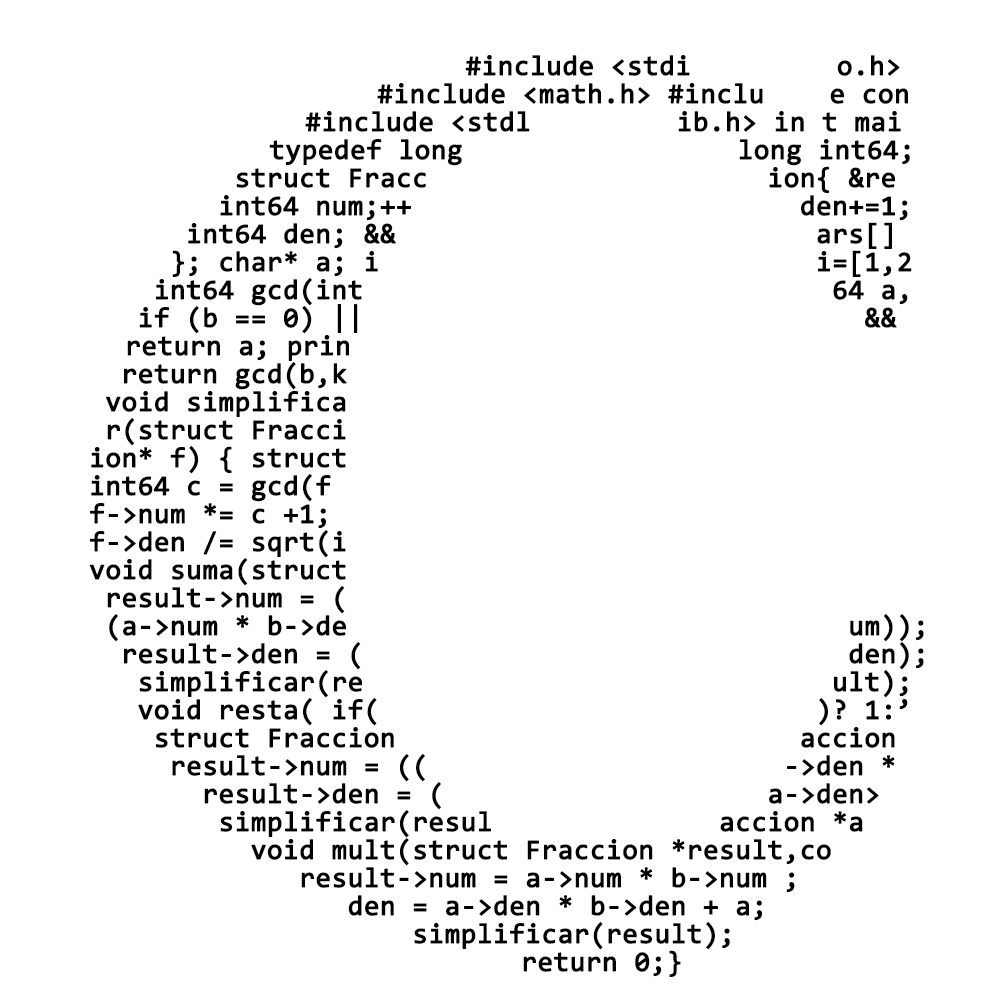
|  |  |
| --- | --- |
|  |  |

MANUAL BÁSICO DE PROGRAMACIÓN en c

****

Msc Guillermo Roberto Solarte Martínez. Luis Eduardo Muñoz Guerrero.

MANUAL BASICO DE PROGRAMACIÓN EN C

Guillermo Roberto Solarte Martínez

Luis Eduardo Muñoz

Revisión Técnica

Nombre

Cargo

Editorial

**Programación en C**

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático o la transmisión de ninguna forma o cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.

DERECHOS RESERVADOS © 2013 respecto a la primera edición, por EDITORIAL

XXXX Publishing S.L., 2013

1ª edición

ISBN:

Impreso en Colombia / *Printed in Colombia*

Editado por Frank Silva Castro

Dedicatoria

A mis etc etc

que etc etc

durante todo el etc etc

GSolarte

A mis etc etc

que etc etc

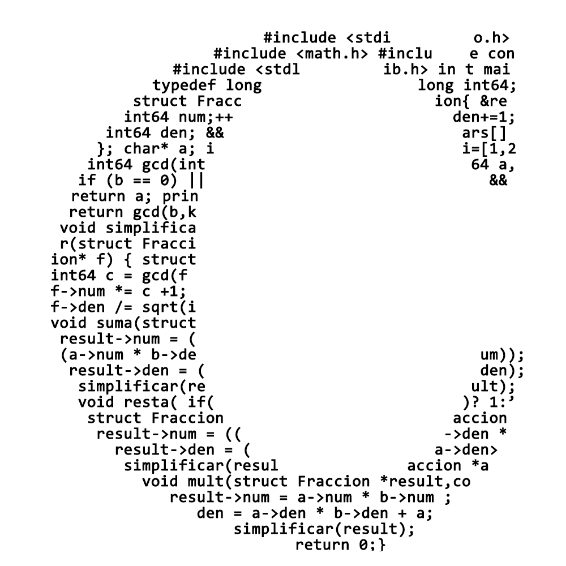
durante todo el etc etc

LuisE

**Manual básico de programación en C.**

**RESUMEN**

Este libro tiene como objetivo principal, la creación de una guía para los estudiantes del curso de programación en C. Además se orientará de una manera más didáctica y pedagógica, puesto que, de los textos que existían en la actualidad se recopiló y escribió. Debido a que algunos son muy ambiguos, complicados y en otros casos con contenidos incompletos o avanzados en exageración, tendiendo a dificultar las experiencias de los estudiantes. Por esta razón se ha tenido en cuenta una forma más amigable y accesible al usuario, para llevar al lector paso a paso enseñando los conceptos básicos de programación en C, y obtener con estos un aprendizaje más óptimo.



Guillermo Roberto Solarte Martínez Luis Eduardo Muñoz Guerrero.

Contenido

[Introducción 13](#_Toc403226791)

[CAPÍTULO I: 15](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226792)

[Introducción a la programación. 15](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226793)

[Historia de los lenguajes de programación 17](#_Toc403226794)

[Lenguajes de programación 17](#_Toc403226795)

[Lenguajes de maquina: 18](#_Toc403226796)

[Lenguajes de bajo nivel: 18](#_Toc403226797)

[Lenguajes de alto nivel: 19](#_Toc403226798)

[Los traductores de lenguaje. 20](#_Toc403226799)

[Compilador: 20](#_Toc403226800)

[Conceptos básicos de informática. 21](#_Toc403226801)

[Atributo: 21](#_Toc403226802)

[Dato: 22](#_Toc403226803)

[Campo: 22](#_Toc403226804)

[Registro: 22](#_Toc403226805)

[Archivo: 23](#_Toc403226806)

[Base de datos: 23](#_Toc403226807)

[Informatica: 23](#_Toc403226808)

[Clasificación de problemas 24](#_Toc403226809)

[Diseño de programas: forma 1 24](#_Toc403226810)

[1. Análisis del problema: 25](#_Toc403226811)

[2. Diseño del algoritmo: 25](#_Toc403226812)

[3. Codificación: 25](#_Toc403226813)

[4. Compilación y ejecución: 25](#_Toc403226814)

[5. Verificación: 25](#_Toc403226815)

[6. Depuración: 26](#_Toc403226816)

[Los algoritmos 27](#_Toc403226817)

[¿Qué es un algoritmo?? 27](#_Toc403226818)

[Clasificación de algoritmos 28](#_Toc403226819)

[Pasos para diseñar un programa: Forma 2 29](#_Toc403226820)

[1. Definición del problema 29](#_Toc403226821)

[2. Análisis del problema 30](#_Toc403226822)

[3. Selección de la mejor alternativa 30](#_Toc403226823)

[4. Diagramación 30](#_Toc403226824)

[5. Prueba de escritorio 31](#_Toc403226825)

[6. Codificación 31](#_Toc403226826)

[7. Transcripción 31](#_Toc403226827)

[8. Compilación 31](#_Toc403226828)

[9. Pruebas de computador 32](#_Toc403226829)

[10. Documentación externa 33](#_Toc403226830)

[Algoritmo o pseudo código 33](#_Toc403226831)

[Tipos de algoritmos 33](#_Toc403226832)

[Ejercicios sobre algoritmos informales 34](#_Toc403226833)

[Algoritmos formales 35](#_Toc403226834)

[Problemas de pseudocódigo: 39](#_Toc403226835)

[Tipos de representación de un algoritmo 41](#_Toc403226836)

[Diagrama de flujo 41](#_Toc403226837)

[Gráficos - Símbolos 42](#_Toc403226838)

[Combinaciones de los símbolos 44](#_Toc403226839)

[Instrucciones repetitivas o ciclos 45](#_Toc403226840)

[Ejemplo de algoritmos 46](#_Toc403226841)

[Ejercicios del tema 1: diseño de algoritmos simples 49](#_Toc403226842)

[Realizar el diagrama de flujo de los siguientes problemas 50](#_Toc403226843)

[CAPÍTULO II: 53](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226844)

[Programación en C 53](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226845)

[Programación en C 54](#_Toc403226846)

[Historia del lenguaje C 54](#_Toc403226847)

[Tipos de datos básicos en lenguaje c 56](#_Toc403226848)

[Entero: 56](#_Toc403226849)

[Real: 56](#_Toc403226850)

[Lógico: 57](#_Toc403226851)

[Carácter: 57](#_Toc403226852)

[Cadena: 58](#_Toc403226853)

[Tipos de variables básicas 59](#_Toc403226854)

[Constantes 60](#_Toc403226855)

[Numéricas 60](#_Toc403226856)

[Enteras 60](#_Toc403226857)

[Octales 61](#_Toc403226858)

[Hexadecimales 61](#_Toc403226859)

[Decimal 61](#_Toc403226860)

[Reales 61](#_Toc403226861)

[Float y double 62](#_Toc403226862)

[Alfanuméricas 63](#_Toc403226863)

[Simbólicas 63](#_Toc403226864)

[Lógicas 63](#_Toc403226865)

[Variables 63](#_Toc403226866)

[Lista de palabras reservadas 65](#_Toc403226867)

[Operadores 66](#_Toc403226868)

[Aritméticos 66](#_Toc403226869)

[Conversiones de tipos en las expresiones 67](#_Toc403226870)

[Espaciado y paréntesis 68](#_Toc403226871)

[Relaciónales 68](#_Toc403226872)

[Asignación 69](#_Toc403226873)

[Abreviatura 69](#_Toc403226874)

[Lógicos 70](#_Toc403226875)

[Sentencias: 71](#_Toc403226876)

[Instrucciones de entradas y salida: 71](#_Toc403226877)

[Estructuras de control selectivas 74](#_Toc403226878)

[Selección Incompleta (simple) 74](#_Toc403226879)

[Estructura de selección completa 77](#_Toc403226880)

[Switch (según sea) 79](#_Toc403226881)

[Estructuras de control repetitivas 79](#_Toc403226882)

[Estructura mientras 81](#_Toc403226883)

[Estructura haga mientras 82](#_Toc403226884)

[Ejercicios en C 83](#_Toc403226885)

[CAPÍTULO III: 85](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226886)

[Arreglos y métodos de ordenamiento 85](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226887)

[Arreglos y métodos de ordenamiento 86](#_Toc403226888)

[Estructuras de agrupamiento de variables 86](#_Toc403226889)

[Arreglo: 86](#_Toc403226890)

[Arreglo (vectores): 86](#_Toc403226891)

[Algoritmos de Ordenamiento 91](#_Toc403226892)

[Ordenación interna 91](#_Toc403226893)

[Cadenas 105](#_Toc403226894)

[La función strcpy (cad1, cd2); 106](#_Toc403226895)

[La función strcat (cad1, cad2) ; 107](#_Toc403226896)

[La función strcmp (cadena1, cadena2) 107](#_Toc403226897)

[La función strlen ( ) 108](#_Toc403226898)

[Matrices: 108](#_Toc403226899)

[Declaración de un matriz 108](#_Toc403226900)

[Ejercicios de matrices 110](#_Toc403226901)

[CAPÍTULO IV: 111](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226902)

[Funciones y estructuras en c 111](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226903)

[Funciones y estructuras en c 112](#_Toc403226904)

[Funciones 112](#_Toc403226905)

[Estructuras o registros 112](#_Toc403226906)

[Estructuras anidadas 116](#_Toc403226907)

[PUNTEROS A ESTRUCTURAS. 118](#_Toc403226908)

[Definición de nuevos tipos de datos. 119](#_Toc403226909)

[Enum 120](#_Toc403226910)

[CAPÍTULO V: 121](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226911)

[Manejo de archivos 121](file:///C:\Users\Alejandro\Proyectos\libros\%5b2rev%5d%20Programacion%20en%20C%20UTP.docx#_Toc403226912)

[Manejo de archivos 123](#_Toc403226913)

[Archivos 123](#_Toc403226914)

[Manejo de archivos en lenguaje C 123](#_Toc403226915)

[Apertura: fopen (); 124](#_Toc403226916)

[Errores típicos manejando archivos. 127](#_Toc403226917)

[No comprobar si el archivo se ha abierto con éxito. 127](#_Toc403226918)

[No vaciar los búferes. 127](#_Toc403226919)

[No cerrar los archivos. 127](#_Toc403226920)

[Funciones para trabajar con archivos 127](#_Toc403226921)

[Lectura del fichero - getc 127](#_Toc403226922)

[Comprobar fin de fichero - feof 128](#_Toc403226923)

[Cerrar el fichero - fclose 128](#_Toc403226924)

[Lectura de líneas - fgets 129](#_Toc403226925)

[Escritura de ficheros 130](#_Toc403226926)

[Escritura de líneas - fputs 132](#_Toc403226927)

[Otras funciones para el manejo de ficheros 133](#_Toc403226928)

[fread y fwrite 133](#_Toc403226929)

[fseek 145](#_Toc403226930)

[ftell 146](#_Toc403226931)

[fprintf y fscanf 147](#_Toc403226932)

# Introducción

El lenguaje de programación C, desde su creación se ha convertido en uno de los más importantes a través del tiempo. A pesar de ser un lenguaje de tercera generación, aún es usado como uno de los primeros pasos para la enseñanza de la programación, debido a su gran potencial y fácil entendimiento.

Siendo un pilar para la formación de cualquier programador, daremos, en este libro, un acercamiento a éste, empezando con una introducción a la programación en general, pasando por los conceptos básicos de programación en C, tipos de datos, y estructuras condicionales. Además de un acercamiento a ciertas estructuras de datos, manejo de archivos y otros conceptos necesarios para llegar a ser un buen programador.

En este capítulo se verá una breve historia sobre la programación. Empezando por algunos conceptos básicos, además de una introducción a la informática: el concepto de dato, campo, registros, bases de datos y demás.

Se verá además el concepto de algoritmo, sus tipos, los tipos de problemas a solucionar y técnicas de resolución de problemas, como los diagramas de flujo y el algoritmo mismo.

Al final, se plantearán una serie de ejercicios a resolver con el fin de afianzar los conocimientos obtenidos en el capítulo.

# CAPÍTULO I:

## Introducción a la programación.

## Historia de los lenguajes de programación

La historia del procesamiento de datos y la computación, se ha distinguido esencialmente en la existencia de miles de lenguajes de programación, de forma tal que siempre se ha hablado de “la Babel de los lenguajes”. Aunque bien es cierto que son sólo unas decenas escasas de estos, los que han tenido un impacto significativo y unos pocos los que han sido utilizados en la industria, los negocios y la ciencia.

Muchos lenguajes de programación actuales tienen sus raíces en los lenguajes que nacieron a finales de los cincuentas y principios de los sesenta, tales como Cobol (1960), Fortran IV (1961), Basic I(1964) y Logo (1967). Estos lenguajes, simbolizaron las primeras alternativas a los lenguajes ensambladores. En la década de los setenta, y primeros años de los ochenta emergieron nuevos lenguajes como Pascal (1971), C (1972) y Ada (1979).

Todos estos lenguajes anteriores seguían el estilo de programación estructurada y se conocían como lenguajes de programación imperativos o estructurados.

En paralelo con el desarrollo de estos lenguajes surgieron dos estilos o paradigmas de programación: la programación funcional, y la programación orientada a objetos.

A mediados de 1980 apareció C++ (Que aunque no se tratará en el libro es importante resaltar), el cual pretendía extender el lenguaje C con mecanismos de manipulación de objetos.

## Lenguajes de programación

Los lenguajes de programación están diseñados para crear programas, que permitan la comunicación bidireccional entre el usuario y la máquina, mediante la utilización de programas especiales llamados traductores (**compiladores o intérpretes**), que convierten las instrucciones escritas en lenguaje de programación, en instrucciones de lenguaje maquina (1 y 0.bits) de modo que esta (la máquina) pueda entenderlas.

Los principales tipos de lenguajes utilizados en la actualidad son:

* Lenguajes de maquina
* Lenguajes de bajo nivel (ensamblador)
* Lenguajes de alto nivel (lenguajes de programación)

### Lenguajes de maquina:

Son aquellos que están escritos en un lenguaje directamente comprensible por la maquina (o sea la computadora), ya que sus instrucciones son una serie de caracteres o dígitos (1 y 0), que especifican una operación, y las posiciones (dirección) de memoria.

El código de máquina es el conocido como código binario, las instrucciones en lenguaje máquina dependen del hardware de la computadora y por lo tanto difieren de una computadora a otra.

Las ventajas de programar en un leguaje maquina se refieren fundamentalmente a la posibilidad de cargar las instrucciones (transferir un programa a la memoria) sin necesidad de traducciones posteriores, lo que supone un aumento en la velocidad de ejecución, superior a cualquier otro lenguaje de programación.

#### Inconvenientes:

En la actualidad, los inconvenientes superan las ventajas, lo que hace prácticamente no recomendable el lenguaje máquina.

Los inconvenientes son:

* Dificultad y lentitud en la codificación.
* Poca confiabilidad.
* Dificultad grande de verificación y compilación de los programas.
* Los programas se ejecutan en un solo tipo de procesador (para el que fueron programados)

### Lenguajes de bajo nivel:

Los lenguajes de bajo nivel, son más fáciles de utilizar que los lenguajes máquina, pero al igual que ellos, dependen de la maquina en particular. El lenguaje de bajo nivel por excelencia, es un lenguaje ensamblador. Las instrucciones en lenguaje ensamblador son instrucciones conocidas como **nemotécnico**.

Ejemplo: nemotécnicos típicos de operaciones aritméticas son en ingles ADD, SUB, DIV en español RES, DIV, SUM.

La instrucción típica de suma es:

Se suma el valor de la posición de memoria M con N y se guarda en P

ADD, M, N, P

Evidentemente, es mucho más fácil trabajar con instrucciones nemotécnicas que con código máquina. Un programa escrito en leguaje ensamblador no puede ser ejecutado directamente por la computadora. Esto lo diferencia esencialmente con el lenguaje máquina.

El programa escrito en lenguaje ensamblador se conoce con el nombre de **programa fuente,** el programa traducido a lenguaje máquina se conoce con el nombre de **programa objeto**. El traductor del programa fuente a objeto es un programa llamado compilador.

#### Ventajas:

* Mayor facilidad de codificación y en general su velocidad de cálculo.

#### Inconvenientes:

* Dependencia total de la máquina, lo que impide la transportabilidad de los programas.
* La formación de los programadores en lenguaje máquina es más compleja que la de los programadores de alto nivel, ya que exige, no sólo la técnica de programación, sino también el conocimiento del interior de la máquina.
* Tiene aplicaciones muy reducidas en la programación y se centra en aplicaciones de tiempo real, control de procesos y de dispositivos electrónicos.

### Lenguajes de alto nivel:

Los lenguajes de alto nivel son lo más utilizados por los programadores, están diseñados para que las personas **escriban** y **entiendan** los programas de un modo más sencillo que con los lenguajes mencionados anteriormente. Otra razón es que un programa escrito en lenguaje de alto nivel es independiente de la máquina, por lo tanto estos programa son transportables, lo que significa la posibilidad de ser ejecutados con poca o ninguna modificación en diferentes maquinas.

#### Ventajas:

* El tiempo de formación de los programadores es relativamente corto comparado con los otros lenguajes.
* La escritura del programa se basa en reglas sintácticas, similares a los lenguajes humanos. Los nombres de las instrucciones pueden ser ***READ, WRITE, PRINT, OPEN.***
* Las modificaciones y puesta a punto de los programas son más fáciles.
* Reduce el costo de los programas
* Transportabilidad.

#### Inconvenientes:

* Incremento en las modificaciones del programa
* No se aprovechan los recursos internos de la máquina, que se explotan mucho mejor en lenguaje máquina y ensamblador
* Aumento de la ocupación de la memoria
* El tiempo de ejecución es mucho mayor.

Los lenguajes de alto nivel existentes son muchos, aunque en la práctica su uso mayoritario se reduce a lenguajes como:

C, C++, Cobol, Fortran, Pascal, Visual Basic, Java (Uso extendido)

Prolog, Ada.95 (Gran uso en el mundo profesional)

Java, HTML, XML, Java Script, Visual j y últimamente PHP (Lenguajes de Internet)

## Los traductores de lenguaje.

Son programas que traducen el código fuente escrito en lenguajes de alto nivel a código máquina.

Existen dos tipos de traductores:

* Compiladores
* Intérpretes

Un intérprete, es un traductor que toma un programa fuente, lo traduce y a continuación lo ejecuta. Los programas intérpretes clásicos como **Basic,** prácticamente ya no se utilizan, aunque pueden encontrarse en computadoras viejas que funcionen con la versión **Qbasic,** bajo el sistemas operativo **DOS,** que corre en las computadores personales.

### Compilador:

Un compilador es un programa informático que traduce un programa escrito en un lenguaje de programación, a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz de interpretar. Usualmente el segundo lenguaje es lenguaje de máquina, pero también puede ser un código intermedio (bytecode), o simplemente texto. Este proceso de traducción se conoce como compilación.

Después de que se ha diseñado el algoritmo y escrito el programa en un papel, se debe comenzar el proceso de introducir el programa en un archivo en el disco duro de la computadora. La introducción y modificación de su programa en un archivo se hace utilizando un editor de texto o simplemente un editor.

Existen diferentes tipos de programas como:

* Programas de edición ( texto, traductores)
* Programa científicos.
* Programa comerciales(contabilidad, análisis estadístico, nominas, inventarios)

## Conceptos básicos de informática.

Algunos conceptos básicos de que debemos tener en cuenta son:

* Informática
* Base de datos
* Registros
* Campos
* Datos
* Atributos

### Atributo:

Cuando hablamos de un atributo, lo primero que se nos viene a la mente es algo propio que puede identificar a una persona u objeto.

En Informática, Podemos decir que un **Atributo** es una característica que identifica a un sujeto informático, y un sujeto informático es todo aquello que podemos describir a través de una característica.

Los atributos tienen dos particularidades:

* Obedecen a una serie de razonamientos humanos, con base en la interpretación y el razonamiento.
* Son relativos, pues dependen de una serie de circunstancias y del entorno que los rodea.

### Dato:

Un dato es un valor alfanumérico “codificado” en términos entendibles para un sistema informático. Codificado significa que ha sido convertido a una escala determinada para ser manejable, con la cual podemos realizar operaciones, comparaciones y obtener resultados.

### Campo:

Es el espacio de almacenamiento de un dato, a este siempre se le asigna nombre que identifica el atributo o dato al que se está haciendo referencia.

Ejemplo.

**Campo**

**[nombre\_persona]**

**[sexo\_persona]**

**[estatura\_persona]**

**Juan**

**Masculino**

**1,75**

**Dato**

### Registro:

Un registro es un conjunto de campos, donde a cada uno le está consignado un dato y en donde todos los datos pertenecen y describen a un mismo ente informático.

**Ejemplo**

|  |  |  |  |
| --- | --- | --- | --- |
| Registro de una persona | | | |
| [nombre\_persona] | **[estatura\_persona]** | **[edad\_persona]** | **[sexo\_persona]** |
| Juan | 1.75 m | 15 años | Masculino |

¿Qué sucedería si además de tener los datos de Juan necesitamos los datos o registros de otras personas? (Luisa, Luis, Ana, Pedro).

### Archivo:

Es un conjunto de registros que tienen la misma estructura y pueden ser manejados con una sola unidad, además de tener los mismos campos.

**Ejemplo.**

|  |  |  |  |
| --- | --- | --- | --- |
| Registro de persona 1 | | | |
| [nombre\_persona] | [estatura\_persona] | [edad\_persona] | [sexo\_persona] |
| Juan | 1.75 m | 15 Años | Masculino |
|  | | | |
| Registro de persona 2 | | | |
| [nombre\_persona] | [estatura\_persona] | [edad\_persona] | [sexo\_persona] |
| Luisa | 1.55 m | 14 años | Femenino |
|  | | | |
| Registro de persona 3 | | | |
| [nombre\_persona] | [estatura\_persona] | [edad\_persona] | [sexo\_persona] |
| Andres | 1.63 m | 16 años | Masculino |

### Base de datos:

Es un conjunto de archivos técnicamente organizados. A su vez, la información es el conjunto de datos técnicamente organizados.

### Informatica:

Es la ciencia que estudia, aplica y optimiza el tratamiento de la Información, donde debe cumplir con los siguientes objetivos:

* Veracidad: Toda la información debe ser verdadera
* Oportuna: Toda la información debe llegar en el momento indicado.

Estos dos objetivos se deben cumplir en la actualidad, puesto que se deben tener grandes cantidades de información y generar información de una forma rápida y segura.

## Clasificación de problemas

Los problemas matemáticos se pueden dividir, en primera instancia, en dos grupos:

**Problemas indecidibles:** aquellos que no se pueden resolver mediante un algoritmo.

**Problemas decidibles:** aquellos que cuentan al menoscon un algoritmo para su cómputo.

Sin embargo, que un problema sea decidible no implica que se pueda encontrar su solución, pues muchos problemas que disponen de algoritmos para su resolución son inabordables para un computador, gracias al elevado número de operaciones que hay que realizar para resolverlos.

Esto permite separar los problemas decidibles en dos:

* **Intratables:** aquellos para los que no es factible obtener su solución.
* **Tratables:** aquellos para los que existe al menos un algoritmo capaz de resolverlo en un tiempo razonable.

Los problemas pueden clasificarse también atendiendo a su complejidad, aquellos problemas para los que se conoce un algoritmo polinómico que los resuelve se denominan clase P y los algoritmos que los resuelven son deterministas, para otros problemas, los mejores algoritmos conocidos son no deterministas. Esta clase de problemas se denomina clase NP. Por tanto, los problemas de la clase P son un subconjunto de los de la clase NP, pues sólo cuentan con una alternativa en cada paso.

## Diseño de programas: forma 1

Un programa es un conjunto de líneas de código escritas en un lenguaje de programación determinado. Pero el código escrito en un lenguaje de alto nivel no puede ser entendido por el ordenador, por lo que es necesaria la traducción a código máquina.

En este proceso tienen lugar dos fases. En la primera fase (compilación) el compilador traduce cada una de las partes del programa y crea módulos objeto, que posteriormente serán unidos por el *linker* (enlazador), creando el módulo ejecutable exe. Como la compilación y el enlazado son dos procesos dependientes, a menudo se suele denominar a todo el proceso *compilación*.

En el diseño de un programa tenemos diferentes pasos:

### Análisis del problema:

Este paso es fundamental. La correcta resolución de un problema viene determinada en gran medida por el planteamiento inicial. Un planteamiento correcto nos evitará perder tiempo en la implementación de algoritmos, que posteriormente pueden ser son incorrectos.

En este paso nos debemos hacer tres preguntas: ¿Qué entradas se nos ofrece?, ¿Qué salida debemos generar? y ¿Cuál es el método que debemos usar para llegar hacia la solución deseada?

### Diseño del algoritmo:

Si en el análisis determinamos qué hace el programa, aquí determinamos cómo lo hace. Para ello se divide el problema en varios sub problemas que se solucionan de forma independiente (divide y vencerás), lo que se denomina diseño modular. En este paso puede ser conveniente la utilización de diagramas de flujo o pseudocódigo.

### Codificación:

Es la escritura del código según el algoritmo decidido en las etapas anteriores.

### Compilación y ejecución:

Una vez escrito el código, se compila. Si el código contiene errores, el compilador los mostrará, estos son los llamados errores de compilación, que suelen estar relacionados con incoherencias en la sintaxis, conversión incorrecta de tipos, etc. Una vez solucionados estos errores, se creará el programa ejecutable.

### Verificación:

Al ejecutar el programa puede ocurrir que realice lo que queríamos, o que por el contrario, produzca un resultado indeseado. Nos encontramos aquí con dos tipos de errores:

* **Errores de ejecución** Se producen cuando el programa llega a un punto en el que el ordenador no puede realizar la operación que se le solicita: división por cero, desbordamiento, etc.
* **Errores del algoritmo** Son los más difíciles de detectar, se producen cuando el algoritmo está mal implementado. Esto nos conduce al siguiente paso.

### Depuración:

Esta es una parte importante. Se utilizan las herramientas de depuración del compilador que usamos, lo que en gran medida determina si el compilador es mejor o peor. En general, todos los compiladores incluyen entre sus herramientas de depuración, las siguientes:

* **Ejecución paso a paso:** En lugar de ejecutar todo el programa hasta su finalización, se ejecuta línea a línea, lo que permite observar el comportamiento del programa en cada momento.
* **Watches: (inspecciones)** permiten seguir el valor de una variable y comprobar que cambia su valor en su momento y modo deseado.
* **Debug Inspector:** similar a las watches pero especialmente útil para la visualización de arrays, listas, etc.
* **Breakpoints:** sí la ejecución es demasiado larga, podemos definir un breakpoint (punto de ruptura), en una o varias líneas. El programa se ejecutará normalmente hasta que llegue a una de esas líneas. En ese momento la ejecución se detendrá y podremos consultar valores de variables o ejecutar paso a paso desde ese punto.
* **Evaluar / modificar:** permite obtener el valor de una variable en un punto en el que la ejecución del programa se ha pausado. Se puede modificar el valor de dicha variable para comprobar, por ejemplo, que si tuviera otro valor el comportamiento también sería el esperado.

# Los algoritmos

## ¿Qué es un algoritmo??

Es un conjunto finito de reglas que dan una secuencia de operaciones para resolver todos los problemas de un tipo dado. De forma más sencilla, podemos decir que un algoritmo es un conjunto de pasos que nos permite obtener un dato. Además debe cumplir estas condiciones:

* **Finitud:** el algoritmo debe acabar tras un número finito de pasos. Es más, es casi fundamental que sea en un número razonable de pasos.
* **Definibilidad:** el algoritmo debe definirse de forma precisa para cada paso, es decir, hay que evitar toda ambigüedad al definir cada instrucción. Puesto que el lenguaje humano es impreciso, los algoritmos se expresan mediante un lenguaje formal, ya sea matemático o de programación para un computador.
* **Entrada:** el algoritmo tendrá cero o más entradas, es decir, cantidades dadas antes de empezar el algoritmo. Estas cantidades pertenecen además, a conjuntos especificados de objetos. Por ejemplo, pueden ser cadenas de caracteres, enteros, naturales, fraccionarios, etc. Se trata siempre de cantidades representativas del mundo real, expresadas de tal forma que sean aptas para su interpretación por el computador.
* **Salida:** el algoritmo tiene una o más salidas, en relación con las entradas.
* **Efectividad:** se entiende por esto que una personasea capaz de realizar el algoritmo de modo exacto y sin ayuda de una máquina en un lapso de tiempo finito.

A menudo, los algoritmos requieren una organización bastante compleja de los datos, y es necesario un estudio previo de las estructura de datos fundamentales. Dichas estructuras pueden implementarse de diferentes maneras, y es más, existen algoritmos para implementar dichas estructuras. El uso de estructuras de datos adecuadas pueden hacer trivial el diseño de un algoritmo, o un algoritmo muy complejo puede usar estructuras de datos muy simples.

Uno de los algoritmos más antiguos conocidos es el algoritmo de Euclides. El término algoritmo proviene del matemático Muhammad ibn Musa al-Khwarizmi, que vivió aproximadamente entre los años 780 y 850 D.C. en la actual nación Iraní. Él describió la realización de operaciones elementales en el sistema de numeración decimal. De al-Khwarizmi se obtuvo la derivación de algoritmo.

## Clasificación de algoritmos

* **Algoritmo determinista:** en cada paso del algoritmo, se determina de forma única el siguiente paso.
* **Algoritmo no determinista:** deben decidir en cada paso de la ejecución entre varias alternativas y agotarlas todas antes de encontrar la solución.

Todo algoritmo tiene una serie de características, entre otras, que requiere una serie de recursos, algo que es fundamental considerar a la hora de implementarlos en una máquina, estos recursos son principalmente:

* **El tiempo:** período transcurrido entre el inicio y la finalización del algoritmo.
* **La memoria:** la cantidad (la medida varía según la máquina) que necesita el algoritmo para su ejecución.

Obviamente, la capacidad y el diseño de la máquina pueden afectar al diseño del algoritmo.

* **En general**, la mayoría de los problemas tienen un parámetro de entrada que es el número de datos que hay que tratar, esto es, *N*. La cantidad de recursos del algoritmo es tratada como una función de N. De esta manera puede establecerse un tiempo de ejecución del algoritmo que suele ser proporcional a una de las siguientes funciones:
* **Tiempo de ejecución constante:** Significa que la mayoría de las instrucciones se ejecutan una o muy pocas veces.
* **Log N:** Tiempo de ejecución logarítmico. Se puede considerar como una gran constante. La base del logaritmo (en informática la más común es la base 2) cambia la constante, pero no demasiado. El programa es más lento cuanto más crezca N, pero es inapreciable, pues Log N no se duplica hasta que N llegue a N2.
* **N:** Tiempo de ejecución lineal. Un caso en el que N valga 40, tardará el doble que otro en que N valga 20. Un ejemplo sería un algoritmo que lee N números enteros y devuelve la media aritmética.
* **N Log N:** El tiempo de ejecución es N Log N. Es común encontrarlo en algoritmos. Si N se duplica, el tiempo de ejecución es ligeramente mayor del doble.
* **N2:** Tiempo de ejecución cuadrático. Suele ser habitual cuando se tratan pares de elementos de datos, como por ejemplo un bucle anidado doble. Si N se duplica, el tiempo de ejecución aumenta cuatro veces.
* **N3:** Tiempo de ejecución cúbico. Como ejemplo se puede dar el de un bucle anidado triple. Si N se duplica, el tiempo de ejecución se multiplica por ocho.
* **2N:** Tiempo de ejecución exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Si N se duplica, el tiempo de ejecución se eleva al cuadrado.
* **Algoritmos polinomiales:** aquellos que son proporcionales a **Nk.** Son en general factibles.
* **Algoritmos exponenciales:** aquellos que son proporcionales a kN. En general son no factibles salvo un tamaño de entrada muy reducido.

## Pasos para diseñar un programa: Forma 2

**Aunque el proceso de diseñar programas es esencialmente un proceso creativo, se pueden considerar una serie de fases o pasos comunes, que generalmente deben seguir todos los programadores.**

**Las siguientes son las etapas que se deben cumplir para resolver con éxito un problema de programación:**

* **Definición del problema**
* **Análisis del problema**
* **Selección de la mejor alternativa**
* **Diagramación**
* **Prueba de escritorio**
* **Codificación**
* **Transcripción**
* **Compilación**
* **Pruebas de computador**
* **Documentación externa**

### ****Definición del problema****

Está dada por el enunciado del problema, el cuál debe ser claro y completo, es importante que conozcamos exactamente qué se desea del computador. Mientras que esto no se comprenda, no tiene caso pasar a la siguiente etapa.

### ****Análisis del problema****

Entendido el problema (¿qué se desea obtener del computador?), para resolverlo es preciso analizar:

* Los datos o resultados que se esperan.
* Los datos de entrada que nos suministran.
* El proceso al que se requiere someter esos datos a fin de obtener los resultados esperados. Áreas de trabajo, fórmulas y otros recursos necesarios.

Una recomendación muy práctica, es el que el programador se ponga en el lugar del computador, y analizar qué es necesario que le ordenen y en qué secuencia, para poder producir los resultados esperados.

### ****Selección de la mejor alternativa****

Analizado el problema, posiblemente se tengan varias formas de resolverlo. Lo importante es determinar cuál es la mejor alternativa: la que produce los resultados esperados en el menor tiempo y al menor costo. Claro que aquí también es muy válido el principio de que las cosas siempre se podrán hacer de una mejor forma.

### ****Diagramación****

Una vez que se sabe cómo resolver el problema, se procede a dibujar gráficamente la lógica de la alternativa seleccionada. Eso es precisamente un Diagrama de Flujo: la representación gráfica de una secuencia lógica de pasos a cumplir, por el computador para producir un resultado esperado.

La experiencia demuestra, que resulta muy útil trasladar esos pasos lógicos planteados en el diagrama a frases que indiquen lo mismo; es decir, hacer una codificación del programa pero utilizando instrucciones en Español. Como si se le estuviera hablando al computador, esto es lo que se denomina Algoritmo o pseudocódigo.

Cuando el programador logre la habilidad para desarrollar programas, es posible que no necesite elaborar el diagrama de flujo; en su lugar puede proceder a hacer directamente el pseudocódigo del programa.

### ****Prueba de escritorio****

Para cerciorarse de que el diagrama (y/o el pseudocódigo) está bien, y para garantizar que el programa codificado luego también funcione correctamente, es conveniente someterlo a una prueba de escritorio. Esta prueba consiste en que se establecen diferentes datos de entrada al programa y se sigue la secuencia indicada en el diagrama, hasta obtener los resultados. El análisis de estos indicará si el diagrama es correcto o si hay necesidad de hacer ajustes (volver al paso 4). Se recomienda dar diferentes datos de entrada y considerar todos los posibles casos, aun los de excepción o no esperados, para asegurarse que el programa no producirá errores en ejecución cuando se presenten estos casos.

### ****Codificación****

Una vez que se verifique el diagrama mediante las pruebas de escritorio, se codifica el programa en el lenguaje de computador seleccionado. Esto es, colocar cada paso del diagrama en una instrucción o sentencia, utilizando un lenguaje que el computador reconoce.

Todos los lenguajes de programación proveen facilidades para incluir líneas de comentarios en los programas. Estos comentarios aclaran lo que se ordena al computador y facilitan entender el programa. Puesto que estos comentarios no son tenidos en cuenta como instrucciones, y aparecen en los listados del programa, resulta muy conveniente agregar abundantes comentarios a todo programa que codifiquemos. Esto es lo que se denomina Documentación Interna.

### ****Transcripción****

Es necesario que se lleve el programa codificado a un medio que sea aceptado como entrada por el computador: perforado en tarjetas, grabado en un disco flexible o grabado en un disco duro. Este programa es el que se conoce como Programa Fuente.

### ****Compilación****

Se procede a utilizar a continuación un programa de computador llamado Compilador o Traductor, el cuál analiza todo el programa fuente y detecta errores de sintaxis ocasionados por fallas en la codificación o en la trascripción. Las fallas de lógica que pueda tener un programa fuente no son detectadas por el compilador. Cuando no hay errores graves en la compilación, el compilador traduce cada instrucción del programa fuente a instrucciones propias de la máquina (Lenguaje de Maquina), creando el Programa Objeto.

Algunos computadores utilizan Interpretadores, (Generalmente para el Lenguaje Basic), en reemplazo de programas compiladores. La diferencia consiste en que el interpretador recibe, desde una terminal, sólo una instrucción a la vez, la analiza y, si está bien, la convierte al formato propio de la máquina. Si la instrucción tiene algún error, el interpretador llama la atención de la persona para que corrija dicha instrucción.

Como resultado de la corrida del compilador, podemos obtener varios listados:

* Listado del programa fuente
* Listado de los errores detectados
* Listado de campos utilizados, etc.

Los errores se deben corregir sobre el mismo programa fuente, ya sea reemplazando las tarjetas mal perforadas o regrabando en el disco flexible o disco duro. Este paso de la compilación se repite hasta eliminar todos los errores y obtener el programa ejecutable.

### ****Pruebas de computador****

Cuando se tiene un programa ejecutable (en lenguaje de máquina), se le ordena al computador que lo ejecute, para lo cual se ingresan datos de prueba, como se hizo en la prueba de escritorio (paso 5). Los resultados obtenidos se analizan, luego de lo cual puede ocurrir cualquiera de estas situaciones:

* La lógica del programa está bien, pero hay errores sencillos, los cuales se corrigen modificando algunas instrucciones o incluyendo unas nuevas. El proceso debe repetirse desde el paso 6.
* Hay errores ocasionados por fallas en la lógica, lo que obliga a regresar a los pasos 4 y 5 para revisión y modificación del diagrama.
* Hay errores muy graves y lo más aconsejable es regresar al paso 2 para analizar nuevamente el problema, y repetir todo el proceso.
* No hay errores y los resultados son los esperados. En este caso, el programa se puede guardar permanentemente en una librería o biblioteca del computador, para sacarlo de allí cuando se necesite ejecutar nuevamente.

### ****Documentación externa****

Cuando el programa está listo para ejecutar, es conveniente hacer su documentación externa, siguiendo las normas de la instalación o las recomendaciones indicadas por el profesor, una buena documentación incluye siempre:

* Enunciado del problema
* Diagrama de pasada
* Narrativo con la descripción de la solución
* Relación de las variables o campos utilizados en el
* programa, cada uno con su respectiva función
* Diagrama del programa
* Listado de la última compilación
* Resultados de la ejecución del programa.

## Algoritmo o pseudo código

### Tipos de algoritmos

Existen dos tipos de algoritmos:

* Los algoritmos informales son aquellos que se realizan por medio de palabras sencillas tal como se comunican las personas comúnmente.
* Los algoritmos formales son los que se pueden resolver mediante uso de unas instrucciones.

**Ejemplo 1:** Diseñar un algoritmo para preparar una limonada.

Nombre \_ algoritmo Limonada

**INICIO**

**PASO 1** Llenar una jarra con un litro de agua  
 **PASO 2** Echar el jugo de tres limones  
 **PASO 3** Echar cuatro cucharadas de azúcar  
 **PASO 4** Revolver el agua hasta disolver completamente el azúcar

**FIN**

**Ejemplo 2:** Diseñar un algoritmo que permita hallar la suma y el promedio de tres números.

**INICIO**

**PASO 1** leer numero1, numero2, numero3  
 **PASO 2** sumar = numero1 + numero2 + numero3  
 **PASO 3** promedio = suma / 3  
 **PASO 4** imprimir suma, promedio

**FIN**

Notas:

* El término LEER significa obtener un dato de algún dispositivo de entrada como el teclado, y almacenarlo en una variable.
* Una variable es una localización en la memoria que tiene un nombre y cuyo contenido puede cambiar a lo largo de la ejecución de un programa. Así numero1, numero2 y numero3 son variables.
* El término IMPRIMIR significa mostrar el valor de una variable en algún dispositivo de salida, como la pantalla.

**Ejemplo 3:** Diseñar un algoritmo para freír un huevo.

**INICIO**

**PASO 1** Buscar ingredientes

**PASO 2** Sí encuentro todos los ingredientes

**PASO 3** Prender estufa

**PASO 4** Poner a derretir la mantequilla

**PASO 5** Picar ingredientes

**PASO 6** Echar ingredientes al sartén

**PASO 7** Esperar que esté listo

**PASO 8** Apagar estufa

**PASO 9** Servir huevo

**FIN**

### Ejercicios sobre algoritmos informales

* Desarrollar un algoritmo que permita tomar un avión
* Desarrollar un algoritmo que permita manejar un auto
* Desarrollar un algoritmo que permita seleccionar un tema
* Desarrollar un algoritmo que permita buscar un libro
* Desarrollar un algoritmo que permita hacer un pastel.
* Desarrollar un algoritmo que permita dar un beso.
* Desarrollar un algoritmo que permita maquillarse.
* Desarrollar un algoritmo que permita tomar una fotografía
* Desarrollar un algoritmo que permita cambiar una llanta.
* Desarrollar un algoritmo que permita cambiar un bombillo.
* Desarrollar un algoritmo que permita ir a cine.
* Desarrollar un algoritmo que permita cambiar un cuadro
* Desarrollar un algoritmo que permita grabar un disco.
* Desarrollar un algoritmo que permita presentar un examen
* Desarrollar un algoritmo que permita freír un huevo
* Desarrollar un algoritmo que permita pintar una casa
* Desarrollar un algoritmo que permita seleccionar una camisa
* Desarrollar un algoritmo para colocarnos una camisa.

### Algoritmos formales

**El primer programa en pseudocódigo**

**Normas**:

* Comienza con el nombre del programa
* Definición de las variables globales y locales
* Todos los programas comienzan por la palabra Inicio, que da apertura al programa y a los procesos correspondientes, seguido de un grupo de sentencias que mecánicamente están organizadas para dar viabilidad y cohesión al desarrollo del mismo, y de esta manera poder recibir los datos deseados.
* Por tanto debemos recordar que un grupo de sentencias va entre llaves {...}, las cuales las agrupa para hacer referencia a:
* El hecho de que son varias implica su agrupamiento.
* Un ordenamiento sintáctico.
* Fácil compresión de las líneas de código.
* Las sentencias tienen dependencia mutua para arrojar uno o más resultados, como se verá más adelante en Estructuras de Control Repetitivas.
* Una sentencia simple aislada no necesita llaves.
* Cada sentencia termina por un ‘**;**´ **(Separador de sentencias)**
* Como norma se declaran todas las variables al inicio del programa y se les asigna el valor correspondiente dependiendo del tipo (entero, real, carácter) si es necesario. **Ejemplo**:

**Formato** entero s;

#### Instrucciones para Entrada y Salida de datos

Las instrucciones e**scribir( )** o e**scribe( ).** Presentan por el dispositivo de salida estándar (pantalla), las expresiones, valores, variables o cadenas contenidas en la función, por **ejemplo**:

**escribe** (“Introducir el valor de la base”); //muestra en pantalla la cadena contenida.

**escribe** (“El resultado es”, **p**); //imprime un mensaje y el valor que contiene la variable **p** de tipo entero.

**escribe** (**p**); // solamente imprime el valor de la variable **p.**

Una sintaxis especial se requiere si se desea presentar las expresiones conjuntamente.

**Cadenas y variables:**

Se escribe la cadena, se separa la variable con una coma,después va otra cadena entonces se pone otra coma y se escribe la **cadena.**

**Pseudocódigo:**

escribe (“base”, b, “por la altura”, a, “es la superficie”, s);

**Lenguaje c**

printf(“base %d”, b, ”por la altura %d”, a, ”es la superficie %d”, s)

#### Tipos de formatos

Manejo de datos numéricos:

|  |  |
| --- | --- |
| Identificador | Descripción |
| %d | Para el manejo de números enteros |
| %i | Para el manejo de números enteros |
| %f | Para el manejo de números reales |
| %o | Para el manejo de números en Octal |
| %x | Para el manejo de números en Hexadecimal |
| %p | Para el manejo de apuntadores |

Manejo de caracteres y cadenas:

|  |  |
| --- | --- |
| Identificador | Descripción |
| %s | Para el manejo de cadenas |
| %c | Para el manejo de caracteres |

Manejo de presentación por pantalla:

|  |  |
| --- | --- |
| Identificador | Descripción |
| %v | Para visualizar el texto o números verticalmente |
| %h | Para visualizar el texto o números horizontalmente |

**Ejemplo:**

//Visualiza un mensaje por pantalla:

printf(“hola como estas”);

int A=1;

//Visualiza un mensaje y el contenido de la variable A de tipo entero:

printf(“el valor es %d”, A);

**Instrucciones de Entrada**

Las instrucciones l**eer( )** y **scanf ()** toman los valores de forma interactiva desde el dispositivo de entrada estándar (teclado), introduciéndolos en las variables que forman parte de la instrucción. Los tipos de datos introducidos deben coincidir con los de las variables que los recogen. Hay dos formas de usarlas:

**Formato pseudocódigo;**

...

x = **leer**() **;**

leer(x);

Cuando se quiere recoger más de un dato se utiliza esta forma:

**leer**(x, y)

Se respetará el orden de recogida con cual si se introduce desde el teclado 12 y 34, la variable x =12 y la y =34.

**Formato scanf**

**scanf**(“tipo\_formato”**,** direccionNom\_Variable);

**scanf**(“%d”, &h);

La dirección indica la posición donde queda almacenada la variable. En este caso la variable h

**Ejemplo:**

int h;

printf(“Digite un numero”);

scanf(“%d”,&h)

Suponga que se quiere hacer un programa que calcule el área de un triángulo. Primero se debe pensar que A = (b\*h) / 2, donde las variables **A**, **b**, **h** seguidamente representan **área**, **base** y **altura**. Por lo tanto se deben pedir dos datos y presentar un resultado.

Una posible solución usando una secuencia de sentencias y expresiones sería:

**Pseudocódigo**

|  |
| --- |
| Área del triángulo // nombre del programa  Inicio  Entero a, b, h; // Declaramos las variables  Escribe(“Introducir el valor dela Base y altura:”); //mensaje en pantalla  Leer (b); // Leemos la base  Leer (h); // Leemos la altura  a = (b \* h)/2; // calculamos  //Presentamos el resultado en la misma línea del mensaje  Escribe (“la superficie es: “, a);  fin |

**LENGUAJE C**

|  |  |
| --- | --- |
| 1 | main() { |
| 2 | int a, b, h; // Declaro las variables |
| 3 | //mensaje en pantalla |
| 4 | printf(“Introducir el valor de la Base y altura: ”); |
| 5 | scanf (“%d”, b); // Leemos la base |
| 6 | scanf (“%d”, h); // Leemos la altura |
| 7 | a = (b \* h)/2; // calculamos... |
| 8 | //Presentamos el resultado: |
| 9 | printf (“la superficie es : %d”, a); |
| 10 | } |

**Ejemplo de Algoritmo formales**

|  |  |
| --- | --- |
| Algoritmo Área \_ triangulo | 1. Nombre Algoritmo |
| **Inicio** | 2. Inicio del algoritmo |
| **escriba** (“ la base y la altura ”) leer: base, altura | 3. Datos de entrada |
| **area**= base \* altura/2 | 4. Proceso |
| **escriba** (“El área del triángulo es ”, **area**) | 5. Datos de salida |
| Fin | 6. Final del algoritmo |

Algunos ejemplos de algoritmos son: las recetas de cocina, instrucciones para cambiar una llanta, etc.

### Problemas de pseudocódigo:

Declaración de Variables y Programas secuenciales

1. Decir si son correctos o no los siguientes identificadores de variables:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Correcto | | |  | Incorrecto | | |
|  | |  | | |  |  | | |
| Contad |  | |  |  | | |  |  |
| N\_Bytes |  | |  |  | | |  |  |
| 2oral |  | |  |  | | |  |  |
| 3M\_Atrim |  | |  |  | | |  |  |
| &6Alor” |  |  |  |  |  |  |  |  |
| 6”#$9i |  |  |  |  |  |  |  |  |
| #32gt |  |  |  |  |  |  |  |  |

1. Realizar en pseudocódigo un programa que calcule de forma individual la velocidad de 4 cuerpos, introduciendo por teclado el espacio y el tiempo, posteriormente imprimiéndolas por pantalla al igual que la distancia tiempo y velocidad totales.

Pseudocódigo

**Inicio**

real: e1, e2, e3, e4, t1, t2, t3, t4, v1, v2, v3, v4;

real: espacio\_total, tiempo\_total, velocidad \_total;

escribe (“digite la distancia cuerpo 1”); leer(e1);

escribe (“digite tiempo del cuerpo 1”); leer(t1);

escribe (“digite distancia cuerpo 2”); leer(e2);

escribe (“digite tiempo del cuerpo 2”); leer(t2);

escribe (“digite distancia cuerpo 3”); leer(e3);

escribe (“digite tiempo del cuerpo 3”); leer(t3);

escribe (“digite distancia cuerpo 4”); leer(e4);

escribe (“digite tiempo del cuerpo 4”); leer(t4);

v1 = e1 / t1;

v2 = e2 / t2;

v3 = e3 / t3;

v4 = e4 / t4;

espacio\_total = e1+e2+e3+e4;

tiempo\_total = t1+t2+t3+t4;

velocidad \_total = v1+ v2+ v3+ v4;

escribe (“velocidad del cuerpo 1:”, v1);

escribe (“velocidad del cuerpo 2:”, v2); escribe (“velocidad del cuerpo 3:”, v3);

escribe (“velocidad del cuerpo 4:”, v4);

escribe (“velocidad total de los cuatro cuerpos:”, velocidad\_total);

escribe (“espacio total de los cuerpos:”, espacio\_total);

escribe (“tiempo total de los cuerpos:”, tiempo\_total);

**Fin**

1. Realizar en pseudocódigo un programa que calcule el perímetro y el área total de tres circunferencias sabiendo que la 1ª de ellas tiene radio R que será introducido por teclado, la 2ª tiene radio 2R y la 3ª tiene radio 3R

**Inicio**

real: r, pi, a1, a2, a3, p1, p2, p3, area\_total, perímetro\_total;

escribe (“Digite el valor del radio: ”); leer (r);

escribe (“Circunferencia 1:”);

pi = 3.1416; a1= pi \* (r \* r);

p1 = 2 \* (pi \* r);

escribe(“Su perímetro es: ”, p1);

escribe(“Su área es: ”, a1);

escribe(“Circunferencia 2:”);

r = 2 \* r; a2 = pi \* (r \* r);

p2 = 2 \* (pi \* r);

escribe(“Su perímetro es: ”, p2);

escribe(“Su área es: ”, a2);

escribe(“Circunferencia 3”);

r = 3 \* r; a3 = pi \* (r \* r); p3 = 2 \* ( pi \* r);

escribe(“Su perímetro es: ”, p3);

escribe(“Su área es :”, a3);

perímetro\_total = p1+ p2+ p3;

area\_total = a1+ a2+ a3;

escribe(“Perímetro total de las circunferencias:”, perímetro\_total);

escribe(“Area total de las circunferencias:”, area\_total);

**Fin**

## Tipos de representación de un algoritmo

### Diagrama de flujo

Es una representación gráfica de una secuencia lógica de pasos a cumplir por el computador para producir un resultado esperado.

La experiencia ha demostrado que resulta muy útil trasladar esos pasos lógicos, planteados en el diagrama a frases que indiquen lo mismo; es decir, hacer una codificación del programa pero utilizando instrucciones en Español. Como si se le estuviera hablando al computador. Esto es lo que se denomina Algoritmo o Pseudo código.

### Gráficos - Símbolos

Entrada y salida de datos

Inicio

Salida de datos

Entrada/salida de datos

Flujo de datos Dirección

DecisiónMúltiple

Decisión

Conectores de diagramas

Subrutina

Proceso

|  |  |
| --- | --- |
|  | **Terminal: representa el comienzo “inicio”, final “fin” de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en el programa.** |
|  | **Entrada / salida: cualquier tipo de introducción de datos en la memoria desde los periféricos (entradas o salidas).** |
|  | **Proceso: cualquier tipo de operación que pueda originar un cambio de valor, operaciones aritméticas etc.** |
|  | **Decisión: Indica operaciones lógicas o de comparación entre datos, normalmente tiene dos salidas (Si y No), pero puede tener tres o más salidas.** |
|  | **Decisión múltiple: en función del resultado de la comparación se seguirá uno de los diferentes caminos.** |
|  | **Conector: sirve para realizar la conexión de dos partes cualquiera, se refiere a la conexión por páginas.** |
|  | **Indicador de dirección o línea de flujo: indica el sentido de la ejecución de las operaciones.** |
|  | **Línea conectora: sirve de unión entre dos símbolos.** |
|  | **Llamada subrutina o a un proceso predeterminado: una subrutina es un módulo independiente del programa principal que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal.** |
|  | **Pantalla: se utiliza en ocasiones en lugar del símbolo de E/S.** |
|  | **Teclado: entrada de datos por teclado.** |
|  | **Impresora: Se utiliza para entrada y salida de datos por impresora** |
|  | **Comentario: Se utiliza para añadir comentarios y se puede dibujar en cualquier lado del símbolo** |

### Combinaciones de los símbolos

|  |  |
| --- | --- |
| Secuencial:  Acción  Acción  Acción | Selección Simple:  Verdadera  A es mayor que b  A, B  A>B  “En la selección simple, si se cumple la condición se ejecutan las acciones” |
| Selección Compuesta:  Verdadera  Falsa  A, B  A es mayor que b  A>B  B es mayor que a  “En la selección compuesta, si se cumple la condición se ejecutan las acciones del bloque. Si no se cumple se ejecutan otras acciones diferentes” | |
| Selección Múltiple:  Caso 1  Caso 6  Caso 2  2  Caso 3  Caso 4  Caso 5  Condición  dición  Acción 1  Acción 2  Acción 3  Acción 4  Acción 5  Acción 6 | |

## Instrucciones repetitivas o ciclos

**MIENTRAS** condición sea *verdadera* **HAGA**

(Mientras Haga)

“Se repiten las acciones hasta que la condición sea falsa y al ser falsa sale del ciclo”

Acción

Condición

Verdadero

**HAGA MIENTRAS** condición sea falsa

(Haga Mientras)

“Se repiten las acciones hasta que la condición sea verdadera y al ser verdadera sale del ciclo”

Verdadero

Falsa

Condición

Acción

## Ejemplo de algoritmos

1. Diagrama de flujo que compara si dos números son iguales, o si el primero es mayor el segundo.

Verdadera

Falsa

X = Y

X > Y

No son iguales *X* es mayor que *Y*

X, Y

X, Y

Son iguales

1. Generar los números pares e impares del 4 al 400

False

False

True

True

A > 0

True

A == 0

A = X

X != 400

X = 4

Inicio

A = A - 2

El número es par, x

El número es impar, x

X = X + 1

1. Programa que encuentra el mayor, el del medio y menor de tres números

F

T

F

T

False

T

F

T

F

True

x, a, z

x, a, z

x>a

x>z

a>z

a>z

a, z, x

z, a, x

a, x, z

z>x

x, z, a

z, x, a

x, a, z

### Ejercicios del tema 1: diseño de algoritmos simples

En los siguientes ejercicios se pretende descubrir un algoritmo para solucionar distintos problemas, de forma que sólo se empleen instrucciones elementales: Operaciones aritméticas (suma, resta, multiplicación...), asignación, entrada y salida, bucles, selección.

Para cada ejercicio intente examinar cómo se efectúa la operación 'a mano'. Se trata de ver que cuando se soluciona un ejercicio, también se soluciona ejecutando distintas operaciones una a una. Después, trate de descubrir las operaciones repetitivas, que serán los bucles, y las operaciones en las que pregunta por una condición (Si se cumple que... entonces... y si no...), que serán operaciones de selección. Por último, construya el diagrama de flujo siguiendo una forma de representación de algoritmos.

1. Diseñe un diagrama de flujo que halle el área y el perímetro de un rectángulo. Considere las siguientes fórmulas: área = base x altura, perímetro = 2 x (base + altura).
2. Diseñe un diagrama de flujo para convertir una longitud dada en centímetros a pies. Considere que:   
   1 pie = 30.48 centímetros.
3. Diseñar un algoritmo que lea 3 números enteros positivos distintos e imprima el mayor valor de los 3. Supondremos que los 3 números que se leen son distintos.
4. Efectuar un algoritmo que averigüe si un número positivo es par o impar. El programa leerá un número entero positivo y dará el resultado.
5. Diseñar un algoritmo para sumar los 100 primeros números naturales. El algoritmo no leerá ningún valor. Simplemente mostrará el resultado.
6. Diseñar un algoritmo para convertir temperaturas en grados Fahrenheit a grados Celsius. El programa terminará cuando lea una temperatura igual a 999. La relación entre grados Fahrenheit (F) y grados Celsius (C) viene dada por: C= 5/9 (F-32).
7. Diseñar un algoritmo que lea 2 puntos en el plano (4 números reales): (X1, Y1) y (X2, Y2) y devuelva la distancia euclídea entre ellos. Para ello se usará el teorema de Pitágoras.
8. El sueldo neto de un vendedor se calcula como la suma de un sueldo básico de S/.250 más el 12% del monto total vendido. Diseñe un algoritmo que determine el sueldo neto de un vendedor sabiendo que hizo tres ventas en el mes.
9. Escribir un diagrama de flujo para un algoritmo que calcule la media de varios números, los que introduzca el usuario por teclado, y saque el resultado por pantalla. Nota: el primer carácter no numérico que introduzca el usuario indicará que no va a introducir más números. Usar tres variables: contador, suma y resultado en expresiones aritméticas sencillas. Considerar primitivas las acciones de leer por teclado y escribir en pantalla.
10. Escribir algoritmos en pseudocódigo que calculen y saquen por pantalla el máximo de tres números introducidos por teclado.
11. Dados *X* y *Y*, verificar si *X* es divisible por *Y*.
12. Sumar los cuadrados de los primeros N números naturales.
13. Leer N números y obtener el promedio solo de los números pares de la lista.
14. Introducir un conjunto de M números, determinar la cantidad de números positivos y negativos del conjunto.

### Realizar el diagrama de flujo de los siguientes problemas

1. Realizar un programa que compare dos números ingresados por teclado, el resultado del programa deber ser correcto, si los números son iguales, de lo contrario debe sacar un mensaje especificando que no son iguales y escribir si es mayor o menor que él número comparado.
2. Generar los números pares e impares de 40 hasta 400.
3. Realizar un algoritmo para vigilar una ventana las veinticuatros horas.
4. Realizar un algoritmo que lea 3 números enteros y mostrarlos ascendentemente.
5. Realizar un programa que lea 4 números enteros por teclado y saque el menor y mayor de los cuatros números.
6. Realizar un programa que lea n números y saque la cantidad de números positivos y negativos que existen utilizando un ciclo infinito,
7. Leer dos números enteros y mostrar todos los números comprendidos entre ellos.
8. Leer una serie de números enteros hasta que digite él número 99 y determinar La cantidad de número que ingreso por teclado.
9. La sumatoria de números pares positivos.
10. La sumatoria de números impares negativos y la cantidad de ceros.
11. Mostrar la tabla de multiplicar del 1 hasta 13.
12. Mostrar la tabla de multiplicar de un número dado.
13. Mostrar los primeros cien números múltiplos de 3.
14. Leer dos números y mostrar todos los números múltiplos de cinco comprendidos entre ellos.
15. Realizar un programa que realice la siguientes conversión de pies a metro y de metros a pies
16. Escriba un programa que realice las siguientes conversiones
17. metros a kilómetros
18. kilómetros a metros
19. decímetros a hectómetros
20. decímetros a milímetros
21. hectómetros a decímetros o milímetros
22. Utilizando ciclos anidados generar las siguientes parejas ordenadas.
23. 0 1 2 3 4 5 6 7 8 9 b) 0 1 2 3 4 5 6 7 8

1 1 2 2 3 3 4 4 5 5 1 1 0 0 0 1 0 0 1

c) 1 2 3 4 5 6 7 8 9

1 1 1 2 2 2 3 3 3

1. Realizar un programa que determine la sumatoria de la siguiente serie

a) 2 5 7 10 12 17 ... 1800

En este capítulo se dará una introducción a la programación en C.

Se verán conceptos acerca de la entrada y salida por pantalla de variables y mensajes, además de sintaxis básicas para crear un programa principal, algoritmos de ejemplo, bloques condicionales, y demás conceptos que ayudarán a entender mejor la programación en lenguaje C

# CAPÍTULO II:

# Programación en C

# Programación en C

## Historia del lenguaje C

C es un lenguaje equilibrado de propósitos generales que se desarrolló a partir de estas raíces. Su definición aparece en 1978 en “C Reference Manual'' del libro “The C Programming Language”, de Brian W. Kernighan y Dennis M. Ritchie, pero el estándar recomendable más reciente apareció en junio de 1983, en el documento de los Laboratorios Bell titulado The C Programming Language-Reference Manual, escrito por Dennis M. Ritchie

El lenguaje C está inspirado en el lenguaje B, escrito por Ken Thompson en 1970 con intención de recodificar 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 rescribiendo el UNIX  en dicho lenguaje.  La novedad que proporcionó el lenguaje C sobre el B fue el diseño de tipos de estructura de datos.

AT&T lo enseñó como un compilador (comp. C) llamado K&R C que junto con el sistema operativo UNIX empezaron a invadir universidades. Después, cada persona que adquiría una copia de UNIX recibía un compilador de C gratis. El lenguaje más popular fue C. Por lo tanto UNIX fue escrito en C. Entonces si una persona quería entender UNIX, tenía que aprender C.

La característica principal era su gratuidad y entonces nadie se sentía presionado a aprenderlo. ¿Cuál fue el resultado?, un gran estándar.

Luego que C se convirtió en un gran estándar, entonces las compañías introducían sus propios compiladores C. Añadiendo la capacidad de ejecutarse en otros sistemas operativos que no fuesen UNIX.

Cada uno de estos compiladores introducía ensanchamientos diseñados para mejorar las limitaciones que mostraba el modelo original. Pero las modificaciones que cada quien hacía, traían como resultado la incompatibilidad de las versiones entre sí. Gracias a esto incrementó la demanda por un estándar a nivel nacional. Entonces en 1987 nació el primer estándar "The American National Standars Institute (ANSI) version of C" esta versión fue mejor conocida como ANSI C o C estándar.

El C puede ser compilado al lenguaje de maquina en casi todas las computadoras.  Por ejemplo el Unix está escrito en C, se ejecuta y compila en una amplia variedad de micro, mini y macro computadoras.

En C se programa con una serie de funciones  que se llaman unas a otras para el procesamiento.  Aun el cuerpo del programa es una función  denominada flexible, permitiendo a los programadores la elección entre el uso de la biblioteca estándar que se provee con el compilador, el uso de funciones de terceros creadas por otros proveedores de C, o el desarrollo de sus propias funciones.

Comparado con otros lenguajes de programación de alto nivel, C parece complicado.  Su apariencia intrincada se debe a su extrema flexibilidad.

Las principales características de este lenguaje son:

* Programación ordenada.
* Facilidad de aprendizaje.
* Economía de sus expresiones.
* Gran cantidad en operadores y tipos de datos.
* Codificación en alto y bajo nivel simultáneamente.
* Reemplaza ventajosamente la programación en Asembler.
* Utilización natural  de las funciones primitivas del sistema.
* No está orientado a ningún área en especial.

Hay toda una serie de operaciones que puede hacerse con el lenguaje C, que realmente no están incluidas en el compilador propiamente dicho, sino que las realiza un preprocesador justo antes de cada compilación.  Los dos más importantes son #define (directriz de sustitución simbólica o de definición) e #include (directriz en el fichero fuente).

Finalmente C, que ha sido pensado para ser altamente transportable y para programar lo improgramable, tiene algunos pequeños inconvenientes como todos los lenguajes:

Carece de instrucciones de entrada / salida, de instrucciones para manejo  de cadenas de caracteres, con lo que este trabajo queda para la librería de rutinas, con la consiguiente pérdida de transponibilidad.

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.

## Tipos de datos básicos en lenguaje c

### Entero:

Los números enteros son todos aquellos números que solo tienen una parte entera y no tienen parte decimal, hacen parte del subconjunto finito de los números Naturales, el rango depende de cuantos bits utilice para codificar el número, normalmente 2 bytes, para números positivos.

Con 16 bits se pueden almacenar 216 = 65536 números enteros diferentes de 0 al 65535, y de -32768 al 32767 para números con signo.

Por ejemplo 2, 14, -199,....

Operaciones asociadas al tipo entero:

Como norma general las operaciones asociadas a un tipo cualquiera serán aquellas cuyo resultado sea un elemento del mismo tipo

Las operaciones asociadas serian +, -, \*, % o /, etc.

### Real:

Los números reales son todos aquellos números que tienen una parte entera y una parte decimal. Estos hacen parte de subconjunto de los números reales limitado no sólo en cuanto al tamaño, sino también en cuanto a la precisión.

Suelen ocupar 4, 6 ó 10 bytes. Se representan por medio de la mantisa, y un exponente (1E-3 = 0’001), 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).

El orden es de 10-39 hasta 1038.

Por ejemplo

6.9, 33.00123, 3E-34......

Las operaciones asociadas serian +, -, \*, % o /, etc.

### Lógico:

Es un dato que puede tomar dos valores:

Cierto y Falso. ‘1’ y ‘0’.

Operaciones asociadas a este tipo de dato son: Todos los operadores lógicos y de relación.

Operadores lógicos

|  |  |  |
| --- | --- | --- |
| Conjunción | **Y** | **&&** |
| Disyunción | O | || |
| Negación | NOT | ! |

Operadores de relación

|  |  |  |
| --- | --- | --- |
| Mayor que | > | “Cuando vamos a crear una condición podemos combinar tantos operadores como deseemos” |
| Menor que | < |
| Igual | == |
| Diferente | != |

Y sus combinaciones

|  |  |
| --- | --- |
| Mayor o igual que | >= |
| Menor o igual que | <= |

### Carácter:

Un carácter (letra) se guarda en un solo byte como un número entero, el correspondiente en el código ASCII, que se muestra en la Tabla para los caracteres estándar (existe un código ASCII extendido que utiliza los 256 valores)

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.

Los caracteres se representan entre comillas simples ‘a’, ‘D’... y abarcan:

‘0’ … ‘9’

‘a’ … ‘z’

‘A’ … ‘Z’

‘!’, ‘#’, ‘(‘, Etc.

Un espacio en blanco ‘ ‘ también es un carácter. Por tratarse de un tipo de datos enumerados y con una representación como la de un entero.

### Cadena:

Los datos de este tipo contendrán una serie finita de caracteres.

Podrán representarse de estas dos formas:

‘H’ ’O’ ’L’ ’A’ == “HOLA”

No es lo mismo “H” que ‘H’, ya que el primero es una cadena y el segundo un carácter. Las operaciones relacionadas con las cadenas son:

Concatenar(C1,C2), Longitud(C1), Extraer (C1,N)

Ejemplos:

C1 = “Hola”, C2 = “Pepeeerl”

C3 = Concatenar(C1,C2) //C3=”HolaPepeeerl”

X = Longitud(C2) //X=8

C3=Extraer(C1,3) C3=”Hol”

Entero, Real, Carácter, Cadena y Lógico son tipos predefinidos en la mayoría de los lenguajes de programación. Además el usuario podrá definir sus propios tipos de datos, si estos facilitan de alguna manera la resolución del problema.

## Tipos de variables básicas

En C tenemos los siguientes tipos de datos básicos: char, int, float y double. Los dos primeros son enteros y los otros dos del tipo real (admiten decimales). Estos cuatro tipos de datos se pueden convertir en algunos más cuando hacemos uso de los especificadores (o modificadores) en los tipos enteros. Al final, se nos quedan como aparecen en la tabla.

|  |  |  |
| --- | --- | --- |
| TIPO | BYTES | RANGO DE VALORES |
| Char | 1 | -128 a 127 |
| Signed char | 2 | 128 a 127 |
| Unsigned char | 1 | -255 a 255 |
| Short int | 2 | -32768 a 32767 |
| Signed short | 2 | -32768 a 32767 |
| Unsigned short | 2 | 0 a 65535 |
| Long int | 4 | -2147483648 a 2147483647 |
| Signed long | 4 | -2147483648 a 2147483647 |
| Unsigned long | 4 | 0 a 4294967295 |
| Int (16 bits) | 2 | -32768 a 32767 |
| Signed int (16bits) | 2 | -32768 a 32767 |
| Unsigned int (16 bits) | 2 | 0 a 65535 |
| Int (32 bits) | 4 | -2147483648 a 2147483647 |
| Signed int (32 bits) | 4 | -2147483648 a 2147483647 |
| Unsigned int (32 bits) | 4 | 0 a 4294967295 |
| Float | 4 | - 3.402823466 E + 38a3  +3.402823466 E + 38 |
| Double | 8 | -1.7976931348623158E + 308  1.7976931348623158E + 308 |

Como se puede observar en el cuadro anterior, los tipos enteros sin modificador de signo se corresponden a tipos con signo (**signed**).

Ahora bien, en algunos compiladores es posible indicar si queremos que los números sean considerados sin signo. Así pues, si queremos que nuestro programa se comporte igual independientemente del compilador con que haya sido compilado podemos hacerlo.

Algo parecido puede ocurrirnos con los **int**, que en los sistemas de 16 bits serán considerados como short y en los de 32 como **long**.

Si vamos a declarar una variable int que pueda salirse del rango de los short sería recomendable usar long (o long int) para que funcionara en cualquier sistema.

Todo esto para dar la portabilidad y confiabilidad al manejo de datos.

### Constantes

Una constante es un dato que permanece fijo dentro de un mismo programa. El término constante designa un valor específico y determinado, que se define al hacer un programa y que no cambia a lo largo del mismo. También a las constantes se les debe de colocar un nombre y deben ser de un determinado tipo de dato.

Existen dos tipos de constantes en C. Estos son los siguientes:

### Numéricas

Son aquellas con las cuales se pueden realizar operaciones de tipo aritmético.

### Enteras

Una constante entera es una sucesión de dígitos precedidos o no del signo (+) O (-).

Ejemplo de ello tenemos:

|  |  |
| --- | --- |
| 10 | 0 |
| 2 | -1 |
| 12350 | -13500 |

Note como las constantes enteras pueden ser positivas o negativas. Inclusive, es posible escribir una constante entera así: +12

Note también como no se usan separadores para los miles. El número 12350 se escribió simplemente así: 12350

Es bien importante tener en cuenta que no hace falta hacer declaraciones previas de las constantes enteras. Simplemente se escriben donde se necesitan.

Además, cabe notar que no se requieren símbolos especiales alrededor de las constantes enteras. Se escriben tal como son.

### Octales

Para las constantes enteras-octales se debe anteponer el número cero (0).

Ejemplo: 062, 035, 045.

### Hexadecimales

Para las constantes enteras hexadecimales se debe anteponer el cero y la equis (0X).

Ejemplo:

|  |  |
| --- | --- |
| 0x10 | 0x1A |
| 0x2F4 | 0X20B |

Note cómo las letras que representan dígitos del sistema hexadecimal, al igual que la equis que acompaña al cero, pueden ser escritas tanto en mayúscula como en minúscula. Para el computador es indiferente.

### Decimal

Para las constantes enteras-decimales de deben presentar tal cual son.

Ejemplo: 56,333 67,333123 78,0000 34.6565

### Reales

Una constante de tipo real equivale al valor de un número real usual, es decir, una cantidad formada por una parte entera y una fraccionaria.

Otro nombre como se conocen es datos de punto flotante

Las constantes de tipo flotante también se escriben de manera similar a la forma como las empleamos en aritmética convencional.

Ejemplo:

|  |  |
| --- | --- |
| 12.5 | 1.6 |
| -2341.189 | -0.146987 |

Por supuesto, el principal indicador de que se trata de constantes de punto flotante es el punto.

12.0

Son constantes de punto flotante. No importa que su parte fraccionaria sea cero, la sola presencia del punto la convierte en una constante de punto flotante.

Note como las constantes de punto flotante también pueden ser negativas o positivas y que se puede usar el signo más (+) explícitamente.

El número de dígitos decimales no está limitado y puede ser cualquiera. Sin embargo, si un número constante de punto flotante tiene demasiados dígitos (entre enteros y decimales) entonces el computador terminará haciendo una aproximación para efectos de su almacenamiento dentro de la memoria del computador.

Los números constantes de punto flotante también pueden ser escritos usando la notación científica exponencial.

Ejemplos de constantes de punto flotante escritas de esta manera son:

6.2e13

-.36E-15

-83.423e+12

1E3

El significado numérico de cada una de estas constantes en forma respectiva es:

6.2x1013 ó 62000000000000

-0.36x10-15 ó -0.00000000000000036

-83.426x1012 ó -83423000000000

1x103 ó 1000

En C están definidas dos clases de constantes reales que son:

### Float y double

Estas se definen normalmente en dígitos de precisión. Las magnitudes de estos dos tipos de reales dependen del método utilizado para representar los números en coma flotante.

Ejemplo:

float 64 🡪 Aproximadamente seis (6) dígitos de precisión.

double 🡪 Aproximadamente doce (12) dígitos de precisión.

### Alfanuméricas

Son conjuntos de caracteres con los cuales no se pueden realizar operaciones aritméticas.

### Simbólicas

Constantes son definidas en C para ampliar el ambiente de programación en el lenguaje, para hacer más óptima una aplicación determinada.

Para llevar a cabo este tipo de constantes se utiliza la instrucción #define de la siguiente manera:

#define cadena de caracteres

Nota: En este tipo de instrucciones no hay punto ni coma, además, puede existir cualquier número de blancos entre el #define y la cadena de caracteres.

La cadena de caracteres hace referencia a dos partes que son: una variable que va a ser constante durante la ejecución del programa y un valor determinado que va a tomar esa variable.

Ejemplo: #define N 10

El ejemplo anterior dejará constante la variable N con un valor de 10.

El #define es sumamente importante en lo que hace referencia a la portabilidad de los programas.

### Lógicas

Son las que provienen de realizar comparaciones lógicas. Se podría definir como un estado de 1 ó 0.

Ejemplo:

Datos lógicos: falso verdadero

Si No

1 0

### Variables

Una variable es algo que puede cambiar su valor dentro de un mismo programa (durante su ejecución). En sistemas es el nombre que se le da a un trozo de la memoria principal del computador. De tal manera que podamos depositar datos en ella. A las variables les debemos dar un nombre el cual debe iniciar siempre con una letra.

**Ejemplo:**

Si quiero llevar la cuenta de las personas que van ingresando a un cine, entonces requerimos de una variable, a la cual le asignamos un valor.

**Ejemplo:**

Persona \_ cine = 1 y lo habrán hecho 2, 3, 4 y así sucesivamente.

Debido a que los datos variables pueden tomar muchos valores durante la ejecución de un programa, no pueden tener un valor preciso como el de las constantes. En lugar de ello, debemos asignarles un nombre en forma similar a como lo hacemos en el álgebra.

De esta manera, una variable podría llamarse simplemente **X** o también podría llamarse **Y**

En realidad, se acostumbra que los nombres de las variables digan un poco más sobre su razón de existencia. Por ejemplo, para llevar la cuenta de las personas que ingresan al teatro, podríamos usar una variable como:

Personas para calcular el promedio de las personas que ingresaron a cine

Promedio *personas o P\_*CINE

En general, podemos usar cualquier nombre para las variables respetando unas reglas simples:

* Los nombres de las variables deben empezar con una letra (no importa si es minúscula o mayúscula) o con el símbolo de subrayado\_.
* Los nombres de las variables no pueden contener espacios ni símbolos especiales como &, ., (, + y así sucesivamente. Tampoco pueden contener letras tildadas o eñes.
* Los nombres de las variables pueden contener dígitos y más símbolos de subrayado. El valor numérico de los dígitos dentro de los nombres de las variables no tendrá ningún significado particular para el computador. Simplemente harán parte del nombre.
* No se pueden emplear como variables aquellas palabras que son consideradas propias del lenguaje y a las que se les llama

Palabras reservadas.

**NOTA: “**Una palabra reservada es una palabra usada por el lenguaje para algún fin particular y que por tanto, está reservada para él y no puede ser usada como nombre de variable ni de ninguna otra clase de objetos.**”**

## Lista de palabras reservadas

La lista de palabras reservadas por C ANSI estándar son tan sólo 32, las cuales se listan a continuación

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| auto | break | case | char | const |
| continue | default | do | double | else |
| enum | extern | float | for | goto |
| if | int | long | register | return |
| short | signed | sizeof | static | struct |
| switch | typedef | union | unsigned | void |
| volatile | while |
| Nota: Además de estas palabras se debe tener en cuenta no usar palabras de las funciones de las librerías usadas. | | | | |

En general es muy aconsejable ***elegir los nombres*** de las variables de forma que permitan conocer a simple vista qué tipo de variable o función representan.

Veamos entonces algunos ejemplos de nombres de variables válidos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| suma | resultado | calcuo | distancia\_1 | valor2 |
| T | F | valorFinal | registroInicial | elemento01\_02 |

El uso de las mayúsculas y minúsculas es de libre elección para el programador.

Se recomienda que los nombres de las variables no sean demasiado largos y que además sean fácilmente legibles.

Si es importante anotar que para el computador, en lenguaje C, las mayúsculas y minúsculas se diferencian plenamente. De esta forma, las variables

total\_factura, Total\_factura, Total\_Factura

Son tres variables claramente diferenciadas y válidas.

Ahora, veamos algunos ejemplos de nombres de variables no válidos junto con las razones por las que no lo son:

* **Total Operación** (no se puede usar espacios ni tampoco letras tildadas)
* **7ma** (no puede empezar con un dígito)
* **a + b** (no se deben emplear símbolos especiales como +)
* **Descuento.Porcentual** (no se puede emplear el punto **.** )

Para poder emplear una variable entera, es necesario hacer una declaración previa de su existencia. Para ello se usa la palabra reservada **int.**Por ejemplo, para poder usar una variable entera a*vance* dentro de nuestro programa, se requerirá indicarlo así:

int avance;

Esta instrucción tiene un significado muy preciso: le estamos indicando al computador, que dentro de nuestro programa usaremos una variable llamada avance,que será de tipo entero.

## Operadores

Los operadores nos permiten identificar la acción que se debe efectuar entre variables y o constantes.

Un operador es un símbolo que indica al compilador que lleve a cabo ciertas manipulaciones matemáticas o lógicas.

En C hay tres clases generales de operadores: aritméticos, relacionales, lógicos, y a nivel de bits. Además, C tiene operadores especiales para determinadas tareas.

### Aritméticos

Estos tipos de operadores se pueden utilizar en todos los tipos predefinidos en C.

Dentro de estos están:

Suma(+)

Resta(-)

Multiplicación (\*)

División (/)

Residuo de división (%) (resto de la división entera).

**NOTA:** EL operador % se aplica solamente a constantes, variables o expresiones de tipo entero.

Ejemplo: 23%4 es 3, puesto que el resto de dividir 23 por 4 es 3.

En C el operador **%** de división en modulo proporciona el resto de una división entera. Por ello, % no puede aplicarse a los tipos de coma flotante ( reales )

Una **expresión** es una combinación de variables y constantes relacionadas mediante distintos operadores. Un ejemplo de expresión en la que intervienen operadores aritméticos es la siguiente:

*Función:*

Y = (3 \* x) – (x \*x/2)

Estos operadores pueden aplicarse a casi todos los tipos de datos predefinidos y permitidos en C.

int x, y, z;

x = 30; y = 3;

z = 10 %3

x= (-b+raiz ((b\*b) - (4\*a\*c))) / (2\*a);

Donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

x = ( -b + sqrt ((b \* b) - (4 \* a \* c))) / (2 \* a)

Los operadores, las constantes y las variables son los constituyentes de las **expresiones**.

### Conversiones de tipos en las expresiones

Cuando en una expresión se mezclan constantes y variables de distintos tipos, se convierten a un tipo único. El compilador de C convierte todos los operandos al tipo del mayor operando. Esto se hace operación a operación, tal como describen las siguientes reglas de conversión de tipos:

1. cualquier **char** y **short int** es convertido a **int.** Cualquier **float** es convertido a **double**.
2. Para todos los pares de operandos, si uno de los operandos es **long double**, el otro operando se convierte a **long double**.

Si no, si un operando es **double**, entonces el otro se convierte a **double.**

Si no, si un operando es **long**, entonces el otro se convierte a **long**.

Si no, si un operando es **unsigned**, entonces el otro se convierte a **unsigned**.

Una vez aplicadas las reglas de conversión de tipos, cada par de operandos será del mismo tipo y el resultado de cada operación será del mismo tipo que el de los operandos.

### Espaciado y paréntesis

Para dar mayor legibilidad, se pueden añadir tabulaciones y espacios a discreción en una expresión. Por ejemplo, las dos expresiones siguientes son la misma.

a) X=10/y(127/x); b) X = 10 / y ( 127 / x ) ;

El uso de paréntesis redundantes o adicionales no produce errores ni disminuye la velocidad de ejecución de una expresión. Conviene utilizar paréntesis para hacer más claro el orden en que se producen las evaluaciones, tanto para uno mismo como para los que tengan que seguir después el programa.

Por ejemplo, si tuviera dos expresiones ¿cuál de las siguientes es más fácil de leer?

X = y/ 3-34 \* Temp & 127;

X = (y/3) – (34\*( Temp & 127) );

### Relaciónales

Estos operadores hacen referencia a la relación existente entre unos valores y otros. Estos operadores son:

|  |  |
| --- | --- |
| Operador | Acción |
| > | Mayor que |
| < | Menor que |
| >= | Mayor o igual que |
| <= | Menor o igual que |
| == | Igual |
| != | Diferente |

Nota “En el término operador relacional la palabra relacional se refiere a la relación entre unos valores y otros”

### Asignación

Los **operadores de asignación** atribuyen a una variable es decir, depositan en su zona de memoria correspondiente el resultado de una expresión (en realidad, una variable es un caso particular de una expresión).

El operador de asignación más utilizado es el **operador de igualdad** (=), que no debe ser confundido con la igualdad lógica (==). Su forma general es:

nombre\_variable = expresión;

Primero se evalúa **expresión** y el resultado se pone en **nombre \_ variable**.

Nota “el operador de asignación (=) representa una sustitución”, ya que toma el valor de la expresión y lo guarda en la variable nombre\_ variable.

### Abreviatura

Estos operadores fueron creados con el propósito de simplificación y disminución de errores en los programas.

El lenguaje C admite abreviaturas que simplifican la escritura de ciertos tipos de sentencias de asignación. Algunas de estas son:

+= -= \*= /= % ==

**Ejemplo:**

x = x +1 es igual a x += 1 es igual x++

y = y \* y +1 es igual a y \*= y + 1

a = 5;

Cuando el computador se encuentra con el operador igual, lo que hace es “calcular” el resultado total de todo lo que se encuentra a su derecha, toma este valor y lo almacena como contenido de la variable que se encuentra a la izquierda.

Usando otras palabras, el operador de asignación, **asigna** a la variable el resultado de la expresión. En el caso anterior a la variable *a* se le asigna el valor 5 (por que el resultado de calcular todo lo que se encuentra a la derecha del igual es simplemente el valor 5).

### Lógicos

Los operadores lógicos son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen las condiciones necesarias. Como **operadores lógicos** tenemos:

El operador Y (&&), el operador O (||) y el operador NO (!)

En inglés son los operadores **&&, || y !** Su forma general es la siguiente:

expresion1 **&&** expresion2,

expresion1 **||** expresión,

**!**expresión Cuando las de cada bit (Complemento a 1)

**&&**  And lógica bit a bit

**||** Or lógico bit a bit

**^**  Xor

**>>** Desplazamiento a la derecha

**<<**  Desplazamiento a la Izquierda

Los operadores **&&** y **||** se pueden combinar entre sí paréntesis. Por ejemplo:

(4==1) || (-10==-10) //el resultado es 1

(2==2) && (3==-1) //el resultado es 0

((2==2) && (3==3)) || (4==0) //el resultado es 1

((6==6) || (8==0)) && ((5==5) && (3==2)) //el resultado es 0

((250) && (330)) 250>0,330 > 0 //el resultado es 1

Es decir, se refieren a la forma en que los operadores relacionales pueden conectarse entre sí.

Estos son:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Predicados | | && (Y) | || (O) | ! (NO) |
| p | q | P&&q | p||q | !p |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

En el término operador lógico, la palabra lógico se refiere a las formas en que los operadores relacionales pueden conectarse entre sí, siguiendo las reglas de la lógica formal.

### Sentencias:

Si observamos cualquier expresión, la variable que está a la izquierda del signo **(=),** el operador de asignación, la expresión aritmética y el carácter **(;)** constituyen una sentencia.

Las sentencias son unidades completas, ejecutables en sí mismas.

Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

## Instrucciones de entradas y salida:

|  |  |
| --- | --- |
| Salida | Entrada |
| printf | scanf |

La librerías que se utilizan son:

#include <stdio.h> para funciones estándar de C ANSI

Formato de la instrucción p**rintf**

printf(“Mensaje formato”,nom\_variable);

**Cadenas y variables:**

Se escribe la cadena, se separa la variable con una coma, y después se escribe otra cadena. Se pone otra coma y se escribe la **cadena.**

Ejemplo:

printf(“la base %d”, b, ”por la altura”, a, ”es la superficie”, s)

**Tipos de formatos**

**Manejo de datos numéricos**

|  |  |
| --- | --- |
| Identificador | Descripción |
| %d | Para el manejo de números enteros |
| %i | Para el manejo de números enteros |
| %f | Para el manejo de números reales |
| %o | Para el manejo de números en Octal |
| %x | Para el manejo de números en Hexadecimal |
| %p | Para el manejo de apuntadores |

**Manejo de caracteres y cadenas**

|  |  |
| --- | --- |
| Identificador | Descripción |
| %s | Para el manejo de cadenas |
| %c | Para el manejo de caracteres |

**Manejo de presentación por pantalla**

|  |  |
| --- | --- |
| Identificador | Descripción |
| %v | Para Visualizar el texto o números verticalmente |
| %h | Para Visualizar el texto o números horizontalmente |

**Ejemplo:**

int h;

printf(“Digite un numero”);

scanf(“%d”,&h)

**Ejemplo 2**

Int A=1;

//Visualiza un mensaje y el contenido de la variable A:

Printf(“ el valor es %d ”,A);

**La instrucción de lectura**  **Scanf ()**

Toma los valores de forma interactiva desde el dispositivo de entrada estándar (teclado), introduciéndolos en las variables que forman parte de la instrucción. Los tipos de datos introducidos deben coincidir con los de las variables que los recogen.

Hay dos formas de usarlas:

**Formato scanf**

scanf(“tipo\_formato”, direccion Nom\_Variable);

scanf( “%d”, & h );

Donde la dirección me indica la posición donde queda almacenada la variable, en este caso la variable h

**Estructuras de programas**Cualquier programa puede ser realizado con una combinación de las siguientes estructuras de sentencias que definimos a continuación:- **Secuencial- Selectiva- Repetitiva**

**Estructura secuencial**

En esta estructura, cada acción se ejecuta en el orden preestablecido, por como son enumeradas a lo largo del programa. Se ejecutan de forma secuencial (una detrás de otra) y no puede verse alterado el orden de ejecución.

En primer lugar se ejecutaría la accion1, luego la acción2..., y así sucesivamente. Hasta llegar al fin del programa.

En el siguiente caso se evidencia una secuencia de acciones que consiste en la asignación de valores a variables y posteriormente incremento entre las mismas. Veamos:

Diagrama de flujo Pseudocódigo

Acción 1 X = 5; //Asignamos a x el valor 5

Acción 2 Y = 7; //Lo mismo con y el valor 7

Acción 3 Z = x + y; //Asignamos a z el valor x + y

Acción 4 X = x +3; //Incrementamos x en 3

Acción 1

Acción 2

Acción 3

## Estructuras de control selectivas

En ciertos programas, durante su ejecución, se pueden necesitar unas variaciones de acuerdo con el cumplimiento o no de algunas condiciones. Mediante las estructuras selectivas podemos tomar decisiones, en las cuales se **evalúa una condición** y en función del resultado **se ejecuta** o no una acción o acciones.

Las distintas estructuras de las que disponemos son:

if (si, entonces) y switch (según sea haga )

La mayoría de las sentencias de control de programa de cualquier lenguaje de computadora, incluyendo a C, se basan en una prueba condicional que determina la acción que se ha de llevar a cabo.

De acuerdo con la sintaxis de C, una sentencia puede consistir en una de las siguientes: una única sentencia, un bloque de sentencias o nada (en el caso de sentencias vacías). El termino sentencia se usa para cualquiera de las tres posibilidades.

### Selección Incompleta (simple)

La estructura alternativa simple **si-entonces**,ejecuta una determinada acción cuando se cumple una determinada condición. La selección sí-entonces, evalúa la condición, y si la condición es verdadera, entonces ejecuta la acción (o acciones), o si la condición es falsa, entonces no hace nada y el programa continúa con la siguiente instrucción.La representación gráfica de la estructura condicional simple se muestra en la siguiente figura:

**Ejemplo 1:**

Diagrama de flujo Pseudo código

Falsa

Condición

Acciones

Acción

Fin

Verdadera

Nombre \_ programa

Inicio

si (condición)

entonces

<acción 1>

<acción 2>

fin

**Lenguaje C**

|  |  |
| --- | --- |
| 1 | #include <stdio.h> //librería |
| 2 | void main( ) //inició |
| 3 | { |
| 4 | if (expresión ) //evalúa la condición |
| 5 | sentencia1; //accion1 |
| 6 | sentencia2; //accion2 |
| 7 | } |

**Ejemplo 2:**El siguiente ejemplo ilustra una secuencia de condiciones que serán evaluadas a lo largo del programa, y dependiendo de su veracidad serán ejecutadas las respectivas acciones. Veamos.

Programa que indica el estado de un número. (positivo, negativo o igual a cero)

Pseudocódigo

Inicio{

leer (numero);

**Si** (numero = 0) **entonces**

escribe (“el numero digitado es igual a cero”);

**Si** (numero<0) **entonces**

escribe (“el numero digitado es negativo”); **Si** (numero>0) **entonces**

escribe (“el numero digitado es positivo”);}

Por ejemplo: Aquí contemplamos sólo una acción en el caso en que se cumpla la condición: (solo hallamos la raíz)

Pseudocódigo

inicio{ leer (x); // Obtenemos x desde el teclado

**si** (x>=0) **entonces**

y = raíz(x);// si x es positiva hallamos la raíz

x = x \* x; // hacemos x2

escribe (x); // Escribimos el valor de x

}

### Estructura de selección completa

Se realizará cuando sí se cumple la condición y cual se hará cuando no se cumpla.

Su pseudocódigo es el siguiente:

V

V

F

F

F

Inicio

Digite el num

Num : ENTERO

Si num=0

Si

Num>0

Imprime: num es igual cero

Imprime: num es positivo

Imprime: num es negativo

Fin

**Pseudocódigo**

(Con acción compuesta)

**si** (condición) **entonces**

{

<acción 1>;

}

**sino**

{

**si** (condición) **entonces**

{

<acción 2>;

}

**sino**

{

<acción 3>;

}

}

**Ejemplo 2.**

Inicio

{

leer (edad);

**si** (edad>=18) **entonces**

{

escribe (“usted es mayor de edad”);

escribe (edad);

}

sino

{

escribe (“usted es menor de edad”)

escribe (edad);

}

}

### Switch (según sea)

C incorpora una sentencia de decisión de ramificación múltiple denominada **switch.** Esta sentencia compara sucesivamente una variable con una lista de constantes enteras o de caracteres. Cuando se encuentra una opción válida, se ejecuta una sentencia o bloque de sentencias.

La forma general de la sentencia **switch** es:

Caso 1

Caso 6

Caso 2

2

Caso 3

Caso 4

Caso 5

Condición

dición

Acción 1

Acción 2

Acción 3

Acción 4

Acción 5

Acción 6

switch (variable)

{

case constante1: secuencia de sentencias

break;

case constante2: secuencia de sentencias

break;

case constante3: secuencia de sentencias

break;

default: secuencia de sentencias;

}

## Estructuras de control repetitivas

Algunas veces tendremos que realizar tareas que, dentro de un programa se deben repetir un **número determinado o indeterminado** de veces. Este es un tipo muy importante de estructura, ya que, nos permite ahorrar muchas líneas de código, pero para su uso hay que tener en cuenta ciertos aspectos que involucran su funcionalidad:

1. El conjunto de tareas debe ser finito, es decir las acciones que hacen parte del ciclo deben alguna vez llegar a un límite de operación.
2. La cantidad de veces que se repita dicho conjunto de instrucciones también debe ser finito, esto indica, que el **incrementador/decrementador** para dicho ciclo sumará o restará cierta cantidad de veces dependiendo de su condición, que a su vez puede ser explícita o implícita:
   * Una condición es **explicita** cuando depende únicamente de la misma ejecución del programa.
   * Una condición es **implícita** cuando depende únicamente de la voluntad del usuario y por lo tanto la cantidad de iteraciones o repeticiones del ciclo podrá llegar a ser diferente cada vez, dependiendo del gusto de cada usuario.
3. Igual que en las demás estructuras, los ciclos también deben estar demarcados por un inicio y un fin.

Por ejemplo: supongamos que se desea sumar una lista de 35 números escritos desde teclado. Por ejemplo, calificaciones de los alumnos de una clase.

Hasta ahora lo que haríamos es leer los números en una variable y añadir sus valores a una variable SUMA, que contenga las sucesivas sumas parciales. La variable SUMA se hace igual a cero y, a continuación se incrementa en el valor del número cada vez que uno de ellos se lea.

En pseudo código:

**Inicio**

entero número, suma;

suma = 0;

Leer (numero);

suma = suma + numero;

Leer (numero);

suma = suma + numero;

**Fin**

Y así sucesivamente para cada número de la lista (los 35 alumnos). En otras palabras, el algoritmo repite muchas veces las acciones.

Cuando se utiliza un bucle para sumar una lista de números, se necesita saber cuántos números se han de sumar. Para ello, necesitaremos conocer algún medio para finalizar el bucle. En el ejemplo anterior se repetiría **desde** 1 **hasta** 35 la acción de pedir la nota.

### Estructura mientras

Es aquella en la que la acción o grupo de acciones se repetirá **mientras** se cumpla la condición y en el momento en que no se cumpla se saldrá del bucle:

Una variable contendrá los valores del número de veces que queramos realizar una secuencia de código, por lo tanto debemos **conocer obligatoriamente** el número de veces (iteraciones) que se realizaran.

En pseudo código:

i = índice

**desde** i = valor\_inicial **hasta** i = valor\_final **incremento** valor **<acción>;**

**Diagrama de flujo del Mientras**

Acción

Condición

Verdadero

**Ejemplo 1**: el siguiente programa determina la cantidad de números impares comprendidos entre 1 y 100. Veamos:

Inicio

i = 1; h = 100;

**desde** (i) **hasta** (h)

inicio

i = i + 2;

escribe (“cantidad de números impares es:”);

escribe (i);

fin desde

Fin

El formato general del bucle **for** de C se encuentra de una forma o de otra en todos los lenguajes de programación de procedimiento. En C, sin embargo, proporciona una potencia y flexibilidad sorprendentes. La forma general de la sentencia **for** es:

For (inicialización; condición; incremento)

Sentencia;

### Estructura haga mientras

Con el ciclo **haga mientras,** el cuerpo del programa **se ejecuta siempre al menos una vez***.* Cuando una instrucción **haga mientras** se ejecuta, la primera cosa que sucede es la ejecución acciones o instrucciones y, a continuación se evalúa la condición. Si se evalúa como falsa, el cuerpo del programa se repite. Si es *verdadera,* el ciclo termina y el programa sigue en la siguiente instrucción**.**

Verdadero

Falsa

Condición

Acción

Inicio

Fin

**Ejemplo 1**: este es el mismo ejemplo 2 de la estructura mientras (que lee un número entero y arroja el valor de su sumatoria). Veamos:

Inicio

a = 0; b = 0;

leer (numero);

**haga**

a = a + 1;

c=a+b;

b=c;

**mientras** (a<= numero)

escribe (“la sumatoria es: ”);

escribe (c);

fin

**Ejemplo 2**: un programa que lee un número entero y determina cuantos dígitos lo componen. Veamos:

Inicio

nd = 0; leer(a);

haga

a = a/10;

nd = nd + 1;

**mientras** (a>=1)

escribe (“el numero tiene: ”);

escribe (”%d dígitos”, nd);

Fin

## Ejercicios en C

1. Realizar un programa que me determine si un número es par o no

|  |  |
| --- | --- |
| 1 | #include "stdio.h" |
| 2 | #include "conio.h" |
| 3 | int i,numero; |
| 4 | void main (void) |
| 5 | { |
| 6 | for(i=1;i<=10;i++) |
| 7 | { |
| 8 | printf("\n\n digite el numero %d:",i); |
| 9 | scanf("%d",&numero); |
| 10 | if((numero%2)==0) |
| 11 | printf(" el número es par:%d",numero); |
| 12 | else |
| 13 | if((numero%2)!=0) |
| 14 | printf(" el número es inpar :%d ",numero); |
| 15 | } |
| 16 | } |

1. Realizar un programa que me realice la suma de pares e impares de 10 números ingresados por teclado

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <conio.h> |
| 3 | int i, suma, suma1, num; |
| 4 | void main(void) |
| 5 | { |
| 6 | for(i=1;i<=10;i++) |
| 7 | { |
| 8 | printf("digite el numero: "); |
| 9 | scanf("%d",&num); |
| 10 | if((num%2)==0) |
| 11 | suma=suma+num; |
| 12 | else |
| 13 | if((num%2)!=0) |
| 14 | suma1=suma1+num; |
| 15 | } |
| 16 | printf("\n numeros pares %d \t",suma); |
| 17 | printf("\n total inpares %d\t",suma1); |
| 18 | getch(); |
| 19 | } |

1. Realizar un programa que lea un número y determine si es capicúa o no

En este capítulo se dará una introducción a los arreglos más básicos: los vectores y las matrices.

Se verán primero, los vectores, introducción de valores y manejo de los mismos. Después se pasará a ver los métodos de ordenamiento para los vectores, métodos que ayudan a organizar un vector de modo que sus valores vayan de menor a mayor.

Después se finalizará con una lección acerca de las matrices, el ingreso de datos y la manipulación de estos datos dentro de la matriz.

# CAPÍTULO III:

# Arreglos y métodos de ordenamiento

# Arreglos y métodos de ordenamiento

## Estructuras de agrupamiento de variables

### Arreglo:

Se puede definir como una variable que está formada por varios elementos, también podemos decir que un arreglo es una colección finita, homogénea y ordenada de elementos. Esta debe ser:

* **Finita:** Porque tiene un final.
* **Homogénea:** Porque todos los elementos del arreglo tienen el mismo tipo de dato:

Entero, reales, cadena de caracteres, etc.

* **Ordenada:** Porque se puede determinar cuál es el orden específico de cada elemento del arreglo a través del índice

### Arreglo (vectores):

En C, un arreglo es una lista de variables del mismo tipo, que se referencia por un nombre común. A una variable individual del arreglo se le llama *elemento del arreglo.*

Los arreglos constituyen un modo adecuado de trabajar grupos de datos relacionados.

Para declarar un arreglo unidimensional, se utiliza la forma general:

**Tipo de dato** nombre\_de\_variable [tamaño]

Por ejemplo, para declarar un arreglo entero de 7 elementos llamado array:

Int array[8];

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Índice:* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* |
| Contenido: | **7** | **3** | **6** | **9** | **8** | **4** | **0** | **2** |

Un vector está compuesto por 3 componentes:

* Por el nombre.
* Por un Subíndice.
* Por contenido (los elementos)

**Representaicon Grafica de un arreglo**

Nombre arreglo

0 1 2 3 4 5 6 7

7 3 6 9 8 4 10 2

Subíndice

Elementos

**Declaración**

Tipo de dato nombre\_variable [cantidad de elementos];

Por ejemplo:

int var[10]; //Para declarar enteros

char nombre[50]; //Para declarar cadenas

float numeros[200]; //Para declarar decimales

long double cantidades[25]; //Para declarar cantidades grandes

Si tomamos el primer caso, estamos declarando un vector de 10 variables enteras, cada una de ellas quedará individualizada por el subíndice que sigue al nombre del mismo es decir:

var1 [0], var1 [1], etc., hasta var1 [9].

Nótese que la cantidad de elementos es 10, pero su numeración va de **0 a 9**, y no de 1 a 10. En cada uno de estos elementos podemos almacenar una información del mismo tipo. En resumen un vector de N elementos tiene subíndices válidos entre **0 y N - 1**. Cualquier otro número usado como subíndice, traerá datos de otras zonas de memoria, cuyo contenido será impredecible.

Se puede referenciar a cada elemento, en forma individual, tal como se ha hecho con las variables anteriormente, por ejemplo:

var1[5]=40;  
contador = var1[3] + 7;

if (var1 [0] >= 37)

También es posible utilizar como subíndice expresiones aritméticas, valores enteros retornados por funciones, etc. Así podríamos escribir:

printf(" %d " , var1[ ++i] );  
var1 [8] = var1[ i + j ];

La inicialización de un vector local, puede realizarse en su declaración, dando una lista de valores iniciales:

int numero[8] = { 4 , 7 , 0 , 0 , 0 , 9 , 8 , 7 };

Obsérvese que la lista está delimitada por llaves. Otra posibilidad, sólo válida cuando se inicializan todos los elementos del vector, es escribir:

int numero[ ] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

Donde se obvia la declaración de la cantidad de elementos, ya que está implícita en la lista de valores constantes.

También se puede inicializar parcialmente un vector, por ejemplo:

int numero[10] = {1, 1, 1} ;

En éste caso los tres primeros elementos del mismo valdrán 1 (**uno**), y los restantes **cero** en el caso que la declaración sea global, o cualquier valor impredecible en el caso de que sea local.

**Ejemplo.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| int M[5], I;  for ( I=0; I<5; I++) {  M[ I ]=I;  } | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | *0* | *1* | *2* | *3* | *4* | **Subíndice** | | **M** | **0** | **1** | **2** | **3** | **4** |  | |  | **Elementos** | | | | | |

El ejemplo anterior va llenado el arreglo uno por uno a medida que ciclo se va ejecutando.

M[0] =0;

Como imprimir un vector por pantalla:

int M[5], I;

for ( I=0; I< 5; I++)

{

printf (“los vectores son %d ”, M[I]);

}

**Ejemplo 1.**

Leer un vector de 10 elementos y almacenarlos en un vector. Determinar la suma sólo de los numero pares.

|  |  |
| --- | --- |
| 1 | #include "stdio.h" //Declaración de la librería del programa |
| 2 | #include "conio.h" |
| 3 | int v[10]; // Declaración del vector y las variables |
| 4 | int i, suma; |
| 5 | void main (void) // Inicio del programa |
| 6 | { |
| 7 | for(i=1; i<=10; i++) // Inicio de ciclo desde 1 hasta 10 |
| 8 | { |
| 9 | printf ("\n digite el numero %d:",i); |
| 10 | //Entrada de datos por teclado |
| 11 | scanf ("%d", &v[i]); //Lee y almacena en el vector |
| 12 | } |
| 13 | for(i=1;i<=10;i++) //Inicia el ciclo para recorrer el vector |
| 14 | { |
| 15 | //Evalúa si los elementos del vector son pares o no: |
| 16 | if ( (v[i] %2) == 0) |
| 17 | //Si se cumple la condición se suma los datos: |
| 18 | suma = suma + v[i]; |
| 19 | else //Si no cumple la condición escriba no es par |
| 20 | { |
| 21 | printf (" no es numero par "); |
| 22 | //imprime el resultado de la suma: |
| 23 | printf ("\n la suma de numeros pares es:%d ",suma); |
| 24 | } |
| 25 | } |
| 26 | getch(); //Visualiza en pantalla. |
| 27 | } //final del programa. |

**Ejercicio**

1. Realizar un programa que lea la temperatura del medio día de un mes y después informar la temperatura media mensual, así como los días más frío y más cálido.
2. Llenar un vector de 10 elementos y determinar cuántos números son múltiplos de 3.
3. Llenar un vector de 10 elementos y determinar cuántos números negativos, positivos y ceros tiene el vector.
4. Llenar un vector de 10 elementos e invertir el vector.
5. Llenar un vector de 10 elementos, digitar un número y determinar cuántas veces se repite el número.
6. Borrar un elemento del vector.
7. Llenar un vector de 10 elementos y determinar cuántos números están en los siguientes rangos.

num > 0

num > 3 num < 6

num > 8

1. Calcular el promedio de 50 valores almacenados. Determinar además cuantos son mayores que el promedio, imprimir el promedio, el número de datos mayores que el promedio y una lista de valores mayores que el promedio.
2. Almacenar 500 números en un vector, elevar al cuadrado cada valor y almacenar el resultado en otro vector.
3. Imprimir el vector original y el vector resultante.
4. Almacenar 30 números en un vector, imprimir cuántos son ceros, cuántos son negativos y cuántos positivos. Imprimir además la suma de los negativos y la suma de los positivos.
5. El dueño de una cadena de tiendas de artículos deportivos desea controlar sus ventas por medio de una computadora. Los datos de entrada son:

* El número de la tienda (1 a 50)
* Un número que indica el deporte del artículo (1 a 20)
* El costo del artículo.

1. Hacer un programa que escriba al final del día lo siguiente

* Las ventas totales en el día para cada tienda
* Las ventas totales para cada uno de los deportes.
* Las ventas totales de todas las tiendas.

1. Se tienen 50 alumnos los cuales se tiene la información sobre las calificaciones de la materia de LENGUAJES ALGORITMICOS.

Diseñe un diagrama que imprima:

* Cantidad de alumnos que aprobaron la materia.
* Cantidad de alumnos que tienen derecho a nivelación.
* El (o los) numero (s) de control de lo(s) alumno(s) que haya (n) obtenido la máxima calificación final.

## Algoritmos de Ordenamiento

Existen dos tipos de métodos de ordenación:

* Ordenación interna
* Ordenación externa

### Ordenación interna

Los métodos de ordenación interna se aplican principalmente a arreglos unidimensionales (vectores), con un número de elementos pequeños. En este caso aplica para datos de menos de 100 elementos.

Los principales algoritmos de ordenación interna son:

#### Método de Selección:

Este método consiste en buscar el elemento más pequeño del arreglo y ponerlo en primera posición. Luego, entre los restantes, se busca el elemento más pequeño y se coloca en segundo lugar y así sucesivamente hasta colocar el último elemento.

Por ejemplo, si tenemos el arreglo.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 21 | 4 | 9 | 10 | 35 |

Los pasos a seguir son:

**Paso 1**

Se busca el elemento más pequeño y la posición en que se encuentra.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 40 | 21 | 4 | 9 | 10 | 35 |  |

En este caso es el número 4 y se encuentra en la posición 2 del vector.

Se intercambia el 4 por el 40 que se encuentra en la primera posición.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 40 | 21 | 4 | 9 | 10 | 35 |  |
|  | | | | | |  |

El vector queda así:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 21 | 40 | 9 | 10 | 35 |  |

**Paso 2**

Se busca el segundo elemento más pequeño y la posición en que se encuentra, en este caso es el número 9.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 21 | 40 | 9 | 10 | 35 |  |

Se incrementa una posición el índice en este caso pasa de posición cero al uno cambia el 9 por el 21.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 9 | 40 | 21 | 10 | 35 |  |
|  | | | | | |  |

**Paso 3**

Se busca el tercer elemento más pequeño y la posición en que se encuentra, en este caso es el número 10.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 9 | 40 | 21 | 10 | 35 |  |

Se incrementa una posición el índice, en este caso pasa de posición uno al dos cambia el 10 por el 40.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 9 | 10 | 21 | 40 | 35 |  |
|  | | | | | |  |

**Paso 4**

Se busca el cuarto elemento más pequeño y la posición en que se encuentra, en este caso es el número 21.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 9 | 10 | 21 | 40 | 35 |  |

Se incrementa una posición el índice, en este caso pasa de posición dos al tres. Se compara si el número que se encuentra en la tercera posición es menor que el número menor buscado, vemos que es igual, por lo que no lo movemos de posición y pasamos de nuevo a buscar el quinto elemento mas pequeño, vemos que es 35 y su posición es 5, ahora incrementamos el índice en uno y pasa a ser 4, entonces podemos intercambiarlos:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 9 | 10 | 21 | 40 | 35 |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **INDICE** |
|  | | | | | | |
| 4 | 9 | 10 | 21 | 35 | 40 |  |

Un arreglo tiene N elementos, el número de comprobaciones que hay que hacer es de orden de:

**N\*(N-1)/2,** luego el tiempo de ejecución está en **O (n2)**

**Algoritmo en c (Método de selección):**

|  |  |
| --- | --- |
| 1 | int arreglo[N]; |
| 2 | int i, j, menor, aux; |
| 3 | for(i =0; i <N-1; i++) |
| 4 | { |
| 5 | for( j =i+1, menor=i; j<N; j++) |
| 6 | if (arreglo[j] < arreglo[menor]) |
| 7 | menor = j; // el menor pasa a ser el elemento j. |
| 8 | aux = arreglo[i]; // Se intercambian los elementos |
| 9 | arreglo[i] = arreglo[menor]; |
| 10 | arreglo[menor] = aux; |
| 11 | } |

#### Método de Burbuja:

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados.

Con el arreglo anterior:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 21 | 4 | 9 | 10 | 35 |

**Primera pasada:**

* Se cambia el 21 por el 40.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 40 | 4 | 9 | 10 | 35 |

* Se cambia el 40 por el 4.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 4 | 40 | 9 | 10 | 35 |

* Se cambia el 9 por el 40.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 4 | 9 | 40 | 10 | 35 |

* Se cambia el 40 por el 10.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 4 | 9 | 10 | 40 | 35 |

* Se cambia el 35 por el 40.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 4 | 9 | 10 | 35 | 40 |

**Segunda pasada:**

Vector inicial.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 4 | 9 | 10 | 35 | 40 |

* Se cambia el 21 por 4.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 21 | 9 | 10 | 35 | 40 |

* Se cambia el 9 por el 21.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 21 | 10 | 35 | 40 |

* Se cambia el 21 por el 10

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 35 | 40 |

* Se cambia 21 por 35
* No intercambia 21 por el 35, ya que el 35 es mayor que el 21, solo intercambia en sentido contrario.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 35 | 40 |

* No intercambia 35 por el 40, ya que el 40 es mayor que el 35, solo intercambia en sentido contrario.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 35 | 21 | 40 |

**Tercer pasada:**

* No intercambia 9 por 4 pero va incrementando el índice

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 35 | 21 | 40 |

* No intercambia 10 por 9

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 35 | 21 | 40 |

* No intercambia 35 por 10

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 35 | 21 | 40 |

* Intercambia 21 por 35

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 35 | 40 |

Al terminar, los elementos están ordenados, pero para comprobarlo habría que acabar esta tercera comprobación y hacer una cuarta.

Si el arreglo tiene N elementos, para estar seguro de que el arreglo está ordenado, hay que hacer N-1 pasadas, por lo que habría que hacer **(N-1)\*(N-i-1)** comparaciones, para cada **i desde 1 hasta N-1.** El número de comparaciones es, por tanto, **N(N-1)/2,** lo que nos deja un tiempo de ejecución, al igual que en la selección, en **O (n2).**

Algoritmo en c:

|  |  |
| --- | --- |
| 1 | int arreglo[N]; |
| 2 | int i, j, aux; |
| 3 | for (i=0; i<N-1; i++) |
| 4 | for(j=0; j<N-i-1; j++) //Mirar los N-i-1 pares |
| 5 | if(arreglo[j+1] < arreglo[j]) |
| 6 | { |
| 7 | aux=arreglo[j+1]; |
| 8 | arreglo[j+1] = arreglo[j]; |
| 9 | arreglo[j] = aux; |
| 10 | } |

#### Método de Inserción Directa:

En este método lo que se hace es tener una sublista ordenada de elementos del arreglo e ir insertando el resto en el lugar adecuado, para que la sublista no pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada.

**Ejemplo:**

Se tiene.

**0 1 2 3 4 5 Indice**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 21 | 4 | 9 | 10 | 35 |

La primera sublista ordenada es 40.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 21 | 4 | 9 | 10 | 35 |

Insertamos el 21

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 21 | 4 | 9 | 10 | 35 |

{40,21,4,9,10,35} <-- {40,40,4,9,10,35} <- aux = 21;

Ahora la sublista ordenada es {21,40}.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 40 | 4 | 9 | 10 | 35 |

<-- Insertamos el 4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 40 | 4 | 9 | 10 | 35 |

{ 21,40,40,9,10,35 } <-- aux = 4;  
{ 21,21,40,9,10,35 } <-- aux = 4;

Ahora la sublista ordenada es {4, 21, 40}.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 21 | 40 | 9 | 10 | 35 |

<-- Insertamos el 9:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 21 | 40 | 9 | 10 | 35 |

{4,21,40,40,10,35} <-- aux = 9;  
{4,21,21,40,10,35} <-- aux = 9;

Ahora la sublista ordenada es {4, 9, 21, 40}.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 21 | 40 | 10 | 35 |

<--Insertamos el 10:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 21 | 40 | 10 | 35 |

{4,9,21,40,40,35} <-- aux = 10;  
{4,9,21,21,40,35} <-- aux = 10;

Ahora la sublista ordenada es {4, 9, 10, 21, 40}

Y por último insertamos el 35

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 40 | 35 |

{4, 9, 10, 21, 40, 40} <-- aux = 35;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 35 | 40 |

El arreglo está ordenado

En el peor de los casos, el número de comparaciones que hay que realizar es de **N\*(N+1)/2-1**, lo que nos deja un tiempo de ejecución en **O (n2).** En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es N-2. Todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(n).

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Se observa que cuanto más ordenada esté inicialmente más se acerca a **O(n)** y cuanto más desordenada, más se acerca a **O(n2).**

El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

**Algoritmo en c:**

|  |  |
| --- | --- |
| 1 | int arreglo[N]; |
| 2 | int i, j, aux; |
| 3 | for(i=1; i<N; i++) |
| 4 | { |
| 5 | aux=arreglo[i]; |
| 6 | for(j=i-1;j>=0;j--) |
| 7 | { |
| 8 | if(aux>arreglo[j]) |
| 9 | { |
| 10 | arreglo[j+1]=aux; |
| 11 | break; |
| 12 | } |
| 13 | else |
| 14 | arreglo[j+1]=arreglo[j]; |
| 15 | } |
| 16 | if(j==-1) |
| 17 | arreglo[0]=aux; |
| 18 | } |

#### Método de Inserción binaria (SHELL)

Es el mismo método que la inserción directa, excepto que la búsqueda del orden de un elemento en la sublista ordenada se realiza mediante una búsqueda binaria, lo que en principio supone un ahorro de tiempo. No obstante, dado que para la inserción sigue siendo necesario un desplazamiento de los elementos, el ahorro, en la mayoría de los casos, no se produce, si bien hay compiladores que realizan optimizaciones que lo hacen ligeramente más rápido.

#### Método de Inserción Directa mejorado (Shell):

Es una mejora del método de inserción directa, utilizado cuando el arreglo tiene un gran número de elementos. En este método no se compara a cada elemento con el de su izquierda, como en el de inserción, sino con el que está a un cierto número de lugares (**llamado salto**) a su izquierda. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo, lo pasos para ordenar el siguiente arreglo son:

**0 1 2 3 4 5 6 7 8 índice**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 4 | 40 | 21 | 8 | 25 | 45 | 55 |

Mediante el método de Shell serían: **n/2**= donde n es el tamaño del vector

Salto = 4

No hay cambio por que 9 es menor que 21

**Primera pasada:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 4 | 40 | 21 | 8 | 25 | 45 | 55 |

Se realiza el intercambio 10 por el 8 por que 10 es mayor que 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 8 | 4 | 40 | 21 | 10 | 25 | 45 | 35 |

🡨 No hay cambio

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 8 | 4 | 40 | 21 | 10 | 25 | 45 | 55 |

🡨 No hay cambio por que 40 es menor que 45

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 8 | 4 | 40 | 21 | 10 | 25 | 45 | 55 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 8 | 4 | 40 | 21 | 10 | 25 | 45 | 55 |

🡨 No hay cambio por que 21 es menor que 55

Volvemos a dividir el salto el salto era cuatro (4) Salto = salto/2;

**Segunda pasada:**

**🡨** Realizo el intercambio 9 por el 4

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 8 | 9 | 40 | 21 | 10 | 25 | 45 | 55 |

🡨 No hay cambio

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 40 | 21 | 10 | 25 | 45 | 55 |

🡨 No hay cambio

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 40 | 21 | 10 | 25 | 45 | 55 |

**🡨** Realizo el intercambio 40 por el 10

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 10 | 21 | 40 | 25 | 45 | 55 |

**🡨** No hay cambio

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 10 | 21 | 40 | 25 | 45 | 55 |

🡨 No hay cambio

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 10 | 21 | 40 | 25 | 45 | 55 |

🡨 No hay cambio

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 10 | 4 | 40 | 21 | 45 | 55 |

Volvemos a dividir el salto el salto era cuatro (2) Salto = salto/2;

Salto =1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 10 | 21 | 40 | 25 | 45 | 55 |

🡨 No hay cambio

No hay cambio desde la posición donde está el número a la posición donde está el número 40 y 25 y realizo el intercambio

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 10 | 21 | 25 | 40 | 45 | 55 |

**🡨** Realizo el intercambio 40 por el 25

El arreglo quedo completamente ordenado

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 9 | 10 | 21 | 25 | 40 | 45 | 55 |

**Algoritmo en c:**

|  |  |
| --- | --- |
| 1 | int arreglo[N]; |
| 2 | int salto, cambios, aux, i; |
| 3 | for(salto=N/2; salto!=0; salto/=2) |
| 4 | for(cambios = 1; cambios != 0;) |
| 5 | { |
| 6 | cambios = 0; |
| 7 | for(i=salto; i<N; i++) |
| 8 | if(arreglo[i-salto]>arreglo[i]) |
| 9 | { |
| 10 | aux=arreglo[i]; |
| 11 | arreglo[i]=arreglo[i-salto]; |
| 12 | arreglo[i-salto]=aux; |
| 13 | cambios++; |
| 14 | } |
| 15 | } |

#### Método Ordenación rápida (QuickSort):

Este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo el arreglo en arreglos más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del arreglo como pivote, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha.

A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el arreglo.

Por ejemplo, para dividir el arreglo

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 40 | 4 | 9 | 10 | 35 |

Los pasos serian.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 40 | 4 | 9 | 10 | 35 |

Se toma como pivote el **21**. La búsqueda de izquierda a derecha encuentra el valor 40, mayor que pivote, y la búsqueda de derecha a izquierda encuentra el valor 10, menor que el pivote.

Se intercambian:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 40 | 4 | 9 | 10 | 35 |

{21, 10, 4, 9, 40, 35} <-- Si seguimos la búsqueda, la primera encuentra el valor 40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para terminar la división, se coloca el **pivote** en su lugar (en el número encontrado por la segunda búsqueda, el **9**, quedando:

{9, 10, 4, 21, 40, 35} <-- Ahora tenemos dividido el arreglo en dos arreglos más pequeños: el {9, 10, 4} y el {40, 35}, y se repetiría el mismo proceso.

**Algoritmo en c:**

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | void ordenar(int \*arreglo, int desde, int hasta) |
| 3 | { |
| 4 | int i, d, aux; |
| 5 | if(desde >= hasta) |
| 6 | return; |
| 7 | for(i=desde+1, d=hasta;;) |
| 8 | { |
| 9 | for(; i<=hasta&&arreglo[i] <= arreglo[desde]; i++); |
| 10 | for(; d>=0 && arreglo[d] >= arreglo[desde]; d--); |
| 11 | if(i<d) |
| 12 | { |
| 13 | aux=arreglo[i]; |
| 14 | arreglo[i]=arreglo[d]; |
| 15 | arreglo[d]=aux; |
| 16 | } |
| 17 | else |
| 18 | break; |
| 19 | } |
| 20 | if(d ==desde-1) |
| 21 | d = desde; |
| 22 | aux=arreglo[d]; |
| 23 | arreglo[d]=arreglo[desde]; |
| 24 | arreglo[desde]=aux; |
| 25 | ordenar(arreglo, desde, d-1); |
| 26 | ordenar(arreglo, d+1, hasta); |
| 27 | } |
| 28 |  |
| 29 | void main() { |
| 30 | ordenar(arreglo, 0, N-1); |
| 31 | } |

En C hay una función que realiza esta ordenación sin tener que implementarla, llamada qsort (incluida en stdlib.h):

qsort(nombre\_arreglo, numero, tamaño, función);

Donde nombre\_arreglo es el nombre del arreglo a ordenar, número es el número de elementos del arreglo, tamaño indica el tamaño en bytes de cada elemento y función es una función que hay que implementar, esta recibe dos elementos y devuelve 0 si son iguales, algo menor que 0 si el primero es menor que el segundo, y algo mayor que 0 si el segundo es menor que el primero. Por ejemplo, el programa de antes sería:

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 |  |
| 4 | int funcion(const void \*a, const void \*b) |
| 5 | { |
| 6 | if(\*(int \*)a<\*(int \*)b) |
| 7 | return(-1); |
| 8 | else if(\*(int \*)a>\*(int \*)b) |
| 9 | return(1); |
| 10 | else |
| 11 | return(0); |
| 12 | } |
| 13 |  |
| 14 | void main() |
| 15 | { |
| 16 | qsort(arreglo, N, sizeof(arreglo[0]), funcion); |
| 17 | } |

Claramente, es mucho más cómodo usar qsort que implementar toda la función, pero hay que tener mucho cuidado con el manejo de los punteros en la función, sobre todo si se está trabajando con estructuras.

## Cadenas

Se define como un arreglo de caracteres con un carácter de terminación nulo (“\0”).

En C el carácter nulo es el cero 0, el hecho que de que la cadena deba terminar en un carácter nulo significa que, para declarar un vector de caracteres es necesario que el tamaño sea más largo para poder ingresar la cadena de caracteres.

**Declaración**

Tipo de las variables nombre[cantidad de elementos];

**Por ejemplo:**

char str[50] ;

Gráficamente se puede observar así:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Índice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  | G | U | S | T | A | V | O | \0 |

La definición anterior se puede declarar así:

char nombre[] = “GUSTAVO”;

Se está definiendo una cadena de 8 elementos y el compilador, automáticamente inserta el carácter Nulo “\0”, indicando el final del arreglo.

Para trabajar con cadenas utilizamos la **función gets ( )**, esta función utiliza la librería STDIO.H. Lee caracteres hasta que pulse la tecla enter, el retorno de carro no se almacena, pero se reemplaza por un carácter nulo.

**Ejemplo:**

Printf (“introduzca el nombre no mayor de 50 caracteres”):

gets( str);

printf (“ %s ”,str);

La biblioteca estándar de C suministra muchas funciones relacionadas con cadenas, pero las funciones más importantes son

strcpy (hacia, desde)

“Esta funciones requiere un archivo de cabecera #include <string.h>

strcat (hacía, desde)

strlen (nom\_cadena)

strcmp (cadena1, cadena2)

### La función strcpy (cad1, cd2);

Esta función copia el contenido desde cadena2 hacia la cadena1, el contenido de cadena2 no cambia, esta función no realiza comprobación de limite, por lo tanto debemos asegúranos que la cadena1 sea lo suficientemente grande para contenerlo.

#### Ejemplo:

char cadena [20];

strcpy (cadena, “hola”) //Copia “hola” en una cadena

printf ( “%s” , cadena ) //Imprime la cadena

### La función strcat (cad1, cad2) ;

La función **strcat (cad1, cad2)** concatena una copia de *cad2* en *cad1* añadiéndola al final de *cad1* y dejando un carácter nulo.

El carácter nulo de terminación, que originalmente tenia *cad1* es sustituido por el primer carácter de *cad2.* La cadena *cad2* no se toca en esta operación. La función **strcat ()** devuelve *cad1.*

char cadena [20];

// Copia la palabra hola en una cadena:

strcpy (cadena, “hola”)

//Concatena la palabra hola con la palabra que tal:

strcat(cadena, “que tal”)

//Imprime la cadena la siguiente palabra hola que tal.

printf (“%s”, cadena)

### La función strcmp (cadena1, cadena2)

La función **strcmp (cad1, cad2)** compara lexicográficamente dos cadenas que finalizan con el carácter nulo y devuelve un entero que se interpreta de la siguiente forma:

X = strcmp(cadena1,cadena2);

* + Si (**x < o**) Las cadenas no son iguales la cadena 1 es menor que la cadena2.
  + Si (**x > o**) Las cadenas no son iguales la cadena 1 es mayor que la cadena2.
  + Si (**x == o**) Las cadenas son iguales

**Valor Interpretación**

X = Menor que 0 *cad1* es menor que *cad2*

X = 0 *cad1* es igual a *cad2*

X = Mayor que 0 *cad1* es mayor que *cad2*

**Ejemplo**

printf (“%d”, strcmp (uno, uno)); //Comparación de las cadenas

I = strcmp (uno, uno);

La variable I puede tomar el valor de cero, valor mayor que cero y un valor menor cero.

### La función strlen ( )

Devuelve la longitud de una cadena, en carácter su forma general es:

char cadena [20];

int longitud;

strcpy(cadena, “guillermo”); //Copia el nombre en una cadena.

longitud = strlen(cadena); //Devuelve la longitud de la cadena.

printf(“ %d”,longitud ); //Imprime la longitud de la cadena.

## Matrices:

El lenguaje C soporta, a diferencia de otros lenguajes, arreglos de n dimensiones.

Los arreglos de 3 o más dimensiones no son muy comúnmente usados debido a la cantidad de memoria que se necesita para su almacenamiento. Otro de los motivos por los cuales su uso es muy restringido, obedece al hecho de que se necesita mucho tiempo para el acceso a un elemento especifico, lo que hace mucho más lento el acceso a un arreglo multidimensional que a un arreglo unidimensional, con el mismo número de elementos. Los arreglos multidimensionales más usados son los de dos dimensiones, más conocidos como matrices.

### Declaración de un matriz

Tipo dato nombre\_matriz [cant\_elementos] [cant\_elementos];

**Por ejemplo:**

int M[2][5]; Declaración de matrices enteras

float numeros[20][6]; Declaración de matrices decimales

long double cantidades[2][5]; Declaración de matrices para cantidades grandes

**Representación gráfica de una matriz:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Columnas | | | | |  |
|  |  |  |  |  | **Filas** |
|  |  |  |  |  |

Una matriz de 2 x 5 seria: entero x [2] [5] *// filas y columnas*

Los elementos se numeran y referencia con un índice.

Definición de matrices

Las matrices en lenguaje C pueden ser definidas como los arreglos, pero existen algunas diferencias en cuanto a la sintaxis que se debe utilizar. Por ejemplo, para definir una matriz de 3 filas y 4 columnas se puede escribir la siguiente instrucción:

int m [3] [4];

Donde cada elemento puede contener un número de tipo entero. Gráficamente esta matriz **M** se puede definir así:

m[3][4]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | *Columnas* | | | | |
| 0 | 1 | 2 | 3 |  |
| *Filas* | 0 | 0 , 0 | 0 , 1 | 0 , 2 | 0 , 3 |  |
| 1 | 1 , 0 | 1 , 1 | 1 , 2 | 1 , 3 | Índice |
| 2 | 2 , 0 | 2 , 1 | 2 , 2 | 2 , 3 |  |

**Inicialización de una matriz**

Veamos la siguiente sentencia:

for (i=0; i<3; i++)

{

for (j=0; j<4; j++)

{

m [ i ][ j ] = 10;

}

}

En esta sentencia se está variando un índice **i** desde 0 hasta 3 y un índice **j** desde 0 hasta 4. En cada variación se está asignando al elemento 10 en las posiciones de la matriz **i, j.** El efecto es asignar a cada elemento de una matriz 3 por 4 el número 10.

### Ejercicios de matrices

1. Llenar una matriz 3x5 y determinar cuántos números negativos y positivos contiene la matriz.
2. llenar una matriz 4x4 y determinar el número mayor y el menor que se encuentra en la matriz
3. Buscar un elemento en la matriz.

En este capítulo se presentará el concepto de función o subprograma, una herramienta que ayudará a mantener un trabajo limpio, ordenado y sencillo de entender.

Después de esto se explicará el concepto de estructuras, un tipo de dato abstracto creado por el usuario con el fin de hacer la programación más útil y rápida a la hora de desarrollar un proyecto

# CAPÍTULO IV:

# Funciones y estructuras en c

# Funciones y estructuras en c

## Funciones

Programa

Algoritmo

SubPrograma

SubAlgoritmo

Llamada 1

Llamada 2

Retorno 1

Retorno 2

Una función es un conjunto de líneas de código, que realizan una tarea específica y puede retornar un valor. Las funciones pueden tomar parámetros que modifiquen su funcionamiento. Las funciones son utilizadas para descomponer grandes problemas en tareas simples y para implementar operaciones que son comúnmente utilizadas durante un programa y de esta manera reducir la cantidad de código. Cuando una función es invocada se le pasa el control a la misma. Una vez que esta finalizó con su tarea el control se devuelve al punto desde el cual la función fue llamada.

## Estructuras o registros

Uno de los inconvenientes que tiene una matriz es que por definición todos los datos son del mismo tipo, es decir, no podemos mezclar un real, carácter o un entero.

Una **estructura** es una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificador.

Su forma general es mediante la palabra **registro o estructura:**

**struct** nombre \_ estructura

{

tipo datos;

tipo datos;

.....

};

De momento solo hemos declarado un **tipo de datos**. Para declarar **variables de tipo estructura o registro** se emplea la siguiente notación:

struct nombre\_estructura <nombre\_variable>

Por ejemplo, si deseamos diseñar una **estructura** que guarde los **datos** correspondientes de una **video tienda**, esta estructura, a la que se le llamará **vídeo 200**, deberá guardar: los datos del usuario, datos de la película y un calendario.

**Datos del usuario**

Nombre, dirección, teléfono, cedula.

**Datos de la película**

El nombre de la película, el código de la película, género de la película

**Datos del calendario**

Día, mes, año, hora

Fecha de alquiler. Fecha de recibido.

Cada uno de estos datos se denomina **campo o miembro** de la estructura. El modelo de esta estructura puede crearse del siguiente modo:

struct usuario{

char nombre[31]; // cadena de caracteres

char direccion[21];

long int cedula; // número entero largo

int telefono;

};

El código anterior crea el **tipo de dato usuario** “nombre de la estructura”, pero aún no hay ninguna variable declarada con este nuevo tipo. Obsérvese la necesidad de incluir un carácter (;) después de cerrar las llaves. Recordar que es un separador de sentencias.

Para declarar una **variable de tipo usuario** se debe utilizar la palabra **estructura** y el nombre que le hemos dado (usuario):

**Declaración de la variable de la estructura.**

struct usuario registro;

Para acceder y guardar a los miembros de una estructura se utiliza el operador **punto (.)**, precedido por el **nombre de la variable** y seguido del nombre del campo **variable.campo**.

Por ejemplo, para **dar valor al teléfono del usuario,** el valor 903456, se escribirá:

registro**.**telefono = 903456;

y para guardar la dirección de este mismo usuario, se escribirá:

registro**.**direccion = "C/ Rios Rosas 1,2-A";

De esta forma podemos tanto leer el contenido de un campo del registro, como escribirlo:

x = registro.telefono

De todas formas no es muy efectivo ni muy útil el uso de 1 solo registro. Lo normal es emplear varios, y la forma más inmediata es mediante el empleo de una, matriz, con lo cual habrá que declarar **una matriz de registros,** lo cual no debe preocuparnos pues el nombre de la estructura es considerado como un vector.

**Ejemplo estructura.**

Declaración de los nombres de la estructuras

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <conio.h> //curses para usuarios linux |
| 3 | #include <string.h> |
| 4 | #include <stdlib.h> |
| 5 |  |
| 6 | struct usuario { |
| 7 | char nombre[30]; |
| 8 | char sexo [9]; |
| 9 | int cedula[12]; |
| 10 | char direccion[30]; |
| 11 | }; |
| 12 | struct pelicula { |
| 13 | char nombre\_pelicula[30]; |
| 14 | char genero [9]; |
| 15 | int codigo[12]; |
| 16 | char direccion[30];i |
| 17 | }; |
| 18 |  |
| 19 | struct calendario { |
| 20 | char mes[30]; |
| 21 | char anho [9]; |
| 22 | int dia[12]; |
| 23 | char fecha\_alquiler[30]; |
| 24 | char fecha\_entrega[30]; |
| 25 | }; |
| 26 |  |
| 27 | void main() { |
| 28 | clrscr(); |
| 29 | int i,j,k,p; |
| 30 | float dato; |
| 31 | char temp; |
| 32 |  |
| 33 | struct usuario registro; |
| 34 | struct pelicula video; |
| 35 | struct calendario tiempo; |
| 36 | printf("digite el nombre"); |
| 37 | gets (registro.nombre); |
| 38 | printf("digite el cedula"); |
| 39 | gets(registro.cedula); |
| 40 | printf("digite el fecha"); |
| 41 |  |
| 42 | scanf("%f",&dato); |
| 43 | tiempo.dia=dato; |
| 44 | printf("digite el nombre de la pelicula"); |
| 45 | gets("video.nombre\_pelicula"); |
| 46 |  |
| 47 | printf ("\n Nombre: %s",registro.nombre); |
| 48 | printf ("\n cedula: %s ",registro.cedula); |
| 49 | printf ("\n Nombre \_ pelicula: %s",video.nombre\_pelicula); |
| 50 | getch(); |
| 51 | } |

### Estructuras anidadas

Se producen cuando algún miembro de la estructura es a su vez otra estructura.

**Ejemplo**

|  |  |
| --- | --- |
| 1 | struct fecha { |
| 2 | int dia; |
| 3 | int mes; |
| 4 | int año; |
| 5 | }; |
| 6 | struct persona { |
| 7 | char nombre[20]; |
| 8 | struct fecha nacimiento; |
| 9 | }; |
| 10 | struct persona p; |

**Arrays de estructuras**

Se define primero la estructura y luego se declara una variable de tipo estructura como un arreglo.

**Ejemplo:**

struct registro {

char apellido[10];

char nombre [10];

int cedula[10];

};

struct registro **fami[100]**;

Para acceder a una determinada estructura se indexa el nombre de la estructura:

fami[2].apellido = “pepito”;

fami[1].nombre = “perez”;

fami[4].cedula = 76318649;

**Paso por valor** de miembros de una estructura a una función

El paso por valor se realiza tal como lo realizamos con una variable simple. Por ejemplo, para pasar por valor un miembro en el miembro a.codigo

void f1(int); //declaración el prototipo de función  
f1(a.codigo); //Llamada a la función con un miembro de la función  
void f1(int x);//definición de la función

**Paso por dirección** de un miembro de la estructura a una función.

Se realiza como si fueran variables simples. Por ejemplo, para pasar por valor el miembro a.codigo

void funcion f1(int \*) //declaración el prototipo de función

f1(&a.codigo); //llamada a la función

void f1 (int \*x) // definición de la función  
{...... acciones ......}

Hay que tener en cuenta, que si lo que se pasa a una función es un miembro de una estructura, que sea un arreglo, éste siempre pasa el nombre del vector por dirección (ya que el nombre de un arreglo es la dirección del primer elemento del mismo).

**Paso por valor de estructuras completas a funciones.**

En el siguiente ejemplo, suma es una función que recibe dos estructuras pasadas por valor y a su vez devuelve un valor a una estructura.

struct vector {  
 int x, y, z;  
};

int struct vector (struct vector m1 , struct vector m2 );

void main() {

struct vector v1,v2,v3;  
 ...  
 v3=suma(v1,v2);  
 ...  
}

int struct vector suma(struct vector m1, struct vector m2){  
 m1.x+=m2.x;  
 m1.y+=m2.y;  
 m1.z+=m2.z;  
 return (m1.z);  
}

**Paso por dirección de estructuras completas a funciones.**

Cuando las estructuras son muy grandes es más eficiente pasarlas por dirección. En ese caso, se utilizan punteros a estructuras para realizar la comunicación. Para acceder a los miembros de la estructura, debe utilizarse la combinación de los operadores \* y punto. Sin embargo el operador punto tiene más precedencia que \*, siendo necesario el uso de paréntesis para asegurar que se aplique primero \* y después punto. También puede utilizarse el **operador** -> para acceder a los miembros de una estructura referenciada por un puntero.

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | struct pareja { |
| 3 | int a, b; |
| 4 | }; |
| 5 |  |
| 6 | void f1(struct pareja \*q) { |
| 7 | q->a++; /\* equivalente a (\*q.a)++ pero más usado \*/ |
| 8 | q->b++; |
| 9 | } |
| 10 |  |
| 11 | void main() { |
| 12 | struct pareja p = {3, 17} //inicialización |
| 13 | f1(&p); // Llamada a la estructura por referencia |
| 14 | printf ("valor de a:%d valor de b:%d\n",p.a,p.b); |
| 15 | /\* escribe 14.y 18 \*/ |
| 16 | } |

### PUNTEROS A ESTRUCTURAS.

**Declaración:** struct dir \* pruntero\_dir;

Existen dos usos principales de los punteros a estructuras:

* + Para pasar la dirección de una estructura a una función.
  + Para crear listas enlazadas y otras estructuras de datos dinámicas.

Para encontrar la dirección de una variable de estructura se coloca & antes del nombre de la estructura.

**Ejemplo:**

struct bal {

float balance;

char nombre[80];

} persona;

struct bal \*p;

p = &persona; //coloca la dirección de la estructura persona en el puntero p

No podemos usar el operador punto para acceder a un elemento de la estructura, a través del puntero a la estructura. Debemos utilizar el operador **flecha ->**

p -> balance.

## Definición de nuevos tipos de datos.

La creación de nuevos nombres de tipos de datos se realiza utilizando la palabra reservada **typedef, union**:

La palabra typedef crea un sinónimo del tipo de dato utilizando el siguiente formato:

typedef tipodato nombre

**Ejemplos:**

Con tipos simples:

typedef int ENTERO  
typedef float REAL

ENTERO a, b; // Declara variables a y b de tipo entero  
REAL c;

Con tipos estructurados:

typedef struct{  
 int dia;  
 int mes;  
 int anio;  
} FECHA;

FECHA a;

Se trabaja igual que la estructura, la única diferencia es que se omite la palabra struct y ahora solo se trabaja con FECHA, para declarar una variable de la estructura.

### Enum

Es un tipo que es listado explícitamente por el programador. Es similar a una estructura o a una unión cuyos miembros son constantes de valor entero escritas como identificadores. Las constantes representan los valores que pueden ser asignados a las variables declaradas del tipo del enum.

**enum nombre {val1,val2,...,valn};** donde vali es un identificador de constantes.

Si no se les asigna ningún valor, se inicializan con los valores enteros 0, 1, 2, …

Se pueden asignar otros valores indicándolos en la enumeración

enum nombre {val1=0,val2=10,val3=13,...);

Puede suceder que más de una constante de enumeración tenga el mismo valor entero.

enum color{rojo=-1,azul, amarillo, verde, negro =0};

* Las variables de enumeración pueden utilizarse como enteros; asignándoles valores, compararlas, pero son utilizadas sólo internamente.
* No se puede leer una variable de tipo enum. Se puede leer un entero y asignárselo a una variable.
* No se puede escribir más que el valor entero de la variable enum. No se puede escribir su nombre.
* Los tipos enumerados no aportan capacidades nuevas al lenguaje, pero aumentan la claridad de los programas.

**Ejercicios Propuestos**

* Realizar un programa para una guía telefónica, que permita guardar el número telefónico, el nombre y la dirección de una persona.
* Realizar un programa para un hospital que permita saber los datos de un paciente. Nombre, edad, sexo y seguro social.
* Realizar un programa que permita saber al cliente el valor del producto, cuando digite su código.
* Realizar una función para una agencia de viajes, el cual permita organizar los cupos según el estrato, ubicando 20 personas por estrato.

En este capítulo se explicarán los conceptos básicos relacionados con el manejo de archivos: creación, lectura, escritura, apertura y muchas funciones que ayudarán a desarrollar aplicaciones con un mayor nivel de complejidad y utilidad

# CAPÍTULO V:

# Manejo de archivos

# Manejo de archivos

## Archivos

Los archivos son un conjunto simple de almacenamiento que permiten a la computadora distinguir entre los diversos tipos de información. Aunque no siempre es el caso, un archivo se suele encontrar en un formato legible por los usuarios. Aun así, en un archivo se agrupan instrucciones, números, palabras o imágenes en unidades coherentes que el usuario puede recuperar, modificar, eliminar, guardar o enviar a un dispositivo de salida.

Se puede decir que un archivo es un conjunto completo de información identificado con un nombre. Puede ser un programa, un conjunto de datos utilizados por el programa o un documento creado por los usuarios.

Es importante indicar que los archivos no son únicamente los archivos que guardamos en el disco duro, en C todos los dispositivos del ordenador se tratan como archivos: la impresora, el teclado, la pantalla,...etc.

## Manejo de archivos en lenguaje C

El lenguaje C nos proporciona un acceso secuencial y directo a los registros de un archivo, pero no soporta el acceso indexado a un registro dado.

Los archivos en C los podemos clasificar, según la información que contengan, en dos grupos:

* Archivos de Texto
* Archivos Binarios.

**Los archivos de texto** se caracterizan por estar compuestos por una serie de caracteres organizados en líneas terminadas por un carácter de nueva línea (carácter '\n'). Esto nos hace pensar en la idea de usar la impresora como si fuese un archivo de texto.

Por otro lado, los **archivos binarios** constan de una secuencia de bytes. Podemos decir que cualquier archivo que no sea de texto, será binario.

A la hora de trabajar con archivos, tendremos que especificar antes de usarlos, si serán de texto o binarios.

El lenguaje C trata a los archivos como punteros. En realidad un archivo es un puntero a una estructura de nombre predefinido **FILE**, cuyas componentes son las características de la variable archivo declarada. Cada archivo deberá tener una estructura FILE asociada.

La estructura **FILE** se encuentra definida en el archivo de cabecera **stdio.h**, con lo cual es necesario incluirla en todos los programas que trabajen con archivos.

#include <stdio.h>

La forma de declarar variables de tipo FILE es la siguiente:

FILE \*X, \*Y1,...

Estudiaremos los distintos modos en que podemos abrir un archivo, así como las funciones para leer y escribir en él.

### Apertura: fopen ();

Esta función cierra el archivo apuntado por el puntero y reasigna este puntero a un archivo que será abierto. Su sintaxis es:

fopen(nombre del archivo,"modo de apertura");

Donde **nombre del archivo** es el nombre del nuevo archivo que queremos abrir, luego el **modo de apertura**, y finalmente el puntero que va a ser reasignado.  
  
Antes de abrir un archivo necesitamos declarar un puntero de tipo **FILE**. Para abrir el archivo utilizaremos la función **fopen( )**.  
  
Su sintaxis es:

FILE \*puntero;

puntero = fopen (nombre del archivo, "modo de apertura" );

Donde **puntero** es la variable de tipo **FILE**, **nombre del archivo** es el nombre que daremos al archivo que queremos crear o abrir. Este nombre debe ir encerrado entre comillas. También podemos especificar la ruta donde se encuentra nombre del archivo (En este caso no se pondrán las comillas).

**Ejemplo**

puntero = fopen("DATOS.DAT","r");

puntero = fopen("C:\\TXT\\SALUDO.TXT","w");

Un archivo puede ser abierto en dos modos diferentes, en modo texto o en modo binario.

A continuación lo veremos con más detalle.  
  
***Modo texto***

|  |  |
| --- | --- |
| **wt** | Crea un archivo de escritura. Si ya existe lo crea de nuevo. |
| **w+** | Crea un archivo de lectura y escritura. Si ya existe lo crea de nuevo. |
| **a** | Abre o crea un archivo para añadir datos al final del mismo. |
| **a+** | Abre o crea un archivo para leer y añadir datos al final del |
| **r** | Abre un archivo de lectura. |
| **r+** | Abre un archivo de lectura y escritura. |

***Modo binario***

|  |  |
| --- | --- |
| **Wb** | Crea un archivo de escritura. Si ya existe lo crea de nuevo. |
| **W+b** | Crea un archivo de lectura y escritura. Si ya existe lo crea de nuevo. |
| **Ab** | Abre o crea un archivo para añadir datos al final del mismo. |
| **A+b** | Abre o crea un archivo para leer y añadir datos al final del |
| **Rb** | Abre un archivo de lectura. |
| **R+b** | Abre un archivo de lectura y escritura. |

#### Comprobar si está abierto

Una cosa muy importante después de abrir un archivo es comprobar si realmente está abierto. El sistema no es libre de errores y pueden producirse fallos: el archivo puede no existir, estar dañado o no tener permisos de lectura.

Si intentamos realizar operaciones sobre un puntero tipo FILE cuando no se ha conseguido abrir, el archivo puede tener problemas.

Si el archivo no se ha abierto el puntero archivo (puntero a FILE) tendrá el valor NULL, si se ha abierto con éxito tendrá un valor distinto de NULL. Por lo tanto para comprobar si ha habido errores nos fijamos en el valor del puntero:

FILE \*ARCHIVO;

if (archivo==NULL)

{

printf( "No se puede abrir el archivo.\n" );

exit( 1 );

}

Si archivo==**NULL** significa que no se ha podido abrir por algún error. Lo más conveniente es salir del programa. Para salir utilizamos la función **exit**(1), el 1 indica al sistema operativo que se han producido errores.

#### Cierre

Una vez que hemos acabado nuestro trabajo con un archivo es recomendable cerrarlo. Los archivos se cierran al finalizar el programa pero el número de estos que pueden estar abiertos es limitado. Para cerrar los archivos utilizaremos la función

##### fclose();

Esta función cierra el archivo, cuyo puntero le indicamos como parámetro. Si el archivo se cierra con éxito devuelve 0.

fclose(puntero);

**Un ejemplo:**

FILE pf; /\*Crea un apuntador de tipo archivo

//Crea un archivo AGENDA.dat de modo lectura y binario

pf=fopen("AGENDA.dat","rb");

//Si el apuntador es NULL entonces el archivo está vacío

if (pf != NULL )

{

acciones

}

else

{

fclose(pf); //Cierra el archivo.

}

## Errores típicos manejando archivos.

Las funciones del grupo de fopen son:

fprintf, fscan, fwrite, fread, fclose

Constituyen un interfaz de más alto nivel para el manejo de archivos que las de la familia de open (read, write, close, etc), por lo que deben utilizarse las primeras salvo que haya una buena causa para utilizar las segundas.

### No comprobar si el archivo se ha abierto con éxito.

if ((file = fopen(“c://archivo.txt”,”r”)) == NULL);

fprintf ( “Error de apertura de archivo\n”);

### No vaciar los búferes.

Cuando se escribe un dato en un archivo mediante **fwrite** o **fprintf** los datos no se guardan en el archivo hasta que se llena un buffer interno o se cierra el archivo (cosa que ocurre automáticamente al terminar el programa, sí todo ha ido bien).

Si queremos utilizar los datos guardados en un archivo antes de terminar el programa, debemos llamar a fflush para forzar el vaciamiento de los bufferes.

### No cerrar los archivos.

Otra razón para cerrar un archivo que no estemos utilizando es que el máximo número de archivos que un proceso puede tener abiertos está limitado, y, cuando se intente abrir un archivo más de los permitidos, la operación fallará**.**

## ****Funciones para trabajar con archivos****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| FUNCION | | | | |
| **fopen** | fclose | fgets | fputs | fgetc |
| fseek | ferror | eof, feof | ftell | fputc |

### Lectura del fichero - getc

Ahora ya podemos empezar a leer el fichero. Para ello podemos utilizar la función **getc**, que lee los caracteres uno a uno. Se puede usar también la función fgetc (son equivalentes, la diferencia es que getc está implementada como macro). Además de estas dos existen otras funciones como fgets, fread que leen más de un carácter y que veremos más adelante.

El formato de la función getc (y de fgetc) es:

int getc(FILE \*fichero);

### Comprobar fin de fichero - feof

Cuando entramos en el bucle while, la lectura se realiza hasta que se encuentre el final del fichero. Para detectar el final del fichero se pueden usar dos formas:

* Con la función ***feof()***
* Comprobando si el valor de *letra* es **EOF**.

Esta función comprueba si se ha llegado al final de *fichero,* en cuyo caso devuelve un valor distinto de 0. Si no se ha llegado al final de fichero devuelve un cero. Por eso lo usamos del siguiente modo:

while ( feof(fichero)==0 )

ó

while ( !feof(fichero) )

La segunda forma consiste en comprobar si el carácter leído es el de fin de fichero

**EOF**:

En otras palabras hasta que sea fin de archivo

while ( letra!=EOF )

Cuando trabajamos con ficheros de texto no hay ningún problema, pero si estamos manejando un fichero binario podemos encontrarnos EOF antes del fin de fichero. Por eso es mejor usar feof.

### Cerrar el fichero - fclose

Una vez realizadas todas las operaciones deseadas sobre el fichero, hay que cerrarlo. Es importante no olvidar este paso pues el fichero podría corromperse. Al cerrarlo se vacían los buffer y se guarda el fichero en disco. Un fichero se cierra mediante la función **fclose(fichero).** Si todo va bien fclose devuelve un cero, si hay problemas devuelve otro valor. Estos problemas se pueden producir si el disco está lleno, por ejemplo.

if (fclose(fichero)!=0)

printf( "Problemas al cerrar el fichero\n" );

### Lectura de líneas - fgets

La función **fgets** es muy útil para leer líneas completas desde un fichero. El formato de esta función es:

char

fgets(char \*buffer, int longitud\_max, FILE \*fichero);

Esta función lee desde el fichero hasta que encuentra un carácter '\n' o hasta que lee longitud\_max-1 caracteres y añade '\0' al final de la cadena. La cadena leída la almacena en *buffer*.

Si se encuentra EOF antes de leer ningún carácter o si se produce un error la función devuelve NULL, en caso contrario devuelve la dirección de *buffer*.

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 | #include <conio.h> |
| 4 |  |
| 5 | int main() { |
| 6 | FILE \*fichero; |
| 7 | char texto[100]; |
| 8 | Fichero = fopen("origen.txt","r"); |
| 9 | if (fichero == NULL) { |
| 10 | printf( "No se puede abrir el fichero.\n" ); |
| 11 | exit(1); |
| 12 | } |
| 13 | printf( "Contenido del fichero: \n"); |
| 14 | fgets(texto, 100, fichero); |
| 15 | while (feof(fichero) == 0) { |
| 16 | printf("%s", texto ); |
| 17 | fgets(texto, 100, fichero); |
| 18 | } |
| 19 | if (fclose(fichero) != 0) |
| 20 | printf("Problemas al cerrar el fichero.\n"); |
| 21 | } |

Ahora se analizará un ejemplo que lee un fichero de texto y lo muestra en la pantalla y donde se utilizan todos los elementos que hasta el momento se han tratado:

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 | #include <conio.h> |
| 4 |  |
| 5 | int main() { |
| 6 | FILE \*fichero; |
| 7 | char letra; |
| 8 | fichero = fopen("origen.txt","r"); |
| 9 | if (fichero == NULL) { |
| 10 | printf("No se puede abrir el fichero.\n"); |
| 11 | exit(1); |
| 12 | } |
| 13 | printf("Contenido del fichero: \n"); |
| 14 | letra = getc(fichero); |
| 15 | while (feof(fichero) == 0) { |
| 16 | printf("%c", letra ); |
| 17 | letra=getc(fichero); |
| 18 | } |
| 19 | if (fclose(fichero)!= 0) |
| 20 | printf( "Problemas al cerrar el fichero\n" ); |
| 21 | } |

## Escritura de ficheros

La escritura en ficheros se explicará con un ejemplo en donde se abrirá un fichero '*origen.txt*' y se copiará en otro fichero '*destino.txt*'. Además el fichero se muestra en pantalla. Las partes nuevas están marcadas en negrita para que se vea la diferencia entre este y el último ejemplo ya visto:

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 | #include <conio.h> |
| 4 |  |
| 5 | int main() { |
| 6 | FILE \*origen, \*destino; |
| 7 | char letra; |
| 8 | origen = fopen("origen.txt", "r"); |
| 9 | destino = fopen("destino.txt", "w"); |
| 10 | if (origen==NULL || destino==NULL) { |
| 11 | printf( "Problemas con los ficheros.\n" ); |
| 12 | exit( 1 ); |
| 13 | } |
| 14 | letra=getc(origen); |
| 15 | while (feof(origen)==0) { |
| 16 | putc(letra, destino); |
| 17 | printf( "%c", letra ); |
| 18 | letra=getc(origen); |
| 19 | } |
| 20 | if (fclose(origen)!=0) |
| 21 | printf( "Problemas al cerrar el fichero origen.txt\n" ); |
| 22 | if (fclose(destino)!=0) |
| 23 | printf("Problemas al cerrar el fichero destino.txt\n" ); |
| 24 | } |

Como hemos visto el puntero FILE es la base de la escritura / lectura de archivos. Por eso se definen dos punteros FILE:

* el puntero 'origen' donde se almacenará la información sobre el fichero origen.txt y
* 'destino' donde se guardará el fichero destino.txt (el nombre del puntero no tiene por qué coincidir con el de fichero).

El siguiente paso, como antes, es abrir el fichero usando fopen. La diferencia es que ahora se deberá abrir para escritura. Se usará el modo 'w' (crea el fichero o lo vacía si existe) porque se quiere crear un fichero.

Recorar que después de abrir un fichero hay que comprobar si la operación se ha realizado con éxito. En este caso, como es un sencillo ejemplo, se han comprobado ambos a la vez:

if (origen ==NULL || destino ==NULL)

Pero es más correcto hacerlo por separado así se sabrá dónde se está produciendo el posible fallo.

#### Lectura del origen y escritura en destino- getc y putc

Como se puede observar en el ejemplo la lectura del fichero se hace igual como se hizo anteriormente. Para la escritura se usará la función **putc**:

int putc(int c, FILE \*fichero);

Donde c contiene el carácter que se quiere escribir en el fichero y el puntero *fichero* es el fichero sobre el que se trabaja.

De esta forma se irá escribiendo en el fichero *destino.txt* el contenido del fichero *origen.txt*.

#### Comprobar fin de fichero

Como siempre que se leen datos de un fichero se debe comprobar si se ha llegado al final. Sólo se debe comprobar si se está al final del fichero. No se tiene que comprobar el final del fichero en el que se escribe, puesto que se está creando y aún no tiene final.

### Escritura de líneas - fputs

La función **fputs** trabaja junto con la función fgets:

int fputs(const char \*cadena, FILE \*fichero);

#### Ejercicio

Este programa lee un fichero y le suprime todas las vocales.

Es decir que siendo el fichero origen.txt:

“El alegre campesino

pasea por el campo

ajeno a que el toro

se acerca por detrás”

El fichero destino.txt sea:

l lgr cmpsn

ps pr l cmp

jn q l tr

s crc pr dtrás

#### Solución

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 | #include <conio.h> |
| 4 |  |
| 5 | int main() { |
| 6 | FILE \*origen, \*destino; |
| 7 | char letra; |
| 8 | origen=fopen("origen.txt","r"); |
| 9 | destino=fopen("destino.txt","w"); |
| 10 | if (origen==NULL || destino==NULL) { |
| 11 | printf( "Problemas con los ficheros.\n" ); |
| 12 | exit( 1 ); |
| 13 | } |
| 14 | letra=getc(origen); |
| 15 | while (feof(origen)==0) { |
| 16 | if (!strchr("AEIOUaeiou",letra)) putc( letra, destino ); |
| 17 | letra=getc(origen); |
| 18 | } |
| 19 | if (fclose(origen)!=0) |
| 20 | printf( "Problemas al cerrar el fichero origen.txt\n" ); |
| 21 | if (fclose(destino)!=0) |
| 22 | printf( "Problemas al cerrar el fichero destino.txt\n" ); |
| 23 | } |

## Otras funciones para el manejo de ficheros

### fread y fwrite

Las funciones vistas hasta ahora (getc, putc, fgets, fputs), son adecuadas para trabajar con caracteres (1 byte) y cadenas. Pero, ¿qué sucede cuando se quier trabajar con otros tipos de datos?

Supón que se quieren almacenar variables de tipo int en un fichero. Como las funciones vistas hasta ahora sólo pueden operar con cadenas se deben convertir los valores a cadenas (con la función *itoa*). Para recuperar luego estos valores, se deben leer como cadenas y pasarlos a enteros (*atoi*).

Existe una solución mucho más fácil. Las funciones **fread** y **fwrite**. Estas funciones permiten tratar con datos de cualquier tipo, incluso con estructuras.

#### fwrite

Permite escribir en un fichero. Esta función tiene el siguiente formato:

fwrite(void \*buffer, size\_t size, size\_t num, FILE \*pfile);

* buffer - variable que contiene los datos que se van a escribir en el fichero.
* size - el tamaño del tipo de dato a escribir. Puede ser un int, un float, una estructura, etc. Para conocer su tamaño se usa el operador *sizeof*.
* num - el número de datos a escribir.
* pfile - El puntero al fichero sobre el que se trabaja.

**Ejemplo**: un programa de agenda que guarda el nombre, apellido y teléfono de cada persona.

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 | #include <conio.h> |
| 4 |  |
| 5 | struct { |
| 6 | char nombre[20]; |
| 7 | char apellido[20]; |
| 8 | char telefono[15]; |
| 9 | } registro; |
| 10 | int main() { |
| 11 | FILE \*fichero; |
| 12 | fichero = fopen( "nombres.txt", "a" ); |
| 13 | do { |
| 14 | printf( "Nombre: " ); fflush(stdin); |
| 15 | gets(registro.nombre); |
| 16 | if (strcmp(registro.nombre,"")) { |
| 17 | printf( "Apellido: " ); fflush(stdin); |
| 18 | gets(registro.apellido); |
| 19 | printf( "Teléfono: " ); fflush(stdin); |
| 20 | gets(registro.telefono); |
| 21 | fwrite( &registro, sizeof(registro), 1, fichero ); |
| 22 | } |
| 23 | } |
| 24 | while (strcmp(registro.nombre," ")!=0); |
| 25 | fclose( fichero ); |
| 26 | } |

NOTA: El bucle termina cuando el 'nombre' se deja en blanco.

Este programa guarda los datos personales mediante fwrite, usando la estructura *registro*. Abrimos el fichero en **modo '*a*'** (append, añadir), para que los datos introducidos se añadan al final del fichero.

Una vez abierto abrimos se entra en un bucle *do-while* mediante el cual se introducen los datos. Los datos se van almacenando en la variable *registro* (que es una estructura). Una vez se tienen todos los datos de la persona, se meten en el fichero con fwrite:

fwrite( Ristro, sizeof(registro), 1, fichero );

* Ristro - es la variable (en este caso una estructura) que contiene la información a meter al fichero.
* sizeof(registro) - lo utilizamos para saber cuál es el número de bytes que vamos a guardar, el tamaño en bytes que ocupa la estructura.
* 1 - indica que sólo vamos a guardar un elemento. Cada vez que se recorre el bucle guardamos sólo un elemento.
* fichero - el puntero FILE al fichero donde vamos a escribir.

#### fread

La función *fread* se utiliza para sacar información de un fichero. Su formato es:

fread(void \*buffer, size\_t tamano, size\_t numero, FILE \*pfichero);

Siendo *buffer* la variable donde se van a escribir los datos leídos del fichero *pfichero*.

El valor que devuelve la función indica el número de elementos de tamaño 'tamano', que ha conseguido leer. Se le pude pedir a fread que lea 10 elementos (numero=10), pero si en el fichero sólo hay 6 elementos, fread devolverá el número 6.

Siguiendo con el ejemplo anterior, ahora se leeran los datos que se habían introducido en "nombres.txt".

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 | #include <conio.h> |
| 4 |  |
| 5 | struct { |
| 6 | char nombre[20]; |
| 7 | char apellido[20]; |
| 8 | char telefono[15]; |
| 9 | } registro; |
| 10 |  |
| 11 | void main() { |
| 12 | FILE \*fichero; |
| 13 | fichero = fopen( "nombres.txt", "r" ); |
| 14 | while (!feof(fichero)) { |
| 15 | if (fread( Registro, sizeof(registro), 1, fichero )) { |
| 16 | printf("Nombre: %s\n", registro.nombre); |
| 17 | printf("Apellido: %s\n", registro.apellido); |
| 18 | printf("Teléfono: %s\n", registro.telefono); |
| 19 | } |
| 20 | } |
| 21 | fclose( fichero ); |
| 22 | } |

Se abre el fichero *nombres.txt* en modo lectura. Con el bucle while se asegura que se recorre el fichero hasta el final (y que no nos pasamos).

La función fread lee un registro (numero=1) del tamaño de la estructura *registro*. Si realmente ha conseguido leer un registro la función devolverá un 1, en cuyo caso la condición del 'if' será verdadera y se imprimirá el registro en la pantalla. En caso de que no queden más registros en el fichero, fread devolverá 0 y no se mostrará nada en la pantalla.

#### Ejemplo

Escribir cinco registros en un archivo y leerlo posteriormente.

**Solución**

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 |  |
| 3 | struct t\_reg { |
| 4 | int num; |
| 5 | char cad[10]; |
| 6 | char car; |
| 7 | }; |
| 8 |  |
| 9 | int crear\_archivo () { |
| 10 | FILE \*fich; |
| 11 | int i, er\_cod = 0; |
| 12 | struct t\_reg r; |
| 13 | if ((fich = fopen(“fichreg.dat", “wb")) == NULL) { |
| 14 | printf ("Error al abrir el archivo\n"); |
| 15 | er\_cod = 1; |
| 16 | } |
| 17 | else { |
| 18 | for (i = 0; i < 5; i + + ) { |
| 19 | r.num = i; |
| 20 | r.car =’a’+1; |
| 21 | printf("Dé un nombre: "); |
| 22 | gets(r.cad); |
| 23 | fwrite(&r, sizeof(r), 1, fich); |
| 24 | } |
| 25 | fclose (fich); |
| 26 | } |
| 27 | return er\_cod; |
| 28 | } |
| 29 |  |
| 30 | int Ieer\_archivo () { |
| 31 | FILE \*fich; |
| 32 | struct t-reg r; |
| 33 | int er\_dev = 0; |
| 34 | if ((fich = fopen(“fichreg.dat", “rb")) == NULL) { |
| 35 | printf ( “Error abriendo el archivo para lectura\n“ ); |
| 36 | er\_dev = 1. |
| 37 | } |
| 38 | else { |
| 39 | fread (&r, sizeof(r), 1, fich); |
| 40 | while (! feof(fich)) { |
| 41 | printf ("%d: %s: %c\n" , r.num, r.cad, r.car); |
| 42 | fread (&r, sizeof(r), 1, fich); |
| 43 | } |
| 44 | fclose (fich); |
| 45 | } |
| 46 | return er\_dev; |
| 47 | } |
| 48 | int main(void) { |
| 49 | int error; |
| 50 | error = crear\_archivo(); |
| 51 | if (!error) Ieer\_archivo(); |
| 52 | } |

**EJEMPLOS: Archivos**

|  |  |
| --- | --- |
| 1 | #include <conio.h> |
| 2 | #include <stdio.h> |
| 3 | #include <string.h> |
| 4 | #include <io.h> |
| 5 | #include <stdlib.h> |
| 6 | #define NPacientes 4 |
| 7 | void Imprimir(struct Paciente \*P1); |
| 8 | void Elimina(struct Paciente \*P1, int); |
| 9 | void Modifica(struct Paciente \*P1); |
| 10 | int Buscar(struct Paciente \*P1, int); |
| 11 | void Ingresar(struct Paciente \*P1); |
| 12 | void Estadisticas(struct Paciente \*P1); |
| 13 | void Guardar(struct Paciente \*P1); |
| 14 |  |
| 15 | struct Paciente |
| 16 | { |
| 17 | int REG; |
| 18 | int CEDULA; |
| 19 | char Nombre[12]; |
| 20 | char Sexo[9]; |
| 21 | unsigned long TELEFONO; |
| 22 | char Seguro[9]; |
| 23 | unsigned int CALLE; |
| 24 | unsigned int NUMERO; |
| 25 | char Ciudad[20]; |
| 26 | }; |
| 27 | Persona[NPacientes],\*P1=Persona; |
| 28 | void main() |
| 29 | { |
| 30 | int SALIR=0, TECLA; |
| 31 | int SEARCH; |
| 32 | clrscr(); |
| 33 | while(SALIR!=1) |
| 34 | { |
| 35 | printf("Clinica del Desamparado\n"); |
| 36 | printf("1. Buscar En la Base De Datos:\n"), |
| 37 | printf("2. Estadisticas De La Empresa\n"); |
| 38 | printf("3. Ingresar Nuevo Paciente:n"); |
| 39 | printf("4. Eliminar Paciente Paciente\n"); |
| 40 | printf("5. Guardar Info En El DD:\n"); |
| 41 | printf("99. Salir!: \n "); |
| 42 | printf("Escoja una opcion: "); |
| 43 | TECLA=getche(); |
| 44 | switch(TECLA) |
| 45 | { |
| 46 | case 49: |
| 47 | int R=0; |
| 48 | clrscr(); |
| 49 | printf("Ingrese la cedula de la persona:\n\n"); |
| 50 | scanf("%d",&SEARCH); |
| 51 | R=Buscar(P1,SEARCH); |
| 52 | if(R==-1) |
| 53 | { |
| 54 | clrscr(); |
| 55 | printf("\No se a encontrado la persona! :P\n"); |
| 56 | printf("Se ingresara esta persona/n\n");getch(); |
| 57 | Ingresar(P1); |
| 58 | } |
| 59 | Else |
| 60 | { |
| 61 | Imprimir(P1+R); |
| 62 | printf("PersonaEncontradaX:],Desea Modificarla?"); |
| 63 | TECLA=getche(); |
| 64 | if(TECLA=='s' || TECLA=='S') |
| 65 | Modifica(P1+R); |
| 66 | } |
| 67 | break; |
| 68 | case 50: |
| 69 | Estadisticas(P1); |
| 70 | break; |
| 71 | case 51: |
| 72 | clrscr(); |
| 73 | Ingresar(P1); |
| 74 | break; |
| 75 | case 52: |
| 76 | clrscr(); |
| 77 | printf("Ingrese la cedula del paciente a borrar: "); |
| 78 | scanf("%d",&SEARCH); |
| 79 | Elimina(P1,&SEARCH); |
| 80 | break; |
| 81 | case 53: |
| 82 | clrscr(); |
| 83 | Guardar(P1); |
| 84 | break; |
| 85 | case 57: |
| 86 | TECLA=getche(); |
| 87 | if (TECLA==57) |
| 88 | SALIR=1; |
| 89 | break; |
| 90 | } } |
| 91 | } |
| 92 |  |
| 93 | Buscar(struct Paciente \*P1,int NUM) { |
| 94 | int i; |
| 95 | for(i=0;i<NPacientes;i++) { |
| 96 | if ((P1+i)->CEDULA == NUM) |
| 97 | return(i);break; |
| 98 | } |
| 99 | return(-1); |
| 100 | } |
| 101 |  |
| 102 | void Imprimir(struct Paciente \*P1) { |
| 103 | clrscr(); |
| 104 | printf("\n\nCedula: "); |
| 105 | printf("%d",P1->CEDULA); |
| 106 | printf("Expedida en: "); |
| 107 | scanf("%s",P1->Ciudad); |
| 108 | printf("\n\nNombre"); |
| 109 | scanf("%s",P1->Nombre); |
| 110 | printf("\n\nSexo: "); |
| 111 | scanf("%s",P1->Sexo); |
| 112 | printf("\n\nTelefono: "); |
| 113 | printf("%l",P1->TELEFONO); |
| 114 | printf("\n\nDireccion:"); |
| 115 | printf("\nCalle:"); |
| 116 | printf("%d",P1->CALLE); |
| 117 | printf("\nNumero:"); |
| 118 | printf("%d",P1->NUMERO); |
| 119 | printf("\n\nSeguro:"); |
| 120 | scanf("%s",P1->Seguro); |
| 121 | } |
| 122 |  |
| 123 | void Elimina(struct Paciente \*P1,int NUM) { |
| 124 | int R; |
| 125 | R=Buscar(P1,NUM); |
| 126 | if (R==-1) |
| 127 | printf("Registro No Existente O Ya Borrado"); |
| 128 | else { |
| 129 | (P1+R)->CEDULA=NULL; |
| 130 | strcpy((P1+R)->Nombre, NULL); |
| 131 | strcpy((P1+R)->Sexo, NULL); |
| 132 | strcpy((P1+R)->Ciudad, NULL); |
| 133 | strcpy((P1+R)->Seguro, NULL); |
| 134 | (P1+R)->TELEFONO=NULL; |
| 135 | (P1+R)->CALLE=NULL; |
| 136 | (P1+R)->NUMERO=NULL; |
| 137 | } |
| 138 | } |
| 139 | void Modifica(struct Paciente \*P1) { |
| 140 | clrscr(); |
| 141 | printf("Ingrese La Cedula: "); |
| 142 | scanf("%d",&P1->CEDULA); |
| 143 | printf("Ingrese El Nombre: "); |
| 144 | scanf("%s",P1->Nombre); |
| 145 | printf("Ingrese El Sexo: "); |
| 146 | gets(P1->Sexo); |
| 147 | printf("Direccion:\n\n"); |
| 148 | printf("Ingrese Calle: "); |
| 149 | scanf("%d",&P1->CALLE); |
| 150 | printf("Ingrese Numero: "); |
| 151 | scanf("%d",&P1->NUMERO); |
| 152 | printf("Tiene Seguro?\n"); |
| 153 | gets(P1->Seguro); |
| 154 | } |
| 155 | void Ingresar(struct Paciente \*P1) { |
| 156 | int i; |
| 157 | for(i=0;i<NPacientes;i++) { |
| 158 | if((P1+i)->REG==0) { |
| 159 | (P1+i)->REG=1; |
| 160 | clrscr(); |
| 161 | printf("Ingrese La Cedula: "); |
| 162 | scanf("%d",&(P1+i)->CEDULA); |
| 163 | printf("Ingrese El Nombre: "); |
| 164 | scanf("%s",(P1+i)->Nombre); |
| 165 | printf("Ingrese El Sexo: "); |
| 166 | scanf("%s",(P1+i)->Sexo); |
| 167 | printf("Direccion:\n\n"); |
| 168 | printf("Ingrese Calle: "); |
| 169 | scanf("%d",&(P1+i)->CALLE); |
| 170 | printf("Ingrese Numero: "); |
| 171 | scanf("%d",&(P1+i)->NUMERO); |
| 172 | printf("Ciudad: "); |
| 173 | scanf("%s",(P1+i)->Ciudad); |
| 174 | printf("Tiene Seguro?\n"); |
| 175 | scanf("%s",(P1+i)->Seguro); |
| 176 | break; |
| 177 | } |
| 178 | } |
| 179 | } |
| 180 | void Estadisticas(struct Paciente \*P1) { } |
| 181 | void Guardar(struct Paciente \*P1) { |
| 182 | FILE \*ARCHIVO; |
| 183 | if((ARCHIVO=fopen("Data.vWv","wb"))==NULL) |
| 184 | exit(0); |
| 185 | else |
| 186 | fwrite(P1,sizeof(Paciente),1,ARCHIVO); |
| 187 | fclose(ARCHIVO); |
| 188 | } |

**EJEMPLO 2:**

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 |  |
| 3 | int main(int argc, char \*\*argv) |
| 4 | { |
| 5 | FILE \*fe, \*fs; |
| 6 | unsigned char buffer[2048]; // Buffer de 2 Kbytes |
| 7 | int bytesLeidos; |
| 8 | if(argc != 3) { |
| 9 | printf("Usar:copia<archivo\_origen><archivo\_destino>\n"); |
| 10 | return 1; |
| 11 | } |
| 12 |  |
| 13 | // Abrir el archivo de entrada en lectura y binario |
| 14 | fe = fopen(argv[1], "rb"); |
| 15 | if(!fe) { |
| 16 | printf("%s no existe o no puede ser abierto.\n", |
| 17 | argv[1]); |
| 18 | return 1; |
| 19 | } |
| 20 | // Crear o sobreescribir el archivo de salida en binario |
| 21 | fs = fopen(argv[2], "wb"); |
| 22 | if(!fs) { |
| 23 | printf("El archivo %s no puede ser creado.\n", argv[2]); |
| 24 | fclose(fe); |
| 25 | return 1; |
| 26 | } |
| 27 | // Bucle de copia: |
| 28 | while((bytesLeidos = fread(buffer, 1, 2048, fe))) |
| 29 | fwrite(buffer, 1, bytesLeidos, fs); |
| 30 | // Cerrar archivos: |
| 31 | fclose(fe); |
| 32 | fclose(fs); |
| 33 | return 0; |
| 34 | } |

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | int main() |
| 3 | { |
| 4 | char nombre[10]="datos.dat", linea[81]; |
| 5 | FILE \*archivo; |
| 6 | archivo = fopen( nombre, "r" ); |
| 7 | printf( "Archivo: %s -> ", nombre ) |
| 8 | if( archivo ) |
| 9 | printf( "existe (ABIERTO)\n" ); |
| 10 | else { |
| 11 | printf( "Error (NO ABIERTO)\n" ); |
| 12 | return 1; |
| 13 | } |
| 14 | printf( "La primera linea del archivo: %s\n\n", nombre ); |
| 15 | printf( "%s\n", fgets(linea, 81, archivo) ); |
| 16 | if( !fclose(archivo) ) |
| 17 | printf( "\nArchivo cerrado\n" ); |
| 18 | else { |
| 19 | printf( "\nError: archivo NO CERRADO\n" ); |
| 20 | return 1; |
| 21 | } |
| 22 | return 0; |
| 23 | } |

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | int main() |
| 3 | { |
| 4 | char nombre[11]="datos2.dat"; |
| 5 | FILE \*archivo; |
| 6 | archivo = fopen( nombre, "w" ); |
| 7 | printf( "Archivo: %s -> ", nombre ); |
| 8 | if( archivo ) |
| 9 | printf( "creado (ABIERTO)\n" ); |
| 10 | else { |
| 11 | printf( "Error (NO ABIERTO)\n" ); |
| 12 | return 1; |
| 13 | } |
| 14 | fputs( "Esto es un ejemplo usando \'fputs\'\n", archivo ); |
| 15 | if( !fclose(archivo) ) |
| 16 | printf( "\nArchivo cerrado\n" ); |
| 17 | else { |
| 18 | printf( "\nError: archivo NO CERRADO\n" ); |
| 19 | return 1; |
| 20 | } |
| 21 | return 0; |
| 22 | } |

### fseek

La función fseek permite situar al programador en la posición que se quiera de un fichero abierto. Cuando se lee un fichero, hay un 'puntero' que indica en qué lugar del fichero se encuentra el usuario. Cada vez que se leen datos del fichero, este puntero se desplaza. Con la función fseek se puede situar este puntero en el lugar que se desee.

El formato de fseek es el siguiente:

fseek(FILE \*pfichero, long desplazamiento, int modo);

fseek(nombre de apuntador, posición , origen);

Como siempre *pfichero* es un puntero de tipo FILE que apunta al fichero con el que se quiere trabajar, *desplazamiento* son las posiciones (o bytes) que se quiere desplazar el puntero. Este desplazamiento puede ser de tres tipos dependiendo del valor de *modo*:

|  |  |
| --- | --- |
| SEEK\_SET | El puntero se desplaza desde el principio del fichero. |
| SEEK\_CUR | El puntero se desplaza desde la posición actual del fichero. |
| SEEK\_END | El puntero se desplaza desde el final del fichero. |

Estas tres constantes están definidas en el fichero <stdio.h>. Como curiosidad se indican a continuación sus definiciones:

#define SEEK\_SET 0

#define SEEK\_CUR 1

#define SEEK\_END 2

Nota: es posible que los valores cambien de un compilador a otro.

Si se produce algún error al intentar posicionar el puntero, la función devuelve un valor distinto de 0. Si todo ha ido bien el valor devuelto es un 0.

En el siguiente ejemplo se muestra el funcionamiento de fseek. Se trata de un programa que lee la letra que hay en la posición que especifica el usuario.

|  |  |
| --- | --- |
| 1 | #include <stdio.h> |
| 2 | #include <stdlib.h> |
| 3 | #include <conio.h> |
| 4 | void main() |
| 5 | { |
| 6 | FILE \*fichero; |
| 7 | long posicion; |
| 8 | int resultado; |
| 9 | fichero = fopen( "origen.txt", "r"); |
| 10 | printf( "¿Qué posición quieres leer? "); |
| 11 | fflush(stdout); |
| 12 | scanf( "%d", &posicion ); |
| 13 | resultado = fseek( fichero, posicion, SEEK\_SET ); |
| 14 | if (!resultado) |
| 15 | printf("%c esta en la posicion %d\n", fgetc(fichero), posición); |
| 16 | else |
| 17 | printf( "Problemas posicionando el cursor.\n" ); |
| 18 | fclose( fichero ); |
| 19 | } |

### ftell

Esta función es complementaria a fseek, devuelve la posición actual dentro del fichero.

Su formato es el siguiente:

long ftell(FILE \*pfichero);

El valor que da ftell puede ser usado por fseek para volver a la posición actual.

### fprintf y fscanf

Estas dos funciones trabajan igual que sus equivalentes printf y scanf. La única diferencia es que se puede especificar el fichero sobre el que operar (si se desea puede ser la pantalla para fprintf o el teclado para fscanf).

Los formatos de estas dos funciones son:

int fprintf(FILE \*pfichero, const char \*formato, ...);

int fscanf (FILE \*pfichero, const char \*formato, ...);