

Tema 3. Estructura y componentes de un programa

1. Concepto de programa

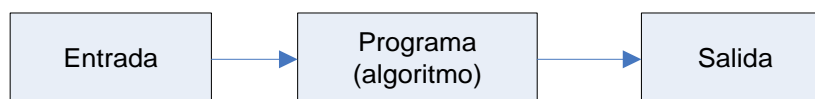
Se llama “programa” al conjunto de instrucciones (órdenes) que se dan a la computadora para la ejecución de una tarea determinada.

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 el desarrollo tradicional, se sigue el denominado “ciclo de vida”, que comprende las siguientes actividades: a) Análisis del problema, b) Diseño del algoritmo de solución, c) Codificación, d) Compilación y ejecución, e) Verificación y depuración, f) Documentación, g) Mantenimiento. En esta asignatura se propone un método de resolución de problemas que propone: definir el objetivo, especificar la estrategia de solución, descomponer la solución en tareas con nombres representativos, y verificar la solución con casos de pruebas predefinidos.

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.

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, representado en pseudocódigo, podemos ver un algoritmo sencillo que lee datos de alumnos: N° 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 pseudocódigo 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. En este ejemplo, a la derecha se muestran:

- Por un lado, los datos de ENTRADA son los “casos de prueba” definidos, que representan a tres alumnos y sus respectivas notas. Los valores de la última fila son datos ficticios creados intencionalmente para generar la condición de fin de programa.
- Por otra parte, se muestra el contenido de cada variable almacenada en la memoria RAM de la computadora, después de cada iteración del proceso.
- 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

RAM – Lectura alumno 1		
345	8	5
LU	N1	N2
6,50		
Promedio		

RAM – Lectura alumno 2		
156	3	7
LU	N1	N2
5		
Promedio		

RAM – Lectura alumno 3		
678	9	10
LU	N1	N2
9,50		
Promedio		

RAM – Lectura alumno 4		
000	0	0
LU	N1	N2
9,50		
Promedio		

Es importante comprender los siguientes conceptos:

- Las **variables** indicadas en un programa son **posiciones de memorias** (celdas) que internamente serán reconocidas con una “**dirección**” (un número binario).
- 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.
- 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.
- La instrucción de repetición de un bucle (**MIENTRAS**) repite el conjunto de instrucciones hasta que N1=0 (bifurcación condicional) que posibilita cambiar la linealidad de ejecución de las instrucciones, en este caso, para terminar el programa.
- 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.
- La instrucción de salida muestra, en el dispositivo indicado (en este caso una impresora), 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**, que consiste en ejecutar el programa con los casos de prueba definidos y para comprobar los resultados.

El mismo algoritmo, codificado en C, se muestra más abajo. Pueden realizar la verificación con los mismos casos de prueba.

```
#include <stdio.h>
/* Declaración de prototipos de funciones */
void ingresarDatos();
void calcularPromedio();
/* Declaración de variables globales */
int lu;
float nota1, nota2, promedio;

/* Programa principal*/
int main() {
    ingresarDatos();
    while(nota1 != 0) {
        calcularPromedio();
        ingresarDatos();
    }
    return 0;
}

/* Implementación de funciones */
void ingresarDatos() {
    printf("\nIngrese nro de LU: ");
    scanf("%d", &lu);
    printf("Ingrese nota 1: ");
    scanf("%f", &nota1);
    printf("Ingrese nota 2: ");
    scanf("%f", &nota2);
}

void calcularPromedio() {
    promedio = (nota1 + nota2)/2;
    printf("\nLU: %d\n", lu);
    printf("El promedio de notas es: %.2f\n", promedio);
}
```

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, apóstrofes, 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. Es usual que se establezcan determinadas convenciones para conformar los identificadores, buscan mejorar la legibilidad del código.

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 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

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 pierden 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”. Para esto hay que incluir la librería `<stdbool.h>`

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 código ASCII (código que representa caracteres dentro de la computadora, que veremos más adelante) y el orden de los caracteres es el que le asigna este código.

Un valor constante de tipo carácter se escribe entre apóstrofes, 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)**.

Constantes y variables

Constantes

Los programas muchas veces requieren valores que no deben cambiar durante la ejecución del programa. Estos valores se denominan *constantas*. 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 apóstrofes.

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



En C, las constantes de tipo **char** se encierran entre apóstrofes 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 pseudocódigo:

VAR

REAL: precio_unitario, precio_final

ENTERO: cantidad

STRING: domicilio [30]

CHAR: codigo

El mismo ejemplo, en C:

float precioUnitario; precioFinal;

int cantidad;

char 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;

3.2. 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: $a + (b+3) + \sqrt{C}$. Los paréntesis indican el orden de cálculo y la $\sqrt{}$ 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 (*Resultado de tipo Carácter*)
- Cadena de caracteres (*resultado de tipo Cadena*)

a) Expresiones aritméticas:

Son análogas a las fórmulas matemáticas. Las variables y constantes (operandos) son numéricas (entera o real) y las operaciones (operadores) son las aritméticas. En la tabla 3 se muestran los operadores aritméticos usuales y en la tabla 4 los operadores aritméticos en C.

Tabla 3: Operadores aritméticos

Operadores aritméticos	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 con el 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)

19 | 6
 1 3 → cociente
 ↑
 resto



Tabla 4: 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:

$$\begin{array}{c} 3 + \underbrace{6 * 14 / 2} \\ 3 + \underbrace{84 / 2} \\ \underbrace{3 + 42} \\ 45 \end{array}$$

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 $(7 * 5 \% 3)$:

$$(35 \% 3) * 4 + 9$$

$$2 * 4 + 9$$

$$8 + 9 = 17 \rightarrow \text{resultado final de la expresión}$$

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.

Ejemplos:

a++ es equivalente a la expresión $a = a + 1$

n-- es equivalente a la expresión $n = n - 1$

El operador puede colocarse antes o después de la variable: ++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 lógicos* **not**, **and** y **or** y los *operadores relacionales* (de relación o comparación) que se muestran en la tabla.

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 5. La notación de estos puede variar en los distintos lenguajes de programación.

Tabla 5: 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 son **NO (NOT)**, **Y (AND)** y **O (OR)**. El funcionamiento de estos operadores se muestra en la tabla 6.

Tabla 6: 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	

En la tabla 7 se muestran las definiciones de las operaciones **NO**, **Y**, **O**, llamadas *tablas de verdad*.

Tabla 7: Tablas de verdad para cada operador lógico

A	NO A	A	B	A Y B	A	B	A O B
verdadero	falso	verdadero	verdadero	verdadero	verdadero	verdadero	verdadero
falso	verdadero	verdadero	falso	falso	verdadero	falso	verdadero
		falso	verdadero	falso	falso	verdadero	verdadero
		falso	Falso	falso	falso	falso	falso

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

Así, por ejemplo:

$(11 < 20) \text{ Y } (3 < 9) \rightarrow$ la evaluación de la expresión es **verdadera**

$(15 > 30) \text{ O } ('X' < 'Z') \rightarrow$ la evaluación de la expresión es **verdadera**

Los operadores lógicos se utilizan generalmente en expresiones condicionales.

C 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");
}
```

De la serie "Se encuentra un hombre normal a un profesor de lógica proposicional dentro de un ascensor"

¿Sube o baja?



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 altas (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 detalle.



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. Ej.

```
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 *explícitas* (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 = 16 */
```

```
m = k / 5; /* realiza la división entera k/5, el resultado es m = 2.0 */
```

```
m = 4;
```

```
m = m / 5; /* Convierte 5 a float, realiza la división y el resultado es m = 0,8 */
```

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 float */
```

```
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 en m es 2.4.