

Ingreso de caracteres por teclado:

No todos los símbolos que necesitamos utilizar se encuentran disponibles en el teclado, o el teclado tiene una configuración diferente. En este caso, se puede ingresar códigos ASCII utilizando su valor decimal, de la siguiente manera:

- Para ingresar el símbolo corchete "[": Presionar la tecla "Alt" en el teclado y el número "91" en el teclado numérico (sin dejar de presionar Alt).
- Para ingresar la letra "Ñ", presionar la tecla ALT y el número "165" en el teclado numérico.

1.2. UNICODE

Este código fue propuesto por un consorcio de empresas y entidades con el objetivo de representar texto de muy diversas culturas.

Los códigos anteriores presentan varios inconvenientes, tales como:

- Los símbolos son insuficientes para representar los caracteres especiales que requieren numerosas aplicaciones.
- Los símbolos y códigos añadidos en las versiones ampliadas a 8 bits no están normalizados.
- Los lenguajes escritos de diversas culturas orientales, como la china, japonesa y coreana se basan en la utilización de ideogramas o símbolos que representan palabras, frases o ideas completas, siendo, por tanto, inoperantes los códigos que sólo codifican letras individuales.

Unicode está reconocido como estándar **ISO/IEC 10646**, y presenta las siguientes propiedades:

- *Universalidad*, trata de cubrir la mayoría de los lenguajes escritos: 16 bits → 65.536 símbolos.

- *Unicidad*, a cada carácter se le asigna exactamente un único código.
- *Uniformidad*, ya que todos los símbolos se representan con un número mínimo de bits (16).

Unicode trata los caracteres alfabéticos, ideográficos y símbolos de forma equivalente, lo que significa que se pueden mezclar en un mismo texto sin la introducción de marcas o caracteres de control. La tabla 7 muestra un esquema de cómo se han asignado los códigos Unicode.

Tabla 7: Esquema de asignación de códigos en Unicode

Zona	Códigos		Símbolos codificados	N° de caracteres
A	0000	0000 00FF	Código ASCII Latín-1	256
			Otros alfabetos	7.936
		2000	Símbolos generales y caracteres fonéticos chinos, japoneses y coreanos	8.192
I	4000		Ideogramas	24.576
O	A000		Pendiente de asignación	16.384
R	E000 FFFF		Caracteres locales y propios de los usuarios Compatibilidad con otros códigos	8.192

UTF-8 (8-bit Unicode Transformation Format) es un formato de codificación de caracteres Unicode e ISO 10646 utilizando símbolos de longitud variable. Se usa mucho en la web debido a que contiene todos los caracteres necesarios para la representación HTML del código de una página.

Video: Representación digital de textos - Alberto Prieto Espinosa
https://www.youtube.com/watch?v=86f47DCTMms&t=488s

2. Representación de Números

Sistemas de representación restringidos a n bits

Las computadoras disponen de registros y buses de tamaños específicos que limitan la cantidad de bits disponibles para la representación de los datos. Es habitual mencionar que el sistema trabaja con un número finito de **n** bits**,** denominado "**palabra".** La palabra puede ser de 8, 16, 32 bits. Por lo tanto, **las representaciones que se utilizan tienen limitaciones**, y los **cálculos están siempre sujetos a aproximaciones** y por ende **a errores**.

Para caracterizar los sistemas de representación y compararlos, se definen dos parámetros importantes:

Capacidad de representación:
Es la cantidad de combinaciones distintas que se pueden representar. Por ejemplo, si el sistema está restringido a 5 bits, son 2^5 combinaciones de unos y ceros que permiten representar 32 elementos diferentes.

Rango:
El rango de un sistema está dado por el número mínimo y el número máximo representables. Por ejemplo, en binario sin signo con cinco dígitos el rango es [0, 31], donde "0" es el menor valor a representar y "31" el máximo.

Magnitud y signo: La notación habitual que se usa para representar cantidades numéricas con signo consiste en añadir un símbolo adicional para diferenciar un número positivo de uno negativo. Ej. 5 y -5.
En una computadora, la traducción directa de esta notación consiste en considerar a uno de los bits del número como bit de signo, por ejemplo, señalando a un número como positivo con un bit 0 y a un número negativo con un bit 1. Ej. 0111 (7) y 1111 (-7). Por tanto, la representación tiene: bit de signo y la magnitud del número.

En general los números utilizados por la computadora se dividen en dos grandes grupos: los **enteros**[1] y los **reales**. Para los números reales se utiliza el método de *coma flotante*.

[1] Los números enteros comprenden a los números naturales, sus opuestos (negativos) y el cero.

2.1. Representación de enteros

Existen diferentes formas para representar un número entero en una computadora. Hay que tener en cuenta que almacenar un número entero requiere que no sólo se almacenen los dígitos sino también el signo, o por lo menos que se utilice una forma de almacenamiento que permita reconocer los enteros positivos y los negativos. Algunos de los métodos de representación de enteros son: Módulo y signo, Complemento a 1, Complemento a 2 y Exceso a 2 elevado a la N-1.

En esta asignatura, se comentan con detalle, solo 2 de ellos:

2.1.1. Complemento a 2 (C-2)

Este sistema de representación utiliza el bit de la izquierda para el signo, correspondiendo el 0 para el positivo y el 1 para el negativo. Para los números positivos el resto de los bits (N-1) representan el módulo del número. El negativo de un número positivo se obtiene en dos pasos:

1°) Se complementa el módulo del número positivo en todos sus bits (cambiando ceros por unos y viceversa) incluido el bit de signo. Es decir, se realiza el "complemento a 1".

2°) Al resultado obtenido en el primer paso se le suma 1 (en binario) **despreciando el último acarreo si existe**. Ejemplo:

Número 10		0	0	0	0	1	0	1	0
complemento		**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**
	+								**1**
Número -10	~~1~~	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**

Un caso particular es el número -128, no puede ser representado siguiendo este método, y se le asigna la representación por convención.

El rango de representación es asimétrico, lo que constituye su mayor inconveniente y viene dado por:

$$-2^{N-1} <= x <= 2^{N-1} -1$$

Se puede observar que el término de la izquierda no se suma el 1 dado que no existe el cero negativo.

Los rangos de valores para tamaños de palabra de hasta 4 byte son:

1 byte (8 bits)	**-128**	**<= X <=**	**+127**
2 byte (16 bits)	**-32768**	**<= X <=**	**+32767**
4 byte (32 bits)	**-2147483648**	**<= X <=**	**+2147483647**

La principal ventaja es la de tener una única representación del cero:

En el caso de palabras de 8 bits tendríamos:

Número 0		0 0 0 0 0 0 0 0
Número -0	*Primer paso*	1 1 1 1 1 1 1 1
	Segundo paso	1 1 1 1 1 1 1 1
	+	1
		1 0 0 0 0 0 0 0 0

El último acarreo se desprecia, por lo tanto, el 0 y el -0 tienen una sola representación.

2.1.2. Exceso a 2 elevado a N-1

Este método no utiliza ningún bit para el signo, todos los bits se utilizan para representar el valor del número. La idea es sumarles a todos los valores *un off-set* o *corrimiento*, excediendo el valor cero por $2^{(n-1)}$, que se denomina “exceso”, donde **n** es la cantidad de bits que se utilizará para almacenar el número. El número entero en exceso $N_{(E)}$ = **Exceso** + **N**, donde Exceso = $2^{(n-1)}$ y N es el número a representar.

La idea del desplazamiento o «exceso» es que todos los bits 0 (valor 0) correspondan al máximo número negativo y todos los bits 1 (valor 255) al máximo número positivo

Número -128 → (-128 + 128) = 0: **0 0 0 0 0 0 0 0**

Número 127 → (127 + 128) = 255: **1 1 1 1 1 1 1 1**

Ejemplo: Para representar el número 10 con n = 8 bits el exceso es de 2^7 = 128, con lo que el número 10 se representa como 10 + 128 = 138; para el caso de -10 tendremos -10 + 128 = 118.

Número 10: 10001010

Número -10: 01110110

en este caso, el 0 tiene una única representación, que para 8 bits corresponde a:

Número 0 (0 + 128): 10000000

El rango de representación es asimétrico y viene dado por: $-2^{N-1} <= X <= 2^{N-1} -1$

1 byte (8 bits)	**-128**	**<= X <=**	**+127**
2 byte (16 bits)	**-32768**	**<= X <=**	**+32767**
4 byte (32 bits)	**-2147483648**	**<= X <=**	**+2147483647**

Desbordamiento (Overflow)

Si tenemos 8 bits para representar un entero, tendremos un rango de representación de 0 a 255. ¿Qué sucede si se quiere representar el valor 256? NO SE PUEDE, se requieren 9 bits.

Se conoce como Overflow (desbordamiento) a la condición que se produce al sumar dos números a través de un determinado método de representación y una determinada cantidad de bits, cuyo resultado no puede ser representado utilizando la misma cantidad de bits, obteniéndose un resultado erróneo.

Sucede principalmente cuando se realiza una suma, y el resultado obtenido es mayor al máximo número representable en el sistema.

Ejemplo de overflow:

Decimal	Operación en módulo y signo	Decodificación
52	00110100	52
+ 97	+ 01100001	+ 97
149	10010101	-21

Dado que la suma supera el valor máximo de 127 para este método de representación en 8 bits, la ALU hace la suma colocando en el bit de signo una parte del resultado, produciendo un resultado inválido. La ALU posee una bandera o *flag* para indicar este tipo de errores.

Conclusiones sobre los sistemas de representación de enteros

El método más utilizado en la actualidad para la representación de enteros es el **Complemento a 2**, ello se debe a la facilidad de efectuar las sumas y restas con esta representación, porque en todos los casos, las operaciones se resuelven con sumas. Este método reduce la complejidad de los circuitos de la unidad aritmética lógica, dado que no son necesarios circuitos específicos para restar.

En la tabla 8 se muestran los límites aproximados de valores enteros representables con distintas longitudes de palabras:

Tabla 8: Límites o rangos de representación para distintas longitudes de palabras

Longitud de palabra	Límite superior N (max)	Límite inferior N (min)	
		Complemento a 1	Complemento a 2
8	127	-127	-128
16	32.767	-32.767	-32.768
32	2.147.483.649	-2.147.483.649	-2.147.483.650
64	$9{,}223372 * 10^{18}$	$-9{,}223372 * 10^{18}$	$-9{,}223372 * 10^{18} -1$

Si como resultado de las operaciones, se obtiene un número fuera de los límites o rango, se dice que se ha producido un **desbordamiento u overflow**. La figura 1 muestra gráficamente el rango de representación de los números enteros.

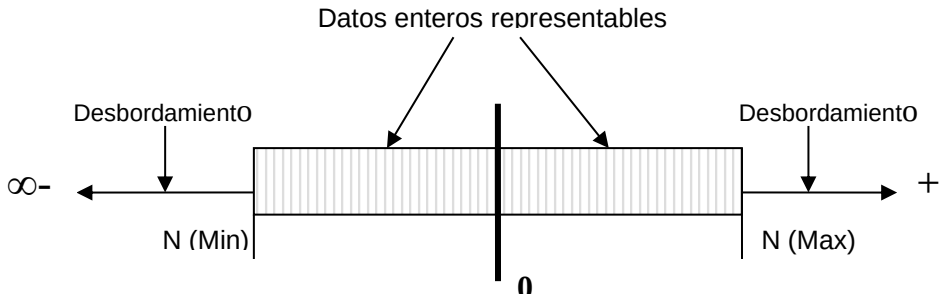


Figura 1: Rango de posibles representaciones de enteros

Video: Representación digital de números enteros - Alberto Prieto Espinosa
https://www.youtube.com/watch?v=pSPuGiQ0vdE&t=368s

2.2. Coma Flotante (Números Reales)

La representación de coma flotante es una forma de notación científica usada en los procesadores (CPU) y otros componentes de procesamiento, con la cual se pueden representar números reales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas. El estándar para la representación en coma flotante es el IEEE 754.

Notación científica

La notación científica es un recurso matemático empleado para simplificar cálculos y representar en forma concisa números muy grandes o muy pequeños. Para hacerlo se usan potencias de la base del sistema
En notación científica estándar, los números se expresan de la forma:

$$N = +- m\ E^{+-p} = +- m * b^{+- p}$$

Donde **m** es un número real**, b** es la base del sistema y **p** es un número entero, cuyo signo indica si la coma se desplaza a la derecha **(+)** o a la izquierda **(-).**

Ejemplos:

a) $-246{,}36_{10} = -2{,}4636\ E+2 = -2{,}4636 * 10^{2}$

b) $82000000000_{10} = 8{,}2\ E+10 = 8{,}2 * 10^{10}$

c) $0{,}00003 = 3{,}0\ E-5 = 3 * 10^{-5}$

d) $-1010000_{2} = -1{,}01\ E+6 = -1{,}01 * 2^{6}$

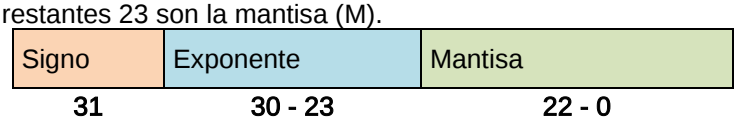
e) $0{,}0000012 = 1{,}0\ E-6 = 1{,}0 * 2^{-6}$

Normalización IEEE 754

Existen muchas maneras de representar números en formato de punto flotante. Cada uno tiene características propias en términos de rango, precisión y cantidad de elementos que pueden representarse. En un esfuerzo por mejorar la portabilidad de los programas y asegurar la uniformidad en la exactitud de las operaciones en este formato, el IEEE (Instituto de Ingeniería Eléctrica y Electrónica de Estados Unidos) creó un estándar que especifica cómo deben representarse los números en coma flotante con simple precisión (32 bits) o doble precisión (64 bits), y también cómo deben realizarse las operaciones aritméticas con ellos.

Precisiones usuales en IEEE754:

a) ***Simple Precisión (32 bits).*** El primer bit es el bit de signo (S), los siguientes 8 son los bits del exponente (E) y los restantes 23 son la mantisa (M).



Exponente: Se destinan 8 bits para el exponente. Se utiliza la representación de entero en Exceso. El exceso se suma al exponente original, y el resultado es el que se almacena: E = EO + E

Mantisa: Utiliza 23 bits. Dado que la normalización implica un bit implícito, son 24 bits efectivos. La normalización toma la forma **1, bb..b**, donde **bb..b** representa los 23 bits de la mantisa que se almacenan. Cuando la mantisa se **normaliza** situando la coma decimal a la derecha del bit más significativo, dicho bit siempre vale **1**. Por tanto, se puede prescindir de él (bit implícito), y tomar en su lugar un bit más de la mantisa para aumentar la precisión del número.

b) ***Doble precisión (64 bits).*** El primer bit es el bit de signo (S), los siguientes 11 son los bits del exponente (E) y los restantes 52 son la mantisa (M).

Signo	Exponente	Mantisa
63	**62 - 52**	**51 - 0**

Los mecanismos de representación del exponente y la mantisa son similares al descripto en precisión simple. *Valores límites:* Con toda representación se obtienen unos valores máximos y mínimos representables que para precisión simple son:

Número Mayor (N max)	$3{,}402823466.\ 10^{38}$
Número menor normalizado (N min, nor)	$1{,}2.\ 10^{-38}$
Número menor denormalizado (N min, den)	$1{,}1401.\ 10^{-45}$

En la figura 2 se muestra el rango de representación de los números reales. Obsérvese que los números reales que cumplen las siguientes condiciones no pueden ser representados:

- Los números comprendidos entre –N(min,den) y N(Min,den) con N distinto de cero. Si como resultado de una operación el número N está incluido en esa zona, se dice que se produce un **agotamiento** (*underflow*).
- Los números menores que –N(max) y mayores que N(Max) con N distinto de infinito positivo o negativo. Si como resultado de una operación el número N cae en esa zona, se dice que se produce un **desbordamiento** *(overflow).*

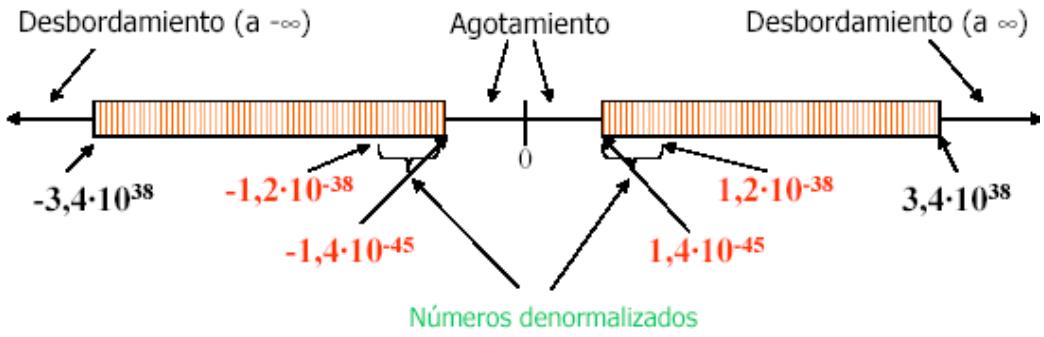


Figura 2: Rango de posibles representaciones de números reales

Observaciones finales

En importante que el programador tenga en cuenta cómo se almacenan los números en la computadora, ya que se pueden presentar problemas inherentes a la forma en que se representan los números (con un número limitado de bits).

Dificultades:

- Por la obtención, en resultados intermedios, de **números excesivamente pequeños.** Esto puede ocurrir por restar dos números extremadamente próximos entre sí o por la división entre números en los que el divisor es mucho mayor que el dividendo. En estos casos puede perderse la precisión de los cálculos o producirse un **agotamiento**.
- Por la obtención de **resultados numéricos excesivamente altos**, es decir por **desbordamiento**. Esto ocurre, por ejemplo, al dividir un número por otro mucho menor que él o al efectuar sumas o productos sucesivos con números muy elevados.

- En la comparación de dos números. Hay que tener en cuenta que cada dato real en la computadora representa a infinitos números reales (un intervalo de la recta real), por lo que en general una mantisa decimal no puede representarse exactamente con *n* bits, con lo que genera un error **"de representación".**
- Esto da lugar a problemas al comparar si un número es igual a otro (sobre todo si estos números se han obtenido por cálculos o procedimientos distintos), ya que la computadora considera que dos números son iguales únicamente si son iguales todos sus bits. La detección de igualdad se debe hacer con números enteros o considerando que dos números son iguales si la diferencia entre ellos es menor que un valor dado.
- Una consecuencia de lo dicho anteriormente es que, la suma y multiplicación de datos de tipo real no siempre cumplen las propiedades asociativas y distributivas, se pueden obtener resultados distintos dependiendo del orden en que se realizan las operaciones.

Video: Representación digital de números reales - Alberto Prieto Espinosa
https://www.youtube.com/watch?v=L2YUAIWXlco&t=38s