

CAPITULO 2

METODOLOGÍA DE LA PROGRAMACIÓN

2.1 Introducción. Objetivos.

Este segundo capítulo tiene como finalidad principal introducirnos en el conocimiento de la estructura general de un programa, el uso de las herramientas de representación de algoritmos y las técnicas básicas de programación.

Los objetivos principales serán:

- § Conocer el concepto de información, datos y sus sistemas de representación y operación.
- § Conocer la metodología de la programación.
- § Dominar los métodos y técnicas para el desarrollo de programas.
- § Usar herramientas de diseño de algoritmos
- § Ejercitar la interpretación de problemas y sus algoritmos que los resuelvan.

2.2 La información y su representación.

La información, como comentamos en el capítulo anterior, surge como resultado de un proceso de datos. Es algo que se fabrica a partir de una materia prima, en este caso los datos. No surge de forma espontánea, sino como resultado de una elaboración o proceso.

El fin de la información es aumentar el conocimiento de las personas y por ende la capacidad de decidir de forma adecuada. Por eso se dice que la información es poder. Quien tiene información, tiene la posibilidad de elegir, y por lo tanto de actuar según su interés.

Las personas para registrar los datos y las informaciones, utilizamos unos sistemas de representación compuestos de una serie de símbolos o códigos adecuados. Estos sistemas de representación han ido evolucionando desde los más primitivos (símbolos gráficos) hasta los utilizados actualmente (alfabetos, números, etc.).

Los ordenadores actuales, para tratar los datos y las informaciones, usan unos sistemas de representación que no son los que normalmente utilizamos las personas. Los dispositivos de entrada y salida son los encargados de traducir los datos e informaciones entre los sistemas de representación de las personas y los que necesitan los ordenadores.

2.2.1 Sistemas de numeración.

Se define un sistema de numeración como un conjunto de símbolos y reglas utilizados para representar cantidades. Los más usados son los sistemas de numeración posicionales, aquellos en los que el valor que representa cada símbolo depende de su valor absoluto y de la posición que ocupa dicho símbolo.

Un sistema de numeración se diferencia de otro en el número de símbolos usados para representar una cantidad. Es lo que se denomina base. Así el sistema de numeración base 10 o decimal usa 10 símbolos distintos.

El **teorema fundamental de la numeración** nos permite representar una cantidad en cualquier sistema de numeración mediante sumas de potencias de su base. La fórmula es:

$$\dots + X_3B^3 + X_2B^2 + X_1B^1 + X_0B^0 + X_{-1}B^{-1} + X_{-2}B^{-2} + X_{-3}B^{-3} + \dots$$

Donde X_i representa cada una de las cifras con la condición $0 \leq X_i < B$, siendo B la base de numeración.

Como veremos posteriormente, este teorema nos permitirá, a partir de una cantidad expresada en cualquier sistema de numeración, calcular su equivalente en el sistema decimal.

2.2.2. Sistema decimal.

El sistema decimal utiliza 10 símbolos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) para expresar cantidades. Cada uno de los símbolos se denomina **dígito decimal**.

Como dice el TFN (Teorema Fundamental de la Numeración), cualquier cantidad, expresada en este caso en base diez, podemos descomponerla como sumas de potencias de diez. Por ejemplo:

$$12,34 = 1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

podemos comprobarlo si calculamos la expresión anterior

$$12,34 = 1 \cdot 10 + 2 \cdot 1 + 3 \cdot 0,1 + 4 \cdot 0,01 = 10 + 2 + 0,3 + 0,04 = 12,34$$

2.2.3 Sistema binario.

El sistema de numeración binario es un sistema de numeración base 2, donde usa dos símbolos (0, 1) para expresar cantidades. Cada uno de los símbolos se denomina **dígito binario**. En términos informáticos se denomina también **bit**, palabra proveniente de la contracción inglesa *binary digit*.

Una cantidad expresada en binario pudiera ser 1010. Vemos que sólo se usan dos símbolos: el cero (0) y el uno (1).

Las computadoras actuales usan este sistema de numeración para representar cantidades. El motivo es que los ordenadores usan dispositivos electrónicos digitales. La electrónica digital se caracteriza en que los componentes electrónicos sólo pueden manejar señales eléctricas con dos valores de tensión o potencial. Por ejemplo un transistor puede tener tensión eléctrica, (se dice que está en abierto) o no tener tensión (cerrado). Esto nos permite asociar a cada valor de tensión eléctrica un valor lógico representado por un dígito binario.

Por ejemplo, si un transistor está en abierto decimos que representa un 1, si el transistor está en corto o cerrado representa un 0. De esta forma una cantidad expresada en binario, como podría ser la anterior 1010, se podría representar o almacenar con cuatro transistores donde sus estados fueran abierto, cerrado, abierto, cerrado. (Realmente las cosas son un poco más complicadas...).

La capacidad de memoria de un ordenador expresa la cantidad de dígitos binarios que puede almacenar. Así si un ordenador pudiera almacenar 45 dígitos binarios (bits), se diría que su capacidad de memoria es de 45 bits. Vemos que se utiliza la palabra bit para dos cuestiones diferentes: como cada uno de los dígitos de una cantidad expresada en binario, y como la cantidad de memoria necesaria para almacenar un dígito binario.

Al conjunto de 8 bits, se denomina **byte** u **octeto (B)**. Esta unidad es la más utilizada para medir la cantidad de información. Un byte es la cantidad de memoria necesaria para almacenar un carácter en los sistemas de representación ASCII o EBCDIC (los usados actualmente para representar caracteres). Las memorias actuales permiten almacenar grandes cantidades de información, por lo que para expresar sus capacidades de almacenamiento se usan prefijos o múltiplos del byte.

| | |
|-----------|---|
| K = Kilo. | Expresa $1024 = 2^{10}$ |
| M = Mega. | Expresa $1048576 = 1024 \cdot 1024 = 2^{10} \cdot 2^{10} = 2^{20}$ |
| G = Giga. | Expresa $1073741824 = 1024 \cdot 1024 \cdot 1024 = 2^{10} \cdot 2^{10} \cdot 2^{10} = 2^{30}$ |
| T = Tera. | Expresa 2^{40} |

Así, un disco con capacidad de 40 GB, podrá almacenar un poco más de 40 billones de caracteres.

complemento a 1 y luego sumando uno a este resultado. En el cálculo de complementos es necesario considerar el número de bytes utilizados para representar las cantidades. En los ejemplos siguientes supondremos un byte para almacenar número enteros.

Ejemplo. Calcular los complementos a 1 y a 2 del número binario 11001.

El número se almacenaría como 00011001. Complemento a 1: 11100110. Complemento a 2: 11100111.

Podemos restar dos números sumando al minuendo el complemento a 1 del sustraendo, y al resultado obtenido se le quita la cifra más significativa, la cual se suma al número anterior. También se puede hacer sumando al minuendo el complemento a 2 del sustraendo y al resultado se le quita la cifra más significativa.

Ejemplo. Restar 11001 menos 1011 mediante suma a complementos a 1.

$$\begin{array}{r}
 \begin{array}{cccccccc} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} & 25 \\
 + \begin{array}{cccccccc} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{array} & \text{Complemento a 1 de 11} \\
 \hline
 \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array} & 14 \\
 \begin{array}{c} \text{Carry} \\ \xrightarrow{\hspace{1.5cm}} \end{array} & 1 \\
 \hline
 \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array} & 14
 \end{array}$$

Esta misma resta usando suma a complementos a 2 sería:

$$\begin{array}{r}
 \begin{array}{cccccccc} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} & 25 \\
 + \begin{array}{cccccccc} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{array} & \text{Complemento a 2 de 11} \\
 \hline
 \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array} & 14
 \end{array}$$

Normalmente los procesadores actuales realizan la resta mediante complementos a 2.

Para **multiplicar** sólo nos basta conocer la tabla de multiplicar y operar como de costumbre.

$0 * 0 = 0$; $0 * 1 = 0$; $1 * 0 = 0$; $1 * 1 = 1$

$$\begin{array}{r}
 \begin{array}{cccccc} & & 1 & 1 & 0 & 0 & 1 \end{array} & 25 \\
 * \begin{array}{cccccc} & & 1 & 0 & 1 & 1 \end{array} & 11 \\
 \hline
 \begin{array}{cccccc} & & 1 & 1 & 0 & 0 & 1 \end{array} \\
 \begin{array}{cccccc} & 1 & 1 & 0 & 0 & 1 \end{array} \\
 \begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 \end{array} \\
 \begin{array}{cccccc} 1 & 1 & 0 & 0 & 1 \end{array} \\
 \hline
 \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array} & 275
 \end{array}$$

La **división** es algo más complicada, se suele hacer mediante restas sucesivas de forma que se resta al dividendo el divisor, el resultado obtenido se le vuelve a restar el divisor, así hasta que el resultado obtenido sea menor que el divisor. El número de restas realizadas sería el cociente de la división, y el resultado de la última resta sería el resto de la división.

Así para dividir 11001 entre 1011 sería:

$$\begin{array}{r}
 \begin{array}{cccccccc} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} & 25 \\
 + \begin{array}{cccccccc} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{array} & \text{Complemento a 2 de 11} \\
 \hline
 \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array} & 14 \\
 + \begin{array}{cccccccc} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{array} & \text{Complemento a 2 de 11} \\
 \hline
 \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} & 3
 \end{array}$$

Así el cociente sería 2 y el resto 3. Expresado en binario tendríamos:

$$\begin{array}{r} 11001 \overline{) 1011} \\ \underline{11} \\ 10 \end{array}$$

2.2.4 Sistema hexadecimal.

El sistema de numeración hexadecimal es un sistema de base 16, utiliza 16 símbolos distintos para expresar cantidades, de los cuales los diez primeros coinciden con el sistema decimal, y los otros seis deben ser añadidos. Los símbolos son: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

| DEC | HEX | BIN |
|-----|-----|------|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

A los símbolos A, B, C, D, E, F se le asignan los valores absolutos de 10, 11, 12, 13, 14, 15 respectivamente.

El sistema hexadecimal es muy usado ya que permite trabajar de forma más cómoda que con el sistema binario. Esto es así por que cada dígito hexadecimal equivale a 4 dígitos binarios según la tabla de la izquierda.

De esta forma, cantidades binarias que deben usar muchos dígitos para expresarse, quedan significativamente más reducidas en tamaño si se expresan en hexadecimal. Por ejemplo:

C4F7 equivaldría a: 1100010011110111

§ Conversión decimal a hexadecimal

La conversión se realiza con el mismo método que vimos para la conversión de decimal a binario, salvo que las operaciones de dividir o multiplicar por 2 se realizan ahora con 16. El número decimal 472,3229 en hexadecimal sería 1D8,52A9 (4 dígitos precisión).

§ Conversión hexadecimal a decimal

Se realiza aplicando el TFN al número hexadecimal y realizando los cálculos:

El número hexadecimal 20A sería en decimal 522.

§ Conversión hexadecimal a binario

Para convertir un número hexadecimal a binario, se sustituye cada dígito hexadecimal por su representación binaria con cuatro dígitos.

El número hexadecimal CAFE se convertiría en 1100101011111110

§ Conversión binario a hexadecimal

Se agrupan los dígitos binarios de cuatro en cuatro a partir de la coma fraccionaria, sustituyendo cada cuarteto por su correspondiente dígito hexadecimal. Si el número binario no tiene un número de bits múltiplo de cuatro, se añadirán ceros hasta conseguirlo.

Para el número binario 101101010,11 se agruparía de la forma
Se añaden ceros hasta formar cuartetos
Se sustituyen los cuartetos por sus correspondientes hexadecimales

1 0110 1010 , 11
0001 0110 1010 , 1100
16A,C

2.3 Objetos de un programa.

Son **objetos** de un programa todos aquellos manipulados por las instrucciones. Una **variable**, es un objeto mediante el cual podremos realizar el almacenamiento de los datos y de los resultados de las distintas operaciones que conducen a la resolución del problema. Se corresponde con una zona de la memoria.

Toda variable tiene tres **atributos**:

- § **Identificador**: Es el nombre del mismo. En realidad el nombre simbólico que se le da a una zona de memoria. Las normas generales de empleo son: puede constituirse por letras, dígitos y generalmente por el carácter subrayado (_), deben comenzar por una letra y no pueden contener espacios.
- § **Tipo**: Conjunto o rango de valores que puede tomar. El tipo determina el espacio de memoria a usar.
- § **Valor**: elemento que se asigna perteneciente al tipo definido.

Las **constantes** son objetos cuyo valor permanece invariable a lo largo de la ejecución de un programa. Una constante es la denominación de un valor concreto. Se puede expresar de forma explícita mediante su valor, o utilizando un identificador para definir la constante en memoria, asignándole un valor.

Ejemplos: 4096, "Antonio", PI = 3.141592, SIMBOL_MONETARIO = '€'

Las **variables** son objetos cuyo valor puede ser modificado a lo largo de la ejecución del programa.

Ejemplos:

```
R = 0
R = R + 1
L = 2 * PI * R
```

Los **tipos** de las variables y constantes se clasifican de forma general en:

- § **Númericos enteros**. Lo forman el conjunto de los números enteros: 27, -32, +567
- § **Númericos reales**. Lo forman el conjunto de los reales: 78.5, -3.02, 0.45E-2
- § **Alfanuméricos o carácter**. Lo forman el conjunto de todos los caracteres disponibles: Se representan entre comillas. "Dolores", "GR-1234-AB", "24876964".
- § **Booleanos**: Lo forman el conjunto de los dos valores lógicos: Verdadero y Falso (Sí, No).

A parte de estos tipos generales, cada lenguaje puede añadir algunos tipos más especializados. Existen lenguajes que no tienen definido el tipo Booleano, como son los casos de COBOL o C.

2.4 Expresiones.

Una **expresión** es una combinación de valores, constantes, variables, funciones y operadores, cumpliendo unas determinadas reglas de combinación, que devuelven un valor. Los **operadores** permiten elaborar las expresiones, y se clasifican en los siguientes tipos:

- § **Aritméticos**:

| | |
|----------|---|
| Potencia | ^ |
| Producto | * |
| División | / |
| Suma | + |
| Resta | - |
- § **Relacionales**:

| | | |
|-------------------|----|------|
| Igual que | = | (==) |
| Distinto de | <> | (!=) |
| Mayor que | > | |
| Mayor o igual que | >= | |
| Menor que | < | |
| Menor o igual que | <= | |

§ Lógicos:

| | | |
|------------|-----|------|
| Negación | NOT | (!) |
| Conjunción | AND | (&&) |
| Disyunción | OR | () |

En C existen algunos operadores más: **módulo** (%), **decremento** (--), **incremento**(++) y los operadores a **nivel de bits** que se verán con el estudio del lenguaje.

En las expresiones se pueden usar los paréntesis para anidar expresiones.

Los operadores aritméticos son de sobra conocidos y operan según las reglas de cálculo matemático. Los operadores relacionales nos permite conocer si una relación de orden es verdadera o falsa. Los operadores lógicos nos permite conectar relaciones entre sí siguiendo las reglas de la lógica formal.

En C, al no existir el tipo Booleano, se considera falso el valor cero y verdadero cualquier valor distinto de cero. Las expresiones en C que utilizan los operadores relacionales y/o lógicos devuelven el valor **0** en caso de falso y el valor **1** en caso de verdadero. Las **tablas de verdad** para los operadores lógicos son:

| A | NOT A | A | B | A AND B | A | B | A OR B |
|---|-------|---|---|---------|---|---|--------|
| V | F | F | F | F | F | F | F |
| F | V | F | V | F | F | V | V |
| | | V | F | F | V | F | V |
| | | V | V | V | V | V | V |

El orden de prioridad al evaluar los operadores en una expresión es el siguiente:

1. Paréntesis.
2. Aritméticos. En el orden potencias, multiplicaciones y divisiones, sumas y restas
3. Relacionales. Todos con la misma prioridad.
4. Lógicos. En el orden negación, conjunción, disyunción

A igualdad de prioridad de operadores se resuelven de izquierda a derecha.

Ejemplos:

1. Resolver paso a paso las expresiones siguientes:

$$5 - 2 > 4 \text{ AND NOT } 0.5 = 1 / 2$$

$$5 - 2 > 4 \text{ AND NOT } 0.5 = 0.5$$

$$3 > 4 \text{ AND NOT } 0.5 = 0.5$$

$$F \text{ AND NOT } V$$

$$F \text{ AND } F$$

$$F$$

$$1 > 3 \text{ AND } (4 = 4 \text{ OR } 8 \text{ AND } 1)$$

$$1 > 3 \text{ AND } (V \text{ OR } 8 \text{ AND } 1)$$

$$1 > 3 \text{ AND } (V \text{ OR } V)$$

$$1 > 3 \text{ AND } V$$

$$F \text{ AND } V$$

$$F$$

2. Expresar las fórmulas matemáticas siguientes en expresiones computacionales válidas:

$$(A + B)^2$$

$$\frac{(A + B)^2}{A^2 + B^2}$$

$$(A + B) ^ 2 / (A ^ 2 + B ^ 2)$$

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

$$2a$$

$$(-b + (b ^ 2 - 4 * a * c) ^ 0.5) / (2 * a)$$

2.5 Representación de algoritmos. Herramientas y notaciones.

Para representar un algoritmo, se debe utilizar una notación que nos permita plasmar la solución encontrada. Dicha representación debe ser independiente del lenguaje de programación que se escoja para la codificación posterior del algoritmo. Con esto conseguimos una representación que podrá implementarse en diferentes sistemas y lenguajes según convenga. Los métodos o herramientas más usuales de representación de algoritmos son: *diagramas de flujo*, *pseudocódigo*, *tablas de decisión*, y *diagramas estructurados*.

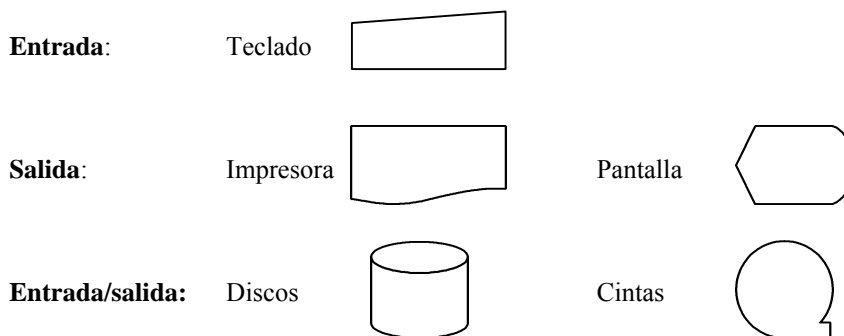
Estas herramientas pretenden representar de forma gráfica o textual los flujos que van a seguir los datos manipulados, así como la secuencia lógica de operaciones en la resolución de un problema. Las representaciones deben ser: sencillas de construir, claras de entender, normalizadas en el diseño y flexibles en sus modificaciones.

2.5.1 Diagramas de flujo de sistema. Organigramas.

Los organigramas sirven para representar gráficamente el flujo de datos e informaciones que maneja un programa. En el caso de aplicaciones que comprenden más de un programa, se realizará un organigrama por cada uno de ellos, más uno general que englobe todo el conjunto. El organigrama debe reflejar:

- § Los soportes y periféricos para los datos.
- § Los nombres de los programas.
- § El flujo de los datos.

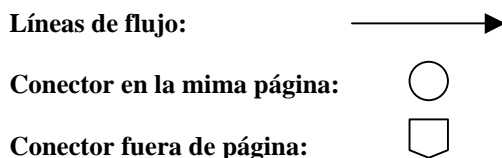
Para representar los elementos anteriores se usan elementos gráficos distintos. Los soportes son los medios capaces de almacenar información. Los periféricos son los dispositivos que manejan a dichos soportes. Se clasifican en: soportes de entrada, salida y entrada/salida. Los símbolos para soportes y periféricos son:



El nombre del programa aparece dentro de un rectángulo. El símbolo se llama de proceso:



Los flujos de datos se representan con flechas que conectan los símbolos anteriores, indicándose de esta manera la circulación de las informaciones entre los soportes y los procesos. También se utilizan conectores para indicar que el flujo continúa en otra parte de la página o en otra página.

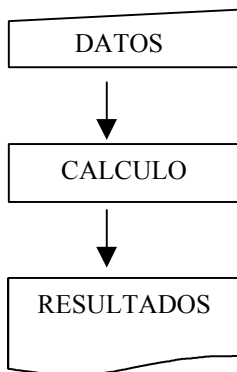


Las reglas para construir un organigrama son:

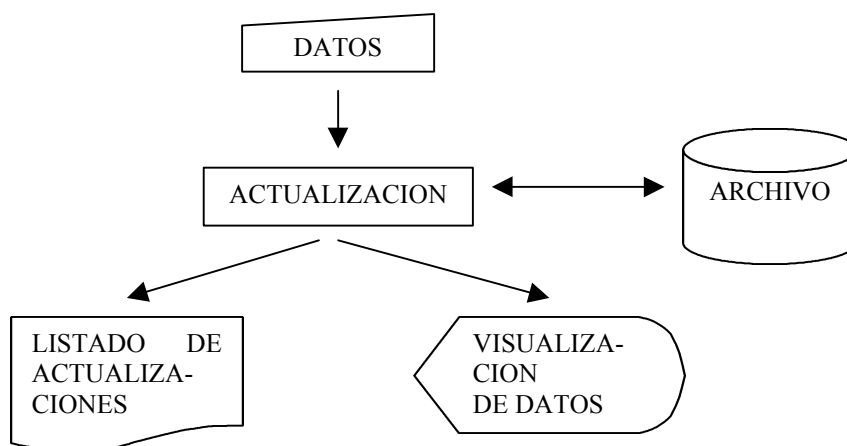
- § En el centro del organigrama figurará el símbolo de proceso correspondiente al programa
- § En la parte superior los soportes y periféricos de entrada

- § En las partes inferiores los soportes y periféricos de salida
- § En el mismo nivel que el símbolo de proceso y a ambos lados, los soportes de E/S.

Ejemplo: Organigrama de una aplicación que tome números desde teclado, realice alguna operación de cálculo con ellos, dando el resultado por impresora:



Ejemplo: Organigrama de una aplicación de **actualización** de un archivo soportado en disco, con entrada de datos por teclado, consulta de datos por pantalla y salida de datos modificados por impresora:



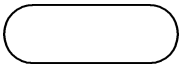
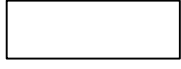
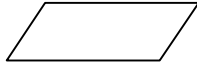
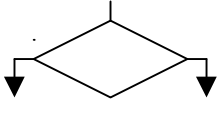

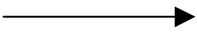

2.5.2 Diagrama de flujo de programa. Ordinogramas.

Los ordinogramas (*flowchart*) permiten representar gráficamente la secuencia lógica de las operaciones que se realizan en un programa de ordenador. Por cada programa se realiza un ordinograma, a partir del cual se realiza la codificación en un lenguaje. Esta técnica no es obligada, ya que hay otras con el mismo fin que pueden sustituir a ésta. Aunque es la más antigua, cada vez se emplea menos sobre todo partir de la aparición de los lenguajes de programación estructurados.

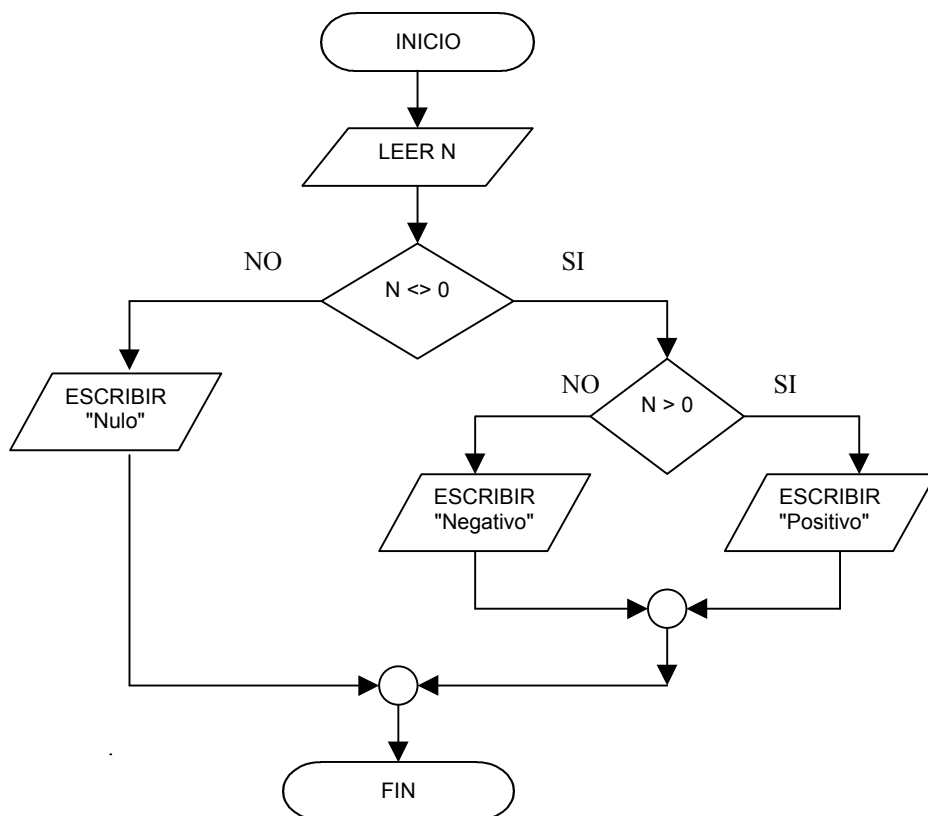
El diagrama de flujo debe reflejar paso a paso todas las operaciones del programa y su secuencia desde el comienzo hasta el final de éste. En la representación de los diagramas de flujo se siguen las reglas:

- § El comienzo del programa debe aparecer en la parte superior.
- § Los símbolos de comienzo y de final deberán aparecer una sola vez.
- § El flujo de las operaciones será de arriba abajo y de izquierda a derecha usando líneas y flechas.
- § Debe ser lo más simétrico y equilibrado posible.
- § Se evitarán los cruces de las líneas de flujo utilizando conectores.

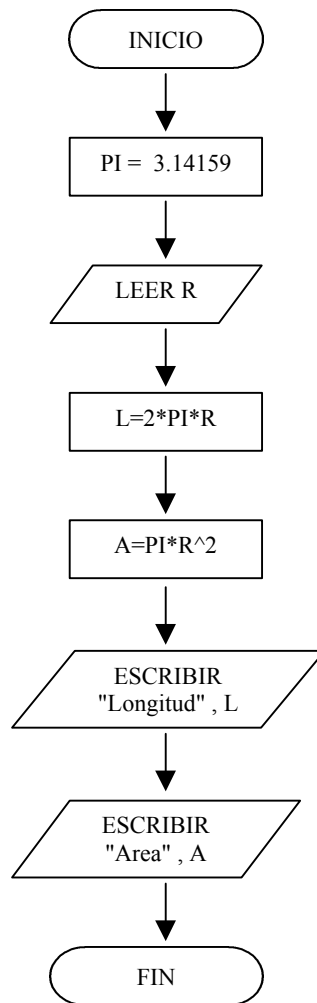
Los símbolos son:

| | | |
|------------------------|--|--|
| Terminal |  | Representan el comienzo o final de un programa |
| Proceso |  | Cualquier operación en la que cambie el valor de una zona de memoria |
| Entrada/Salida |  | Operación de introducción de datos desde un periférico a la memoria, o de salida de información por un periférico. |
| Decisión |  | Indica cual de los caminos alternativos del programa se seguirá en función del resultado de evaluar una expresión lógica |
| Conexión |  | Permite conectar dos puntos del diagrama en la misma u otra página |
| Líneas de flujo |  | Indica el sentido de la ejecución de las operaciones. |
| Subprograma |  | Indica la llamada a una subrutina o a un proceso determinado |

Ejemplo: programa que lee un número de teclado, y comprueba y escribe si dicho número es positivo, negativo o nulo.



Ejemplo: programa que lee un número que corresponde al radio de una circunferencia y calcula e imprime la longitud de la misma y el área del círculo correspondiente.



2.5.3 Pseudocódigo.

Además de la representación gráfica, un programa puede describirse mediante un lenguaje intermedio entre el lenguaje natural y el lenguaje de programación. Esto permite flexibilidad para expresar las acciones que se han de realizar y, sin embargo, imponga algunas limitaciones importantes desde el punto de vista de su posterior codificación.

Al igual que otras técnicas, podemos diseñar el programa sin depender de ningún lenguaje de programación. Un **pseudocódigo** es una notación mediante la que se puede describir la solución de un problema en forma de algoritmo dirigido a un computador, utilizando palabras y frases del lenguaje natural sujetas a unas determinadas reglas.

Debido a su flexibilidad, permite obtener la solución a un problema mediante aproximaciones sucesivas, es decir, mediante un diseño descendente. Todo pseudocódigo debe posibilitar la descripción de los siguientes elementos:

- § Instrucciones de entrada y salida
- § Instrucciones de proceso.
- § Sentencias de control de flujo de ejecución
- § Acciones compuestas (subprogramas).
- § Comentarios.

Programa: Nombre del programa
Entorno:
Declaración de los objetos del programa
Algoritmo:
Secuencia de instrucciones que forman el programa
Fin de programa.

```
Programa: calculo
Entorno:
    PI, R, L, A:  numéricas reales
Algoritmo:
    PI = 3.1415926
    LEER  R
    L = 2 * PI * R
    A = PI * R ^ 2
    ESCRIBIR "La longitud de la circunferencia es ", L
    ESCRIBIR " El área del círculos es ", A
Fin de programa.
```

Una tabla de decisión es una representación tabular de la lógica de un problema, en el que se presentan varias situaciones y diferentes alternativas para cada una de ellas. Se usa para el análisis de problemas y la detección de errores u omisiones en la solución del problema. En la programación representa un programa o parte del mismo, de tal forma que a partir de unos datos de entrada determinados, realiza una acción de acuerdo con las condiciones del problema que se trate.

| | |
|--|---|
| <p>CONDICIONES</p> <p>Vector columna donde figuran las condiciones que intervienen de mayor a menor importancia</p> | <p>ENTRADA DE CONDICIONES</p> <p>Matriz de tantas filas como condiciones y columnas como situaciones distintas se puedan presentar</p> |
| <p>ACCIONES</p> <p>Vector columna donde aparecen las acciones a realizar</p> | <p>SALIDA DE ACCIONES</p> <p>Matriz de tantas filas como acciones y columnas como situaciones distintas se pueden presentar.</p> |

| | | | | | | | | | |
|-------------|---|---|---|---|---|---|---|---|---------------|
| | S | S | S | N | N | N | N | | ← Situación |
| CONDICION 1 | S | S | S | N | N | N | N | | |
| CONDICION 2 | S | S | N | N | S | S | N | N | |
| CONDICION 3 | S | N | S | N | S | N | S | N | |
| ACCION 1 | X | | X | X | | | X | | |
| ACCION 2 | | X | | | X | X | | X | ← Tratamiento |

IES Zaidín Vergeles de Granada. Departamento de Informática. Desarrollo de Aplicaciones Informáticas.

Ejemplo: Una compañía de seguros tiene establecida las siguientes tarifas para sus dos tipos de póliza de seguros de automóviles:

1. Tarifa base para seguro obligatorio: 50.000 ptas.
2. Tarifa base para seguro a todo riesgo: 100.000 ptas.
3. Estas tarifas pueden incrementarse o reducirse según la situación del cliente:
 - Si el vehículo es industrial, se aumenta el 70 % de la tarifa base.
 - Si el asegurado es mayor de 60 años, su vehículo no es industrial y la póliza es de seguro obligatorio, se incrementa la tarifa base en un 8%.

| | |
|--------------------|------------------------|
| Tarifa base | O O O O T T T T |
| Industrial | S S N N S S N N |
| Mayor 60 | S N S N S N S N |
| 50.000 | X X X X |
| 100.000 | X X X X |
| Aumento 70% | X X X X |
| Aumento 8% | X |

2.5.5 Diagramas Estructurados.

Son métodos gráficos que conducen a la representación de una solución algorítmica. Existen diferentes tipos según los autores que los crearon y diseñaron. Así existen los diagramas Warnier, Jackson, Bertini, Tabourier, Chapin. Estos métodos de representación no se verán en este curso. Como ejemplo mostraremos como se representa el algoritmo que determina si un número leído de teclado es negativo, positivo o nulo usando los diagramas de Warnier y Chapin.

Diagrama de Warnier

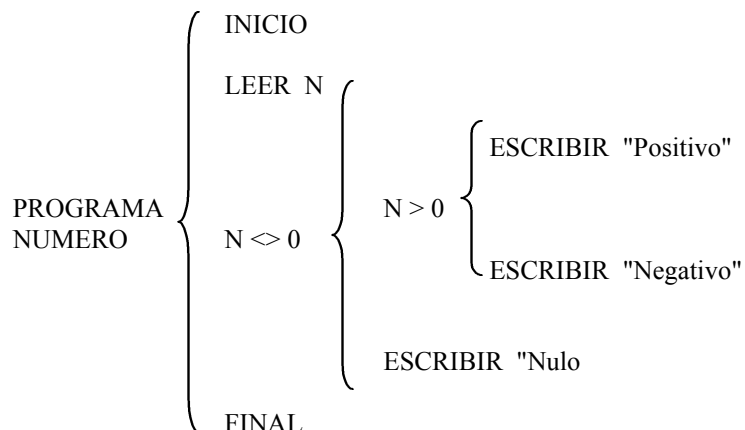
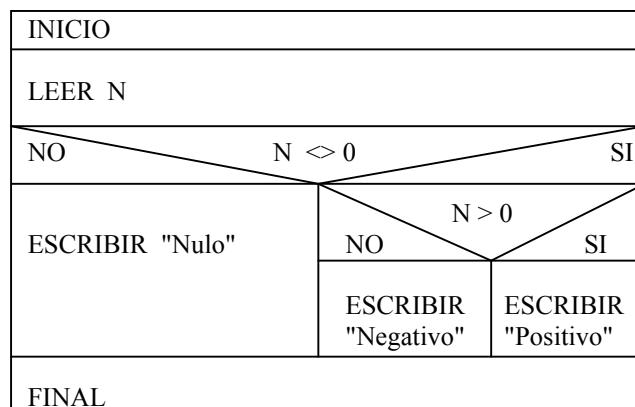


Diagrama de Chapin



2.6 Estructura general de un programa.

Un programa puede considerarse como una secuencia de acciones (instrucciones) que manipulan un conjunto de objetos (datos). Contendrá por tanto dos bloques para la descripción de los dos aspectos citados.

- § **Bloque de declaraciones.** En el se especifican todos los objetos que utiliza el programa (constantes, variables, tablas, archivos, etc.).
- § **Bloque de instrucciones.** Constituido por el conjunto de operaciones que se han de realizar para la obtención de los resultados deseados.

En algunos lenguajes de programación no figura explícitamente el bloque de declaraciones (FORTRAN, BASIC), pero, en general, es necesario declarar todos los objetos que se van a usar (COBOL, Pascal, PL/I, C, Ada).

La ejecución de un programa consiste en la realización secuencial del conjunto de instrucciones, desde la primera a la última, de una en una. Este orden de realización solamente será alterado mediante instrucciones denominadas de ruptura de secuencia. Las instrucciones de un programa consisten, en general, en modificaciones sobre los objetos de un programa, que constituyen su entorno, desde un estado inicial hasta otro final.

2.6.1 Partes de un programa.

Dentro del bloque de instrucciones de un programa podemos diferenciar tres partes fundamentales. En algunos casos, estas tres partes están perfectamente delimitadas; pero, en la mayoría, sus instrucciones quedan entremezcladas a lo largo del programa.

- § **Entrada de datos.** La constituyen todas aquellas instrucciones que toman datos de un dispositivo externo, almacenándolos en la memoria central para que puedan ser manipulados. También se consideran dentro de este apartado las instrucciones de depuración de los datos de entrada, es decir, aquellas que se encargan de comprobar la validez de los mismos.
- § **Proceso.** Están formados por las instrucciones que modifican los objetos a partir de su estado inicial, hasta el estado final, dejando éstos disponibles en la memoria central.
- § **Salida de resultados.** Conjunto de instrucciones que toman los datos finales de la memoria central y los envía a los dispositivos externos.

El código de un programa escrito en un lenguaje simbólico se denomina **código fuente**. Este código normalmente reside en un archivo de disco llamado **archivo fuente**. El resultado de la traducción a código máquina se denomina **código o módulo objeto**. El archivo en que reside se denomina **archivo objeto**.

2.6.2 Clasificación de las instrucciones.

Las instrucciones se clasifican en tres grandes grupos. Para cada tipo de instrucción se mostrarán sus representaciones en ordinograma y pseudocódigo.

- § Instrucciones primitivas:
 - Asignación.
 - Entrada.
 - Salida.
- § Instrucciones de declaración.
- § Instrucciones de control:
 - Decisión simple.
 - Decisión compuesta.
 - Repetitivas.

Instrucciones primitivas. Son aquellas que ejecuta el procesador de modo inmediato.

- **Asignación.** Consiste en calcular el valor de una expresión y almacenarlo en una variable. Su formato es:

Variable = expresión

Variable = expresión

Ejemplos:

```
salario = bruto_anual / 12
poblacion = "Alomartes"
contador = contador + 1
```

- **Entrada.** Toma un dato de un dispositivo de entrada y lo almacena en un objeto. Formato:

LEER variable1, variable2,

LEER variable1, variable2,

Ejemplos:

```
LEER bruto_anual
LEER nombre, apellidos
```

- **Salida.** Toma el valor de una expresión u objeto y lo lleva a un dispositivo externo. Formato:

ESCRIBIR variable1 u expresión1, variable2 u expresión2, ...

ESCRIBIR variable1 u expresión1,
variable2 u expresión2, ...

Ejemplos:

```
ESCRIBIR nombre, apellidos
ESCRIBIR "Lugar de nacimiento: ", poblacion
ESCRIBIR "Salario mensual: ", bruto_anual / 12, " Euros."
```

Instrucciones de declaración. Indican el tipo, características e identificación de los objetos que componen un programa. Por ejemplo si es numérico, alfanumérico, número de caracteres, etc. En la notación de pseudocódigo, las declaraciones van colocadas en el bloque denominado entorno.

Ejemplo:

Programa: Salarios

Entorno:

```
nombre, apellido: alfabético de 25 caracteres
poblacion: alfabético de 20 caracteres
bruto_anual: entero
salario: real
nif: alfanumérico de 9 caracteres
.....
```

Algoritmo:

Secuencia de instrucciones que forman el programa

Fin de programa.

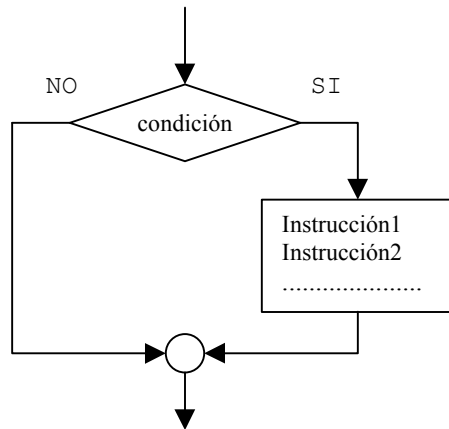
Instrucciones de control. Evalúan expresiones lógicas con el objetivo de controlar la ejecución de otras instrucciones o alterar el orden de ejecución normal de otras.

- **Instrucción de decisión o alternativa simple.**

```
.....
SI condición
    Instruccion1
    Instruccion2
.....
FINSI
.....
```

Ejemplo:

```
SI (a = b)
    ESCRIBIR "Valores iguales"
FINSI
```

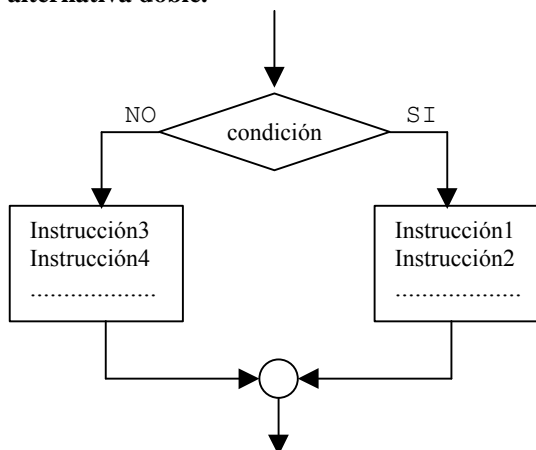


- **Instrucción de decisión compuesta o alternativa doble.**

```
.....
SI condición
    Instruccion1
    Instruccion2
.....
SINO
    Instruccion3
    Instruccion4
.....
FINSI
.....
```

Ejemplo:

```
SI (a = b)
    ESCRIBIR "Valores iguales"
SINO
    ESCRIBIR "Valores distintos"
FINSI
```

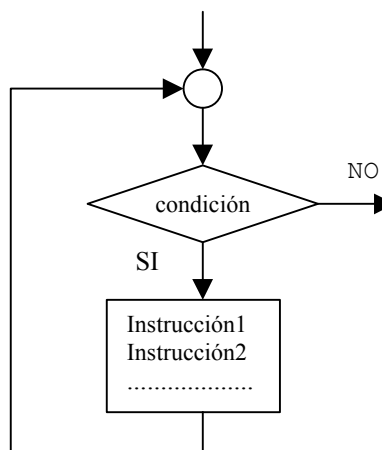


- **Instrucción de control repetitiva mientras (do while).** Permite repetir un bloque de instrucciones mientras se cumpla una condición. La condición se evalúa antes entrar a ejecutar el bloque de instrucciones.

```
.....
MIENTRAS condición
    Instruccion1
    Instruccion2
.....
FINMIENTRAS
.....
```

Ejemplo:

```
.....
N = 1
MIENTRAS (N <= 10)
    ESCRIBIR N
    N = N + 1
FINMIENTRAS
```



- **Instrucción de control repetitiva hasta (*until*).** Permite repetir un bloque de instrucciones hasta que se cumpla una condición. La condición se evalúa después de ejecutar el bloque de instrucciones.

```

.....
HASTA
    Instruccion1
    Instruccion2
    .....
FINHASTA condición
.....

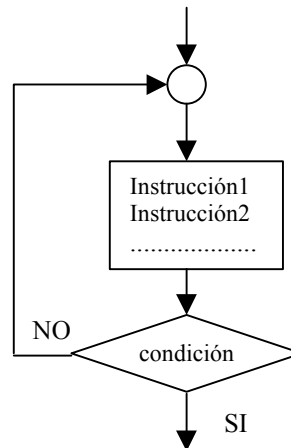
```

Ejemplo:

```

N = 1
HASTA
    ESCRIBIR N
    N = N + 1
FINHASTA (N > 10)

```



- **Instrucción de control repetitiva para (*for*).** Hace que se repita un bloque de instrucciones un número determinado de veces fijado de antemano. El bucle normalmente utiliza una variable llamada de control de bucle. Cuando se llega a una instrucción *for*, la variable de control se inicializa, posteriormente se evalúa la condición. Si es cierta se ejecuta el bloque de instrucciones. Por último se realiza el incremento de la variable de control.

```

.....
PARA (Iniciación, condición, incremento)
    Instruccion1
    Instruccion2
    .....
FINPARA
.....

PARA (VC=VI, VC<=VF, VC=VC+IN)
    Instruccion1
    Instruccion2
    .....
FINPARA
.....

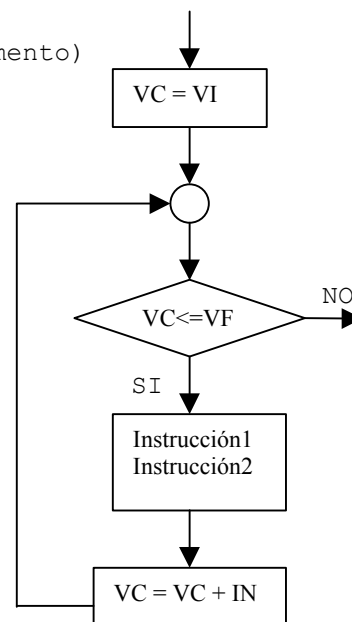
```

Ejemplo:

```

PARA (N=1, N<=10, N=N+1)
    ESCRIBIR N
FINPARA

```



Ejemplo: Pseudocódigo de un programa que lea una secuencia de 10 números enteros y vaya determinando si son positivos negativos o nulos.

```

PROGRAMA: Números
ENTORNO:
    C numérica entera
    N numérica entera

```

```

ALGORITMO:
  PARA (C = 1, C <= 10, C=C+1)
    LEER N
    SI (N <> 0)
      SI (N > 0)
        ESCRIBIR N, " es positivo"
      SINO
        ESCRIBIR N, " es negativo"
      FINSI
    SINO
      ESCRIBIR N, " es nulo"
    FINSI
  FINPARA
FIN Números

```

2.6.3 Variables auxiliares.

Son variables que realizan funciones específicas dentro de un programa. Debido a su utilidad y frecuencia de uso, vamos a estudiarlas de forma separada.

- **Contadores.** Es una variable cuyo valor se incrementa en una cantidad fija, positiva o negativa, generalmente asociado a un bucle. Se suele utilizar para contar el número de veces que es necesario repetir una acción (variable de control de bucle), o para contar un suceso particular solicitado en el enunciado del problema.

Ejemplo: pseudocódigo de un programa que lea una secuencia de notas de una asignatura y que contabilice al final el número de suspensos y de aprobados. La secuencia de notas termina al teclear una nota negativa.

```

PROGRAMA: Notas
ENTORNO:
  CA, CS: numérica entera
  N: numérica real
ALGORITMO:
  CA=0
  CS=0
  ESCRIBIR "Introduzca nota: "
  LEER N
  MIENTRAS (N >= 0)
    SI (N >= 5)
      CA = CA + 1
    SINO
      CS = CS + 1
    FINSI
    ESCRIBIR "Introduzca nota: "
    LEER N
  FINMIENTRAS
  SI (CA = 0 AND CS = 0)
    ESCRIBIR "No introdujo ninguna nota"
  SINO
    ESCRIBIR "El número de aprobados es: ", CA
    ESCRIBIR "El número de suspensos es: ", CS
  FINSI
FIN Notas

```

- **Acumuladores.** Un acumulador es una variable cuyo valor se incrementa sucesivas veces en cantidades variables. Se utiliza en aquellos casos en que se desea obtener el total acumulado de un conjunto de cantidades, siendo preciso inicializarlos con valor cero. También, cuando hay que obtener un total como producto de distintas cantidades, se utiliza un acumulador, debiéndose inicializar con el valor uno.

Ejemplo: Realizar el pseudocódigo de un programa que permita calcular la suma y el producto de los 10 primeros números naturales pares.

```
PROGRAMA: Acumular
ENTORNO:
    S, P, N: numérica entera
ALGORITMO:
    S=0
    P=1
    PARA (N=2, N<=20, N=N+2)
        S=S+N
        P=P*N
    FINPARA
    ESCRIBIR "La suma es: ", S
    ESCRIBIR "El producto es: ", P
FIN Acumular
```

- **Conmutadores , interruptores o switches.** Un interruptor es una variable que puede tomar exclusivamente dos valores (0 y 1, 1 y -1, verdadero y falso, etc.). Se utiliza para recordar en un determinado punto de un programa la ocurrencia o no de un suceso anterior, para salir de un bucle, o para decidir en una instrucción alternativa que acción realizar. También se suele utilizar para hacer que dos acciones diferentes se ejecuten alternativamente dentro de un bucle.

Ejemplo: Realizar el pseudocódigo de un programa que sume independientemente los pares e impares de los números comprendidos entre -100 y 100.

```
PROGRAMA: Pares_impares
ENTORNO:
    SP, SI, N, SW: numérica entera
ALGORITMO:
    SP=0
    SI=0
    SW=-1
    PARA (N=-100, N <= 100, N=N+1)
        SW = (-1)*SW
        SI (SW = 1)
            SP=SP+N
        SINO
            SI=SI+N
        FINSI
    FINPARA
    ESCRIBIR "La suma de los pares es: ", SP
    ESCRIBIR "La suma de los impares es: ", SI
FIN Pares_impares
```

2.7 Técnicas de programación.

En la resolución de problemas, la primera tarea a realizar es el **análisis** de éste. Esta tarea requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución deseada. Para poder definir bien un problema, deberemos obtener la respuesta a las siguientes preguntas:

- ¿Qué entradas (tipo y cantidad) se requieren?.
- ¿Cuál es la salida (tipo y cantidad) deseada?.
- ¿Qué método produce la salida deseada?.

Una vez realizado el análisis del proceso de programación, lo que hemos conseguido es saber *qué* hace el programa. En la fase de **diseño** debemos determinar *cómo* hace el programa la tarea solicitada. Las técnicas y métodos más eficaces se basan en la división del problema en otros más simples o subproblemas, lo que se conoce como *diseño modular*.

2.7.1 Programación modular.

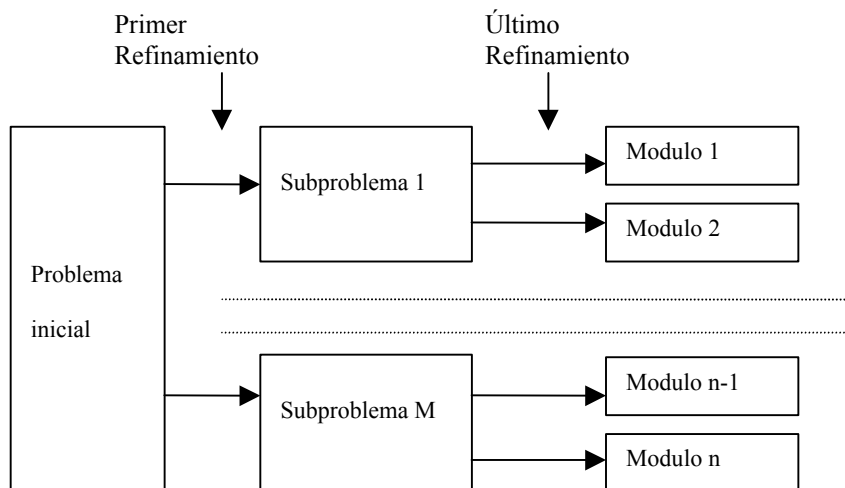
Los problemas reales que se plantean a un programador normalmente requieren programas de una cierta complejidad y a veces de gran tamaño. Abordar el diseño de un programa de estas características de una forma directa es una tarea muy complicada, si no imposible.

Lo más adecuado es descomponer el problema, ya desde su fase de análisis, en partes cuya resolución sea más asequible. La programación de cada una de estas partes se realiza independientemente de las otras, incluso, en ocasiones por otras personas.

Por otro lado la depuración y puesta a punto del programa hace necesario que el listado del mismo sea fácilmente comprensible. En esta sentido conviene subdividir el programa de tal manera que cada parte sea suficientemente reducida y sencilla para su desarrollo y mantenimiento.

El **diseño modular y descendente** (*top-down*) consiste en una serie de descomposiciones sucesivas del problema inicial en subproblemas, y a continuación dividir estos subproblemas en otros de nivel más bajo. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina **refinamiento sucesivo**.

Cada subprograma es resuelto mediante un **módulo** que tiene un solo punto de entrada y un solo punto de salida. La división modular termina cuando cada módulo tenga solamente una tarea específica que ejecutar.



Un programa bien diseñado consta de un *modulo principal* (el módulo de nivel más alto) que llama a submódulos (módulos de nivel más bajo), que a su vez pueden llamar a otros submódulos. Cada submódulo devuelve el control al submódulo del cual se recibió originalmente el control.

Los módulos son independientes en el sentido de que ningún módulo puede tener acceso directo a cualquier otro módulo excepto al módulo o módulos a los que llama.

El diseño modular tiene los siguientes objetivos básicos:

- Simplificación del problema y de los subproblemas resultantes de cada descomposición.
- Las diferentes partes del problema pueden ser programadas de modo independiente e incluso por diferentes personas.
- El programa final queda estructurado en forma de bloques o módulos, lo que hace más sencilla su lectura y mantenimiento.
- Si los módulos están bien diseñados y son lo suficientemente independientes, se podrán reutilizar en la solución y diseño de otros problemas software.

Con la utilización de esta técnica de diseño, surgen los conceptos de:

- Programa principal y subprogramas.
- Subprogramas internos y externos
- Objetos globales y locales
- Parámetros de enlace.

2.7.2 Programa principal y subprogramas.

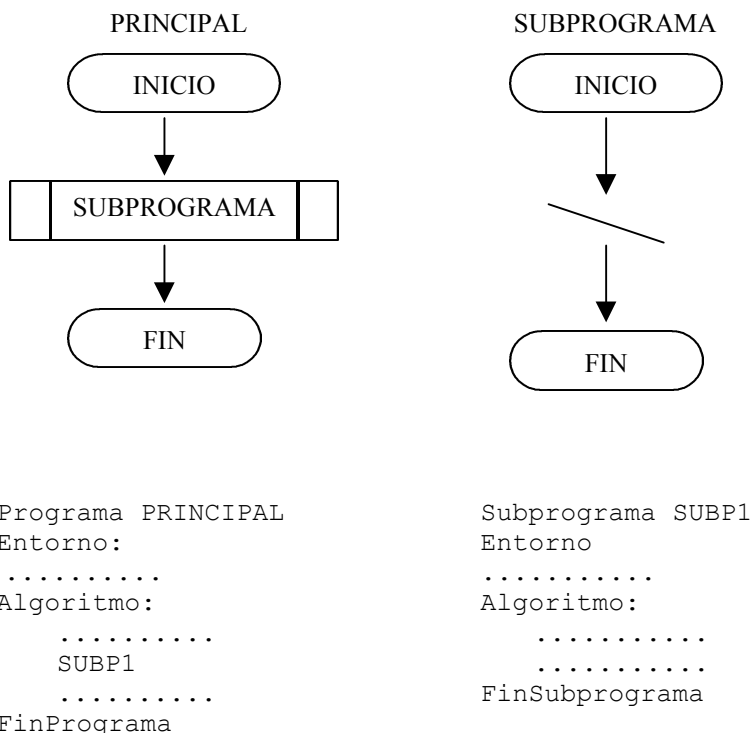
Un **programa principal** describe la solución completa del problema y consta principalmente de llamadas a subprogramas. Estas llamadas son indicaciones al procesador de que debe continuar la ejecución del programa en el subprograma llamado, regresando al punto de partida una vez lo haya concluido.

El programa principal contendrá pocas líneas, y en él se verán claramente los diferentes pasos del proceso que se ha de seguir. El programa principal implementa el módulo principal de la solución.

La estructura de un **subprograma** es básicamente la de un programa, con alguna diferencia en el encabezamiento y finalización. Su función es resolver de modo independiente una parte del problema; es decir implementa un submódulo de la solución. Es importante que realice una función concreta en el contexto del problema. Un subprograma es ejecutado por el procesador exclusivamente al ser llamado por el programa principal o por otro subprograma.

Si el subprograma figura en el mismo archivo fuente que el principal, se le denomina **interno**. La denominación varía según los lenguajes de programación: Párrafos en COBOL, procedimientos y funciones en Pascal, Subrutinas en Basic, funciones en C, etc.

Si el subprograma figura físicamente separado del programa principal en otro archivo fuente, se le denomina **externo**. Pueden ser compilados separadamente, incluso haber sido codificado en un lenguaje de programación distinto al principal. Generalmente se enlazan con el programa principal en la fase de montaje o de enlazado (*linkage*), cuando ya son módulos objetos. La representación de los subprogramas mediante diagramas de flujo y pseudocódigo son las siguientes:



2.7.3 Objetos globales y locales.

Los diferentes objetos que manipula un programa (constantes, variables, tablas, etc.) se clasifican según su **ámbito**. Es decir, según la porción de programa y/o subprograma en que son conocidos, y por lo tanto pueden ser utilizados.

Son **objetos globales**, los declarados en el programa principal, cuyo ámbito se extiende al mismo y a todos los subprogramas declarados en él. Son **objetos locales** a un subprograma, los declarados en dicho subprograma, cuyo ámbito está restringido a él mismo y a los subprogramas declarados en él.

Lenguajes como COBOL y BASIC todos los objetos son globales. Pascal y C si contemplan esta clasificación.

2.7.4 Programación estructurada.

Como hemos visto, una de la técnicas que simplifican la programación es la programación **modular descendente**. Además de ésta y como complemento existe la técnica de **programación estructurada**. Fue desarrollada en sus principios por Dijkstra y continuada por Chapin, Warnier, Jackson, Bertini, etc. como autores de sistemas de representación de algoritmos estructurados, y por Niklaus Wirth, Dennis Ritchie y Kenneth Thompson como autores de lenguajes de programación estructurados.

Existen varias definiciones de programación estructurada, basadas en el denominado **Teorema de estructura**. Se basa en el concepto de **diagrama o programa propio**. Un programa se define como propio si cumple las siguientes características:

1. Posee un solo punto de entrada y un solo punto de salida.
2. Existen caminos desde la entrada hasta la salida que se pueden recorrer y que pasan por todas las partes del programa.
3. Todas las instrucciones son ejecutables y no existen bucles infinitos.

El teorema de la estructura dice:

*Todo diagrama o programa propio, cualquiera que sea el trabajo que tenga que realizar, se puede hacer utilizando tres únicas estructuras de control que son la **secuencia, alternativa y repetitiva**.*

Básicamente, la programación estructurada significa escribir un programa de acuerdo con las siguientes reglas:

- El programa tiene un diseño modular.
- Los módulos son diseñados de forma descendente
- Cada módulo se diseña utilizando las tres estructuras de control básicas: secuencia, selección y repetición.

La programación estructurada pretende que los programas no usen de forma indiscriminada instrucciones de salto condicional o incondicional, que producen complejidad en la lectura y modificación de los programas. Gráficamente se le denominan "programas espaguetis".

2.8 Introducción a las funciones.

Una función es un conjunto de sentencias que realizan una tarea determinada. En muchos lenguajes, como es el caso del C, se puede distinguir entre las funciones que crean los usuarios y las que ofrece el propio lenguaje (funciones de biblioteca). Comentaremos las operaciones básicas a realizar para usar funciones.

2.8.1 Declaración de funciones.

Es lo primero que hay que realizar antes de definir y usar una función. La declaración también se conoce como **prototipo**. Aquí se indica el nombre de la función, el tipo de valor retornado, y el número y tipo de los

parámetros que utilizará. Los parámetros son variables que permiten recibir o entregar valores entre el programa que hace la llamada y la función.

```
tipo_de_retorno nombre_función(lista de los tipos de parámetros)
```

Los prototipos sirven para que el compilador del lenguaje pueda chequear si los parámetros y el valor de retorno de la función son los adecuados.

Ejemplo: `entero suma(entero, entero)`

Se está declarando la función *suma*, que tiene que recibir dos valores enteros cuando sea llamada para devolver un valor entero. No se dice en ningún momento lo que se hace con los valores enteros que recibe, sólo se declara las características de la función *suma*.

En el lenguaje C, todos los prototipos de las funciones de biblioteca se encuentran situados en los archivos de cabecera. Archivos cuyas extensiones son ".h" y que habrá que especificarlos en el programa .

2.8.2 Definición de funciones.

Cuando se define una función, hay que empezar colocando una línea llamada **cabecera de la función**. La cabecera especifica el tipo de valor retornado, el nombre de la función y el número y tipo de los parámetros formales a utilizar dentro de la función. Los parámetros formales, son variables locales a la función , que reciben los valores de los argumentos cuando se llama a la función.

Después de indicar la cabecera, se colocan las líneas que componen el grueso de la función.

```
tipo_de_retorno nombre_función(parámetros formales)
Entorno:
.....
Algoritmo:
.....
    retorna expresión
Fin nombre_función
```

Las variables declaradas en el entorno de la función , al igual que los parámetros formales, tienen carácter local. El resultado de la función se realiza mediante la sentencia **retorna expresión**. El sistema evalúa la expresión y devuelve dicho valor.

Puede que una función tenga varias sentencias retorna. También puede que no devuelva ningún valor, por lo que sólo se utiliza la palabra **retorna**. Si en la función no aparece ninguna sentencia retorna, normalmente casi todos los lenguajes interpretan que se vuelve al programa que hizo la llamada cuando termina de ejecutarse la última instrucción existente en la función. En algunos lenguajes, aquellas funciones que no tienen valor de retorno de denominan *procedimientos*.

Ejemplo: Definición de la función *suma*

```
entero suma(entero aa, entero bb)
Entorno:
    s: entero
Algoritmo:
    s = aa + bb
    retorna s
Fin suma
```

Los parámetros formales *aa* y *bb* son las variables que recibirán los valores a sumar cuando se invoque o llame a la función *suma*. La variable *s*, al igual que los parámetros formales *aa* y *bb* son variables locales a la función, y sólo son conocidas en este ámbito.

2.8.3 Llamadas a funciones.

Cuando se necesite ejecutar una función dentro de un programa, necesitamos llamarla o invocarla de la forma:

```
nombre_de_función (parámetros actuales)
```

La llamada a una función se realiza colocando su nombre, seguida de una lista de argumentos entre paréntesis llamados **parámetros actuales**. Los parámetros actuales son los valores que se pasan a la función en el momento de la llamada.

Ejemplo:

```
PROGRAMA: suma_numeros
ENTORNO:
    entero suma(entero, entero)    // prototipo de la función suma
    a, b, resul: entero
ALGORITMO:
    ESCRIBIR "Introduzca un número entero"
    LEER a
    ESCRIBIR "Introduzca otro entero"
    LEER b
    resul = suma(a, b)             // llamada a la función suma
    ESCRIBIR "La suma de " a, " y ", b, " es: ", resul
FIN suma-numeros.
```

El siguiente esquema de memoria aclara los contenidos de las variables cuando se invoca a la función *suma*.

| | | | | | | |
|-------------------------|------|------|-------|------|------|------|
| Direcciones de memoria: | 1498 | 1500 | 1502 | 1504 | 1506 | 1508 |
| | 3 | 4 | 7 | 3 | 4 | 7 |
| Variables de memoria | a | b | resul | aa | bb | s |