**UNIVERSIDAD TECNOLÓGICA NACIONAL**
**INSTITUTO NACIONAL SUPERIOR**
**DEL**
**PROFESORADO TÉCNICO**

# Tecnicatura Superior en Informática Aplicada

# PROGRAMACIÓN I

## MÓDULO I

**Mg. Ing. Patricia Calvo**
**Mg. Lic. Mónica Hencek**

# Resolución De Problemas Y Algoritmos

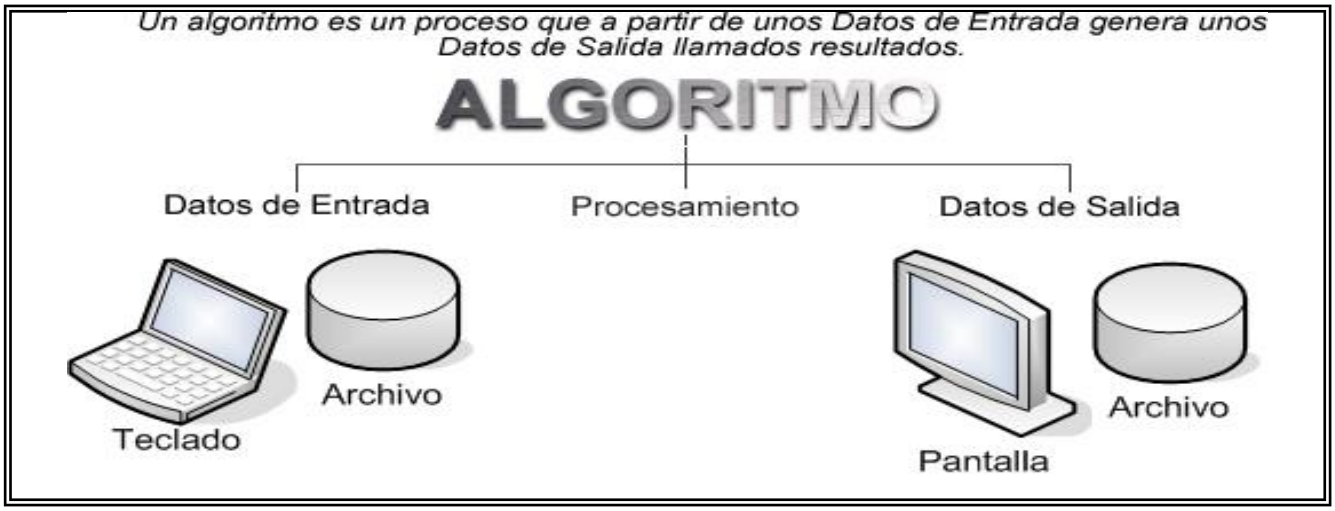
*La principal razón para que las personas aprendan lenguajes de programación es utilizar una computadora como una **herramienta para la resolución de problemas**. En el proceso pueden identificarse cinco fases:*

1. **Análisis del problema:** primer paso para encontrar la solución a un problema es el análisis del mismo.
2. **Estudio de su solución:** Se debe examinar cuidadosamente el problema a fin de obtener una idea clara sobre lo que se solicita y determinar los datos necesarios para conseguirlo.
3. **Diseño del Algoritmo:** Un algoritmo puede ser definido como la secuencia ordenada de pasos, sin ambigüedades, que conducen a la resolución de un problema dado y expresado en lenguaje natural, por ejemplo el castellano. Todo algoritmo **debe ser**:
   - **Preciso**: Indicando el orden de realización de cada uno de los pasos.
   - **Definido**: Si se sigue el algoritmo varias veces proporcionándole (consistente) los mismos datos, se deben obtener siempre los mismos resultados.
   - **Finito**: Al seguir el algoritmo, este debe terminar en algún momento, es decir tener un número finito de pasos.

     **Codificación del programa:** en un lenguaje de programación.

     **Depuración y prueba:** verificación, corrección de errores y ejecución.

**TIPS:** *En un algoritmo se deben de considerar tres partes: **Entrada:** Información dada al algoritmo. **Proceso:** Operaciones o cálculos necesarios para encontrar la solución del problema. **Salida:** Respuestas dadas por el algoritmo o resultados finales de los procesos realizados. Es aconsejable utilizar antes de la codificación: Diagrama de flujos o pseudolenguaje.*




**Ejemplo de algoritmo**: Supongamos que desea desarrollar un algoritmo que calcule la superficie de un rectángulo proporcionándole su base y altura. Lo primero que debemos hacer es plantearnos las siguientes preguntas:

| Especificaciones de entrada: | Especificaciones de salida: | El algoritmo que podemos utilizar es el siguiente: |
|---|---|---|
| • ¿Que datos son de entrada?<br>• ¿Cuántos datos se introducirán?<br>• ¿Cuántos son datos de entrada válidos? | • ¿Cuáles son los datos de salida?<br>• ¿Cuántos datos de salida se producirán?<br>• ¿Qué formato y precisión tendrán los resultados? | Paso 1. Entrada desde el teclado, de los datos de base y altura.<br>Paso 2. Cálculo de la superficie, multiplicando la base por la altura.<br>Paso 3. Salida por pantalla de base, altura y superficie calculada. |

**Ejemplos Intuitivos De Algoritmos**

Problema: calcular el factorial de un número utilizando la calculadora estándar de Windows.

- ☒ Fases 1 y 2: Análisis y estudio de la solución: *tomamos como ejemplo el número 5, y sabemos que el factorial de 0 (cero) es 1, → 5*4=*3=*2=*1=*1.*
- ☒ Fase 3: Diseño del algoritmo:

1. *Encender la calculadora (on).*
2. *Pulsar el número 5.*
3. *Pulsar la tecla *.*
4. *Pulsar el número 4.*
5. *Pulsar la tecla =*
6. *Aparece en el display el número 20*
7. *Pulsar la tecla *.*
8. *Pulsar el número 3.*
9. *Pulsar la tecla =*
10. *Aparece en el display el número 60*
11. *Pulsar la tecla *.*
12. *Pulsar el número 2.*
13. *Pulsar la tecla =*
14. *Aparece en el display el número 120*
15. *Pulsar la tecla *.*
16. *Pulsar el número 1.*
17. *Pulsar la tecla =*
18. *Aparece en el display el número 120*
19. *Pulsar la tecla *.*
20. *Pulsar el número 1.*
21. *Pulsar la tecla =*
22. *Aparece en el display el número 20*
23. *Apagar la calculadora (Off).*

**Otro Ejemplo**

Problema: Preparar un café con leche.

- ☒ Fases 1 y 2: Análisis y estudio de la solución: *tenemos una taza vacía, podemos cargarla con café hasta dónde nos parezca bien (hasta dónde?), agregarle azúcar (cuánta?) y leche (cuánta?).*
- ☒ Fase 3: Diseño del algoritmo:

1. *Tomar una taza.*
2. *Agregarle azúcar.*
3. *Agregarle café*
4. *Agregarle leche.*
5. *Agitar*

***No le parece muy vago este algoritmo?, vamos por otro…***

1. *Tomar una taza vacía.*
2. *Agregarle 3 cucharaditas de azúcar.*
3. *Agregarle café hasta la mitad de la taza.*
4. *Agregarle leche hasta llenar la taza.*
5. *Agitar 10 segundos el contenido*

***Sigue siendo muy vago, no es cierto?, vamos a refinarlo más…***

1. *Tomar una taza vacía.*
2. *Sacar la azucarera de la alacena.*
3. *Abrir la azucarera.*
4. *Tomar una cucharita.*
5. *Repetir*
   a. *Cargar la cucharita con azúcar.*
   b. *Echarla a la taza.*
6. *Hasta llegar a 3.*
7. *Guardar la azucarera.*
8. *Agregarle café hasta la mitad de la taza.*
9. *Agregarle leche hasta llenar la taza.*
10. *Repetir*
    a. *Agitar el contenido*
11. *Hasta llegar a 10 segundos.*

***Qué le parece ahora?...***

**Algoritmo…** Dado un **procesador** y un **entorno** bien definido, es el enunciado de una secuencia finita de **acciones primitivas** que resuelven determinado problema.

**Dónde..**

**Procesador es…** *Toda entidad capaz de comprender un enunciado y ejecutar el trabajo indicado en el mismo.Puede ser una persona que entienda las tareas descriptas y cuente con los elementos necesarios para ejecutarlas. Pero no puede realizar el trabajo demandado si no cuenta con los recursos necesarios.*

**Ambiente o entorno es…** *El conjunto de todos los recursos necesarios para la ejecución de un trabajo. Por ejemplo números y operadores en el primer caso y taza, café, azúcar,etc, en el segundo caso.*

**Acción es…** *Un evento que modifica el ambiente o entorno, lo transforma desde el estado inicia, a otro que puede ser un estado intermedio o el estado final. Para los ejemplos dados son los pasos que se llevan a cabo. Deben ser ejecutadas en orden.*

**Primitiva es…** *Para un procesador dado, una acción es primitiva si su enunciado es suficiente, para que pueda ejecutarla sin información adicional. Una acción no primitiva debe ser descompuesta en acciones primitivas para un procesador dado.*

| | |
|---|---|
| 1. Hacer la división decimal exacta de 1 entre 3.<br>2. Imprimir | 1. Hacer la división decimal exacta de 1 entre 3 hasta obtener un resultado con 2 cifras decimales.<br>2. Imprimir |
| Ejemplo incorrecto pués la primera acción no tiene fin. No es un algoritmo. | Ejemplo correcto pués la primera acción tiene fin. Constituye un algoritmo. |

La palabra algoritmo tiene su origen de un matemático árabe, Al-Khwarizmi. Deben considerarse 3 aspectos: Primitivas de las que partimos, lenguaje simbólico a utilizar y representación de los datos.

**Hay problemas y problemas…**

Ahora bien, hay que tener en cuenta que para resolver un problema, puede existir más de una forma y su correspondiente algoritmo. Por ejemplo para trasladarnos hasta aquí pueden existir muchos caminos, medios de transporte, horarios, etc. pero sin profundizar mucho sabemos que no es posible hallar una solución computacional para tal fin, ¿no es cierto? ¿Y si se nos presenta el problema de sumar nuestros gastos mensuales?, básicamente podríamos acudir al lápiz y el papel, o quizás utilizar una calculadora, incluso hacerlo en forma mental, pero también intuimos que una suma de números tiene solución computacional, ¿no es cierto?

**¿Por qué deseamos soluciones computacionales?**

Por rapidez, eficacia, precisión, economía de tiempos y esfuerzos, porque se puede manejar gran cantidad de datos, etc, hay muchas razones. Cuando decidimos una solución computacional tenemos que construir un programa basado en un algoritmo, que pueda ser ejecutado por un procesador, que en este caso sería un conjunto de circuitos electrónicos, y que realice un determinado trabajo para obtener el resultado deseado. Entonces podemos decir que la programación está constituida por dos fases: 1) resolución del problema propuesto con la determinación de un algoritmo, y 2) la adaptación del algoritmo a la computadora.

**Elementos De Programación - Representación de Algoritmos:**

Existen dos herramientas de programación usadas como Lenguajes algorítmicos, para representar un algoritmo: Diagramas de Flujo y Ppseudocódigo. El lenguaje algorítmico debe ser independiente de cualquier lenguaje de programación particular, pero fácilmente traducible a cada uno de ellos. Alcanzar estos objetivos conducirá al empleo de métodos normalizados para la representación de algoritmos.

**Pseudocódigo**

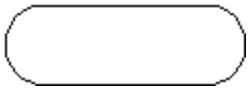
La **formalización de las acciones** se realiza a través de la estructura del algoritmo en pseudocódigo. El pseudocódigo se considera una herramienta para el diseño que permite obtener una solución mediante aproximaciones sucesivas. Se denomina notación de pseudocódigo a aquella que permite describir la solución de un problema en forma de algoritmo dirigido al computador utilizando palabras y frases del lenguaje natural sujetas a determinadas reglas.

| **Ejemplo 1:** | **Ejemplo 2:** |
|---|---|
| {Suma el entero 5 y el entero 7 }<br><br>principal<br>comienza<br>int a ←5 //en la variable a se almacena el valor de 5<br>int b ←7 //en la variable b se almacena el valor de 7<br>int c ←a + b //en c se almacena la suma de a con b<br>termina | {suma valores ingresados por teclado}<br><br>principal<br>comienza<br>int a, b, c<br>escribe “De un valor entero”<br>lee a<br>escribe “de otro valor entero”<br>lee b<br>c ← a + b<br>escribe “La suma de”<br>escribe a<br>escribe “con”<br>escribe b<br>escribe “es”<br>escribe c<br>termina |

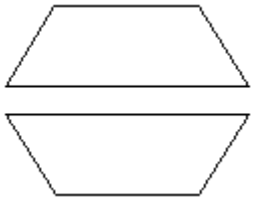
## Diagramas De Flujo

También conocidos como flowchart, en estos se utilizan símbolos estándar, en el que cada paso para la elaboración del programa se representa con el símbolo y orden adecuados, unidos o conectados por flechas, también llamadas líneas de flujo, esto por que indican el sentido en el que se mueve el proceso. En resumen el diagrama de flujo es un medio de presentación visual y gráfica del flujo de datos a través de un algoritmo.

### Bloques terminales



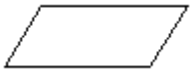
**Bloques de inicio y fin de programa:** Indican los límites del procedimiento considerado como principal. Generalmente se trata de un programa completo o de un módulo funcionalmente autónomo.



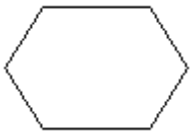
**Bloques de inicio y fin de procedimiento:** Indican los límites de un procedimiento considerado como una parte dependiente de otro mayor. Delimitan la explosión de un grupo de acciones que han sido consideradas como un procedimiento en otra parte del diagrama. Generalmente se trata de una función que hace una tarea específica.

### Bloques de acción

**Bloque de acción simple**: Representa una acción sencilla que puede ser considerada como única y que generalmente se codifica con una sola instrucción. Por ejemplo: incrementar contador, ubicar cursor, abrir archivo, etc.



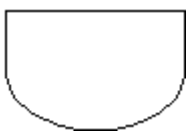
**Bloque de entrada/salida:** Representa una acción simple de entrada o salida de datos, generalmente desde o hacia un dispositivo periférico como el teclado, la pantalla o el disco. Por ejemplo: ingresar valor, leer registro, mostrar resultado, etc.



**Bloque de procedimiento**: Representa un conjunto de acciones que se consideran juntas, sin analizar su detalle. Este grupo de acciones se describe generalmente como procedimiento en otra parte del diagrama. Por ejemplo: buscar elemento, ordenar conjunto, procesar dato, etc.

### Bloques de decisión

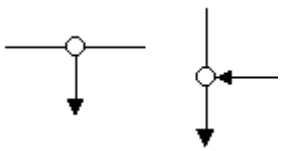
**Bloque de decisión simple**: Representa la acción de analizar el valor de verdad de una condición, que sólo puede ser verdadera o falsa (selección simple). Según el resultado de esta evaluación se sigue uno u otro curso de acción. Por lo tanto, de un bloque de decisión simple siempre salen exactamente dos flujos, uno por V (sí) y otro por F (no).



**Bloque de decisión múltiple**: Representa la acción de analizar el valor de una variable, que puede tomar uno entre una serie de valores conocidos (selección múltiple). Según el resultado de esta evaluación, se sigue uno entre varios cursos de acción. Por lo tanto, de un bloque de decisión múltiple siempre salen varios flujos, uno por cada valor esperado de la variable analizada.

### Flujos y conectores

**Flecha o flujo**: Indica la secuencia en que se van ejecutando las acciones al pasar de un bloque a otro.



**Conector**: Indica la convergencia de dos o más flujos. En la práctica determina el comienzo o el fin de una estructura.

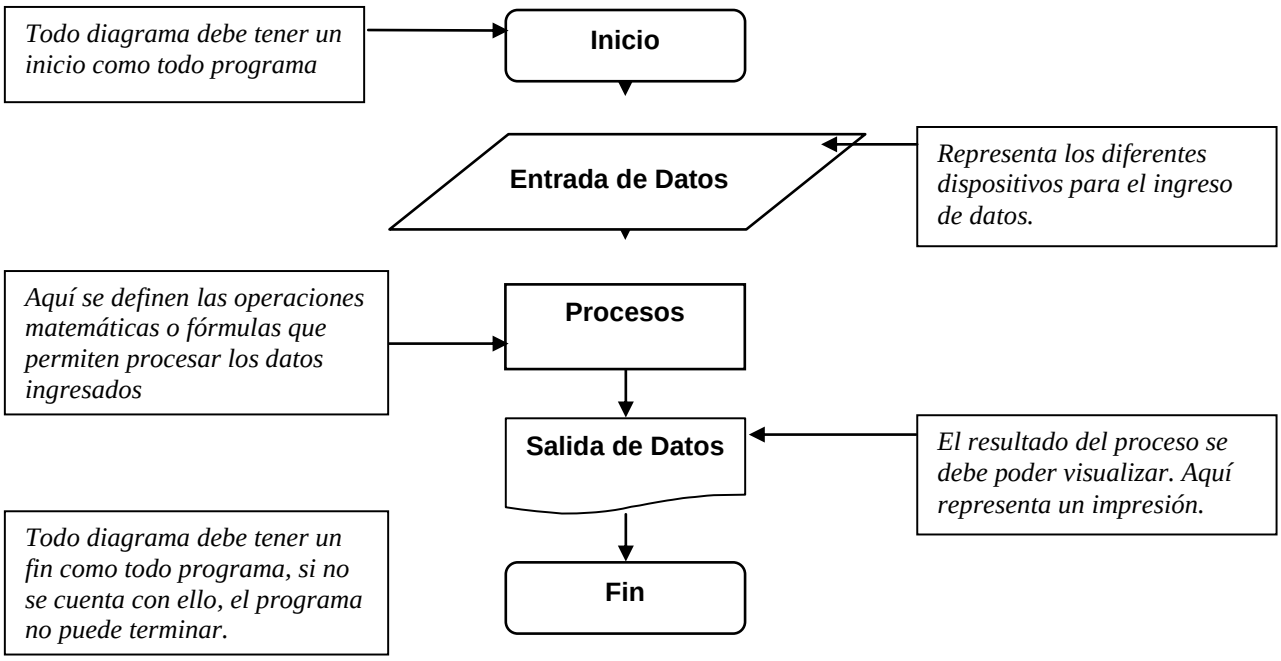
Veamos a continuación un conjunto de reglas o normas que nos permiten construir un diagrama de flujo.
1. Todo diagrama de flujo debe tener un inicio y un fin.

2. Las líneas utilizadas para indicar la dirección del diagrama deben ser rectas, horizontales o verticales, nunca se deben cruzar entre si.
3. No deben haber líneas sin conexión a los demás elementos del diagrama de flujo.
4. Un diagrama de flujo se debe construir de arriba hacia abajo y de requerirse de izquierda a derecha.
5. La notación o símbolos utilizados en el diagrama de flujo son independientes del lenguajes de programación utilizado para la elaboración del programa o aplicación.
6. No puede llegar más de una línea de conexión a un símbolo.

**TIPS:**

*En programación utilizamos las llamadas expresiones lógicas, que están constituidas por: números, constantes o variables y operadores lógicos o relacionales, que pueden tomar un valor falso o verdadero, dependiendo del resultado de la evaluación, para seguir por un camino determinado. Los operadores relacionales son aquellos que permiten la comparación y los operadores lógicos son los que permiten formular condiciones.*

**Estructura Básica De Un Diagrama De Flujo**




## Introducción

La computadora es una herramienta poderosa. La información (datos de entrada) puede almacenarse es la memoria y manejarse a velocidades excepcionalmente altas para producir resultados (salida del programa).

Podemos describirle a la computadora una tarea de manejo de datos presentándole una lista de instrucciones (llamada programa) que deben ser llevadas a cabo. Una vez que esta lista le ha sido proporcionada, esta puede llevar a cabo (ejecutar) dichas instrucciones. A elaborar una lista de instrucciones, o sea escribir un programa, se le llama programación. Escribir un programa de computadora es muy similar a describirle las reglas de un juego a gente que nunca lo ha jugado, para que las aplique. En ambos casos se requiere de un lenguaje de descripción intangible por todas las partes involucradas en la comunicación. Por ejemplo, las reglas del juego deben ser escritas en algún lenguaje y después se leen y se aplican. Los lenguajes utilizados para la comunicación entre el hombre y la computadora se llaman lenguajes de programación. Todas las instrucciones presentadas deben ser representadas y combinadas (para formar un programa) de acuerdo con las reglas de sintaxis (gramática) del lenguaje de programación. Sin embargo, hay una diferencia significativa entre el lenguaje de programación y un lenguaje como el español, el inglés o el ruso: **las reglas de un lenguaje de programación son muy precisas y no se permiten excepciones o ambigüedades**.

**Niveles de los lenguajes**

| Lenguaje de Máquina: | Lenguaje de Bajo Nivel: | Lenguaje de Alto Nivel: |
|---|---|---|
| *Directamente interpretable por un circuito microprogramable, como el microprocesador de una computadora o el microcontrolador de un autómata (un PLC). Los circuitos microprogramables son sistemas digitales, lo que significa que trabajan con dos únicos niveles de tensión. Dichos niveles, por abstracción, se simbolizan con el cero, 0, y el uno, 1, por eso el lenguaje de máquina sólo utiliza dichos signos. Esto permite el empleo de las teorías del álgebra booleana y del sistema binario en el diseño de este tipo de circuitos y en su programación. El lenguaje de programación de primera generación (por sus siglas en inglés, 1GL), es el lenguaje de código máquina. Es el único lenguaje que un microprocesador entiende de forma nativa. El lenguaje máquina no puede ser escrito o leído usando un editor de texto, y por lo tanto es raro que una persona lo use directamente.* | *Es el que proporciona poca o ninguna abstracción del microprocesador. Es fácilmente trasladado a lenguaje de máquina. La palabra "bajo" no implica que el lenguaje sea inferior a un lenguaje de alto nivel; se refiere a la reducida abstracción entre el lenguaje y el hardware. El lenguaje de programación de segunda generación (por sus siglas en inglés, 2GL), es el lenguaje ensamblador. Se considera de segunda generación porque, aunque no es lenguaje nativo del microprocesador, un programador de lenguaje ensamblador debe conocer la arquitectura del microprocesador (como por ejemplo las particularidades de sus registros o su conjunto de instrucciones).* | *Se caracterizan por expresar los algoritmos de una manera adecuada a la capacidad cognitiva humana, en lugar de a la capacidad ejecutora de las máquinas. En los primeros lenguajes de alto nivel la limitación era que se orientaban a un área específica y sus instrucciones requerían de una sintaxis predefinida. Su función principal radica en que a partir de su desarrollo, existe la posibilidad de que se pueda utilizar el mismo programa en distintas máquinas, es decir que es independiente de un hardware determinado. La única condición es que la PC tenga un programa conocido como traductor o compilador, que lo traduce al lenguaje específico de cada máquina. Existe gran diversidad de ellos (C, PASCAL, BASIC, FORTRAN, C++, VISUAL BASIC, COBOL, ALGOL, entre muchos otros).* |

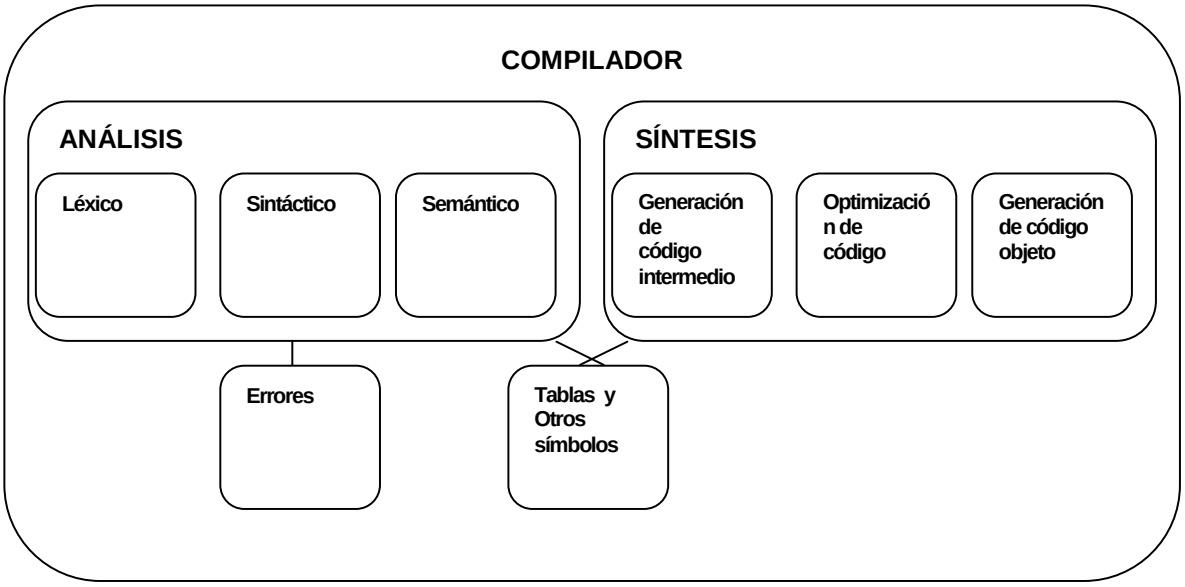
**Diagrama representativo de las instrucciones**


**Traduccion Entre Lenguajes**

Es necesario que la computadora sea capaz de ejecutar un programa expresado en un lenguaje de alto nivel.

- COMPILADOR: Traduce un programa escrito en un lenguaje de alto nivel a su equivalente en lenguaje máquina.
- INTÉRPRETE: Hace posible que la computadora sea capaz de ejecutar directamente un programa escrito en un lenguaje de alto nivel.

**Esquema de un compilador**




ANÁLISIS LEXICOGRÁFICO

- ☒ Agrupa los símbolos del programa fuente en unidades léxicas denominadas tokens.
- ☒ Elimina los comentarios.
- ☒ Elimina los espacios en blanco, retornos de carro, tabuladores, etc.
- ☒ Agrega los identificadores a la tabla de símbolos.
- ☒ Avisa de los errores léxicos que detecte.

ANÁLISIS SINTÁTICO

- ☒ Crea un árbol sintáctico a partir de la secuencia de tokens creados en la fase de análisis léxico.
- ☒ El árbol sintáctico sirve como base para el análisis semántico y la generación de código.
- ☒ Avisa de los errores sintácticos que detecte.

ANÁLISIS SEMÁNTICO

- ☒ Comprueba que una frase sintácticamente correcta lo es también semánticamente.
- ☒ Ejemplo asignación de valores de distintos tipos.
- ☒ Ejemplo aplicación de operadores a tipos apropiados.
- ☒ Avisa de los errores semánticos que detecte.

GENERACIÓN DE CÓDIGO INTERMEDIO

- ☒ Código independiente de la máquina para la que se hace el compilador.
- ☒ Debe de ser fácil de producir a partir del análisis y fácil de traducir al código definitivo.

OPTIMIZACIÓN DE CÓDIGO

- ☒ Genera un código más compacto y eficiente.
- ☒ Elimina las acciones redundantes e innecesarias.

GENERACIÓN DE CÓDIGO OBJETO

- ☒ Genera el código objeto final a partir del optimizado.

ENLAZADO (LINK)

- ☒ Enlazado de los diferentes códigos objeto que componen los módulos del programa para generar código ejecutable.

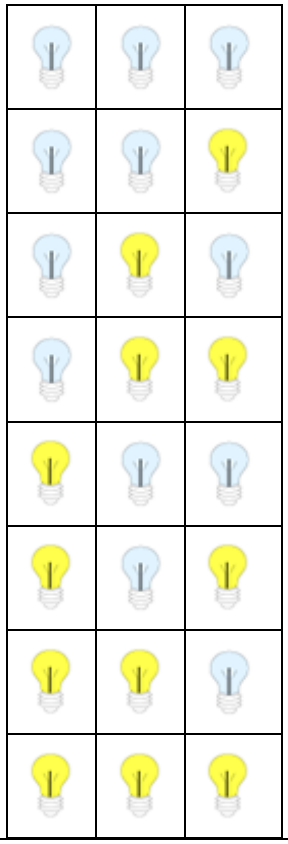
## Codificación de la Información

BIT: es la unidad mínima de información (2 estados diferentes)

| | |
|---|---|
|  |  |
| *apagado* | *encendido* |

2 BITs representan 4 ($2^2$) estados diferentes

3 BITs representan 8 ($2^3$) estados diferentes



BYTE 8 bits, representa 256 ($2^8$) estados diferentes

Unidad mínima de información

| | | | |
|---|---|---|---|
|  | | | |
| 0 | 1 | 1 | 0 |
|  | | | |
| 1 | 0 | 1 | 1 |

## Entrada Y Salida De Información - Códigos de entrada / salida

### Caracteres:

- ☒ Letras Mayúsculas: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
- ☒ Letras Minúsculas: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
- ☒ Dígitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- ☒ Símbolos de puntuación: , ; . : ¿ ? ¡ ! ( ) [ ] { }
- ☒ Otros símbolos: = + - / * > < # $ % & _ @
- ☒ Caracteres de control: Se representan numéricamente según una tabla de correspondencia

### Ejemplos

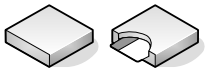
| **Entrada/Salida de Caracter** | | | **Entrada/Salida de Números (mediante caracteres)** | | |
|---|---|---|---|---|---|
| **Entrada** | **Representación Interna del caracter** | **Salida** | **Entrada** | **Representación interna numérica** | **Salida** |
| Pulso tecla a | 65 | Se ve en pantalla: A | Pulso tecla 6 y tecla 5 | 65 | Se ve en pantalla: 65 |

## Tabla Ascii

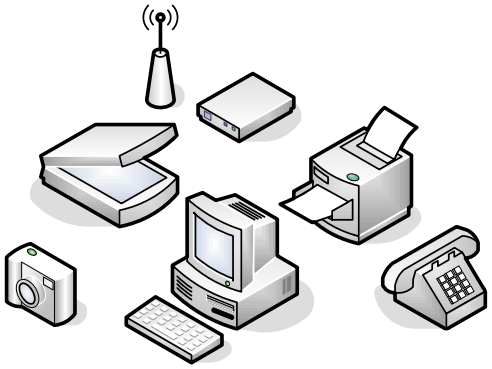
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | 16 | | 32 | | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p | 128 | € | 144 | • | 160 | | 176 | ° | 192 | À | 208 | Ð | 224 | à | 240 | ð |
| 1 | | 17 | | 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q | 129 | • | 145 | ‘ | 161 | ¡ | 177 | ± | 193 | Á | 209 | Ñ | 225 | á | 241 | ñ |
| 2 | | 18 | | 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r | 130 | , | 146 | ’ | 162 | ¢ | 178 | ² | 194 | Â | 210 | Ò | 226 | â | 242 | ò |
| 3 | | 19 | | 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s | 131 | ƒ | 147 | “ | 163 | £ | 179 | ³ | 195 | Ã | 211 | Ó | 227 | ã | 243 | ó |
| 4 | | 20 | µ | 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t | 132 | „ | 148 | ” | 164 | ¤ | 180 | ´ | 196 | Ä | 212 | Ô | 228 | ä | 244 | ô |
| 5 | | 21 | § | 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u | 133 | … | 149 | • | 165 | ¥ | 181 | µ | 197 | Å | 213 | Õ | 229 | å | 245 | õ |
| 6 | | 22 | | 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v | 134 | † | 150 | – | 166 | ¦ | 182 | ¶ | 198 | Æ | 214 | Ö | 230 | æ | 246 | ö |
| 7 | | 23 | | 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w | 135 | ‡ | 151 | — | 167 | § | 183 | · | 199 | Ç | 215 | × | 231 | ç | 247 | ÷ |
| 8 | | 24 | | 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x | 136 | ˆ | 152 | ˜ | 168 | ¨ | 184 | ¸ | 200 | È | 216 | Ø | 232 | è | 248 | ø |
| 9 | | 25 | | 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y | 137 | ‰ | 153 | ™ | 169 | © | 185 | ¹ | 201 | É | 217 | Ù | 233 | é | 249 | ù |
| 10 | | 26 | | 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z | 138 | Š | 154 | š | 170 | ª | 186 | º | 202 | Ê | 218 | Ú | 234 | ê | 250 | ú |
| 11 | | 27 | | 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { | 139 | ‹ | 155 | › | 171 | « | 187 | » | 203 | Ë | 219 | Û | 235 | ë | 251 | û |
| 12 | | 28 | | 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| | 140 | Œ | 156 | œ | 172 | ¬ | 188 | ¼ | 204 | Ì | 220 | Ü | 236 | ì | 252 | ü |
| 13 | | 29 | | 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } | 141 | | 157 | | 173 | | 189 | ½ | 205 | Í | 221 | Ý | 237 | í | 253 | ý |
| 14 | | 30 | - | 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ | 142 | Ž | 158 | ž | 174 | ® | 190 | ¾ | 206 | Î | 222 | Þ | 238 | î | 254 | þ |
| 15 | ¤ | 31 | | 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | □ | 143 | | 159 | Ÿ | 175 | ¯ | 191 | ¿ | 207 | Ï | 223 | ß | 239 | ï | 255 | ÿ |

## Estructura General De Una Computadora

Medios de almacenamiento

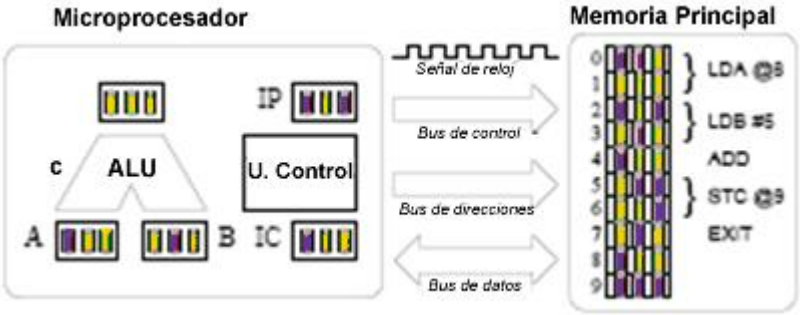


Periféricos



## Estructura Funcional De Las Computadoras

## Representación, almacenamiento y tratamiento de la información en un sistema informático



## Visión General De Un Sistema Informático


## Lenguajes De Programación

**TIPS:**

*Los lenguajes de programación son sólo herramientas. Aunque son necesarios, debes concentrarte más en aprender técnicas y metodologías de diseño y programación.*

Los lenguajes de programación sirven para escribir programas que permitan la comunicación hombre-máquina y presentan las siguientes diferencias esenciales con el lenguaje natural:

1. Tienen un vocabulario y una sintaxis muy limitados, lo cual implica que los programas sólo pueden describir algoritmos y son inadecuados para describir otros tipos no algorítmicos de prosa.
2. Los objetos se declaran y las acciones se describen. Todo objeto referenciado en alguna parte de un programa debe haber sido definido previamente en el área reservada a las declaraciones.
3. El vocabulario de un lenguaje de programación contiene sólo aquellos tipos de acciones básicas que una computadora puede entender y realizar y no otras, por ejemplo, un lenguaje de programación soporta las cuatro operaciones fundamentales, comparación, acciones propias del procesamiento de textos, de gráficos y acciones de entrada y salida. Las acciones que una computadora no puede realizar son muchas y variadas, por ejemplo, crear pinturas al óleo es una acción que no se encuentra dentro del vocabulario de una computadora.
4. La sintaxis de un lenguaje de programación es muy rígida y no permite ninguna variación de estilo, por ejemplo, el cálculo del cociente entre dos números se expresa como n1/n2 y no existe forma alternativa.

## Tipos De Programación

Secuencial: Las acciones se ejecutan una tras otra, en un sentido estricto, esto es una acción posterior no se inicia hasta que se haya completado la anterior. Ejemplo: Basic, Cobol.

Estructurada: Es la escritura de programas de manera que el programa tiene un diseño modular: el código del programa está dividido en módulos o segmentos de código que son ejecutados conforme sea requerido; los módulos son diseñados de modo descendente: los problemas se dividen de tal manera que las partes complejas del programa sean implementadas (realizadas) por módulos independientes que a su vez pueden invocar a otros módulos. Ejemplo: Turbo Pascal.

Orientada a Objetos: Estilo de programación que utiliza objetos como bloque esencial de construcción. Los objetos son en realidad como los tipos abstractos de datos. Un TAD es un tipo definido por el programador junto con un conjunto de operaciones que se pueden realizar sobre ellos. Se denominan abstractos para diferenciarlos de los tipos de datos fundamentales o básicos.

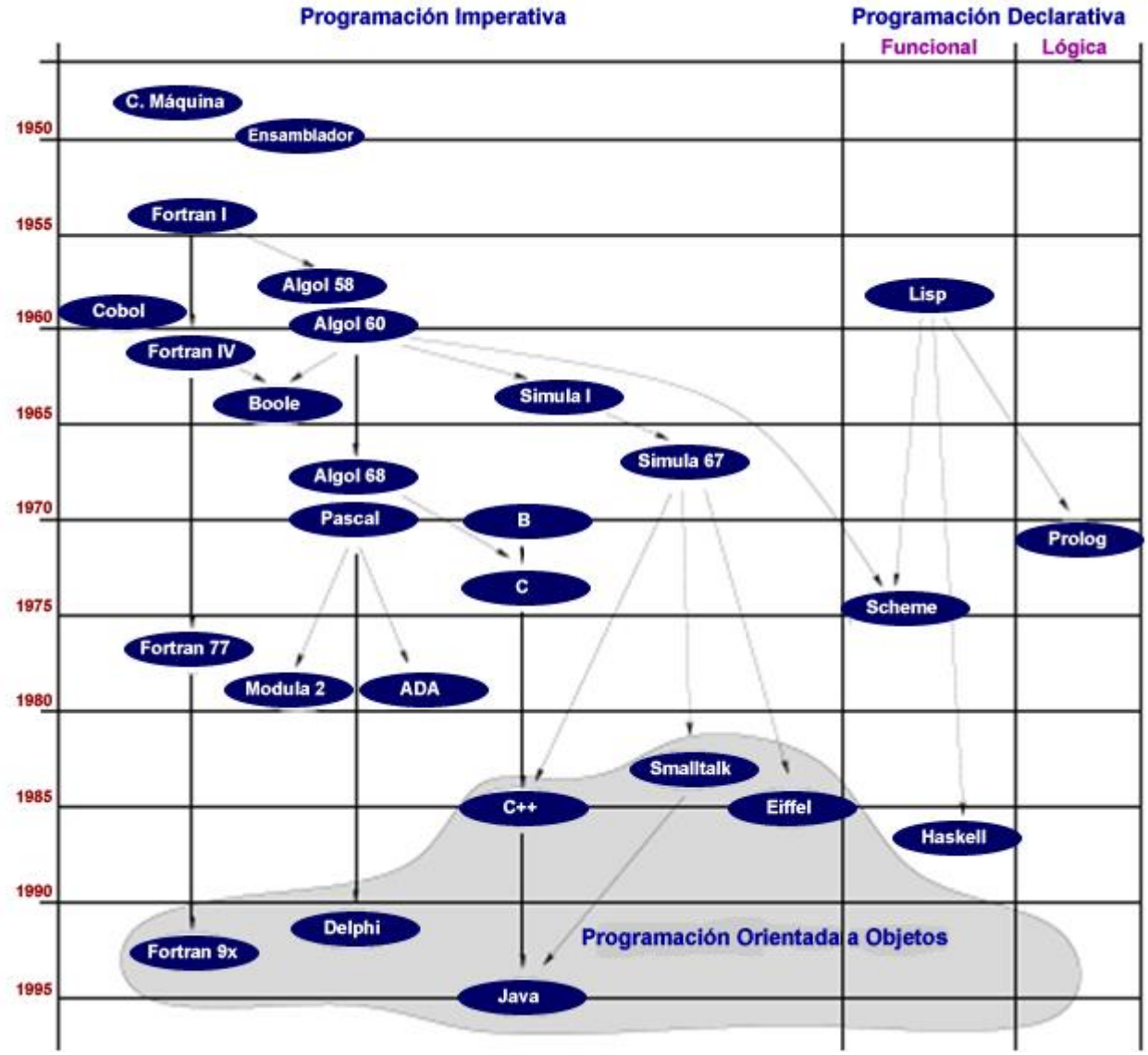
La orientación a objetos es un paradigma de programación que facilita la creación de software de calidad por sus factores que potencian el mantenimiento, la extensión y la reutilización del software generado bajo este paradigma. Trata de amoldarse al modo de pensar del hombre y no al de la máquina. Esto es posible gracias a la forma racional con la que se manejan las abstracciones que representan las entidades del dominio del problema, y a propiedades como la jerarquía o el encapsulamiento.

El elemento básico de este paradigma no es la función (elemento básico de la programación estructurada), sino un ente denominado objeto. Un objeto es la representación de un concepto para un programa, y contiene toda la información necesaria para abstraer dicho concepto: los datos que describen su estado y

las operaciones que pueden modificar dicho estado, y determinan las capacidades del objeto. Java incorpora el uso de la orientación a objetos como uno de los pilares básicos de su lenguaje.

**Clasificaciones**

| **Nivel de Abstracción** | | **Paradigma de Programación** | | **Propósito** | |
|---|---|---|---|---|---|
| **Bajo Nivel** | **Alto Nivel** | **Imperativo** | **Declarativo** | **Científicos** | Algol, Fortran, etc |
| *Programas expresados en términos de la arquitectura y operaciones básicas que realiza una computadora.* | *Programas expresados en términos del problema a resolver y en un lenguaje cercano al programador.* | *Programación estructurada, modular, orientada a objetos.* | *Funcional, Lógica* | | |
| | | | | **Ingeniería** | Ada, Dynamo, etc |
| | | | | **Gestión** | Cobol, dBase, |
| | | | | **Inteligencia Artificial** | Lisp, Prolog |
| *Lenguaje de máquina, ensamblador, etc* | *Lisp, Prolog, Java, C++, ADA, etc* | | | **Propósito General** | Modula-2, C++, Java |



## Representación: Sintaxis, Instrucción, Programa

Sintaxis: son reglas que deben seguirse en la escritura de cada parte de un programa.

Instrucción: forma de indicarle a la computadora que se necesita llevar a cabo. Las instrucciones se forman con los estatutos del lenguaje correspondiente y siguiendo las reglas de sintaxis que el mismo determine.

Programa: conjunto de instrucciones que indican a la computadora lo que se necesite se lleve a cabo. Es necesario que se especifiquen de acuerdo a la sintaxis del lenguaje y en el orden lógico apropiado.

## ¿Qué Significa Programar?

*Por programar se entiende un proceso mental complejo dividido en varias etapas. La finalidad de la programación es comprender con claridad el problema que va a resolverse o simularse por medio de la computadora, y entender también con detalle cuál es el procedimiento mediante el cual la máquina llegará a la solución deseada.*

*La codificación constituye una etapa necesariamente posterior a la programación y consiste en describir, en el lenguaje de programación adecuado, la solución ya encontrada o sugerida, por medio de la programación. Es decir, primero se programa la solución de un problema, y después hay que traducirla para que la computadora pueda entenderla.*

*Programar requiere de:*

- *Talento*
- *Creatividad*
- *Inteligencia*
- *Lógica*
- *Experiencia*
- *Habilidad para construir y analizar abstracciones*

*Dentro de la historia del software se han tenido presentes dos características:*

1. *Los sistemas son más grandes y complejos*
2. *La evolución de los lenguajes de programación*

*Esta evolución de la tecnología del software puede verse representada por una estructura de capas, en donde cada una de las capas continua siendo funcional. Cuando la computación se encontraba en su infancia, muchos programas fueron escritos en lenguajes ensamblador por un solo individuo. Cuando los programas empezaron a ser más complejos, los programadores encontraron difícil recordar toda la información necesaria para desarrollar o mantener los sistemas. La introducción de lenguajes de alto nivel resolvió algunas dificultades. Simultáneamente las expectativas de resolver problemas más complejos se incrementaron. Así, empezaron a formarse grupos de programadores que trabajaban juntos. Cuando esto paso, un fenómeno interesante fue observado: Una tarea que tomaba 2 meses a un programador no podía completarse en un mes por dos programadores. La razón de este comportamiento no lineal fue la complejidad. Se ha observado que esta complejidad se deriva de:*

- *Complejidad del dominio del problema*
- *Dificultad en el manejo del proceso de desarrollo*
- *Flexibilidad del software*

*Las formas que se han utilizado para tratar la complejidad de los sistemas son:*

- *Procedimientos*
- *Módulos*
- *Objetos*

**TIPS:**

*Aprender a programar es aprender a resolver problemas, no a usar un lenguaje de programación.*

## Buenas Prácticas De Programación

1. *Modularidad a través del diseño top-down*
2. *Modificable (uso de constantes)*
3. *Interfaz con el usuario*
    - *entrada interactiva (letreros)*
    - *salida*
        - *con datos de entrada*
        - *etiquetada*
4. *Programación libre de errores*
    - *de entrada*
    - *lógicos*

*Ventajas de la modularidad:*

- *Contrucción*
- *Depuración*
- *Fácil de leer*
- *Fácil de modificar*
- *Eliminación de código redundante*
- *Desarrollo independiente de módulos*

*Estilo:*

- *Extenso uso de módulos*
- *Evitar uso de variables globales*
- *En general las funciones NO deben:*
  - ♦ *asignar valores a variables globales*
  - ♦ *realizar funciones de E/S*
- *Manejo de errores*
- *Uso de nombres significativos de variables y módulos*
- *Documentación que pueda leerse, usarse y modificarse por otros*
  - ♦ *Documentación del programa*
  - ♦ *Documentación general*

*Documentación del programa:*

- *Comentario principal del programa debe contener:*
  - *propósito*
  - *autor y fecha*
  - *breve descripción de los algoritmos y estructuras de datos globales*
  - *descripción de cómo usar el programa*
  - *descripción de variables globales*
  - *comentarios acerca de qué tipo de datos espera el programa*
- *Miniencabezado en cada módulo similar al del programa*
- *Comentarios en el cuerpo de cada módulo explicando lo importante y confuso del programa*

*Documentación general:*

- *Diseño modular del programa*
- *Pseudocódigo*
- *Listados*
- *Manual de usuario*
- *Ejemplos*

**TIPS:**

*Cuando programes observa siempre la Regla No. 1 de la Informática: Realiza copias de seguridad. La práctica de esta simple norma puede salvarte de un virus, un fallo de hardware, un dispositivo de almacenamiento defectuoso y hasta de ti mismo cuando hagas una mala modificación al código o una acción con la que elimines información.*

**Recomendaciones generales para tener en cuanta al resolver problemas:**

- Comprenda a fondo el problema antes de delinear la solución.
- Establezca cuales son los datos, si los conoce o no, y si puede averiguarlos de algún modo.
- Analice las condiciones que deban ser tenidas en cuenta.
- No se lance sobre el teclado a escribir alocadamente; planee la solución.
- Para orientarse, considere las siguientes preguntas:
- ¿Es conocido el problema?
- Si es desconocido: ¿conoce la solución de alguno similar?
- ¿Puede resolver el problema en su totalidad?
- Si no es así: ¿Puede resolver parte del problema?
- Si no es así: ¿Puede resolverlo en distintas condiciones?
- Es importante que desarrolle todo lo que pueda de la solución y que determine con claridad cual es la dificultad que no puede salvar. Explicitarla, en muchos casos orienta hacia la solución de la misma.
- Desarrolle el plan de la solución (algoritmo), chequeando cada paso.
- Examine la solución en su conjunto. Recién entonces, escriba el código.

**Lenguaje C**

**HISTORIA**

El lenguaje C nació en los Laboratorios Bell de AT&T y ha sido estrechamente asociado con el Sistema Operativo UNIX, ya que su desarrollo se realizó en este sistema y debido a que tanto UNIX como el propio compilador de C y la casi totalidad de los programas y herramientas de UNIX, fueron escritos en C. Su eficacia y claridad han hecho que el lenguaje Assembler apenas haya sido utilizado en UNIX. Está inspirado en el lenguaje B escrito por Ken Thompson en 1970 con intención de recodificar el UNIX, que en la fase de arranque está escrito en Assembler, en vistas a su transportabilidad a otras máquinas. B era un lenguaje evolucionado e independiente de la máquina, inspirado en el lenguaje BCPL concedido por Martin Richard en 1967. En 1972, Dennis Ritchie, toma el relevo y modifica el lenguaje B, creando el lenguaje C y reescribiendo el UNIX en dicho lenguaje. La novedad que proporcionó el lenguaje C sobre el B fue el diseño de tipos y estructuras de datos. Con la popularidad de las microcomputadoras muchas compañías comenzaron a implementar su propio C por lo cual surgieron discrepancias entre sí. Por esta razón ANSI (American National Standars Institute, por sus siglas en inglés), estableció un comité en 1983 para crear una definición no ambigua del lenguaje C e

independiente de la máquina que pudiera utilizarse en todos los tipos de C. Algunos de las C existentes son: Quick C, C++, Turbo C, Turbo C ++, Borland C, Microsoft C, Visual C, C Builder.

## CARACTERÍSTICAS PRINCIPALES

- Una de las particularidades de C es su riqueza de operadores, puede decirse que prácticamente dispone de un operador para cada una de las posibles operaciones en código máquina.
- C es un lenguaje de programación de nivel medio ya que combina los elementos del lenguaje de alto nivel con la funcionalidad del ensamblador.
- Es estructurado, es decir, el programa se divide en módulos (funciones) independientes entre sí.
- Actualmente, debido a sus características, puede ser utilizado para todo tipo de programas.
- Ha sido pensado para ser altamente transportable y para programar lo improgramable,

## INCONVENIENTES

- Carece de instrucciones de entrada/salida, de instrucciones para manejo de cadenas de caracteres, con lo que este trabajo queda para la biblioteca de rutinas, con la consiguiente pérdida de transportabilidad
- La excesiva libertad en la escritura de los programas puede llevar a errores en la programación que, por ser correctos sintácticamente no se detectan a simple vista
- Por otra parte las precedencias de los operadores convierten a veces las expresiones en pequeños rompecabezas.
- A pesar de todo, C ha demostrado ser un lenguaje extremadamente eficaz y expresivo.

## OBJETIVO DE SU CREACIÓN

El lenguaje C inicialmente fue creado para la programación de:

- Sistemas operativos
- Intérpretes
- Editores
- Ensambladores
- Compiladores
- Administradores de bases de datos.

### Creación, edición, compilación y ejecución de un programa

### Creación del programa

Los programas C y C++ se escriben con la ayuda de un editor de textos del mismo modo que cualquier texto corriente. Los archivos que contiene programas en C o C++ en forma de texto se conocen como archivos fuente y el texto del programa que contiene se conoce como programa fuente. Nosotros **siempre** escribiremos programas fuente y los guardaremos en archivos fuente.

El contenido del archivo deberá obedecer la sintaxis de C.

Los programas fuente no pueden ejecutarse. Son archivos de texto, pensados para que los comprendan los seres humanos, pero incomprensibles para las computadoras.

### Compilación - Archivos objeto, código objeto y compiladores

Para conseguir un programa ejecutable hay que seguir algunos pasos. El primero es compilar o traducir el programa fuente a su código objeto equivalente. Este es el trabajo que hacen los compiladores de C y C++. Consiste en obtener un archivo equivalente a nuestro programa fuente comprensible para la computadora, este archivo se conoce como archivo objeto y su contenido como código objeto.

Los compiladores son programas que leen un archivo de texto que contiene el programa fuente y generan un archivo que contiene el código objeto.

El código objeto no tiene ningún significado para los seres humanos, al menos no directamente. Además es diferente para cada computadora y para cada sistema operativo. Por lo tanto existen diferentes compiladores para diferentes sistemas operativos y para cada tipo de computadora.

Existen muchos compiladores de C y en general todos los compiladores operan esencialmente de la misma forma y comparten muchas opciones comunes en la línea de opciones.

Si hay errores obvios en el programa (tales como palabras mal escritas, caracteres no tecleados u omisiones de punto y coma), el compilador se detendrá y los reportará.

Podría haber desde luego errores lógicos que el compilador no podrá detectar. En el caso que esta fuera la situación se le estará indicando a la computadora que haga las operaciones incorrectas.

**TIPS:**

*Cuando programes observa siempre la Regla No. 1 de la Informática: Realiza copias de seguridad. La práctica de esta simple norma puede salvarte de un virus, un fallo de hardware, un dispositivo de almacenamiento defectuoso y hasta de ti mismo cuando hagas una mala modificación al código o una acción con la que elimines información.*

## Ejecución del programa - Archivos ejecutables y enlazadores

Cuando obtenemos el archivo objeto, aún no hemos terminado el proceso. El archivo objeto, a pesar de ser comprensible para la computadora, no puede ser ejecutado. Hay varias razones para eso:

1. Nuestros programas usarán, en general, funciones que estarán incluidas en librerías externas, ya sean ANSI o no. Es necesario combinar nuestro archivo objeto con esas librerías para obtener un ejecutable.

2. Frecuentemente, nuestros programas estarán compuestos por varios archivos fuente, y de cada uno de ellos se obtendrá un archivo objeto. Es necesario unir todos los archivos objeto, más las librerías en un único archivo ejecutable.

3. Hay que dar ciertas instrucciones a la computadora para que cargue en memoria el programa y los datos, y para que organice la memoria de modo que se disponga de una pila de tamaño adecuado, etc. La pila es una zona de memoria que se usa para que el programa intercambie datos con otros programas o con otras partes del propio programa.

4. Existe un programa que hace todas estas cosas, se trata del "link", o enlazador. El enlazador toma todos los archivos objeto que componen nuestro programa, los combina con los archivos de librería que sea necesario y crea un archivo ejecutable.

Una vez terminada la fase de enlazado, ya podremos ejecutar nuestro programa. Se ejecuta el programa, mostrando algún resultado en la pantalla. En éste estado, podría haber errores en tiempo de ejecución (run-time errors), tales como división por cero, o bien, podrían hacerse evidentes al ver que el programa no produce la salida correcta. Si lo anterior sucede, entonces se debe regresar a editar el archivo del programa, recompilarlo, y ejecutarlo nuevamente.

## El modelo de compilación de C

En la figura se muestran las distintas etapas que cubre el compilador para obtener el código ejecutable.


## El preproceador

El preprocesador acepta el código fuente como entrada y es responsable de:

- quitar los comentarios
- interpretar las directivas del preprocesador las cuales inician con #.

Por ejemplo:

- #include -- incluye el contenido del archivo nombrado. Estos son usualmente llamados archivos de cabecera (header). Por ejemplo:
  - #include <math.h> -- Archivo de la biblioteca estándar de matemáticas.
  - #include <stdio.h> -- Archivo de la biblioteca estándar de Entrada/Salida.
- #define -- define un nombre simbólico o constante. Sustitución de macros.
  - #define TAM_MAX_ARREGLO 100

## Compilador de C

El compilador de C traduce el código fuente en código de ensamblador. El código fuente es recibido del preprocesador.

**TIPS:**

*Cuando compiles, activa todas las advertencias del compilador para verlas y evitar problemas posteriores. El corregirlas te permitirá tener el código más claro y con menos errores.*

## Enlazador

Si algún archivo fuente hace referencia a funciones de una biblioteca o de funciones que están definidas en otros archivos fuentes, el enlazador combina estas funciones (con main()) para crear un archivo ejecutable. Las referencias a variables externas en esta etapa son resueltas.

## Errores

Por supuesto, somos humanos, y por lo tanto nos equivocamos. Los errores de programación pueden clasificarse en varios tipos, dependiendo de la fase en que se presenten.

***Errores de sintaxis:*** son errores en el programa fuente. Pueden deberse a palabras reservadas mal escritas, expresiones erróneas o incompletas, variables que no existen, etc.Se detectan en la fase de compilación. El compilador, además de generar el código objeto, nos dará una lista de errores de sintaxis. De hecho nos dará sólo una cosa o la otra, ya que si hay errores no es posible generar un código objeto.

*Avisos:* además de errores, el compilador puede dar también avisos (warnings). Los avisos son errores, pero no lo suficientemente graves como para impedir la generación del código objeto. No obstante, es importante corregir estos avisos, ya que el compilador tiene que decidir entre varias opciones, y sus decisiones no tienen por qué coincidir con lo que nosotros pretendemos, se basan en las directivas que los creadores decidieron durante su creación.

***Errores de enlazado:*** el programa enlazador también puede encontrar errores. Normalmente se refieren a funciones que no están definidas en ninguno de los archivos objetos ni en las librerías. Puede que hayamos olvidado incluir alguna librería, o algún archivo objeto, o puede que hayamos olvidado definir alguna función o variable, o lo hayamos hecho mal.

***Errores de ejecución:*** después de obtener un archivo ejecutable, es posible que se produzcan errores. Normalmente no obtendremos mensajes de error, sino que simplemente el programa terminará bruscamente. Estos errores son más difíciles de detectar y corregir. Existen programas auxiliares para buscarlos, son los llamados depuradores (debuggers). Estos permiten detener la ejecución de nuestros programas, inspeccionar variables y ejecutarlo paso a paso.

***Errores de diseño:*** finalmente los errores más difíciles de corregir y prevenir. Si nos hemos equivocado al diseñar nuestro algoritmo, no habrá ningún programa que nos pueda ayudar a corregir los nuestros. Contra estos errores sólo cabe practicar y pensar.

## Uso de las bibliotecas

C es un lenguaje pequeño. Muchas de las funciones que tienen otros lenguajes no están en C, por ejemplo, no hay funciones para E/S, manejo de cadenas o funciones matemáticas.

La funcionalidad de C se obtiene a través de un rico conjunto de bibliotecas de funciones.

Las librerías o biblioteca de funciones, contienen el código objeto de muchos programas que permiten hacer cosas comunes, como leer el teclado, escribir en la pantalla, manejar números, realizar funciones matemáticas, etc. Las librerías están clasificadas por el tipo de trabajos que hacen, hay librerías de entrada y salida, matemáticas, de manejo de memoria, de manejo de textos, etc.

Hay un conjunto de librerías muy especiales, que se incluyen con todos los compiladores de C y de C++. Son las librerías ANSI o estándar. Pero también hay librerías no estándar y dentro de estas las hay públicas y comerciales.

Como resultado, muchas implementaciones de C incluyen bibliotecas estándar de funciones para varias finalidades. Para muchos propósitos básicos estas podrían ser consideradas como parte de C. Pero pueden variar de máquina a máquina.

Un programador puede también desarrollar sus propias funciones de biblioteca e incluso bibliotecas especiales de terceros.

**TIPS:**

*Los conocimientos y la experiencia que tengas con la sintaxis del lenguaje y sus herramientas, así como las facilidades que ofrece la interfaz del compilador son elementos clave para el éxito de la codificación.*

- Comandos del preprocesador.
- Definiciones de tipos.
- Prototipos de funciones - declara el tipo de función y las variables pasadas a la misma.
- Variables
- Funciones

Para un programa se debe tener una función main().

Una función tiene la forma:

```
tipo nombre_de_la_funcion (parámetros)
{
    variables locales
    sentencias de C
}
```

Si la definición del tipo es omitida, C asume que la función regresa un tipo entero.

A continuación se muestra un ejemplo:

```
/* Programa ejemplo */

main()
{
  printf( "Me gusta C\n" );
  return(0);
}
```

NOTAS:

- C requiere un punto y coma al final de cada sentencia.
- `printf` es una función estándar de C, la cual es llamada en la función `main()`.
- \n significa salto de línea. Salida formateada.
- `exit()` es también una función estándar que hace que el programa termine. En el sentido estricto no es necesario ya que es la última línea de `main()` y de cualquier forma terminará el programa.

En caso de que se hubiera llamado a la función printf de la siguiente forma:

```
printf(".\n.1\n..2\n...3\n");
```

La salida tendría la siguiente forma:

```
.1
..2
...3
```

## Un primer programa en C

Observa el siguiente programa:

```
int main()
{
  int numero;
  numero = 2 + 2;
  return 0;
}
```

Ahora veamos el programa línea a línea:

- Primera línea: "int main()"

Es el principio de la definición de una función. Todas las funciones C toman unos valores de entrada, llamados parámetros o argumentos, y devuelven un valor de retorno. La primera palabra: "int", nos dice el tipo del valor de retorno de la función, en este caso un número entero. La función "main" siempre devuelve un entero. La segunda palabra es el nombre de la función, en general será el nombre que usaremos cuando queramos usar o llamar a la función.

La programación estructurada parte de la idea de que los programas se ejecutan secuencialmente, línea a línea, sin saltos entre partes diferentes del programa, con un único punto de entrada y un punto de salida. Pero si ese tipo de programación se basase sólo en esa premisa, no sería demasiado útil, ya que los programas sería poco manejables llegados a un cierto nivel de complejidad. La solución es crear funciones o procedimientos, que se usan para realizar ciertas tareas concretas y/o repetitivas.

Por ejemplo, si frecuentemente necesitamos mostrar un texto en pantalla, es mucho más lógico agrupar las instrucciones necesarias para hacerlo en una función, y usar la función como si fuese una instrucción cada vez que queramos mostrar un texto en pantalla.

La diferencia entre una función y un procedimiento está en los valores que devuelven cada vez que son invocados. Las funciones devuelven valores, y los procedimientos no. Lenguajes como **Pascal** hacen distinciones entre funciones y procedimientos, pero C y C++ no, en éstos sólo existen funciones y **para crear un procedimiento se hace una función que devuelva un valor vacío.**

Llamar o invocar una función es ejecutarla, la secuencia del programa continúa en el interior de la función, que también se ejecuta secuencialmente, y cuando termina, se regresa a la instrucción siguiente al punto de llamada. Las funciones a su vez, pueden invocar a otras funciones. De este modo, considerando la llamada a una función como una única instrucción (o sentencia), el programa sigue siendo secuencial.

En este caso "main" es una función muy especial, ya que nosotros no la usaremos nunca explícitamente, es decir, nunca encontrarás en ningún programa una línea que invoque a la función "main". Esta función será la que tome el control automáticamente cuando se ejecute nuestro programa. Todas las definiciones de funciones incluyen una lista de argumentos de entrada entre paréntesis, en nuestro ejemplo es una lista vacía, es decir, nuestra función no admite parámetros.

- Segunda línea: "{"

Aparentemente es una línea muy simple, las llaves encierran el cuerpo o definición de la función.

- Tercera línea: "int numero;"

Esta es nuestra primera sentencia, todas las sentencias terminan con un punto y coma. Esta concretamente es una declaración de variable. Una declaración nos dice, a nosotros y al compilador, que usaremos una variable "numero" (*como es una variable no lleva tilde*) de tipo entero. Esta declaración obliga al compilador a reservar un espacio de memoria para almacenar la variable "numero", pero no le da ningún valor inicial. En general contendrá "basura", es decir, lo que hubiera en esa zona de memoria cuando se le reservó espacio. En C y C++ es obligatorio declarar las variables que usará el programa.

C y C++ distinguen entre mayúsculas y minúsculas, así que "int numero;" es distinto de "int NUMERO;", y también de "INT numero;".

- Cuarta línea: ""

Una línea vacía no sirve para nada, al menos desde el punto de vista del compilador, pero sirve para separar visualmente la parte de declaración de variables de la parte de código que va a continuación. Se trata de una división arbitraria. Desde el punto de vista del compilador, tanto las declaraciones de variables como el código son sentencias válidas. La separación nos ayudará a distinguir visualmente dónde termina la declaración de variables. Una labor análoga la desempeña el tabulado de las líneas: ayuda a hacer los programas más fáciles de leer.

- Quinta línea: "numero = 2 + 2;"

Se trata de otra sentencia, ya que acaba con punto y coma. Esta es una sentencia de asignación. Le asigna a la variable "numero" el valor resultante de la operación "2 + 2".

- Sexta línea: "return 0;"

De nuevo una sentencia, "return" es una palabra reservada, propia de C y C++. Indica al programa que debe abandonar la ejecución de la función y continuar a partir del punto en que se la llamó. El 0 es el valor de retorno de nuestra función. Cuando "main" retorna con 0 indica que todo ha ido bien. Un valor distinto suele indicar un error. Imagina que nuestro programa es llamado desde un archivo de comandos, un archivo "bat" o un "script". El valor de retorno de nuestro programa se puede usar para tomar decisiones dentro de ese archivo. Pero somos nosotros, los programadores, los que decidiremos el significado de los valores de retorno.

- Séptima línea: "}"

Esta es la llave que cierra el cuerpo o definición de la función. Lo malo de este programa, a pesar de sumar correctamente "2+2", es que aparentemente no hace nada. No acepta datos externos y no proporciona ninguna salida de ningún tipo. En realidad es absolutamente inútil, salvo para fines didácticos, que es para lo que fue creado.

## Tipos de Datos

C trabaja con tipos de datos que son directamente tratables por el hardware de la mayoría de computadoras actuales, como son los caracteres, números y direcciones. Estos tipos de datos pueden ser manipulados por las operaciones aritméticas que proporcionan las computadoras. Pero no proporciona mecanismos para tratar tipos de datos que no sean los básicos, debiendo ser el programador el que los desarrolle. Esto permite que el código generado sea muy eficiente y de ahí el éxito que ha tenido como lenguaje de desarrollo de sistemas. Tampoco proporciona otros mecanismos de almacenamiento de datos que no sea el estático y no proporciona mecanismos de entrada ni salida. Ello permite que el lenguaje sea reducido y los compiladores de fácil implementación en distintos sistemas. Contrariamente, estas carencias se compensan mediante la inclusión de funciones de librería para realizar todas estas tareas, que normalmente dependen del sistema operativo.

## Tipos de Datos Primitivos

El C, como cualquier otro lenguaje de programación, tiene posibilidad de trabajar con datos de distinta naturaleza: texto formado por caracteres alfanuméricos, números enteros, números reales con parte entera y parte fraccionaria, etc. Además, algunos de estos tipos de datos admiten distintos números de cifras (rango y/o precisión), posibilidad de ser sólo positivos o de ser positivos y negativos, etc.

Tipos de datos fundamentales (notación completa)

| | | | |
|---|---|---|---|
| Datos enteros | *char* | *signed char* | *unsigned char* |
| | *signed short int* | *signed int* | *signed long int* |
| | *unsigned short int* | *unsigned int* | *unsigned long int* |
| Datos reales | *float* | *double* | *long double* |

La palabra char hace referencia a que se trata de un carácter (una letra mayúscula o minúscula, un dígito, un carácter especial, ...).

La palabra int indica que se trata de un número entero.
float se refiere a un número real (también llamado de punto o coma flotante).

Los números enteros pueden ser positivos o negativos (signed), o bien esencialmente no negativos (unsigned); los caracteres tienen un tratamiento muy similar a los enteros y admiten estos mismos

cualificadores. En los datos enteros, las palabras short y long hacen referencia al número de cifras o rango de dichos números. En los datos reales las palabras double y long apuntan en esta misma dirección, aunque con un significado ligeramente diferente, como más adelante se verá.

Esta nomenclatura puede simplificarse: las palabras signed e int son las opciones por defecto para los números enteros y pueden omitirse, resultando una tabla que indica la nomenclatura más habitual para los tipos fundamentales del C.

Tipos de datos fundamentales (notación abreviada):

| | | | |
|---|---|---|---|
| Datos enteros | *char* | *signed char* | *unsigned char* |
| | *short* | *int* | *long* |
| | *unsigned short* | *unsigned* | *unsigned long* |
| Datos reales | *float* | *double* | *long double* |

A continuación se va a explicar cómo puede ser y cómo se almacena en C un dato de cada tipo fundamental.

Recuérdese que **en C es necesario declarar todas las variables que se vayan a utilizar.** Una variable no declarada produce un mensaje de error en la compilación. Cuando una variable es declarada se le reserva memoria de acuerdo con el tipo incluido en la declaración.

Es posible inicializar –dar un valor inicial– las variables en el momento de la declaración; ya se verá que en ciertas ocasiones el compilador da un valor inicial por defecto, mientras que en otros casos no se realiza esta inicialización y la memoria asociada con la variable correspondiente contiene basura informática (combinaciones sin sentido de unos y ceros, resultado de operaciones anteriores con esa zona de la memoria, para otros fines).

**Caracteres (tipo char)**

Las variables carácter (tipo char) contienen un único carácter y se almacenan en un byte de memoria (8 bits). En un bit se pueden almacenar dos valores (0 y 1); con dos bits se pueden almacenar 22 = 4 valores (00, 01, 10, 11 en binario; 0, 1 2, 3 en decimal). Con 8 bits se podrán almacenar 28 = 256 valores diferentes (normalmente entre 0 y 255; con ciertos compiladores entre -128 y 127).

La declaración de variables tipo carácter puede tener la forma: char nombre; char nombre1, nombre2, nombre3;

Se puede declarar más de una variable de un tipo determinado en una sola sentencia. Se puede también inicializar la variable en la declaración. Por ejemplo, para definir la variable carácter letra y asignarle el valor a, se puede escribir: char letra = 'a';

A partir de ese momento queda definida la variable letra con el valor correspondiente a la letra a. Recuérdese que el valor 'a' utilizado para inicializar la variable letra es una constante carácter. En realidad, letra se guarda en un solo byte como un número entero, el correspondiente a la letra a en el código ASCII, que se muestra en la tabla siguiente para los caracteres estándar (existe un código ASCII extendido que utiliza los 256 valores y que contiene caracteres especiales y caracteres específicos de los alfabetos de diversos países, como por ejemplo las vocales acentuadas y la letra ñ para el castellano).

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | nul | soh | stx | etx | eot | enq | ack | bel | bs | ht |
| 1 | nl | vt | np | cr | so | si | dle | dc1 | dc2 | dc3 |
| 2 | dc4 | nak | syn | etb | can | em | sub | esc | fs | gs |
| 3 | rs | us | sp | ! | " | # | $ | % | & | ' |
| 4 | ( | ) | * | + | , | - | . | / | 0 | 1 |
| 5 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| 6 | < | = | > | ? | @ | A | B | C | D | E |
| 7 | F | G | H | I | J | K | L | M | N | O |
| 8 | P | Q | R | S | T | U | V | W | X | Y |
| 9 | Z | [ | \ | ] | ^ | | ` | a | b | c |
| 10 | d | e | f | g | h | i | j | k | l | m |
| 11 | n | o | p | q | r | s | t | u | v | w |
| 12 | x | y | z | { | \| | } | ~ | del | | |

Se utiliza de la siguiente forma. La primera cifra (las dos primeras cifras, en el caso de los números mayores o iguales que 100) del número ASCII correspondiente a un carácter determinado figura en la primera columna de la Tabla, y la última cifra en la primera fila de la Tabla. Sabiendo la fila y la columna en la que está un determinado carácter puede componerse el número correspondiente. Por ejemplo, la letra A está en la fila 6 y la columna 5. Su número ASCII es por tanto el 65. El carácter % está en la fila 3 y la columna 7, por lo que su representación ASCII será el 37. Obsérvese que el código ASCII asocia números consecutivos con las letras mayúsculas y minúsculas ordenadas alfabéticamente. Esto simplifica notablemente ciertas operaciones de ordenación alfabética de nombres. Además aparecen muchos caracteres no imprimibles (todos aquellos que se expresan con 2 ó 3 letras). Por ejemplo, el ht es el tabulador horizontal, el nl es el carácter nueva línea, etc.

Volviendo al ejemplo de la variable letra, su contenido puede ser variado cuando se desee por medio de una sentencia que le asigne otro valor, por ejemplo:

```
letra = 'z';
```

También puede utilizarse una variable char para dar valor a otra variable de tipo char:

```
caracter = letra; // Ahora caracter es igual a 'z'
```

Como una variable tipo char es un número entero pequeño (entre 0 y 255), se puede utilizar el contenido de una variable char de la misma forma que se utiliza un entero, por lo que están permitidas operaciones como:

```
letra = letra + 1;
letra_minuscula = letra_mayuscula + ('a' - 'A');
```

En el primer ejemplo, si el contenido de letra era una a, al incrementarse en una unidad pasa a contener una b. El segundo ejemplo es interesante: puesto que la diferencia numérica entre las letras minúsculas y mayúsculas es siempre la misma (según el código ASCII), la segunda sentencia pasa una letra mayúscula a la correspondiente letra minúscula sumándole dicha diferencia numérica.

Recuérdese, para concluir, que las variables tipo char son y se almacenan como números enteros pequeños. Ya se verá más adelante que se pueden escribir como caracteres o como números según que formato de conversión se utilice en la llamada a la función de escritura.

**Números enteros (tipo int)**

De ordinario una variable tipo int se almacena en 2 bytes (16 bits), aunque algunos compiladores utilizan 4 bytes (32 bits). El ANSI C no tiene esto completamente normalizado y existen diferencias entre unos compiladores y otros. Los compiladores de Microsoft para PCs utilizan 2 bytes.

Con 16 bits se pueden almacenar 216 = 65536 números enteros diferentes: de 0 al 65535 para variables sin signo, y de -32768 al 32767 para variables con signo (que pueden ser positivas y negativas), que es la opción por defecto. Este es el rango de las variables tipo int.

Una variable entera (tipo int) se declara, o se declara y se inicializa en la forma:

```
unsigned int numero;
int nota = 10;
```

En este caso la variable numero podrá estar entre 0 y 65535, mientras que nota deberá estar comprendida entre -32768 al 32767. Cuando a una variable int se le asigna en tiempo de ejecución un valor que queda fuera del rango permitido (situación de overflow o valor excesivo), se produce un error en el resultado de consecuencias tanto más imprevisibles cuanto que de ordinario el programa no avisa al usuario de dicha circunstancia. Cuando el ahorro de memoria es muy importante puede asegurarse que el computador utiliza 2 bytes para cada entero declarándolo en una de las formas siguientes:

```
short numero;
short int numero;
```

Como se ha dicho antes, ANSI C no obliga a que una variable int ocupe 2 bytes, pero declarándola como short o short int sí que necesitará sólo 2 bytes (al menos en los PCs).

**Números enteros (tipo long)**

Existe la posibilidad de utilizar enteros con un rango mayor si se especifica como tipo long en su declaración:

```
long int numero_grande;
```

o, ya que la palabra clave int puede omitirse en este caso,

```
long numero_grande;
```

El rango de un entero long puede variar según el computador o el compilador que se utilice, pero de ordinario se utilizan 4 bytes (32 bits) para almacenarlos, por lo que se pueden representar 232 = 4.294.967.296 números enteros diferentes. Si se utilizan números con signo, podrán representarse números entre -2.147.483.648 y 2.147.483.647. También se pueden declarar enteros long que sean siempre positivos con la palabra unsigned:

```
unsigned long numero_positivo_muy_grande;
```

En algunos computadores una variable int ocupa 2 bytes (coincidiendo con short) y en otros 4 bytes (coincidiendo con long). Lo que garantiza el ANSI C es que el rango de int no es nunca menor que el de short ni mayor que el de long.

**Números reales (tipo float)**

En muchas aplicaciones hacen falta variables reales, capaces de representar magnitudes que contengan una parte entera y una parte fraccionaria o decimal. Estas variables se llaman también de punto flotante. De ordinario, en base 10 y con notación científica, estas variables se representan por medio de la mantisa, que es un número mayor o igual que 0.1 y menor que 1.0, y un exponente que representa la potencia de 10 por la que hay que multiplicar la mantisa para obtener el número considerado. Por ejemplo, PI se representa como $0.3141592654 \cdot 10^{1}$.

Tanto la mantisa como el exponente pueden ser positivos y negativos.

Los computadores trabajan en base 2. Por eso un número de tipo float se almacena en 4 bytes (32 bits), utilizando 24 bits para la mantisa (1 para el signo y 23 para el valor) y 8 bits para el exponente (1 para el signo y 7 para el valor). Es interesante ver qué clase de números de punto flotante pueden representarse de esta forma. En este caso hay que distinguir el rango de la precisión. La precisión hace referencia al número de cifras con las que se representa la mantisa: con 23 bits el número más grande que se puede representar es, 223 = 8.388.608 lo cual quiere decir que se pueden representar todos los números decimales de 6 cifras y la mayor parte –aunque no todos– de los de 7 cifras (por ejemplo, el número 9.213.456 no se puede representar con 23 bits). Por eso se dice que las variables tipo float tienen entre 6 y 7 cifras decimales equivalentes de precisión.

Respecto al exponente de dos por el que hay que multiplicar la mantisa en base 2, con 7 bits el número más grande que se puede representar es 127. El rango vendrá definido por la potencia,

$$2127 = 1.7014 \cdot 1038$$

lo cual indica el número más grande representable de esta forma. El número más pequeño en valor absoluto será del orden de

$$2\text{-}128 = 2.9385 \cdot 10\text{-}39$$

Las variables tipo float se declaran de la forma:

```
float numero_real;
```

Las variables tipo float pueden ser inicializadas en el momento de la declaración, de forma análoga a las variables tipo int.

**Números reales (tipo double)**

Las variables tipo float tienen un rango y –sobre todo– una precisión muy limitada, insuficiente para la mayor parte de los cálculos técnicos y científicos. Este problema se soluciona con el tipo double, que utiliza 8 bytes (64 bits) para almacenar una variable. Se utilizan 53 bits para la mantisa (1 para el signo y 52 para el valor) y 11 para el exponente (1 para el signo y 10 para el valor). La precisión es en este caso,

$$252 = 4.503.599.627.370.496$$

lo cual representa entre 15 y 16 cifras decimales equivalentes. Con respecto al rango, con un exponente de 10 bits el número más grande que se puede representar será del orden de 2 elevado a 2 elevado a 10 (que es 1024):

$$21024 = 1.7977 \cdot 10308$$

Las variables tipo double se declaran de forma análoga a las anteriores:

```
double real_grande;
```

Por último, existe la posibilidad de declarar una variable como long double, aunque el ANSI C no garantiza un rango y una precisión mayores que las de double. Eso depende del compilador y del tipo de computador. Estas variables se declaran en la forma:

```
long double real_pero_que_muy_grande;
```

cuyo rango y precisión no está normalizado. Los compiladores de Microsoft para PCs utilizan 10 bytes (64 bits para la mantisa y 16 para el exponente).

**Duración y visibilidad de las variables: Modos de almacenamiento.**

El tipo de una variable se refiere a la naturaleza de la información que contiene (ya se han visto los tipos char, int, long, float, double, etc.). El modo de almacenamiento (storage class) es otra característica de las variables de C que determina cuándo se crea una variable, cuándo deja de existir y desde dónde se puede acceder a ella, es decir, desde dónde es visible. En C existen 4 modos de almacenamiento fundamentales: auto, extern, static y register.

Seguidamente se exponen las características de cada uno de estos modos.

1. **auto** (automático). Es la opción por defecto para las variables que se declaran dentro de un bloque {...}, incluido el bloque que contiene el código de las funciones. En C la declaración debe estar siempre al comienzo del bloque. En C++ la declaración puede estar en cualquier lugar y hay autores que aconsejan ponerla justo antes del primer uso de la variable. No es necesario poner la palabra auto. Cada variable auto es creada al comenzar a ejecutarse el bloque y deja de existir cuando el bloque se termina de ejecutar. Cada vez que se ejecuta el bloque, las variables auto se crean y se destruyen de nuevo. Las variables auto son variables locales, es decir, sólo son visibles en el bloque en el que están definidas y en otros bloques anidados en él, aunque pueden ser ocultadas por una nueva declaración de una nueva variable con el mismo nombre en un bloque anidado. No son inicializadas por defecto, y –antes de que el programa les asigne un valor– pueden contener basura informática (conjuntos aleatorios de unos y ceros, consecuencia de un uso anterior de esa zona de la memoria).

   A continuación se muestra un ejemplo de uso de variables de modo auto.

```
{
int i=1, j=2; // se declaran e inicializan i y j
...
{
float a=7., j=3.; // se declara una nueva variable j
...
j=j+a; // aqui j es float
... // la variable int j es invisible
... // la variable i=1 es visible
}
... // fuera del bloque, a ya no existe
... // la variable j=2 existe y es entera
}
```

2. **extern**. Son variables globales, que se definen fuera de cualquier bloque o función, por ejemplo antes de definir la función main(). Estas variables existen durante toda la ejecución del programa. Las variables extern son visibles por todas las funciones que están entre la definición y el fin del archivo. Para verlas desde otras funciones definidas anteriormente o desde otros archivos, deben ser declaradas en ellos como variables extern. Por defecto, son inicializadas a cero. Una variable extern

es definida o creada (una variable se crea en el momento en el que se le reserva memoria y se le asigna un valor) una sola vez, pero puede ser declarada (es decir, reconocida para poder ser utilizada) varias veces, con objeto de hacerla accesible desde diversas funciones o archivos. También estas variables pueden ocultarse mediante la declaración de otra variable con el mismo nombre en el interior de un bloque. La variables extern permiten transmitir valores entre distintas funciones, pero ésta es una práctica considerada como peligrosa. A continuación se presenta un ejemplo de uso devariables extern.

```
int i=1, j, k; // se declaran antes de main()
main()
{
int i=3; // i=1 se hace invisible
int func1(int, int);
... // j, k son visibles
}

int func1(int i, int m)
{
int k=3; // k=0 se hace invisible
... // i=1 es invisible
}
```

3. **static**. Cuando ciertas variables son declaradas como static dentro de un bloque, estas variables conservan su valor entre distintas ejecuciones de ese bloque. Dicho de otra forma, las variables static se declaran dentro de un bloque como las auto, pero permanecen en memoria durante toda la ejecución del programa como las extern. Cuando se llama varias veces sucesivas a una función (o se ejecuta varias veces un bloque) que tiene declaradas variables static, los valores de dichas variables se conservan entre dichas llamadas. La inicialización sólo se realiza la primera vez. Por defecto, son inicializadas a cero. Las variables definidas como static extern son visibles sólo para las funciones y bloques comprendidos desde su definición hasta el fin del fichero. No son visibles desde otras funciones ni aunque se declaren como extern. Ésta es una forma de restringir la visibilidad de las variables. Por defecto, y por lo que respecta a su visibilidad, las funciones tienen modo extern. Una función puede también ser definida como static, y entonces sólo es visible para las funciones que están definidas después de dicha función y en el mismo fichero. Con estos modos se puede controlar la visibilidad de una función, es decir, desde qué otras funciones puede ser llamada.
4. **register**. Este modo es una recomendación para el compilador, con objeto de que –si es posible– ciertas variables sean almacenadas en los registros de la CPU y los cálculos con ellas sean más rápidos. No existen los modos auto y register para las funciones.

**Conversiones de tipo implícitas y explícitas (casting)**

Por ejemplo, para poder sumar dos variables hace falta que ambas sean del mismo tipo. Si una es int y otra float, la primera se convierte a float (es decir, la variable del tipo de menor rango se convierte al tipo de mayor rango), antes de realizar la operación. A esta conversión **automática** e **implícita** de tipo (el programador no necesita intervenir, aunque sí conocer sus reglas), se le denomina **promoción**, pues la variable de menor rango es promocionada al rango de la otra.

Así pues, cuando dos tipos diferentes de constantes y/o variables aparecen en una misma expresión relacionadas por un operador, el compilador convierte los dos operandos al mismo tipo de acuerdo con los rangos, que de mayor a menor se ordenan del siguiente modo:

**long double > double > float > unsigned long > long > unsigned int > int > char**

Otra clase de conversión implícita tiene lugar cuando el resultado de una expresión es asignado a una variable, pues dicho resultado se convierte al tipo de la variable (en este caso, ésta puede ser de menor rango que la expresión, por lo que esta conversión puede perder información y ser peligrosa). Por ejemplo, si i y j son variables enteras y x es double,

```
x = i * j – j + 1;
```

En C existe también la posibilidad de realizar conversiones explícitas de tipo (llamadas casting, en la literatura inglesa). El casting es pues una conversión de tipo, forzada por el programador. Para ello basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se desea convertir, encerrado entre paréntesis. En el siguiente ejemplo,

k = (int) 1.7 + (int) masa;

la variable masa es convertida a tipo int, y la constante 1.7 (que es de tipo double) también. El casting se aplica con frecuencia a los valores de retorno de las funciones.

## Datos: Variables Y Constantes

**Dato:** es cualquier objeto manipulable por la computadora. Se distinguen dos clases de datos:
**Variables:** Objeto cuyo valor cambia durante la ejecución de un programa.
**Constantes:** objeto cuyo valor NO cambia durante la ejecución de un programa

## Variables

C tiene los siguientes tipos de datos simples:

| *Tipo* | *Tamaño (bytes)* | *Límite inferior* | *Límite superior* |
|---|---|---|---|
| **char** | 1 | -- | -- |
| **unsigned char** | 1 | 0 | 255 |
| **short int** | 2 | -32768 | 32767 |
| **unsigned short int** | 2 | 0 | 65536 |
| **(long) int** | 4 | -2147483648 | 2147483647 |
| **float** | 4 | $- 3.2 \times 10^{+-38}$ | $+ 3.2 \times 10^{+-38}$ |
| **double** | 8 | $- 1.7 \times 10^{+-308}$ | $+ 1.7 \times 10^{+-308}$ |

Los tipos de datos básicos tiene varios *modificadores* que les preceden. Se usa un modificador para alterar el significado de un tipo base para que encaje con las diversas necesidades o situaciones. Los modificadores son: signed, unsigned, long y short. En los sistemas UNIX todos los tipos int son long int, a menos que se especifique explícitamente short int.

**Nota:** no hay un tipo booleano en C, se debe utilizar char, int o aún mejor unsigned char.

`signed`, `unsigned`, `long` y `short` pueden ser usados con los tipos `char` e `int`. Aunque es permitido el uso de `signed` en enteros, es redundante porque la declaración de entero por defecto asume un número con signo. Para declarar una variable en C, se debe seguir el siguiente formato:

tipo lista_variables;

*tipo* es un tipo válido de C y *lista_variables* puede consistir en uno o más indentificadores separados por una coma. Un identificador debe comenzar con una letra o un guión bajo.

Ejemplo:

```
int i, j, k;
float x,y,z;
char ch;
```

## Definición de variables globales

Una varible global se declara fuera de todas las funciones, incluyendo a la función main(). Una variable global puede ser utilizada en cualquier parte del programa.

Por ejemplo:

```
short numero, suma;
int numerogr, sumagr;
char letra;

main()
{
...
}
```

Es también posible preinicializar variables globales usando el operador de asignación =, por ejemplo:

```
float suma= 0.0;
int sumagr= 0;
```

```
char letra= 'A';

main()
{
...
}
```

Que es lo mismo que:

```
float suma;
int sumagr;
char letra;

main()
{
   suma  = 0.0;
   sumagr= 0;
   letra = 'A';

...
}
```

Dentro de C también se permite la asignación múltiple usando el operador =, por ejemplo:

```
a = b = c = d = 3;
```

...que es lo mismo, pero más eficiente que:

```
a = 3;
b = 3;
c = 3;
d = 3;
```

La asignación múltiple se puede llevar a cabo, si todos los tipos de las variables son iguales. Se pueden redefinir los tipos de C usando typedef. Como un ejemplo de un simple uso se considera como se crean dos nuevos tipos real y letra. Estos nuevos tipos pueden ser usados de igual forma como los tipos predefinidos de C.

```
typedef float real;
typedef char letra;

/* Declaracion de variables usando el nuevo tipo */
real suma=0.0;
letra sig_letra;
```

**Lectura y escritura de variables**

El lenguaje C usa salida formateada. La función printf tiene un caracter especial para formatear (%) -- un caracter enseguida define un cierto tipo de formato para una variable.

```
%c caracteres
%s cadena de caracteres
%d enteros
%f flotantes
```

Por ejemplo:

```
printf("%c %d %f",ch,i,x);
```

La sentencia de formato se encierra entre " ", y enseguida las variables. Hay que asegurarse que el orden de formateo y los tipos de datos de las variables coincidan.

`scanf()` es la función para entrar valores a variables. Su formato es similar a `printf`. Por ejemplo:

```
scanf("%c %d %f %s",&ch, &i, &x, cad);
```

Observar que se antepone & a los nombres de las variables, excepto a la cadena de caracteres.

**Constantes**

ANSI C permite declarar *constantes*. Cuando se declara una constante es un poco parecido a declarar una variable, excepto que el valor no puede ser cambiado.

La palabra clave const se usa para declarar una constante, como se muestra a continuación:

```
const a = 1;
int a = 2;
```

**Notas:**

- Se puede usar const antes o después del tipo.
- Es usual inicializar una constante con un valor, ya que no puede ser cambiada *de alguna otra forma*.

La directiva del preprocesador #define es un método más flexible para definir *constantes* en un programa. Frecuentemente se ve la declaración const en los parámetros de la función. Lo anterior simplemente indica que la función no cambiara el valor del parámetro. Por ejemplo, la siguiente función usa este concepto:

```
char *strcpy(char *dest, const char *orig);
```

El segundo argumento orig es una cadena de C que no será alterada, cuando se use la función de la biblioteca para copiar cadenas.

## Operadores Aritméticos

Lo mismo que en otros lenguajes de programación, en C se tienen los operadores aritméticos más usuales (+ suma, - resta, * multiplicación, / división y % módulo).

El operador de asignación es =, por ejemplo: i=4; ch='y';

Incremento ++ y decremento -- unario. Los cuales son más eficientes que las respectivas asignaciones. Por ejemplo: x++ es más rápido que x=x+1. Los operadores ++ y -- pueden ser prefijos o postfijos. Cuando son prefijos, el valor es calculado antes de que la expresión sea evaluada, y cuando es postfijo el valor es calculado después que la expresión es evaluada.

En el siguiente ejemplo, ++z es prefijo y -- es postfijo:

```
int x,y,z;

main()
{
   x=( ( ++z ) - ( y-- ) ) % 100;
}
```

Que es equivalente a:

```
int x,y,z;

main()
{
   z++;
   x = ( z-y ) % 100;
   y--;
}
```

El operador % (módulo o residuo) solamente trabaja con enteros, aunque existe una función para flotantes (15.1 fmod() ) de la biblioteca matemática. El operador división / es para división entera y flotantes. Por lo tanto hay que tener cuidado. El resultado de x = 3 / 2; es uno, aún si x es declarado como float. La regla es: si ambos argumentos en una división son enteros, entoces el resultado es entero. Si se desea obtener la división con la fracción, entonces escribirlo como: x = 3.0 / 2; o x = 3 / 2.0 y aún mejor x = 3.0 / 2.0.

Por otra parte, existe una forma más corta para expresar cálculos en C. Por ejemplo, si se tienen expresiones como: i = i + 3; o x = x * (y + 2); pueden ser reescritas como:

```
expr1 oper = expr2
```

Lo cual es equivalente, pero menos eficiente que:

```
expr1 = expr1 oper expr2
```

*Por lo que podemos reescribir las expresiones anteriores como: i += 3; y x *= y + 2; respectivamente.*

## Operadores de Comparación

El operador para probar la igualdad es ==, por lo que se deberá tener cuidado de no escribir accidentalmente sólo =, ya que:

```
if ( i = j ) ...
```

Es una sentencia legal de C (sintácticamente hablando aunque el compilador avisa cuando se emplea), la cual copia el valor de ``j'' en ``i'', lo cual será interpretado como VERDADERO, si j es diferente de cero. Diferente es !=, otros operadores son: < menor que, > mayor que, <= menor que o igual a y >= (mayor que o igual a).

## Operadores lógicos

Los operadores lógicos son usualmente usados con sentencias condicionales o relacionales, los operadores báscios lógicos son:

`&&` Y lógico, `||` O lógico y `!` negación.

## Orden de precedencia

Es necesario ser cuidadosos con el significado de expresiones tales como a + b * c, dependiendo de lo que se desee hacer

(a + b) * c
o
a + (b * c)

Todos los operadores tienen una prioridad, los operadores de mayor prioridad son evaluados antes que los que tienen menor prioridad. Los operadores que tienen la misma prioridad son evaluados de izquierda a derecha, por lo que:

`a - b - c` es evaluado como `(a - b) - c`

| Prioridad | Operador |
|---|---|
| **Más alta** | `( ) [ ] ->` |
| | `! ~ ++ -- -` (tipo) `* & sizeof` |
| | `* / %` |
| | `+ -` |
| | `<< >>` |
| | `< <= > >=` |
| | `== !=` |

| | |
|---|---|
| | & |
| | ^ |
| | \| |
| | && |
| | \|\| |
| | ? |
| | = += -= *= /= |
| **Más baja** | , |
| | |

De acuerdo a lo anterior, la siguiente expresión: a < 10 && 2 * b < c

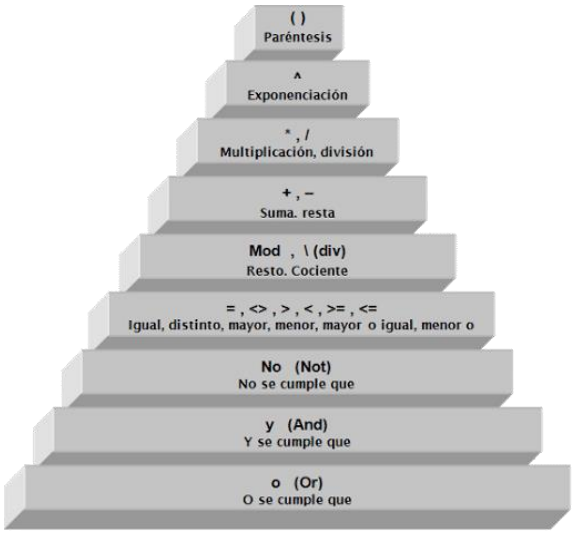
Es interpretada como: (a < 10) && ( (2 * b) < c )

y

```
a =
    b =
        10 / 5
            + 2;
```

como

```
a =
    ( b =
        ( 10 / 5 )
            + 2 );
```




## Funciones De Entrada/Salida

A diferencia de otros lenguajes, *C no dispone de sentencias de entrada/salida*. En su lugar se utilizan funciones contenidas en la librería estándar y que forman parte integrante del lenguaje. Las funciones de entrada/salida (Input/Output) son un conjunto de funciones, incluidas con el compilador, que permiten a un programa recibir y enviar datos al exterior. Para su utilización es necesario incluir, al comienzo del programa, el archivo **stdio.h** en el que están definidos sus prototipos:

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

donde ***stdio*** proviene de ***standard-input-output***.

## Función printf()

La función **printf()** imprime en la unidad de salida (el monitor, por defecto), el texto, y las constantes y variables que se indiquen. La forma general de esta función se puede estudiar viendo su *prototipo*:

```
int printf("cadena_de_control", tipo arg1, tipo arg2, ...)
```

**Explicación:** La función **printf()** imprime el texto contenido en **cadena_de_control** junto con el valor de los otros argumentos, de acuerdo con los *formatos* incluidos en **cadena_de_control**. Los puntos suspensivos (...) indican que puede haber un número variable de argumentos. Cada formato comienza con el carácter (%) y termina con un *carácter de conversión*. Considérese el ejemplo siguiente:

```
int i;
double tiempo;
float masa;
printf("Resultado nº: %d. En el instante %lf la masa vale %f\n",
i, tiempo, masa);
```

en el que se imprimen 3 variables (**i**, **tiempo** y **masa**) con los formatos (**%d**, **%lf** y **%f**), correspondientes a los tipos (***int***, ***double*** y ***float***), respectivamente. La cadena de control se imprime con el valor de cada variable intercalado en el lugar del formato correspondiente.

Caracteres de conversión para la función **printf()**:

| Carácter | Tipo de argumento | Carácter | Tipo de argumento |
|---|---|---|---|
| d,i | int decimal | o | octal unsigned |
| u | int unsigned | x,X | hex. unsigned |
| c | Char | s | cadena de char |
| f | Flotat | e,g | flotat not. cientif o breve |
| p | puntero(void*) | | |

Lo importante es considerar que debe haber correspondencia uno a uno (el 1º con el 1º, el 2º con el 2º, etc.) entre los formatos que aparecen en la **cadena_de_control** y los otros argumentos (constantes, variables o expresiones). Entre el carácter % y el *carácter de conversión* puede haber, por el siguiente orden, uno o varios de los elementos que a continuación se indican:

- Un número entero positivo, que indica la *anchura* mínima del campo en caracteres.
- Un signo (-), que indica *alineamiento* por la izquierda.
- Un punto (.), que separa la anchura de la *precisión*.
- Un número entero positivo, la *precisión*, que es el nº máximo de caracteres a imprimir en un *string*, el nº de decimales de un *float* o *double*, o las cifras mínimas de un *int* o *long*.
- Un *cualificador*: una (h) para *short* o una (l) para *long* y *double*

A continuación se incluyen algunos ejemplos de uso de la función **printf()**. El primer ejemplo contiene sólo texto, por lo que basta con considerar la **cadena_de_control**.

```
printf("Con cien cañones por banda,\nviento en popa a toda vela,\n");
```

El resultado serán dos líneas con las dos primeras estrofas de la famosa poesía. *No es posible partir **cadena_de_control** en varias líneas con caracteres **intro***, por lo que en este ejemplo podría haber problemas para añadir más estrofas. Una forma alternativa, muy sencilla, clara y ordenada, de escribir la poesía sería la siguiente:

```
printf("%s\n%s\n%s\n%s\n",
"Con cien cañones por banda,",
"viento en popa a toda vela,",
"no cruza el mar sino vuela,",
"un velero bergantín.");
```

En este caso se están escribiendo 4 cadenas constantes de caracteres que se introducen como argumentos, con formato %s y con los correspondientes saltos de línea. Un ejemplo que contiene una constante y una variable como argumentos es el siguiente:

```
printf("En el año %s ganó %ld ptas.\n", "1993", beneficios);
```

donde el texto **1993** se imprime como cadena de caracteres (%s), mientras que **beneficios** se imprime con formato de variable ***long*** (%ld). Es importante hacer corresponder bien los formatos con el tipo de los argumentos, pues si no los resultados pueden ser muy diferentes de lo esperado. La función **printf()** *tiene un valor de retorno de tipo int*, que representa el número de caracteres escritos en esa llamada.

**Función scanf()**

La función **scanf()** es análoga en muchos aspectos a **printf()**, y se utiliza para leer datos de la entrada estándar (que por defecto es el teclado). La forma general de esta función es la siguiente:

```
int scanf("%x1%x2...", &arg1, &arg2, ...);
```

donde x1, x2, ... son los *caracteres de conversión*, que representan los formatos con los que se espera encontrar los datos. La función **scanf()** devuelve como valor de retorno el número de conversiones de formato realizadas con éxito. La cadena de control de **scanf()** puede contener caracteres además de formatos. Dichos caracteres se utilizan para tratar de detectar la presencia de caracteres idénticos en la entrada por teclado. Si lo que se desea es leer variables numéricas, esta posibilidad tiene escaso interés. A veces hay que comenzar la cadena de control con un espacio en blanco para que la conversión de formatos se realice correctamente.

*En la función **scanf()** los argumentos que siguen a la **cadena_de_control** deben ser **pasados por referencia***, ya que la función los lee y tiene que trasmitirlos al programa que la ha llamado. Para ello, dichos argumentos deben estar constituidos por las *direcciones de las variables* en las que hay que depositar los datos, y no por las propias variables. Una excepción son las *cadenas de caracteres*, cuyo nombre es ya de por sí una dirección (un puntero), y por tanto no debe ir precedido por el *operador* (&) en la llamada.

| Carácter | Caracteres leídos | Argumento |
|---|---|---|
| c | cualquier carácter | char* |
| d,i | entero decimal con signo | int* |
| u | entero decimal sin signo | unsigned int |

| | | |
|---|---|---|
| o | entero octal | unsigned int |
| x,X | Entero hexadecimal | unsigned int |
| e,E,f,g,G | Número de punto flotante | float |
| s | Cadena de caracteres con ‘’ | char |
| h,l | Para short, long, y double | |
| l | Modicador para long double | |

Por ejemplo, para leer los valores de dos variables *int* y *double* y de una cadena de caracteres, se utilizarían la sentencia:

```
int n;
double distancia;
char nombre[20];
scanf("%d%lf%s", &n, &distancia, nombre);
```

En la que se establece una correspondencia entre **n** y **%d**, entre **distancia** y **%lf**, y entre **nombre** y **%s**. Obsérvese que **nombre** no va precedido por el operador (&). La lectura de cadenas de Caracteres se detiene en cuanto se encuentra un espacio en blanco, por lo que para leer una línea completa con varias palabras hay que utilizar otras técnicas diferentes. En los formatos de la cadena de control de **scanf()** pueden introducirse *corchetes* [...], que se utilizan como sigue. La sentencia:

```
scanf("%[AB \n\t]", s); // se leen solo los caracteres indicados
```

lee caracteres hasta que encuentra uno diferente de (’A’,’B’,’ ’,’\n’,’\t’). En otras palabras, se leen sólo los caracteres que aparecen en el corchete. Cuando se encuentra un carácter distinto de éstos se detiene la lectura y se devuelve el control al programa que llamó a **scanf()**. Si los corchetes contienen un carácter (^), se leen todos los caracteres distintos de los caracteres que se encuentran dentro de los corchetes a continuación del (^). Por ejemplo, la sentencia:

```
scanf(" %[^\n]", s);
```

lee todos los caracteres que encuentra hasta que llega al carácter ***nueva línea*** ’\n’. Esta sentencia puede utilizarse por tanto para leer líneas completas, con blancos incluidos. Recuérdese que con el formato ***%s*** la lectura se detiene al llegar al primer delimitador (carácter blanco, tabulador o nueva línea).

## Programación Estructurada

Un programa se escribe utilizando los siguientes tipos de estructuras de control:

- **Secuencial:** una acción se ejecuta tras otra, es decir una instrucción sigue a otra en secuencia.
- **Selectiva:** se evalúa la condición y en función del resultado se ejecuta un conjunto de instrucciones u otro, depende de como se implemente. Hay 3 tipos de selectivas: simple, doble y múltiple.
- **Repetitiva:** contienen un bucle que es un conjunto de instrucciones que se repiten un número finito de veces. Cada repetición del bucle se llama iteración. Todo bucle tiene que llevar asociada una condición que determina si el bucle se repite o no.

**Sentencias Simples:**

- Lectura de datos del teclado → LEE importe
- Escritura de datos en pantalla → ESCRIBE importe
- Expresiones aritméticas
- Asignaciones:
- importe← precio ) * 1.13
- total ← total + (precio + 3) * 2.5
- factorial ← valor * factorial
- x ← x + 1
- Expresiones lógicas (valores booleanos)
- (factorial > 34 Y factorial < 54000) o contador < 32

**Estructura secuencial**

- Sentencia 1

- ☒ Sentencia 2
- ☒ …
- ☒ Sentencia N

Se caracteriza porque las acciones se ejecutan una tras otra, es decir una sentencia sigue a otra en secuencia.

**Ejemplo de Estructura secuencial**

***Enunciado:*** Escriba un algoritmo con pseudocódigo tal que dadas 3 variables que contienen la longitud de los lados de un triángulo a, b y c, -leídos del teclado-, calcule su área s aplicando las siguientes fórmulas:

p= (a + b + c)/2
s= Vp(p – a) (p – b) (p – c)

```
ALGORITMO
VARIABLES
a,b,c,p,s SON REALES
INICIO
        LEE a
        LEE b
        LEE c
        p ← (a + b + c) / 2
        s ← raíz (p * (p - a) * (p - b) * (p - c))
        ESCRIBE “a = “, a, “b = “, b, “c = “, c,
        ESCRIBE “p = “, p, “s = “, s
        FIN
```

**Bifurcaciones**

Una bifurcación es una estructura de control en la cual, a partir del valor de una expresión, se sigue por uno u otro camino en el algoritmo.

**Sentencia if:**

**Ejemplo:** Considerar que se lee un numero entero, y se emite un mensaje indicando si el numero es positivo o no. El dato a tratar es el número que se ingresa, llamémoslo x. Entonces, el algoritmo puede ser este:

```
LEER x
Si x >0 Emitir mensaje indicando que es positivo
  Si no, Emitir mensaje indicando que no es positivo
```

**Estructura Selectiva simple**

```
SI condición ENTONCES
Sentencias
FINSI
```

Se evalúa la condición: si es verdadera, se ejecutan las sentencias; en caso contrario, ignora el grupo de sentencias.

**Ejemplo de Estructura Selectiva Simple**

***Enunciado:*** Construye un algoritmo en pseudo código tal que, dados los valores enteros P y Q, que deben leerse del teclado, determine si los mismos satisfacen la siguiente expresión:

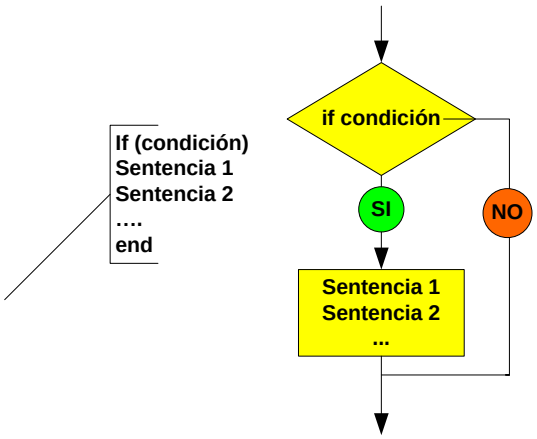
$$P^3 + Q^4 - 2P^2 < 547$$

En caso afirmativo debe emitir los valores de P y Q en la pantalla de la computadora

```
ALGORITMO
VARIABLES
p,q SON ENTEROS
INICIO
        LEE P
        LEE Q
        SI (P^3 + Q^4 - 2P^2 < 547) ENTONCES
```

```
ESCRIBE “Los valores de P y Q son:”
ESCRIBE “P = “, p, “Q = “, q
FINSI
```

**Diagrama de Flujo**




En C una bifurcación puede codificarse con la sentencia:

if <expresión lógica> sentencia1 [else sentencia2]

Lo que está entre corchetes es opcional, el else corresponde a lo que hay que hacer si la expresión lógica no es verdadera. Recordar que en C el falso se asocia al valor entero cero (0) y el verdadero a lo que es diferente de 0. Eso significa que el compilador hara una evaluación de la expresión lógica (condición) que le dara un valor 0 o diferente de 0. Si el valor es diferente de 0 se ejecutara la sentencia (simple o compuesta, en este ultimo caso será un bloque encerrado entre {}) que esta a continuación del cierre de paréntesis de la expresión lógica; si es cero, se ejecutara la sentencia (simple o compuesta) que esté después del else. Si no hay else, no se ejecutara nada, y se pasara a la siguiente línea del programa. Para el ejemplo que dimos, la codificación es;

```
int main()
{
int x;
printf(“ingrese un numero entero\n”);
scanf(“%d”, &x);
if (x>0) printf(“El numero es positivo”);
  else printf(“El numero no es positivo”);
system (“pause”);
return 0;
}
```

**Ejemplo de Estructura Selectiva Doble**

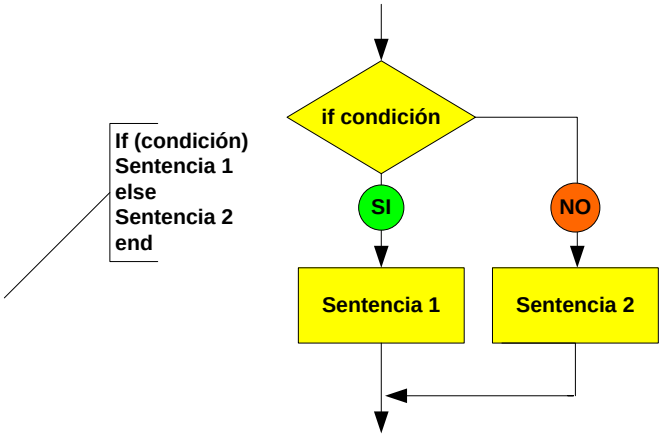
```
SI condición ENTONCES
Sentencias 1
SINO
Sentencias 2
FINSI
```

Se evalúa la condición, si es verdadera: se ejecutan las sentencias 1, si es falsa: se ejecutan las sentencias 2

**Ejemplo de Estructura Selectiva Doble**

```
ALGORITMO
VARIABLES
ex1, ex2, ex3, cal SON REALES
INICIO
        LEE ex1, ex2, ex3
        Cal ← (ex1 _ ex2 + ex3) / 3
```

```
        SI cal >= 7 ENTONCES
        ESCRIBE “Aprobado”
        SINO
        ESCRIBE “No aprobado”
        FINSI
FIN
```




**Estructura Selectiva Múltiple**

```
SI condición 1 ENTONCES
Sentencias 1
SINO
        SI condicion 2 ENTONCES
        Sentencias 2
SINO
Sentencias 3
FINSI
```

**Ejemplo de estructura selectiva múltiple**

***Enunciado:*** Elabore un algoritmo en pseudocódigo que lea del teclado una temperatura en grados centígrados, calcule los grados Fahrenheit y escriba por pantalla el deporte que es apropiado practicar a esa temperatura, tenieno en cuenta la siguiente tabla:

| Deporte | Temperatura |
|---|---|
| Natación | >85 |
| Tenis | 70<TEMP<=85 |
| Golf | 35<TEMP<=70 |
| Esquí | <=35 |

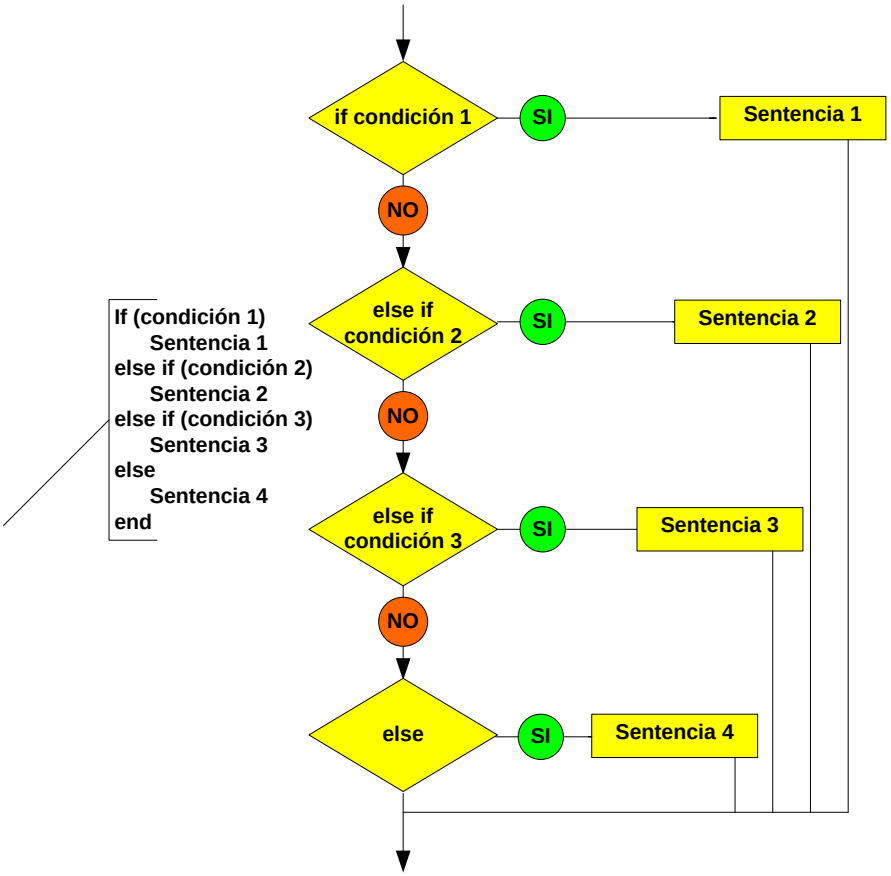
**Ejemplo de Estructura Selectiva Múltiple**

```
ALGORITMO
VARIABLES
centi, farenh SON REALES
INICIO
        LEE centi
        farenh ← (9/5) * centi + 32

        SI farenh > 85 ENTONCES
            ESCRIBE “Natación”
                SINO
                    SI farenh <= 85 Y farenh > 70 ENTONCES
                            ESCRIBE “Tenis”
                        SINO
                            SI farenh <= 70 Y farenh > 35 ENTONCES
                                    ESCRIBE “Golf”
```

```
                    SINO
                        ESCRIBE “Tenis”
                                FINSI
                            FINSI
                        FINSI
                FINSI
                FIN
```

**Diagrama de flujo**

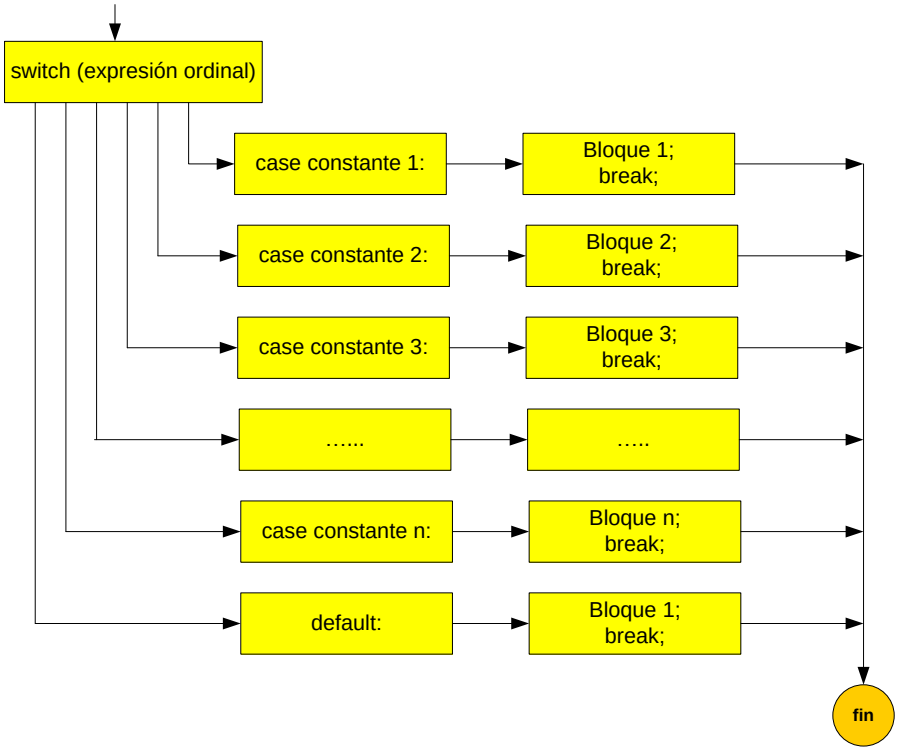



**Sentencia switch..case:**

La sentencia switch es una instrucción de control que controla múltiples selecciones y enumeraciones pasando el control a una de las instrucciones **case** de su cuerpo. Se utiliza cuando el algoritmo se divide en más de dos “ramas” a partir del valor de cierta expresión. La sintaxis o estructura general de la sentencia switch es la siguiente:

```
switch (expresión_aritmetica) {
case constante1: sentencia_1;  break;
...
case constante2: sentencia_2;  break;
...
case constante n:  sentencia_n;  break;
default: sentencia_m; }
```

**Diagrama de flujo**




La expresión aritmética debe corresponder a un tipo entero. (No se permite un valor real, por ejemplo). En cada case se establece que debe hacerse si la expresión aritmética tiene el valor indicado. El **break** es imprescindible (analice lo que sucede si no se lo coloca). Cada sentencia incluye a su ";" de terminación. El default es opcional, indicando lo que debe hacerse si ninguna de las constantes anteriores en la de la expresión o variable que se analiza.

**Ejemplo:** Se ingresa un entero correspondiente a un dia de la semana (1 por el lunes, 2 por el martes, etc). Se desea emitir un mensaje por pantalla, indicando de que día de la semana se trata, o un mensaje indicando que el número es incorrecto si supera a 7. El dato a considerar es el número entero que se ingresa, llamémoslo x.

## Ciclos

Los ciclos o bucles son sentencias de control mediante las cuales se logra que una sentencia se ejecute varias veces, mientras se verifica determinada condición. Existen ciclos con control en la entrada, y con control en la salida. Los ciclos con control en la entrada analizan la condición y si ésta se verifica ejecutan la sentencia, la cual puede ser simple o compuesta (en este último caso será un bloque encerrado entre {}). Esto significa que existe la posibilidad de que la sentencia en cuestión no se ejecute ni una vez, ya que la condición puede no verificarse la primera vez que se analiza. En C hay dos sentencias que corresponden a ciclos con control en la entrada: el while y el for. Los ciclos con control en la salida en primer lugar ejecutan el bloque y luego analizan la condicion; si esta se verifica, se procede a ejecutar nuevamente la sentencia y asi sucesivamente. En C la sentencia do..while corresponde a un ciclo con control en la salida.

## Estructura Repetitiva …mientras (while)

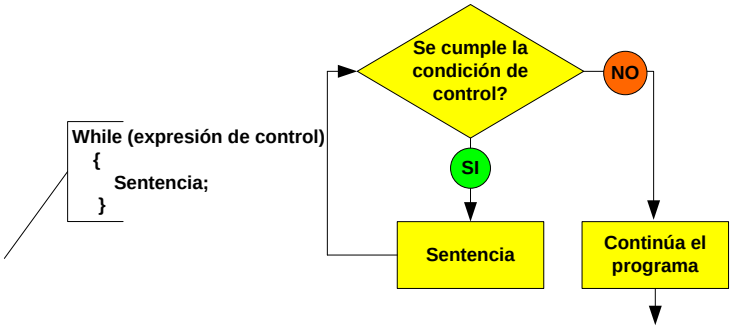
MIENTRAS condición HACER

Sentencias

FIN MIENTRAS

La condición del bucle se evalúa al principio, antes de ejecutar las sentencias del bucle. Si es verdadera, se ejecutan las sentencias del bucle y después se vuelve a preguntar por la condición. En el momento en que la condición es falsa se sale del bucle. Como la condición es evaluada, la primera vez, antes de entrar en el bucle, puede que el bucle se ejecute 0 (cero) veces. Para finalizar un bucle, el valor de la condición debe ser modificado en las sentencias que componen el bucle.

## Diagrama de flujo




**Estructura Repetitiva hacer… mientras (do..while)**

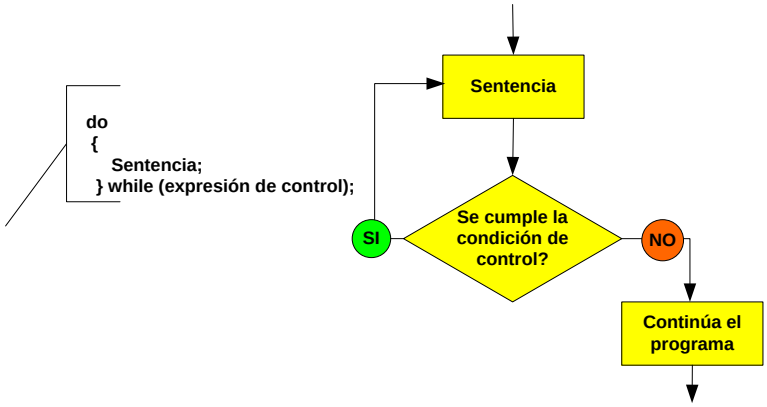
HACER

Sentencias

MIENTRAS condición

La condición se evalúa siempre al final del bucle, si es verdadera volvemos a ejecutar las acciones, si es falsa, se sale del bucle. Como la condición se evalúa al final, incluso aunque la primera vez sea falsa, se ejecuta al menos una vez por el bucle. Para finalizar un bucle, el valor de la condición debe ser modificado en las sentencias que componen el bucle.

**Diagrama de flujo**




**Estructura Repetitiva para (for)**

PARA inicio, condición, incremento

Sentencias

Se inicializa la variable de control y el bloque se ejecuta hasta que la condición es falsa. Se evalúa al principio y se ejecuta hasta el tope. Para finalizarlo, el valor de la condición debe ser modificado con el incremento que es parte de la estructura que compone el bucle.

**Diagrama de flujo**